Ferrous Lineup por dentro

Ferrous Lineup contributors

Contenido

Introducción 5

Compilarlo desde el código fuente 8

Parte I: La forma del programa

Cinco crates, y qué puede saber cada uno 12

Dos hilos: la ventana y el audio 22

El proyecto: patrones, canales, pistas, clips y marcas 36

Decirle al hilo de audio lo que ha cambiado 51

Deshacer y rehacer 70

Guardar: el archivo .lineup, la recuperación y los archivos dañados 80

Dibujar las ventanas con egui: distribución, zoom, temas, iconos 101

Los editores: rack de canales, piano roll y lista de reproducción 131

Dos idiomas: traducciones, y letras o solfeo 171

Ejercicios: la forma del programa 182

Parte II: El sonido

El secuenciador: temporización a la muestra 196

Tempo, compás, swing y sensación 210

Instrumentos: batería, sampler y Lineup Osc 237

SoundFonts y General MIDI 256

El mezclador: niveles, balance, reverberación, efectos y orden de partitura 276

MIDI: mensajes, dispositivos, grabación y archivos 298

Exportar: WAV, MIDI y video 350

La traza de reproducción 383

Ejercicios: el sonido 398

Parte III: La teoría musical en código

Clases de altura, notas y su escritura 414

Escalas, modos y tonalidades 428

Acordes: construirlos, nombrarlos e invertirlos 446

Números romanos, las teclas de armonía y las funciones de los acordes 467
Herramientas de edición como funciones simples 487
Oír la armonía: acordes y tonalidades a partir de las notas 501
Conjuntos de clases de altura, y una armonía con cuatro nombres 522
Series dodecafónicas y sus reglas 537
Micromodos 565
Conducción de voces y movimiento de las partes 587
Importar una grabación (experimental) 600
Ejercicios: la teoría musical en código 623

Parte IV: La banda

La biblioteca: estilos y ritmos como datos 639
La hoja de acordes: modelo, distribución y texto 661
Cómo toca la banda una hoja de acordes 697
Una orquesta para música que cambia de compás 732
Forma, secciones, variaciones y bloqueos 747
Golpes, sostenidos, silencios y anticipaciones 769
El solista 779
Tus propios estilos: el editor de estilos 793
Herramientas de práctica: metrónomo, entrenador de velocidad, entrenamiento auditivo 804
La partitura guía: de las notas a la notación 834
Ejercicios: la banda 856

Parte V: Las canciones como datos

Cómo se escribe una demostración 874
De una interpretación MIDI o una partitura a una demostración 891
Ejercicios: las canciones como datos 901

Parte VI: Compilar, probar y distribuir

Cómo se prueba el código: tests unitarios, tests de ventana, trazas 908
Tres sistemas e integración continua 914
Paquetes de cada versión 922
Cómo se construyen los libros 935

Referencia

Formato del archivo de proyecto 942
Glosario de Rust 948
Glosario de música 966
Ideas para lo que viene 973
Índice alfabético 974

Introducción

Ferrous Lineup es un programa de código abierto, escrito en Rust, para tocar canciones con una **banda de acompañamiento**. Escribe los acordes de una canción compás por compás, elige un estilo, y una banda (bajo, batería y piano, con guitarra, cuerdas, vientos u órgano cuando el estilo los tiene) la toca en cualquier tonalidad y tempo. Cada parte que toca la banda cae en una pista propia, donde puedes abrirla en el piano roll, cambiarla a mano, o dejarla afuera y tocarla tú.

También es una manera de **aprender armonía** mientras tocas. Nombra cada acorde que oyes, con su número romano en la tonalidad y lo que hace ahí, y conoce también las otras maneras en que la música nombra la armonía: conjuntos de clases de altura, series dodecafónicas y los micromodos de Kröpfl.

Este libro trata del interior: **cómo está organizado el código, cómo está diseñado y por qué**, y cómo está escrita en Rust la teoría musical que conoce la aplicación. Explica el Rust que usa cada parte a medida que avanza, señalando los archivos reales, así que también es un curso de Rust construido alrededor de un programa de verdad. No hace falta saber Rust antes de leerlo; si conoces otro lenguaje de programación, puedes seguirlo. Para *usar* la aplicación, lee en cambio el manual de usuario: no tiene nada de código.

Cómo está organizado el libro

Los capítulos siguen el código, de la forma del programa a cómo se distribuye. Cada uno dice qué hace una parte de la aplicación, con un enlace a la página del manual sobre cómo usarla; cómo está diseñada y por qué; el Rust que usa; los archivos donde vive; y los tests que la protegen. Cada parte termina con ejercicios.

Parte	De qué trata	El Rust del camino
I. La forma del programa	los cinco crates, los dos hilos, los datos del proyecto, deshacer, guardar, las ventanas, dos idiomas	workspaces y crates, módulos, enums con datos, reglas guardadas por un solo portero, `Arc` y un ring buffer sin bloqueos, `History<T>`, serde, enums de errores, `Drop`, dibujo en modo inmediato, opciones por hilo

Parte	De qué trata	El Rust del camino
II. El sonido	el secuenciador, el tempo y el compás, los instrumentos, los SoundFonts, el mezclador, MIDI, exportar, la traza de reproducción	temporización a la muestra, atómicos, closures `FnMut`, un `Mutex` y colas `mpsc`, cadenas de bytes y patrones de slices, el único bloque `unsafe`, tipos `Copy` con reglas
III. La teoría musical en código	la altura, escalas y tonalidades, acordes, números romanos y funciones, herramientas de edición, oír la armonía, conjuntos, series, micromodos, conducción de voces, importar una grabación	tablas en un `match`, conjuntos de bits, iteradores (`find_map`, `cycle`, `chunk_by`), recoger `Option`s, `FromStr` y `Display`, `try_from`, programación dinámica
IV. La banda	la biblioteca, la hoja de acordes, cómo toca la banda, la orquesta, la forma, las paradas, el solista, tus propios estilos, las herramientas de práctica, la partitura guía	tablas `static` y `const fn`, `macro_rules!`, hashes y generadores con semilla, leer texto, dibujar un PDF
V. Las canciones como datos	cómo se escribe una demostración, y cómo una interpretación MIDI o una partitura se convierte en una	datos `const` en módulos hijos, sintaxis de actualización de structs, `LazyLock`, ordenar con `Ordering`
VI. Compilar, probar y distribuir	los tests, los tres sistemas y su CI, los paquetes de cada versión, cómo se construyen estos libros	`#[cfg(test)]`, tests de integración, `catch_unwind`, `cfg` y features, scripts de compilación

Al final están el formato del archivo de proyecto, un glosario de Rust, un glosario de música e ideas para lo que viene.

Por dónde empezar

- **Para aprender Rust:** lee las partes en orden, con el glosario de Rust abierto, y prueba los ejercicios al final de cada parte. Compilarlo desde el código fuente

pone primero el código en marcha.
- **Para aprender cómo se programa la teoría musical:** empieza por la Parte III, y ten cerca el glosario de música. La teoría en sí, sin código, se enseña en el manual.
- **Para cambiar algo:** busca la parte de la aplicación en el índice, lee su capítulo, y ejecuta sus tests antes y después de tu cambio.

Cómo leerlo

1. En línea, en https://naelolaiz.github.io/ferrous-lineup/es/.
2. Como un solo PDF, para leer sin conexión o imprimir: ferrous-lineup-book-es.pdf. Tiene números de página, el índice de contenido al principio, un índice alfabético al final, y marcadores para cada capítulo.
3. Como archivos Markdown, en la carpeta `book-es/src` del repositorio, o como sitio web en tu computadora con `mdbook serve --open` en `book-es/`.

El libro también está en inglés, en https://naelolaiz.github.io/ferrous-lineup/. Cómo se construyen los libros cuenta cómo se hacen los sitios web y los PDF.

Los enlaces al código apuntan a la rama `main`, y los fragmentos de los capítulos están copiados de ella. Donde un fragmento deja líneas afuera, lo dice con `// ...`.

Nombre y licencia

«Ferrous» quiere decir «de hierro» en inglés, y el óxido (*rust*) es óxido de hierro; un *lineup* es la formación de una banda, la banda que toca tus canciones.

Ferrous Lineup es software libre, publicado bajo la Licencia Pública General de GNU, versión 3 o posterior.

Compilarlo desde el código fuente

Para usar Ferrous Lineup alcanzan las descargas ya preparadas (ver Instalarlo, en el manual). Para leer su código con este libro, cambiarlo y ejecutar sus tests, compílalo tú. Son unos pocos comandos en Linux, macOS o Windows.

1. Instalar Rust

Rust se instala con **rustup**, una pequeña herramienta que descarga el compilador (`rustc`) y la herramienta de compilación (`cargo`) y los mantiene al día:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

En Windows, descarga y ejecuta en cambio `rustup-init.exe` desde https://rustup.rs. Después reinicia la terminal y comprueba que funcionó:

```
cargo --version
rustc --version
```

Ferrous Lineup necesita Rust 1.95 o más nuevo, como dice `rust-version` en el `Cargo.toml` raíz: usa la edición 2024 del lenguaje (ver Ediciones), las cadenas de `if let`, que llegaron en Rust 1.88, y los bloqueos de archivos, que llegaron en 1.89, y egui y eframe, que dibujan la ventana, necesitan 1.95. Si tu Rust es más viejo, `rustup update` lo arregla.

2. Instalar las herramientas de compilación del sistema

Rust descarga sus propias bibliotecas, los *crates*, por su cuenta, pero necesita un enlazador del sistema, y en Linux algunos crates hablan con bibliotecas que trae el sistema: ALSA para el sonido, y X11 o Wayland para las ventanas.

Debian o Ubuntu:

```
sudo apt install build-essential pkg-config libasound2-dev \
    libxkbcommon-dev libxkbcommon-x11-0 libwayland-dev
```

Fedora:

```
sudo dnf install gcc pkgconf-pkg-config alsa-lib-devel \
    libxkbcommon-devel libxkbcommon-x11 wayland-devel
```

Los diálogos de archivos usan el portal del escritorio, que GNOME y KDE ya tienen; en un gestor de ventanas sin escritorio, instala `xdg-desktop-portal-gtk`. El MIDI pasa por el *secuenciador* de ALSA, que los sistemas de escritorio ya cargan; si el menú MIDI de la aplicación dice que MIDI no está disponible, cárgalo con `sudo modprobe snd-seq`.

macOS: las herramientas de línea de comandos de Apple traen el enlazador y las bibliotecas del sistema:

```
xcode-select --install
```

Windows: cuando `rustup-init.exe` lo pregunte, deja que instale las Visual Studio Build Tools; Rust usa su enlazador. Después usa los comandos de abajo en PowerShell o en el Símbolo del sistema.

3. Conseguir el código y ejecutarlo

Clona el repositorio (Git viene con las herramientas de compilación en macOS y Linux; en Windows, instálalo desde https://git-scm.com) y compila:

```
git clone https://github.com/naelolaiz/ferrous-lineup
cd ferrous-lineup
cargo run --release
```

La primera compilación tarda unos minutos, porque Cargo compila cada dependencia; las siguientes compilan solo lo que cambió. `--release` activa las optimizaciones, que el código de audio necesita para llegar a tiempo. Mientras trabajas en el código, `cargo run` solo compila más rápido, y conserva la consola donde aparecen los mensajes y los pánicos.



La primera vez, una ventana encima ofrece sonidos de instrumentos reales; SoundFonts y General MIDI dice qué hace.

La ventana principal, en el manual, dice para qué sirve cada parte.

4. Desde la línea de comandos

El programa acepta algunos argumentos propios. Después de `cargo run`, `--` dice que lo que sigue es para el programa, no para Cargo:

```
cargo run --release -- my-song.lineup                          # open a
project
cargo run --release -- --demos                                 # list the
demos
cargo run --release -- --trace "Frère Jacques" --out round.tsv   # play a
demo headless, note by note
cargo run --release -- --help
```

`--trace` es la traza de reproducción: toca un proyecto o una demostración como lo hace **Archivo > Exportar WAV…**, sin ventana ni tarjeta de sonido, y escribe en una tabla cada nota que empieza.

5. Ejecutar los tests

```
cargo test
```

ejecuta cada test de cada crate del workspace. Cómo se prueba el código dice qué tipos hay y cómo ejecutar algunos de ellos.

6. Construir los libros

Los libros se escriben con mdBook, una herramienta de Rust. Instálalo una vez, en la versión que usa el flujo de trabajo, y construye:

```
cargo install mdbook --version 0.5.4 --locked
cd book-es
mdbook serve --open
```

`mdbook serve` construye el sitio web, lo abre, y lo vuelve a construir cada vez que guardas una página. `book/`, `manual/` y `manual-es/` funcionan igual. Cómo se construyen los libros también imprime el PDF.

Comandos útiles

Comando	Qué hace
`cargo build`	Compila todo, en modo de depuración
`cargo run`	Compila y arranca la aplicación
`cargo test`	Ejecuta todos los tests
`cargo fmt`	Da formato a todo el código en el estilo estándar de Rust
`cargo clippy`	Ejecuta el linter, que sugiere mejores maneras de escribir las cosas
`cargo doc --open`	Construye la documentación de la API a partir de los comentarios `///` y la abre

Cinco crates, y qué puede saber cada uno

Archivos: `Cargo.toml`, el `Cargo.toml` de cada crate en `crates/`, y `crates/lineup-core/src/lib.rs`

Ferrous Lineup es un solo programa, `ferrous-lineup`, construido a partir de cinco crates de Rust. Este capítulo trata de cómo están divididos y por qué: qué es Cargo, qué son un crate, un workspace y un módulo, un primer tipo pequeño y el Rust que muestra, y las pocas reglas que respeta todo el programa, cada una explicada en un capítulo propio más adelante en el libro. La última sección es un mapa de dónde vive cada parte de la aplicación.

Cargo

Cargo es la herramienta de compilación y el gestor de paquetes de Rust en uno. Lee los archivos `Cargo.toml`, descarga dependencias de crates.io, compila todo en el orden correcto, ejecuta las pruebas y mucho más. Casi nunca llamarás al compilador `rustc` directamente.

Crates

Un **crate** es la unidad de compilación de Rust: una librería o un programa. Ferrous Lineup son cinco crates, cada uno con un trabajo que puede probarse por su cuenta:

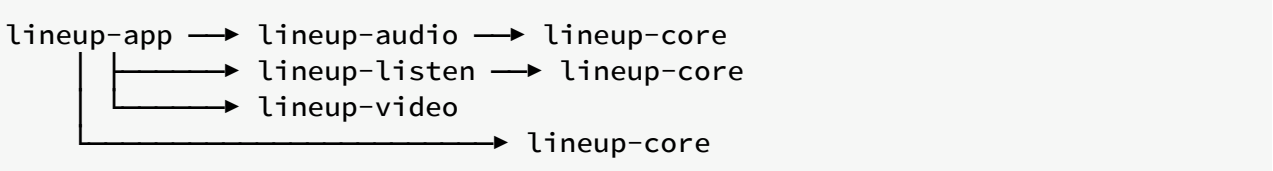
Crate	Tipo	Función
`lineup-core`	librería	La música y la canción: notas, escalas, acordes y armonía, la hoja de acordes, los patrones y la lista de reproducción, los archivos de proyecto y los diffs, el historial de deshacer, los archivos MIDI, las partituras guía, la banda, y la biblioteca de estilos, ritmos y canciones. Sin audio y sin interfaz.
`lineup-audio`	librería	El motor en tiempo real y el renderizador sin conexión: baterías y samplers, el Lineup Osc, los instrumentos SoundFont, la reproducción

Crate	Tipo	Función
		de canciones y patrones, los efectos, la exportación WAV, y la traza de cada nota tocada.
`lineup-listen`	librería	Decodifica una grabación (MP3, WAV, FLAC u Ogg Vorbis) y escucha sus pulsos, tonalidad, acordes y compás. Experimental.
`lineup-video`	librería	Pinta las vistas de la aplicación en memoria y las codifica como H.264 y MP3 en un archivo MP4.
`lineup-app`	binario (programa)	La ventana, construida con `egui`, y el MIDI a través de `midir`. Es dueño del estado que une los otros cuatro, y produce el ejecutable `ferrous-lineup`.

¿Por qué cinco crates en lugar de uno?

- **Límites claros.** `lineup-core` *no puede* depender de la interfaz por accidente, porque no incluye `eframe` como dependencia. El compilador hace cumplir la arquitectura por nosotros.
- **Pruebas más rápidas.** `lineup-core` depende solo de `serde` y `ron`, así que `cargo test -p lineup-core` compila un crate pequeño, no todo el toolkit de GUI, los drivers de audio ni los codificadores de video.
- **Las dependencias pesadas quedan donde hacen falta.** Solo `lineup-listen` decodifica formatos de audio, y solo `lineup-video` compila los codificadores de H.264 y MP3 desde sus fuentes en C.
- **Reutilización.** Sobre `lineup-core` y `lineup-audio` podría construirse un renderizador de línea de comandos o un plugin sin tocar el código de la ventana. La traza de reproducción ya funciona así: `ferrous-lineup --trace` toca una canción con `lineup-audio` sin abrir ninguna ventana (ver La traza de reproducción).

Las dependencias solo apuntan en una dirección:



`lineup-video` no sabe nada de canciones: recibe imágenes y sonido y hace un archivo con ellos.

Nombre del paquete y nombre del programa

El programa se compila a partir del paquete `lineup-app`, pero el archivo que escribe Cargo se llama `ferrous-lineup`. Los dos nombres están en lugares distintos de `crates/lineup-app/Cargo.toml`:

```
[package]
name = "lineup-app"

[[bin]]
name = "ferrous-lineup"
path = "src/main.rs"
```

`[package] name` es como Cargo y los demás crates se refieren al paquete (`cargo test -p lineup-app`). `[[bin]]` describe un programa que compila el paquete; los corchetes dobles son la forma de TOML de decir «una entrada de una lista», porque un paquete puede compilar varios programas. Sin una entrada `[[bin]]`, Cargo llamaría al programa como el paquete, `lineup-app`. Un crate llamado `lineup-core` se escribe `lineup_core` en las líneas `use`, porque un guion es un signo menos en Rust y no puede formar parte de un nombre; Cargo convierte los guiones en guiones bajos por ti.

El workspace

Un **workspace** es un grupo de crates que se compilan juntos y comparten un archivo `Cargo.lock` y una carpeta de salida `target/`. Mira el `Cargo.toml` de nivel superior:

```toml
[workspace]
resolver = "3"
members = [
    "crates/lineup-core",
    "crates/lineup-audio",
    "crates/lineup-listen",
    "crates/lineup-video",
    "crates/lineup-app",
]

[workspace.package]
version = "0.4.0"
edition = "2024"
license = "GPL-3.0-or-later"
repository = "https://github.com/naelolaiz/ferrous-lineup"
rust-version = "1.95"

[workspace.dependencies]
lineup-core = { path = "crates/lineup-core" }
cpal = "0.18"
...
```

- `members` enumera los crates.
- `[workspace.package]` contiene los ajustes compartidos por todos los crates. Cada crate escribe `version.workspace = true` para heredarlos, de modo que el número de versión vive en un solo lugar. `rust-version` es el compilador de Rust más antiguo que puede compilar el código; uno más antiguo se detiene con un mensaje claro en lugar de un error confuso.
- `[workspace.dependencies]` fija la versión de cada dependencia una vez. Un crate escribe `cpal.workspace = true` para usarla. Así todos los crates usan la misma versión de cada librería.

Una versión como `"0.18"` significa «0.18.0 o cualquier 0.18.x más nueva». Las versiones exactas que eligió Cargo quedan registradas en `Cargo.lock`, que se incluye en el repositorio para que todos compilen exactamente el mismo código.

Ediciones

`edition = "2024"` elige la **edición** del lenguaje Rust. Las ediciones permiten a Rust hacer pequeños cambios incompatibles cada pocos años sin romper el código antiguo: cada crate declara en qué edición está escrito, y los crates de ediciones distintas funcionan bien juntos.

Módulos

Dentro de un crate, el código se organiza en **módulos**. Un módulo suele ser un archivo. En `crates/lineup-core/src/lib.rs`:

```
pub mod pitch;   // loads src/pitch.rs as the module `pitch`
pub mod tempo;   // loads src/tempo.rs as the module `tempo`
// ... and about thirty more

pub use pitch::Note;   // re-export: users can write lineup_core::Note
pub use tempo::Tempo;
```

- `mod pitch;` le dice al compilador que busque `pitch.rs` junto a `lib.rs`.
- `pub` lo hace visible fuera del crate. Sin `pub`, todo en Rust es **privado** por defecto, visible solo dentro de su propio módulo.
- `use` trae un nombre al ámbito, como `import` en otros lenguajes. `pub use` además lo reexporta.

Un módulo puede tener módulos propios. `harmony.rs` declara `pub mod chord;`, y el compilador lo encuentra en `harmony/chord.rs`: la carpeta lleva el nombre del módulo al que pertenece.

`lib.rs` es la **raíz del crate** de una librería; `main.rs` es la raíz del crate de un programa. Observa que el crate se llama `lineup-core` en `Cargo.toml` pero `lineup_core` en el código: el código Rust no puede contener `-` en los nombres, así que Cargo lo sustituye por `_`.

Comentarios de documentación

Los comentarios que empiezan por `///` documentan el elemento que tienen debajo, y `//!` documenta el módulo o crate que los contiene. Están escritos en Markdown. Ejecuta `cargo doc --open` para verlos renderizados como un sitio web navegable: cada tipo y función de Ferrous Lineup aparece con su explicación.

Un primer tipo: `Note`

Casi todo `lineup-core` son tipos pequeños con funciones propias. El más pequeño, `Note` en `pitch.rs`, muestra las piezas que usa cada capítulo posterior. Una nota se guarda como su número MIDI, de 0 a 127, y el do central es el 60:

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash,
Serialize, Deserialize)]
#[serde(from = "u8", into = "u8")]
pub struct Note(u8);
```

`Note` es un **tuple struct**: un struct cuyos campos no tienen nombre y se leen como `.0`, `.1`, etc. Con un solo campo que le da a un número un tipo propio, es un *newtype*, que Clases de altura, notas y su escritura explica junto con la música de la nota. El campo no tiene `pub`, así que el código fuera de `pitch.rs` no puede escribir `Note(200)`.

Métodos: bloques `impl`

Las funciones de un tipo se escriben aparte de sus datos, en un bloque `impl`:

```rust
impl Note {
    /// The lowest MIDI note.
    pub const MIN: Note = Note(0);
    /// The highest MIDI note.
    pub const MAX: Note = Note(127);
    /// Middle C.
    pub const C4: Note = Note(60);
    // ...

    /// Creates a note from a MIDI number, clamping it to 0..=127.
    pub fn new(midi: u8) -> Self {
        Note(midi.min(Self::MAX.0))
    }

    /// The MIDI note number.
    pub fn midi(self) -> u8 {
        self.0
    }
    // ...
}
```

- `pub const C4` es una **constante asociada**, que se escribe `Note::C4`.
- `Self` es el tipo para el que es el `impl`, aquí `Note`.
- Una función cuyo primer parámetro es `self` es un **método**, y se llama como `note.midi()`. Una sin `self`, como `new`, es una **función asociada**, y se llama como `Note::new(60)`. Rust no tiene constructores; `new` es solo el nombre de costumbre.
- La última expresión de una función, escrita sin `;`, es lo que devuelve.
- `midi` recibe `self` por valor, no `&self`: una `Note` es `Copy`, así que pasarla copia un solo byte.

Traits, y `derive`

Un **trait** es un conjunto de métodos que un tipo promete tener, como una interfaz en otros lenguajes. El `fmt::Display` de la biblioteca estándar dice cómo escriben un valor `{}` en `format!` y `to_string()`:

```
impl fmt::Display for Note {
    /// Writes the note as a name and octave, such as `C#4`.
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}{}", self.pitch_class(), self.octave())
    }
}
```

Así, `Note::new(61).to_string()` es `"C#4"` (o `"Do#4"` si las notas se nombran en solfeo), y cada lugar que muestra una nota la escribe de la misma manera.

Muchos traits son iguales para cualquier tipo: comparar campo por campo, copiar, imprimir para depurar. `#[derive(...)]` le pide al compilador que los escriba por ti:

Trait	Da
`Debug`	la impresión con `{:?}`, para depurar y para los tests que fallan
`Clone`, `Copy`	una copia del valor; con `Copy`, asignar copia en lugar de mover
`PartialEq`, `Eq`	`==` y `!=`
`PartialOrd`, `Ord`	`<`, `>` y ordenar
`Hash`	usarlo como clave en un `HashMap`

`Serialize` y `Deserialize` vienen de serde, no del compilador (Guardar). `Default` es el trait de «un valor razonable para empezar»; `Tempo` escribe el suyo, 130 BPM:

```
impl Default for Tempo {
    fn default() -> Self {
        Tempo { bpm: 130.0 }
    }
}
```

Números: as

Rust nunca cambia el tipo de un número por su cuenta. `as` lo dice, como en `octave`:

```
pub fn octave(self) -> i8 {
    (self.0 / 12) as i8 - 1
}
```

`self.0 / 12` es un `u8`, que no puede ser negativo, y la octava de las notas más graves es -1, así que el resultado de la división se vuelve un `i8` antes de restarle 1. Los índices de una lista deben ser `usize`, así que `as usize` aparece donde un número elige un elemento.

Pánicos

Cuando un programa llega a algo que nunca debe pasar, como un índice más allá del final de una lista, Rust entra en **pánico** (*panic*): detiene el hilo y escribe dónde y por qué. `assert!`, `assert_eq!`, `unwrap` y `expect` entran en pánico a propósito cuando su condición falla, que es como falla un test (Cómo se prueba el código). Un pánico significa un error del programa, nunca una entrada mala: un archivo de proyecto dañado es un error que la aplicación muestra, y los tests se aseguran de que ningún archivo pueda hacer entrar en pánico al lector (Guardar).

Reglas que respeta todo el programa

Unas pocas decisiones dan forma a cada parte del código. Cada una tiene un capítulo que la muestra en acción:

1. **El hilo de audio nunca espera.** La tarjeta de sonido pide muestras cada pocos milisegundos, y una respuesta tardía es un clic. El hilo de audio nunca toma un lock, nunca reserva memoria y nunca hace una llamada al sistema; lee comandos de una cola y deja sus mensajes en otras. Ver Dos hilos.
2. **La ventana es dueña del proyecto; el hilo de audio guarda su propia copia** de lo justo que necesita para sonar. Los dos nunca comparten datos, así que nunca necesitan un lock.
3. **Una sola manera de avisar al hilo de audio.** Cada edición, y cada deshacer y rehacer, compara el proyecto antes y después y envía las diferencias. Hay un solo trozo de código que acertar, y se prueba por su cuenta. Ver Avisar al hilo de audio de lo que cambió.
4. **Guardar copias, no instrucciones.** Deshacer guarda copias enteras del proyecto, no una lista de cómo revertir cada edición. Una copia no puede revertirse mal. Ver Deshacer y rehacer.
5. **Las cosas que se refieren unas a otras por número cambian juntas.** Un clip dice «patrón 2»; si se elimina el patrón 1, el clip tiene que decir ahora «patrón 1».

El proyecto es dueño de todas sus listas y solo deja que cambien a través de sus métodos. Ver El proyecto.

6. **Un archivo es un contrato con el futuro.** El formato de archivo tiene sus propios tipos, separados de los que usa el programa, y un número de versión que solo cambia cuando hace falta. Ver Guardar.
7. **Dibujar está separado de cambiar.** El código que dibuja una ventana devuelve una lista de acciones, y la aplicación las aplica en un solo lugar, con la misma función de edición. Ver Los editores.
8. **La teoría musical como tipos.** Las clases de alturas, las escalas, los acordes y «lo que está sonando» son tipos pequeños de Rust con métodos, probados como cualquier otro código. Ver Clases de altura, notas y su escritura.
9. **Las herramientas son funciones simples.** Cada herramienta de edición recibe notas y devuelve notas nuevas, sin saber nada de ventanas ni de ratones. Ver Herramientas de edición como funciones simples.
10. **El conocimiento como datos.** Los estilos, los ritmos y las canciones se escriben como cadenas cortas en tablas que una persona puede leer y revisar, en números romanos y pasos de un compás, de modo que la misma tabla suena en cualquier tonalidad y compás. Ver La biblioteca.
11. **La banda solo escribe patrones.** Probar un estilo o tocar una hoja de acordes es una edición corriente, así que deshacer, guardar y el hilo de audio funcionan sin saber que la banda existe. Ver Cómo toca la banda una hoja de acordes.
12. **El mismo código, más rápido que el tiempo real.** Exportar un archivo WAV, un video o una traza ejecuta exactamente el mismo código que usa la tarjeta de sonido, en un hilo propio, tan rápido como puede el equipo. Ver Exportar.

Dónde vive cada parte

Cada archivo fuente empieza con un comentario `//!` que dice para qué sirve; este mapa está hecho con sus primeras líneas. `lineup-core`:

Módulo	Qué contiene
`pitch.rs`, `tempo.rs`, `meter.rs`	las notas y sus nombres, el tempo, los compases
`pattern.rs`, `project.rs`, `playlist.rs`, `arrangement.rs`	patrones, canales, pistas y clips, y la canción tal como suena
`sequencer.rs`, `swing.rs`	dónde está la reproducción, paso a paso
`diff.rs`, `history.rs`, `file.rs`	qué cambió, deshacer, guardar

Módulo	Qué contiene
`harmony/`	escalas, acordes, cifrados, tonalidades, conjuntos, series, micromodos, oír la armonía, conducción de voces
`tools.rs`	las herramientas del piano roll
`chord_sheet/`, `band/`, `bars.rs`	la hoja de acordes y la banda que la toca
`library/`	estilos, ritmos, canciones y demostraciones, escritos como datos
`lead_sheet/`	la melodía en notación
`midi.rs`, `midi_file.rs`, `record.rs`	mensajes MIDI, archivos MIDI, grabación
`mixer.rs`, `effects.rs`, `score_order.rs`	los ajustes del mezclador
`i18n/`	el castellano de cada texto que escribe la aplicación

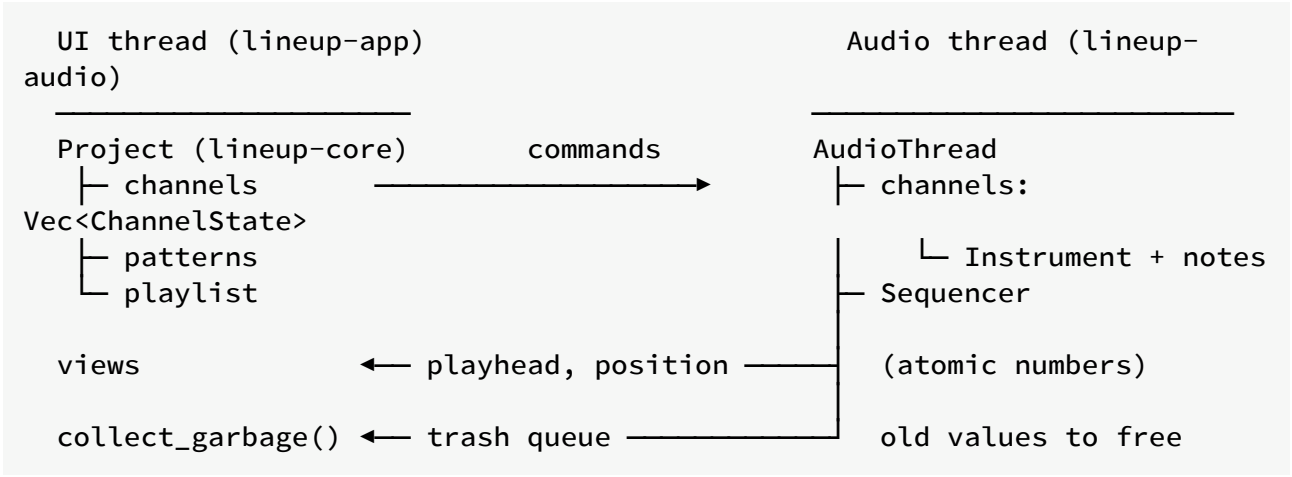
`lineup-audio`: `engine.rs` (la tarjeta de sonido y el hilo de audio), `instrument.rs`, `drums.rs`, `timbre.rs`, `sample.rs` y `soundfont.rs` (los instrumentos), `effects.rs` y `reverb.rs`, `levels.rs`, `metronome.rs`, `render.rs` (tocar sin tarjeta de sonido) y `trace.rs`.

`lineup-app`: `app.rs` y los módulos de `app/` (menús, archivos, edición, exportación, MIDI, ajustes), un módulo por ventana (`channel_rack.rs`, `piano_roll.rs`, `playlist.rs`, `chord_sheet.rs`, `harmony_keys.rs`, `library_browser.rs`, `lead_sheet.rs`, `twelve_tone.rs`, `micromodes.rs`, `style_editor.rs`, `ear_training.rs`), y las piezas compartidas: `theme.rs`, `widgets.rs`, `icons.rs` e `i18n.rs`.

Dos hilos: la ventana y el audio

Archivos: `crates/lineup-audio/src/engine.rs`, `crates/lineup-audio/src/dsp.rs` y `crates/lineup-app/src/midi.rs`

La ventana y el sonido viven en hilos distintos. El hilo de la ventana dibuja unas sesenta veces por segundo y responde a clics y teclas; el hilo de audio rellena el búfer de la tarjeta de sonido cada pocos milisegundos y no debe llegar tarde nunca. Cómo se hablan los dos es la decisión más importante del programa, y es donde las reglas de propiedad de Rust empiezan a ganarse el sueldo.



Cómo funciona el audio digital

El sonido es una vibración. Una tarjeta de sonido lo reproduce moviendo un altavoz según una lista de números llamados **muestras**, típicamente 44.100 o 48.000 por segundo (la **frecuencia de muestreo**). Cada muestra es un valor entre −1.0 y 1.0, la posición del altavoz en ese instante. Con dos **canales** (estéreo), las muestras van **entrelazadas**: izquierda, derecha, izquierda, derecha…

La tarjeta de sonido no nos espera. Reproduce desde un búfer pequeño y, cada pocos milisegundos, pide al programa que lo rellene llamando a una función que nosotros proporcionamos: el **callback de audio**. Se ejecuta en un **hilo de audio** separado y de alta prioridad.

La regla de oro del hilo de audio

Si el callback llega tarde, aunque sea una vez, el búfer se vacía y se oye un clic o un hueco (un *underrun* o *xrun*). Por eso el código del hilo de audio nunca debe hacer nada que pueda tardar un tiempo impredecible:

- no usar **cerrojos** (un `Mutex` retenido por la interfaz podría bloquearlo),
- no **reservar ni liberar memoria** (`Vec::push` más allá de su capacidad, `String`, `Box::new`, ni soltar el último dueño de cualquiera de ellos),
- no hacer **I/O de archivos, red o consola** (`println!` incluido),
- no esperar a otros hilos.

Todo lo demás en este capítulo se deduce de esta regla.

Enviar comandos con un ring buffer

La interfaz nunca cambia el estado del audio directamente. Envía mensajes, valores de un enum con casi cuarenta variantes. Algunas de ellas:

```
pub enum Command {
    Play,
    Seek(usize),
    Stop,
    SetTempo(Tempo),
    SetMasterVolume(f32),
    InsertChannel {
        index: usize,
        instrument: Instrument,
        notes: ChannelNotes,
        volume: f32,
        muted: bool,
    },
    RemoveChannel(usize),
    SetNotes {
        pattern: usize,
        channel: usize,
        notes: Vec<NoteEvent>,
    },
    // ...
}
```

Es un **enum** de Rust. A diferencia de los enums de C o Java, cada variante puede llevar sus propios datos: `Seek` lleva un número de paso, `SetNotes` toda una lista de notas. Un valor de enum es exactamente una de sus variantes.

Los mensajes viajan por un **ring buffer** del crate `rtrb`: una cola de tamaño fijo, reservada una sola vez, con un escritor (el `Producer`) y un lector (el `Consumer`). Añadir y extraer nunca bloquea ni reserva memoria, así que el hilo de audio puede leerlo con seguridad.

```
let (command_producer, command_consumer) =
rtrb::RingBuffer::new(COMMAND_QUEUE_SIZE);
let (trash_producer, trash_consumer) =
rtrb::RingBuffer::new(TRASH_QUEUE_SIZE);
let (live_producer, live_consumer) =
rtrb::RingBuffer::new(LIVE_QUEUE_SIZE);
let (midi_producer, midi_consumer) =
rtrb::RingBuffer::new(MIDI_OUT_QUEUE_SIZE);
```

`RingBuffer::new` devuelve una **tupla** de dos valores, que separamos con un **patrón** `let (a, b) = ...`. Hay cuatro colas: los comandos de la interfaz, los valores que la interfaz debe liberar (la basura, más abajo), las teclas de un teclado MIDI (MIDI) y los mensajes para los canales de salida MIDI. En cada una, un extremo se queda en un hilo y el otro se mueve al otro.

La propiedad lo hace seguro

Aquí es donde brilla Rust. `Producer` se puede usar desde un hilo y `Consumer` desde otro, pero solo hay uno de cada, y las reglas de **propiedad** de Rust lo garantizan:

- Cada valor tiene exactamente un **dueño**. Cuando el dueño desaparece, el valor se libera (se hace drop).
- Asignar o pasar un valor lo **mueve** al nuevo dueño, y el nombre antiguo ya no se puede usar.

Una vez que el `consumer` se mueve al callback de audio, el código de la interfaz físicamente ya no puede tocarlo: el compilador lo rechazaría. Las condiciones de carrera no son «improbables»; son errores de compilación.

Comandos que llevan datos

`Command` no es `Copy`, porque un `Instrument` puede tener un `Arc<SampleData>` y `SetNotes` tiene un `Vec`. Moverlo por el ring buffer sigue estando bien: el valor se mueve a la casilla ya reservada de la cola y sale de nuevo, sin reservar memoria. El `Vec` de dentro se reservó en el hilo de la interfaz, antes de enviarlo.

Cuando la cola está llena

La cola guarda 4096 comandos, suficientes para un proyecto grande entero. Si alguna vez se llena, la interfaz se adelantó al hilo de audio, que la vacía al principio de cada bloque. `send_wait` no suelta el comando: cede y vuelve a intentarlo, hasta mil veces.

```
pub fn send_wait(&mut self, mut command: Command) -> bool {
    for _ in 0..SEND_WAIT_ROUNDS {
        match self.commands.push(command) {
            Ok(()) => return true,
            Err(rtrb::PushError::Full(returned)) => command = returned,
        }
        // A stream that has reported an error will not empty the
        // queue, so waiting for room would only freeze the window.
        if self.problems.broken.load(Ordering::Relaxed) {
            return false;
        }
        std::thread::yield_now();
    }
    false
}
```

`push` recibe el comando por valor. Cuando la cola está llena lo devuelve dentro del error, `PushError::Full(returned)`, para que el bucle pueda intentarlo de nuevo con él: no se copia nada ni se pierde nada.

Abrir la tarjeta de sonido

`AudioEngine::start` usa el crate `cpal`, que funciona en Linux (ALSA), macOS (CoreAudio) y Windows (WASAPI), así que el mismo código funciona en los tres.

```
pub fn start() -> Result<Self, EngineError> {
    let host = cpal::default_host();
    let device = host
        .default_output_device()
        .ok_or(EngineError::NoOutputDevice)?;
    let supported = device
        .default_output_config()
        .map_err(EngineError::Config)?;
    ...
}
```

`Option`, `Result` y el operador `?`

Rust no tiene `null` ni excepciones. En su lugar:

- **`Option<T>`** es `Some(value)` o `None`. `default_output_device()` devuelve `Option<Device>` porque puede no haber tarjeta de sonido.
- **`Result<T, E>`** es `Ok(value)` o `Err(error)`. `default_output_config()` devuelve `Result<SupportedStreamConfig, cpal::Error>`.

Ambos son enums normales de la librería estándar. Al ser tipos, no puedes olvidar manejar el caso que falta o falla: el compilador no te dejará usar un `Device` que quizá no tienes.

- `.ok_or(EngineError::NoOutputDevice)` convierte un `Option` en un `Result`, usando nuestro error para el caso `None`.
- `.map_err(EngineError::Config)` envuelve el error de cpal en nuestra propia variante. Observa que `EngineError::Config` se usa aquí como una función: las variantes de enum con datos se pueden llamar como constructores.
- **`?`** significa «si esto es un `Err`, devuélvelo de la función ahora mismo; si no, dame el valor `Ok`». Sustituye largas cadenas de `if error then return`.

Un tipo de error propio

```
#[derive(Debug)]
pub enum EngineError {
    NoOutputDevice,
    Config(cpal::Error),
    UnsupportedFormat(SampleFormat),
    BuildStream(cpal::Error),
    Play(cpal::Error),
}

impl fmt::Display for EngineError { ... }
impl std::error::Error for EngineError {}
```

Un enum enumera todas las formas en que puede fallar el arranque del motor. Implementar `Display` da a cada una un mensaje legible, y el `impl std::error::Error` vacío lo marca como un tipo de error en regla que funciona con el resto del ecosistema de Rust. La aplicación lo convierte en texto con `err.to_string()` y sigue funcionando sin sonido: todas las ventanas funcionan, y **Edición > Ajustes… > Audio** ofrece volver a intentarlo.

Genéricos: una función, muchos tipos de muestra

Distintas tarjetas de sonido quieren muestras en distintos formatos: floats de 32 bits, enteros de 16 bits, etc. Generamos `f32` internamente y convertimos en el último momento. En lugar de escribir el código del stream seis veces, lo escribimos una vez como **función genérica**:

```
fn build_stream<T>(
    device: &cpal::Device,
    config: StreamConfig,
    mut audio_thread: AudioThread,
    problems: Arc<StreamProblems>,
) -> Result<cpal::Stream, EngineError>
where
    T: SizedSample + FromSample<f32>,
{ ... }
```

`T` es un tipo marcador de posición, y la cláusula `where` enumera los **trait bounds**: `T` puede ser cualquier tipo que sea una muestra de cpal (`SizedSample`) y que se pueda crear a partir de un `f32` (`FromSample<f32>`). Dentro, `T::from_sample(x)` hace la conversión.

Luego `start` elige la versión correcta con un `match`:

```
let stream = match sample_format {
    SampleFormat::F32 => {
        build_stream::<f32>(&device, config, audio_thread,
Arc::clone(&problems))
    }
    SampleFormat::I16 => {
        build_stream::<i16>(&device, config, audio_thread,
Arc::clone(&problems))
    }
    ...
    other => return Err(EngineError::UnsupportedFormat(other)),
}?;
```

- `match` compara un valor con **patrones** y ejecuta la primera rama que encaja. El compilador comprueba que todos los valores posibles están cubiertos. `other` es un patrón comodín que además da nombre al valor.
- `build_stream::<f32>` (la sintaxis «turbofish» `::<>`) elige `T = f32` . El compilador genera una copia separada y totalmente optimizada de la función para cada tipo usado. Esto se llama **monomorfización**, y significa que los genéricos no cuestan nada en tiempo de ejecución.
- `audio_thread` se mueve a la rama que se ejecute. El compilador sabe que solo una rama se ejecuta, así que esto está permitido.

El callback de audio es un closure

Dentro de `build_stream`:

```
let data_callback = move |output: &mut [T], _: &cpal::OutputCallbackInfo|
{
    audio_thread.process(output);
};

let error_callback = move |err: cpal::Error| {
    problems.broken.store(true, Ordering::Relaxed);
    if let Ok(mut slot) = problems.error.lock() {
        *slot = Some(err.to_string());
    }
    eprintln!("audio stream error: {err}");
};

device
    .build_output_stream(config, data_callback, error_callback, None)
    .map_err(EngineError::BuildStream)
```

`|args| cuerpo` es un **closure**: una función anónima que puede usar variables del lugar donde se escribió. La palabra clave `move` hace que el closure tome la **propiedad** de `audio_thread`, y con ella del consumer de comandos y de todo lo demás con lo que trabaja el hilo de audio. Eso es necesario porque el closure se ejecutará en otro hilo, mucho después de que `build_stream` haya devuelto, así que no puede limitarse a tomar prestadas variables locales.

La firma de cpal exige que el callback sea `FnMut(&mut [T], &OutputCallbackInfo) + Send + 'static`:

Límite	Significado
`FnMut`	un closure que puede cambiar los valores que posee (los instrumentos avanzan su estado)
`Send`	seguro de mover a otro hilo
`'static`	posee todo lo que usa, sin tomar prestado nada que pueda desaparecer

Si intentáramos compartir algo inseguro, como una referencia simple a los datos de la interfaz, el programa no compilaría. Así convierte Rust la regla de oro del audio en comprobaciones que el compilador hace por ti.

`&mut [T]` es un **slice mutable**: una vista prestada del búfer de la tarjeta de sonido, que rellenamos pero no poseemos.

El callback de error es el único lugar que puede tomar un cerrojo e imprimir: cpal lo llama fuera del callback de audio, cuando algo falla mientras el stream funciona, como una interfaz desenchufada. Guarda el problema más reciente para que la ventana lo muestre.

`AudioThread::process`

El callback de audio entrega cada búfer a un solo método. Su esquema:

```
pub(crate) fn process<T>(&mut self, output: &mut [T])
where
    T: SizedSample + FromSample<f32>,
{
    self.flush_trash();
    self.flush_midi_backlog();
    while let Ok(command) = self.commands.pop() {
        self.apply(command);
    }
    while let Some(message) = self.live_input.as_mut().and_then(|keys|
keys.pop().ok()) {
        self.play_live(message);
    }

    for frame in output.chunks_mut(self.output_channels) {
        if self.playing
            && let Some(step) = self.sequencer.tick()
        {
            self.playhead.store(step, Ordering::Relaxed);
            self.start_notes(step);
        }
        // ... mix the channels, the reverb and the metronome ...
    }
    self.send_midi();
}
```

- `while let Ok(command) = ...` sigue en bucle mientras el patrón encaja, de modo que vacía todos los comandos en espera y se detiene cuando la cola está vacía.
- `as_mut` convierte `&mut Option<Consumer>` en `Option<&mut Consumer>`, y `and_then` le pide una tecla solo si hay un consumer.
- `chunks_mut(channels)` divide el búfer en **frames**, una muestra por canal de salida.
- `if a && let Some(x) = b { ... }` es una **let chain**: una condición y un patrón combinados en un solo `if`.

El estado del hilo de audio vive en una struct, `AudioThread`, en lugar de en variables sueltas dentro del closure. Así las pruebas pueden ejecutarlo con un `Vec` simple en lugar de una tarjeta de sonido, y el renderizador puede tocar una canción a un archivo con el mismo código.

Mezclar un frame

En cada frame, el instrumento de cada canal hace una muestra. La muestra pasa por los efectos y el volumen del canal, y luego va a los lados izquierdo y derecho según su balance, y a la reverberación tanto como dice su envío (El mezclador):

```
for channel in &mut self.channels {
    let volume = channel.volume.next_value();
    let sample = channel.effects.process(channel.instrument.next_sample())
* volume;
    ...
    dry[0] += kept * gains[0];
    dry[1] += kept * gains[1];
    send += sample * amount;
}
...
// `tanh` gently rounds off peaks above 1.0 instead of clipping them hard.
let stereo = stereo.map(f32::tanh);
```

- `for channel in &mut self.channels` itera sobre referencias mutables, para que cada instrumento pueda avanzar su propio estado.
- Mezclar es simplemente sumar: los sonidos en el aire se suman, y las muestras también.
- Varios tambores fuertes a la vez pueden sumar más de 1.0, y una tarjeta de sonido lo recortaría con dureza. `tanh` (la tangente hiperbólica) deja pasar los valores pequeños casi sin cambios y dobla los grandes suavemente hacia ±1.0, un **soft clipper** (recortador suave).
- `volume.next_value()` es un valor `Smoothed`. Un volumen que salta hace saltar el altavoz, y eso se oye como un clic, así que en cambio se desliza hacia su objetivo: en cada muestra recorre una pequeña fracción de la distancia que queda. Es un *suavizador de un polo*, que se asienta en pocos milisegundos, demasiado rápido para oírse como un fundido pero suficiente para quitar el clic.

Espacio reservado de antemano

```
channels: Vec::with_capacity(MAX_CHANNELS),
```

Un `Vec` tiene una **longitud** (cuántos elementos tiene) y una **capacidad** (cuántos caben en la memoria que tiene). `push` solo reserva memoria cuando la longitud superaría la capacidad. Reservar espacio para `MAX_CHANNELS` al arrancar, en el hilo de la interfaz, significa que `push` en el hilo de audio nunca reserva memoria. La interfaz se niega a añadir más canales que esos, así que el límite no se alcanza nunca. La lista de notas que esperan para empezar tarde, las longitudes de los patrones y el backlog de la basura se reservan del mismo modo.

`Vec::remove(index)` desplaza hacia abajo los elementos siguientes, lo que tampoco reserva memoria. Así es como `RemoveChannel` mantiene los números de canal del hilo de audio iguales a los de la lista de la interfaz.

Liberar memoria en otro sitio: la cola de basura

Cuando se elimina un canal o se sustituyen sus notas, el valor viejo puede tener el último `Arc` de una muestra, o un `Vec` de notas. Liberarlo soltaría esa memoria, y liberar puede tomar un cerrojo dentro del asignador de memoria. Así que el hilo de audio no lo libera. Lo devuelve:

```
fn set_instrument(&mut self, channel: usize, instrument: Instrument) {
    match self.channels.get_mut(channel) {
        Some(state) => {
            let mut old = std::mem::replace(&mut state.instrument,
instrument);
            silence(&mut old, &mut self.midi_out, &mut self.midi_backlog);
            ...
            self.throw_away(Garbage::Instrument(old));
        }
        None => self.throw_away(Garbage::Instrument(instrument)),
    }
}
```

`std::mem::replace(&mut place, new)` pone `new` en `place` y devuelve lo que había. Es la forma de sacar un valor de un campo de un struct del que solo tienes un `&mut`, ya que Rust nunca deja un campo vacío. (`silence` envía los note-offs que el instrumento viejo todavía le debe a un sintetizador de salida MIDI.)

`throw_away` empuja el valor a un segundo ring buffer, que va en la dirección contraria. `Garbage` es un enum con una variante para cada clase de cosa que devuelve el hilo de audio: instrumentos, listas de notas, canales, canciones, listas de tempo, frases, líneas de retardo. El hilo de la interfaz vacía la cola una vez por frame:

```
pub fn collect_garbage(&mut self) {
    while let Ok(garbage) = self.trash.pop() {
        // Dropping a value frees its memory. Here, on the UI thread,
        // that is allowed to take as long as it needs.
        match garbage {
            Garbage::Instrument(instrument) => drop(instrument),
            Garbage::Notes(notes) => drop(notes),
            ...
        }
    }
}
```

`drop(value)` es una función normal del preludio que toma la propiedad y deja que el valor salga de ámbito. Escribirla explícitamente documenta la intención: aquí es donde se libera la memoria.

Una ráfaga de cambios entre dos frames podría llenar la cola de basura. Entonces `throw_away` guarda el valor en un backlog reservado de antemano, y `flush_trash` lo pasa a la cola al principio del bloque siguiente. Solo si también se llena el backlog, lo que significa que la interfaz lleva mucho tiempo sin recoger, el valor se libera en el hilo de audio, como último recurso.

El cursor de reproducción: un entero atómico

La interfaz quiere resaltar el paso que se está reproduciendo. El hilo de audio lo sabe. Compartirlo con un `Mutex` significaría que el hilo de audio podría esperar por el cerrojo, así que usamos un **atómico**:

```
let playhead = Arc::new(AtomicUsize::new(NO_STEP));
// audio thread:
self.playhead.store(step, Ordering::Relaxed);
// UI thread:
match self.playhead.load(Ordering::Relaxed) { NO_STEP => None, step =>
Some(step) }
```

Un `AtomicUsize` es un número que el procesador puede leer y escribir en una sola operación indivisible, de modo que dos hilos pueden usarlo sin cerrojo y sin ver nunca

un valor a medio escribir. Ambos hilos tienen un `Arc` al mismo (`Arc::clone(&playhead)` crea el segundo puntero).

`Ordering::Relaxed` dice que solo nos importa este valor, no el orden de otras escrituras de memoria a su alrededor. Es el orden más barato y basta para mostrarlo. Los órdenes más estrictos (`Acquire`, `Release`, `SeqCst`) son para cuando un valor protege el acceso a otros.

Rust no tiene un paso «nulo», así que `usize::MAX` representa «parado», y `playhead()` lo convierte enseguida en un `Option<usize>` honesto para que ningún otro código vea nunca el número mágico.

La posición con fracción de paso, que necesitan la grabación y las vistas, es un `f64`, y no hay float atómico. Así que se guarda como los bits del `f64` en un `AtomicU64`: `position.to_bits()` en el hilo de audio, `f64::from_bits(bits)` del otro lado. Dos números que siempre deben leerse juntos, como el patrón que suena sobre la canción y su paso, comparten un `AtomicU64`: uno en los 32 bits altos, el otro en los bajos.

Mantener vivo el stream: `Drop`

```
pub struct AudioEngine {
    // The stream is never read, but it must be kept: dropping it stops
audio.
    _stream: cpal::Stream,
    commands: rtrb::Producer<Command>,
    trash: rtrb::Consumer<Garbage>,
    playhead: Arc<AtomicUsize>,
    position: Arc<AtomicU64>,
    ...
}
```

El `cpal::Stream` no se vuelve a usar después de arrancar, pero hay que conservarlo: cuando un valor se libera, Rust ejecuta su código de limpieza (su implementación de `Drop`), y liberar un `Stream` cierra la tarjeta de sonido. Guardarlo en el motor ata la vida del sonido a la del motor. El guion bajo inicial le dice al compilador «sé que este campo no se lee nunca».

Este patrón, en el que poseer un valor significa poseer un recurso y liberarlo lo suelta, se llama **RAII**. Rust lo usa en todas partes: los archivos se cierran, los cerrojos se desbloquean y la memoria se libera sola cuando su dueño desaparece.

Cuando la tarjeta de sonido desaparece

Una interfaz desenchufada, un cambio de dispositivo predeterminado o un servidor de sonido que se reinició hacen que el stream informe de un error. **Edición > Ajustes… > Audio** ofrece entonces **Reintentar** (y **Reiniciar audio** mientras el stream funciona), que vuelve a abrir el dispositivo de salida predeterminado del sistema. El proyecto se queda: se envía entero al hilo de audio nuevo, los puertos MIDI se vuelven a conectar al motor nuevo, y una canción que estaba sonando vuelve a la misma posición y sigue.

El motor viejo se suelta antes de que arranque el nuevo, porque un dispositivo que se vuelve a abrir no puede seguir retenido por el stream muerto.

Dos cosas necesitan cuidado cuando se va un hilo de audio:

Notas que suenan en un sintetizador externo. Una nota enviada por salida MIDI recibe su note-off del hilo de audio que la tocó; cuando ese hilo ya no está, se van con él los note-offs que todavía debía. Los mensajes salen por un hilo pequeño, `midi-out`, que vacía la cola del hilo de audio cada milisegundo. Cuando su cola queda abandonada, su bucle termina, y antes de acabar envía **All Notes Off** (controlador 123) en los dieciséis canales:

```
let sender = move || {
    while !messages.is_abandoned() {
        // ... send what the audio thread queued ...
    }
    // The engine is gone, and with it the note offs of notes still
    // sounding on the synth.
    if let Some(connection) = lock(&output).as_mut() {
        for channel in 0..16 {
            let _ = connection.send(&all_notes_off(channel));
        }
    }
};
```

El hilo de envío del motor nuevo no debe arrancar antes de que el viejo haya enviado su All Notes Off, o ese mensaje podría cortar las primeras notas del motor nuevo. Así que `reconnect_audio` espera al hilo viejo, consultando `JoinHandle::is_finished` como mucho 100 ms (en la práctica cerca de un milisegundo, porque el motor viejo ya no está). No llama a `join()`, que podría esperar para siempre a un puerto MIDI colgado: la ventana nunca debe congelarse por un puerto.

Mensajes que deben llegar. Un note-off perdido deja una nota sonando para siempre en un sintetizador externo, así que la salida MIDI nunca pierde ninguno. Los mensajes que encuentran la cola de salida llena esperan en un backlog reservado y

salen al principio del bloque siguiente. Quitar un canal o sustituir un instrumento envía sus note-offs por el mismo camino, y desconectar un puerto envía All Notes Off en los dieciséis canales antes de que se cierre la conexión.

La traza de reproducción es el caso contrario: graba cada nota para un informe, no para el sonido, así que si su cola se llena, el hilo de audio cuenta los eventos perdidos y sigue. Esperar ahí pondría la traza antes que la música. Cuando el audio se reinicia, la traza sigue en el mismo archivo: se escribe lo que dejó el hilo viejo, y se envía una cola nueva al hilo nuevo.

Probar sin tarjeta de sonido

Las pruebas al final de `engine.rs` construyen un `AudioThread` con ring buffers, empujan comandos, renderizan en un `Vec<f32>` y comprueban el resultado: un golpe en el frame 0 y otro 6000 frames después a 120 BPM, silencio al silenciar, ediciones que surten efecto durante la reproducción e instrumentos viejos que llegan a la basura. Una pequeña struct `Harness` guarda las piezas para que cada prueba sea corta.

El proyecto: patrones, canales, pistas, clips y marcas

Archivos: `crates/lineup-core/src/pattern.rs`, `crates/lineup-core/src/project.rs`, `crates/lineup-core/src/playlist.rs` y `crates/lineup-core/src/arrangement.rs`

Todo aquello con lo que trabaja el usuario es un solo valor de tipo `Project`: los canales del rack de canales, los patrones de notas que tocan, la lista de reproducción que ordena los patrones en una canción, y las marcas a lo largo de ella (secciones, tempo, compás, tonalidad y acordes). Es también exactamente lo que guarda un archivo de proyecto. Este capítulo trata de su forma y de las reglas que mantiene. Las ventanas que lo editan están en Los editores; cómo usarlas está en el capítulo Patrones y la lista de reproducción del manual de usuario.

Notas, pasos y patrones

Un **patrón** es un tramo corto de música, a menudo un compás, con una fila de notas para cada canal. Una nota es una altura, un comienzo, una duración y una velocidad, todo contado en **pasos** (una semicorchea cada uno en 4/4):

```
pub struct NoteEvent {
    pub pitch: Note,
    /// The step the note starts on, counted from 0.
    pub start: usize,
    /// How many steps the note is held for, at least 1.
    pub length: usize,
    /// How hard the note is played, from 1 to 127 (as in MIDI).
    pub velocity: u8,
    /// How many ticks after the start of its first step the note really
    /// starts, from 0 to [`TICKS_PER_STEP`] - 1.
    pub nudge: u8,
    /// How many ticks before the end of its last step the note really
    /// ends.
    pub trim: u8,
}
```

Los botones de paso del rack de canales y el piano roll editan las mismas notas. Un paso «encendido» no es más que una nota de un paso de largo en la altura por defecto, así que `is_on` pregunta si alguna nota empieza ahí:

```rust
pub fn is_on(&self, channel: usize, step: usize) -> bool {
    self.notes(channel).iter().any(|note| note.start == step)
}
```

Una sola clase de datos con dos vistas es como funcionan los estudios basados en patrones, y significa que el hilo de audio tiene una sola clase de cosa que tocar. `nudge` y `trim` colocan una nota entre pasos, para música tocada por una persona o importada de un archivo MIDI; las notas en la cuadrícula los dejan en 0.

Canales: enums con datos

```rust
pub enum SampleSource {
    /// One of the drums built into Ferrous Lineup.
    Builtin(Drum),
    /// A WAV file on disk.
    File(PathBuf),
}

pub enum ChannelKind {
    /// Plays a recorded or built-in sound from start to end for each
    /// note, faster for higher notes and slower for lower ones.
    Sampler(SampleSource),
    /// The built-in "Lineup Osc" synthesizer, which can play chords.
    Synth,
    /// Makes no sound itself: sends its notes out through MIDI, on
    /// MIDI channel `channel` (0 to 15, shown as 1 to 16).
    MidiOut { channel: u8 },
}
```

Un canal tiene un nombre, un volumen, un interruptor de silencio, un número de instrumento General MIDI y un tipo. Los enums se anidan: un canal es un sampler, el Lineup Osc o un canal de salida MIDI, y el sonido de un sampler es o un tambor incorporado o un archivo. Cada hecho se guarda una vez y toda combinación tiene sentido. `MidiOut { channel: u8 }` muestra que una variante también puede tener **campos con nombre**, como un struct.

`PathBuf` es el tipo estándar para una ruta de archivo poseída. No es un `String`, porque los nombres de archivo en algunos sistemas no son texto UTF-8 válido; `path.display()` convierte uno en algo imprimible.

`Drum::ALL` es una constante asociada que contiene todas las variantes, para que los menús puedan iterar sobre los tambores en lugar de enumerarlos a mano.

La forma de un proyecto

Un proyecto tiene una lista de patrones, y una lista de reproducción que dice dónde suena cada uno:

```
pub struct Project {
    pub mixer: crate::Mixer,
    pub tempo: Tempo,
    pub scale: Scale,
    pub time_signature: TimeSignature,
    pub loose_timing: u8,
    pub swing: u8,
    pub(crate) channels: Vec<Channel>,
    pub(crate) patterns: Vec<Pattern>,
    pub(crate) playlist: Playlist,
    pub(crate) band: Option<Band>,
    // ...where each part came from, and the locked channels
}
```

Cada patrón tiene una fila de notas por canal. Los estudios basados en patrones funcionan así: un patrón es «lo que toca cada instrumento durante estos compases», así que el patrón de batería tiene filas vacías para el sintetizador.

La lista de reproducción son dos listas:

```
pub struct Track {
    pub name: String,
    pub muted: bool,
    // ...a display height, which older files leave out
}

pub struct Clip {
    pub pattern: usize,
    pub track: usize,
    pub start: usize,
    /// How many times the pattern plays, one after the other.
    pub repeats: usize,
}

pub struct Playlist {
    pub(crate) tracks: Vec<Track>,
    pub(crate) clips: Vec<Clip>,
    // ...sections, a loop region, and time, tempo, key and chord markers
}
```

Un clip son tres números y un contador de repeticiones: qué patrón, en qué pista, empezando en qué paso de la canción y cuántas veces suena el patrón. Fíjate en lo que *no* está: la duración del clip. Es la duración del patrón por el número de repeticiones, así que se calcula cuando hace falta:

```
pub fn clip_length(&self, clip: &Clip) -> usize {
    self.patterns.get(clip.pattern).map_or(0, Pattern::length) *
clip.repeats
}

impl Clip {
    /// A clip that plays its pattern once.
    pub fn new(pattern: usize, track: usize, start: usize) -> Self {
        Clip { pattern, track, start, repeats: 1 }
    }
}
```

Guardar también la duración supondría dos sitios que deben coincidir. Haz el patrón más largo y la duración guardada de cada clip quedaría mal. Un valor que se puede calcular a partir de otros casi siempre es mejor calcularlo que guardarlo. Lo mismo vale para el final de un clip repetido: es `start + clip_length`, no otro campo.

`Project::playback_arrangement` desenrolla las repeticiones en `Play`s antes de que la canción llegue al hilo de audio (ver Secciones, repeticiones y bucles más abajo); el clip es lo que dibujas, y su `repeats` es el pequeño «×2» que lleva encima.

`map_or(0, Pattern::length)` dice «si hay un patrón, llama a `Pattern::length` sobre él; si no, usa 0». Un método se puede pasar por nombre, como una función, cuando su único argumento es `self`.

Cuatro reglas, un guardián

El comentario de `Project` enumera las reglas que mantiene:

- cada clip de la lista de reproducción se refiere a un patrón y a una pista que existen;
- las secciones están ordenadas y nunca se solapan, y las repeticiones no se salen de rango.

La mayoría de estas reglas tratan de dos listas a la vez, así que ninguna lista puede mantenerlas por sí sola. El `Project` las posee todas, mantiene los campos privados para `lineup-core` y los cambia solo en métodos que respetan todas las reglas. Añadir un canal añade una fila a cada patrón:

```
pub fn add_channel(&mut self, channel: Channel) -> Option<usize> {
    if self.channels.len() >= MAX_CHANNELS {
        return None;
    }
    self.channels.push(channel);
    for pattern in &mut self.patterns {
        pattern.push_row();
    }
    Some(self.channels.len() - 1)
}
```

Borrar un patrón es lo interesante. Los clips se refieren a los patrones por su posición en la lista, así que cuando el patrón 1 desaparece, los clips del patrón 1 deben desaparecer también, y los clips de los patrones 2, 3, 4… deben bajar uno cada uno para que sigan reproduciendo la misma música:

```
pub fn remove_pattern(&mut self, index: usize) -> Option<Pattern> {
    if index >= self.patterns.len() || self.patterns.len() == 1 {
        return None;
    }
    self.playlist.clips.retain(|clip| clip.pattern != index);
    for clip in &mut self.playlist.clips {
        if clip.pattern > index {
            clip.pattern -= 1;
        }
    }
    Some(self.patterns.remove(index))
}
```

- `retain` conserva los elementos para los que el closure devuelve `true` y descarta los demás, en una sola pasada.
- `for clip in &mut self.playlist.clips` da una referencia mutable a cada clip por turno, así que `clip.pattern -= 1` cambia el clip de la lista.
- El método devuelve el patrón borrado, envuelto en `Some`. Quien lo llama puede ignorarlo, guardarlo, o comprobar que la eliminación llegó a ocurrir.

La prueba `removing_a_pattern_renumbers_its_followers` comprueba toda la historia: tras borrar «Ritmo», los cinco clips de Ritmo desaparecen, «Acordes» es el patrón 0, y el fill sigue sonando en el compás 8.

¿Por qué usar posiciones, en lugar de dar a cada patrón un id permanente? Los ids harían más simple la eliminación (nada que renumerar) pero todo lo demás más difícil: cada búsqueda tendría que buscar un id en lugar de indexar un `Vec`. Para una lista de como máximo 999 patrones (normalmente muchos menos), las posiciones más un método cuidadoso son la opción más simple. Es el tipo de intercambio sin una única respuesta correcta; uno de los ejercicios de esta parte te pide que pienses en ello.

Repartir `&mut Pattern` con seguridad

La app necesita cambiar notas de un patrón. `Project` podría copiar cada método de `Pattern` (`toggle`, `add_note`, `replace_note`…) como método propio. En su lugar reparte el patrón mismo:

```
pub fn pattern_mut(&mut self, index: usize) -> Option<&mut Pattern> {
    self.patterns.get_mut(index)
}
```

¿Es eso seguro? Con un `&mut Pattern`, ¿podría la app añadir una fila y romper la regla de «una fila por canal»? No, por cómo están declarados los métodos de filas en `pattern.rs`:

```
pub(crate) fn push_row(&mut self) { ... }
pub(crate) fn remove_row(&mut self, channel: usize) { ... }
```

`pub(crate)` significa «público solo dentro de este crate». `project.rs` puede llamarlos, porque está en `lineup-core`; `lineup-app` no puede, ni siquiera con un `&mut Pattern` en las manos. Cada método que `lineup-app` *sí puede* llamar mantiene el número de filas tal como está. La privacidad en Rust va de módulos y crates, no de quién tiene una referencia, y eso permite a un tipo dar una referencia a otro sin regalar sus reglas.

En el lado de la app, cada edición de un patrón pasa por un pequeño ayudante:

```
fn edit_pattern(&mut self, label: &'static str, change: impl FnOnce(&mut
Pattern)) {
    let index = self.pattern;
    self.edit(label, |project| {
        if let Some(pattern) = project.pattern_mut(index) {
            change(pattern);
        }
    });
}

// For example:
RackAction::ToggleStep { channel, step } => {
    self.edit_pattern("change steps", |pattern| {
        pattern.toggle(channel, step);
    });
}
```

- `impl FnOnce(&mut Pattern)` acepta cualquier closure que reciba un patrón. `FnOnce` es el más permisivo de los tres traits de closures: el closure solo tiene que poder llamarse una vez, así que puede incluso mover valores que capturó.

- `self.edit(label, ...)` es la única función por la que pasa cada cambio del proyecto. Guarda el cambio para deshacerlo y le dice al hilo de audio qué cambió (Deshacer y rehacer, Avisar al hilo de audio de lo que cambió). `label` es lo que dirá **Edición > Deshacer**.

Nombres, e `impl Into<String>`

Los patrones tienen nombre. El constructor recibe el nombre primero:

```
pub fn new(name: impl Into<String>, channels: usize, length: usize) ->
Self {
    Pattern {
        name: name.into(),
        color: 0,
        time: None,
        player: None,
        length: length.clamp(1, MAX_STEPS),
        rows: vec![Vec::new(); channels],
    }
}
```

`impl Into<String>` acepta cualquier cosa que se pueda convertir en un `String`: `Pattern::new("Beat", 5, 16)` con un `&str`, o `Pattern::new(format!("Pattern {n}"), 5, 16)` con un `String`, sin que quien llama escriba `.to_string()`. Dentro, `name.into()` hace la conversión (para un `String`, simplemente lo mueve).

`name` es un campo público mientras que `length` y `rows` son privados. Mezclar está bien en Rust: `name` no tiene ninguna regla que mantener, así que no hay razón para esconderlo detrás de métodos. `color`, `time` y `player` dicen de qué color se dibuja el patrón, en qué compás se escribió, y cuál de los músicos de la banda lo escribió, si lo escribió alguno.

Clonar un patrón usa el derive `Clone` de `Pattern`:

```
pub fn clone_pattern(&mut self, index: usize) -> Option<usize> {
    let mut copy = self.patterns.get(index)?.clone();
    copy.name = format!("{} copy", copy.name);
    // A copy is yours, not the band's.
    copy.player = None;
    self.push_pattern(copy)
}
```

`.clone()` copia el patrón entero, cada fila y cada nota, así que la copia y el original nunca comparten nada; la prueba `cloned_patterns_are_independent` comprueba

que cambiar uno deja al otro en paz. El `?` después de `get(index)` devuelve `None` de la función inmediatamente si no existe tal patrón.

Nunca dos notas en el mismo lugar

`add_note` se niega a añadir una nota en el mismo lugar que otra. `replace_note`, usado al arrastrar, también debe negarse, o una nota podría arrastrarse exactamente encima de otra, así que comprueba cada nota excepto la que se está reemplazando:

```
let duplicate = row
    .iter()
    .enumerate()
    .any(|(other_index, other)| other_index != index &&
other.same_place(&note));
```

Así que «no hay dos notas de la misma altura que empiecen en el mismo momento» es cierto para cada patrón, y cargar un archivo puede insistir en ello. `same_place` compara la altura, el paso y el nudge, así que dos notas que empiezan a unos pocos ticks de distancia siguen siendo dos notas.

¿Cuánto dura la canción?

La canción termina donde termina su último clip, redondeado hacia arriba a un compás entero:

```
pub fn song_length(&self) -> usize {
    let clips = self
        .playlist
        .clips
        .iter()
        .map(|clip| clip.start + self.clip_length(clip));
    let sections = self.playlist.sections.iter().map(Section::end);
    let end = clips.chain(sections).max().unwrap_or(0);
    self.meter().round_up_to_bar(end)
}
```

`max` devuelve un `Option`, porque una lista vacía no tiene máximo; `unwrap_or(0)` convierte «no hay clips» en una duración de 0. `chain` une los dos iteradores, porque una canción puede terminar con un clip o con una sección que no tiene ningún clip justo al final. `self.meter()` conoce el compás, y su `round_up_to_bar` es una división que redondea hacia arriba: en 4/4, 17 pasos son `17.div_ceil(16) = 2` compases.

¿Dónde está un patrón mientras suena la canción?

En modo canción, el hilo de audio informa de un paso de la *canción*. El rack de canales y el piano roll muestran un patrón, así que necesitan saber qué paso de *ese* patrón está sonando, si es que hay alguno:

```
pub fn pattern_step_in_song(&self, pattern: usize, song_step: usize) ->
Option<usize> {
    self.playlist
        .audible_clips()
        .filter(|clip| clip.pattern == pattern)
        .find_map(|clip| self.step_in_clip(&clip, song_step))
}

/// The step of its pattern the clip plays on song step `step`.
fn step_in_clip(&self, clip: &Clip, step: usize) -> Option<usize> {
    let pattern_length = self.patterns.get(clip.pattern).map_or(0,
Pattern::length);
    let end = clip.start + pattern_length * clip.repeats;
    (clip.start..end)
        .contains(&step)
        .then(|| (step - clip.start) % pattern_length)
}
```

Léelo como una frase: de los clips que se pueden oír, toma los de este patrón, encuentra el primero que cubra este paso de la canción y dime cuánto se adentra en el clip ese paso. El `% pattern_length` es lo que hace que un clip repetido vuelva a empezar su patrón en cada vuelta: un clip de 16 pasos que se repite en el paso 16 de la canción está allí en el paso 0 del patrón. `audible_clips` se salta los clips de pistas silenciadas:

```
pub fn audible_clips(&self) -> impl Iterator<Item = Clip> + '_ {
    self.clips
        .iter()
        .copied()
        .filter(|clip| self.tracks.get(clip.track).is_some_and(|track|
!track.muted))
}
```

`.copied()` convierte un iterador de `&Clip` en uno de `Clip`. Funciona porque `Clip` es `Copy` (tres números), y significa que quien llama recibe valores simples en lugar de referencias a la lista de reproducción.

La canción de partida

`Project::demo` construye una canción de ocho compases, que tocan muchas pruebas, con los mismos métodos que usa la app:

```
for bar in [0, 2, 4, 6] {
    project.add_clip(Clip::new(chords, chord_track, bar * 16));
}
for bar in 2..7 {
    project.add_clip(Clip::new(beat, drum_track, bar * 16));
}
project.add_clip(Clip::new(fill, drum_track, 7 * 16));
```

`2..7` es un rango de 2 hasta, pero sin incluir, el 7: compases 3 a 7 contando desde 1, como hace la regla. El fill ocupa el lugar del beat en el compás 8. **Archivo > Nuevo ritmo** empieza en cambio desde `Project::template`: la batería y un Lineup Osc, sin notas y con una pista. **Archivo > Nuevo** empieza desde `Project::with_band`, una hoja de acordes vacía con una banda (Cómo toca la banda una hoja de acordes).

Secciones, repeticiones y bucles

Una canción tiene más forma que sus clips: una introducción, una estrofa, un estribillo que se oye dos veces, un ritmo que se repite durante ocho compases. La lista de reproducción guarda tres cosas más para eso:

- **Secciones**: tramos con nombre de la canción (Introducción, Estrofa, Estribillo…) en una franja sobre los números de compás. Una sección se puede tocar varias veces seguidas: se dibuja una vez y se marca «×2».
- **Repeticiones de clip**: un clip puede tocar su patrón una y otra vez, como al estirar un clip de patrón en otros estudios basados en patrones.
- **Una región de bucle**: el modo canción puede repetir solo una parte de la canción.

Las secciones no son dueñas de clips. Una sección es solo un nombre y un rango de pasos: los clips que quedan debajo forman parte de ella. Eliminar una sección deja los clips como están.

Dos tipos de paso

Aquí está la única idea que hace sencillo todo lo demás. Una vez que una sección puede repetirse, la lista de reproducción tal como se *dibuja* y la canción tal como se

toca ya no miden lo mismo:

```
timeline:   | Intro | Chorus ×2 | Outro |
canción:    | Intro | Chorus | Chorus | Outro |
```

Así que el código habla de dos tipos de paso:

- un **paso de timeline** es un lugar de la lista de reproducción, tal como se dibuja;
- un **paso de canción** es un momento de la canción, tal como se oye.

`Project::song_length` es la longitud del timeline. `Arrangement` es la canción tal como se oye:

```rust
pub struct Arrangement {
    /// The pieces of patterns to play, in song steps.
    pub plays: Vec<Play>,
    /// The stretches of the timeline, in the order they play.
    pub passes: Vec<Pass>,
    /// How long the song is, in song steps.
    pub length: usize,
}

pub struct Play {
    pub pattern: usize,
    pub start: usize,  // the song step it starts on
    pub skip: usize,   // how far into the clip it starts
    pub length: usize, // how many steps it lasts
}
```

Un `Pass` dice «los pasos de timeline 32 a 63 suenan en los pasos de canción 64 a 95, y esta es la ronda 2 de 2 de la sección 1». Un `Play` dice «toca el patrón 3 desde el paso de canción 64 durante 32 pasos, empezando 0 pasos dentro de él». El patrón se repite en bucle mientras dura el `Play`, así que un clip repetido tres veces es un `Play` de tres patrones de largo. `skip` no es cero cuando una sección o la región de bucle empieza a mitad de un clip.

`Project::arrangement()` lo construye para la canción entera (las exportaciones lo usan), y `Project::playback_arrangement()` para lo que toca el modo canción: la región de bucle si la hay, la canción entera si no.

Construir el arreglo

`arrange` en `arrangement.rs` corta el timeline en trozos en los bordes de las secciones. Las secciones suenan `repeats` veces; los huecos entre ellas suenan una vez:

```rust
fn pieces(project: &Project, from: usize, to: usize)
    -> Vec<(usize, usize, Option<usize>, usize)>
{
    let mut pieces = Vec::new();
    let mut cursor = from;
    for (index, section) in
project.playlist().sections().iter().enumerate() {
        if section.end() <= cursor || section.start >= to {
            continue;
        }
        if section.start > cursor {
            pieces.push((cursor, section.start, None, 1)); // a gap
        }
        let start = section.start.max(cursor);
        let end = section.end().min(to);
        pieces.push((start, end, Some(index), section.repeats));
        cursor = end;
    }
    if cursor < to {
        pieces.push((cursor, to, None, 1));
    }
    pieces
}
```

Esto solo funciona porque las secciones se mantienen **ordenadas y sin solaparse nunca**. Eso es una regla, así que, como con los clips, la lista es privada y solo los métodos de `Project` la cambian. `add_section` y `set_section` comprueban la regla y rechazan una sección que la rompería; `insert_section` mantiene el orden con `partition_point`, una búsqueda binaria que devuelve el primer lugar donde la condición deja de ser cierta:

```rust
let index = sections.partition_point(|other| other.start < section.start);
sections.insert(index, section);
```

Luego, para cada trozo y cada pasada, todo clip que se solape con el trozo añade un `Play` para la parte de él que está dentro del trozo. `max` y `min` encajan el clip en el trozo:

```rust
let begin = clip.start.max(piece_from);
let end = clip_end.min(piece_to);
if begin < end {
    arrangement.plays.push(Play {
        pattern: clip.pattern,
        start: start + (begin - piece_from),
        skip: begin - clip.start,
        length: end - begin,
    });
}
```

El hilo de audio no sabe nada de secciones

`Command::SetSong` lleva los `plays` del arreglo, y lo único que hace el hilo de audio con ellos es un resto:

```
pub fn pattern_step(&self, step: usize, pattern_length: usize) ->
Option<usize> {
    if pattern_length == 0 || !(self.start..self.end()).contains(&step) {
        return None;
    }
    Some((self.skip + step - self.start) % pattern_length)
}
```

`%` envuelve el paso alrededor del patrón, que es todo lo que significa «repetir». Todo lo de nombres, rondas y bucles se queda en `lineup-core`, donde puede probarse sin tarjeta de sonido. El hilo de audio sigue siendo pequeño, y no reserva memoria: el `Vec<Play>` se construye en el hilo de la UI, y el viejo se devuelve para liberarlo allí.

La exportación a WAV y la exportación a MIDI usan el mismo `Arrangement`, así que las tres coinciden en qué es la canción. El escritor de MIDI necesita la pregunta contraria: ¿en qué pasos de canción suena una nota del paso de patrón 2? Eso es `Play::song_steps_of`, que usa un iterador con `step_by`:

```
(first..self.length)
    .step_by(pattern_length.max(1))
    .map(move |offset| self.start + offset)
```

De vuelta de la canción a la lista de reproducción

El hilo de audio cuenta pasos de canción, así que el cabezal de reproducción que comunica es un paso de canción. Para dibujarlo, la app lo convierte de nuevo en un paso de timeline con los passes del arreglo:

```
fn song_position(&self) -> Option<(usize, Pass)> {
    if self.mode != PlayMode::Song {
        return None;
    }
    let step = self.playhead()?;
    let song = self.project.playback_arrangement();
    let pass = *song.pass_at(step)?;
    Some((pass.from + (step - pass.start), pass))
}
```

El `Pass` viaja con él hasta la lista de reproducción, que es como una sección sabe que debe mostrar «▶ 2 de 2». El rack de canales, el piano roll y las teclas de armonía reciben el paso de timeline, así que muestran el paso correcto del clip correcto también durante una repetición.

Guardar: formato 4

`Clip` y `Section` tienen las dos un campo `repeats`, y la mayoría de las veces es 1. serde puede dejar un campo fuera del archivo cuando tiene su valor habitual, y volver a rellenarlo al leer un archivo que no lo tiene:

```
#[serde(default = "once", skip_serializing_if = "is_once")]
pub repeats: usize,
```

`default` nombra una función que devuelve el valor que usar, y `skip_serializing_if` una función que dice cuándo dejar el campo fuera. Las listas del archivo (`sections`, `loop_region`) usan el `#[serde(default)]` normal con `Vec::is_empty` y `Option::is_none`. Así que un proyecto que no usa nada de esto se escribe byte a byte como antes, y un Ferrous Lineup más antiguo lo abre.

Una versión más antigua también abriría un proyecto que *sí* usa repeticiones: se salta los campos que no conoce. Pero tocaría cada repetición una vez, que es una canción que suena mal sin una palabra de aviso. Así que `format_for` devuelve **4** para cualquier proyecto con secciones, una región de bucle o un clip repetido, y las versiones antiguas lo rechazan con un mensaje claro (Guardar).

Dibujarlo: un módulo dentro de un módulo

Las franjas se dibujan en un archivo propio. Un módulo de archivo puede tener hijos: `mod sections;` dentro de `playlist.rs` carga `playlist/sections.rs`. Un módulo hijo puede ver los elementos privados de su padre, así que `sections.rs` usa `Layout`, `Snap` y `PlaylistAction` de `playlist.rs` con `use super::{Layout, PlaylistAction, Snap};` sin hacerlos públicos para el resto de la app.

Cada sección es su propia zona clicable, hecha con `ui.interact` sobre un rectángulo que elegimos nosotros, que devuelve un `Response` como cualquier widget:

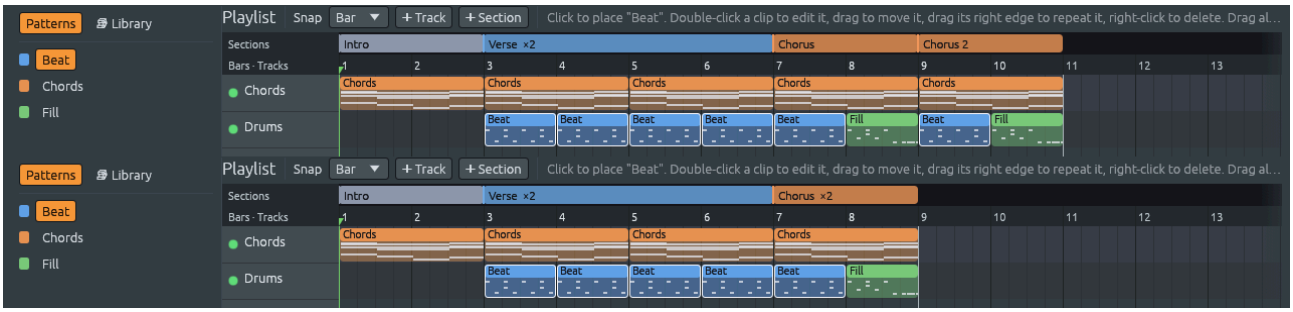
```
let response = ui.interact(rect, ui.scope_id().with(("section", index)),
                           Sense::click_and_drag());
```

El id debe ser distinto para cada sección y el mismo de un frame a otro, así que se construye a partir del número de la sección. El menú es un `egui::Popup` atado a esa respuesta, que se abre con un clic y solo se cierra con un clic fuera, porque contiene un campo de texto en el que escribes.

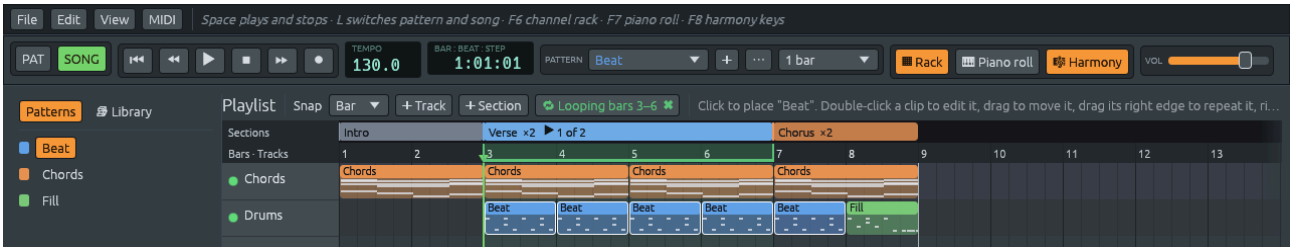
Arrastrar una sección no cambia el proyecto en cada frame, como hace arrastrar un clip. Solo se dibuja un contorno, y el proyecto cambia una vez, cuando el botón sube. Una sección se mueve dentro de una lista ordenada: si el movimiento de un frame la saltara por encima de una vecina, su número cambiaría a mitad del arrastre, y el siguiente frame movería la sección equivocada.

Escribir las repeticiones

Escribir las repeticiones en el menú de una sección convierte «Estribillo ×3» en Estribillo, Estribillo 2 y Estribillo 3, uno detrás de otro, cada uno tocado una vez, para que puedas cambiar el tercer estribillo sin cambiar los otros. La canción suena igual antes y después; una prueba lo comprueba comparando los dos arreglos.



Comparte su código con **Duplicar**: `make_room` mueve todo lo que hay después de la sección más tarde, y luego los clips que empiezan dentro de la sección se copian en el hueco. Una cosa que hay que vigilar: los clips se copian enteros. El relleno de la imagen empieza dentro del estribillo y sigue más allá de su final, así que el original y su copia se solapan. Ferrous Lineup no puede cortar un clip en dos (un clip no tiene «empieza a esta distancia dentro del patrón»); uno de los ejercicios de esta parte lo añade.



Decirle al hilo de audio lo que ha cambiado

Archivos: `crates/lineup-core/src/diff.rs`, `crates/lineup-app/src/app/edit.rs`, `crates/lineup-audio/src/engine.rs` y `crates/lineup-audio/src/sample.rs`

Una sola entrada para cada cambio

El hilo de audio guarda su propia copia de todo lo que reproduce: el instrumento, el volumen, el paneo y las notas de cada canal, las longitudes de los patrones, la canción (ver Dos hilos). La ventana debe mantener esa copia al día, y un cambio puede tocar cualquier cosa. Un clic en un paso cambia una lista de notas. Borrar un patrón renumera todos los clips que vienen después. Escribir un acorde hace que la banda escriba patrones nuevos. Deshacer sustituye todo el proyecto por uno más antiguo, que puede diferir en cualquier cosa.

Hay tres formas de contarle un cambio al hilo de audio:

1. **Enviar el proyecto entero otra vez**: todos los patrones y canales, como al abrir un archivo. Sencillo, pero cada instrumento se construiría de nuevo, cortando toda nota que esté sonando, y un archivo de muestra se leería otra vez por un deshacer de un solo paso.
2. **Escribir, para cada tipo de edición, los comandos que necesita**, y para deshacer, los comandos que la invierten. Es un par escrito a mano para cada edición, y cada par es una ocasión de equivocarse al invertir (el capítulo siguiente cuenta más sobre esta elección).
3. **Comparar el proyecto antes y después**, y enviar solo lo que difiere.

Ferrous Lineup hace lo tercero, para *todos* los cambios, no solo para deshacer. Ninguna edición envía sus propios comandos. Todo cambio pasa por un único método, `edit`:

```
pub(super) fn edit<R>(
    &mut self,
    label: &'static str,
    change: impl FnOnce(&mut Project) -> R,
) -> R {
    if self.history.is_open() {
        // A drag or a typing session is under way, and its first change
        // made the undo step. Keep changing the project without copying
        // the whole of it every frame; the audio catches up steadily
        // from the last state it was given.
        let (patterns, channels) =
            (self.project.patterns().len(),
self.project.channels().len());
        let result = change(&mut self.project);
        self.project.follow_chord_sheet();
        self.file.project_edited();
        self.recovery.project_changed();
        self.step_unsynced = true;
        if self.project.patterns().len() != patterns
            || self.project.channels().len() != channels
        {
            // A pattern or channel came or went: the views and the
            // audio thread's rows have to move now, not in 50 ms.
            self.sync_step();
        } else {
            self.sync_step_if_due();
        }
        return result;
    }
    // A step left open by a direct history call: send it before the
    // audio's state is assumed to be this project.
    self.finish_step_sync();
    let before = self.project.clone();
    let result = change(&mut self.project);
    // The band plays what the chord sheet says now, in the same step.
    self.project.follow_chord_sheet();
    if self.project != before {
        self.sync_audio(&before);
        self.synced = self.project.clone();
        self.step_unsynced = false;
        self.last_sync = Instant::now();
        self.file.project_reverted();
        self.recovery.project_changed();
        // `before` is moved into the history here, after its last use.
        self.history.record(label, before);
    }
    result
}
```

Lee primero la segunda mitad, el caso normal. `edit` guarda una copia del proyecto, deja que el closure lo cambie y, si algo cambió, hace cuatro cosas en un solo sitio:

- `sync_audio(&before)` le dice al hilo de audio qué difiere. El resto de este capítulo trata de eso.
- `follow_chord_sheet` deja que la banda vuelva a escribir sus patrones si cambiaron la hoja de acordes, la tonalidad o los compases, como parte del mismo paso (ver Cómo toca la banda una hoja de acordes).
- `file` y `recovery` se enteran de que el proyecto cambió: la barra de título recibe su asterisco de cambios sin guardar, y se escribirá una copia de recuperación (ver Guardar). `project_reverted` vuelve a calcular el asterisco desde cero, porque un cambio hecho de una vez puede dejar exactamente lo que estaba guardado, como un paso que se activa y se vuelve a desactivar.
- `history.record` guarda `before` como el paso de deshacer llamado `label` (ver Deshacer y rehacer).

Tener un único camino compensa dos veces. Solo hay un fragmento de código que convierte cambios del proyecto en comandos, así que se puede probar a fondo por separado. Y se usa en cada clic, así que si tuviera un error lo oirías mientras editas, no solo en el raro momento en que deshaces algo inusual.

Un arrastre es un paso, no uno por fotograma

La primera mitad, la rama `is_open`, es el detalle de rendimiento importante. Un arrastre cambia notas en cada fotograma, sesenta veces por segundo, pero solo su primer fotograma necesita la copia para deshacer y la comparación completa. Mientras el paso está abierto, el proyecto se cambia directamente, y `synced` guarda el último estado que recibió el hilo de audio. `sync_step_if_due` compara el proyecto con ese estado y envía las diferencias como mucho cada 50 ms:

```
/// How often, at most, a drag's changes reach the audio thread. The
/// window still draws every frame; only the sound waits this long, and
/// the whole project is copied only when this comes round.
const SYNC_INTERVAL: Duration = Duration::from_millis(50);

fn sync_step_if_due(&mut self) {
    if self.last_sync.elapsed() >= SYNC_INTERVAL {
        self.sync_step();
    }
}

fn sync_step(&mut self) {
    let before = std::mem::replace(&mut self.synced, Project::empty());
    self.sync_audio(&before);
    self.synced = self.project.clone();
    self.step_unsynced = false;
    self.last_sync = Instant::now();
}
```

`std::mem::replace` saca el `synced` antiguo de la estructura y deja un proyecto vacío en su lugar, así que `sync_audio` puede tomarlo prestado como `before` sin copiarlo (no podría tomar prestado `self.synced` mientras también toma prestado `self` de forma mutable). Cuando un patrón o un canal aparece o desaparece en mitad de un paso, esperar no basta: las vistas cuentan filas que el hilo de audio aún no tiene, así que `sync_step` se ejecuta enseguida.

Al soltar el ratón, `finish_step_sync` envía los últimos cambios que faltan, y el paso se cierra. Un arrastre es un paso de deshacer, una copia y un puñado de mensajes.

Deshacer y rehacer sustituyen el proyecto entero, así que siempre hacen la comparación completa:

```
pub(super) fn undo(&mut self) {
    self.finish_step_sync();
    let before = self.project.clone();
    if self.history.undo(&mut self.project).is_some() {
        self.sync_audio(&before);
        self.synced = self.project.clone();
        self.step_unsynced = false;
        self.last_sync = Instant::now();
        self.file.project_reverted();
        self.recovery.project_changed();
        self.restore_song_library();
        self.keep_place_in_song(&before);
    }
}
```

Las dos últimas líneas son para la ventana, no para el sonido: la biblioteca muestra el estilo y el ritmo que toca esta versión del proyecto, y si deshacer cambió el compás, la reproducción sigue en el mismo compás de la canción.

Leer `edit`

`edit` recibe un **closure** que cambia el proyecto, y el closure recibe el proyecto como argumento:

```
self.edit("move clip", |project| project.move_clip(index, track, start));
```

¿Por qué no dejar que el closure alcance `self.project` por sí mismo? Porque `edit` ya tiene `&mut self`, y un closure que tomara prestado `self` también sería un segundo préstamo del mismo valor mientras el primero sigue en uso. El borrow checker lo rechaza. Entregarle al closure `&mut self.project` como argumento le

presta exactamente la parte que necesita, durante exactamente lo que dura la llamada.

El tipo del closure es `impl FnOnce(&mut Project) -> R`. `FnOnce` es el trait de los closures que se llaman exactamente una vez: deja que el closure mueva un valor al proyecto, como un `channel` nuevo, porque nadie lo llamará una segunda vez. (Los closures `Fn` se pueden llamar muchas veces y solo leen lo que capturan; los `FnMut` pueden cambiarlo.)

`R` es lo que devuelva el closure, y `edit` se lo devuelve a quien lo llama. Muchos métodos de `Project` devuelven algo: `add_clip` devuelve el índice del clip nuevo como `Option<usize>`, `remove_pattern` el patrón que quitó, `insert_bars` si había sitio. Como `R` puede ser cualquier tipo, un closure puede ser simplemente la llamada al método, `|project| project.add_clip(clip)`, y quien llama recibe la respuesta. Si `edit` pidiera `impl FnOnce(&mut Project)`, que debe devolver `()`, cada una de esas necesitaría llaves y un punto y coma para tirar el valor.

`edit_pattern` es un pequeño envoltorio para el caso más común, un cambio en el patrón seleccionado:

```
fn edit_pattern(&mut self, label: &'static str, change: impl FnOnce(&mut
Pattern)) {
    let index = self.pattern;
    self.edit(label, |project| {
        if let Some(pattern) = project.pattern_mut(index) {
            change(pattern);
        }
    });
}
```

`self.project != before` compara los dos proyectos por completo: el `PartialEq` derivado compara cada campo, cada lista, cada nota. Si nada cambió, como un clip arrastrado de vuelta a la misma casilla o una nota borrada donde no había ninguna, no se envía nada y no se registra ningún paso de deshacer.

`diff`: la lista de cambios

La comparación vive en `lineup-core`, que conoce los proyectos pero nada del hilo de audio. Describe las diferencias con las propias palabras del proyecto:

```rust
pub enum Change {
    RemovePattern(usize),
    InsertPattern(usize),
    RemoveChannel(usize),
    InsertChannel(usize),
    Instrument(usize),
    Volume(usize),
    Muted(usize),
    Pan(usize),
    ReverbSend(usize),
    Effects(usize),
    PatternLength(usize),
    Notes { pattern: usize, channel: usize },
    Tempo,
    LooseTiming,
    Mixer,
    Song,
}

pub fn diff(before: &Project, after: &Project) -> Vec<Change>
```

Los cambios son pasos que hay que seguir **en orden**, y cada número es una posición en el momento en que se aplica su cambio. Después de `RemovePattern(0)`, el patrón que solía ser el número 1 es el número 0, y cualquier cambio posterior lo llama 0. Los valores (un volumen, una lista de notas) no están en el `Change`: se leen del proyecto más nuevo cuando el cambio se convierte en un comando. Eso mantiene `Change` pequeño y `Copy`.

Solo se compara lo que produce sonido. Los nombres de patrón, de canal, de pista y de sección y la tonalidad no dan ningún cambio, porque el hilo de audio nunca los ve. Silenciar una pista sí da uno: saca los clips de esa pista de la canción, así que los clips que reproduce el hilo de audio cambian.

El mezclador también se oye. Mover un canal entre los altavoces da `Pan`, cambiar cuánto va de él a la reverberación da `ReverbSend`, cambiar su ecualizador, su compresor o su delay da `Effects`, y cambiar el volumen maestro o la propia reverberación da `Mixer`. Poner un canal en Solo es un cambio de mezclador *y* cambia qué canales se oyen, así que `diff` también informa de `Muted` para cada canal cuyo silencio efectivo cambió. La Soltura da `LooseTiming`: decide cuánto puede alejarse de la cuadrícula el comienzo de una nota.

Alinear dos listas

La parte difícil son las listas. Digamos que los patrones eran `[Beat, Chords, Fill]` y ahora son `[Beat, Fill]`. Comparar elemento a elemento diría que el patrón 1

cambió de Chords a Fill y que el patrón 2 desapareció, lo cual es cierto pero torpe: enviaría otra vez todas las notas de Fill. Lo que pasó de verdad es que «se quitó el patrón 1».

`Alignment` lo encuentra. Cuenta cuántos elementos son iguales al principio de ambas listas, y cuántos al final. Lo que queda en medio se quitó de la lista antigua y se insertó en la nueva:

```
  antes:    Beat │ Chords │ Fill       start = 1 (Beat es el mismo)
  después:  Beat │        │ Fill       end   = 1 (Fill es el mismo)
                                       eliminados = 3 - 1 - 1 = 1
                                       insertados = 2 - 1 - 1 = 0
```

```
fn new<T>(before: &[T], after: &[T], same: impl Fn(&T, &T) -> bool) ->
Self {
    if before.len() == after.len() {
        return Alignment { start: 0, removed: 0, inserted: 0 };
    }
    // The same at the start...
    let start = before
        .iter()
        .zip(after)
        .take_while(|(old, new)| same(old, new))
        .count();
    // ...and the same at the end, without counting an item twice.
    let room = before.len().min(after.len()) - start;
    let end = before[start..]
        .iter()
        .rev()
        .zip(after[start..].iter().rev())
        .take_while(|(old, new)| same(old, new))
        .count()
        .min(room);
    Alignment {
        start,
        removed: before.len() - start - end,
        inserted: after.len() - start - end,
    }
}
```

La cadena de iteradores se lee bien cuando conoces las piezas: `zip` recorre dos listas lado a lado, dando pares; `take_while` conserva pares mientras son iguales y se detiene en el primero que no lo es; `count` dice cuántos se conservaron. `rev` recorre las listas hacia atrás, para el final.

`.min(room)` es fácil de pasar por alto e importante. Con `[A, A]` convirtiéndose en `[A]`, el principio encuentra un elemento igual, y el final también, *el mismo* `A`. Las dos cuentas juntas dirían que se conservan dos elementos de una lista de uno, y

`after.len() - start - end` sería `1 - 1 - 1`, que para un `usize` no es -1 sino un panic («attempt to subtract with overflow»). Limitar el final a lo que queda después del principio mantiene honesta la aritmética.

Cuando ambas listas miden lo mismo, no se añadió ni se quitó nada, así que los elementos se comparan posición por posición. Eso cubre las ediciones más comunes: un clic en un paso cambia un patrón, en su sitio.

`Alignment` funciona para cualquier `T`, porque quien llama dice qué significa «el mismo» con el closure `same`. Para los canales es la igualdad completa: `|old, new| old == new`. Para los patrones **no** lo es, y esa es la parte sutil. Las filas de un patrón cambian cuando los canales van y vienen: cuando entra la banda, cada patrón recibe una fila para cada uno de sus canales, así que comparar patrones enteros diría que todos los patrones cambiaron. El alineado enviaría entonces otra vez las notas de cada patrón, y la aplicación seleccionaría un patrón que solo parece nuevo.

Así que los patrones se alinean por las cosas que los identifican, no por su contenido:

```
/// Whether two patterns are the same pattern, perhaps with other notes.
fn same_pattern(old: &Pattern, new: &Pattern) -> bool {
    old.name == new.name && old.color == new.color && old.player ==
new.player
}
```

Las notas y la longitud son lo que el alineador deja cambiar en su sitio. El nombre, el color y el intérprete (`player`) dicen *qué* patrón es; las notas que difieren se envían después como cambios `Notes` y `PatternLength` de todos modos.

Punteros fn: las variantes de un enum son funciones

Una vez alineadas las dos listas, los cambios son: quitar el medio de la lista antigua y luego insertar el medio de la nueva.

```
fn changes(
    self,
    remove: fn(usize) -> Change,
    insert: fn(usize) -> Change,
) -> impl Iterator<Item = Change> {
    let removals = (self.start..self.start +
self.removed).rev().map(remove);
    let insertions = (self.start..self.start + self.inserted).map(insert);
    removals.chain(insertions)
}
```

y se llama así:

```
patterns.changes(Change::RemovePattern, Change::InsertPattern)
```

`Change::RemovePattern` no es aquí un valor, es una **función**. Toda variante de un enum que guarda datos entre paréntesis es también una función que la construye: `Change::RemovePattern` tiene el tipo `fn(usize) -> Change`. El `fn(usize) -> Change` en minúscula es un tipo de **puntero a función**: la dirección de una función normal, sin variables capturadas. Un closure que no captura nada también puede serlo. `map(remove)` lo llama para cada número, construyendo `RemovePattern(2)`, `RemovePattern(1)` y así sucesivamente.

Las eliminaciones van desde el **último** hacia atrás. Quitar primero el patrón 2 deja los patrones 0 y 1 donde estaban, así que `RemovePattern(1)` sigue significando el patrón correcto. Hacia delante, quitar primero el patrón 1 bajaría el patrón 2 al 1, y la segunda eliminación daría con el equivocado. Las inserciones van hacia delante por la misma razón: una vez llena la posición 1, la posición 2 está justo después.

`chain` une dos iteradores en uno: primero todos los de `removals`, luego todos los de `insertions`. La función devuelve `impl Iterator<Item = Change>`, «algún iterador de cambios», sin nombrar su largo tipo real (`Chain<Map<Rev<Range<usize>>, fn(usize) -> Change>, ...>`).

El orden de todo

```
pub fn diff(before: &Project, after: &Project) -> Vec<Change> {
    let patterns = Alignment::new(before.patterns(), after.patterns(),
same_pattern);
    let channels = Alignment::new(before.channels(), after.channels(),
|old, new| old == new);

    let mut changes: Vec<Change> = patterns
        .changes(Change::RemovePattern, Change::InsertPattern)
        .chain(channels.changes(Change::RemoveChannel,
Change::InsertChannel))
        .collect();
    ...
```

Primero los patrones, luego los canales, y luego todo lo que hay dentro. El orden importa por lo que hace el hilo de audio con cada cambio:

- Un **patrón** insertado llega al hilo de audio sin notas en ningún canal; sus notas vienen después como cambios `Notes`.
- Un **canal** insertado llega completo: su instrumento, su volumen y sus notas en *cada* patrón. Así que los patrones ya deben estar correctos cuando llegue, o

traería notas para un número equivocado de patrones.

Luego, para cada canal que ya existía (`old_index` dice dónde estaba, o `None` si es nuevo), se comparan el instrumento y la mezcla:

```
for (index, channel) in after.channels().iter().enumerate() {
    let Some(old_index) = channels.old_index(index) else {
        continue; // a new channel arrives complete
    };
    let old = &before.channels()[old_index];
    if old.kind != channel.kind
        || old.program != channel.program
        || old.sent_on != channel.sent_on
    {
        changes.push(Change::Instrument(index));
    }
    if old.volume != channel.volume {
        changes.push(Change::Volume(index));
    }
    if old.pan != channel.pan {
        changes.push(Change::Pan(index));
    }
    if old.reverb_send != channel.reverb_send {
        changes.push(Change::ReverbSend(index));
    }
    if old.effects != channel.effects {
        changes.push(Change::Effects(index));
    }
    if before.channel_muted(old_index) != after.channel_muted(index) {
        changes.push(Change::Muted(index));
    }
}
```

El instrumento se construye a partir de tres campos: el tipo (batería, archivo de muestra, Lineup Osc, MIDI Out), el programa General MIDI y `sent_on`, el puerto MIDI al que se envía una batería en lugar de sonar aquí. `channel_muted` no es simplemente el botón de silencio del canal: también tiene en cuenta el Solo del mezclador. Por eso un cambio de Solo aparece como un cambio `Muted` en los demás canales, además de un cambio `Mixer`.

`let ... else` o saca el valor del `Some`, o ejecuta el bloque `else`, que debe salir: aquí `continue` salta al canal siguiente.

Luego las notas, para cada patrón y cada canal antiguo:

```
for (index, pattern) in after.patterns().iter().enumerate() {
    let old_index = patterns.old_index(index);
    let old_pattern = old_index.map(|old| &before.patterns()[old]);
    if let Some(old) = old_pattern
        && old.length() != pattern.length()
    {
        changes.push(Change::PatternLength(index));
    }
    for channel in 0..after.channels().len() {
        let Some(old_channel) = channels.old_index(channel) else {
            continue; // a new channel brought its notes with it
        };
        // The notes as they are played, with the song's swing: a
        // new amount, or a pattern now in a time signature that does
        // not swing, plays other notes. A new pattern starts with no
        // notes in the audio thread.
        let old_notes = match old_index {
            Some(old) => before.played_notes(old, old_channel),
            None => Cow::Borrowed(&[][..]),
        };
        if old_notes != after.played_notes(index, channel) {
            changes.push(Change::Notes { pattern: index, channel });
        }
    }
}
```

El `if let ... && ...` es una **let chain**: el bloque se ejecuta solo si el patrón es antiguo *y* su longitud cambió.

Las notas que se comparan son `played_notes`, no las escritas. El swing retrasa las notas a contratiempo sin cambiar lo escrito (ver Tempo, compás, swing y feel), y al hilo de audio hay que enviarle lo que se *oye*. Así que subir el swing no cambia ninguna nota escrita, pero da un cambio `Notes` por cada lista que mueve. `played_notes` devuelve un `Cow` («clone on write», copiar al escribir): un `Cow::Borrowed` cuando las notas escritas suenan tal cual, lo que no cuesta nada, y un `Cow::Owned` con una copia cambiada solo cuando el swing las mueve. Dos `Cow` se comparan por su contenido, sea cual sea su tipo.

`&[][..]` es un slice vacío: `[]` es un array vacío, y `[..]` lo toma entero como slice. Un patrón nuevo se compara con «sin notas», que es exactamente lo que el hilo de audio tiene para él, así que un cambio `Notes` se envía solo para los canales que tienen notas en él.

Por último vienen el tempo, la soltura, el mezclador y la canción. La canción se compara a través del arreglo que el modo canción reproduce de verdad:

```
let (old_song, new_song) = (before.playback_arrangement(),
after.playback_arrangement());
if old_song.plays != new_song.plays
    || old_song.length != new_song.length
    || before.played_tempos(&old_song) != after.played_tempos(&new_song)
{
    changes.push(Change::Song);
}
```

Un `Arrangement` son los clips con sus repeticiones y las repeticiones de las secciones ya desenrolladas, recortado a la región de bucle si la hay (ver El proyecto). Compararlo detecta un clip movido, un silencio, un número de repeticiones, un bucle cambiado y un marcador de tempo en un solo cambio `Song`. Los marcadores de tempo forman parte de la canción porque la longitud de un paso cambia a mitad de ella.

Cuando la historia es ambigua

Digamos que hay tres canales Lineup Osc con los mismos ajustes pero notas distintas, y borras el del medio. Comparando solo los *canales* (nombre, instrumento, volumen, silencio), la lista antigua es `[Osc, Osc, Osc]` y la nueva `[Osc, Osc]`, y el alineado no puede saber cuál se fue: encuentra dos iguales al principio y dice que se quitó el *último*.

No pasa nada. Las notas del segundo canal ahora difieren (tiene las que solían ser las notas del tercer canal), así que viene un cambio `Notes`, y el hilo de audio acaba con exactamente los canales y las notas correctos. El diff no necesita adivinar lo que hiciste de verdad. Necesita una lista de pasos que acabe en el mismo sitio, y corta donde pueda.

Probar con un modelo

¿Cómo se prueba una función como `diff`? Comprobar su salida exacta para cada caso sería frágil, y no demostraría que la salida es *correcta*. Lo que de verdad queremos saber es: *si el hilo de audio sigue estos cambios, ¿acaba con el proyecto nuevo?*

Así que las pruebas construyen un pequeño **modelo** de lo que sabe el hilo de audio: las longitudes de los patrones, el instrumento, la mezcla, los efectos y las notas por patrón de cada canal, el tempo y la canción. Es lo bastante sencillo como para fiarse de un vistazo:

```rust
struct ModelChannel {
    kind: ChannelKind,
    program: Option<u8>,
    volume: f32,
    muted: bool,
    pan: f32,
    reverb_send: f32,
    effects: crate::Effects,
    /// One list per pattern.
    notes: Vec<Vec<NoteEvent>>,
}

struct Model {
    lengths: Vec<usize>,
    channels: Vec<ModelChannel>,
    tempo: Tempo,
    loose_timing: u8,
    plays: Vec<Play>,
    song_length: usize,
    tempos: Vec<Tempo>,
}
```

`Model::apply` le hace al modelo lo que el hilo de audio hace con cada cambio, y `check` lo junta todo:

```rust
/// Diffs two projects, checks that following the changes turns the
/// first into the second, and returns the changes.
fn check(before: &Project, after: &Project) -> Vec<Change> {
    let changes = diff(before, after);
    let mut model = Model::of(before);
    for &change in &changes {
        model.apply(change, after);
    }
    assert_eq!(model, Model::of(after), "changes: {changes:?}");
    changes
}

/// Checks both directions, as undo and then redo would.
fn check_both_ways(before: &Project, after: &Project) -> Vec<Change> {
    check(after, before);
    check(before, after)
}
```

Cada prueba comprueba primero esta propiedad, y solo después, donde importa, los cambios exactos: que un clic en un paso da un único `Notes`, que los nombres y la tonalidad no dan nada, que deshacer un patrón borrado empieza con `InsertPattern(0)`, que el swing cambia las notas que suenan. La última prueba monta un lío a propósito, quitando un patrón, añadiendo un canal, activando un paso, cambiando un volumen y el tempo a la vez, y también compara un proyecto vacío con la canción de demostración, como haría abrir un archivo.

Este tipo de prueba se llama **basada en propiedades**: comprueba una regla que debe cumplirse para cualquier entrada, en lugar de una respuesta para una entrada. Uno de los ejercicios va más allá, con proyectos cambiados al azar.

El lado de la aplicación: `sync_audio`

`sync_audio` convierte cada `Change` en un comando, leyendo los valores del proyecto actual:

```
fn sync_audio(&mut self, before: &Project) {
    let sample_rate = self.sample_rate();
    let mut patterns_moved = false;
    let mut channels_changed = false;
    for change in diff(before, &self.project) {
        match change {
            Change::RemovePattern(index) => {
                self.send(Command::RemovePattern(index));
                // Keep the same pattern selected. If it was the one
                // removed, the next one takes its place.
                if index < self.pattern {
                    self.pattern -= 1;
                }
                patterns_moved = true;
            }
            Change::InsertPattern(index) => {
                let pattern = &self.project.patterns()[index];
                let (length, bands) = (pattern.length(),
pattern.player.is_some());
                self.send(Command::InsertPattern { index, length });
                if bands {
                    // A part the band wrote, as you typed a chord:
                    // the pattern you had stays selected.
                    if index <= self.pattern {
                        self.pattern += 1;
                    }
                } else {
                    // A new pattern (added, cloned, or brought back
                    // by undo) is the one you want to see, from its
                    // start.
                    self.pattern = index;
                    self.rewind_for_new_pattern();
                }
                patterns_moved = true;
            }
            Change::Notes { pattern, channel } => {
                let notes = self.project.played_notes(pattern,
channel).into_owned();
                self.send(Command::SetNotes { pattern, channel, notes });
            }
            ...
        }
    }
    ...
}
```

Las demás ramas son igual de directas. `InsertChannel` construye el instrumento del canal y lo envía con sus notas para cada patrón. `Instrument` construye un instrumento nuevo y envía `SetInstrument`. `Volume`, `Pan`, `ReverbSend` y `Effects` envían su único valor, `Muted` envía el silencio efectivo del canal, `LooseTiming` envía la cantidad nueva, `Mixer` envía el volumen maestro y los ajustes de la reverberación, y

`Song` envía los tres comandos que sustituyen la canción, sus tempos y los clics del metrónomo. `Tempo` envía el tempo a la velocidad de práctica, si ralentizaste la canción para practicar (ver Herramientas de práctica).

`into_owned` convierte el `Cow` de `played_notes` en un `Vec` que el comando puede llevar: clona una lista prestada, y simplemente entrega una que ya es propia.

`sync_audio` también mantiene los propios números de la ventana apuntando a lo correcto. El patrón seleccionado es una posición, así que baja cuando se va un patrón que estaba antes, y sube cuando la banda inserta uno antes. El piano roll recuerda qué canal muestra, así que se le avisa cuando los canales van y vienen (`channel_removed` y `channel_inserted`). Como estas actualizaciones están aquí, añadir, clonar y borrar patrones selecciona el patrón correcto tanto si lo hiciste tú como si lo hizo deshacer.

El hilo de audio inserta en su sitio

Deshacer «delete pattern 0» debe volver a poner un patrón en la posición 0, no al final. Por eso los comandos del hilo de audio reciben una posición:

```
/// Add an empty pattern at position `index`; the patterns from there
/// on move up one place. An index past the end adds it at the end.
InsertPattern { index: usize, length: usize },

/// Add a channel at position `index`, with its notes in every pattern.
/// The channels from there on move down one row. An index past the
/// end adds it at the bottom.
InsertChannel {
    index: usize,
    instrument: Instrument,
    notes: ChannelNotes,
    volume: f32,
    muted: bool,
},
```

`Vec::insert(index, value)` mueve cada elemento desde `index` un lugar hacia arriba y pone `value` en el hueco. ¿Reservaría eso memoria en el hilo de audio? Solo si el `Vec` estuviera lleno. El sitio para `MAX_CHANNELS` canales y `MAX_PATTERNS` patrones se reserva al crear el hilo de audio (ver Espacio reservado de antemano), y ambos comandos comprueban primero el límite, así que `insert` siempre cabe en memoria que ya está ahí. Mover elementos es solo copiar, lo cual está permitido en tiempo real:

```
fn insert_pattern(&mut self, index: usize, length: usize) {
    if self.pattern_lengths.len() < MAX_PATTERNS {
        let index = index.min(self.pattern_lengths.len());
        self.pattern_lengths.insert(index, length);
        for channel in &mut self.channels {
            channel.notes.insert_empty(index);
        }
        ...
    }
}
```

Un índice más allá del final se limita (`index.min(len)`) en lugar de rechazarse. El hilo de audio nunca debe entrar en pánico, y «insertar después del último» es la lectura más sensata de un número demasiado grande.

`InsertChannel` es uno de los comandos más grandes. Cada ranura de la cola de comandos es tan grande como el comando más grande, así que una prueba mantiene los comandos en 80 bytes como mucho. Las voces de un instrumento están detrás de un `Box` , así que un comando lleva un puntero a ellas, nunca todas ellas.

Los instrumentos vuelven de una caché

Deshacer «cargar muestra» le devuelve a un canal su instrumento antiguo, y puede que sea un archivo de muestra. Leer un archivo en el hilo de la ventana para un deshacer haría tartamudear a la ventana, y el archivo podría incluso haber desaparecido ya.

Así que la aplicación guarda una `SampleCache` : cada archivo de muestra leído hasta ahora, por ruta.

```
pub struct SampleCache {
    loaded: HashMap<PathBuf, Arc<SampleData>>,
}

impl SampleCache {
    pub fn get(&mut self, path: &Path) -> Result<Arc<SampleData>,
SampleError> {
        // A `HashMap<PathBuf, _>` can be searched with a `&Path`, because
        // `PathBuf` implements `Borrow<Path>`: no `PathBuf` is built
here.
        if let Some(sample) = self.loaded.get(path) {
            return Ok(Arc::clone(sample));
        }
        self.reload(path)
    }

    pub fn reload(&mut self, path: &Path) -> Result<Arc<SampleData>,
SampleError> {
        let sample = Arc::new(SampleData::load_wav(path)?);
        self.loaded.insert(path.to_path_buf(), Arc::clone(&sample));
        Ok(sample)
    }
}
```

Un `HashMap` encuentra un valor por su clave en más o menos el mismo tiempo tenga las entradas que tenga. Sus claves son `PathBuf`s con propiedad, pero `get` recibe un `&Path` prestado. Eso funciona gracias al **trait** `Borrow`: `PathBuf` implementa `Borrow<Path>`, una promesa de que un `PathBuf` y el `Path` que contiene hacen hash y se comparan de la misma manera. Así el mapa puede buscar un `&Path` directamente, sin construir un `PathBuf` solo para buscar. (Un `HashMap<String, _>` se puede buscar con un `&str` por la misma razón.)

La caché entrega clones de `Arc`, así que la caché y cada sampler que usa el sonido comparten una copia en memoria. Una prueba lo comprueba con `Arc::ptr_eq`, que pregunta si dos `Arc` apuntan a exactamente la misma memoria, no solo a valores iguales:

```
assert!(Arc::ptr_eq(&first, &second), "the second get shares the first");
assert!(!Arc::ptr_eq(&first, &reloaded), "reload reads the file again");
```

Soltar un archivo sobre el rack llama a `reload`, no a `get`: puede que hayas cambiado el archivo en un editor de audio desde entonces, y volverlo a soltar debe recogerlo. Un archivo que no se puede leer no se recuerda, así que el siguiente intento lo lee otra vez. Abrir otro proyecto vacía la caché.

`Instrument::for_channel` recibe la caché como argumento (`&mut SampleCache`) en lugar de leer los archivos por sí mismo. Exportar vuelve a construir cada instrumento

(ver Exportar), y gracias a la caché tampoco lee archivos.

Deshacer y rehacer

Archivos: `crates/lineup-core/src/history.rs` y `crates/lineup-app/src/app/edit.rs`

Todo cambio en una canción se puede deshacer con Ctrl+Z y rehacer con Ctrl+Mayús+Z o Ctrl+Y, y el menú Edición nombra el paso que desharía («Deshacer: mover clip»). Este capítulo trata de cómo se guarda el historial, y de cuándo termina un paso y empieza el siguiente.

Dos formas de deshacer

Hay dos formas clásicas de construir el deshacer.

El **patrón comando** registra cada cambio como una instrucción que sabe revertirse a sí misma: «añadir la nota Do4 en el paso 3» se deshace con «quitar la nota Do4 del paso 3». El historial es una lista de instrucciones. Usa muy poca memoria, pero cada tipo de cambio necesita su propia operación inversa, escrita a mano, y cada una es un lugar nuevo para equivocarse. Piensa en lo que necesitaría deshacer «eliminar patrón»: volver a poner el patrón en su antigua posición, con su nombre y su longitud y las notas de cada canal, volver a poner cada clip que lo usaba y renumerar de nuevo los clips de los patrones que van detrás. Si se olvida un detalle, deshacer devuelve en silencio una canción ligeramente distinta.

Las **instantáneas** guardan una copia completa del proyecto de antes de cada paso que se puede deshacer. Deshacer vuelve a poner la copia. No hay nada que revertir, así que no hay nada que revertir mal: haga lo que haga el cambio, la copia es exactamente lo que había antes. El coste es la memoria y el tiempo de hacer una copia.

Ferrous Lineup usa instantáneas. Un `NoteEvent` ocupa 24 bytes, y una canción tiene unos cientos o unos miles, así que una copia de todo el proyecto son algunos kilobytes. Incluso un proyecto grande con 10.000 notas guarda unos 240 KB de notas, y hacer una copia tarda mucho menos de un milisegundo. La aplicación guarda como mucho 256 MiB de instantáneas: una canción pequeña obtiene los 100 pasos completos, mientras que una muy grande obtiene al menos 10 (que pueden pasarse del presupuesto). Es un precio justo por un deshacer que es correcto por construcción. (Uno de los ejercicios muestra cómo abaratar mucho las copias, si un proyecto llega a ser lo bastante grande para necesitarlo.)

`History<T>`

El historial vive en `lineup-core`, y no sabe nada de proyectos:

```
struct Step<T> {
    label: &'static str,
    state: T,
}

pub struct History<T> {
    /// The oldest step is at the front, the newest at the back. A
    /// `VecDeque` can drop the oldest step from the front cheaply.
    undo: VecDeque<Step<T>>,
    redo: Vec<Step<T>>,
    limit: usize,
    /// Whether the newest undo step still takes in new changes.
    open: bool,
}

impl<T> History<T> {
    pub fn new(limit: usize) -> Self { ... }
    pub fn record(&mut self, label: &'static str, before: T) { ... }
    pub fn is_open(&self) -> bool { ... }
    pub fn close(&mut self) { ... }
    pub fn reopen(&mut self, label: &'static str) -> bool { ... }
    pub fn undo(&mut self, current: &mut T) -> Option<&'static str> { ...
}
    pub fn redo(&mut self, current: &mut T) -> Option<&'static str> { ...
}
    pub fn undo_label(&self) -> Option<&'static str> { ... }
    pub fn redo_label(&self) -> Option<&'static str> { ... }
    pub fn clear(&mut self) { ... }
}
```

`History<T>` es **genérico**: `T` es aquello que se está recordando. La aplicación usa un `History<Project>`; las pruebas usan un `History<i32>`, lo que las hace cortas y fáciles de leer:

```
#[test]
fn changes_join_the_open_step() {
    let mut history = History::default();
    let mut value = 0;
    // A drag: three changes before the mouse button is released.
    for new in [1, 2, 3] {
        history.record("drag", value);
        value = new;
    }
    history.close();
    history.undo(&mut value);
    assert_eq!(value, 0, "one undo goes back to before the drag");
}
```

Mira la línea `impl`: `impl<T> History<T>`, sin ningún **trait bound** sobre `T`. Ni siquiera `Clone`. El historial nunca copia un `T`; solo mueve los valores que recibe de una lista a otra. Quien llama hace la copia (`self.project.clone()`) y se la entrega. Si un método de `History` llamara alguna vez a `.clone()` sobre un estado, el compilador se negaría con «no method named `clone` found for type parameter `T`», y tendríamos que escribir `impl<T: Clone>`. Un tipo genérico sin bounds es una promesa fuerte: el código no puede hacer nada con un `T` salvo guardarlo, moverlo y soltarlo.

El compilador hace una copia separada del código para cada `T` con el que se usa (esto es la **monomorfización**), así que un `History<Project>` es tan rápido como si se hubiera escrito a mano para `Project`.

Las etiquetas `&'static str`

Cada paso tiene una etiqueta, como `"move clip"`, para el menú Edición. Su tipo es `&'static str`: una referencia a texto que vive durante todo el programa. Cada string literal escrito en el código tiene este tipo, porque el texto se guarda dentro del propio archivo del programa. Así que una etiqueta no cuesta nada: no se construye ningún `String`, no se reserva nada, y el historial no guarda ningún préstamo que pueda terminar demasiado pronto. El precio es que una etiqueta no se puede crear en tiempo de ejecución («rename pattern Beat»), porque ese texto no existiría hasta que el programa lo construyera.

Las etiquetas se escriben en inglés, y el menú Edición las traduce al mostrarlas, así que el historial no necesita saber el idioma de la aplicación (ver Dos idiomas).

VecDeque: una lista con dos extremos baratos

Cuando ya hay `limit` pasos, registrar uno más olvida el más antiguo. En un `Vec`, quitar el primer elemento (`remove(0)`) mueve todos los demás un lugar hacia abajo. Un `VecDeque` («cola de dos extremos», se dice «vec deck») es un ring buffer, como las colas entre los hilos (ver Dos hilos): lleva la cuenta de dónde está su frente, así que `push_back` y `pop_front` son ambos baratos.

El límite no es una constante en la aplicación. `History::new` lo recibe, y `undo_limit_for` lo calcula a partir del tamaño del proyecto:

```
const UNDO_BYTES: usize = 256 * 1024 * 1024;
const MIN_UNDO_STEPS: usize = 10;

pub(super) fn undo_limit_for(project: &Project) -> usize {
    limit_for_bytes(project.heap_bytes())
}

fn limit_for_bytes(bytes: usize) -> usize {
    (UNDO_BYTES / bytes.max(1)).clamp(MIN_UNDO_STEPS, DEFAULT_LIMIT)
}
```

`heap_bytes` cuenta las notas, los clips, las marcas y los nombres, las cosas cuyo número crece con la canción. Un proyecto pequeño divide 256 MiB entre unos pocos kilobytes y obtiene más que `DEFAULT_LIMIT`, así que se limita a 100. Un proyecto enorme obtiene menos, pero nunca menos de 10, para que deshacer no desaparezca solo porque la canción es larga. Abrir un archivo construye un `History` nuevo con el límite de ese proyecto. `History::default()` significa 100, que es lo que usan las pruebas pequeñas.

```
pub fn record(&mut self, label: &'static str, before: T) {
    self.redo.clear();
    if self.open {
        return;
    }
    self.undo.push_back(Step { label, state: before });
    if self.undo.len() > self.limit {
        self.undo.pop_front();
    }
    self.open = true;
}
```

Cada cambio nuevo vacía la lista de rehacer. Después de deshacer tres pasos y cambiar algo, esos tres pasos pertenecen a un futuro que ya no existe. (Unos pocos programas guardan un árbol entero de historiales, para no perder nunca ninguna rama; la mayoría simplemente descarta el futuro antiguo, como hace este.)

La lista de rehacer es un `Vec` normal, porque los pasos solo se añaden y se quitan por su final.

`mem::swap`: deshacer sin copiar

```
pub fn undo(&mut self, current: &mut T) -> Option<&'static str> {
    let mut step = self.undo.pop_back()?;
    // Now `current` holds the old state and `step` the newer one.
    std::mem::swap(current, &mut step.state);
    let label = step.label;
    self.redo.push(step);
    self.open = false;
    Some(label)
}
```

`pop_back()` devuelve un `Option`, y el operador `?` funciona sobre `Option` en una función que devuelve un `Option`: si no hay nada que deshacer, `undo` devuelve `None` al momento y no cambia nada.

`std::mem::swap` intercambia los valores que hay detrás de dos referencias `&mut`. Después, `current` (el proyecto de la aplicación) guarda la versión más antigua, y el paso guarda la versión de la que acabamos de salir, lista para rehacer. No se copia ni se reserva nada: un `Project` es sobre todo `Vec`s, y un `Vec` es un puntero, una longitud y una capacidad, así que intercambiar dos proyectos intercambia unos pocos de esos valores pequeños, sea cual sea el tamaño de la canción. El borrow checker es lo que hace esto sencillo: `swap` necesita dos referencias `&mut`, y dos referencias `&mut` nunca pueden apuntar al mismo valor, así que `swap` puede intercambiar los bytes de los dos valores sin comprobar nada.

`redo` es la imagen especular: saca de `redo` y mete en `undo`.

¿Cuándo termina un paso?

Arrastrar un clip por la lista de reproducción lo mueve un poco en muchos fotogramas, y cada movimiento es una edición. Si cada uno fuera su propio paso, deshacer el arrastre necesitaría tantas pulsaciones de Ctrl+Z como fotogramas hubiera. Así que el historial mantiene el paso más nuevo **abierto**, y `record` no hace nada mientras lo esté: el paso ya guarda el proyecto de antes de que empezara el arrastre, que es exactamente adonde debería volver deshacer.

La aplicación decide cuándo termina un paso, porque solo ella sabe qué está haciendo el ratón. Llama al historial al principio y al final de cada fotograma:

```
/// Called at the start of each frame. A press of a mouse button
/// always begins a new undo step.
pub(super) fn start_frame_for_undo(&mut self, ctx: &egui::Context) {
    if ctx.input(|input| input.pointer.any_pressed()) {
        self.close_step();
    }
}

/// Called at the end of each frame. While a mouse button is held (a
/// drag), a text field has the keyboard (typing a name) or a take is
/// being recorded, changes join one undo step; once all of them are
/// over, the step ends.
pub(super) fn end_frame_for_undo(&mut self, ctx: &egui::Context) {
    let holding = ctx.input(|input| input.pointer.any_down());
    if !holding && !ctx.text_edit_focused() && !self.is_recording() {
        self.close_step();
    } else if self.step_unsynced {
        // The step goes on, but its last change may still be waiting
        // for its turn: a drag held still, or the last key typed, has
        // no next edit to carry it to the audio thread. Send it when
        // it is due, and make sure a frame comes then.
        self.sync_step_if_due();
        if self.step_unsynced {

ctx.request_repaint_after(SYNC_INTERVAL.saturating_sub(self.last_sync.elap
sed()));
        }
    }
}
```

Sigue unos cuantos gestos a través de estas dos funciones:

- **Un clic en un paso.** egui informa del clic en el fotograma en que se suelta el botón. La edición de ese fotograma abre un paso, y el final del mismo fotograma lo cierra, porque ya no hay ningún botón pulsado: un cambio, un paso.
- **Un arrastre.** La pulsación cierra lo que estuviera abierto. Cada fotograma del arrastre edita el proyecto mientras el botón está pulsado, así que las ediciones se unen en un paso. El fotograma en que se suelta el botón lo cierra.
- **Escribir un nombre de patrón.** Mientras el campo de texto tiene el foco del teclado, cada letra se une al mismo paso. Hacer clic en otro sitio le quita el foco y lo cierra, así que deshacer devuelve todo el nombre antiguo de una vez.
- **Escribir un tempo.** Hacer clic en la casilla de BPM la convierte en un campo de texto, así que se aplica la misma regla: todo lo escrito ahí es un paso, que termina cuando sales de la casilla.
- **Grabar una toma.** Cada nota tocada mientras se graba se une a un mismo paso, así que un deshacer quita la toma entera.

`start_frame_for_undo` importa en un caso: un clic que empieza cuando un campo de texto aún tiene el foco. La pulsación cierra el paso de escritura, para que el cambio del clic no se le una.

La rama `else if` trata del sonido, no de deshacer. Mientras un paso está abierto, los cambios llegan al hilo de audio como mucho cada 50 ms, llevados por la edición siguiente (ver Decirle al hilo de audio lo que ha cambiado). Un arrastre que se queda quieto no hace ninguna edición siguiente, así que el fotograma envía lo que aún espera cuando toca, y `request_repaint_after` le pide a egui un fotograma en ese momento aunque nada se mueva.

Cerrar un paso es `close_step`:

```
fn close_step(&mut self) {
    self.finish_step_sync();
    if self.history.is_open() {
        self.file.project_reverted();
    }
    self.history.close();
}
```

Envía lo que al hilo de audio aún le falte, y vuelve a calcular el asterisco de cambios sin guardar (una nota arrastrada lejos y de vuelta deja la canción como estaba guardada). Después el historial deja de aceptar cambios en el paso.

Varios clics, un paso

Algunos cambios llegan de clic en clic pero van juntos. Marcar un tempo con golpes es un clic por pulso; retocar el volumen maestro o el número de vueltas es un clic o un arrastre por retoque. Deshacer debería quitarlos todos de una vez. `reopen` deja que la aplicación vuelva a abrir el paso más nuevo:

```
/// Opens the newest step again, if it is called `label` and nothing
/// was undone since, so the next change joins it: taps of a tempo,
/// each a click of its own, make one step. Returns whether it did.
pub fn reopen(&mut self, label: &'static str) -> bool {
    if self.redo.is_empty() && self.undo_label() == Some(label) {
        self.open = true;
    }
    self.open
}
```

y la aplicación la llama justo antes de la edición:

```
pub(super) fn set_master_volume(&mut self, master: f32) {
    const STEP: &str = "master volume";
    self.history.reopen(STEP);
    self.edit(STEP, |project| project.mixer.master = master);
}
```

Si el último paso fue otro cambio, o se deshizo algo desde entonces, la etiqueta no coincide y empieza un paso nuevo como siempre.

Todo pasa por `edit`

El campo del nombre del patrón muestra lo estricta que es la regla «cada cambio pasa por `edit`». El campo de texto de egui quiere un `&mut String` para cambiar. Darle el nombre del propio proyecto cambiaría el proyecto directamente, a espaldas del historial: sin paso de deshacer y sin mensaje al hilo de audio tampoco. Así que el campo edita una copia, y la copia pasa por `edit` cuando ha cambiado:

```
// The field edits a copy, so the change can go through `edit`.
let mut name = self.project.patterns()[self.pattern].name.clone();
if ui.text_edit_singleline(&mut name).changed() {
    self.edit_pattern("rename pattern", |pattern| pattern.name = name);
}
```

El tipo `Project` ayuda aquí: sus listas son privadas, y solo sus métodos pueden cambiarlas (ver El proyecto). Cualquier cosa que cambie un proyecto necesita un `&mut Project`, y en la aplicación el único sitio que lo entrega es `edit`.

Atajos de teclado

```
const UNDO: KeyboardShortcut = KeyboardShortcut::new(Modifiers::COMMAND,
Key::Z);
const REDO: KeyboardShortcut =
    KeyboardShortcut::new(Modifiers::COMMAND.plus(Modifiers::SHIFT),
Key::Z);
/// The other common shortcut for redo, used on Windows.
const REDO_Y: KeyboardShortcut = KeyboardShortcut::new(Modifiers::COMMAND,
Key::Y);

pub(super) fn edit_shortcuts(&mut self, ctx: &egui::Context) {
    let pressed = |shortcut| ctx.input_mut(|input|
input.consume_shortcut(&shortcut));
    // Ctrl+Z would also match Ctrl+Shift+Z, so redo is checked first.
    if pressed(REDO) || pressed(REDO_Y) {
        self.redo();
    } else if pressed(UNDO) {
        self.undo();
    }
}
```

`Modifiers::COMMAND` es Ctrl en Linux y Windows y ⌘ en un Mac. El `consume_shortcut` de egui es indulgente con las teclas de más: un atajo sin Mayús también encaja cuando Mayús está pulsado. Normalmente eso es lo que quieres (en muchos teclados `+` necesita Mayús, y Ctrl++ debería seguir funcionando), pero aquí significa que Ctrl+Mayús+Z contaría también como Ctrl+Z. Comprobar primero rehacer, y usar `else if`, lo resuelve.

`||` se detiene en el primer `true`, así que si se pulsó Ctrl+Mayús+Z, Ctrl+Y ni siquiera se mira. «Consume» significa que la pulsación se saca de la entrada de egui, para que nada más en el mismo fotograma reaccione a ella.

Estos atajos se omiten mientras un campo de texto tiene el teclado. Los campos de texto de egui tienen su propio deshacer para escribir, y allí Ctrl+Z debe deshacer las últimas letras, no el último cambio de la canción.

El menú Edición muestra los mismos atajos, escritos como los escribe el sistema («Ctrl+Z» en Linux, «⌘Z» en un Mac), y pone la etiqueta traducida en el elemento, «Deshacer: mover clip».

Deshacer y el asterisco de cambios sin guardar

El título de la ventana muestra un asterisco cuando el proyecto difiere del archivo guardado. No compara los dos proyectos en cada fotograma: una edición pone una marca, y la comparación ocurre la próxima vez que se calcula el título. Deshacer y rehacer también ponen la marca, para que puedan hacer desaparecer el asterisco cuando el proyecto vuelve a ser igual que el guardado. Un simple `dirty: bool` puesto en cada edición habría mantenido el asterisco después de deshacer. Guardar muestra el resto.

Abrir un archivo, o Archivo > Nuevo, sustituye todo el proyecto y vacía el historial. Deshacer metiéndote en otra canción sería desconcertante, y la pregunta «¿Guardar cambios?» ya protege lo que dejas atrás.

Guardar: el archivo .lineup, la recuperación y los archivos dañados

Archivos: `crates/lineup-core/src/file.rs`, `crates/lineup-app/src/app/files.rs`, `crates/lineup-app/src/app/recovery.rs`, `crates/lineup-app/src/app/recent.rs` y `crates/lineup-core/tests/damaged_files.rs`

Una canción se guarda como un archivo `.lineup`. Este capítulo trata de qué hay en ese archivo y por qué tiene esa forma, de cómo se lee un archivo de una versión más vieja o más nueva, de cómo guardar evita dejar nunca un archivo escrito a medias, y de cómo la aplicación guarda una copia del trabajo sin guardar por si se cae. La lista numerada de versiones del formato está en la página de referencia Formato del archivo de proyecto.

Cómo es un archivo de proyecto

Un archivo `.lineup` es texto plano. Para ver uno, abre una demostración en Archivo > Demostraciones, usa Archivo > Guardar como… y abre el archivo guardado en un editor de texto. Archivo > Nuevo ritmo empieza una canción con un patrón vacío y una batería; este es el archivo entero que guarda:

```
(
    format: 1,
    tempo: 130.0,
    scale: (
        root: C,
        kind: Major,
    ),
    channels: [
        (
            name: "Kick",
            kind: Sampler(Builtin(Kick)),
            volume: 0.8,
            muted: false,
        ),
        (
            name: "Clap",
            kind: Sampler(Builtin(Clap)),
            volume: 0.8,
            muted: false,
        ),
        ...
        (
            name: "Lineup Osc",
            kind: Synth,
            volume: 0.6,
            muted: false,
        ),
    ],
    patterns: [
        (
            name: "Pattern 1",
            length: 16,
            notes: [
                [],
                [],
                [],
                [],
                [],
            ],
        ),
    ],
    tracks: [
        (
            name: "Track 1",
            muted: false,
        ),
    ],
    clips: [],
)
```

Otras canciones llevan más. El canal de sintetizador de la canción de inicio toca un piano General MIDI, así que lleva `program: Some(0)` y su archivo dice `format: 9`. Sus

acordes se leen como notas, una por línea:

```
                    (pitch: 48, start: 0, length: 8, velocity: 100),
                    (pitch: 60, start: 0, length: 8, velocity: 100),
                    (pitch: 64, start: 0, length: 8, velocity: 100),
                    (pitch: 67, start: 0, length: 8, velocity: 100),
```

Archivo > Nuevo, una hoja de acordes vacía con una banda, guarda `format: 17` y termina con la banda: `band: Some((style: "Pop beat", ...))`.

El formato es **RON**, «Rusty Object Notation». Se parece a Rust: un struct es `( field: value, ... )`, una lista es `[ ... ]`, y las variantes de enum se escriben por su nombre, `Sampler(Builtin(Kick))`, igual que en el código. JSON también serviría, pero JSON no tiene enums, así que el mismo canal se leería como `{"Sampler": {"Builtin": "Kick"}}`. RON es más fácil de leer y de arreglar a mano, algo que importa en un programa con el que se aprende: puedes abrir una canción en un editor de texto y ver tus propias estructuras de datos.

serde: describir los datos una sola vez

La forma estándar de Rust de convertir valores en texto (o bytes) y de vuelta es el crate **serde**, abreviatura de *ser*ialize y *de*serialize. serde no conoce ningún formato por sí mismo. Define dos traits, `Serialize` y `Deserialize`, y los crates de formato como `ron` o `serde_json` los usan.

Casi nunca escribes esos traits a mano. Una **macro derive** los escribe por ti:

```rust
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Channel {
    pub name: String,
    pub kind: ChannelKind,
    pub volume: f32,
    pub muted: bool,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub program: Option<u8>,
    #[serde(default, skip_serializing_if = "is_centred")]
    pub pan: f32,
    // ...the reverb send, the MIDI channel a drum is sent on, and the
    // channel's effects, each left out at its default.
}
```

La macro lee la definición del struct en tiempo de compilación y genera código que escribe cada campo por su nombre y los lee de vuelta. Para que esto funcione, el tipo de cada campo debe ser también `Serialize`, así que `ChannelKind`, `SampleSource`, `Drum`, `NoteEvent`, `Scale`, `PitchClass`, `Track`, `Clip` y los demás también lo derivan. `String`, los números, `bool`, `Vec`, `Option` y `PathBuf` vienen con serde.

El derive se activa pidiendo la feature `derive` de serde en `Cargo.toml`. Las **features** son partes opcionales de un crate que enciendes cuando las necesitas:

```
serde = { version = "1", features = ["derive"] }
```

Las líneas `#[serde(default, skip_serializing_if = ...)]` son la forma silenciosa de crecer del formato de archivo. Un campo que está en su valor por defecto no se escribe en absoluto, y un campo que falta en un archivo más viejo se rellena con su valor por defecto. Por eso la batería de arriba, que no usa nada más allá del primer formato, se escribe exactamente como lo escribía la primera versión del formato: un canal en el centro y con todo en la reverberación no dice nada del paneo ni del envío a la reverberación.

Mantener las reglas al leer: `from` e `into`

Un `Tempo` no se puede crear fuera de 10 a 522 BPM, porque su campo es privado y `Tempo::new` lo limita. Un `Deserialize` derivado pondría el campo directamente desde el archivo, así que un archivo que diga `tempo: 9000.0` crearía un tempo que el resto del programa promete que no puede existir.

El arreglo es decirle a serde que pase por un tipo que controlamos:

```rust
#[derive(Debug, Clone, Copy, PartialEq, PartialOrd, Serialize,
Deserialize)]
#[serde(from = "f32", into = "f32")]
pub struct Tempo {
    bpm: f32,
}

impl From<f32> for Tempo {
    fn from(bpm: f32) -> Self {
        Tempo::new(bpm)
    }
}

impl From<Tempo> for f32 {
    fn from(tempo: Tempo) -> Self {
        tempo.bpm
    }
}
```

`#[serde(from = "f32")]` significa «lee un `f32` y luego conviértelo con `From<f32>`». Esa conversión es `Tempo::new`, que limita. La prueba `values_out_of_range_are_clamped` edita un archivo guardado para que diga 9000 BPM y comprueba que carga como 522. `into = "f32"` hace lo contrario al guardar, lo que además hace que el archivo diga `tempo: 130.0` en lugar de `tempo: (bpm: 130.0)`. `Note` hace lo mismo con su número MIDI.

Siempre en letras y en inglés

La aplicación puede mostrar los nombres de las notas en letras (C, D, E) o en solfeo (Do, Re, Mi), y sus textos en inglés o en español (ver Dos idiomas). Un archivo no debe depender de ninguna de las dos cosas: una canción guardada por alguien que lee solfeo debe abrirse para alguien que lee letras. Así que `to_text` y `from_text` fijan ambas mientras se ejecutan:

```rust
let file = Language::English.with(|| NoteNaming::Letters.with(||
ProjectFile::from(project)));
let mut text = NoteNaming::Letters.with(||
ron::ser::to_string_pretty(&file, style))?;
```

`with` fija la forma de nombrar para el hilo, ejecuta el closure y vuelve a poner la anterior, así que la ventana vuelve a dibujar en solfeo en su siguiente fotograma.

Un tipo separado para el archivo

`Project` en sí no deriva serde. Sus campos son privados porque deben estar de acuerdo entre ellos (una fila por canal, clips que apuntan a patrones reales), y un `Deserialize` derivado los rellenaría directamente desde el archivo, estén de acuerdo o no. Así que el archivo tiene su propio struct:

```
#[derive(Debug, Serialize, Deserialize)]
struct ProjectFile {
    format: u32,
    #[serde(default, skip_serializing_if = "crate::Mixer::is_default")]
    mixer: crate::Mixer,
    tempo: Tempo,
    scale: Scale,
    #[serde(default, skip_serializing_if = "TimeSignature::is_common")]
    time_signature: TimeSignature,
    channels: Vec<Channel>,
    patterns: Vec<PatternFile>,
    tracks: Vec<Track>,
    clips: Vec<Clip>,
    // ...sections, the loop region, time, tempo and key markers, loose
    // timing, swing, the chord sheet, the band, where each part came
    // from, and the locked channels: each left out when it is empty.
}

#[derive(Debug, Serialize, Deserialize)]
struct PatternFile {
    name: String,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    color: Option<usize>,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    time: Option<TimeSignature>,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    player: Option<Player>,
    length: usize,
    /// One list of notes per channel, in the order of `channels`.
    notes: Vec<Vec<NoteEvent>>,
}
```

Un `ProjectFile` acepta cualquier cosa con la forma correcta. Convertirlo en un `Project` es donde ocurre la comprobación, y se escribe como una implementación de `TryFrom`, el trait estándar para «una conversión que puede fallar»:

```
impl TryFrom<ProjectFile> for Project {
    type Error = FileError;

    fn try_from(file: ProjectFile) -> Result<Self, Self::Error> {
        let mut project = Project::empty();
        // ...checks on the mixer, the time signature and each channel's
        // mix, effects and MIDI channel...
        for channel in file.channels {
            project.add_channel(channel);
        }
        for (index, saved) in file.patterns.into_iter().enumerate() {
            // `Project::empty` already has one pattern; add the others.
            if index > 0 {
                project.add_pattern();
            }
            let Some(pattern) = project.pattern_mut(index) else { ... };
            pattern.set_length(saved.length);
            for (channel, notes) in saved.notes.into_iter().enumerate() {
                for note in notes {
                    if pattern.add_note(channel, note).is_none() {
                        return invalid(/* where, and what is wrong */);
                    }
                }
            }
        }
        // ...tracks, clips, sections, markers, the chord sheet and the
band...
        Ok(project)
    }
}
```

El proyecto se reconstruye **por la puerta principal**: `add_channel`, `add_pattern`, `add_note`, `add_clip`, los mismos métodos que llama la aplicación cuando haces clic. Esos métodos ya mantienen todas las reglas (ver El proyecto), así que un archivo nunca puede crear un proyecto que la ventana no habría podido crear. Cuando un método se niega (una nota que no cabe, un clip que reproduce un patrón que no está), el error dice dónde, para que alguien que edite el archivo a mano pueda arreglarlo: «el archivo del proyecto está dañado: el clip 1 (patrón 99, pista 1) no entra en la lista de reproducción».

`type Error = FileError;` es un **tipo asociado**: parte del trait que elige cada implementación. Con `TryFrom` en su sitio, la biblioteca estándar también proporciona el `try_into` correspondiente, así que `let project: Project = file.try_into()?;` también funcionaría.

Guardar va en la otra dirección con `From<&Project> for ProjectFile`, que copia los datos hacia fuera con `to_vec()` y `map`. No puede fallar, así que es un `From`, no un `TryFrom`.

El número de versión

```
pub const FORMAT_VERSION: u32 = 29;
```

`FORMAT_VERSION` es la disposición más nueva que este programa puede leer. Un archivo no siempre la usa: `format_for` escribe el formato más **bajo** que puede contener el proyecto. Un proyecto que no usa nada nuevo se queda en formato 1, para que un Ferrous Lineup más viejo pueda seguir abriéndolo. La canción de inicio necesita el formato 9, porque su canal de sintetizador toca un programa General MIDI; una canción con hoja de acordes necesita al menos el formato 12; una con sección de vientos o con un efecto en un canal necesita el 28, y una banda que toca la batería de otro estilo, el 29. El largo comentario encima de `FORMAT_VERSION` dice, para cada número, qué añadió y qué haría mal un programa más viejo sin él; Formato del archivo de proyecto tiene la misma lista como tabla.

Un archivo de una versión **más nueva** debe rechazarse con un mensaje claro, no leerse mal. La forma obvia es leer primero un struct diminuto:

```
#[derive(Deserialize)]
struct Header {
    format: u32,
}
```

Por defecto, serde se salta los campos que un struct no enumera, así que un `Header` se puede leer del archivo entero. Pero para saltarse el resto del archivo, RON tiene que mirar hacia delante a través de todo él, número a número. Con una canción larga como el Boléro, eso tardaba más de un minuto, incluso en una compilación release. Así que primero se lee el archivo entero, y solo si eso falla `format_number` lee el número de los primeros bytes:

```
pub fn from_text(text: &str) -> Result<Project, FileError> {
    // A text editor may start the file with a byte-order mark, which is
    // not part of the RON text or of the format number.
    let text = text.strip_prefix('\u{feff}').unwrap_or(text);
    let file: ProjectFile = match NoteNaming::Letters.with(||
ron::from_str(text)) {
        Ok(file) => file,
        Err(err) => {
            return Err(match format_number(text) {
                Some(format) if format > FORMAT_VERSION =>
FileError::NewerFormat(format),
                _ => err.into(),
            });
        }
    };
    if file.format > FORMAT_VERSION {
        return Err(FileError::NewerFormat(file.format));
    }
    NoteNaming::Letters.with(|| Project::try_from(file))
}

/// The format number at the start of a project file, `(format: 25, ...`,
/// read without the RON reader.
fn format_number(text: &str) -> Option<u32> {
    let rest = text.trim_start().strip_prefix('(')?.trim_start();
    let rest = rest.strip_prefix("format")?.trim_start();
    let rest = rest.strip_prefix(':')?.trim_start();
    let digits = rest
        .find(|c: char| !c.is_ascii_digit())
        .unwrap_or(rest.len());
    rest[..digits].parse().ok()
}
```

`to_text` siempre escribe `format` primero, así que esto mira solo donde debe estar el número. Un archivo más nuevo a menudo no se puede leer en absoluto, por campos o nombres que esta versión no conoce; entonces el número de formato por sí solo explica por qué. Si el archivo está tan roto que ni siquiera el número se puede leer, el error de RON original es el que se informa. Y si el archivo se lee pero aun así dice un formato más nuevo, también se rechaza, porque el mismo texto puede significar otra cosa en ese formato. Una marca de orden de bytes al principio (algunos editores de texto añaden una) se elimina antes, para que no pueda esconder el `(`.

La prueba `files_from_the_future_are_refused` lee un archivo un formato más nuevo que esta versión, hoy `(format: 30, sparkles: true)`, y espera `NewerFormat(30)`, con el mensaje «este archivo usa el formato 30, hecho por una versión más nueva de Ferrous Lineup (esta lee hasta el formato 29)».

Errores como enum

```
#[derive(Debug)]
pub enum FileError {
    /// The file could not be read or written: missing, no permission,
disk full...
    Io(io::Error),
    /// The text is not a valid project file. The error says where.
    Parse(ron::error::SpannedError),
    /// The project could not be turned into text (should never happen).
    Write(ron::Error),
    /// The file was saved by a newer version of Ferrous Lineup.
    NewerFormat(u32),
    /// The file is shaped right, but what it says does not make sense.
    Invalid(String),
}
```

Es el mismo patrón que `EngineError` (ver Un tipo de error propio) y `SampleError`: una variante por cada tipo de problema, una implementación de `Display` con un mensaje para las personas (en el idioma de la aplicación), y `std::error::Error` para que funcione con cualquier tratamiento de errores de Rust. Dos cosas más lo hacen cómodo de usar.

`From` para el `?`. Dentro de una función que devuelve `Result<_, FileError>`, el operador `?` solo puede pasar un error si sabe cómo convertirlo en un `FileError`. Estas implementaciones se lo enseñan:

```
impl From<io::Error> for FileError {
    fn from(err: io::Error) -> Self {
        FileError::Io(err)
    }
}
```

Con una para `io::Error`, una para el error de lectura de `ron` y una para su error de escritura, una función puede mezclar llamadas de ambas bibliotecas y usar `?` en todas ellas:

```
pub fn load(path: &Path) -> Result<Project, FileError> {
    let text = fs::read_to_string(path)?; // io::Error -> FileError::Io
    from_text(&text)
}
```

`source`. El trait `Error` tiene un método opcional que devuelve el error de nivel inferior que envuelve un valor. Las herramientas que informan de errores lo siguen para imprimir una cadena como «no se pudo cargar: no existe el archivo».

`FileError::Io` devuelve su `io::Error`; `Invalid` no tiene nada debajo y devuelve `None`.

```
fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
    match self {
        FileError::Io(err) => Some(err),
        FileError::Parse(err) => Some(err),
        FileError::Write(err) => Some(err),
        FileError::NewerFormat(_) | FileError::Invalid(_) => None,
    }
}
```

`&(dyn Error + 'static)` es un **objeto trait**: una referencia a *algún* valor que implementa `Error`, sin decir de qué tipo. Eso permite a un método devolver un `io::Error` en un caso y un error de `ron` en otro.

Guardar sin perder el archivo viejo

```
pub fn save(project: &Project, path: &Path) -> Result<(), FileError> {
    let text = to_text(project)?;
    let temporary = path.with_extension(format!("{EXTENSION}.tmp"));
    if let Err(err) = fs::write(&temporary, text) {
        let _ = fs::remove_file(&temporary);
        return Err(err.into());
    }
    if let Err(err) = fs::rename(&temporary, path) {
        let _ = fs::remove_file(&temporary);
        return Err(err.into());
    }
    Ok(())
}
```

Escribir directamente sobre `song.lineup` tiene un fallo feo: si el programa se cae o el disco se llena a mitad de camino, el archivo queda mitad viejo, mitad nuevo, y ninguna de las dos versiones se puede cargar. Escribir primero en `song.lineup.tmp` y luego renombrarlo sobre el original lo evita, porque el sistema operativo reemplaza un archivo al renombrar en un único paso. O el renombrado ocurrió y el archivo nuevo está completo, o no ocurrió y el archivo viejo queda intacto. En Windows, el `fs::rename` de Rust también reemplaza un archivo existente, así que el mismo código funciona en todas partes. Las exportaciones WAV y de video y la copia de recuperación se escriben de la misma manera.

`to_text` termina siempre las líneas con `\n` (RON usaría `\r\n` en Windows), así que una canción es el mismo archivo en todos los sistemas, y un proyecto guardado en un

ordenador no muestra diferencias en el control de versiones al guardarlo en otro.

Las pruebas guardan en `std::env::temp_dir()`, la carpeta del sistema para archivos temporales (`/tmp` en Linux, algo dentro de tu carpeta de usuario en Windows), con el id del proceso en el nombre para que dos ejecuciones de pruebas a la vez no choquen, y borran el archivo después.

Los archivos dañados nunca hacen caer la aplicación

Un archivo de proyecto puede quedar cortado por un disco lleno, editarse a mano, o ser otra cosa con un nombre `.lineup`. Los archivos MIDI vienen de todos los programas que existen. Leer un archivo debe terminar en un proyecto o en un mensaje de error, nunca en una caída. El archivo de pruebas `damaged_files.rs` lo comprueba como lo hace un **fuzzer**: daña archivos buenos de muchas formas pequeñas y lee cada resultado.

El daño viene de un pequeño generador de números aleatorios, xorshift, escrito en la prueba:

```
struct Damage(u64);

impl Damage {
    fn next(&mut self) -> u64 {
        self.0 ^= self.0 << 13;
        self.0 ^= self.0 >> 7;
        self.0 ^= self.0 << 17;
        self.0
    }
}
```

Empieza desde una semilla fija, así que cada ejecución daña los archivos de la misma manera, y un fallo vuelve a aparecer en la siguiente ejecución. No necesita ninguna dependencia nueva. En los archivos de proyecto y las hojas de acordes, sustituye números por valores extremos (`-1`, `65535`, `18446744073709551615`, `NaN`), recorta trozos y repite trozos. En los archivos MIDI, cambia, recorta y repite bytes. Cada prueba lee 600 copias dañadas. Un proyecto que se lee también debe guardarse y volver a leerse.

La prueba captura un panic con `std::panic::catch_unwind`, así que un fallo imprime el archivo dañado que lo causó, hasta sus primeros 4000 caracteres:

```rust
fn must_not_panic<T: std::fmt::Debug>(input: T, read: impl FnOnce(&T)) {
    if panic::catch_unwind(AssertUnwindSafe(|| read(&input))).is_err() {
        let shown = format!("{input:?}");
        let shown: String = shown.chars().take(4000).collect();
        panic!("reading this damaged file panicked:\n{shown}");
    }
}
```

El desbordamiento es el tipo de error que atrapa. Un archivo con un cambio de tonalidad en el paso 18446744073709551615, el `usize` más grande, no pasa la validación, como debe ser, pero escribir el mensaje de error calculaba `marker.start + 1` para decir «paso …», y eso desbordaba. Rust trata el desbordamiento de forma distinta en los dos perfiles de compilación. En una compilación **debug**, `+` entra en pánico al desbordar («attempt to add with overflow»). En una compilación **release** da la vuelta hasta 0, en silencio. Así que una ejecución en modo release sobre 15.000 archivos dañados no vio nada, y los mismos archivos en modo debug se caían. Los mensajes usan `saturating_add(1)`, que se queda en el número más grande en lugar de desbordar, y la integración continua ejecuta las pruebas en modo debug (ver Cómo se prueba el código).

En la aplicación: el menú Archivo

El lado de la aplicación está en `crates/lineup-app/src/app/files.rs`. Sus primeras líneas explican una disposición poco habitual:

```rust
//! This is a child module of `app` (its file is `app/files.rs`), so it
can
//! use the private fields of [`LineupApp`]: in Rust, private means
"visible
//! in this module and the modules inside it".
```

`app.rs` declara `mod files;`, y Rust busca ese módulo en `app/files.rs`. El archivo contiene otro bloque `impl LineupApp`. Un tipo puede tener cualquier número de bloques `impl`, en cualquier módulo de su crate, así que un tipo grande se puede dividir por temas. Al ser hijo de `app`, `files` ve los campos privados `self.project`, `self.audio` y demás, cosa que un módulo hermano como `playlist` no podría.

Elegir un archivo

Los diálogos de archivos vienen del crate **rfd** («Rusty File Dialog»), que muestra el diálogo propio del sistema: GTK o KDE en Linux (a través del portal del escritorio, o del

programa `zenity`), el panel del Finder en macOS, el diálogo del Explorador en Windows.

```
fn project_dialog() -> rfd::FileDialog {
    rfd::FileDialog::new().add_filter(tr("Ferrous Lineup project"), &
[file::EXTENSION])
}

if let Some(path) = project_dialog().pick_file() {
    self.open_now(path);
}
```

`pick_file` espera hasta que el usuario elige un archivo o cancela, y devuelve un `Option<PathBuf>`. La ventana no se redibuja mientras el diálogo está abierto, pero la música sigue sonando, porque corre en el hilo de audio. En un sistema Linux sin portal ni `zenity`, `pick_file` devuelve `None` inmediatamente; el diálogo simplemente no aparece.

Guardar como… le da al archivo la extensión `.lineup` aunque escribas otra, así que `song.txt` se guarda como `song.lineup`, no como algo que otro importador intentaría abrir.

«¿Guardar cambios?»

¿Cómo sabe la aplicación que hay cambios sin guardar? Mantiene una copia del proyecto tal como estaba la última vez que se abrió o guardó, y compara. Pero `Project` deriva `PartialEq`, así que `!=` recorre cada canal, nota y clip. Eso es barato una vez, pero un arrastre edita el proyecto en cada fotograma, y comparar la canción entera en cada fotograma es trabajo desperdiciado. `FileState` por eso guarda la respuesta:

```
/// Whether the project differs from `saved`, as last worked out.
dirty: bool,
/// Set when the project may differ from `saved`; `dirty` is worked
/// out again the next time it is asked for.
check_dirty: bool,
```

Una edición pone `check_dirty`, pero solo si el proyecto no tenía ya cambios. Mientras los tiene, los siguientes fotogramas de un arrastre no vuelven a pedir la comparación entera:

```
pub(super) fn project_edited(&mut self) {
    if !self.dirty {
        self.check_dirty = true;
    }
}
```

Un cambio hecho de una vez (un clic), deshacer, rehacer y el final de un arrastre pueden hacer que el proyecto vuelva a coincidir con la copia guardada, así que siempre piden una comparación nueva con `project_reverted` (ver Decirle al hilo de audio lo que ha cambiado). La respuesta guardada se lee en cada fotograma para el título, pero la comparación en sí solo se hace cuando la marca está puesta:

```
pub(super) fn has_unsaved_changes(&mut self) -> bool {
    if self.file.check_dirty {
        self.file.dirty = match &self.file.saved {
            Some(saved) => self.project != *saved,
            None => true, // recovered work has never been saved
        };
        self.file.check_dirty = false;
    }
    self.file.dirty
}
```

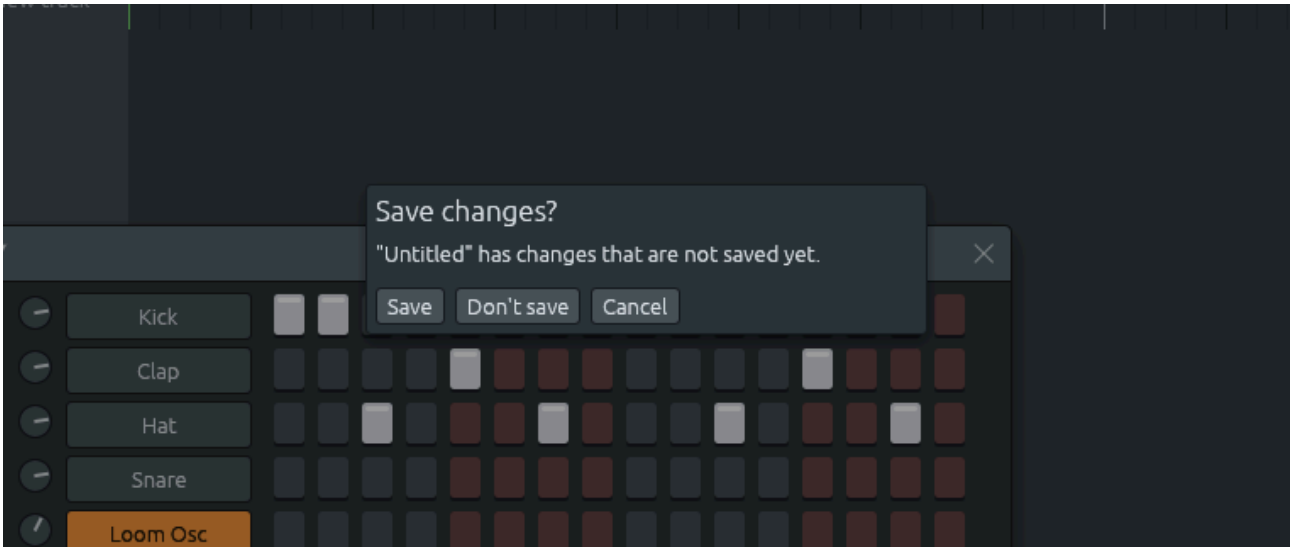
Así, un paso que se activa y se vuelve a desactivar quita el asterisco, y también una nota arrastrada lejos y de vuelta, en cuanto la sueltas.

Cualquier cosa que fuera a tirar el proyecto espera antes una respuesta:

```rust
pub(super) enum Pending {
    /// A chord sheet with a band.
    New,
    /// An empty pattern and a drum kit.
    NewBeat,
    Open,
    OpenPath(PathBuf),
    Demo(Demo),
    Import,
    ImportMidi,
    ImportChordSheet,
    Quit,
}

pub(super) fn request(&mut self, ctx: &egui::Context, action: Pending) {
    // ...a quit during an export asks about the export first...
    if self.has_unsaved_changes() || ... {
        self.file.pending = Some(action);
    } else {
        self.perform(ctx, action);
    }
}
```

Mientras `pending` guarda algo, `unsaved_changes_dialog` muestra un `egui::Modal`, una ventana que oscurece todo lo que hay detrás y se queda con todos los clics:



Guardar guarda y después lleva a cabo la acción que esperaba, a menos que guardar fallara o el usuario cancelara el diálogo de Guardar como; No guardar la lleva a cabo inmediatamente; Cancelar, Esc o un clic fuera la olvida. El valor `pending` es toda la máquina de estados: `None` significa «nada esperando», `Some(Pending::Open)` significa «el usuario quiere abrir un archivo, en cuanto decida sobre los cambios».

`pub(super)` hace el enum visible para el módulo padre (`app`), que lo necesita para `FileState`, pero no más allá.

El editor de estilos tiene su propio borrador sin guardar. Salir con un borrador editado pregunta también **¿Guardar los cambios del estilo?**, así que guardar la canción nunca tira el estilo en silencio.

Cerrar la ventana

Hacer clic en la × de la ventana le pide al sistema operativo que la cierre. egui informa de la petición, y la aplicación puede decir que no:

```
let close_requested = ctx.input(|input|
input.viewport().close_requested());
if close_requested
    && !self.file.quitting
    && (self.has_unsaved_changes() ||
self.style_editor.has_unsaved_changes())
{
    ctx.send_viewport_cmd(egui::ViewportCommand::CancelClose);
    self.file.pending = Some(Pending::Quit);
}
```

Cuando el usuario elige entonces Guardar o No guardar, `perform(Pending::Quit)` pone `quitting` y envía `ViewportCommand::Close`, que esta vez se deja pasar.

Salir mientras una exportación WAV o de video aún se está ejecutando pregunta **Una exportación aún se está ejecutando**, con **Detener y salir** y **Continuar exportando**. Detener espera a que la exportación se haya detenido y haya borrado su archivo sin terminar, y luego cierra, así que nunca queda un archivo escrito a medias que parezca terminado.

Atajos de teclado

```
const SAVE: KeyboardShortcut = KeyboardShortcut::new(Modifiers::COMMAND,
Key::S);
const SAVE_AS: KeyboardShortcut =
    KeyboardShortcut::new(Modifiers::COMMAND.plus(Modifiers::SHIFT),
Key::S);

let pressed = |shortcut| ctx.input_mut(|input|
input.consume_shortcut(&shortcut));
if pressed(SAVE_AS) {
    self.save_as();
} else if pressed(SAVE) {
    self.save();
}
```

- `Modifiers::COMMAND` es Ctrl en Linux y Windows y ⌘ en macOS, así que cada sistema tiene el atajo que esperan sus usuarios. El menú Archivo muestra también el correcto: «Ctrl+S» o «⌘S».
- `consume_shortcut` comprueba las teclas y las retira de la entrada, para que un campo de texto no las vea también.
- Ctrl+S también encajaría con Ctrl+Mayús+S (el Mayús de más se ignora), así que el atajo más largo se comprueba primero.

Estos valores `const` llaman a `KeyboardShortcut::new` mientras se compila. Eso está permitido porque egui lo marcó como `const fn`, una función lo bastante simple para ejecutarse en tiempo de compilación.

La barra de título

El título de la ventana muestra el nombre de la canción y un asterisco para los cambios sin guardar, «song.lineup* · Ferrous Lineup». Una demostración muestra su propio nombre y dice que es de solo lectura, «Für Elise (demostración, solo lectura) · Ferrous Lineup»: las demostraciones vienen dentro de la aplicación, así que Guardar pide un archivo nuevo, como Guardar como…. La aplicación envía un `ViewportCommand::Title` solo cuando cambia el texto, no en cada fotograma.

Abrir un archivo desde fuera

```
let mut args = std::env::args_os().skip(1);
let first = args.next();
```

`ferrous-lineup song.lineup` empieza con la canción abierta, que es también lo que hace un gestor de archivos cuando haces doble clic en un archivo asociado a la aplicación. (El mismo primer argumento puede ser en cambio `--trace`, `--demos` o `--help`, que hacen su trabajo sin abrir ninguna ventana; ver La traza de reproducción.) `args_os` devuelve los argumentos como `OsString`s, exactamente como los dio el sistema. `std::env::args` haría panic con un nombre de archivo que no es UTF-8 válido, cosa que Linux permite. `skip(1)` se salta el primer argumento, que es el nombre del propio programa.

Un archivo `.lineup` soltado sobre la ventana también se abre, tras la misma pregunta «¿Guardar cambios?», y un archivo MIDI, una hoja de acordes o una grabación se importan. Los demás archivos WAV soltados van al rack de canales (en Importar grabaciones está cuál WAV es cuál).

Archivo > Abrir recientes

Archivo > Abrir recientes lista los últimos diez proyectos abiertos o guardados, el más reciente primero. La lista se guarda en los datos que egui conserva, así que sobrevive a un reinicio. Cada elemento muestra el nombre del archivo, con la ruta completa en su descripción emergente. Abrir un archivo que ya no carga lo quita de la lista en lugar de dejar una entrada muerta.

La lista guarda rutas, no copias de proyectos. Si un proyecto se mueve o se renombra fuera de la aplicación, la ruta vieja se queda hasta que falla un intento de abrirlo.

Una copia del trabajo sin guardar

La aplicación escribe una copia de recuperación del proyecto mientras tiene cambios sin guardar. El primer cambio escribe una enseguida, para que la primera edición de una sesión no sea la que pierde una caída; después escribe cada treinta segundos:

```
/// How often unsaved work is written while it keeps changing.
const INTERVAL: Duration = Duration::from_secs(30);
```

Una ventana que nadie toca no dibuja fotogramas, así que nunca llegaría a escribir. Por eso, cuando toca una copia, `autosave` le pide a egui un fotograma en ese momento con `request_repaint_after`: la copia se pone al día en más o menos medio minuto después de un cambio, aunque nadie toque la aplicación.

Los archivos están en una carpeta `recovery` dentro de la carpeta de datos de la aplicación, la que le da `eframe::storage_dir(APP_NAME)`. En Linux suele ser `~/.local/share/ferrouslineup/recovery`. Cada ejecución de la aplicación nombra sus archivos según ella misma, como `4211-1793047912612-0.lineup` y `4211-1793047912612-0.info`: el número de proceso, el momento en que empezó en milisegundos, y un contador. Por qué cada ejecución necesita los suyos está en Dos ventanas a la vez más abajo. La copia **no** se escribe junto al proyecto, porque un proyecto puede estar en un medio de solo lectura, y un archivo de autoguardado en una carpeta de canciones sorprendería a cualquiera que la mirase. El pequeño archivo `.info` lleva la ruta y el nombre del proyecto, así que una canción recuperada vuelve como ella misma, no como «Sin título».

La copia se escribe con `file::save`, que escribe un archivo temporal y lo renombra. Una caída a mitad deja entera la copia anterior; el archivo de información se escribe después del proyecto, así que una caída entre los dos pierde el nombre en lugar de darle al proyecto uno equivocado.

Al arrancar, si hay una copia, un diálogo pregunta **¿Recuperar el trabajo sin guardar?** y nombra la canción:

- **Recuperarlo** vuelve a poner el proyecto, con su ruta y su nombre, y borra la copia. También carga el archivo del disco, si aún existe, como la versión «guardada», así que el trabajo recuperado muestra correctamente el asterisco de cambios sin guardar; deshacer hasta el archivo lo quita. Si el archivo ya no está, el proyecto recuperado simplemente no está guardado.
- **Descartarlo** borra la copia y deja el proyecto que la aplicación habría abierto de todos modos.

Una salida limpia borra la copia, porque no hay nada que recuperar. La copia que queda es la señal de una caída, de un cierre forzado o de un corte de luz. Una copia cuyo proyecto no carga se borra al arrancar en lugar de ofrecerse. Un único `recovery.lineup` directamente en la carpeta de datos, como lo escribían versiones anteriores, también se ofrece.

Dos ventanas a la vez

Nada te impide arrancar Ferrous Lineup dos veces. Si cada ejecución escribiera la misma copia, la segunda ventana encontraría al arrancar la copia de la primera y la ofrecería como si la primera se hubiera caído, mientras seguía abierta; y el autoguardado de cada ventana sustituiría la copia de la otra.

Así que cada ejecución escribe su propia copia, y junto a ella un tercer archivo, `<run>.lock`, que **bloquea** antes de escribir la primera copia y mantiene mientras se ejecuta. Un bloqueo de archivo es una promesa que cumple el sistema operativo: solo un archivo abierto puede tenerlo, y cuando el programa que lo tiene termina, el sistema lo suelta, incluso en una caída o un corte de luz. Eso lo convierte en la pregunta correcta al arrancar: «¿sigue en marcha la ejecución que escribió esta copia?». Si su bloqueo se puede tomar, no sigue, y la copia se ofrece; si no, la copia pertenece a una ventana que sigue abierta y se deja en paz.

La biblioteca estándar tiene bloqueos de archivo desde Rust 1.89 (`File::lock`, `File::try_lock` y compañía). `try_lock` no espera: dice enseguida si consiguió el bloqueo, y su error distingue «otro lo tiene» de «este sistema no puede bloquear archivos»:

```
fn take_lock(path: &Path) -> Lock {
    // A copy whose lock file is gone was left by a run that closed with
    // work unsaved: the file is made again, to be taken.
    let Ok(file) = OpenOptions::new()
        .create(true)
        .truncate(false)
        .write(true)
        .open(path)
    else {
        return Lock::Unknown;
    };
    match file.try_lock() {
        Ok(()) => Lock::Taken(file),
        Err(TryLockError::WouldBlock) => Lock::Held,
        Err(TryLockError::Error(_)) => Lock::Unknown,
    }
}
```

El bloqueo dura lo que dura el valor `File`: soltarlo lo libera. Así que una ventana que encuentra una copia guarda el `File` en el valor `Recovered` hasta que respondes a la pregunta; una tercera ventana abierta mientras tanto no puede tomar el bloqueo y no ofrece la misma copia dos veces. Cuando la aplicación se cierra, el `Drop` de `Recovery` borra su archivo de bloqueo; una copia de trabajo sin guardar se queda, sin su bloqueo, para ofrecerse la próxima vez. Un sistema que no puede bloquear archivos (`Lock::Unknown`) ofrece la copia de todos modos: ofrecerla dos veces no pierde nada, no ofrecerla nunca sí.

Si se cayeron dos ejecuciones, hay dos copias. La más nueva se ofrece primero, y la siguiente espera hasta que el trabajo recuperado antes se guarde o se descarte, así que recuperar una nunca sustituye a otra.

Dibujar las ventanas con egui: distribución, zoom, temas, iconos

Archivos: `crates/lineup-app/src/main.rs`, `crates/lineup-app/src/app.rs`, `crates/lineup-app/src/theme.rs`, `crates/lineup-app/src/widgets.rs`, `crates/lineup-app/src/icons.rs`, `crates/lineup-app/src/app/view.rs`, `crates/lineup-app/src/app/shortcuts.rs` y `crates/lineup-app/src/app/about.rs`

La ventana se dibuja con egui, una librería de GUI escrita en Rust, y eframe, el framework de egui para ejecutarla como app nativa en Linux, macOS y Windows. Este capítulo trata de cómo se arma la ventana principal en cada frame, del aspecto, de cómo las filas de controles caben en cualquier tamaño de ventana, de los zooms, de las ventanas que flotan sobre la lista de reproducción y del menú Ayuda. Los editores en sí, el rack de canales, el piano roll y la lista de reproducción, tienen el capítulo siguiente.

Modo inmediato

La mayoría de los toolkits de GUI son de **modo retenido**: creas los objetos botón una vez, los guardas en un árbol y les adjuntas callbacks. egui es de **modo inmediato**: en cada frame, llamas a una función que describe *toda* la interfaz a partir de tus propios datos, y la librería la dibuja y te dice en qué se ha hecho clic.

```
if ui.button("▶ Play").clicked() {
    self.set_playing(true);
}
```

Esa línea dibuja el botón y comprueba si se ha hecho clic en él. No hay objetos botón que mantener sincronizados con tu estado. El estado son simplemente los campos de tu struct, y la interfaz es siempre una imagen de él. Esto encaja bien con Rust, porque no hay callbacks que retengan referencias a tus datos.

El precio es que un frame no puede volver atrás. Algo dibujado al principio del frame no puede saber lo que se dibuja después. Varias piezas de este capítulo (la barra de ayuda, los grupos que se reparten en filas, el zoom) lo resuelven de la misma manera: usan lo que encontró el frame *anterior*, de hace una sexagésima de segundo.

main.rs

```
fn main() -> eframe::Result {
    // The first command-line argument, if there is one, is a file to
open,
    // or what to do instead of opening the window.
    let mut args = std::env::args_os().skip(1);
    let first = args.next();
    match first.as_ref().and_then(|arg| arg.to_str()) {
        Some("--trace") => std::process::exit(trace(args)),
        Some("--demos") => { /* print the demos' names */ return Ok(()); }
        Some("--help" | "-h") => { /* print how to use it */ return
Ok(()); }
        _ => {}
    }
    let path = first.map(PathBuf::from);

    let mut viewport = eframe::egui::ViewportBuilder::default()
        .with_title(app::APP_NAME)
        .with_inner_size([1280.0, 800.0])
        .with_min_inner_size([800.0, 520.0])
        .with_icon(window_icon());
    // ...inside a Flatpak, the window takes the sandbox's app id...

    let options = eframe::NativeOptions {
        viewport,
        ..Default::default()
    };

    eframe::run_native(
        app::APP_NAME,
        options,
        Box::new(|cc| Ok(Box::new(app::LineupApp::new(cc, path)))),
    )
}
```

- `main` devuelve un `Result`. Si devuelve `Err`, Rust imprime el error y sale con un código de fallo.
- `ViewportBuilder::default().with_title(...)` es el **patrón builder**: cada método toma el builder y lo devuelve modificado, para que las llamadas se encadenen.
- `..Default::default()` es la **sintaxis de actualización de structs**: «todo campo que no enumere toma su valor por defecto».
- `Box::new(|cc| ...)` pasa un closure que eframe llama una vez para crear nuestra app. `Box` pone un valor en el **montículo**. Aquí hace falta porque eframe acepta *cualquier* tipo que implemente su trait `App`, y para guardar un valor cuyo tipo exacto no conoce necesita un puntero a él: `Box<dyn App>`. La palabra clave

`dyn` significa «algún tipo que implementa este trait, decidido en tiempo de ejecución» (un **trait object**).
- `"--help" | "-h"` es un **patrón «o»**: la rama coincide con cualquiera de las dos cadenas.

`LineupApp`: el estado

`LineupApp` guarda todo lo que necesita la ventana: el proyecto, el historial de deshacer, las ventanas abiertas y su propio estado, el archivo, los ajustes y la conexión con el hilo de audio. Esa conexión es un enum:

```
enum Audio {
    /// Boxed: the engine is much bigger than a `String`, and every
    /// `Audio` would otherwise take as much room as the biggest variant.
    Running(Box<AudioEngine>),
    Unavailable(String),
}
```

Captura una situación real: o hay un motor en marcha, o está la razón por la que no lo hay. No puedes tener ambas cosas, y no puedes no tener ninguna, así que el tipo no puede representar un estado imposible. Compáralo con la alternativa habitual, un `Option<AudioEngine>` más un `Option<String>`, que permite cuatro combinaciones de las cuales dos no tienen sentido.

Un enum ocupa lo que su variante más grande, más un poco de espacio para decir cuál es. `Box` deja el motor, que es grande, en el montículo, así que `Audio` ocupa lo que un puntero o un `String`, lo que sea mayor. Clippy lo señala con su lint `large_enum_variant`.

Usarlo:

```
fn send(&mut self, command: Command) {
    if let Audio::Running(engine) = &mut self.audio {
        engine.send(command);
    }
}
```

`if let` es un `match` de una sola rama: si `self.audio` es `Running`, liga el motor que hay dentro a `engine` y ejecuta el bloque. Hacer match sobre `&mut self.audio` toma prestado el motor de forma mutable en lugar de moverlo fuera del struct.

Si la computadora no tiene una tarjeta de sonido que funcione, la app se abre igual y la barra de estado dice por qué, en naranja: «Audio no disponible: …». Un fallo se

convierte en datos que mostramos, en lugar de un cierre inesperado.

Un frame

eframe llama a `ui` cada vez que la ventana necesita repintarse:

```
impl eframe::App for LineupApp {
    fn ui(&mut self, ui: &mut egui::Ui, _frame: &mut eframe::Frame) {
        if let Audio::Running(engine) = &mut self.audio {
            engine.collect_garbage();
        }
        self.check_trace();
        let ctx = ui.ctx().clone();
        self.before_drawing(&ctx);
        self.panels(ui, &ctx);
        let playing = self.song_position().filter(|_| self.playing);
        let area = self.main_area(ui, &ctx, playing);

        // The channel rack and the piano roll float on top of the
playlist.
        let playhead = self.pattern_playhead();
        self.bring_window_to_front(&ctx);
        self.rack_window(&ctx, area, playhead);
        self.piano_roll_window(&ctx, area, playhead);
        // The chord sheet is drawn last, so it opens over the others.
        self.chord_sheet_window(&ctx, area, playing);
        // ...ear training, the lead sheet, the mixer, the shortcuts,
        // About, the style editor, Settings and the dialogs...
        self.unsaved_changes_dialog(&ctx);
        self.update_title(&ctx);
        self.end_frame_for_undo(&ctx);
    }

    fn on_exit(&mut self) {
        self.forget_recovery();
    }
}
```

El orden es el diseño:

1. **`collect_garbage`** libera lo que devolvió el hilo de audio (ver La cola de basura).
2. **`before_drawing`** hace todo lo que debe pasar antes de dibujar nada: apunta si un clic está cerrando un menú, pone el idioma y los nombres de las notas para el frame, cierra un paso de deshacer al pulsar el ratón, escribe la copia de recuperación cuando toca, lee los atajos de teclado y las teclas de zoom, lee la entrada MIDI y mira cómo van las exportaciones e importaciones que corren en segundo plano. Mientras suena, le pide a egui otro frame enseguida

(`ctx.request_repaint()`), para que el cursor de reproducción se mueva; si no, egui solo dibuja cuando pasa algo.
3. `panels` dibuja las barras de los bordes: los menús con la barra de ayuda, la barra de herramientas, la barra de estado y las teclas de armonía. Los paneles deben ir antes del área central, que ocupa el espacio que queda.
4. `main_area` dibuja la lista de reproducción (o la hoja de acordes, cuando llena la ventana principal) y devuelve el rectángulo que ocupó.
5. **Las ventanas** flotan sobre ese rectángulo. Las que se dibujan después quedan encima, por eso la hoja de acordes va la última de los tres editores.
6. `end_frame_for_undo` decide si el paso de deshacer termina (ver Deshacer y rehacer).

`on_exit` se ejecuta una vez cuando la app se cierra bien: no hay trabajo sin guardar que recuperar, así que la copia de recuperación de esta ejecución se borra.

Los muchos métodos `self.algo(ui)` mantienen `ui` corto. Cada closure que se pasa a un panel puede tomar prestado `self` de forma mutable, porque cada uno se ejecuta y termina antes de que empiece el siguiente, algo que el borrow checker puede ver.

Un tema es una función

egui dibuja cada widget a partir de un `Style`: los colores en `style.visuals`, los tamaños y huecos en `style.spacing`. Nada se pinta con un color fijo salvo que el código pida uno. Así que todo el aspecto es una función, `theme::apply`, llamada una vez al arrancar la app:

```
pub fn apply(ctx: &egui::Context) {
    ctx.set_theme(egui::Theme::Dark);
    ctx.style_mut_of(egui::Theme::Dark, |style| {
        let visuals = &mut style.visuals;
        visuals.panel_fill = PANEL;
        visuals.window_fill = WINDOW;
        visuals.selection.bg_fill = ACCENT; // selected and switched-on
things
        visuals.selection.stroke = Stroke::new(1.0, ON_ACCENT);
        // ...shadows, corners, warning and error colours...
        let widgets = &mut visuals.widgets;
        for (state, fill) in [
            (&mut widgets.inactive, BUTTON),
            (&mut widgets.hovered, BUTTON_HOVER),
            (&mut widgets.active, BUTTON_PRESSED),
            (&mut widgets.open, BUTTON_HOVER),
        ] {
            state.bg_fill = fill;
            state.weak_bg_fill = fill;
            state.bg_stroke = Stroke::new(1.0, OUTLINE);
            // ...
        }
        // ...spacing...
    });
}
```

El aspecto toma prestado de los estudios basados en patrones: paneles gris azulado oscuro, botones grises en relieve, una luz naranja para lo que está seleccionado o encendido, y pantallas «LCD» negras para números como el tempo. Tres cosas que hay que notar en el código:

- `style_mut_of(Theme::Dark, ...)` cambia solo el estilo *oscuro*. egui guarda un estilo para oscuro y otro para claro, y la app usa siempre el oscuro.
- El bucle `for` recorre un **array de pares de referencias mutables**. Cada `&mut widgets.inactive` toma prestado un *campo distinto*, así que Rust permite varios a la vez: el borrow checker sigue los campos de un struct por separado.
- Los colores son elementos `const`: `Color32::from_rgb` es un `const fn`, así que se ejecuta en tiempo de compilación y no cuesta nada cuando la app se ejecuta.

Las vistas siguen teniendo sus propias constantes para las cosas que solo dibujan ellas (las filas de la lista de reproducción, las teclas del piano roll, los botones de paso), con valores que pegan con el tema.

Cambiar el estilo de unos pocos widgets

La pantalla de tempo es un `DragValue`, el número de egui que se arrastra o se escribe, pero dibujado en negro con dígitos verde azulados. Su estilo se cambia solo

dentro de su marco:

```
pub fn lcd_frame(
    ui: &mut Ui,
    label: &str,
    width: f32,
    add: impl FnOnce(&mut Ui) -> Response,
) -> Response {
    wrapping(ui, |ui| {
        egui::Frame::new()
            .fill(LCD)
            // ...
            .show(ui, |ui| {
                // ...the label above, written smaller if it is long...
                ui.vertical(|ui| {
                    ui.label(label);
                    let visuals = &mut ui.visuals_mut().widgets;
                    for state in [&mut visuals.inactive, &mut
visuals.hovered, &mut visuals.active] {
                        state.bg_fill = LCD;
                        state.fg_stroke.color = LCD_TEXT;
                        // ...
                    }
                    ui.style_mut().override_font_id =
Some(FontId::monospace(16.0));
                    add(ui)
                })
                .inner
            })
    })
    .inner
}
```

Cada `Ui` tiene su propia copia del estilo (un `Arc<Style>`; `style_mut` hace una copia privada la primera vez que se cambia). El contenido del marco recibe un `Ui` hijo, así que el cambio se queda dentro del marco y el resto de la barra de herramientas no se toca. `add` es un closure que *devuelve* el `Response` del widget, así que quien lo llama puede seguir preguntando `changed()` fuera:

```
let tempo = theme::lcd_frame(ui, tr("TEMPO"), 62.0, |ui| {
    ui.add(egui::DragValue::new(&mut bpm).range(Tempo::MIN..=Tempo::MAX))
});
```

El campo de `Tempo` es privado, así que el valor se copia a un `bpm` local, el widget edita ese, y se construye un `Tempo` nuevo si cambió. El `Tempo` sigue siendo válido se escriba lo que se escriba. `Tempo::MIN..=Tempo::MAX` es un **rango inclusivo**: `a..b` deja fuera `b`, `a..=b` lo incluye.

Una etiqueta más larga que la pantalla, como algunas en español, se escribe un poco más pequeña para que quepa, así que la barra de herramientas conserva su ancho en todos los idiomas.

La barra de ayuda: un mensaje del futuro

Junto a los menús, una línea dice qué hace el control que está bajo el ratón. Esa línea se dibuja *primero*, arriba del todo de la ventana, pero los widgets que saben sobre qué está el ratón se dibujan *después*, en el mismo frame. En modo inmediato no hay vuelta atrás para repintar la barra.

Así que la barra muestra lo que se encontró en el frame **anterior**. Los widgets dejan su ayuda en la memoria de egui, `ctx.data`, que es un mapa de un `Id` a un valor de cualquier tipo:

```
pub fn hint(response: Response, text: impl Into<String>) -> Response {
    let text = text.into();
    if response.hovered() {
        response
            .ctx
            .data_mut(|data| data.insert_temp(hint_id(), text.clone()));
    }
    response.on_hover_text(text)
}
```

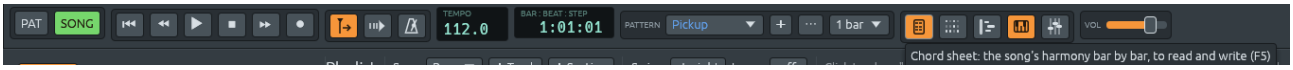
Al principio del frame siguiente, la barra la saca, para que no se quede cuando el ratón se va:

```
pub fn start_frame(&mut self, ctx: &egui::Context) {
    let text = ctx.data_mut(|data| data.remove_temp::<String>(hint_id()));
    if text != self.text {
        // The mouse moved onto or off a widget: draw once more, so the
        // bar does not lag a frame behind until the next event.
        ctx.request_repaint();
    }
    self.text = text;
}
```

`remove_temp::<String>` usa el **turbofish** `::<String>` para decir qué tipo sacar: el mapa puede guardar valores de muchos tipos bajo el mismo `Id`. El `request_repaint` es la parte sutil. egui solo dibuja un frame cuando pasa algo. Cuando el ratón se pone sobre un botón, ese frame apunta la ayuda, pero la barra sigue mostrando la antigua. Pedir un frame más muestra la ayuda nueva enseguida, en lugar de la próxima vez que se mueva el ratón.

Filas que caben en cualquier ventana

La barra de herramientas tiene el modo de reproducción, el transporte, las pantallas de tempo y de posición, el selector de patrones, los cinco botones de ventana, la cuenta de entrada, el metrónomo y el volumen maestro. En una ventana de 1280 puntos de ancho cabe en una fila; una prueba, `the_toolbar_fits_on_one_row_at_1280_points`, comprueba que siga siendo así. Una ventana más estrecha o con zoom tiene que seguir llegando a todos los controles.



egui coloca los widgets uno detrás de otro. `ui.horizontal` los pone en una fila y sigue más allá del borde si no caben: lo que queda más allá del borde sigue estando, solo que no se dibuja. `ui.horizontal_wrapped` es la misma fila, salvo que un widget que no cabría empieza una fila nueva. La barra de herramientas y la barra de teclas de armonía son filas que se reparten, así que en una ventana estrecha su segunda parte baja.

Grupos que se reparten: el ancho del frame pasado

Los botones de la barra de herramientas van en grupos, cada uno dentro de un marco oscuro. Una fila que se reparte decide si pasa a la fila siguiente preguntándole a un widget cuánto quiere ocupar antes de colocarlo. Un marco no puede responder a eso: `Frame::show` dibuja su contenido en el espacio que quede, y solo sabe su tamaño después. Así que `theme::wrapping` usa el frame anterior. egui vuelve a dibujar la ventana muchas veces por segundo, así que el ancho de la última vez es una muy buena estimación:

```
let id = ui.next_auto_id().with("wrapping");
let last_width = ui.ctx().data(|data| data.get_temp::<f32>(id)).or(guess);
let row_started = ui.cursor().min.x > ui.max_rect().min.x;
if let Some(width) = last_width
    && ui.layout().main_wrap
    && row_started
    && width > ui.available_size_before_wrap().x
{
    ui.end_row();
}
let response = add(ui);
let width = response.response.rect.width();
ui.ctx().data_mut(|data| data.insert_temp(id, width));
```

- `ui.next_auto_id()` es el id que egui le daría al siguiente widget. Depende de cuántos widgets vinieron antes en este `Ui`, así que es el mismo en cada frame mientras la barra se dibuje en el mismo orden. Eso lo convierte en una clave para recordar algo de un grupo.
- `get_temp::<f32>` le pide a la memoria de egui un `f32`; el turbofish le dice a Rust qué tipo buscar, porque nada más en la línea lo dice.
- **La cadena** `if let ... && ...` (una «let chain») solo ejecuta `end_row` cuando hay un ancho de la vez anterior, la distribución se reparte, la fila ya tiene algo y el grupo no cabe. El primer frame de todos todavía no tiene ancho, así que el grupo puede sobresalir durante un frame, cosa que nadie ve.

`theme::group` y `lcd_frame` envuelven sus marcos en él, y `keep_together` ofrece lo mismo a cualquier grupo de widgets, con un `guess` para el primer frame de todos. Ese primer frame importa dentro de una ventana: las ventanas de egui crecen para que quepa su contenido, y **un solo frame de contenido demasiado ancho basta para ensanchar una ventana para siempre**. La barra de herramientas del piano roll mantiene juntos sus grupos de esta manera, así que un nombre de escala largo (el de Fa# mayor, «Fa# Sol# La# Si Do# Re# Mi#») pasa su grupo a una segunda fila en lugar de ensanchar la ventana.

Texto que se acorta, y de derecha a izquierda

Repartirse en varias filas les va bien a los botones. Para una línea larga de texto, como un mensaje de estado, a menudo es mejor cortarla. Todo texto en egui tiene un *modo de ajuste*: `Wrap` lo parte en más líneas, `Truncate` lo corta con «…» y `Extend` lo deja seguir. Un área entera puede ponerlo para todo su texto mediante su estilo:

```
ui.style_mut().wrap_mode = Some(egui::TextWrapMode::Truncate);
```

Una etiqueta truncada muestra el texto entero cuando apuntas a ella, así que no se pierde nada.

La barra de estado tiene dos partes: mensajes a la izquierda y los controles de zoom de la lista de reproducción a la derecha. Si los mensajes se dibujaran primero, uno largo («Audio no disponible: no se pudieron leer los ajustes del dispositivo…») ocuparía todo el espacio y empujaría fuera el zoom. Así que la barra se distribuye *de derecha a izquierda*: el zoom entra primero, desde el borde derecho, y los mensajes se quedan con lo que sobra:

```
fn status_bar(&self, ui: &mut egui::Ui) {
    ui.with_layout(egui::Layout::right_to_left(egui::Align::Center), |ui|
{
        view::zoom_control(ui);
        ui.separator();
        ui.with_layout(egui::Layout::left_to_right(egui::Align::Center),
|ui| {
            ui.style_mut().wrap_mode = Some(egui::TextWrapMode::Truncate);
            self.status_messages(ui);
        });
    });
}
```

En una distribución de derecha a izquierda, el primer widget es el que está más a la derecha, por eso `zoom_control` añade su botón **+** antes del porcentaje y del botón **−**.

Zoom

Hay dos tipos de zoom.

El **tamaño de la interfaz** hace todo más grande o más pequeño: texto, botones, menús. egui dibuja en *puntos*, no en píxeles. Cuántos píxeles ocupa un punto es `pixels_per_point`: la escala propia de la pantalla (1.0, 1.5 o 2.0 en la mayoría) multiplicada por el *factor de zoom* de egui. `ctx.set_zoom_factor` cambia el segundo.

Las **áreas** hacen zoom de lo que dibujan sin cambiar el texto. Como en la mayoría de los programas de música, la lista de reproducción y el piano roll hacen zoom en sus dos direcciones por separado: el zoom horizontal hace más anchos los compases y los pasos (menos tiempo en pantalla, con más detalle), el vertical hace más altas las pistas y las teclas del piano. Los pasos del rack de canales son botones cuadrados, así que tiene un solo zoom para las dos direcciones.

Puntero sobre	Ctrl y + / − (o rueda)	Alt y + / − (o rueda)	Ctrl+0
Lista de reproducción	compases más anchos / estrechos	pistas más altas / bajas	las dos al 100%
Piano roll	pasos más anchos / estrechos	teclas más altas / bajas	las dos al 100%
Rack de canales	más grande / pequeño	lo mismo	100%

Puntero sobre	Ctrl y + / − (o rueda)	Alt y + / − (o rueda)	Ctrl+0
Cualquier otra cosa, también la hoja de acordes	interfaz más grande / pequeña	lo mismo	100%

¿Qué hay bajo el puntero?

Las teclas y la rueda hacen zoom de lo que está bajo el puntero. En Rust, «una de estas clases» es un `enum`, y una variante puede llevar un valor:

```
pub(crate) enum Target {
    Area(Area), // Area::Main (the playlist), Area::Rack, Area::PianoRoll
    Interface,
}
```

`Target::at` lo encuentra. egui mantiene una pila de *capas*: el fondo, donde se dibujan los paneles, luego cada ventana encima, luego los menús y los tooltips. `ctx.layer_id_at(pos)` dice qué capa está arriba en un punto:

```
fn at(ctx: &egui::Context, pos: egui::Pos2, playlist: egui::Rect) ->
Target {
    match ctx.layer_id_at(pos) {
        Some(layer) if layer.id == super::rack_id() =>
Target::Area(Area::Rack),
        Some(layer) if layer.id == PianoRoll::id() =>
Target::Area(Area::PianoRoll),
        Some(layer) if layer.order == egui::Order::Background &&
playlist.contains(pos) => {
            Target::Area(Area::Main)
        }
        _ => Target::Interface,
    }
}
```

Cada rama tiene una **guarda**, el `if` después del patrón. El patrón `Some(layer)` coincide con cualquier capa, y la rama solo se toma si la guarda también es verdadera; si no, `match` prueba la rama siguiente. La última rama, `_`, recoge todo lo que queda.

Las barras de herramientas, las teclas de armonía y la lista de reproducción se dibujan todas en la misma capa de fondo, así que la capa sola no puede distinguirlas. Por eso la lista de reproducción recuerda dónde se dibujó, y la guarda comprueba `playlist.contains(pos)`. Las teclas de zoom se leen al principio de un frame, antes

de dibujar nada, así que este es el rectángulo del frame anterior. Si el puntero está fuera de la ventana, las teclas hacen zoom de lo último en que se hizo clic.

La hoja de acordes es texto, así que sigue el tamaño de la interfaz, en su propia ventana o en el lugar de la lista de reproducción. En ese lugar tapa la lista de reproducción, y el último rectángulo de la lista mentiría, así que `zoom_shortcuts` pasa uno vacío en su lugar:

```
let playlist = match self.chord_sheet.fills_window() {
    true => egui::Rect::NOTHING,
    false => self.playlist.area(),
};
```

`Rect::NOTHING` no contiene ningún punto, así que la guarda nunca es verdadera, y las teclas caen en `Target::Interface`.

Todas las maneras de hacer zoom (teclas, rueda, pellizco en el touchpad) llaman a los mismos pocos métodos de `Target`, y solo ellos miran de qué clase es:

```
fn step(self, ctx: &egui::Context, axis: Axis, bigger: bool) {
    match self {
        Target::Area(area) => area.step_zoom(ctx, axis, bigger),
        Target::Interface => {
            ctx.set_zoom_factor(interface_step(ctx.zoom_factor(),
bigger));
        }
    }
}
```

`Target::Area(area)` comprueba la variante y a la vez saca de ella el `Area` a la variable `area`. La interfaz tiene un solo tamaño, así que ahí `axis` no se usa: Ctrl y Alt hacen lo mismo.

Un zoom por área y por eje

Cada área guarda un número por dirección, en la memoria de egui, bajo una clave hecha de las dos cosas:

```
fn key(self, axis: Axis) -> egui::Id {
    let axis = if self.two_way() { axis } else { Axis::Horizontal };
    egui::Id::unique(("zoom-2", self, axis))
}
```

El rack no es `two_way`, así que sus dos ejes comparten la clave horizontal: hacerle zoom «vertical» le hace zoom en las dos direcciones. Los valores se guardan con

`insert_persisted`, así que egui los guarda con las posiciones de las ventanas y son los mismos la próxima vez que arranca la app. El nombre `"zoom-2"` está porque un significado anterior de «zoom» (una escala para todo lo de la ventana, texto incluido) se guardaba bajo `"zoom"`: un nombre nuevo hace que los valores viejos simplemente no se encuentren, en lugar de leerse de la manera equivocada.

Todas las maneras de hacer zoom por pasos pasan por `step_zoom`, que recorre una escalera de tamaños:

```
const STEPS: [f32; 19] = [
    0.25, 0.33, 0.4, 0.5, 0.6, 0.75, 0.9, 1.0, 1.1, 1.25, 1.5, 1.75, 2.0,
2.5, 3.0, 4.0, 5.0, 6.0,
    8.0,
];
```

Cada paso es más o menos una quinta parte mayor que el anterior, así que cuesta tantos pasos ir de 100% a 200% como de 400% a 800%. Así es como el zoom se siente natural: los ojos comparan tamaños por proporción, no por diferencia. Cada área y eje tiene su propio tramo dentro de la escalera (las pistas y las teclas solo pueden achicarse hasta que sus nombres dejan de caber), y los botones se ponen grises en los extremos. Los tamaños de la interfaz son una lista más corta propia, del 50% al 250%, los mismos que ofrece **Ajustes > Apariencia**.

Leer la rueda nosotros mismos

egui da un único `zoom_delta()` para Ctrl+rueda, sin idea de dirección. Así que la app lee los eventos de rueda en bruto del frame y los clasifica según las teclas modificadoras pulsadas:

```
for event in &input.raw.events {
    if let egui::Event::MouseWheel { unit, delta, modifiers, .. } = event
{
        // Up or right zooms in. Shift turns a wheel sideways.
        let amount = if delta.y != 0.0 { delta.y } else { delta.x };
        if modifiers.alt {
            tall.add(*unit, amount);
        } else if modifiers.command {
            wide.add(*unit, amount);
        }
    }
}
```

`if let` con un patrón es como Rust elige una variante de un enum y nombra sus campos al mismo tiempo; `..` ignora los campos que no necesitamos.

En Linux (X11), un golpe de rueda a veces llega como dos eventos, y hacer zoom de dos pasos por golpe se siente a saltos. Así que los golpes se suman y hacen zoom de un paso como mucho por frame:

```
fn apply(self, target: Target, ctx: &egui::Context, axis: Axis) {
    if self.notches != 0.0 {
        target.step(ctx, axis, self.notches > 0.0);
    }
    if self.smooth != 1.0 {
        target.scale(ctx, axis, self.smooth);
    }
}
```

Un frame dura unos pocos milisegundos, así que girar la rueda rápido sigue haciendo zoom paso a paso. Un touchpad manda en cambio muchos movimientos diminutos en lugar de golpes (`MouseWheelUnit::Point`); esos multiplican el zoom de forma suave.

Hacer zoom sobre lo que señalas

Cuando el zoom cambia, el paso y la nota que están bajo el puntero deben seguir bajo el puntero. La cuadrícula está dentro de un scroll area, así que eso significa elegir un desplazamiento nuevo. Lo que está bajo el puntero está en `from + anchor` dentro del contenido, donde `from` es cuánto se ha desplazado la vista y `anchor` dónde está el puntero dentro de la vista. Hacer zoom por `ratio` lo mueve a `(from + anchor) * ratio`. Para que se quede bajo el puntero, el desplazamiento nuevo debe ser eso menos `anchor`. En el piano roll:

```
let x = KEY_WIDTH + (from.x + anchor.x - KEY_WIDTH) * ratio.x - anchor.x;
let y = (from.y + anchor.y) * ratio.y - anchor.y;
self.scroll_to = Some(vec2(x.max(0.0), y.max(0.0)));
```

Los términos `KEY_WIDTH` están porque el teclado de piano de la izquierda no se estira: solo crece lo que está a su derecha.

Hubo que apagar una cosa más. Los scroll areas de egui tienen una **banda elástica**: te desplazas más allá del final y la vista vuelve de un salto. En el frame de un zoom, el scroll area todavía conoce el tamaño *viejo* del contenido, así que el desplazamiento nuevo parece ir más allá del final, y la vista se deslizaría de vuelta después de cada zoom. `widgets::no_rubber_band` la apaga para las cuadrículas:

```rust
pub fn no_rubber_band(ui: &mut egui::Ui) {
    ui.spacing_mut().scroll.kinetic.rubber_band = false;
}
```



El rack de canales se dibuja con widgets normales de egui, que no tienen zoom propio. `widgets::zoom_contents` los hace más grandes multiplicando los tamaños del texto y los espacios del estilo de la ventana, igual que `lcd_frame` cambia sus colores: solo dentro del rack.

Ventanas sobre la lista de reproducción

El rack de canales, el piano roll, la hoja de acordes, el mezclador y las demás son `egui::Window`, que flotan sobre todo. Se mantienen dentro del rectángulo de la lista de reproducción con `Window::constrain_to(area)`, así que una ventana nunca tapa la barra de herramientas, las teclas de armonía ni la barra de estado:

```rust
let playlist = egui::CentralPanel::default_margins().show(ui, |ui| { /*
... */ });
let area = playlist.response.rect;
```

Cada `show` de egui devuelve un `InnerResponse`: `.inner` es lo que devolvió el closure (aquí las acciones de la lista de reproducción) y `.response` dice dónde se dibujó.

Con qué ventanas se abre una canción

Lo que estudias en una canción son sus compases y acordes; lo que editas en un ritmo son los pasos de un patrón. Así que la app decide qué mostrar según lo que se abrió:

```
/// What a song opens on: the chord sheet in front, since its bars and
/// chords are what you study and write in a song, and the rack out of
/// the way (F6 brings it back).
fn show_chord_sheet(&mut self) {
    self.chord_sheet.open = true;
    self.rack_open = false;
    self.bring_to_front = Some(chord_sheet::window_id());
}

/// What a beat opens on: the rack in front, and no chord sheet.
fn show_beat(&mut self) {
    self.chord_sheet.open = false;
    self.show_rack();
}
```

Las demostraciones de canciones, **Archivo > Nuevo** y una hoja de acordes importada llaman a `show_chord_sheet`; **Archivo > Nuevo ritmo** llama a `show_beat`. Abrir un archivo guardado deja las ventanas como están, porque las elegiste tú. Una demostración es una canción o un estilo, un `enum`, y solo las canciones se abren en la hoja:

```
if let Demo::Song(_) = demo {
    self.show_chord_sheet();
}
```

La aplicación misma se abre con las ventanas que estaban abiertas la última vez que se cerró. `app/windows.rs` las guarda como un struct pequeño de `bool`, uno por cada ventana del menú Ver:

```
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
pub(super) struct OpenWindows {
    chord_sheet: bool,
    lead_sheet: bool,
    rack: bool,
    piano_roll: bool,
    harmony_keys: bool,
    /// The Patterns and Library panel.
    browser: bool,
    mixer: bool,
    ear_training: bool,
}
```

Su `Default` es la primera vez: la hoja de acordes, las teclas de armonía y la Biblioteca sobre la lista de reproducción, sin rack. `#[serde(default)]` en el struct rellena una ventana añadida más tarde a la lista con su valor en `Default`, así que una copia guardada más antigua se sigue leyendo. El editor de estilos y los diálogos quedan fuera: cada uno se abre para algo, un estilo o una pregunta, que la siguiente ejecución no tiene.

`App::save`, a la que eframe llama cada 30 segundos y cuando se cierra la ventana, escribe el struct entre las preferencias guardadas por nombre (ver Guardado por nombre, no por tipo), y `LineupApp::new` lo vuelve a leer:

```
fn save(&mut self, storage: &mut dyn eframe::Storage) {
    self.preferences.put(windows::NAME, &self.open_windows());
    self.preferences.save(storage);
}
```

```
let windows: OpenWindows = preferences::get(ctx,
NAME).unwrap_or_default();
```

Escribirlas solo al guardar, en lugar de a cada clic en un botón de ventana, libra de esa tarea a los muchos lugares que abren y cierran ventanas.
`the_app_first_opens_on_the_chord_sheet_then_with_the_windows_left_open` comprueba las dos ejecuciones: la primera se abre en la hoja, y después de cerrarla con F5 y abrir el mezclador con F9, el siguiente arranque encuentra el mezclador abierto y la hoja cerrada.

Ventanas del tamaño de lo que muestran

egui recuerda el tamaño de cada ventana y lo guarda con el resto de su memoria cuando la app se cierra. Eso está bien para una ventana en la que trabajas, como el piano roll: la hiciste grande porque la querías grande. Está mal para una ventana que es tan grande como lo que muestra: arrastra el mezclador a lo ancho una vez, y desde entonces se abriría por toda la lista de reproducción, con sus canales en una esquina. Así que hay dos tipos de ventana:

```
pub enum Opening {
    /// Each time it opens: a window as big as what it shows.
    EveryTime,
    /// The first time it opens after the app starts: a window to work
    /// in, which keeps the size you give it while the app runs.
    FirstTime,
}
```

El mezclador, la lista de atajos, el entrenamiento auditivo, la serie dodecafónica, los conjuntos de clases de alturas y los micromodos se abren **cada vez** al tamaño de su contenido, hasta la altura de la lista de reproducción, y hacen scroll más allá. El piano roll, la hoja de acordes, la partitura guía y Ajustes se abren a su propio tamaño la **primera vez** después de que la app arranca, y conservan el tamaño que les des mientras se ejecuta.

`widgets::open_at` lo hace. Averigua si la ventana se está abriendo ahora (no se dibujó en el último frame) y, si es así, le da un tamaño fijo para ese único frame:

```
let first = ctx.data_mut(|data| {
    !std::mem::replace(data.get_temp_mut_or(id.with("has-opened"), false),
true)
});
if opens(ctx, id) && (first || opening == Opening::EveryTime) {
    ctx.request_discard("a window opens at its own size");
    window.fixed_size(size)
} else {
    window
}
```

Tres cosas en él merecen una mirada:

- `std::mem::replace(place, new)` pone `new` en `place` y devuelve lo que había. Aquí lee «¿se había abierto antes esta ventana?» y la marca como abierta, en un solo paso. La bandera se guarda con `get_temp_mut_or`, que egui no guarda, así que vuelve a ser `false` cada vez que la app arranca.
- egui acomoda una ventana a medida que la dibuja, así que en su primer frame la ventana se dibuja antes de que nada sepa cuán grande es su contenido. `request_discard` le pide a egui que tire ese frame y lo dibuje otra vez; solo ves el segundo, ya al tamaño correcto.
- El «tamaño» dado es un ancho estrecho y la mayor altura. egui agranda una ventana para que quepa un contenido más ancho, pero nunca la hace más alta que él, así que la ventana acaba exactamente tan grande como lo que muestra. Para que eso funcione, nada en ella puede llenar todo el espacio que se le da, como un scroll area que no se encoge hasta su contenido.

Una ventana cuyo contenido cambia mientras está abierta, como el mezclador cuando se añade un canal o el rack cuando otro patrón tiene menos canales, se ajusta otra vez con `fit_when_changed`. Calcula el hash de un resumen del contenido (el nombre y el instrumento de cada canal, por ejemplo) y ajusta la ventana otra vez cuando el hash cambia.

Clics que hacen lo que querías decir

Dos ayudantes de `widgets.rs` hacen que las cuadrículas respondan al clic que querías hacer.

Un toque rápido. En un touchpad, un toque puede pulsar y soltar el botón dentro de un mismo frame. El `press_origin()` de egui se olvida en cuanto el botón está arriba, así que en ese frame es `None`. `press_pos` lee en cambio la pulsación de la lista de eventos del frame, que además dice *dónde* bajó el botón, no dónde está el puntero al final del frame:

```
pub fn press_pos(
    input: &egui::InputState,
    only: Option<egui::PointerButton>,
) -> Option<egui::Pos2> {
    input.events.iter().find_map(|event| match *event {
        egui::Event::PointerButton { pos, button, pressed: true, .. }
            if only.is_none_or(|only| only == button) => Some(pos),
        _ => None,
    })
}
```

`find_map` recorre los eventos hasta que el closure devuelve `Some`, y devuelve eso. `Option::is_none_or` se lee bien aquí: «no se pidió ningún botón, o es este».

Un clic que cierra un menú. Abre una lista desplegable y haz clic en la cuadrícula para cerrarla: egui cierra el menú, pero el clic también llega a la cuadrícula, que dibujaría una nota, colocaría un clip o encendería un paso. En la mayoría de los programas ese clic solo cierra el menú. Así que al principio de cada frame, antes de dibujar nada y mientras el menú que se va a cerrar sigue abierto, la app recuerda si el botón que ahora está abajo bajó mientras había un menú abierto:

```
pub fn track_dismissing(ctx: &egui::Context) {
    let id = dismissing_id();
    let (pressed, down, released) = ctx.input(|input| {
        (
            press_pos(input, None).is_some(),
            input.pointer.any_down(),
            input.pointer.any_released(),
        )
    });
    let open = egui::Popup::is_any_open(ctx);
    ctx.data_mut(|data| {
        let dismissing = data.get_temp_mut_or(id, false);
        if pressed {
            *dismissing = open;
        } else if !down && !released {
            *dismissing = false;
        }
    });
}
```

Las cuadrículas preguntan `dismissing(ctx)` e ignoran la pulsación, el arrastre y la soltada que le pertenecen. El valor vive en la memoria temporal de egui, un lugar cómodo para un poco de estado compartido entre widgets que no se conocen entre sí.

Recordar entre ejecuciones

egui guarda algo de estado propio entre frames en su *memoria*: el factor de zoom, dónde está cada ventana, cuán ancho es un panel que se puede redimensionar. eframe puede guardar esa memoria cuando la app se cierra y cargarla cuando arranca. Solo lo hace con su feature `persistence`, que viene apagada por defecto porque trae `serde` y `ron`. Encenderla es una línea en el `Cargo.toml` del workspace:

```
eframe = { version = "0.37", features = ["persistence"] }
```

Con ella, eframe recuerda también el tamaño y la posición de la ventana principal. Escribe un archivo llamado `app.ron` en el lugar donde cada sistema guarda los datos de las apps (en Linux, `~/.local/share/ferrouslineup/`), cada 30 segundos y cuando se cierra la ventana.

La app guarda la mayoría de sus propias elecciones duraderas en la misma memoria, con `insert_persisted` en lugar de `insert_temp`: los zooms de las áreas, los ajustes, los archivos recientes, los filtros de la biblioteca. Cada módulo guarda lo suyo bajo un

`Id` propio, y eframe los guarda todos juntos. Lo que se guarda con `insert_temp` se olvida al cerrar la app.

Guardado por nombre, no por tipo

egui encuentra un valor en su memoria por dos cosas: el `Id` bajo el que se guardó y su tipo de Rust, a través del `TypeId` del tipo. Un `TypeId` es un número que el compilador calcula a partir de la ruta del tipo, sus campos y el crate en el que está, y la parte del crate cambia con la versión del crate y con la versión del compilador. Así que después de una actualización de la app, o de compilar con un Rust más nuevo, o de añadir un campo al tipo, `get_persisted` busca bajo un número nuevo y no encuentra nada: el ajuste vuelve a su valor por defecto, y el valor viejo queda sin leer en `app.ron`. El SoundFont lo mostró: con el mismo `app.ron`, la versión 0.4.0 tocaba con GeneralUser GS y una compilación numerada 0.4.1 lo tenía desactivado.

eframe tiene un almacenamiento propio, el mismo `app.ron`, que encuentra los valores por un nombre: un `&str` que no cambia sea cual sea la versión. eframe se lo da a la app solo dos veces: como `cc.storage` cuando la app arranca, y a `App::save`, cada 30 segundos y al cerrar la ventana. El módulo `app/preferences.rs` une las dos cosas. Guarda cada preferencia por nombre, escrita como texto RON, en un único mapa:

```
/// Each preference by name, written as RON. egui's context holds one
/// handle to the map and the app another, to save it.
#[derive(Clone, Default)]
pub(super) struct Preferences(Arc<Mutex<BTreeMap<String, String>>>);
```

Un acceso al mapa vive en la memoria temporal de egui, así que cualquier código con el `Context` llega a él, como a cualquier otro valor guardado ahí:

```
/// The preference kept as `name`, if there is one and it can be read.
pub(super) fn get<T: DeserializeOwned>(ctx: &egui::Context, name: &str) ->
Option<T> {
    let text = Preferences::of(ctx).map().get(name).cloned()?;
    ron::from_str(&text).ok()
}

/// Keeps `value` as the preference `name`.
pub(super) fn set<T: Serialize>(ctx: &egui::Context, name: &str, value:
&T) {
    if let Ok(text) = ron::to_string(value) {
        Preferences::of(ctx).map().insert(name.to_string(), text);
    }
}
```

La app lee el mapa del almacenamiento de eframe antes de crear nada más,

```
if let Some(storage) = cc.storage {
    preferences::Preferences::of(&cc.egui_ctx).load(storage);
}
```

guarda otro acceso a él, y se lo devuelve a eframe para que lo escriba:

```
fn save(&mut self, storage: &mut dyn eframe::Storage) {
    self.preferences.save(storage);
}
```

El `Arc<Mutex<…>>` deja que los dos accesos compartan un mismo mapa. El bloqueo se mantiene solo mientras se lee o se escribe una línea, y solo lo toma el hilo de la ventana. Como los valores son texto RON hasta que se piden, un tipo puede cambiar sus campos y seguir leyendo lo que guardó una versión anterior, con `#[serde(default)]` para los campos que todavía no existían.

Los ajustes de sonido (SoundFonts y General MIDI) se guardan así, como `"sound-font"`, y también las ventanas abiertas, como `"open-windows"`. Las demás elecciones de arriba siguen en la memoria de egui, así que una actualización puede olvidarlas; pasar una al nuevo sistema es cuestión de llamar a `preferences::get` y `preferences::set` con un nombre en lugar de `get_persisted` e `insert_persisted` con un `Id`.

Features de Cargo

Una *feature* es un interruptor que ofrece un crate para encender código opcional. Dentro de eframe, el código que guarda la memoria está detrás de `#[cfg(feature = "persistence")]`, así que sin la feature ni siquiera se compila. Encender una feature en un crate puede encender features en otros: la `persistence` de eframe enciende la de egui, que añade derives de `serde` a los tipos que egui guarda en su memoria. Puedes ver lo que ofrece un crate en la tabla `[features]` de su `Cargo.toml`, o en su página de docs.rs.

Dibujos pintados, no escritos

Los cinco botones de ventana de la barra de herramientas y los interruptores de cuenta de entrada y metrónomo son botones cuadrados con un dibujo; el nombre, lo que muestra la ventana y su tecla aparecen cuando el ratón está encima. Un emoji

sería la manera fácil de dibujar algo así, pero un emoji es un carácter, dibujado por una fuente, y algunos sistemas no tienen ninguna fuente con esos caracteres: allí los botones mostrarían cajas vacías. Así que los dibujos de `icons.rs` están **pintados**, con líneas, rectángulos y polígonos, de la misma manera en todos los sistemas.

Cada dibujo es una función sencilla con la misma firma, y Rust puede nombrar esa firma como un tipo:

```
/// A picture: drawn by a function of the painter, the square it fills
/// and the colour to draw it in.
pub type Picture = fn(&Painter, Rect, Color32);
```

`fn(&Painter, Rect, Color32)` es un tipo **puntero a función**. Cualquier función con esa firma puede guardarse en una variable de tipo `Picture`, ponerse en un array o pasarse a otra función, sin un `Box` ni un genérico.

Cada dibujo se pinta en un cuadrado, y sus puntos se dan como fracciones del cuadrado, de 0 a 1, para que el mismo dibujo pueda pintarse a cualquier tamaño:

```
/// The point `x`, `y` of the way across and down `rect`, each from 0 to
1.
fn at(rect: Rect, x: f32, y: f32) -> Pos2 {
    rect.min + vec2(x * rect.width(), y * rect.height())
}

/// The mixer: three faders, each at its own level.
pub fn mixer(painter: &Painter, rect: Rect, color: Color32) {
    for (x, level) in [(0.2, 0.62), (0.5, 0.3), (0.8, 0.5)] {
        line(painter, rect, (x, 0.0), (x, 1.0),
color.gamma_multiply(0.6));
        let knob = Rect::from_center_size(at(rect, x, level), vec2(6.0,
3.5));
        painter.rect_filled(knob, 1.0, color);
    }
}
```

La hoja de acordes es una página con nombres de acordes a dos por línea, el rack tres filas de pasos con algunos encendidos, el piano roll un teclado con tres notas, las teclas de armonía un teclado con dos teclas negras, el metrónomo su cuerpo y su brazo.

La barra de herramientas dibuja sus cinco botones de ventana a partir de un array. Cada entrada guarda el nombre, el `bool` que dice si esa ventana está abierta, el dibujo y la ayuda:

```
let windows: [(&str, &mut bool, icons::Picture, &str); 5] = [
    ("Chord sheet", &mut self.chord_sheet.open, icons::chord_sheet,
     "the song's harmony bar by bar, to read and write (F5)"),
    ("Channel rack", &mut self.rack_open, icons::rack,
     "the steps of the selected pattern (F6)"),
    // … the piano roll, the harmony keys, the mixer
];
for (name, open, picture, about) in windows {
    let button = icons::toggle(ui, name, *open, picture);
    // ...the hint, and a click opens or closes the window...
}
```

Hay cinco préstamos `&mut` de `self` en un array. El borrow checker lo permite porque cada uno toma prestado un **campo distinto**: `self.chord_sheet.open`, `self.rack_open`, `self.piano_roll.open` y así. Dos `&mut` al mismo campo no compilarían.

La prueba `every_picture_stays_in_its_square` pinta cada dibujo en un cuadrado de 16 puntos y comprueba que ninguna forma se sale de él, y `the_window_buttons_open_and_close_their_windows` hace clic en cada uno de los cinco y comprueba que su ventana se abre, que el botón se enciende y que un segundo clic la cierra.

Todas las teclas en una lista

Ayuda > Atajos de teclado, o **F1**, abre una ventana con cada tecla a la que responde la app, sección por sección.



La lista son datos: un `Vec` de secciones, cada una con un título, una nota sobre cuándo funcionan sus teclas y sus líneas. Las teclas de una línea son de una de cuatro clases:

```
enum Keys {
    /// Any of these does it.
    Any(Vec<KeyboardShortcut>),
    /// The first does the first half of what the line says, the second
the
    /// second: "Up / down".
    Pair(KeyboardShortcut, KeyboardShortcut),
    /// The mouse wheel, turned while holding these.
    Wheel(Modifiers),
    /// Keys that are not one shortcut, written out.
    Written(&'static str),
}
```

Donde la app tiene una constante para un atajo, la lista usa esa misma constante: `files::SAVE`, `edit::UNDO`, `piano_roll::OCTAVE_UP`, `view::MIXER`. Así, si una tecla cambia, la lista cambia con ella. Convertir unas `Keys` en texto usa el `format_shortcut` de egui, que escribe «Ctrl+Shift+Z» en Linux y Windows y «⇧⌘Z» en un Mac, como lo escriben los menús.

Como la lista son datos, también se puede probar como datos. `no_two_global_shortcuts_share_a_key` reúne las teclas de las secciones que funcionan en cualquier sitio y falla si alguna tecla aparece dos veces, porque esa tecla haría dos cosas a la vez.

Ayuda: el manual y Acerca de

Ayuda > Manual de usuario abre las páginas web del manual de usuario en el idioma de la app. La dirección vive en una función de `about.rs`, así que el menú Ayuda y la ventana Acerca de se la piden a ella:

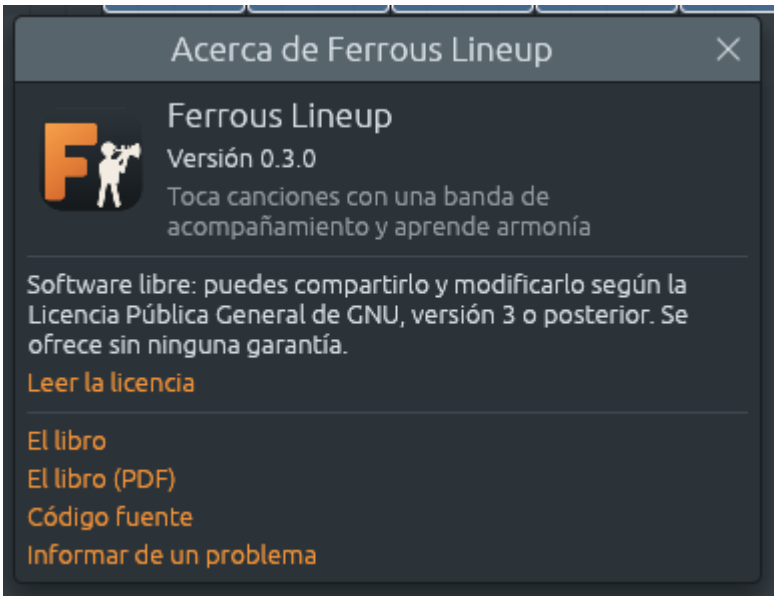
```
pub(super) fn manual() -> &'static str {
    match i18n::current_language() {
        Language::English => MANUAL,
        Language::Spanish => MANUAL_ES,
    }
}
```

El `match` no tiene rama `_` a propósito. Añade un tercer idioma, y el compilador se detiene aquí hasta que decidas qué manual reciben sus lectores. `pub(super)` hace que `manual` se vea desde `app.rs` y desde los módulos que tiene dentro, como `shortcuts.rs`, donde está el menú Ayuda, y desde ningún otro lado.

En el menú, el botón le pide a egui que abra esa dirección en una pestaña nueva del navegador:

```
if ui.button(crate::i18n::tr("User manual")).clicked() {
    ui.ctx()
        .open_url(egui::OpenUrl::new_tab(super::about::manual()));
    ui.close();
}
```

Ayuda > Acerca de Ferrous Lineup dice qué versión estás usando, con qué licencia, y dónde encontrar el manual de usuario, este libro (cada uno también en PDF), el código y la lista de problemas conocidos. Es lo primero que hay que mirar antes de informar de uno.



Todo lo que dice sobre el programa sale de `Cargo.toml`, leído mientras se compila el programa:

```
const ISSUES: &str = concat!(env!("CARGO_PKG_REPOSITORY"), "/issues");

ui.label(format!("{} {}", tr("Version"), env!("CARGO_PKG_VERSION")));
ui.weak(tr(env!("CARGO_PKG_DESCRIPTION")));
```

- `env!("NOMBRE")` es una macro que el compilador sustituye por el valor de la variable de entorno `NOMBRE` *mientras compila*, como un `&'static str`. Cargo pone `CARGO_PKG_VERSION`, `CARGO_PKG_DESCRIPTION`, `CARGO_PKG_REPOSITORY` y algunas más a partir de la sección `[package]` de `Cargo.toml` (la app toma su versión, su licencia y su repositorio del `[workspace.package]` del workspace). Una variable que no existe es un error de compilación, no una sorpresa para el usuario.
- `concat!` junta literales en uno solo en tiempo de compilación, así que `ISSUES` también es una constante. Ahí no se podría usar `format!`: construye un `String` mientras el programa se ejecuta, y una `const` tiene que conocerse antes.

Así, cuando un lanzamiento cambia la versión en `Cargo.toml`, la ventana cambia con ella, y nadie tiene que acordarse de un segundo sitio. Las compilaciones de lanzamiento comparan la etiqueta con este mismo número.

El dibujo es el mismo que el escritorio muestra para la ventana de la app: `WINDOW_ICON`, 256 × 256 píxeles de bytes rojo, verde, azul y alfa, incluidos en el programa con `include_bytes!`. egui dibuja las imágenes a partir de **texturas**, imágenes guardadas en la tarjeta gráfica, así que los bytes se convierten en una textura la primera vez que se abre la ventana y se guardan en la memoria de egui:

```
fn icon(ctx: &egui::Context) -> egui::TextureHandle {
    let id = egui::Id::unique("about-icon");
    if let Some(texture) = ctx.data(|data| data.get_temp(id)) {
        return texture;
    }
    let size = crate::WINDOW_ICON_SIZE as usize;
    let image = egui::ColorImage::from_rgba_unmultiplied([size, size],
crate::WINDOW_ICON);
    let texture = ctx.load_texture("about-icon", image,
egui::TextureOptions::LINEAR);
    ctx.data_mut(|data| data.insert_temp(id, texture.clone()));
    texture
}
```

Un `TextureHandle` funciona como un `Arc`: un clon es barato, y la textura sigue en la tarjeta gráfica mientras viva algún clon. El que se guarda en la memoria de egui hace que la imagen se envíe a la tarjeta una sola vez, no en cada frame. `WINDOW_ICON` es una constante de `main.rs`, sin `pub`; `about.rs` puede leerla igualmente como `crate::WINDOW_ICON`, porque un elemento privado es visible en su propio módulo y en todos los módulos que hay dentro de él.

`ui.hyperlink_to(texto, url)` dibuja un enlace. Al pulsarlo no se abre el navegador directamente: egui añade un `OutputCommand::OpenUrl` a lo que devuelve el frame, y eframe pide al sistema que abra la dirección. Eso hace que el clic sea fácil de probar. `help_about_shows_the_version_the_license_and_where_the_book_is` abre **Ayuda > Acerca de Ferrous Lineup**, comprueba que la línea de la versión y cada enlace se dibujan, pulsa **El libro** y comprueba que el frame pidió la dirección del libro, sin que se abra ningún navegador.
`help_user_manual_opens_the_manual_in_the_apps_language` hace lo mismo con **Ayuda > Manual de usuario**, en inglés y en español.

Probar una ventana sin pantalla

Todas las pruebas nombradas en este capítulo dibujan la app sin pantalla. `egui::Context::run_ui` (o `run_pass`) toma la entrada de un frame, la posición del ratón, las teclas y los clics, y devuelve todo lo que egui le pasaría a la tarjeta gráfica, sin dibujarlo. Una prueba puede entonces encontrar un widget por su `Id` o su texto, comprobar dónde quedó, hacer clic en él en el frame siguiente y leer el estado de la app. `app/ux_tests.rs` tiene la mayoría; Cómo se prueba el código cuenta el resto.

Hay un detalle de estas pruebas que conviene conocer. Lo que devuelve `run_ui` incluye la textura de la fuente. egui comprueba, solo en las compilaciones de depuración, que una textura así no se descarta sin usarse, y si pasa, entra en pánico. Así que las pruebas que se quedan con la salida la vacían antes:

```
// The font texture is not drawn anywhere here.
output.textures_delta.clear();
```

Una prueba que pasa en una compilación de release y falla en una de depuración suele ser una de estas comprobaciones: `debug_assert!` solo se compila en las compilaciones de depuración, y una librería puede tener muchas. Por eso la CI ejecuta las pruebas en una compilación de depuración.

Los editores: rack de canales, piano roll y lista de reproducción

Archivos: `crates/lineup-app/src/channel_rack.rs`, `crates/lineup-app/src/piano_roll.rs`, `crates/lineup-app/src/playlist.rs`, `crates/lineup-app/src/widgets.rs` y `crates/lineup-app/src/app.rs`

Tres ventanas editan la música en sí. La **lista de reproducción** llena la ventana principal: patrones colocados como clips sobre pistas, que juntos forman la canción. El **rack de canales** (F6) muestra un patrón como filas de pasos, una fila por canal. El **piano roll** (F7) muestra un canal de un patrón como notas sobre una cuadrícula, con las notas agudas arriba. El rack y el piano roll flotan sobre la lista de reproducción, como cuenta el capítulo anterior.

Las tres están hechas de la misma manera, y eso es lo primero de lo que trata este capítulo.

Dibujar devuelve acciones

Ninguno de los editores cambia el proyecto. Cada uno solo lo mira y devuelve lo que hizo el usuario, como una lista de acciones. Las del rack son un enum:

```
pub enum RackAction {
    PlayPattern,
    // ...
    SelectChannel(Option<usize>),
    ToggleStep { channel: usize, step: usize },
    ClearStep { channel: usize, step: usize },
    Preview(usize),
    SetVolume { channel: usize, volume: f32 },
    ToggleMute(usize),
    Remove(usize),
    OpenPianoRoll(usize),
    AddDrum(Drum),
    AddSynth,
    // ...sending over MIDI, instruments...
    LoadSample { channel: Option<usize>, path: PathBuf },
}

pub fn show(
    ui: &mut Ui,
    project: &Project,
    pattern: usize,
    selected: Option<usize>,
    playhead: Option<usize>,
    playback: Option<RackPlayback>,
    locks: bool,
) -> Vec<RackAction>
```

`project: &Project` es un **préstamo compartido**: la función puede leer el proyecto pero no cambiarlo, y el compilador lo hace cumplir. Después la app aplica las acciones:

```
let rack_actions = rack
    .show(ctx, |ui| {
        // ...
        channel_rack::show(ui, &self.project, self.pattern, selected,
playhead, /* ... */)
    })
    .and_then(|window| window.inner)
    .unwrap_or_default();
for action in rack_actions {
    self.apply(action);
}
```

`Window::show` devuelve `None` cuando la ventana está cerrada, y su `inner` es `None` cuando la ventana está plegada, así que las acciones vienen envueltas dos veces. `.and_then(|window| window.inner)` desenvuelve el `Option` de fuera y devuelve el de dentro; `.unwrap_or_default()` convierte `None` en un `Vec` vacío.

Esta separación también resuelve un rompecabezas de préstamos. Mientras el rack se dibuja, tiene un préstamo compartido del proyecto, así que no podría cambiarlo a la

vez (eso necesitaría un `&mut` al mismo tiempo, cosa que el borrow checker prohíbe). Reunir primero las acciones y aplicarlas cuando el dibujo ha terminado mantiene los dos préstamos separados en el tiempo.

`apply` es un único `match` con una rama por acción. Cada rama que cambia la canción pasa por `edit` o `edit_pattern`, que llevan el historial de deshacer y le dicen al hilo de audio qué cambió (ver Decirle al hilo de audio lo que ha cambiado):

```
RackAction::ToggleStep { channel, step } => {
    self.edit_pattern("change steps", |pattern| {
        pattern.toggle(channel, step);
    });
}
```

La misma acción hace lo mismo venga de donde venga: un clic, una tecla o un menú. El piano roll devuelve `RollAction` y la lista de reproducción `PlaylistAction`, y la app las aplica de la misma manera.

El patrón seleccionado

El rack y el piano roll muestran **un patrón**, el seleccionado. La app lo guarda:

```
pub struct LineupApp {
    // ...
    /// The selected pattern: the one the channel rack and the piano roll
    /// show, pattern mode plays, and new playlist clips use.
    pattern: usize,
    mode: PlayMode,
    // ...
}
```

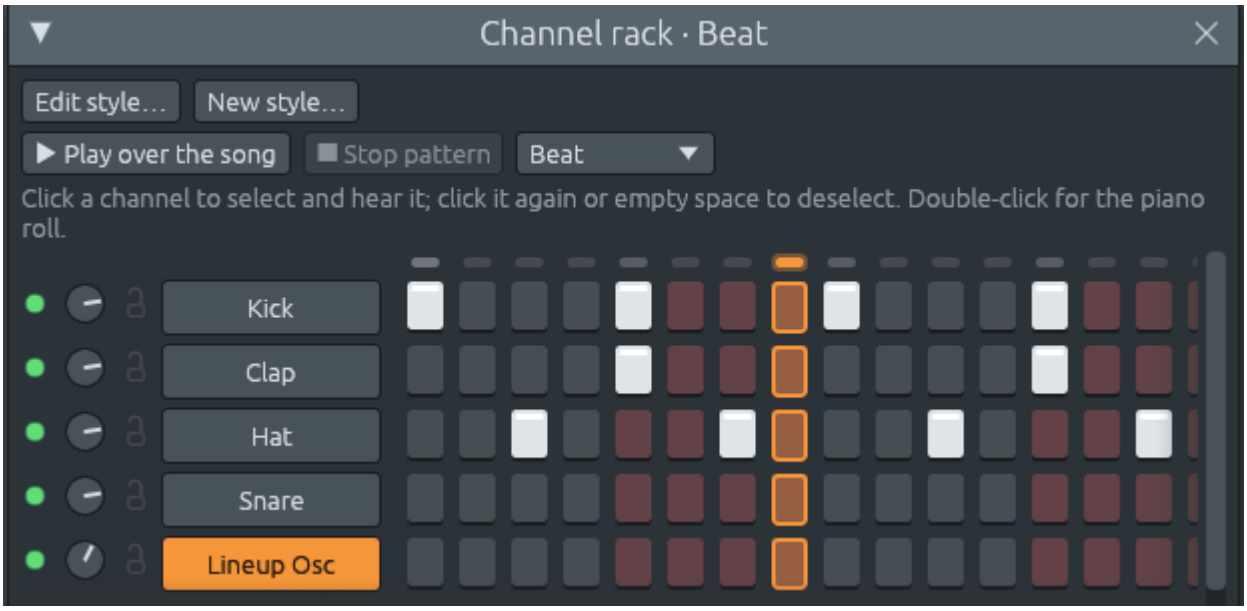
Es un campo de la app, no del proyecto: es *dónde estás mirando*, no parte de la canción, así que no se guarda, y seleccionar un patrón no cuenta como un cambio sin guardar. Los editores lo reciben como argumento, así que siguen siendo funciones de lo que se les da.

El cursor de reproducción que reciben depende del modo. En el modo patrón, el paso del hilo de audio *es* un paso del patrón seleccionado. En el modo canción es un paso de la canción, y la app le pregunta al proyecto dónde está el patrón seleccionado en ese momento:

```
fn pattern_playhead(&self) -> Option<usize> {
    let step = self.playhead()?;
    match self.mode {
        PlayMode::Pattern => Some(step),
        PlayMode::Song => {
            // ...a pattern played over the song gives its own step...
            let (step, _) = self.song_position()?;
            self.project.pattern_step_in_song(self.pattern, step)
        }
    }
}
```

`pattern_step_in_song` busca un clip del patrón, en una pista que no esté silenciada, que cubra el paso de la canción, y devuelve `None` si no hay ninguno. Así, mientras suena la canción, el cursor del rack corre cada vez que suena un clip del patrón seleccionado, y desaparece en los compases donde no.

El rack de canales



Escribir tus propios widgets

egui tiene botones y deslizadores, pero no botones de paso ni perillas. Hacer uno lleva tres movimientos: reservar espacio, leer el ratón, pintar.

```
pub fn step_button(ui: &mut Ui, on: bool, beat_group: usize, is_playhead:
bool) -> Response {
    let (rect, response) = ui.allocate_exact_size(step_size(ui),
Sense::click());

    if ui.is_rect_visible(rect) {
        let mut fill = if on {
            STEP_ON
        } else {
            STEP_OFF[beat_group % 2]
        };
        if response.hovered() {
            fill = fill.gamma_multiply(1.25);
        }
        if is_playhead {
            // The whole column that is sounding lights up, down every
row.
            fill = fill.lerp_to_gamma(PLAYHEAD, if on { 0.25 } else { 0.35
});
        }
        let painter = ui.painter();
        // ...a dark edge along the bottom, the pad, a light strip on a
        // step that is on, an orange outline on the step playing...
    }
    response
}
```

- `allocate_exact_size` reserva un rectángulo en la distribución y dice qué interacciones queremos (`Sense::click()`). Devuelve el rectángulo y un `Response` que describe lo que hizo el ratón ahí en este frame. `step_size(ui)` es el tamaño del paso multiplicado por el zoom del rack.
- `is_rect_visible` se salta el pintado de las filas que quedan fuera de la vista.
- El `Painter` dibuja formas. Los colores son constantes `Color32` definidas arriba del archivo, así que la paleta vive en un solo sitio.
- `lerp_to_gamma` mezcla algo de naranja en el paso que suena, un poco menos en un paso encendido, para que un paso encendido siga blanco.
- Devolver el `Response` permite a quien llama usarlo como cualquier widget de serie: `if response.clicked() { ... }` .

Los pulsos alternan colores, para que puedas contarlos, y vuelven a empezar en cada compás para que todos los compases se vean iguales: en 3/4, oscuro, claro, oscuro. El rack le pregunta al compás en qué pulso cae cada paso (`time.beat_at(step).index`), así que en 6/8 agrupa sus pasos de tres en tres.

La perilla usa `Sense::click_and_drag()` , convierte el movimiento vertical del ratón en un cambio de valor, y llama a `response.mark_changed()` para que quien la llama pueda usar `.changed()` como en un deslizador. Su aguja se dibuja con un poco de

trigonometría: el ángulo recorre 270 grados, y `(sin, -cos)` del ángulo da la dirección (la *y* de la pantalla crece hacia abajo, de ahí el menos).

Menús

```
name.context_menu(|ui| self.channel_menu(ui, index, channel, locked,
actions));
```

`context_menu` abre un menú con el clic derecho; `menu_button` abre uno con un clic normal (el botón «Añadir canal»). Los dos reciben un closure que dibuja el contenido del menú, y `ui.close()` cierra el menú tras una elección. `add_enabled_ui(false, ...)` pone en gris todo lo que hay dentro, y así se desactiva «Añadir canal» cuando el rack está lleno:

```
let rack_full = project.channels().len() >= MAX_CHANNELS;
ui.add_enabled_ui(!rack_full, |ui| {
    ui.menu_button(tr("➕ Add channel"), |ui| { /* ... */ });
});
```

Arrastrar y soltar

egui informa de los archivos que se arrastran sobre la ventana en `input.raw.hovered_files`, y de los que se sueltan en `input.raw.dropped_files`. Mientras dibuja, el rack guarda el rectángulo de cada fila; después busca la fila que está bajo el ratón:

```
let target_row = pointer.and_then(|p| row_rects.iter().position(|rect|
rect.contains(p)));
```

`position` devuelve el índice del primer elemento para el que el closure es verdadero, o `None`. Un archivo WAV soltado sobre una fila sustituye el sonido de ese canal; un archivo soltado en cualquier otro sitio se convierte en un canal nuevo con el nombre del archivo (`path.file_stem()` es el nombre del archivo sin su extensión). Un archivo de proyecto, MIDI o de hoja de acordes, o una grabación, se deja para la app, que lo abre o lo importa.

La carga ocurre en `LineupApp::load_sample`, en el hilo de la ventana, porque leer archivos es lento y nunca debe ocurrir en el hilo de audio. Lee el archivo *antes* de cambiar nada, así que un archivo roto deja el canal como estaba y pone la razón en la barra de estado.

Una fila de luces sobre los pasos

Una caja de ritmos tiene una fila de lucecitas, una sobre cada paso, y la luz sobre el paso que suena corre junto con la música. El rack también tiene esa fila. Las luces deben quedar exactamente sobre los pasos, pero los pasos empiezan después de la luz de silencio, la perilla, el candado y el nombre del canal, y sus anchos cambian con el zoom del rack. En lugar de sumar esos anchos (y equivocarse la próxima vez que se añada un control a una fila), el rack deja que las filas se lo digan:

```
let (lights, _) = ui.allocate_exact_size(
    egui::vec2(1.0, widgets::step_lights_height(ui)),
    egui::Sense::hover(),
);
let mut steps_left = None;
for (index, channel) in project.channels().iter().enumerate() {
    let row = ui.horizontal(|ui| self.row(ui, index, channel, actions));
    steps_left.get_or_insert(row.inner);
    row_rects.push(row.response.rect);
}
if let Some(left) = steps_left {
    let marks = step_marks(self.time, self.pattern.length());
    widgets::step_lights(ui, lights, left, &marks, self.playhead);
}
```

`allocate_exact_size` reserva una franja vacía arriba, así que las filas van debajo. Cada `row` devuelve dónde empiezan sus pasos (`ui.cursor().left()` justo antes del primer paso); `get_or_insert` guarda el valor de la primera fila e ignora los demás. Cuando las filas terminan, `step_lights` pinta en la franja reservada. egui dibuja todo al final del frame, así que pintar en un espacio reservado antes no es problema.

`step_lights` recibe un slice de `StepMark`, un enum pequeño que dice si cada paso empieza un compás, empieza un pulso o ninguna de las dos cosas:

```
pub enum StepMark {
    Bar,
    Beat,
    Plain,
}
```

La luz sobre el primer paso de cada compás es un poco más brillante que las demás, y la luz sobre el primer paso de cada pulso un poco más brillante que las de los pasos intermedios, así que la fila se puede contar como los pasos que tiene debajo. Cada luz se coloca un ancho de paso (el botón más el hueco que lo sigue) después de la anterior, la misma aritmética que usa el mini piano roll, así que luces, botones y vistas previas se alinean con cualquier zoom.

Un mini piano roll para las melodías

Un canal cuyas notas son todas pasos simples muestra botones de paso. Cualquier otro canal muestra `widgets::mini_roll`, un dibujo diminuto de sus notas, y un clic en él abre el piano roll. El widget tiene que alinearse con los botones de paso, que tienen un hueco entre ellos:

```
let gap = ui.spacing().item_spacing.x;
let step_size = step_size(ui);
let step_width = step_size.x + gap;
let size = vec2(step_width * length as f32 - gap, step_size.y);
```

El rango de alturas se comprime en la altura de la fila, pero nunca menos de una octava, para que tres notas cercanas no llenen la fila de arriba abajo:

```
let visible = || notes.iter().filter(|note| note.start < length);
let lowest = visible().map(|note| note.pitch.midi()).min().unwrap_or(0);
let highest = visible().map(|note| note.pitch.midi()).max().unwrap_or(0);
let span = (highest - lowest).max(12) as f32;
```

`visible` es un closure que crea un iterador nuevo cada vez que se le llama, porque `min` y `max` gastan uno cada uno. La columna que suena se rellena de un naranja transparente (`gamma_multiply(0.25)` deja el color a una cuarta parte de su fuerza), igual que se tiñen los botones de paso.

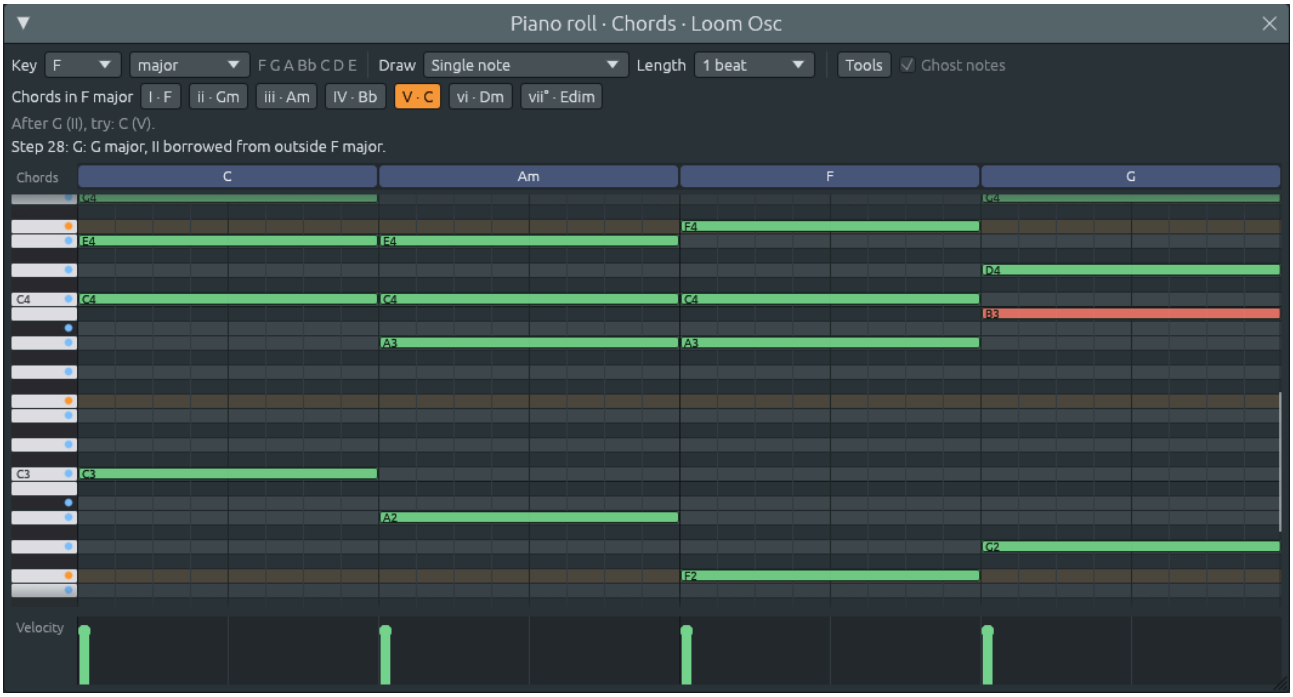
La batería en otro patrón, y un rack que se ajusta

Las canciones suelen guardar su batería en un patrón propio, «Batería de A» junto a «A». El rack de «A» mostraría filas de batería sin nada, y parecería roto. Así que cuando un patrón no tiene pasos de batería y otros sí, `drums_elsewhere` lo dice encima de las filas, con un botón para cada uno de esos patrones.

El rack se abre tan grande como sus filas. `natural_size` las suma (los controles, los pasos, las luces, el menú «Añadir canal», sitio para una barra de desplazamiento), y la ventana recibe ese tamaño durante un frame cada vez que las filas cambian, como las ventanas del tamaño de lo que muestran. Se puede hacer más pequeña igualmente, y entonces se desplaza. Cuando caben todas las filas, la rueda del ratón sola desplaza los pasos hacia los lados. egui tiene un ajuste de estilo justo para eso, que funciona cuando el scroll area permite una sola dirección:

```
ui.style_mut().always_scroll_the_only_direction = true;
let vertical = rows_height > ui.available_height();
let scroll = egui::ScrollArea::new([true, vertical])
```

El piano roll



Una ventana con estado propio

El rack de canales es una función: todo lo que muestra sale del proyecto. El piano roll necesita recordar cosas que el proyecto no conoce, de un frame al siguiente: qué canal muestra, qué dibuja un clic, las notas seleccionadas, un arrastre en curso. Así que es un struct con métodos:

```
pub struct PianoRoll {
    pub open: bool,
    channel: usize,
    stamp: ChordStamp,
    /// The length of new notes, in steps. Resizing a note changes it, so
    /// the next note drawn is as long as the last one edited.
    note_length: usize,
    drag: Option<Drag>,
    group_drag: Option<GroupDrag>,
    selection: Vec<NoteEvent>,
    ghosts: bool,
    zoom: Vec2,
    // ...
}
```

Aun así nunca cambia el proyecto él mismo. `show` devuelve una lista de `RollAction`, como hace el rack:

```
pub enum RollAction {
    /// Add one note, or the notes of a chord.
    AddNotes { channel: usize, notes: Vec<NoteEvent> },
    RemoveNote { channel: usize, index: usize },
    /// Replace a note that was moved or resized.
    ReplaceNote { channel: usize, index: usize, note: NoteEvent },
    /// Replace some notes with others, all at once: a moved selection,
    /// a tool, new velocities. `label` names the undo step.
    ReplaceNotes { channel: usize, old: Vec<NoteEvent>, new:
Vec<NoteEvent>, label: &'static str },
    /// Play some notes right now, without adding them.
    Preview { channel: usize, pitches: Vec<Note> },
    SetScale(Scale),
    /// Show another pattern: one where the channel has notes.
    SelectPattern(usize),
    /// Open the channel rack, where drum steps are edited.
    OpenRack,
    // ...
}
```

Así que el struct guarda estado de la *vista*, que nunca se guarda en el archivo, y el proyecto guarda el estado del *documento* (las notas), que sí.

La ventana se construye así:

```
// `open` is a copy: the window needs `&mut open` while the closure
// below needs `&mut self`, and both cannot borrow `self` at once.
let mut open = self.open;
let window = egui::Window::new(roll_title)
    .id(id)
    .open(&mut open)
    .default_pos([150.0, 160.0])
    .constrain_to(area);
// ...the size it opens at...
window.show(ctx, |ui| {
    self.contents(ui, project, pattern_notes, playhead, held, &mut
actions)
});
self.open = open;
```

- El título cambia con el patrón y el canal, y egui normalmente identifica una ventana por su título. `.id(...)` le da en cambio una identidad fija, para que conserve su posición y su tamaño cuando cambias de canal.
- `.open(&mut open)` añade el botón de cerrar: egui pone `open` a `false` cuando se pulsa.
- Si la ventana tomara prestado `&mut self.open` mientras el closure toma prestado `&mut self` (para llamar a `self.contents`), `self.open` quedaría prestado de forma mutable dos veces a la vez, cosa que Rust prohíbe. Copiar el `bool` fuera y volver a meterlo evita el solape.

Píxeles y música: `Grid`

Casi todo el código de dibujo convierte pasos y alturas en posiciones de pantalla, o la posición del puntero de vuelta en un paso y una altura. Esa aritmética vive en un solo sitio:

```
#[derive(Debug, Clone, Copy)]
struct Grid {
    /// Where all the steps and rows are, even those scrolled out of view.
    rect: Rect,
    step_width: f32,
    /// How tall a row (one pitch) is: the vertical zoom.
    row_height: f32,
    length: usize,
    time: TimeSignature,
    /// The left edge of the keyboard. The keyboard stays at the left of
    /// the view while the grid scrolls sideways under it.
    keys_left: f32,
}

impl Grid {
    /// The x coordinate where `step` begins.
    fn x(self, step: usize) -> f32 {
        self.rect.left() + step as f32 * self.step_width
    }

    /// The y coordinate of the top of a pitch's row (high notes at the
top).
    fn row_top(self, midi: u8) -> f32 {
        self.rect.top() + (HIGHEST - midi) as f32 * self.row_height
    }

    /// The step and pitch under a point. Points outside the grid give the
    /// nearest cell, so a drag can go past the edges.
    fn cell_at(self, pos: Pos2) -> (usize, Note) {
        let row = ((pos.y - self.rect.top()) / self.row_height)
            .floor()
            .clamp(0.0, (HIGHEST - LOWEST) as f32) as u8;
        (self.step_at(pos.x), Note::new(HIGHEST - row))
    }
    // ...keys, cells, step_at, note_rect, text_size
}
```

Las notas agudas están arriba, así que la fila cuenta hacia abajo desde `HIGHEST`. `cell_at` limita el valor, así que un arrastre que sale de la cuadrícula sigue obteniendo la celda más cercana, y devuelve una **tupla**, `(usize, Note)`, que quien llama desarma con `let (step, pitch) = grid.cell_at(pos);`. `note_rect` dibuja una nota desde su inicio exacto hasta su final exacto, incluido cuánto se aparta del paso, así que un tresillo queda entre dos pasos, como se oye.

Como `Grid` es `Copy` y sus métodos toman `self` por valor, se puede pasar a todas las funciones de dibujo sin ningún `&` ni lifetimes. Para un struct de unos pocos números, copiar es tan barato como pasar una referencia.

Al 100% el patrón entero cabe a lo ancho de la ventana, así que el ancho del paso se calcula en cada frame a partir del ancho de la ventana. Al hacer zoom los pasos se ensanchan, y la cuadrícula se desplaza hacia los lados bajo el teclado.

Un teclado que se queda, carriles que siguen

La cuadrícula está dentro de un scroll area, pero dos cosas no deben desplazarse con ella. Las teclas del piano se dibujan en el borde izquierdo de lo *visible* (`keys_left: visible.left()`), encima de la cuadrícula, así que las notas se deslizan por debajo. El carril de acordes sobre la cuadrícula y el carril de velocidad debajo se dibujan fuera del scroll area, así que no saben cuánto se ha desplazado la cuadrícula. Un struct pequeño lleva lo que necesitan para alinearse:

```
/// What the chord and velocity lanes need to line up with the grid.
#[derive(Debug, Clone, Copy)]
struct Lanes {
    step_width: f32,
    /// How far the grid is scrolled sideways, in pixels.
    scroll_x: f32,
    length: usize,
}
```

Cada carril construye su propia `Grid` a partir de él (`Lanes::grid`), desplazada por `scroll_x`, y usa las mismas funciones que la cuadrícula de notas para encontrar dónde está un paso. Una única fuente de verdad para «¿dónde está el paso 12?» es lo que mantiene alineadas tres franjas dibujadas por separado.

La entrada del ratón como una pequeña máquina de estados

egui informa de lo que hace el ratón *en este frame*. Un arrastre dura muchos frames, así que el piano roll lo recuerda en un enum:

```
enum Drag {
    /// Moving a note. `grab` is how many steps after the note's start the
    /// pointer took hold of it, so the note does not jump under the
pointer.
    Move { index: usize, grab: usize },
    /// Changing a note's length by dragging its right end.
    Resize { index: usize },
}
```

`handle_input` es largo, pero cada parte responde a una pregunta:

1. **¿Se acaba de pulsar el botón izquierdo sobre la cuadrícula?** Si dio en una nota, empieza un `Move`, o un `Resize` si la pulsación fue cerca del extremo derecho de la nota. Si dio en una celda vacía, añade lo que dibuja un clic (una nota o un acorde); una sola nota nueva también empieza un `Move`, para que puedas colocar y arrastrar en un solo gesto.
2. **¿Está el botón derecho pulsado sobre una nota?** Bórrala. Comprobarlo en cada frame significa que puedes barrer varias notas para borrarlas todas.
3. **¿Está el botón izquierdo arriba?** Entonces cualquier arrastre ha terminado.
4. **¿Hay un arrastre en curso?** Calcula dónde estaría ahora la nota y, si eso difiere de donde está, envía un `ReplaceNote`.

La cuarta parte usa una **let chain** con tres condiciones, y la sintaxis de actualización de structs para cambiar uno o dos campos:

```
if let Some(drag) = self.drag
    && let Some(pos) = pointer
    && let Some(&note) = notes.get(drag.index())
{
    let (step, pitch) = grid.cell_at(pos);
    let pitch = self.snapped(pitch, scale);
    let changed = match drag {
        Drag::Move { grab, .. } => NoteEvent {
            pitch,
            start: step.saturating_sub(grab),
            ..note
        },
        Drag::Resize { .. } => {
            let length = (step + 1).saturating_sub(note.start).max(1);
            self.note_length = length;
            // ...
            NoteEvent { length, trim, ..note }
        }
    };
    if changed != note {
        // ...play the new pitch, keep the note selected...
        actions.push(RollAction::ReplaceNote { channel, index:
drag.index(), note: changed });
    }
}
```

`Some(&note)` en un patrón copia la nota fuera del `&NoteEvent` que devuelve `get`. Comparar `changed != note` (gracias al `PartialEq` derivado) hace que no se envíe nada mientras el puntero se quede dentro de la misma celda, aunque este código se ejecute en cada frame. `snapped` lleva la altura a la tonalidad cuando **Ajustar a la escala** está encendido.

Dos detalles hacen que los clics caigan donde se quería. Un clic derecho rápido puede pulsar y soltar el botón dentro de un mismo frame, así que «¿está el botón pulsado?» ya es falso cuando se dibuja el frame. El código pregunta en cambio «¿se pulsó en este frame, o está pulsado?»:

```
pointer.secondary_pressed() || pointer.secondary_down(),
```

Y en el frame en que baja el botón izquierdo, la pulsación se atiende donde el botón *bajó*, que `widgets::press_pos` lee de los eventos del frame (ver Clics que hacen lo que querías decir): una mano rápida puede pulsar y seguir moviéndose en el mismo frame.

Cuando el puntero está sobre una celda vacía, el piano roll dibuja el contorno de lo que colocaría un clic, a partir del mismo `stamp_notes` que usa el clic. Con «Acorde de la tonalidad (séptima)» elegido, ves el acorde entero antes de colocarlo.

La selección guarda notas, no posiciones

La manera más simple de recordar qué notas están seleccionadas serían sus posiciones en la lista del canal: `selection: Vec<usize>`. El arrastre de una sola nota hace exactamente eso (`Drag::Move { index, .. }`), y ahí funciona porque nada más cambia la lista durante un arrastre.

Una selección vive mucho más que un arrastre. Mientras hay notas seleccionadas, una herramienta las saca de la lista y pone las nuevas al final, deshacer coloca un proyecto anterior entero, y borrar una nota sube un lugar a todas las que van después. Después de cualquiera de esas cosas, la posición 3 puede ser otra nota, o no ser ninguna nota. Una selección de posiciones seleccionaría en silencio las notas equivocadas, y la siguiente herramienta cambiaría notas que nunca elegiste.

Así que la selección guarda las propias notas:

```
/// The selected notes, by value. A note that is no longer in the
/// pattern (after an undo, say) is simply not found, and so not
/// selected: the selection can never point at the wrong note.
selection: Vec<NoteEvent>,
```

Esto funciona porque un canal nunca tiene dos notas en el mismo lugar: la regla del patrón (misma altura, mismo inicio, mismo desplazamiento) hace único el valor de cada nota, de modo que un valor nombra exactamente una nota. `NoteEvent` es `Copy` y `PartialEq`, así que guardar copias y compararlas es barato y fácil.

Al comienzo de cada frame, las notas que han desaparecido se olvidan:

```
let notes = pattern.notes(self.channel);
self.selection.retain(|note| notes.contains(note));
```

Y cuando el piano roll sustituye algunas notas, hace que la selección las siga, de modo que pulsar ↑ tres veces mueve el mismo acorde tres veces:

```
fn replace(
    &mut self,
    pattern: &Pattern,
    actions: &mut Vec<RollAction>,
    label: &'static str,
    old: Vec<NoteEvent>,
    new: Vec<NoteEvent>,
) -> Option<Vec<NoteEvent>> {
    let new = tidy_notes(new);
    if old == new || !pattern.can_replace_notes(self.channel, &old, &new)
{
        return None;
    }
    let was_selected = old.iter().any(|note|
self.selection.contains(note));
    self.selection.retain(|note| !old.contains(note));
    if was_selected {
        self.selection.extend(&new);
    }
    actions.push(RollAction::ReplaceNotes {
        channel: self.channel,
        old,
        new: new.clone(),
        label,
    });
    Some(new)
}
```

`replace` es la única puerta por la que pasa todo cambio de grupo: las herramientas, el arrastre de grupo y el carril de velocidad. Ordena las notas nuevas igual que lo hará el patrón, le pregunta al patrón si el cambio está permitido, y solo entonces envía la acción. Ordenarlas también aquí significa que la selección guarda exactamente las notas que estarán en el patrón, y comprobar antes significa que un movimiento rechazado no hace nada en absoluto en lugar de hacer media cosa.

La comparación cuesta un poco: `contains` recorre la lista entera, así que comprobar cada nota contra la selección lleva un tiempo proporcional al número de notas multiplicado por el tamaño de la selección. Para un piano roll con unos cientos de notas, eso está muy por debajo de un milisegundo. Si algún día importara, un `HashSet<NoteEvent>` haría cada comprobación de tiempo constante (`NoteEvent` ya deriva `Hash` para eso).

Seleccionar con el ratón

Ctrl+clic sobre una nota la añade a la selección, o la quita:

```
if let Some(at) = self.selection.iter().position(|&n| n == note) {
    self.selection.remove(at);
} else {
    self.selection.push(note);
}
```

`input.modifiers.command` es «Ctrl» en Linux y Windows y «⌘» en un Mac, así que el mismo código sigue las costumbres de cada sistema.

Ctrl+arrastrar sobre un espacio vacío dibuja un recuadro. Solo se guarda la esquina donde empezó; la selección se reconstruye desde cero en cada frame, a partir de esa esquina y del puntero, así que volver a achicar el recuadro deselecciona notas:

```
if let Some(origin) = self.selection_box
    && let Some(pos) = pointer
{
    let area = Rect::from_two_pos(origin, pos);
    self.selection = notes
        .iter()
        .filter(|note| note.start < grid.length &&
grid.note_rect(note).intersects(area))
        .copied()
        .collect();
}
```

`Rect::from_two_pos` hace un rectángulo a partir de dos esquinas cualesquiera, en el sentido en que se haya arrastrado el recuadro, e `intersects` es verdadero cuando dos rectángulos se solapan aunque sea un poco, así que basta con que el recuadro toque una nota.

Mover un grupo

Pulsar una nota seleccionada (en cualquier sitio salvo su extremo derecho, que sigue cambiando el largo) empieza un arrastre de grupo:

```
struct GroupDrag {
    /// The cell where the drag started.
    from_step: usize,
    from_pitch: Note,
    /// The notes as they were when the drag started.
    originals: Vec<NoteEvent>,
    /// The notes as they are now, after the moves so far.
    moved: Vec<NoteEvent>,
    /// Whether the notes have moved at all.
    edited: bool,
}
```

En cada frame del arrastre, la posición nueva se calcula a partir de los *originales*, con la distancia entera desde la celda de inicio hasta el puntero:

```
let steps = step as isize - drag.from_step as isize;
let semitones = i32::from(pitch.midi()) -
i32::from(drag.from_pitch.midi());
drag.edited |= steps != 0 || semitones != 0;
let snap = self.snap_to_scale && drag.edited;
let new = tools::shift(&drag.originals, steps, grid.length)
    .and_then(|shifted| tools::transpose(&shifted, semitones))
    .map(|moved| match snap {
        true => tools::snap_to_scale(&moved, scale),
        false => moved,
    })
    .filter(|moved| fits_view(moved));
```

Mover las notas un poco en cada frame (desde `moved`) sería la manera obvia, pero los pequeños errores se irían sumando: un movimiento rechazado en el borde del patrón se perdería, y las notas acabarían en otro sitio que bajo el puntero. Empezar cada vez desde los originales hace que las notas estén siempre exactamente donde dice el puntero, y arrastrar de vuelta al inicio las deja exactamente como estaban.

Las herramientas (ver Herramientas de edición como funciones simples) se encadenan con `Option::and_then`: desplazar, y *si eso funcionó*, transportar el resultado. `map` ajusta las notas a la tonalidad cuando Ajustar a la escala está encendido, pero solo una vez que se han movido, así que arrastrar una selección un semitono arriba y de vuelta abajo la mete en la tonalidad. `filter` rechaza después las notas que quedan fuera de las filas que muestra el piano roll. La sustitución va de `moved` (lo que está ahora en el patrón) a `new`, y `moved` se actualiza con lo que devolvió `replace`.

El arrastre se saca de `self` mientras se trabaja con él:

```
if let Some(mut drag) = self.group_drag.take()
    && let Some(pos) = pointer
{
    // ... self.replace(pattern, actions, "move notes",
drag.moved.clone(), new) ...
    self.group_drag = Some(drag);
}
```

`self.replace` necesita `&mut self`, el struct entero. Si `drag` fuera una referencia dentro de `self.group_drag`, el borrow checker se negaría: una parte de `self` estaría prestada mientras todo `self` se presta de forma mutable. `Option::take` saca el arrastre y deja `None`, así que `drag` es un valor aparte y `self` queda libre; el arrastre se vuelve a poner al final.

Cada frame del arrastre produce un `ReplaceNotes`, y cada uno pasa por `edit_pattern`. Aun así forman un solo paso de deshacer, porque un paso sigue abierto mientras se mantiene pulsado un botón del ratón (ver Deshacer y rehacer). Y como el paso ya está abierto, `edit` no vuelve a copiar el proyecto entero en cada frame; le envía al hilo de audio lo que cambió como mucho cada 50 ms, y el resto al soltar el botón. Nada en el piano roll tiene que saber nada de ninguna de las dos cosas.

Quién recibe las teclas

Las flechas y Suprimir son útiles en el piano roll, pero el rack y la lista de reproducción también las usan, y una tecla que cambia notas en una ventana que no estás mirando sería una sorpresa desagradable. Así que las teclas de los editores van a la ventana en la que se hizo clic la última vez, y a ninguna otra:

```
/// Editor shortcuts belong to the last clicked window, including its
title
/// and toolbar. Popup controls and text fields keep their own keyboard
input.
pub fn editor_keyboard(ctx: &egui::Context, layer: egui::LayerId) -> bool
{
    editor_active(ctx, layer)
        && !ctx.text_edit_focused()
        && !egui::Popup::is_any_open(ctx)
        && ctx.memory(|memory| memory.is_above_modal_layer(layer))
}
```

`editor_active` recuerda la *capa* de la última pulsación (cada ventana es una capa, ver ¿Qué hay bajo el puntero?), en la memoria de egui, y responde si es esta. Una ventana que la app abre por ti, como cuando un doble clic en un clip abre el piano roll, recibe el teclado con `widgets::give_keyboard`, como si se hubiera hecho clic en ella. La lista de reproducción comparte la capa de fondo con las barras de herramientas, así que para ella también importa el lugar del clic (`editor_keyboard_in`). Las teclas se ignoran mientras un campo de texto tiene el teclado, para que escribir el nombre de un patrón nunca mueva notas, y mientras hay un menú abierto.

Espacio, que toca y detiene, tiene el problema contrario: debe funcionar en todas partes, pero egui también deja pulsar con Espacio un botón que tiene el foco. Haz clic en ▶ Play, pulsa Espacio, y el atajo detendría la música mientras el botón la volvía a poner. Así que los atajos se leen antes de dibujar ningún panel, y `consume_key` se lleva la pulsación una vez usada:

```
let typing = ctx.text_edit_focused() || egui::Popup::is_any_open(ctx);
// `consume_key` reports whether the key was pressed and removes the
// press, so a focused button does not also react to Space.
let pressed = |key| ctx.input_mut(|input|
input.consume_key(egui::Modifiers::NONE, key));
if !typing && !controls_active && pressed(egui::Key::Space) {
    // ...play or stop the song, or the pattern in the rack or piano
roll...
}
```

El orden de los atajos de teclado

Cada tecla es un `KeyboardShortcut`: una tecla y sus modificadores.

```
pub(crate) const KEY_UP: KeyboardShortcut = shortcut(Modifiers::NONE,
Key::ArrowUp);
pub(crate) const OCTAVE_UP: KeyboardShortcut = shortcut(Modifiers::SHIFT,
Key::ArrowUp);
pub(crate) const IN_KEY_UP: KeyboardShortcut = shortcut(Modifiers::ALT,
Key::ArrowUp);
```

`shortcut` es una `const fn`, una función lo bastante simple como para ejecutarse mientras se compila, y por eso sus resultados pueden ser `const`. Son `pub(crate)` para que la lista de **Ayuda > Atajos de teclado** use las mismas constantes (ver Todas las teclas en una lista).

El piano roll le pregunta a egui si se pulsó cada uno con `input.consume_shortcut(&shortcut)`. Eso devuelve `true` una vez, y «consume» la pulsación para que nada más reaccione a ella. Un detalle de egui importa aquí: compara los atajos de forma *lógica*, lo que significa que unas teclas Mayús o Alt de más no impiden la coincidencia. Mayús+↑ cuenta también como una pulsación de ↑. Comprobados en el orden equivocado, Mayús+↑ se lo comería el atajo de «subir un semitono» antes de que el de «subir una octava» llegara a verlo. Así que la lista va de lo más específico a lo menos:

```
// egui counts Shift+↑ and Alt+↑ as ↑ too, so the longer shortcuts
// are checked first.
let keys = [
    (IN_KEY_UP, Tool::InKey(1)),
    (IN_KEY_DOWN, Tool::InKey(-1)),
    (OCTAVE_UP, Tool::Transpose(12)),
    (OCTAVE_DOWN, Tool::Transpose(-12)),
    (KEY_UP, Tool::Transpose(1)),
    // ...
];
keys.into_iter()
    .find(|&(shortcut, _)| pressed(shortcut))
    .map(|(_, tool)| tool)
```

`find` se detiene en el primer atajo pulsado, así que como mucho se usa una herramienta por frame, y a los atajos más cortos de después ni siquiera se les pregunta.

Un enum para el menú y las teclas

El menú Herramientas y las teclas producen los dos una `Tool`:

```
enum Tool {
    Copy,
    Cut,
    Paste,
    Duplicate,
    Transpose(i32),
    InKey(i32),
    Shift(isize),
    Quantize,
    SnapToKey,
    Invert { up: bool },
    Arpeggiate,
    Delete,
    SelectAll,
    SelectNone,
}
```

y `use_tool` convierte cualquier `Tool` en una llamada a la función correspondiente de `lineup_core::tools`, seguida de `replace`. Una herramienta cambia las notas seleccionadas, o todas las notas cuando no hay ninguna seleccionada; Borrar solo borra una selección. Venga del menú o de una tecla, una herramienta sigue exactamente el mismo camino, así que las dos nunca pueden comportarse distinto. El menú muestra la tecla de cada herramienta a su lado, escrita como el sistema escribe las teclas, así que sirve también como lista de atajos.

Un menú que se queda abierto

Los menús de egui se cierran con cualquier clic dentro de ellos, que es lo correcto para una lista de órdenes. Pero el menú Herramientas también tiene opciones: el orden y la velocidad del arpegio. Cerrar el menú al elegir «arriba y abajo» te obligaría a abrirlo otra vez para pulsar «Arpegiar los acordes». Así que a este menú se le dice que se cierre solo con un clic *fuera* de él, y cada herramienta lo cierra a mano:

```
let stay_open = egui::containers::menu::MenuConfig::new()
    .close_behavior(egui::PopupCloseBehavior::CloseOnClickOutside);
let menu =
egui::containers::menu::MenuButton::new(tr("Tools")).config(stay_open);
menu.ui(ui, |ui| {
    // ...
    let mut item =
        |ui: &mut Ui, text: &str, tool: Tool, shortcut:
Option<KeyboardShortcut>| {
            // ...
            if ui.add(button).clicked() {
                picked = Some(tool);
                ui.close();
            }
        };
    item(ui, "Up a semitone", Tool::Transpose(1), Some(KEY_UP));
    // ...
});
```

`item` es un closure guardado en una variable y llamado muchas veces. Asigna a `picked`, una variable de fuera, así que toma prestado `picked` de forma mutable mientras exista el closure, y hay que declararlo `let mut item` (llamarlo cambia lo que capturó: es un `FnMut`). El `ui` se le pasa como argumento en cada llamada en lugar de capturarlo, porque el código entre las llamadas también usa `ui`, y un closure que lo retuviera lo bloquearía. Los menús que tienen un campo de texto, como el menú de sección de la lista de reproducción, se quedan abiertos de la misma manera, para que un clic en el campo no los cierre.

Notas fantasma: oír los otros canales

Un acorde a menudo se reparte entre instrumentos: el pad toca Do y Sol, el bajo toca Mi, y juntos forman C/E. Mirando solo el canal que se edita, el piano roll diría «C5» (un power chord) donde tú oyes C/E.

Así que el piano roll puede escuchar también a los otros canales. No a todos: los canales de batería tienen pasos simples, y un bombo no forma parte de la armonía. Un canal cuenta si alguna de sus notas no es un paso simple:

```
let others: Vec<usize> = (0..pattern.rows())
    .filter(|&other| other != self.channel)
    .filter(|&other| pattern.notes(other).iter().any(|note|
!note.is_step()))
    .collect();
let mut heard = vec![self.channel];
if self.ghosts {
    heard.extend(&others);
}
```

`heard` es la lista de canales cuyas notas cuentan para la armonía: este, y los otros cuando «Notas fantasma» está marcado. La línea que nombra el acorde bajo el puntero, las sugerencias de acorde siguiente y los nombres de las notas reciben todos `heard`, y reúnen sus notas con una pequeña función auxiliar:

```
fn sounding<'a>(
    pattern: &'a Pattern,
    channels: &'a [usize],
    step: usize,
) -> impl Iterator<Item = Note> + 'a {
    channels
        .iter()
        .flat_map(move |&channel| pattern.sounding_at(channel, step))
}
```

- `flat_map` llama al closure por cada canal, recibe cada vez un iterador de notas, y los une todos en un solo iterador, como si las notas de los canales fueran una sola lista.
- El resultado toma prestados `pattern` y `channels`, así que no debe sobrevivirlos. `<'a>` nombra ese préstamo, y `+ 'a` en el tipo de retorno dice «este iterador no vive más que `'a`». `analyze` acepta cualquier iterador de notas, así que lee las notas directamente del patrón, sin copiarlas antes a un `Vec`.
- `move` hace que el closure tome sus propias copias de lo que usa: `step` (un número) y `pattern` (una referencia, que es lo que se copia, no el patrón). Sin `move`, el closure tomaría prestada la variable `step`, que pertenece a `sounding` y desaparece cuando esta devuelve, mientras el iterador sigue vivo.

Las notas fantasma se pintan antes que las de este canal, de modo que las de este canal quedan siempre encima, en gris con un borde fino. Solo se dibujan: los clics miran únicamente las notas de este canal, así que las notas fantasma nunca pueden seleccionarse, moverse ni borrarse por accidente. Cuando ningún otro canal toca melodías o acordes, la casilla aparece en gris, y su texto al pasar el ratón dice por qué.

El carril de acordes: chunk_by, y luego medir

El carril sobre la cuadrícula nombra la armonía. Su primera fila nombra lo que toca este canal en cada paso; su segunda fila, **Todo**, nombra cómo suena ahí la canción entera, pulso a pulso, tal como lo calcula Oír la armonía. Cada fila une los pasos vecinos con la misma armonía en un solo bloque:

```
// `chunk_by` splits a slice wherever two neighbours differ. A
// harmony holds its bass, so C and C/E are two blocks.
let mut start = 0;
for run in harmonies.chunk_by(|a, b| a == b) {
    let end = start + run.len();
    if let Some(label) = run[0].label_in(scale) {
        // ...a block from step `start` to step `end`, with the label...
    }
    start = end;
}
```

`chunk_by` parte un slice en tramos allí donde el closure dice que dos vecinos no van juntos. Cada `run` es un slice del vector original, así que no se copia nada. Comparar armonías enteras, y no sus etiquetas, importa: todo grupo de notas sin nombre de acorde lleva la etiqueta «?», y unir por etiqueta convertiría varios acordes distintos sin nombre seguidos en un solo bloque «?» largo, como si se mantuviera un acorde.

Ejecutar `analyze` en cada paso y en cada frame parece mucho, pero son como mucho unos cientos de bucles pequeños sobre unas pocas notas. Mide antes de optimizar: aquí ni siquiera se nota.

Un nombre tiene que caber en su bloque. Cortado por los bordes, «E7/B» en un bloque estrecho mostraría «7/», que no es un nombre en absoluto, y para quien aprende armonía mirando la pantalla un nombre equivocado es peor que ninguno. egui dibuja el texto en dos pasos. Primero lo **compone**: elige el glifo de cada carácter en la fuente, y calcula dónde va cada uno y cuánto espacio ocupa el conjunto. El resultado es una *galley* (el nombre viene de la imprenta: la galera, una bandeja de tipos compuestos en líneas, lista para imprimir). Después pinta la galley. `painter.text(...)` hace las dos cosas a la vez; para decidir *antes* de dibujar, los dos pasos se separan:

```
let name = painter.layout_no_wrap(label, FontId::proportional(11.0),
LANE_TEXT);
if name.size().x <= chord_rect.width() - 2.0 {
    // A long chord's name stays in view while it scrolls.
    let shown = chord_rect.intersect(cells);
    painter.with_clip_rect(chord_rect).galley(
        shown.center() - name.size() / 2.0,
        name,
        LANE_TEXT,
    );
}
```

`layout_no_wrap` compone el texto en una sola línea, y `name.size()` es cuánto mide de ancho y de alto. Si cabe en el bloque, con un punto de margen a cada lado, se pinta esa misma galley; nada se compone dos veces. `galley` recibe la esquina superior izquierda del texto, así que el centro se corre hacia atrás la mitad del tamaño. Si no cabe, no se dibuja nada; apuntar al bloque sigue nombrando el acorde entero en la línea bajo la barra de herramientas.

Resaltado de la escala

`paint_rows` colorea cada fila preguntándole a la escala por su clase de altura, con un `match` sobre una tupla de dos `bool`:

```
let fill = match (is_root, in_scale) {
    (true, _) => ROOT_ROW,
    (false, true) => SCALE_ROW,
    (false, false) => OTHER_ROW,
};
```

Las teclas de la izquierda llevan un punto en las notas de la tonalidad (naranja para la tónica), y las notas fuera de la tonalidad se dibujan en rojo. La tonalidad es la de la canción allí donde suena el patrón (`project.pattern_key`), así que un patrón usado después de un cambio de tonalidad se muestra en la nueva. Cambia la tonalidad y todos los colores la siguen en el frame siguiente, porque nada se guarda en caché: el modo inmediato dibuja a partir del estado actual cada vez.

El carril de velocidad

Hacerle sitio

El carril está bajo la cuadrícula. La cuadrícula es un `ScrollArea` que llena la altura que se le da. Si se le diera toda la altura que queda en la ventana, empujaría el carril por debajo del borde; las ventanas de egui crecen para que quepa su contenido, así que la ventana crecería, la cuadrícula llenaría la nueva altura, el carril volvería a quedar fuera, y la ventana iría creciendo unos píxeles en cada frame. Así que a la cuadrícula se le da exactamente la altura que queda cuando el carril tiene su parte:

```
// The velocity lane goes below the grid, so the grid leaves room
// for it: the lane, the gap, and egui's spacing between widgets.
// Taking even a pixel too much would make the window grow a
// little on every frame.
let below = VELOCITY_HEIGHT + 4.0 + 2.0 * ui.spacing().item_spacing.y;
let grid_height = (ui.available_height() - below).max(96.0);
```

Lo fácil de olvidar es `item_spacing`: egui pone un pequeño hueco después de cada widget, aquí después del scroll area y después del espacio. Una manera simple de pillar una ventana que crece sola es hacer dos capturas de pantalla con unos segundos de diferencia y compararlas.

Fijar velocidades

El carril es un único widget que detecta clics y arrastres. Mientras el botón está pulsado sobre él, la altura del puntero fija la velocidad, y su posición x elige el paso:

```
if response.is_pointer_button_down_on()
    && !crate::widgets::dismissing(ui.ctx())
    && let Some(pos) = response.interact_pointer_pos()
    && (cells.contains(pos) || response.dragged())
{
    let step = lane.step_at(pos.x);
    let height = (lane.rect.bottom() - pos.y) / lane.rect.height();
    let velocity = (height * 127.0).round().clamp(1.0, 127.0) as u8;
    let selected_here = notes
        .iter()
        .any(|note| note.start == step && self.selection.contains(note));
    let old: Vec<NoteEvent> = notes
        .iter()
        .filter(|note| note.start == step && note.velocity != velocity)
        .filter(|note| !selected_here || self.selection.contains(note))
        .copied()
        .collect();
    let new = old.iter().map(|&note| NoteEvent { velocity, ..note
}).collect();
    self.replace(pattern, actions, "change velocity", old, new);
}
```

- La altura es una fracción de 0 (abajo) a 1 (arriba), escalada al 1 a 127 de MIDI. `clamp` la mantiene dentro incluso cuando el puntero sale del carril durante un arrastre. Convertir un float con `as` nunca da la vuelta (300.0 pasa a 255 y -5.0 a 0), pero 0 sería una nota que no se oye, así que el clamp asegura que la velocidad sea al menos 1.
- Un acorde tiene varias notas en el mismo paso. Cambian todas, salvo que algunas estén seleccionadas: entonces solo cambian las seleccionadas. Así se hace que la nota de arriba de un acorde suene más fuerte que las demás, como hacen los pianistas para destacar una melodía.
- Las notas que ya tienen esa velocidad quedan fuera de `old`, así que mantener el botón quieto no envía el mismo cambio en cada frame.

Barrer el carril cambia cada paso a medida que el puntero pasa por encima, y todo el barrido es un solo paso de deshacer, como cualquier arrastre.

La velocidad también se ve en la cuadrícula. El color de una nota se multiplica por un factor entre 0,55 y 1 según su velocidad (`fill.gamma_multiply(0.55 + 0.45 * f32::from(note.velocity) / 127.0)`), así que las notas suaves son más oscuras, pero nunca tanto como para desaparecer.

Encontrar tus notas

El piano roll muestra un canal de un patrón, y esa pareja puede estar vacía mientras la canción suena de lo lindo: los acordes de una canción pueden estar en «Acordes», no

en «Ritmo». Una cuadrícula vacía sin una palabra de explicación parece notas perdidas. Así que el piano roll hace dos cosas.

Primero, sigue a las notas. Cuando se abre, o cambia el patrón, y el canal mostrado no tiene notas ahí, pasa al canal con más notas de melodía o de acordes:

```
pub fn busiest_channel(pattern: &Pattern) -> Option<usize> {
    (0..pattern.rows())
        .map(|channel| {
            let melodic = pattern.notes(channel).iter().filter(|note|
!note.is_step());
            (channel, melodic.count())
        })
        .filter(|&(_, count)| count > 0)
        .max_by_key(|&(channel, count)| (count,
std::cmp::Reverse(channel)))
        .map(|(channel, _)| channel)
}
```

`max_by_key` devuelve el *último* de varios máximos iguales. Comparar la tupla `(count, Reverse(channel))` deshace los empates hacia el número de canal más bajo: `std::cmp::Reverse` envuelve un valor e invierte su orden. Los pasos de batería se dejan fuera a propósito: en un patrón de solo batería, el piano roll no salta al bombo, cuyas «notas» son todas el mismo paso simple. Un canal que elegiste tú, en el rack o en el editor, se conserva.

Para notar que la ventana *se abre*, el piano roll recuerda si estaba abierta en el frame anterior. `open` es un campo público que cambian el botón de la barra de herramientas, F7 y el menú Ver, así que este es el único sitio que ve todas las maneras de entrar:

```
let opened = self.open && !self.was_open;
// ... at the end of `show`:
self.was_open = open;
```

Al abrirse, también se desplaza al centro de las notas. El centro no se calcula partiendo las alturas por la mitad: con enteros, `/` tira el resto, y `41 / 2 + 67 / 2` da 53, una fila por debajo del centro de verdad, 54. En cambio, le pide a egui que centre todo el bloque de filas, desde arriba de la nota más aguda hasta abajo de la más grave:

```
let top = grid.row_top(highest.clamp(LOWEST, HIGHEST));
let bottom = grid.row_top(lowest.clamp(LOWEST, HIGHEST)) + row_height;
let target = Rect::from_min_max(pos2(rect.left(), top), pos2(rect.left() +
1.0, bottom));
ui.scroll_to_rect(target, Some(Align::Center));
```

Segundo, cuando el canal de verdad no tiene notas en el patrón, lo dice sobre la cuadrícula, con botones para ir adonde están las notas:



Encontrar los patrones donde toca el canal es un filtro sobre todos ellos, que conserva el índice (para seleccionarlo) y el nombre (para mostrarlo):

```
let elsewhere: Vec<(usize, &str)> = project
    .patterns()
    .iter()
    .enumerate()
    .filter(|&(index, other)| {
        index != self.pattern && !other.notes(self.channel).is_empty()
    })
    .map(|(index, other)| (index, other.name.as_str()))
    .collect();
```

El `&str` del vector toma prestado de `project`. Eso está bien porque el vector solo vive mientras se dibuja este frame, y `project` dura más. Un botón para otro patrón no puede seleccionarlo directamente: el patrón seleccionado pertenece a la app. Así

que devuelve `RollAction::SelectPattern`, igual que cualquier otro cambio sale del piano roll.

El mensaje se coloca *sobre* la cuadrícula en lugar de ocupar una línea encima, para que la cuadrícula no salte hacia arriba cuando se dibuja la primera nota y el mensaje desaparece. Se dibuja después de la cuadrícula, en un `Ui` hijo colocado encima, y la cuadrícula recuerda dónde se dibujó en el frame anterior (`self.banner`) y deja en paz los clics de ahí: un clic en uno de sus botones no debe dibujar también una nota debajo.

Conectarlo en la aplicación

La app tiene un `PianoRoll` y, mientras dibuja las ventanas, llama:

```
let roll_actions = self.piano_roll.show(
    ctx,
    &self.project,
    self.pattern,
    playhead,
    &held,
    area,
    roll_zoom,
);
for action in roll_actions {
    self.apply_roll(action);
}
```

`self.piano_roll.show(..., &self.project, ...)` toma prestado un campo de forma mutable y otro de forma inmutable a la vez, cosa que el borrow checker permite porque son campos distintos. Después `apply_roll` cambia el patrón con `edit_pattern`, como hace el rack:

```
RollAction::ReplaceNotes { channel, old, new, label } => {
    self.edit_pattern(label, |pattern| {
        pattern.replace_notes(channel, &old, &new);
    });
}
```

Cuando se quita un canal, `PianoRoll::channel_removed` o bien suelta su canal (ya no existe) o bien baja en uno su número de canal, para seguir mostrando el mismo canal; `channel_inserted` hace lo contrario cuando deshacer devuelve un canal.

En una ventana baja, de menos de 420 puntos, la cuadrícula va primero: las líneas de ayuda bajo la barra de herramientas pasan al tooltip de la línea que nombra el acorde, así que una pantalla pequeña sigue mostrando una docena de filas de notas.

La lista de reproducción

Una rejilla grande: `show_viewport`

La lista de reproducción puede ser mucho más grande que la ventana: decenas de compases, y tantas pistas como use la canción. Va dentro de un scroll area. Pero algunas partes *no* deben desplazarse: los nombres de las pistas a la izquierda, y los números de compás y las franjas de encima arriba.

`ScrollArea::show_viewport` lo hace posible. Su closure recibe la parte visible del contenido, como un rectángulo medido desde la esquina superior izquierda del contenido:

```
let output = area.show_viewport(ui, |ui, viewport| {
    self.grid(ui, viewport, project, current, playhead, &mut actions)
});
```

`grid` reserva el tamaño entero del contenido, y luego calcula dónde está en pantalla la parte visible:

```
let (rect, response) = ui.allocate_exact_size(size,
Sense::click_and_drag());

// `viewport` is the visible part, measured from the top-left of the
// content. Moving it by the content's position on screen gives the
// visible part in screen coordinates, where everything is drawn.
let visible = viewport.translate(rect.min.to_vec2());
```

`rect` es el contenido entero en coordenadas de pantalla (su esquina superior izquierda se mueve hacia arriba y a la izquierda al desplazarte). Sumar esa esquina a `viewport` da `visible`, la parte de la pantalla donde se ve el contenido. Entonces:

- las pistas y los clips se dibujan en su lugar dentro del contenido, así que se desplazan;
- los nombres de las pistas se dibujan en `visible.left()`, y la regla y las franjas en `visible.top()`, así que se quedan quietos mientras los clips se deslizan por debajo.

Se dibujan *después* de los clips, así que los tapan.

La aritmética de pasos, pistas y píxeles vive en un pequeño struct `Copy`, como en el piano roll:

```rust
#[derive(Debug, Clone, Copy)]
struct Layout<'a> {
    /// The top-left corner of step 0 of track 0.
    origin: Pos2,
    /// How wide a step is and how tall a track is: the zoom.
    step_width: f32,
    track_tops: &'a [f32],
    tracks: usize,
    /// Whether there is an empty row below the last track, where a clip
    /// starts a new track. There is unless the playlist is full.
    spare: bool,
}
```

Cada pista puede tener su propia altura, así que `Layout` guarda dónde empieza cada pista (`track_tops`), calculado una vez por frame. Para encontrar la pista bajo el puntero, `cell_at` usa `partition_point`, una búsqueda binaria en esa lista ordenada: el número de comienzos que quedan por encima del puntero, menos uno, es la pista.

Widgets reales en una zona pintada: `ui.put`

Todo en la rejilla se pinta a mano, pero los nombres de las pistas deben ser campos de texto editables y las luces de silencio deben ser la misma luz que en el rack. `ui.put` coloca cualquier widget en el rectángulo que elijas:

```rust
let light_response = ui
    .put(light, |ui: &mut Ui| widgets::mute_light(ui, enabled))
    .on_hover_text(tr("Click to mute or unmute the track"));

let shown = tr_name(&track.name).into_owned();
let mut name = shown.clone();
let field = egui::TextEdit::singleline(&mut name)
    .frame(egui::Frame::NONE)
    .text_color(HEADER_TEXT);
let name_response = ui.put(name_rect, field);
if name_response.changed() && name != shown {
    actions.push(PlaylistAction::RenameTrack { track: index, name });
}
```

- Un closure `|ui: &mut Ui| -> Response` cuenta como widget: egui implementa su trait `Widget` para cualquier closure así. Así es como un widget con forma de función, como `mute_light`, se puede pasar a `put`.
- El campo de texto edita una *copia* del nombre. Si cambió, la copia va en una acción `RenameTrack`, y la app cambia el proyecto. La lista de reproducción mantiene la regla de que nunca cambia el proyecto ella misma.

- `Frame::NONE` quita el borde del campo, así que parece una etiqueta hasta que haces clic en él.

Colocar, mover y estirar clips

Colocar, mover y borrar clips sigue el modelo del piano roll: una pulsación decide qué es el arrastre, y el arrastre sigue al puntero hasta que se suelta el botón.

```
struct ClipDrag {
    index: usize,
    grab: usize,
    stretch: bool,
}
```

Una pulsación en un espacio vacío coloca un clip del patrón seleccionado y empieza a moverlo; una pulsación en un clip empieza a mover ese clip y selecciona su patrón; una pulsación en el borde derecho de un clip empieza a estirarlo, lo que repite su patrón tantas veces como quepan hasta el puntero. La pulsación se atiende donde bajó el botón:

```
// A press is handled where the button went down. A quick hand can
// press and move in the same frame, and the clip or empty cell
// under the pointer *now* may not be the one that was clicked.
// A click that closes a menu does nothing else.
let dismissing = widgets::dismissing(ui.ctx());
let pressed_at = press_origin.filter(|pos| {
    primary_pressed && !dismissing && response.hovered() &&
cells.contains(*pos)
});
if let Some(pos) = pressed_at {
    let (track, step) = layout.cell_at(pos);
    match clip_at(project, layout, pos) {
        Some(index) => { /* start moving or stretching that clip */ }
        None => { /* add a clip of the selected pattern, and start moving
it */ }
    }
}
```

`Option::filter` conserva el valor solo si el closure devuelve `true`, así que `pressed_at` es `Some` solo en el frame de una pulsación, sobre la rejilla.

Las posiciones se ajustan a la rejilla que elijas en la barra de herramientas: un compás, un pulso o un paso. Dónde caen compases y pulsos depende del compás en ese punto de la canción, así que el ajuste le pregunta al `Meter` de la canción:

```
/// The snap point at or before `step`. Where the bars and beats are
/// depends on the time signature there: a bar of 3/4 is 12 steps, a
/// beat of 6/8 is 6.
fn floor(self, step: usize, meter: &Meter) -> usize {
    match self {
        Snap::Bar => meter.floor_to_bar(step),
        Snap::Beat => meter.floor_to_beat(step),
        Snap::Step => step,
    }
}
```

Ctrl o Mayús y un clic añaden un clip a la selección, Ctrl o Mayús y un arrastre en un espacio vacío dibujan un recuadro alrededor de clips, y arrastrar uno de varios clips seleccionados los mueve todos, sin salirse de la lista de reproducción.

Doble clic en un clip para editarlo

En la mayoría de los programas de música, haces doble clic en un clip para editarlo. egui cuenta los clics:

```
let double_clicked = ui.input(|input| {
    input
        .pointer
        .button_double_clicked(egui::PointerButton::Primary)
});
if double_clicked
    && !dismissing
    && response.hovered()
    && let Some(pos) = pointer.filter(|pos| cells.contains(*pos))
    && let Some(index) = clip_at(project, layout, pos)
{
    actions.push(PlaylistAction::EditClip(clips[index].pattern));
}
```

La app elige el editor: el piano roll, en el canal con más notas, cuando el patrón tiene melodías o acordes; el rack de canales cuando solo tiene batería. En los dos casos la ventana pasa al frente, y se lleva el teclado:

```
fn bring_window_to_front(&mut self, ctx: &egui::Context) {
    if let Some(id) = self.bring_to_front.take() {
        let layer = egui::LayerId::new(egui::Order::Middle, id);
        ctx.move_to_top(layer);
        // Its keys go with it: Delete after double-clicking a clip
        // deletes notes in the piano roll, not clips in the playlist.
        widgets::give_keyboard(ctx, layer);
    }
}
```

La capa de una ventana se nombra con el id que se le dio a la ventana, y por eso el rack y el piano roll obtienen el suyo de pequeñas funciones (`rack_id()` y `PianoRoll::id()`) en lugar de escribir el mismo `Id::unique(...)` en varios sitios.



Dibujar las notas de un clip: `Box<dyn Fn>`

Cada clip muestra un esbozo de su patrón. Los patrones de batería se ven mejor como un secuenciador de pasos diminuto (una línea por canal), y cualquier cosa con melodía como un piano roll diminuto (las notas agudas más arriba). Las dos necesitan maneras distintas de convertir una nota en una altura, elegidas una vez por clip:

```
// Turns a channel and a note into a height from 0.0 (bottom) to 1.0
(top).
let height: Box<dyn Fn(usize, u8) -> f32> = if steps_only {
    let used: Vec<usize> = (0..channels)
        .filter(|&channel| !pattern.notes(channel).is_empty())
        .collect();
    let count = used.len().max(1) as f32;
    Box::new(move |channel, _| {
        let row = used.iter().position(|&c| c == channel).unwrap_or(0) as
f32;
        1.0 - (row + 0.5) / count
    })
} else {
    // ...the lowest and highest note, as in the rack's mini piano roll...
    Box::new(move |_, midi| ((midi - lowest) as f32 + centering) / span)
};
```

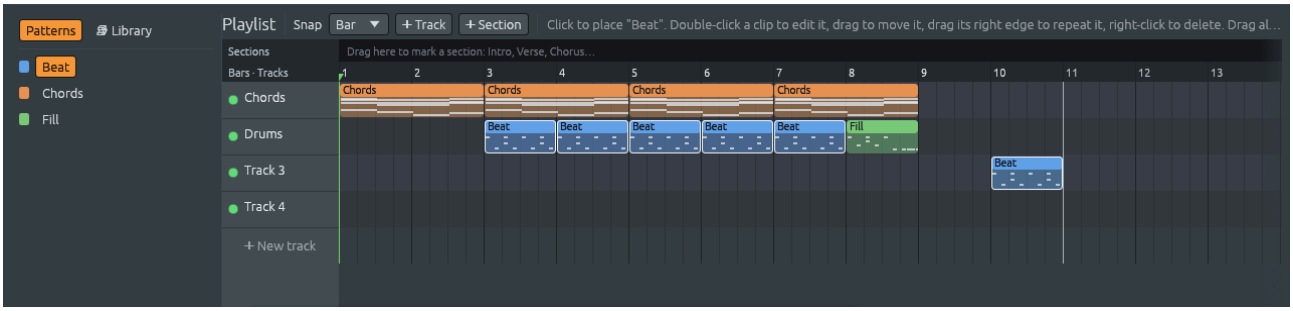
Cada closure tiene su propio tipo, sin nombre, así que las dos ramas del `if` producen valores de tipos distintos, y una variable solo puede tener un tipo. `Box<dyn Fn(usize, u8) -> f32>` es el tipo común: «un closure en el montículo que recibe un canal y una altura y devuelve una altura en pantalla». `dyn Fn` es un trait object, como `dyn App` en `main.rs`. La palabra clave `move` hace que cada closure se quede con lo que usa (`used`, `lowest`…), que debe sobrevivir al `if` que los creó.

Esto se ejecuta en el hilo de la ventana, una vez por clip y por frame, así que la pequeña reserva de memoria del `Box` no importa. En el hilo de audio sí importaría.

El color de un clip es el de su patrón. Cada patrón guarda un `color`, un lugar en la paleta de la lista de reproducción, guardado en el archivo del proyecto. No sale del número del patrón, porque entonces borrar el primer patrón cambiaría el color de todos los demás.

Solo las pistas que usas

Un proyecto tiene solo las pistas hasta su último clip: Archivo > Nuevo tiene una pista, una demostración tiene las que usa. Debajo de la última pista hay una **fila de reserva** más tenue: coloca un clip en ella (o arrastra uno hasta ella) y se convierte en una pista. Su cabecera tiene también un botón ➕ **Nueva pista**.



`Project::clip_fits` decide si un clip puede ir en un sitio. Se permite una fila más que las pistas, la que está justo debajo de la última pista, siempre que la lista de reproducción no esté llena:

```
fn clip_fits(&self, clip: Clip) -> bool {
    let tracks = self.playlist.tracks.len();
    clip.pattern < self.patterns.len()
        && (clip.track < tracks || (clip.track == tracks && tracks <
MAX_TRACKS))
        && clip.start < MAX_SONG_STEPS
        && (1..=MAX_REPEATS).contains(&clip.repeats)
}
```

`add_clip` y `move_clip` llaman después a `open_track`, que añade una pista cuando el clip está en esa fila nueva. Como la pista y el clip se añaden en la misma edición, un solo Ctrl+Z quita los dos. La vista de la lista de reproducción solo tiene que dibujar una fila más: `Layout` tiene su indicador `spare` y un método `rows()`, y `cell_at` limita a `rows()`.

Las demostraciones se construyen sobre una lista de reproducción de dieciséis pistas, porque nombrar la pista 0 y la pista 1 es más fácil cuando existen; al final, `remove_unused_tracks` quita las pistas vacías de abajo. Los archivos guardados antes de que se recortaran las pistas necesitan casi lo mismo, con una diferencia: una pista que alguien renombró o silenció se dejó así a propósito, así que al cargar solo se quitan las pistas que están exactamente como empieza una pista nueva. Dos reglas, un bucle, que recibe la regla como closure:

```
pub fn remove_unused_tracks(&mut self) {
    self.trim_tracks(|_, _| true);
}

pub(crate) fn remove_blank_tracks(&mut self) {
    self.trim_tracks(|index, track| *track == Track::numbered(index + 1));
}

fn trim_tracks(&mut self, drop: impl Fn(usize, &Track) -> bool) {
    // ...from the bottom, pop tracks with no clips while `drop` says
so...
}
```

`impl Fn(usize, &Track) -> bool` significa «cualquier función o closure que reciba un número de fila y una pista y responda sí o no». El compilador hace una copia de `trim_tracks` para cada closure, así que esto no cuesta nada en tiempo de ejecución. Comparar con `Track::numbered(index + 1)` funciona porque `Track` deriva `PartialEq`: una pista en blanco es igual a una completamente nueva para su fila.

Recortar solo quita pistas **por debajo** del último clip. Una pista vacía en medio se queda, porque quitarla subiría una fila todos los clips de debajo.

Insertar, mover y borrar pistas

Un clip nombra su pista por **número**. Inserta una pista en la fila 2, y todo clip en la fila 2 o más abajo debe bajar una, o la batería de repente quedaría en la pista equivocada:

```
pub fn insert_track(&mut self, index: usize) -> Option<usize> {
    let tracks = &mut self.playlist.tracks;
    if tracks.len() >= MAX_TRACKS {
        return None;
    }
    let index = index.min(tracks.len());
    tracks.insert(index, Track::numbered(tracks.len() + 1));
    for clip in &mut self.playlist.clips {
        if clip.track >= index {
            clip.track += 1;
        }
    }
    Some(index)
}
```

`remove_track` es la imagen en espejo: quita los clips de la pista con `retain`, y luego sube una fila los de debajo. `swap_tracks` intercambia dos filas y los clips que hay en ellas. Quien cambia la lista arregla los números que apuntan a ella, y por eso los

campos son privados y solo los métodos de `Project` los cambian (ver El proyecto). Cada uno de ellos es una sola llamada dentro de `edit`, así que deshacer devuelve una pista borrada con sus clips.



El menú es el `context_menu` de egui sobre el campo del nombre de la pista. Un campo de texto sigue abriendo un menú con el clic derecho, porque escribir solo necesita el botón izquierdo. El borde inferior de una pista se puede arrastrar para hacerla más alta o más baja, y un doble clic en él le devuelve su altura habitual.

Seguir el cursor de reproducción más allá de los nombres

Con **Seguir el cursor de reproducción** activado, la lista de reproducción se desplaza para mantener el cursor a la vista. Cuando una canción llega al final y vuelve a empezar, o un bucle vuelve a su inicio, el cursor salta a la izquierda y la vista lo sigue. Pero los nombres de las pistas de la izquierda se dibujan *sobre* los compases, y para el scroll area no existen: mostrar el cursor podría dejarlo debajo de ellos. Así que a `follow_x` se le dice cuánto mide la parte fija, y el rectángulo que muestra llega esa distancia más a la izquierda:

```
pub fn follow_x(ui: &Ui, step: Option<usize>, x: f32, view: Rect, covered:
f32) {
    // ...only when the step changed, and Follow playhead is on...
        let margin = (view.width() * 0.1).min(60.0);
        let y = view.center().y;
        ui.scroll_to_rect(
            Rect::from_min_max(
                pos2(x - margin - covered, y - 0.5),
                pos2(x + margin, y + 0.5),
            ),
            None,
        );
}
```

`scroll_to_rect` se desplaza lo menos posible para que el rectángulo quede dentro de la vista. Hacia la izquierda, eso deja el borde izquierdo del rectángulo en el borde izquierdo de la vista, `covered` puntos a la izquierda del cursor, así que el cursor acaba a la derecha de los nombres. La lista de reproducción pasa `HEADER_WIDTH` (sus nombres), el piano roll `KEY_WIDTH` (su teclado). Hacia la derecha no cambia nada. Solo un paso nuevo desplaza la vista, así que puedes seguir desplazándote a mano mientras suena la música.

Probar las ventanas

Las herramientas y el proyecto están cubiertos por pruebas unitarias, y muchos comportamientos de los editores por pruebas que dibujan la ventana sin pantalla (ver Probar una ventana sin pantalla). Pero algunos problemas solo aparecen cuando una persona usa la ventana: una ventana que crece un poco en cada frame, un menú que se cierra al elegir una opción dentro de él, un acorde que cae por encima del puntero en lugar de alrededor. Es un buen hábito para cualquier código de interfaz: escribe la lógica para que se pueda probar sin la ventana, y después prueba la ventana igualmente.

Dos idiomas: traducciones, y letras o solfeo

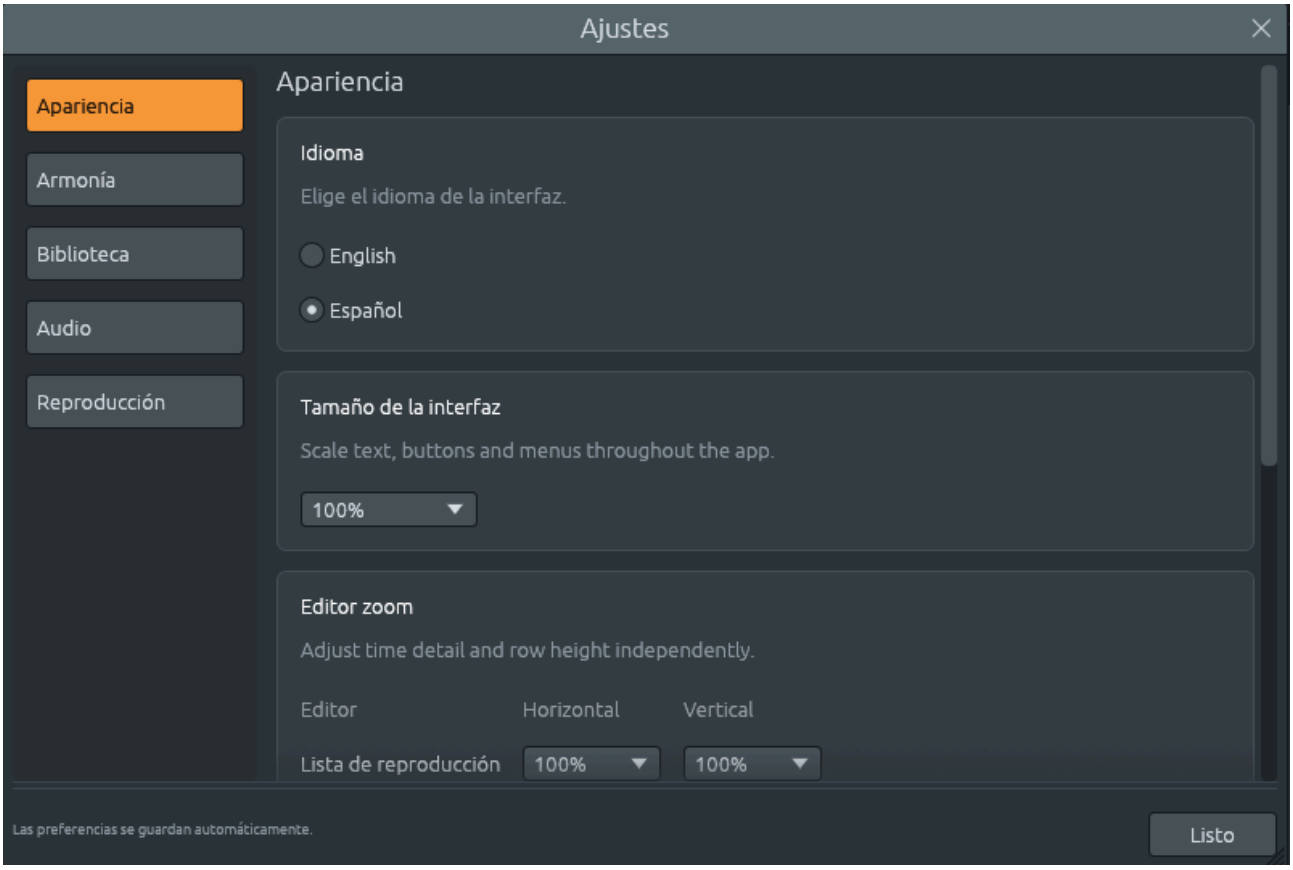
Archivos: `crates/lineup-core/src/i18n.rs`, `crates/lineup-core/src/i18n/es.rs`, `crates/lineup-app/src/i18n.rs`, `crates/lineup-core/src/pitch.rs`, `crates/lineup-core/src/harmony/symbol.rs`, `crates/lineup-core/src/chord_sheet.rs`, `crates/lineup-core/src/file.rs` y `crates/lineup-app/src/app/settings.rs`

Dos ajustes cambian cómo te habla Ferrous Lineup:

- **Edición > Ajustes… > Apariencia > Idioma** (con la app en inglés, **Edit > Settings… > Appearance > Language**) escribe los menús, las ventanas y los ajustes en **English** o en **Español**.
- **Edición > Ajustes… > Armonía > Nomenclatura de notas** (en inglés, **Harmony > Note naming**) nombra las notas, las tonalidades y los acordes con **letras** (C, F#, Bb major, Am7) o en **solfeo** (Do, Fa#, Sib mayor, Lam7).

Son independientes: puedes tener los menús en español con letras, o los menús en inglés con solfeo. Pero mientras no elijas una nomenclatura, sigue al idioma: con los menús en español las notas se nombran en solfeo, como en las escuelas de habla hispana, y con los menús en inglés, con letras. Cuando eliges una, se queda, cambies después al idioma que cambies. Los dos ajustes cambian solo lo que ves. Una canción guardada con una elección se abre exactamente igual para quien tenga la otra.

Con el español elegido, todo lo que escribe la app está en español: los menús, las ventanas, las ayudas, los avisos y los nombres de las demos, los estilos, los instrumentos y los canales. En inglés solo quedan los mensajes que llegan del sistema operativo o de una biblioteca (por ejemplo, por qué no se pudo abrir un dispositivo de audio) y los nombres de géneros que el español usa tal cual, como swing o funk. Este libro nombra los menús, las ventanas y los botones como los escribe la app en español. La mayoría de las capturas de este libro se hicieron con la app en inglés.



Las notas Do, Re, Mi

En inglés y en alemán, las notas se nombran con letras. En español, italiano, francés y portugués, y en los países que los hablan, se nombran con sílabas:

```
letras    C    D    E    F    G    A    B
solfeo    Do   Re   Mi   Fa   Sol  La   Si
```

Los sostenidos y los bemoles se escriben detrás de la sílaba igual que detrás de la letra: Fa sostenido es **Fa#** y Si bemol es **Sib**. Una tonalidad se nombra igual, así que Si♭ mayor es **Sib mayor** (o **Sib major** con los menús en inglés), y el modo dórico sobre Re es **Re dórico**.

Esto es el **Do fijo**: Do es siempre la misma nota, sea cual sea la tonalidad. Algunos métodos de entrenamiento auditivo usan un **do móvil**, en el que *do* es la tónica de la tonalidad (en Sol mayor, *do* es Sol). La app usa siempre el Do fijo, como las escuelas de España y América Latina, porque el nombre de una nota no debe cambiar con la tonalidad.

Los números de octava no cambian: el Do central es C4 con letras y **Do4** en solfeo, y cada octava va de Do a Si. Algunos libros franceses y españoles numeran las octavas

una más abajo y llaman Do3 al Do central. La app mantiene una sola numeración, así que una nota conserva su número la nombres como la nombres.

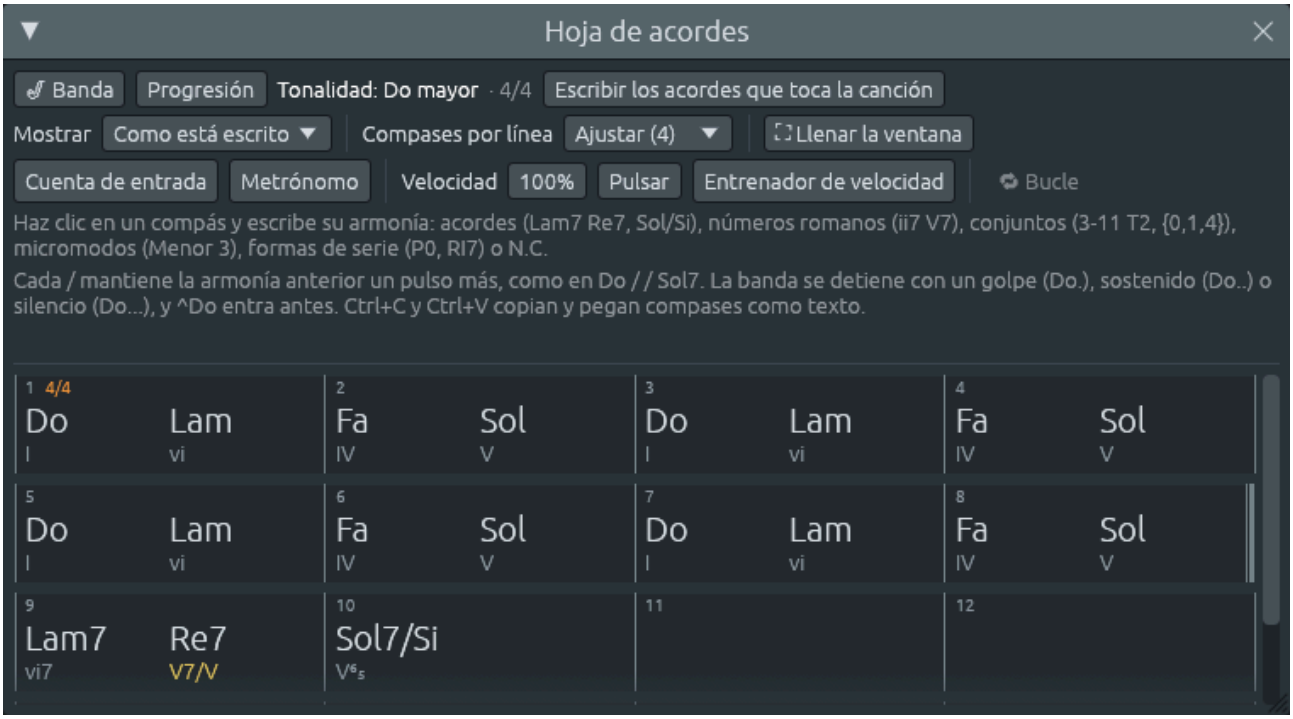


Los acordes en solfeo

Un cifrado es una fundamental, una cualidad y, a veces, una nota en el bajo. En solfeo solo cambian los nombres de las notas:

```
letras    C    Am7     G7/B     Bdim     F#m7b5    Eb+
solfeo    Do   Lam7    Sol7/Si  Sidim    Fa#m7b5   Mib+
```

A la primera manera se le llama *cifrado americano* y a la segunda *cifrado latino*. Las partituras de jazz y de pop suelen ir en cifrado americano, también en español, y por eso este libro escribe así los cifrados; muchos profesores y cancioneros usan el latino. Con el solfeo elegido, la hoja de acordes, las teclas de armonía, el piano roll, los cambios de tonalidad de la lista de reproducción y los ejemplos de las ayudas escriben las notas, las tonalidades y los acordes de la segunda manera.



Escribir un acorde de las dos maneras

Puedes escribir un acorde de cualquiera de las dos maneras, esté elegida la que esté: `Am7` y `Lam7` son el mismo acorde. Al pulsar **Intro**, el compás lo muestra como el ajuste nombra las notas.

Unos pocos nombres se pueden leer de las dos maneras. `o` es una forma de escribir un acorde disminuido (`Bo` es B°), así que `Do` podría ser Do mayor (solfeo) o Re disminuido (letras), y `Do7` podría ser C7 o D°7. La app lee esos nombres como el ajuste nombra las notas: con el solfeo elegido, `Do` es Do mayor. Los nombres que solo tienen sentido de una manera se leen de esa manera: `Faug` es Fa aumentado aunque esté elegido el solfeo, porque «Fa» seguido de «ug» no es un acorde, y `Fam` es Fa menor aunque estén elegidas las letras.

La cualidad se escribe igual en los dos: Fa aumentado es `Faaug` en solfeo. Se lee bien, pero `Fa+` se ve mejor, y la app también lo lee.

En la hoja de acordes, escribir sobre un compás seleccionado empieza a escribir en él, pero **L** es la tecla que cambia entre patrón y canción. Con el solfeo elegido, los acordes de La empiezan por L, así que ahí L empieza a escribir. Con las letras elegidas, ningún acorde empieza por L, y L cambia de modo como siempre.

Qué se guarda

Los proyectos, los estilos guardados, el texto de la hoja de acordes que copia **Ctrl+C** y las trazas de reproducción se escriben siempre con letras, y en inglés. Los ajustes no llegan nunca a ellos. Así que una canción que guardaste viendo Lam7 contiene `Am7`, y se abre con Am7 o con Lam7 según los ajustes de quien la abra.

En el código

Dos ajustes, guardados como el de los números romanos

`Language` y `NoteNaming` (los dos en `lineup-core`) se guardan como el ajuste de los números romanos en menor: un enum con un `#[default]`, un `Cell` dentro de `thread_local!` que toda función puede leer, y un `with` que lo fija un rato y lo devuelve con un guard en un `Drop`, incluso tras un panic.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Default, Serialize,
Deserialize)]
pub enum NoteNaming {
    #[default]
    Letters,
    Solfege,
}

thread_local! {
    static NOTE_NAMING: Cell<NoteNaming> = const {
Cell::new(NoteNaming::Letters) };
}
```

Una nota se escribe en cientos de lugares, a través de su implementación de `Display` (`{}` en `format!`). Un valor por hilo permite que todos sigan el ajuste sin pasarlo hacia abajo por cada función:

```
impl fmt::Display for NoteName {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.write_str(&self.format(NoteNaming::current()))
    }
}
```

La ventana fija los dos valores a partir de los ajustes guardados antes de dibujar cada fotograma:

```
pub fn apply_settings(ctx: &egui::Context) {
    language(ctx).set();
    note_naming(ctx).set();
}
```

Una exportación de video dibuja sus vistas en un hilo propio, que empieza con los valores por defecto, así que se lleva el idioma y la nomenclatura consigo, como ya se llevaba el ajuste de los números romanos.

Solfeo hasta que elijas

egui guarda las preferencias de la ventana como valores bajo un `Id`, y `get_persisted` da `None` para un `Id` bajo el que no se guardó nada. Ese `None` es lo que distingue «nunca elegido» de una elección, así que la nomenclatura recurre al idioma solo entonces:

```
pub fn note_naming(ctx: &egui::Context) -> NoteNaming {
    ctx.data_mut(|data| data.get_persisted(note_naming_id()))
        .unwrap_or_else(|| language_naming(language(ctx)))
}

/// How notes are named where `language` is spoken.
fn language_naming(language: Language) -> NoteNaming {
    match language {
        Language::English => NoteNaming::Letters,
        Language::Spanish => NoteNaming::Solfege,
    }
}
```

Leer nunca guarda nada; solo lo hace `set_note_naming`, que se llama cuando haces clic en una nomenclatura en los Ajustes. Así que hasta ese clic cada fotograma vuelve a preguntar al idioma, y pasar al español pasa las notas al solfeo. El `match` no tiene un brazo `_` a propósito: un tercer idioma no compilaría hasta que alguien diga cómo nombra las notas.

Traducir: la clave es el texto en inglés

Cada texto que muestra la interfaz está escrito en inglés en el código y pasa por `tr`. Vive en `lineup-core`, no en la app, para que también se traduzcan los textos que escribe el núcleo, los nombres de los estilos, los instrumentos, los acordes y las escalas:

```
pub fn tr(text: &'static str) -> &'static str {
    match Language::current() {
        Language::English => text,
        Language::Spanish => in_spanish(text).unwrap_or_else(||
missed(text)),
    }
}
```

El catálogo en español está en `i18n/es.rs`: un gran `match` para los menús, las ventanas principales y los ajustes, y, en los archivos de `i18n/es/`, tablas de pares por área (la app, la teoría, los instrumentos, la biblioteca, las canciones), reunidas una sola vez en un `HashMap`:

```
static TEXTS: LazyLock<HashMap<&'static str, &'static str>> =
    LazyLock::new(|| TABLES.into_iter().flatten().copied().collect());

/// `text` in Spanish, or `None` for a text with no translation yet.
pub fn translate(text: &str) -> Option<&'static str> {
    main_texts(text).or_else(|| TEXTS.get(text).copied())
}
```

`LazyLock` construye el mapa la primera vez que se usa, en el hilo que lo pida primero, y a partir de ahí todos los hilos lo comparten. `flatten` convierte las cinco tablas en una sola serie de pares, y `collect` construye el mapa con ellos.

Usar el propio texto en inglés como clave tiene dos ventajas. El código se sigue leyendo con naturalidad (`tr("Settings")`, no `tr(SETTINGS_TITLE)`), y un texto que aún no tiene traducción se queda simplemente en inglés, gracias a `unwrap_or_else`. El precio es que cambiar un texto en inglés obliga a cambiar también su clave en el catálogo, o se pierde el español; una prueba lo detecta (ver más abajo).

Tanto el argumento como el resultado son `&'static str`: texto escrito en el propio programa, que vive tanto como el programa. Así `tr` puede devolver cualquiera de los dos sin copiar nada.

Unos cuantos parientes de `tr` cubren los demás casos:

- `trf("{} steps a bar", &[&12])` pone sus argumentos donde el texto traducido tiene `{}`, así que el español puede poner el número entre otras palabras: «12 pasos por compás».
- `trc("clef", "Bass")` es para una palabra cuya traducción depende de dónde se muestra: la clave de fa es «Fa», el instrumento «Bajo». El catálogo la tiene como `"clef|Bass"`.
- `tr_name` traduce los nombres que la app da a las cosas, un canal, una pista o un patrón: «Bass» es «Bajo», «Pattern 12» es «Patrón 12» y «A drums» es «Batería de

A». Un nombre que escribiste tú se queda como lo escribiste.
- Un texto del catálogo puede nombrar notas entre llaves, como en `"{Bb} mayor"`; se nombran como dice el ajuste (ver Ejemplos que siguen el ajuste). Un texto así se construye una vez para cada nomenclatura y se conserva, con `Box::leak`, que convierte un `String` en un `&'static str` que nunca se libera. Hay un número limitado de textos y dos nomenclaturas, así que la memoria conservada tiene un límite.

Nombres en pantalla, letras en los archivos

Como `Display` sigue el ajuste, todo lo que convierta una nota en texto y lo guarde guardaría solfeo. Los lugares por donde los datos salen del programa trabajan con letras, a través de `with`:

```
pub fn to_text(project: &Project) -> Result<String, FileError> {
    // …
    let file = Language::English.with(|| NoteNaming::Letters.with(||
ProjectFile::from(project)));
    let mut text = NoteNaming::Letters.with(||
ron::ser::to_string_pretty(&file, style))?;
    // …
}
```

Un marcador de acorde se guarda como su texto, mediante los atributos de serde `into = "String"` y `try_from = "String"` (ver Series dodecafónicas). Sus conversiones también trabajan con letras, así que un marcador se guarda como `Am7` lo guarde quien lo guarde:

```
impl From<HarmonyName> for String {
    fn from(harmony: HarmonyName) -> Self {
        NoteNaming::Letters.with(|| harmony.to_string())
    }
}
```

Los estilos guardados, el texto de la hoja de acordes y la traza de reproducción hacen lo mismo. Los cifrados de la propia biblioteca están escritos con letras en el código fuente, así que también se leen con letras.

Leer un nombre de las dos maneras

`NoteName::read_as` lee una nota al principio de un texto con una nomenclatura, y devuelve el nombre y el resto del texto. Para el solfeo prueba cada sílaba:

```
NoteNaming::Solfege => Letter::ALL.into_iter().find_map(|letter| {
    text.strip_prefix(letter.solfege())
        .map(|rest| (letter, rest))
})?,
```

`strip_prefix` devuelve el texto que sigue al prefijo si el texto empieza por él, y `None` si no. `find_map` devuelve el primer `Some`. «Sol» y «Si» empiezan los dos por S, pero ninguno es el principio del otro, así que solo uno puede coincidir.

`read_either` prueba las dos nomenclaturas, primero la del ajuste, y se queda con la primera lectura cuyo resto encaja (`fits`):

```
pub fn read_either<'a>(
    text: &'a str,
    fits: impl Fn(&'a str) -> bool,
) -> Option<(NoteName, &'a str)> {
    NoteNaming::current()
        .reading_order()
        .into_iter()
        .filter_map(|naming| NoteName::read_as(text, naming))
        .find(|&(_, rest)| fits(rest))
}
```

Qué significa «encajar» depende de quién llama, así que es un closure. Un cifrado encaja cuando el resto es una cualidad y quizá un bajo; un nombre de nota completo encaja cuando no queda nada:

```
pub fn parse(text: &str) -> Option<NoteName> {
    NoteName::read_either(text, str::is_empty).map(|(name, _)| name)
}
```

`str::is_empty` es una función, y una función puede pasarse allí donde se espera un closure con los mismos argumentos.

El lifetime `'a` dice que el resto que recibe `fits`, y que se devuelve, es un trozo de `text`, no una copia. Así funciona «Faug» con el solfeo elegido: «Fa» deja «ug», que `fits` rechaza, así que se prueba la lectura con letras, F con «aug», y se queda esa.

Ejemplos que siguen el ajuste

Las ayudas de la hoja de acordes y de los ajustes dan acordes de ejemplo. Se escriben con las notas entre llaves, `{A}m7 {D}7`, y `with_notes` nombra cada una como dice el ajuste, así que con el solfeo elegido la ayuda dice «Lam7 Re7». Las llaves alrededor de

algo que no es un nombre de nota, como el conjunto `{0,1,4}` de la misma ayuda, se dejan como están.

L de La

`ChordSheet::takes_l` dice si una L escrita ahora le corresponde a la hoja: la hoja tiene el teclado, hay un compás seleccionado, todavía no se está escribiendo en él, y las notas se nombran en solfeo. El atajo L de la app lo mira primero:

```
let l_types = self.chord_sheet.takes_l() && !self.live.typing();
if !typing && !controls_active && !l_types && pressed(egui::Key::L) {
```

`!self.live.typing()` está ahí porque, con el teclado para tocar activado, las letras tocan notas, y L cambia entre patrón y canción como siempre.

Fuera de la app

El archivo de escritorio de Linux tiene líneas en español junto a las inglesas, por ejemplo `GenericName[es]=Banda de acompañamiento`, que un escritorio en español muestra en sus menús. El archivo AppStream que leen las tiendas de software da a cada párrafo de la descripción un gemelo en español, `<p xml:lang="es">`, y `appstreamcli validate` lo comprueba.

Las pruebas

`every_window_writes_in_spanish` abre seis demostraciones en español con todas las ventanas abiertas, y después los atajos, Acerca de, la exportación de video, el editor de estilos, cada página de los Ajustes y cada menú. Falla si algo le pidió a `tr` un texto que no tiene español en el catálogo, o si algún texto dibujado en pantalla es un texto en inglés que el catálogo sí tiene en español, lo que detecta el texto dibujado sin pasar por `tr`. En el núcleo, `every_name_and_description_the_app_shows_has_spanish` hace lo mismo con cada nombre de instrumento, estilo, acorde, escala y demostración que el núcleo le da a la app.

`solfege_changes_how_names_look_not_what_is_saved` construye cada canción de demostración dos veces, una con letras y otra con solfeo, y comprueba que el archivo guardado y el texto de la hoja de acordes son idénticos, y que cada acorde, escrito como lo muestra la hoja en solfeo, se lee de vuelta como el mismo acorde.

`names_are_read_in_letters_or_solfege` lee `Sib7`, `Fa#m`, `Sol` y `Reb`, y comprueba que `Do7` empieza por Re con las letras elegidas y por Do con el solfeo. `solfege_is_a_way_of_showing_names` comprueba que `Am7` se muestra como Lam7, `G/B` como Sol/Si y Si♭ mayor como Sib mayor, y que `Lam7` y `Am7` son el mismo acorde.

`in_solfege_l_types_la_chords_and_in_letters_it_switches_mode` selecciona un compás de la hoja de acordes, pulsa L con el solfeo elegido y escribe `Lam7`, y comprueba que el modo no cambió y que el compás tiene La menor séptima; con las letras, L cambia el modo y deja el compás como estaba.

`spanish_names_notes_in_solfege_until_a_naming_is_chosen` cambia el idioma de un lado a otro sin nomenclatura elegida, y la nomenclatura lo sigue; luego elige una nomenclatura y vuelve a cambiar el idioma, y la nomenclatura se queda. `spanish_names_notes_in_solfege_and_keeps_letters_once_chosen` lo hace como lo harías tú: menús en español, **Armonía**, **Letras (C, D, E…)** y un reinicio de la app, tras el cual las notas siguen en letras.

Ejercicios: la forma del programa

Estos ejercicios practican el Rust de esta parte sobre el código real. Cada uno dice lo que **ya existe**, para que lo amplíes en lugar de construirlo otra vez. Algunos son pequeños trabajos con un final claro; los que se llaman «Piénsalo» son preguntas, sin código que escribir.

Hazlos en una rama tuya. Ejecuta `cargo test` después de cada cambio, y `cargo fmt --all --check` y `cargo clippy --workspace --all-targets -- -D warnings` antes de dar uno por terminado: CI ejecuta los tres.

Crates e hilos

1. Rómpelo a propósito (cualquier crate)

Aprender a leer los errores del compilador es la mitad de aprender Rust. Prueba cada una de estas cosas, lee el error y luego deshazla:

- En `AudioEngine::new` (`engine.rs`), llama dos veces seguidas a `build_stream::<f32>(&device, config.clone(), audio_thread, Arc::clone(&problems))`, de modo que la segunda llamada falle porque `audio_thread` se movió a la primera.
- Quita `move` del closure del callback de audio en `build_stream`.
- Añade una variante `SetName(String)` a `Command`. ¿Por qué `Command` no puede ser `Copy`? ¿Qué variantes ya lo hacen imposible, y qué dice el compilador cuando pides `Clone`?
- Haz `pub` el campo de `Note` (en `lineup-core/src/pitch.rs`) y escribe `Note(200)` en `app.rs`. ¿Por qué compila eso, y por qué es una mala idea?

Practica: propiedad, closures `move`, tipos `Copy` y no `Copy`, privacidad.

2. Piénsalo: el hilo de audio

- ¿Por qué sería peor `Command::InsertChannel { instrument: Box<Instrument>, ... }` que enviar el instrumento por valor? (`Instrument` ya mete en una caja sus arrays de voces; ¿qué ganaría una segunda caja?)
- ¿Qué pasaría si la ventana olvidara llamar a `collect_garbage`?

- `Garbage` es un único enum con una variante por cada clase de cosa que se tira. ¿Qué costaría una cola de basura por cada clase, y qué ganaría?
- `AudioEngine::playhead` usa `Ordering::Relaxed`. ¿Podría la ventana ver alguna vez un número de paso que nunca sonó?

El proyecto

3. Eliminar todas las pistas vacías (lineup-core y lineup-app, fácil)

Eliminar una pista ya existe: lee `Project::remove_track` y el elemento «Eliminar la pista» del menú contextual de la pista. Quita la pista y sus clips, y sube una posición los clips de las pistas de debajo, igual que hace `remove_pattern` con los patrones.

Lo que falta es una forma de quitar *todas* las pistas vacías a la vez. `remove_unused_tracks` solo quita las pistas vacías del final de la lista de reproducción (llama a `trim_tracks`). Añade `remove_empty_tracks`, que quite una pista esté donde esté si no tiene clips, y un elemento para ella en el menú de la pista. ¿Qué índices tienes que visitar, y en qué orden, para que los clips sigan apuntando a las pistas correctas? ¿Qué debería pasar si eso dejara sin pistas? Escribe primero un test con la canción de ejemplo, y luego el método.

Practica: `retain`, mantener reglas que abarcan dos listas, probar el orden de los cambios de índice.

4. Piénsalo: posiciones o identificadores

Los clips se refieren a los patrones por su *posición* en la lista, y por eso `remove_pattern` tiene que renumerarlos. Imagina en cambio que cada patrón tiene un identificador permanente (`struct PatternId(u32)`) que nunca cambia.

- ¿Qué métodos de `Project` se vuelven más sencillos, y cuáles más difíciles?
- ¿Qué cambiaría en el archivo, y en los `ChannelNotes` del hilo de audio?
- ¿Sería más fácil deshacer con identificadores o con posiciones?

Avisar al hilo de audio de lo que cambió

5. Ediciones al azar para el diff (lineup-core, media)

Los tests del diff comprueban casos elegidos a mano. Escribe uno que compruebe mil al azar: empieza con la demo, haz una edición al azar (activar un paso, añadir o quitar un patrón o un canal, cambiar un volumen, mover un clip…), llama a `check_both_ways(&before, &after)` y repite desde el proyecto nuevo.

Para tener azar sin ningún crate, un pequeño generador «xorshift» basta para los tests:

```
struct Random(u64);

impl Random {
    fn next(&mut self, below: usize) -> usize {
        self.0 ^= self.0 << 13;
        self.0 ^= self.0 >> 7;
        self.0 ^= self.0 << 17;
        (self.0 % below as u64) as usize
    }
}
```

Arráncalo con un número fijo (no 0, que se queda en 0 para siempre), para que un fallo vuelva a ocurrir cada vez que ejecutes el test. Si quieres ir más lejos, el crate `proptest` hace esto como es debido, y cuando encuentra un caso que falla lo *reduce* a la lista de ediciones más corta que sigue fallando.

Prueba también varias ediciones al azar entre dos comprobaciones, como las que puede contener un paso de deshacer. ¿Produce `diff` alguna vez una lista larga donde bastaría una corta? ¿Es eso un problema?

Practica: tests basados en propiedades, azar reproducible.

6. Canales emparejados por nombre (lineup-core, media)

`diff` empareja los canales de dos proyectos con `|old, new| old == new`, y los patrones con `same_pattern`. Escribe `same_channel(old: &Channel, new: &Channel) -> bool`, que diga que dos canales son el mismo cuando tienen el mismo nombre, y pásala a `Alignment::new` en su lugar. Ejecuta los tests del diff. Luego busca un caso en que dé la respuesta equivocada: ¿qué pasa con dos canales llamados «Piano»?

Practica: funciones como argumentos, qué significa «el mismo».

7. Piénsalo: qué enviar

- `SetNotes` envía la lista entera de notas de un canal con cada cambio. ¿A partir de qué tamaño sería eso un problema, y qué enviarías en su lugar?
- La ventana envía la duración de la canción junto con sus piezas, en lugar de dejar que la calcule el hilo de audio. ¿Qué haría falta para que dos copias de ese cálculo coincidieran siempre, y cómo notarías que se separan? Las piezas son los `Play` que hace `Project::playback_arrangement`. ¿Por qué es más sencillo desplegar las repeticiones del lado de la ventana que enseñarle al hilo de audio qué son las secciones?
- Deshacer mientras suena la canción. Cuando deshacer quita un canal, sus notas que suenan se cortan en el acto, porque el canal va a la basura. En el caso de tres canales Lineup Osc idénticos, el hilo de audio quita el *último* canal y da sus notas a otro. ¿La nota de qué canal sigue sonando, y podrías llegar a oír la diferencia?

Deshacer y rehacer

8. Una ventana con el historial (lineup-app, fácil)

Añade Edición > Historial…, una ventanita que liste cada paso que se puede deshacer o rehacer, el más nuevo arriba, con la posición actual marcada. Hacer clic en un paso salta hasta él.

Necesitas dos métodos en `History`: uno que dé las etiquetas para deshacer y otro las de rehacer, cada uno como iterador (`self.undo.iter().map(|step| step.label)`). Volver cinco pasos atrás son cinco llamadas a `history.undo`. ¿Debería ejecutarse `sync_audio` después de cada una, o una vez al final, comparando con el proyecto de antes de la primera? ¿Cuál es más barato, y acaba el hilo de audio igual en los dos casos? (El ejercicio 5 responde a la última pregunta.)

Practica: iteradores sobre campos privados, `rev()`, reutilizar `diff`.

9. Copias más baratas (lineup-core, difícil)

Cada paso de deshacer copia el proyecto entero, con todos sus patrones, aunque un clic en un paso cambie un solo patrón. (Un arrastre copia una vez, cuando abre su paso, no en cada fotograma, pero la copia sigue siendo el proyecto entero.) Abarata

las copias compartiendo lo que no cambió: guarda los patrones como `Vec<Arc<Pattern>>`.

Clonar un `Vec<Arc<Pattern>>` copia solo los punteros y suma uno a cada contador; los patrones mismos se comparten entre el proyecto actual y cada copia del historial. Para cambiar uno, usa **`Arc::make_mut`**, que da un `&mut Pattern` y antes hace una copia privada *solo si* el patrón está compartido («clonar al escribir»). Así, `Project::pattern_mut` pasa a ser `self.patterns.get_mut(index).map(Arc::make_mut)`.

Después `diff` puede saltarse rápido los patrones sin cambios: si `Arc::ptr_eq` dice que dos patrones son la misma memoria, seguro que son iguales. Haz un proyecto de prueba con 999 patrones llenos de notas, y mide el tiempo de una edición antes y después con `std::time::Instant`. ¿Sigue funcionando sin cambios el `PartialEq` del proyecto? ¿Tiene que cambiar el formato del archivo?

Practica: `Arc` para compartir, clonar al escribir, medir antes de optimizar.

10. Piénsalo: etiquetas y lo que abarca deshacer

- Las etiquetas de los pasos son `&'static str`, así que «rename pattern» no puede decir qué patrón. Una etiqueta `String` permitiría «rename pattern Beat», pero la mayoría de las etiquetas son texto fijo, y un `String` reservaría memoria para cada una. Busca `std::borrow::Cow<'static, str>`: ¿cómo permite que una etiqueta sea de los dos tipos, y que reserve memoria solo cuando haga falta?
- El volumen master forma parte del mezclador del proyecto, así que deshacer lo devuelve a su valor anterior, y se guarda con la canción. El patrón seleccionado, el zoom y las ventanas abiertas no. ¿Esperarías que deshacer cambiara alguno de ellos? ¿Qué tendría que pasar a `Project` para eso, y qué significaría para el archivo guardado?

Guardar

11. Un campo nuevo del proyecto y su test de formato (lineup-core, media)

Objetivo. Añadir un campo inofensivo a un proyecto, guardarlo sin romper los archivos antiguos, y demostrar que una canción que lo usa se escribe con el formato

siguiente. `ProjectFile`, sus valores por defecto de serde y los tests de archivo ya existen; este es el ejercicio completo más pequeño para hacer crecer el formato.

Archivos. `crates/lineup-core/src/project.rs`, `crates/lineup-core/src/file.rs`.

Pasos.

1. Añade `pub notes: String` a `Project`, inicializado a `String::new()` en `Project::empty`.
2. Añade `notes: String` a `ProjectFile`, con `#[serde(default, skip_serializing_if = "String::is_empty")]`, y cópialo en los dos sentidos.
3. Sube `FORMAT_VERSION` en uno, a 30 (si otro ejercicio ya tomó el 30, usa el siguiente número libre). En `format_for`, devuelve el número nuevo en primer lugar cuando `!project.notes.is_empty()`; deja todo lo demás como está.
4. Escribe tres tests en `file.rs`: un proyecto vacío se queda en su formato anterior y no tiene línea `notes:`; un proyecto con notas es del formato nuevo y sobrevive a guardar y abrir; y un archivo escrito antes de que existiera el campo se lee con las notas vacías.

Ejecuta. `cargo test -p lineup-core file`.

Terminado cuando. Los textos de proyecto antiguos siguen abriéndose, el campo nuevo sobrevive a guardar y abrir, y `format_for` da el número nuevo solo para una canción que lo necesita. `files_from_the_future_are_refused` construye su archivo con `FORMAT_VERSION + 1`, así que sigue pasando.

12. Una copia de recuperación dañada (lineup-app, fácil)

Objetivo. Hacer fiable la recuperación tras un cierre inesperado cuando la copia en disco no lo es. `app/recovery.rs` ya escribe una copia del proyecto, un archivo `.info` y un archivo de bloqueo por cada ventana en la carpeta `recovery` de la carpeta de datos de la app, ofrece la copia en el siguiente arranque y elimina una copia que no se puede cargar. El test `a_recovery_copy_comes_back_with_its_file` cubre el camino feliz. Añade los otros.

Archivos. `crates/lineup-app/src/app/recovery.rs`.

Pasos.

1. Míralo a mano: abre una demo, escribe un acorde en el primer compás, espera un momento y fuerza el cierre de la app. No uses **Archivo > Salir**. Arráncala otra vez y elige **Recuperarlo**. Cierra la canción recuperada sin guardarla.
2. Lee `Files`, `load` y `write`, y los tests al final del archivo.

3. Añade un test que tome `Files::of_run(&folder, "test")`, cree la carpeta, escriba basura en su archivo `project`, ponga al lado un archivo `info` válido, llame a `load(files.clone(), None)` y compruebe que devuelve `None` **y** que los dos archivos han desaparecido.
4. Añade un segundo test que escriba un proyecto válido en `files.project` con `file::save`, pero sin el archivo info. Comprueba que `load` sigue devolviendo el proyecto, sin ruta y sin nombre, para que el diálogo lo llame «Sin título».
5. Borra la carpeta temporal al final de cada test, como hacen los que ya existen.

Ejecuta. `cargo test -p lineup-app recovery`.

Terminado cuando. Los dos tests pasan, y el primero falla si se borra la llamada a `files.remove()` en la rama de error de `load`.

13. Olvidar un proyecto reciente (lineup-app, media)

Archivo > Abrir recientes recuerda los últimos diez proyectos en los datos persistentes de egui, y sigue siendo correcto cuando un archivo se movió o se borró (la apertura falla y la ruta se quita). Lee `app/recent.rs`.

Amplíalo: un elemento «Borrar recientes» que olvide la lista entera, y una pequeña ✕ junto a cada entrada que olvide una. La ✕ no debe abrir además el archivo, así que fíjate en el orden en que compruebas los dos clics. ¿Por qué se guarda la lista como `Vec<String>` y no como `Vec<PathBuf>`, y qué le pasa a una ruta que no es UTF-8?

Practica: menús de egui, persistencia, rutas que no son UTF-8.

14. Leer números despacio (lineup-core y RON, media)

El número de versión explica por qué la app ya no lee el formato de un archivo con un struct de un solo campo: saltarse el resto del archivo hacía que RON mirara hacia delante, número tras número, hasta el final del texto.

1. Supón que una canción guardada tiene 2 MB de texto con 120 000 números. Si saltarse cada número lee, de media, medio archivo, ¿cuántos bytes se leen en total? A unos 2 GB por segundo, ¿cuánto tarda? ¿Y una canción el doble de larga?
2. Busca la búsqueda hacia delante en el código de RON: en la versión que nombra `Cargo.lock`, `src/parse.rs` busca `".."` en `self.src()[skip..]`. Reescribe esa línea para que busque solo en los primeros `raw_float_len` caracteres. (Pista: corta `self.src()[skip..][..raw_float_len]`, y piensa qué pasaría si el corte terminara en medio de un carácter.) Después compáralo con cómo lo hace RON

0.12.3, y lee su comentario sobre por qué un `..` no puede quedar a caballo del final.
3. `format_number` devuelve `None` para `(tempo: 120.0, format: 25)`. ¿Por qué no importa que solo lea el principio del archivo? (Mira el orden de los campos en `ProjectFile`, y `to_text`.)

Practica: pensar en ritmos de crecimiento, leer el código de una biblioteca, porciones de cadenas.

15. Piénsalo: archivos

- `has_unsaved_changes` compara el proyecto entero con la copia guardada, pero solo después de que una edición o un deshacer active `check_dirty`, no en cada fotograma. ¿Cuántas notas, más o menos, necesitaría un proyecto para que una comparación ocupe una parte notable de un fotograma (unos 16 milisegundos)? ¿Por qué `project_edited` activa `check_dirty` solo cuando `dirty` ya es falso? ¿Qué añadiría a este diseño un contador que cada edición incrementara?
- El archivo de proyecto tiene sus propios tipos (`ProjectFile`, `PatternFile`) en lugar de derivar `Serialize` en el propio `Project`. ¿Qué fallaría el día que alguien renombrara un campo de `Project`, si el archivo lo usara directamente?
- Un Ferrous Lineup más antiguo se niega a abrir un archivo con un número de formato más nuevo. ¿Podría cargar lo que entiende e ignorar el resto? ¿Cuándo ayudaría eso, y cuándo perdería sin avisar el trabajo de alguien?
- El test de archivos dañados escribe `NaN` en lugar de números. ¿Qué pasa cuando un archivo de proyecto dice `tempo: NaN`? ¿Y `master: NaN`? Busca el código que decide cada caso, y di por qué son distintos.

Dibujar las ventanas

16. Un doble clic restablece el volumen (lineup-app, fácil)

Haz que un doble clic en el deslizador VOL de la barra de herramientas devuelva el volumen master a `lineup_core::Mixer::default().master` (0.8). Un botón de restablecer no cabría: la barra de herramientas tiene que caber en una fila en una ventana de 1280 puntos de ancho, también en español, y `the_toolbar_fits_on_one_row_at_1280_points` lo comprueba.

Llama a `LineupApp::set_master_volume` (en `app/mixer.rs`) en lugar de asignar el campo, para que el cambio sea un paso que se pueda deshacer y el diff envíe al hilo de audio `Command::SetMasterVolume`. Ejecuta la app y comprueba que el deslizador se mueve.

Practica: entrada en modo inmediato, `Response`, encaminar un cambio a través de `edit`.

17. Recordar el nivel de las pistas (lineup-app y lineup-core, media)

El nivel de las pistas de las teclas de armonía vuelve a Breve cada vez que arranca la app. Otras elecciones se recuerdan: Recordar entre ejecuciones muestra cómo, y `settings.rs` guarda la elección de los grados en menor con `get_persisted` e `insert_persisted` bajo un `Id` propio. Haz lo mismo con `HintLevel`: dale `Serialize` y `Deserialize` de serde en `lineup-core` (que ya depende de serde), y léelo y escríbelo donde `HarmonyKeys` guarda `hints`. ¿Qué otras elecciones recordarías de la misma manera?

Practica: derives de serde entre crates, los datos persistentes de egui.

18. Pistas en todas partes (lineup-app, pequeño)

`theme::hint` da a un widget una línea en la barra de pistas y el mismo texto como tooltip. La barra de herramientas, las teclas de armonía y algunos botones de la lista de reproducción lo usan; el rack de canales y el piano roll todavía usan `on_hover_text` a secas. Cámbialos a `theme::hint`, y haz que la barra de pistas muestre el nombre del patrón cuando no hay nada bajo el ratón. ¿Por qué `theme::hint` toma el `Response` por valor y lo devuelve, en lugar de tomar `&Response`?

Practica: propiedad a través de una función, el `Response` de egui, leer el módulo de otro antes de cambiarlo.

19. F10 para el metrónomo, y su línea en F1 (lineup-app, fácil)

Objetivo. El metrónomo es un interruptor en la barra de herramientas y en **Edición > Ajustes… > Reproducción**. Dale una tecla y añade la tecla a la lista de F1. F10 está libre; no uses **M** a secas, que toca una nota cuando el teclado del equipo hace de piano.

Archivos. `crates/lineup-app/src/app/practice.rs`, `crates/lineup-app/src/app.rs`, `crates/lineup-app/src/app/shortcuts.rs`, `crates/lineup-app/src/app/ux_tests.rs`.

Pasos.

1. En `practice.rs`, añade una constante `KeyboardShortcut` para F10, como `view.rs` guarda F5 a F8.
2. En `LineupApp::keyboard_shortcuts`, léela en el mismo sitio protegido que Inicio y Fin: no mientras un campo de texto tiene el teclado, y no mientras se usa un control. Llama a `set_metronome` con lo contrario de `practice.view().metronome`.
3. Añade una línea a la sección **En cualquier lugar** de `shortcuts.rs`: «Metronome on or off». Usa la constante en lugar de escribir la tecla dos veces.
4. Añade un test sin pantalla en `ux_tests.rs`: pulsa F10 y comprueba `practice.view().metronome`; púlsala otra vez y comprueba que está apagado.

Ejecuta.

```
cargo test -p lineup-app shortcuts
cargo test -p lineup-app metronome
```

Terminado cuando. F10 activa y desactiva el botón de la barra de herramientas, F1 lo lista en «En cualquier lugar», y `no_two_global_shortcuts_share_a_key` sigue pasando. Una línea nueva de la lista de F1 es texto en inglés: el test `every_window_writes_in_spanish` te avisa de que también necesita una en español (mira Dos idiomas).

20. Un dibujo y una versión (lineup-app, fácil)

- Dibuja otro icono para **Seguir el cursor de reproducción** en `icons.rs`: un cursor con dos flechas. ¿Qué test te avisa cuando una línea se sale de su cuadrado?
- Añade un botón **Copiar versión** a la ventana Acerca de Ferrous Lineup que ponga «Ferrous Lineup» y la versión real en el portapapeles con `ctx.copy_text`. ¿Puede el texto ser una `const`, construida con `concat!` y `env!`?

Practica: dibujar con egui, cadenas en tiempo de compilación.

21. Piénsalo: listas y ventanas

- En `shortcuts.rs`, la sección de las teclas de armonía usa `Written("1 to 7")`. ¿Por qué no siete `KeyboardShortcut` en un `Any`? ¿Qué mostraría entonces la lista?
- La partitura guía se abre con `Opening::FirstTime`. ¿Qué cambiaría para ti si usara `EveryTime`? Pruébalo, y di cuál prefieres.

Los editores

22. Vaciar una fila (lineup-app, fácil)

Hacer clic derecho en un paso ya lo vacía (`RackAction::ClearStep`). Añade «Vaciar los pasos» al menú del nombre de un canal en `channel_menu`, con una nueva `RackAction::ClearRow { channel }`. En `LineupApp::apply`, llama a `edit_pattern` para apagar todos los pasos de ese canal, sin enviar a mano un comando al hilo de audio: el diff de la edición envía las notas nuevas. ¿Por qué es mejor una edición de toda la fila que un bucle de ediciones de un paso?

Practica: el patrón de acciones, mantener proyecto y audio a la par.

23. Pintar pasos arrastrando (lineup-app, media)

En el rack haces clic en un paso cada vez. Dale a `widgets::step_button` un `Sense::click_and_drag()` y haz que `Rows::steps` pinte cada paso por el que pasa el puntero. `Rows` se reconstruye en cada fotograma y solo devuelve acciones, así que guarda el primer paso, y si se estaba encendiendo o apagando, en `LineupApp` (o en la memoria de egui) y pásalo.

Añade `RackAction::SetStep { channel, step, on }` en lugar de reutilizar `ToggleStep`, para que un paso que el puntero cruza dos veces no vuelva atrás. Comprueba `response.hovered()` junto con `ui.input(|i| i.pointer.primary_down())`, y deja de pintar cuando el puntero sale de la fila. ¿Debería ser todo el arrastre un solo paso de deshacer? Mira ¿Cuándo termina un paso?

Practica: estado en modo inmediato, entrada de egui, el patrón de acciones.

24. Mostrar el grado de la escala en las teclas (lineup-app, fácil)

`paint_keys` en `piano_roll.rs` dibuja un punto en cada tecla de la escala. Dibuja en su lugar el número del grado (1 para la tónica, 2, 3…), con `Scale::degree_of`. Una serie dodecafónica ya numera sus notas; deja ese camino como está. Los músicos usan estos números todo el tiempo («el acorde de quinto grado», «una séptima menor»).

Practica: `Option` en el código de dibujo, `painter.text`.

25. Las luces del rack (lineup-app, fácil)

1. Haz que la luz encendida sea verde en vez de naranja. ¿Dónde se define el color, y qué más lo usa?
2. Da a las luces del primer paso de cada compás un color propio, en vez de solo uno más brillante.
3. Mientras la canción toca un patrón distinto del que muestra el rack, las luces se quedan apagadas. Añade una nota tenue en el rack, como «No suena ahora», para ese caso. ¿Qué valores de `channel_rack::show` te dicen que suena la canción?

Practica: leer código de dibujo, colores de egui.

26. Nombres de acorde más cortos (lineup-app, media)

La franja de acordes del piano roll omite un nombre que no cabe en su bloque. Haz que pruebe antes un nombre más corto: para un acorde con bajo, el acorde sin su bajo («E7» para «E7/B»), y luego la fundamental sola («E»). Mide cada uno con `layout_no_wrap` y dibuja el primero que quepa. ¿Sigue siendo un acorde sin su bajo el nombre correcto de lo que suena?

Practica: medir el texto antes de dibujarlo, iteradores con `find`.

27. Copiar un clip arrastrando con Alt (lineup-app, fácil)

Ctrl+D duplica los clips seleccionados, pero arrastrar una copia es más rápido cuando la quieres en otro sitio. Mayús ya está ocupada (Mayús+clic añade un clip a la selección), así que usa Alt.

En `PlaylistView::handle_input`, lee las teclas modificadoras con `ui.input(|input| input.modifiers.alt)`. El plan obvio es añadir una copia al pulsar y arrastrar la copia. Pero `add_clip` rechaza un clip del mismo patrón que ya empieza en el mismo sitio, así que no se puede añadir una copia encima del original. En su lugar, recuerda en `ClipDrag` que este arrastre copia, y añade la copia solo cuando el puntero llega a otra celda. ¿Qué índice recibe el clip nuevo en `Project::playlist().clips()`, y cómo se entera el arrastre?

Practica: leer la entrada de egui, reutilizar acciones que ya tienes.

28. Un selector de color para los patrones (lineup-app, media)

Los colores de los patrones ya se guardan. `Pattern` tiene un `color` público (un índice en `PALETTE`, en `playlist.rs`), `PatternFile` lo escribe cuando no es el lugar del patrón en la lista, y los archivos que no lo tienen cargan con el lugar del patrón como color. Lee `PatternFile` y `pattern_color`.

Lo que falta es el selector. Añade uno al menú ⋯ del patrón en la barra de herramientas, con un cuadradito por cada color de la paleta, que cambie el color del patrón con `edit_pattern`. Después añade «Restablecer colores», que devuelva a cada patrón su lugar en la lista. ¿Necesita un cambio de color un `FORMAT_VERSION` nuevo? ¿Qué hace un Ferrous Lineup más antiguo con un archivo que lo tiene?

Practica: hacer evolucionar un formato de archivo, `#[serde(default)]`, encaminar un cambio a través de `edit`.

29. Ir a un compás (lineup-app, media)

Inicio, Fin, RePág y AvPág ya mueven el cursor de reproducción: lee las teclas de transporte en `LineupApp::keyboard_shortcuts`, `seek` y `skip_bars`. Añade una pequeña casilla «compás» junto a la posición, donde escribir un número de compás y pulsar Intro lleve allí. El primer paso de un compás depende de cada cambio de compás anterior: ¿qué función convierte ya un paso en un compás, y puede ir en el otro sentido? Mientras escribes en la casilla, Inicio y Fin tienen que mover el cursor del texto, no el de reproducción: ¿qué impide que hagan las dos cosas?

Practica: el foco del teclado, convertir compases en pasos.

30. Piénsalo: lo que recuerda un arrastre

El arrastre de una sola nota en el piano roll recuerda la nota por su índice en la lista (`Drag::Move { index, .. }`), mientras que una selección recuerda las notas mismas. ¿Qué podría salir mal con el índice si la lista se mantuviera ordenada por inicio, y una nota se moviera mientras se arrastra?

Dos idiomas

31. Letras y solfeo (lineup-core y la app)

1. Con el solfeo elegido, escribe `Do` en un compás vacío de la hoja de acordes. Luego elige letras y escribe `Do` en el compás siguiente. ¿Qué contiene cada compás, y por qué son distintos?
2. Escribe en solfeo los siete acordes de Si♭ mayor, y compruébalos en las teclas de armonía. ¿Qué acorde es disminuido?
3. Un proyecto se guarda con letras más de una vez: `file::to_text` hace su trabajo dentro de `NoteNaming::Letters.with`, y lo mismo hace `From<HarmonyName> for String`. Quita todos los `NoteNaming::Letters.with` de los dos, dejando lo que hay dentro, y ejecuta `cargo test -p lineup-core solfege`. ¿Qué comprobación falla, y qué encontraría en tu archivo un amigo que usa letras? Luego vuelve a poner solo el de `From<HarmonyName>`. Los tests vuelven a pasar: ¿por qué, y por qué conservar también los otros?
4. `with_notes` deja `{0,1,4}` como está. Busca la línea de `with_notes` que lo decide, y di qué haría con `{Do}`.
5. `note_naming` podría guardar la nomenclatura del idioma la primera vez que no encuentra ninguna, con `insert_persisted`, para que la siguiente lectura la encuentre. Quedaría más ordenado. ¿Qué fallaría entonces para alguien que arranca la app en inglés y la cambia a español una semana después?

Practica: ajustes locales a cada hilo, porciones de cadenas, leer tests.

El secuenciador: temporización a la muestra

Archivos: `crates/lineup-core/src/sequencer.rs` y `crates/lineup-audio/src/engine.rs`

El hilo de audio toca un patrón o la canción contando frames de audio. Quien cuenta es un tipo pequeño de `lineup-core`, el **secuenciador**, que no sabe nada de sonido: solo dice en qué frame empieza cada paso. Todo lo que suena en un paso, las notas de cada canal en cada clip, es cosa del hilo de audio, en `engine.rs`. Este capítulo sigue un paso desde la cuenta de frames hasta las notas que empieza. Cómo funciona el hilo de audio en sí está en Dos hilos; cómo doblan los pasos los cambios de tempo, los compases y el swing está en Tempo, compás, swing y sensación.

La aritmética

Un paso es una semicorchea, y un pulso una negra, así que un pulso tiene cuatro pasos (`STEPS_PER_BEAT`). A un tempo de *T* pulsos por minuto y una frecuencia de muestreo de *R* frames por segundo, un pulso dura `R × 60 / T` frames y un paso un cuarto de eso:

```
frames por paso = R × 60 / T / 4

120 BPM, 48 kHz:  48000 × 60 / 120 / 4 = 6000 frames       (exacto)
130 BPM, 44.1 kHz: 44100 × 60 / 130 / 4 = 5088.46... frames (no entero)
```

```rust
pub fn frames_per_step(tempo: Tempo, sample_rate: u32) -> f64 {
    tempo.samples_per_beat(sample_rate) / STEPS_PER_BEAT as f64
}
```

Si 5088.46 se redondeara a 5088, cada paso se quedaría 0.46 frames corto. Son unos 10 microsegundos, nada que fueras a oír una sola vez, pero se acumula: al cabo de unos minutos la batería iría audiblemente por delante de cualquier otra cosa que corra a 130 BPM. Por eso el secuenciador conserva la fracción, en un `f64`.

El tick

El hilo de audio llama a `tick` una vez por cada frame que produce:

```
pub fn tick(&mut self) -> Option<usize> {
    let fired = if self.frames_until_next <= 0.0 {
        let step = self.next_step;
        if self.coming_round {
            self.laps += 1;
        }
        self.next_step = (self.next_step + 1) % self.length;
        self.coming_round = self.next_step == 0;
        self.frames_until_next += self.frames_per_step;
        Some(step)
    } else {
        None
    };
    self.frames_until_next -= 1.0;
    fired
}
```

`frames_until_next` cuenta hacia atrás de uno en uno cada frame. Cuando llega a cero, se dispara el siguiente paso y se *suma* al contador la longitud de un paso entero, no se asigna. La fracción que sobrara se queda en el contador, de modo que los límites de los pasos caen en 0, 5088.46, 10176.92… redondeados al frame más cercano cada vez, pero sin derivar nunca. `%` es el operador de resto: devuelve el número de paso a 0 al final del bucle.

Como un paso empieza en el frame exacto donde empieza, y no al principio del bloque de audio que lo contiene, la temporización no depende del tamaño del búfer de la tarjeta de sonido. Eso es lo que significa **exacto a la muestra**.

Qué más puede decir el secuenciador

Otras partes del programa le preguntan al secuenciador algo más que «qué paso suena ahora»:

- `position` da el paso con la fracción que ha pasado: 4.5 es la mitad del paso 4. La grabación la usa para poner una tecla tocada entre dos pasos en el más cercano (MIDI).
- `laps` cuenta cuántas veces volvió el bucle al principio por seguir tocando después de su último paso. Un salto al principio no cuenta. El entrenador de velocidad acelera en cada vuelta (Herramientas de práctica).
- `due` dice qué paso empieza en el *próximo* tick, sin avanzar. El hilo de audio lo pregunta antes de `tick`, así que una canción cuyo tempo cambia puede poner

el tempo nuevo antes de que empiece ese paso, y el paso entero dura lo que debe.
- **`start_at`** salta, de modo que el paso dado se dispara en el próximo tick; un paso más allá del final da la vuelta, como lo haría la reproducción.

`set_tempo` cambia la longitud de los pasos que vienen; el paso que ya suena conserva los frames que le quedan. `set_length` vuelve al paso 0 si el siguiente paso queda más allá del nuevo final.

Por qué vive en `lineup-core`

El secuenciador no tiene nada que ver con tarjetas de sonido. Solo cuenta. Tenerlo en `lineup-core` significa que se prueba en milisegundos, sin audio, y la cuenta previa usa otro propio.

Probar con iteradores

```
fn run(sequencer: &mut Sequencer, frames: usize) -> Vec<(usize, usize)> {
    (0..frames)
        .filter_map(|frame| sequencer.tick().map(|step| (frame, step)))
        .collect()
}
```

Esta función auxiliar hace avanzar el secuenciador `frames` veces y lista cada paso que se disparó junto con el frame en que se disparó:

- `0..frames` es un **rango**, que es un iterador sobre números.
- `filter_map` llama al closure para cada frame y se queda solo con los resultados `Some`, desenvueltos.
- `Option::map` convierte `Some(step)` en `Some((frame, step))`, una tupla.
- `collect()` reúne los resultados en un `Vec`. Rust deduce qué colección construir a partir del tipo de retorno.

Los tests comprueban luego números de frame exactos:

```
assert_eq!(
    fired,
    vec![(0, 0), (6_000, 1), (12_000, 2), (18_000, 3), (24_000, 0)]
);
```

y que un minuto a 130 BPM da exactamente 520 pasos (130 pulsos × 4), el último a menos de un frame de donde lo ponen 519 pasos exactos. Los guiones bajos en literales numéricos como `6_000` los ignora el compilador y solo sirven para que los números grandes se lean mejor.

Modo patrón y modo canción

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default)]
pub enum PlayMode {
    /// Loop the selected pattern, to work on it.
    #[default]
    Pattern,
    /// Play the playlist from the start, and loop at the end of the song.
    Song,
}
```

El secuenciador es el mismo en los dos modos. Cuenta pasos de 0 a cierta longitud y repite. Solo cambia la longitud: la del patrón seleccionado en modo patrón, la de la canción en modo canción.

```
fn update_loop_length(&mut self) {
    let length = match self.mode {
        PlayMode::Pattern => self.pattern_length(self.pattern),
        PlayMode::Song => self.song_length,
    };
    let length = if length == 0 { STEPS_PER_BAR } else { length };
    self.sequencer.set_length(length);
}
```

Una canción vacía tendría longitud 0, y un bucle de cero pasos no tiene sentido, así que se convierte en un compás en silencio.

La canción llega como `Command::SetSong { plays, length }`: los trozos de patrones que hay que tocar, con cada repetición ya desplegada, y la longitud. El hilo de audio podría calcular la longitud, pero `Project` ya lo hace, y dos copias de un mismo cálculo pueden separarse. Qué es un `Play`, y cómo los construye el proyecto, está en Secciones, repeticiones y bucles.

Notas para cada patrón: `ChannelNotes`

Cada canal necesita una lista de notas por patrón. Un `Vec` de `Vec`s lo resuelve, pero hay una trampa. Cuando se añade un patrón, el hilo de audio tiene que añadir una lista vacía a cada canal. `Vec::new()` en sí no cuesta nada: un `Vec` vacío todavía no tiene memoria propia. Pero añadirlo al `Vec` *exterior* reserva memoria si el `Vec` exterior está lleno. Para ir sobre seguro, el `Vec` exterior tiene que tener sitio para todos los patrones que pueda llegar a haber, reservado antes de llegar al hilo de audio.

Esa es una regla sobre cómo se construyó el valor, y Rust puede hacer cumplir reglas así con un **newtype**, un struct con un único campo privado:

```
pub struct ChannelNotes(Vec<Vec<NoteEvent>>);

impl ChannelNotes {
    /// One list per pattern, in pattern order.
    pub fn new(lists: impl IntoIterator<Item = Vec<NoteEvent>>) -> Self {
        let mut patterns = Vec::with_capacity(MAX_PATTERNS);
        patterns.extend(lists.into_iter().take(MAX_PATTERNS));
        ChannelNotes(patterns)
    }

    /// Adds an empty list for a new pattern at position `pattern`,
without
    /// allocating. The lists from there on move up one place.
    fn insert_empty(&mut self, pattern: usize) {
        if self.0.len() < self.0.capacity() {
            self.0.insert(pattern.min(self.0.len()), Vec::new());
        }
    }
}
```

Como el campo es privado, el código de fuera de `engine.rs` no puede escribir `ChannelNotes(some_vec)`. La única forma de crear uno es `ChannelNotes::new`, que siempre reserva sitio para `MAX_PATTERNS` listas. Así que cada `ChannelNotes` que recibe el hilo de audio tiene el sitio, e `insert_empty` nunca puede reservar memoria. La comprobación `len < capacity` lo asegura del todo: si alguna vez estuviera lleno, preferiría saltarse la inserción antes que reservar memoria. `.0` es la forma de llegar al campo de un struct tupla (uno cuyos campos no tienen nombre). El test `channel_notes_have_room_for_every_pattern` deja fijada la promesa.

Del lado de la ventana, enviar un canal construye uno con un iterador, una lista por patrón:

```
// The notes as they are played, with the song's swing.
let notes = ChannelNotes::new(
    (0..self.project.patterns().len())
        .map(|pattern| self.project.played_notes(pattern,
index).into_owned()),
);
```

`new` recibe `impl IntoIterator<Item = Vec<NoteEvent>>`, así que acepta este iterador, un array como `[steps(&[0]), steps(&[2])]` en los tests, o un `Vec` de listas. Las notas son `played_notes`, no las escritas: con swing, la segunda corchea de cada pulso llega tarde (Tempo, compás, swing y sensación).

Los patrones van y vienen

`RemovePattern` saca la lista del patrón de cada canal. Las listas que vienen después bajan un lugar, exactamente como `Project::remove_pattern` renumera los clips, así que los dos lados siguen de acuerdo en qué significa «el patrón 2». Cada lista quitada va a la cola de basura, para liberarla en el hilo de la ventana:

```
fn remove_pattern(&mut self, pattern: usize) {
    if pattern < self.pattern_lengths.len() {
        self.pattern_lengths.remove(pattern);
        for index in 0..self.channels.len() {
            if let Some(notes) =
self.channels[index].notes.remove(pattern) {
                self.throw_away(Garbage::Notes(notes));
            }
        }
        self.update_loop_length();
        ...
    }
}
```

El bucle cuenta con un índice en lugar de `for channel in &mut self.channels`, porque el cuerpo llama a `self.throw_away`, que necesita `&mut self`. Un bucle `for` sobre `&mut self.channels` mantendría un préstamo mutable de una parte de `self` durante todo el bucle, y `self.throw_away` necesitaría otro. Con un índice, cada préstamo de `self.channels[index]` termina en la misma línea.

`Clear` lo vacía todo antes de enviar un proyecto nuevo, al abrir un archivo:

```
while let Some(mut channel) = self.channels.pop() {
    silence(&mut channel.instrument, &mut self.midi_out, &mut
self.midi_backlog);
    self.throw_away(Garbage::Channel(channel));
}
// `clear` empties the list but keeps its memory: nothing is freed.
self.pattern_lengths.clear();
let old = std::mem::take(&mut self.plays);
self.throw_away(Garbage::Song(old));
```

- `pop` saca el último elemento de un `Vec`, o devuelve `None` cuando está vacío, así que `while let Some(..)` sigue hasta que no queda nada.
- `silence` le dice a un sintetizador externo que suelte las notas que tiene un canal MIDI Out, porque un sintetizador solo para una nota cuando se lo dicen.
- `clear` sobre un `Vec<usize>` pone su longitud a cero pero conserva su memoria, así que aquí no se libera nada, y los `push` posteriores no reservan memoria.
- `std::mem::take` saca el valor y deja en su lugar el valor por defecto del tipo, aquí un `Vec` vacío (que no posee memoria).

Abrir un proyecto tira hasta `MAX_CHANNELS` (32) canales, cada uno con su instrumento y sus notas, y la canción, sus tempos y los golpes de su metrónomo, y por eso la cola de basura tiene 1024 lugares.

Empezar las notas de un paso

Cuando se dispara un paso, el hilo de audio empieza las notas que comienzan en él. En modo patrón es un patrón. En modo canción, un paso de la canción puede estar dentro de varios clips a la vez, uno por pista, y cada clip tiene su propia idea de «qué paso»:

```
PlayMode::Song => {
    // `self.plays` is read while `self.channels` is changed: two
    // different fields, so the borrows do not overlap.
    for play in &self.plays {
        let length =
self.pattern_lengths.get(play.pattern).copied().unwrap_or(0);
        if let Some(local_step) = play.pattern_step(step, length) {
            // Notes stop at the end of the pattern, or of the
            // piece, whichever comes first.
            let room = (length - local_step).min(play.end() - step);
            start_pattern_notes(
                &mut self.channels,
                Origin {
                    pattern: play.pattern,
                    pattern_step: local_step,
                    song_step: step,
                    over_song: false,
                },
                room,
                timing,
                &mut self.pending,
                self.trace.as_deref_mut(),
            );
        }
    }
    ...
}
```

Un clip que empieza en el paso 32 de la canción toca el paso 2 de su patrón en el paso 34 de la canción. Si dos clips se solapan (un clip de acordes y uno de ritmo, en dos pistas), los dos empiezan sus notas. Un clip que repite su patrón es un solo `Play` largo, y `pattern_step` toma el paso de la canción módulo la longitud del patrón, así que el patrón vuelve a empezar una y otra vez dentro de él.

Tomar prestado solo lo necesario

`start_pattern_notes` es una función normal, no un método:

```
fn start_pattern_notes(
    channels: &mut [ChannelState],
    origin: Origin,
    room: usize,
    timing: Timing,
    pending: &mut Vec<Pending>,
    mut trace: Option<&mut Tracer>,
) { ... }
```

Si fuera un método que recibe `&mut self`, llamarlo dentro de `for play in &self.plays` no compilaría: el bucle toma prestado `self.plays` para leer, y la llamada tomaría prestado *todo* `self` para cambiarlo. Como función que recibe `&mut self.channels`, toma prestado solo ese campo, que no se solapa con `self.plays`. Dar forma a los argumentos de una función para que tome prestado solo lo que necesita es una forma habitual de trabajar con el comprobador de préstamos en lugar de contra él.

`Origin` dice de dónde vienen las notas: el patrón, el paso del patrón, el paso de la canción, y si vienen de un patrón tocado sobre la canción (más abajo). Cada voz guarda el origen de la nota que toca, y así el hilo de audio sabe después si la canción todavía la toca.

`Timing` agrupa los números que siempre viajan juntos:

```
/// How long a step is, the sample rate the instruments run at, how
/// loose the timing is, and the frame the step starts on.
#[derive(Debug, Clone, Copy)]
struct Timing {
    step_frames: f64,
    sample_rate: u32,
    loose_frames: f64,
    now: u64,
}
```

Sin él, la función recibiría nueve argumentos, varios de ellos números sueltos, y una llamada con cuatro números seguidos es fácil de escribir en el orden equivocado. (El lint `too_many_arguments` de Clippy protesta a partir de ocho, y la legibilidad sufre antes.)

Cuánto se sostiene una nota

La longitud de una nota está en pasos; el instrumento quiere frames. El hilo de audio la calcula cuando empieza la nota, y la llama **gate**:

```
let tick = timing.step_frames / TICKS_PER_STEP as f64;
let held = if note.length <= room {
    note.length as f64 * timing.step_frames - f64::from(note.nudge +
note.trim) * tick
} else {
    room as f64 * timing.step_frames - f64::from(note.nudge) * tick
};
let gate = held as u32;
```

Una nota que se pasa del final de su patrón, o del trozo de la canción que la toca, se corta ahí (`room`), para que el último acorde de un patrón no quede colgando sobre el clip siguiente. `nudge` y `trim` son fracciones de paso, en ticks: una nota de una interpretación MIDI puede empezar un poco después de su paso y acabar un poco antes del último. Con la temporización suelta, cada nota además empieza un poco tarde a propósito. Una nota que empieza tarde espera en `pending`, una lista reservada de antemano, hasta que llega su frame. Las dos cosas están en Tempo, compás, swing y sensación.

Cuando la canción cambia bajo una nota que suena

La canción puede cambiar mientras suena: se mueve o se borra un clip, se silencia una pista, la banda escribe patrones nuevos. Una nota que ya empezó sonaría hasta el final de su gate, aunque la canción ya no la toque. Así que cuando llega una canción nueva mientras se toca, el hilo de audio suelta las notas que ya no toca:

```
fn release_notes_left_out(&mut self) {
    // The fields are borrowed one by one, so the closure can read the
    // song while the channels are changed.
    let plays = &self.plays;
    let lengths = &self.pattern_lengths;
    let still_played = |origin: Origin| {
        if origin.over_song {
            return true;
        }
        let length = lengths.get(origin.pattern).copied().unwrap_or(0);
        plays.iter().any(|play| {
            play.pattern == origin.pattern
                && play.pattern_step(origin.song_step, length) ==
Some(origin.pattern_step)
        })
    };
    for channel in &mut self.channels {
        channel.instrument.release_unless(still_played);
    }
    ...
}
```

Una nota se sigue tocando si algún trozo de la canción nueva toca el mismo paso del mismo patrón en el mismo paso de la canción. `release_unless` recibe la prueba como closure (`impl Fn(Origin) -> bool`), así que cada clase de instrumento puede soltar a su manera: una voz del sintetizador pasa a su release, un sampler apaga su sonido rápidamente, un MIDI Out envía un note-off. Las teclas pulsadas en un teclado

y las previsualizaciones no tienen origen, así que nunca se cortan. `notes_stop_when_the_song_no_longer_plays_them` lo comprueba.

El closure toma prestados `self.plays` y `self.pattern_lengths`, y el bucle toma prestado `self.channels` para cambiarlo. Está permitido porque son campos distintos: desde la edición 2021, un closure que usa `self.plays` captura solo ese campo, no todo `self`, así que escribir `self.plays` dentro del closure también funcionaría. Las dos referencias locales solo dejan claro qué partes de `self` lee.

Saltar, y cambiar de modo

`Command::Seek(step)` mueve la posición de inicio. Con la reproducción parada, es donde empieza el siguiente `Play`. Mientras suena, la reproducción salta allí en el acto:

```
fn seek(&mut self, step: usize) {
    self.start = step;
    self.pending.clear();
    ...
    if self.playing {
        // Notes that were playing would otherwise hang on
        // until the end they will now never reach.
        for channel in &mut self.channels {
            channel.instrument.release_all();
        }
        ...
        self.sequencer.start_at(step);
    }
}
```

Las notas que esperan en `pending` son de la posición anterior, así que se descartan. Cambiar entre modo patrón y modo canción empieza el bucle nuevo desde arriba, y suelta las notas sostenidas por la misma razón. Parar suelta todas las notas, también las teclas pulsadas: cada una pasa a su release, así que el sonido se apaga en un momento y no queda colgando.

Un patrón sobre la canción

▶ **Tocar sobre la canción**, en el rack de canales, toca el patrón seleccionado sobre la canción mientras suena, para probarlo con el resto de la banda

(`Command::SetOverlay`). Empieza en la siguiente línea de compás de la canción, para entrar a tiempo, y se repite mientras suena:

```
fn overlay_step(&mut self, step: usize) -> Option<(usize, usize)> {
    let bar_line = click_at(&self.song_clicks, step) == Click::Bar;
    let overlay = self.overlay.as_mut()?;
    // After a jump it waits for a bar line, to start again from its
    // top in time with the bars.
    if overlay.last.is_some_and(|last| step != last + 1)
        && let Some((pattern, _)) = overlay.playing.take()
    {
        overlay.coming.get_or_insert(pattern);
    }
    overlay.last = Some(step);
    if bar_line && let Some(pattern) = overlay.coming.take() {
        overlay.playing = Some((pattern, 0));
    }
    let (pattern, at) = overlay.playing?;
    let length = self.pattern_lengths.get(pattern).copied().unwrap_or(0);
    if length == 0 {
        return None;
    }
    let at = at % length;
    overlay.playing = Some((pattern, at + 1));
    Some((pattern, at))
}
```

Las líneas de compás salen de los golpes del metrónomo de la canción, que ya dicen dónde empieza cada compás en cualquier compás. Un patrón elegido mientras otro suena sobre la canción espera como `coming` hasta la siguiente línea de compás. Un salto (un seek, o la canción que vuelve a empezar) se nota porque el paso no es el siguiente al último, y el patrón vuelve a empezar desde arriba en la siguiente línea de compás.

La ventana dibuja un segundo cursor en el rack para el patrón que suena sobre la canción. Dos números, el patrón y su paso, tienen que leerse juntos, o la ventana podría ver un patrón nuevo con un paso viejo. Así que comparten un único `AtomicU64` , el patrón en los 32 bits altos y el paso en los 32 bajos:

```
fn store_overlay(overlay: &AtomicU64, playing: Option<(usize, usize)>) {
    let bits = playing.map_or(NO_POSITION, |(pattern, step)| {
        ((pattern as u64) << 32) | step as u64
    });
    overlay.store(bits, Ordering::Relaxed);
}
```

`<<` desplaza los bits del patrón 32 lugares hacia arriba, y `|` (o bit a bit) pone el paso en la mitad baja, que está vacía. Leer lo deshace con `>> 32` y una máscara, `bits &`

`u64::from(u32::MAX)`. Una carga, un guardado: un entero atómico no se puede leer mitad viejo y mitad nuevo. Un valor especial, `NO_POSITION`, significa «nada suena sobre la canción».

Cómo se prueba

Los tests hacen funcionar el hilo de audio sin tarjeta de sonido (mira Probar sin tarjeta de sonido). Una pequeña función auxiliar encuentra dónde empiezan los sonidos en un bloque de muestras:

```
fn onsets(out: &[f32]) -> Vec<usize> {
    (0..out.len())
        .filter(|&i| out[i] != 0.0 && (i == 0 || out[i - 1] == 0.0))
        .collect()
}
```

El clic de prueba es una ráfaga corta después de silencio, así que cada golpe es un inicio. Entonces una canción se comprueba en una línea:

```
#[test]
fn song_mode_plays_the_clips() {
    let mut h = two_patterns();
    // Pattern 1 at bar 1, pattern 0 at bar 2: a two-bar song.
    let plays = vec![whole(1, 0), whole(0, 16)];
    send(&mut h, Command::SetSong { plays, length: 32 });
    send(&mut h, Command::SetMode(PlayMode::Song));
    send(&mut h, Command::Play);
    let out = render(&mut h, 40 * STEP);
    // Step 2 of the first clip, step 0 of the second, then the song
    // loops back to the start: step 2 of the first clip again.
    assert_eq!(onsets(&out), vec![2 * STEP, 16 * STEP, 34 * STEP]);
    assert_eq!(
        h.playhead.load(Ordering::Relaxed),
        7,
        "step 39 is step 7 of the second time round"
    );
}
```

La última aserción lleva un mensaje que dice *por qué* 7 es correcto: después de 40 pasos de una canción de 32, el secuenciador va por su segunda vuelta. Quien espere 39 ve la razón en cuanto el test falla.

Otros tests cubren el resto del capítulo: clips que se solapan, trozos que repiten su patrón, patrones insertados y quitados debajo de los demás, `Clear` mandándolo

todo a la basura, `Play` empezando donde lo puso un seek, y siete tests sobre un patrón tocado sobre la canción.

Tempo, compás, swing y sensación

Archivos: `crates/lineup-core/src/meter.rs`, `crates/lineup-core/src/tempo.rs`, `crates/lineup-core/src/remeter.rs`, `crates/lineup-core/src/swing.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-app/src/playlist/tempo.rs`, `crates/lineup-app/src/time_picker.rs` y `crates/lineup-app/src/app/swing.rs`

En Ferrous Lineup un paso es siempre una semicorchea. Un vals, un ritardando, un shuffle y un 7/8 búlgaro se escriben todos sobre esa misma cuadrícula. Lo que cambia es cómo se *cuentan* los pasos y cómo se *tocan*:

Qué	Qué cambia	Dónde
Una indicación de compás	Dónde caen las líneas de compás y de pulso; nunca una nota	`meter.rs`
Un cambio de tempo	Cuánto dura cada paso	`tempo.rs`, `engine.rs`
Pasar la canción a otro compás	Todas las notas, para que cada compás sea un compás del nuevo	`remeter.rs`
El swing	Cuándo se toca la segunda corchea de cada pulso, no dónde está escrita	`swing.rs`
La soltura	Cuánto tarde empieza cada nota, al tocar	`engine.rs`

La música que hay detrás de todo esto (pulsos y compases, el dos y el tres, las métricas irregulares, qué es un ritardando y qué es el swing) se explica en el manual de usuario: Ritmo y métrica, Tempo, compases y cambios de tonalidad, Swing y sensación y Métricas irregulares y los Balcanes. Este capítulo trata del código.

`TimeSignature`: dos números y una agrupación

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
#[serde(from = "SavedTime", into = "SavedTime")]
pub struct TimeSignature {
    /// How many notes of `unit` length make a bar: the top number.
    pub beats: u8,
    /// Which note counts as a beat: 4 for quarter notes, 8 for eighths.
    /// The bottom number.
    pub unit: u8,
    /// Where the felt beats start, when they are not the usual ones: bit
    /// `i` is set when a beat starts on the bar's `i`-th `unit` note.
    /// 0 means the usual beats (see [`TimeSignature::usual_groups`]).
    ///
    /// A `u64` keeps `TimeSignature` `Copy`, so it can be passed around
    /// as freely as a number; a `Vec` could not be.
    groups: u64,
}
```

El número de arriba puede ser cualquiera, desde 1 hasta lo que quepa en un patrón; el de abajo es 1, 2, 4, 8 o 16 (`UNITS`). No hay 32: un compás de 7/32 terminaría en mitad de un paso de semicorchea.

Todo lo demás se calcula a partir de esos números, y la aritmética está escrita para que un archivo dañado no pueda hacerla dividir por cero:

```
pub fn unit_steps(self) -> usize {
    (16 / usize::from(self.unit.clamp(1, 16))).max(1)
}

pub fn steps_per_bar(self) -> usize {
    let unit = usize::from(self.unit.clamp(1, 16));
    (usize::from(self.beats) * 16 / unit).max(1)
}
```

`clamp` deja la unidad entre 1 y 16, así que la división siempre tiene por qué dividir, y `max(1)` mantiene un compás de al menos un paso, porque «¿en qué compás está el paso 50?» divide por la longitud del compás. Los menús y `set_time_marker` solo aceptan indicaciones de compás que pasan `is_valid`, pero un archivo editado a mano todavía puede decir `beats: 0` o `unit: 0`, y el test `nonsense_does_not_divide_by_zero` comprueba que la aritmética lo aguanta.

`new` es una `const fn`, así que las indicaciones de compás pueden ser constantes:

```
pub const COMMON: TimeSignature = TimeSignature::new(4, 4);

pub const ALL: [TimeSignature; 10] = [
    TimeSignature::new(2, 4),
    TimeSignature::new(3, 4),
    // ...
];
```

Una agrupación dentro de un tipo Copy

En 7/8 las siete corcheas se sienten como pulsos de dos y de tres, y el orden importa: 2+2+3/8 es una rachenitsa búlgara, 3+2+2/8 un kalamatianós griego. La agrupación es parte de la indicación de compás.

`TimeSignature` se pasa por valor en todas partes, y para eso tiene que ser `Copy`: se copia como un entero, sin memoria en el montón. Lo natural sería guardar la agrupación en un `Vec<u8>`, `[2, 2, 3]`, pero un tipo que contiene un `Vec` no puede ser `Copy`. Así que la agrupación es una **máscara de bits**: el bit `i` está encendido cuando un pulso empieza en la nota `i` del compás. 2+2+3/8 empieza sus pulsos en las notas 0, 2 y 4, así que su máscara es `0b10101`. Un `u64` tiene sitio para 64 notas (`MAX_GROUPED_BEATS`), más que cualquier compás real.

Las canciones y los ritmos son tablas `const`, así que una agrupación también tiene que poder construirse al compilar:

```
pub const fn of_groups(groups: &[u8], unit: u8) -> Self {
    let mut mask = 0_u64;
    let mut at = 0;
    let mut index = 0;
    // `for` loops are not allowed in a `const fn` yet: `while` is.
    while index < groups.len() {
        assert!(groups[index] > 0, "a group of no notes");
        assert!(at < Self::MAX_GROUPED_BEATS as usize, "too many notes");
        mask |= 1 << at;
        at += groups[index] as usize;
        index += 1;
    }
    assert!(at > 0 && at <= Self::MAX_GROUPED_BEATS as usize);
    let mut time = TimeSignature::new(at as u8, unit);
    // The usual grouping is written the plain way, so 3+3/8 and 6/8
    // are the same time signature.
    if mask != time.usual_mask() {
        time.groups = mask;
    }
    time
}
```

`1 << at` es un 1 desplazado `at` posiciones a la izquierda, y `|=` enciende ese bit sin tocar los demás. Un `assert!` que falla dentro de una `const fn` evaluada para una constante detiene la **compilación**, no el programa, así que una agrupación mal escrita en la tabla de una canción nunca llega a nadie. Para las agrupaciones que escribe una persona está `grouped`, que comprueba lo mismo y devuelve un `Option`.

Las últimas líneas mantienen una regla: la agrupación habitual se guarda como 0. Así `"3+3/8"` y `"6/8"` dan el mismo valor, y `==` dice que son iguales. Sin ella, el menú Compás mostraría 6/8 sin seleccionar después de que alguien escribiera 3+3.

El campo `groups` es privado, así que el código de fuera de `meter.rs` solo puede crear una agrupación con `grouped` u `of_groups`. `beats` y `unit` son públicos, eso sí, de modo que alguien podría cambiar `beats` y dejar una máscara que ya no encaja. El `mask()` privado lo cubre: una máscara que no encaja con el número de arriba se ignora, y se usan los pulsos habituales.

Pulsos de distinta longitud

En 2+2+3/8 los pulsos duran 4, 4 y 6 pasos, así que «¿en qué pulso está este paso?» no puede ser una división. Todo le pregunta a `beat_at`:

```
pub struct Beat {
    /// Counted from 0: the first beat of the bar is 0.
    pub index: usize,
    /// The step it starts on, counted from the start of the bar.
    pub start: usize,
    /// How many steps it lasts.
    pub length: usize,
}
```

Del test `irregular_meters_group_their_beats`, donde `typed` es una forma corta de `text.parse().unwrap()`:

```
let balkan = typed("2+2+3/8");
assert_eq!(balkan.beat_starts(), [0, 4, 8]);
assert_eq!(balkan.beat_at(9).length, 6);
assert_eq!(balkan.beat_at(9).index, 2);
```

`beat_at` busca el último comienzo de pulso en el paso o antes con `rposition`, que recorre un slice desde el final.

No hay ninguna función de «pasos por pulso», a propósito. Cualquier código que supusiera pulsos iguales estaría mal en una métrica irregular, y sin una función así

que llamar nadie puede cometer ese error. (Quitar una función es además la forma más rápida de encontrar a todos los que la llaman: el compilador los enumera.)

A un compás se le puede preguntar por sus pulsos de tres maneras, porque tres partes del programa entienden cosas distintas por «pulso»:

Función	Responde	4/4	6/8	7/8	2+2+3/8	5/4
`groups`	los pulsos que marca un metrónomo, en unidades	1 1 1 1	3 3	1×7	2 2 3	1×5
`usual_groups`	las notas de la unidad en un pulso habitual	1	3	1	1	1
`felt_groups`	los pulsos en que se apoya un músico	2 2	3 3	2 2 3	2 2 3	2 3

`felt_groups` es lo que usa la banda para su «um-pa»: la agrupación si la hay, los pulsos de tres corcheas del 6/8, y si no, pulsos de dos con uno de tres al final. Los compases de tres notas o menos, como 3/4, son un solo pulso sentido. Cómo los toca la banda está en La banda.

Leer y escribir «2+2+3/8»

`TimeSignature` implementa `Display`, así que `format!("{time}")` escribe «3/4», o «2+2+3/8» con la agrupación como número de arriba, en todas partes del mismo modo. La dirección contraria es el trait estándar `FromStr`, que es lo que hace funcionar `"7/8".parse::<TimeSignature>()`:

```
impl FromStr for TimeSignature {
    type Err = String;

    fn from_str(text: &str) -> Result<Self, Self::Err> {
        let (top, bottom) = text
            .split_once('/')
            .ok_or(tr("write it as two numbers, like 7/8"))?;
        // ... then the bottom number, the top number or numbers, and the
bar's size
    }
}
```

`split_once('/')` da `Some((antes, después))` o `None`; `ok_or` convierte el `None` en un error, y `?` lo devuelve en el acto. El error es un `String` que el menú Compás muestra tal cual, en rojo, así que los mensajes pasan por `tr` y salen en el idioma de la aplicación («escríbelo como dos números, como 7/8»; ver Dos idiomas).

La fila «Otro» del menú Compás guarda lo que se va escribiendo en un pequeño `Draft` (el número de arriba, el de abajo, y la agrupación como texto) en la memoria de egui entre fotogramas, y construye la indicación de compás con el mismo `parse`. Si los grupos no suman el número de arriba, lo dice: «los grupos suman 8, no 11».

Guardarla sin cambiar los archivos viejos

Una indicación de compás sin agrupación tiene que guardarse exactamente como antes, `(beats: 7, unit: 8)`, para que las versiones anteriores sigan abriendo el archivo. El atributo `#[serde(from = "SavedTime", into = "SavedTime")]` de `TimeSignature` dice: para guardarla, conviértela antes en un `SavedTime`; para cargarla, lee un `SavedTime` y conviértelo de vuelta:

```
#[derive(Serialize, Deserialize)]
struct SavedTime {
    beats: u8,
    unit: u8,
    #[serde(default, skip_serializing_if = "Vec::is_empty")]
    groups: Vec<u8>,
}
```

`SavedTime` puede tener un `Vec` cómodo (`groups: [2, 2, 3]`) porque solo vive un momento, mientras se guarda o se carga. La conversión `From<SavedTime>` deja fuera una agrupación que no suma el número de arriba, así que un archivo editado a mano no puede crear un compás cuyos pulsos no encajen.

`Meter`: indicaciones de compás que cambian

Una canción puede cambiar de compás a mitad de camino: la última línea de un himno en 3/4, la Promenade de Músorgski alternando 5/4 y 6/4. Cada cambio es una marca en la lista de reproducción:

```
pub struct TimeMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub time: TimeSignature,
}
```

Una marca siempre empieza un compás nuevo. Una que se pone en mitad de un compás lo deja más corto, que es como una partitura escribe un compás al que le falta un pulso.

Con marcas, «¿en qué compás está el paso 50?» depende de todo lo que hay antes del paso 50, así que todas las preguntas sobre compases van a un solo tipo:

```
pub struct Meter {
    first: TimeSignature,
    /// Sorted by start, all after step 0, no two on the same step.
    markers: Vec<TimeMarker>,
}
```

El comentario de `markers` es un *invariante*, una regla que siempre se cumple. Los campos son privados y `Meter::new` la hace cumplir (quita las marcas del paso 0, ordena las demás y elimina duplicados con `dedup_by_key`), así que ningún método tiene que volver a comprobarla.

`bar_at` recorre la canción por **tramos**, uno por indicación de compás, contando los compases que pasa:

```
4/4 desde 0, 3/4 desde 32:
tramo 1: pasos 0..32,  16 por compás -> compases 0 y 1
tramo 2: pasos 32..,   12 por compás -> compases 2, 3, 4, ...
```

```
fn stretches(&self) -> impl Iterator<Item = (usize, usize, TimeSignature)>
+ '_ {
    let starts =
        std::iter::once(TimeMarker::new(0,
self.first)).chain(self.markers.iter().copied());
    let ends = self
        .markers
        .iter()
        .map(|marker| marker.start)
        .chain(std::iter::once(usize::MAX));
    starts
        .zip(ends)
        .map(|(from, end)| (from.start, end, from.time))
}

pub fn bar_at(&self, step: usize) -> Bar {
    let mut bars_before = 0;
    for (start, end, time) in self.stretches() {
        let bar = time.steps_per_bar();
        if step < end {
            let index = (step - start) / bar;
            let bar_start = start + index * bar;
            return Bar {
                number: bars_before + index,
                start: bar_start,
                length: bar.min(end - bar_start),
                time,
            };
        }
        bars_before += (end - start).div_ceil(bar);
    }
    unreachable!("the last stretch never ends")
}
```

`stretches` construye un iterador a partir de otros: los comienzos (la primera indicación de compás, luego cada marca) se emparejan con `zip` con los finales (cada marca, luego `usize::MAX` para «nunca»). `div_ceil` divide redondeando hacia arriba, así que un compás corto antes de una marca sigue contando como compás, y `bar.min(end - bar_start)` le da a ese compás corto su longitud real. `unreachable!` le dice al lector, y al compilador, que el bucle siempre devuelve algo: el último tramo termina en `usize::MAX`.

`bars()` da todos los compases, sin fin:

```rust
pub fn bars(&self) -> impl Iterator<Item = Bar> + '_ {
    let mut next = 0;
    std::iter::from_fn(move || {
        let bar = self.bar_at(next);
        next = bar.end();
        Some(bar)
    })
}
```

`std::iter::from_fn` convierte un closure en un iterador: cada llamada a `next` ejecuta el closure. Nunca devuelve `None`, así que el iterador no se detiene solo; la lista de reproducción dibuja su cuadrícula con `meter.bars().take_while(|bar| bar.start <= steps)` y decide ella cuánto usar.

El resto de `Meter` son las preguntas que hace la aplicación: `time_at`, `bar(number)`, `is_bar_start`, `floor_to_bar`, `floor_to_beat`, `beat_end`, `round_up_to_bar`, y `position(step)` para el indicador COMPÁS:PULSO:PASO. `Meter::steady(time)` es una canción sin marcas, y un test comprueba, paso a paso, que responde exactamente igual que la `TimeSignature` sola. `tidy()` quita las marcas que no cambian nada: una que mantiene la indicación de compás anterior y cae en una de sus líneas de compás.

Quién le pregunta al compás

- **La lista de reproducción**: su cuadrícula, los números de compás, las secciones y el ajuste. `Snap::floor` y `Snap::next` dan el principio y el final del compás o del pulso bajo el puntero, que es lo que necesita arrastrar un clip, una sección o el bucle.
- **El rack de canales y el piano roll** muestran un patrón, no la canción. Preguntan `Project::pattern_time(pattern)`: la indicación de compás donde empieza el primer clip del patrón, o, para un patrón que no está en la canción, su propio `Pattern::time` (un vals probado desde la Biblioteca está en 3/4 aunque la canción esté en 4/4), o si no, la del proyecto.
- **La longitud de la canción**, redondeada hasta la siguiente línea de compás.
- **El menú de longitud del patrón**, que ofrece compases enteros (`pattern_lengths`: 1, 2, 3, 4, 6, 8, 12 o 16, mientras quepan) y escribe «4 compases» en vez de «64 pasos» (`describe_length`).

La canción tal como suena

Las secciones pueden repetirse (Secciones, repeticiones y bucles), así que la canción tal como se oye es más larga que la lista de reproducción tal como se dibuja. Si una sección en 3/4 suena dos veces, la canción que se oye tiene dos tramos de 3/4. Las *pasadas* del arreglo dicen qué parte de la línea de tiempo suena cuándo, y `Project::played_meter` construye un segundo `Meter`, en pasos de la canción:

```
pub fn played_meter(&self, arrangement: &Arrangement) -> Meter {
    let meter = self.meter();
    let first = arrangement
        .passes
        .first()
        .map_or(self.time_signature, |pass| meter.time_at(pass.from));
    let mut markers = Vec::new();
    for pass in &arrangement.passes {
        markers.push(TimeMarker::new(pass.start,
meter.time_at(pass.from)));
        for marker in meter.markers() {
            if marker.start > pass.from && marker.start < pass.from +
pass.length {
                markers.push(TimeMarker::new(
                    pass.start + marker.start - pass.from,
                    marker.time,
                ));
            }
        }
    }
    Meter::new(first, markers).tidy()
}
```

Cada pasada recibe una marca con la indicación de compás en que empieza, más las marcas que hay dentro, y luego `tidy` quita las que no cambian nada, como una pasada en 4/4 después de otra en 4/4 sobre una línea de compás. El transporte usa este compás para el indicador de posición y para RePág/AvPág, así que «el compás siguiente» es el siguiente que se oye.

El metrónomo también lo usa. `played_clicks` lo convierte en un `Click` por cada paso de la canción (`Bar`, `Beat` o `None`), y la aplicación envía esa lista al hilo de audio con `Command::SetSongClicks`, así que el metrónomo marca los pulsos de cada compás: las dos negras con puntillo del 6/8, los tres grupos del 2+2+3/8, un clic más agudo en cada «uno». En modo patrón es `pattern_clicks`, en el compás del propio patrón. La cuenta previa y el sonido del metrónomo están en Herramientas de práctica.

Cambios de tempo

Tempo, siempre válido

```
#[derive(Debug, Clone, Copy, PartialEq, PartialOrd, Serialize,
Deserialize)]
#[serde(from = "f32", into = "f32")]
pub struct Tempo {
    bpm: f32,
}
```

El campo es privado, así que la única forma de crear un `Tempo` es `Tempo::new`, que deja el valor entre `MIN` (10) y `MAX` (522) BPM y convierte un NaN en el tempo por defecto, 130. Todo `Tempo` del programa es, por tanto, válido. Los archivos pasan por la misma puerta: un tempo se guarda como un número sin más, y `From<f32>` lo lee con `Tempo::new`, así que un archivo que dice `tempo: 9000.0` se abre a 522 (Guardar).

Marcas, y un mapa

Un cambio de tempo es una marca en la lista de reproducción, como un cambio de compás:

```
pub struct TempoMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub tempo: Tempo,
    /// How many steps the change takes; 0 for at once. Project files
    /// leave it out when it is 0.
    #[serde(default, skip_serializing_if = "is_zero")]
    pub glide: usize,
}
```

Con `glide` en 0 el tempo cambia de golpe, como después de una pausa. Con un `glide` se mueve un poco en cada paso hasta llegar: hacia abajo en un ritardando, hacia arriba en un accelerando. Un paso es una semicorchea, así que un cambio gradual a lo largo de dos compases de 4/4 son 32 cambios pequeños, tan cerca de una curva que nadie oye los escalones. El tempo nunca cambia en mitad de un paso.

```
fn bpm_on(&self, step: usize, from: f32) -> f32 {
    let into = step - self.start + 1;
    if into >= self.glide {
        return self.tempo.bpm();
    }
    let done = into as f32 / self.glide as f32;
    from + (self.tempo.bpm() - from) * done
}
```

Un cambio gradual de 4 pasos de 100 a 60 BPM toca 90, 80, 70 y 60: el último paso del cambio ya está en el tempo nuevo.

`TempoMap` es al tempo lo que `Meter` a los compases: el primer tempo y las marcas, con el mismo invariante, y una pregunta principal:

```
pub fn tempo_at(&self, step: usize) -> Tempo {
    // The marker in effect so far, and the tempo its glide started from.
    let mut current: Option<(&TempoMarker, f32)> = None;
    let bpm_on = |current: Option<(&TempoMarker, f32)>, step: usize| match
current {
        Some((marker, from)) => marker.bpm_on(step, from),
        None => self.first.bpm(),
    };
    for marker in self.markers.iter().take_while(|m| m.start <= step) {
        let from = bpm_on(current, marker.start - 1);
        current = Some((marker, from));
    }
    Tempo::new(bpm_on(current, step))
}
```

Un cambio gradual parte del tempo que sonaba en el paso anterior a su marca, aunque fuera en mitad de otro cambio gradual, así que el bucle lleva `from` consigo: cada marca le pregunta a la anterior «¿dónde estabas un paso antes de mí?». `marker.start - 1` no puede desbordarse hacia abajo, porque ninguna marca está en el paso 0.

La franja de tempo

La franja que está sobre los números de compás de la lista de reproducción dibuja el mapa como una línea. Solo necesita los pasos donde la línea se dobla (el principio, cada marca y el final de cada cambio gradual), unidos de dos en dos con `windows(2)`:

```
let mut bends = vec![0, steps];
for marker in map.markers() {
    bends.push(marker.start);
    bends.push(marker.start + marker.glide);
}
bends.retain(|&step| step <= steps);
bends.sort_unstable();
bends.dedup();
let mut points = Vec::new();
for pair in bends.windows(2) {
    let (from, to) = (pair[0], pair[1]);
    points.push(pos2(layout.x(from), y(map.tempo_at(from))));
    points.push(pos2(layout.x(to),
y(map.tempo_at(to.saturating_sub(1)))));
}
```

Cada tramo va del tempo de su primer paso al tempo del último, así que un cambio de golpe se convierte en un salto vertical y un cambio gradual en una pendiente. Las etiquetas se escriben como las escribe una partitura: `describe` compara el tempo de la marca con el de `tempo_before` de su paso, y escribe «♩ = 90» para un cambio de golpe, «rit. → 60» para uno gradual hacia abajo y «accel. → 140» para uno hacia arriba.

El cuadro TEMPO de la barra de herramientas fija el tempo con el que *empieza* la canción, y `scale_tempo` mueve todas las marcas en la misma proporción:

```
pub fn scale_tempo(&mut self, tempo: Tempo) {
    let ratio = tempo.bpm() / self.tempo.bpm();
    self.tempo = tempo;
    for marker in &mut self.playlist.tempo_markers {
        marker.tempo = Tempo::new(marker.tempo.bpm() * ratio);
    }
}
```

Practicar una pieza más despacio conserva su ritardando, solo que más lento también. Mientras suena una canción cuyo tempo cambia, el cuadro muestra TEMPO AHORA: `tempo_at` del lugar del cabezal en la línea de tiempo.

Un estilo de la Biblioteca trae un tempo propio, y ese tiene que caer donde la canción toca de verdad, no en su primera nota: un archivo MIDI empieza a menudo con unas pocas notas lentas antes de su tempo real. Así que `main_tempo` busca el tempo que la canción mantiene durante más pasos, y `take_tempo` escala toda la canción para que ese se convierta en el del estilo:

```
pub fn take_tempo(&mut self, tempo: Tempo) {
    let ratio = tempo.bpm() / self.main_tempo().bpm();
    self.scale_tempo(Tempo::new(self.tempo.bpm() * ratio));
}
```

`main_tempo` suma, para cada tempo, los pasos que dura: desde el principio hasta la primera marca con el tempo inicial, y luego desde cada marca hasta la siguiente. Un tempo que vuelve más tarde se suma a lo que ya duraba. De los que duran más, gana el primero, así que una canción con un único tempo constante obtiene ese, y un empate va al tempo anterior:

```
let (mut tempo, mut from) = (self.tempo, 0);
for marker in &self.playlist.tempo_markers {
    let start = marker.start.clamp(from, end.max(from));
    hold(tempo, start - from);
    (tempo, from) = (marker.tempo, start);
}
hold(tempo, end.saturating_sub(from));
```

`hold` es un cierre que toma prestada la lista `held` de forma mutable, así que se declara `let mut hold`: llamar a un cierre que cambia lo que capturó cambia el estado del propio cierre. `(tempo, from) = (…, …)` asigna dos variables a la vez, una *asignación por desestructuración*.

Cada paso a su propio tempo

`Project::played_tempos` da el tempo de cada paso de la canción *tal como suena*, así que una sección que frena al final frena cada vez que se repite:

```
pub fn played_tempos(&self, arrangement: &Arrangement) -> Vec<Tempo> {
    let map = self.tempo_map();
    if map.is_steady() {
        return Vec::new();
    }
    (0..arrangement.length)
        .map(|step| {
            let timeline =
arrangement.timeline_step(step).unwrap_or(step);
            map.tempo_at(timeline)
        })
        .collect()
}
```

La lista se construye en el hilo de la interfaz y se envía con `Command::SetSongTempos`, junto con la canción. El hilo de audio no puede reservar memoria, y cuando llega una

lista nueva la vieja vuelve al hilo de la interfaz para liberarse, como las canciones viejas y las notas viejas (Dos hilos). Una lista vacía, el caso habitual, significa «toca el tempo del cuadro TEMPO», y no cuesta nada.

Un paso tiene que durar lo que dice *su propio* tempo, así que el tempo debe fijarse antes de que el paso empiece. `due`, del secuenciador, dice qué paso empieza en el próximo tick (El secuenciador), y el hilo de audio lo pregunta primero, en cada frame:

```
if let Some(step) = self.sequencer.due() {
    // ...
    let tempo = self.tempo_of_step(step);
    if tempo != self.tempo_now {
        self.set_tempo(tempo);
    }
}
if let Some(step) = self.sequencer.tick() {
```

`tempo_of_step` es el tempo de la canción para ese paso en modo canción, y el del cuadro TEMPO en modo patrón, donde se trabaja sobre un patrón y se quiere un bucle estable. (La parte que se omite arriba, `lap_tempos`, es el próximo tempo del entrenador de velocidad, que entra cuando el bucle vuelve a empezar: Herramientas de práctica.)

Las notas que ya suenan tienen una longitud en frames, calculada con el tempo anterior. `set_tempo` las estira para que sigan terminando en el paso que deben:

```
fn set_tempo(&mut self, tempo: Tempo) {
    self.tempo_now = tempo;
    let before = self.sequencer.step_frames();
    let next_step = self.sequencer.frames_to_next_step();
    self.sequencer.set_tempo(tempo);
    let stretch = self.sequencer.step_frames() / before;
    for channel in &mut self.channels {
        channel.instrument.tempo_changed(next_step, stretch);
        channel.effects.set_tempo(tempo);
    }
    // ...
}
```

y cada instrumento calcula lo que queda de cada nota sostenida:

```
pub(crate) fn stretched(left: f64, next_step: f64, stretch: f64) -> f64 {
    if left <= next_step {
        left
    } else {
        (next_step + (left - next_step) * stretch).round()
    }
}
```

Los frames hasta el próximo paso pertenecen al paso que ya suena, que conserva su longitud; solo se estira el resto. A una nota con 10 000 frames por delante, 2 000 de ellos en este paso, a la mitad del tempo (un estiramiento de 2) le quedan 2 000 + 8 000 × 2 = 18 000. Los efectos que siguen el pulso, como un eco ajustado en valores de nota, también reciben el tempo nuevo (El mezclador).

Una exportación toca la canción con el mismo hilo de audio, así que recibe la misma lista. Su longitud ya no es «pasos por la longitud de un paso»: `played_seconds` suma cada paso a su propio tempo, con `tempo::seconds`:

```
pub fn seconds(steps: usize, tempo: impl Fn(usize) -> Tempo) -> f64 {
    (0..steps)
        .map(|step| f64::from(tempo(step).seconds_per_beat()) / 4.0)
        .sum()
}
```

`impl Fn(usize) -> Tempo` acepta cualquier closure que vaya de un paso a un tempo, así que la misma función sirve para un `TempoMap` en un test y para la lista tocada en `played_seconds`.

Pasar una canción a otro compás

Una indicación de compás solo mueve líneas de compás. Pero si eliges un vals en la Biblioteca mientras suena una canción en 4/4, sus compases de doce pasos puestos en compases de dieciséis tocan «un dos tres» y luego un pulso de silencio, en cada compás. Para sonar a vals, cada compás de la canción tiene que convertirse en un compás de 3/4. `remeter` lo hace, con una regla: **el compás n de la canción vieja pasa a ser el compás n de la nueva, y cada nota conserva su lugar en su compás**.

```
4/4: | Do . . . Mi . . . Sol . . . Mi . . . |   16 pasos
3/4: | Do . . Mi . . Sol . . Mi . . |           12 pasos
```

Es un estiramiento mecánico, como lo haría un magnetófono, no como un compositor reescribiría una melodía para otro compás.

Apretar un compás, al tick

Apretar dieciséis pasos en doce deja muchas notas entre pasos: corcheas separadas por dos pasos pasan a estar a 1,5 pasos. Redondearlas a la cuadrícula las volvería

desiguales. Así que el apretón se hace en **ticks**, 24 por paso (`TICKS_PER_STEP`, la unidad que usa la importación MIDI para las notas fuera de la cuadrícula, ver MIDI):

```
pub struct Remap {
    old: Meter,
    /// Steps in one new bar.
    bar: usize,
}

impl Remap {
    pub fn tick(&self, tick: usize) -> usize {
        let bar = self.old.bar_at(tick / TICKS_PER_STEP);
        let start = bar.start * TICKS_PER_STEP;
        let length = bar.length * TICKS_PER_STEP;
        let new_bar = self.bar * TICKS_PER_STEP;
        // Rounded to the nearest tick.
        bar.number * new_bar + ((tick - start) * new_bar + length / 2) /
length
    }
}
```

En la última línea hay dos costumbres de la aritmética con enteros. **Multiplicar antes de dividir**: `(tick - start) / length * new_bar` daría 0 para cada tick dentro del compás. **Sumar la mitad antes de dividir** para redondear al entero más cercano en vez de siempre hacia abajo. La segunda corchea de un compás de 4/4, el tick 48 de 384, cae en (48 × 288 + 192) / 384 = 36 de un compás de 3/4 de 288 ticks: 1,5 pasos, exactos.

Un `Remap` guarda el `Meter` viejo de la canción, así que sirve para canciones que ya cambian de compás: cada compás viejo, en cualquier indicación (también uno corto), se convierte en un compás nuevo entero. `Remap::step` es lo mismo, redondeado al paso más cercano, para lo que vive en pasos.

Cada compás conserva su duración

El tempo cuenta negras. Un compás de 3/4 tiene tres, uno de 4/4 cuatro, así que con el mismo tempo la canción en 3/4 terminaría en tres cuartas partes del tiempo. Por eso `remeter` escala el tempo en la misma proporción que el compás: 4/4 a 112 pasa a ser 3/4 a 84, y un compás dura 2,14 segundos en los dos. Las marcas de tempo se escalan igual, y un cambio de compás en mitad de la canción que cambiaba la longitud del compás recibe una marca de tempo propia, para que cada tramo conserve su paso.

El escalado tiene que volver exacto. 108,5 BPM en 7/8, devuelto a 4/4, tiene que ser 108,5 × 16 / 14 = 124, no 124,00001. Un `f32` tiene unas siete cifras significativas, y la

razón 16/14 no tiene valor exacto en él. Así que el código multiplica primero y divide al final, en `f64`:

```
let scaled = |tempo: Tempo, step: usize| {
    Tempo::new((f64::from(tempo.bpm()) * bar as f64 / old_bar(step) as
f64) as f32)
};
```

108,5 × 16 = 1736 es exacto, y 1736 / 14 = 124 también. Un test lleva varios tempos por 7/8, 3/4, 13/16, 6/8 y 12/8 y de vuelta a 4/4, y comprueba que vuelven sin cambios.

Así que después de `remeter` cada nota sigue empezando en el mismo momento, en segundos, y dura lo mismo. La música suena igual; solo cambia cómo se escribe y cómo se cuenta. Lo que hace que una canción *suene* a vals es la batería y la forma de atacar los acordes, y por eso la Biblioteca y el cuadro Compás pasan la canción al nuevo compás **y** le dan un ritmo en él. Cómo se elige ese ritmo está en La biblioteca.

Patrones que se quedan largos

Un patrón tiene como mucho 256 pasos (`MAX_STEPS`). Dieciséis compases de 4/4 son justo eso, pero dieciséis compases de 12/8 son 384. En vez de perder notas, `remeter` corta un patrón así en trozos de compases enteros que quepan:

```
let chunk = MAX_STEPS / bar * bar;
```

256 / 24 × 24 son 240 pasos: diez compases de 12/8, y luego seis. Los trozos se llaman «Tema», «Tema (2)» y así sucesivamente, y cada clip del patrón se convierte en clips de los trozos, uno detrás de otro. La canción suena igual; la lista de reproducción muestra un clip más.

Todo lo demás se mueve con su compás

Los clips, las secciones, el bucle, las marcas de tonalidad y las marcas de la banda pasan todos por `song.step`. El cambio gradual de una marca de tempo se traslada por sus dos extremos, así que un ritardando a lo largo de dos compases de 4/4 dura dos compases de 3/4. Los acordes cambian en un pulso, así que van al pulso más cercano del compás nuevo (`snap_to_beat`). Las marcas de compás desaparecen: ahora toda la canción está en una sola indicación de compás.

Dos marcas a un paso de distancia pueden quedar apretadas en el mismo paso. `Vec::dedup_by` quita los vecinos que su closure considera iguales; el closure recibe

primero el elemento posterior y después el anterior, y puede cambiar el anterior, que es como se escribe «gana la marca posterior»:

```
playlist_mut.tempo_markers.dedup_by(|later, earlier| {
    later.start == earlier.start && {
        *earlier = *later;
        true
    }
});
```

`a && { ... }` ejecuta el bloque solo cuando `a` es verdadero: la copia solo ocurre para marcas en el mismo paso.

`remeter_song` además calcula todos los patrones nuevos a partir del proyecto sin tocar, se guarda un clon de la lista de reproducción vieja, y solo entonces escribe. Leer el proyecto mientras se escribe necesitaría dos préstamos del mismo `Project` a la vez, uno compartido y otro mutable, cosa que el comprobador de préstamos rechaza con razón. Y se ejecuta dentro de `project.moving_the_band(...)`: las notas de la banda se mueven con todo lo demás, y eso no es una edición tuya, así que la banda sigue la hoja de acordes (La banda).

No perder el sitio

Mientras la canción suena estás en algún lugar, compás 13, pulso 2, por ejemplo. Después del apretón, el paso 196 ya no está en el compás 13. Así que después de cualquier cambio de compás la aplicación busca el compás en que estaba el cabezal, y cuánto había avanzado en él, y va al mismo compás y a la misma fracción:

```
let place = |step: usize| {
    let bar = old.bar_at(step);
    let into = (step - bar.start) as f64 / bar.length.max(1) as f64;
    let target = new.bar(bar.number);
    target.start + (into * target.length as f64).round() as usize
};
```

`old` y `new` son los `played_meter` de antes y de después, así que las repeticiones cuentan. El mismo closure mueve el lugar desde donde toca ▶ y al que vuelve ■. Funciona también después de deshacer y rehacer, que pueden traer una canción con marcas de compás propias, donde un `Remap` no encajaría.

Swing

El swing toca las dos corcheas de cada pulso larga-corta. El proyecto tiene una cantidad, de 0 («sin swing») a `MAX_SWING`, 100 (los dos tercios y el tercio de un tresillo):

```
pub swing: u8,
```

Las notas de los patrones nunca se mueven. En cambio, todo lo que *toca* la canción pide las notas tal como se tocan:

```
pub fn played_notes(&self, pattern: usize, channel: usize) -> Cow<'_,
[NoteEvent]> {
    let Some(written) = self.pattern(pattern).map(|p| p.notes(channel))
else {
        return Cow::Borrowed(&[]);
    };
    match self.pattern_swing(pattern) {
        0 => Cow::Borrowed(written),
        swing => Cow::Owned(swung(written, swing,
self.plays_hits(channel))),
    }
}
```

`Cow` («clone on write», de `std::borrow`) es o bien **prestado** o bien **propio**. La mayoría de las canciones no tienen swing, y para ellas `played_notes` devuelve las notas del propio patrón sin copiar ninguna. Solo un patrón con swing recibe un `Vec` nuevo. Los dos se leen igual, como un slice, y quien necesita un `Vec` propio llama a `.into_owned()`, que solo copia si era prestado. El `'_` dice que el caso prestado toma prestado de `self`: el resultado no puede vivir más que el proyecto.

El hilo de audio, la exportación MIDI y el acompañamiento que sigue lo que tocas usan `played_notes`. El piano roll, los botones de paso y la hoja de acordes leen las notas escritas, así que la cuadrícula que editas sigue siendo una cuadrícula.

Mover la segunda corchea

Un pulso son 96 ticks. La segunda corchea empieza en el tick 48; la tercera nota de un tresillo, en el tick 64. Así que el swing completo llega 16 ticks tarde:

```
const BEAT: usize = 4 * TICKS_PER_STEP;
const OFFBEAT: usize = BEAT / 2;
const FULL_DELAY: usize = BEAT * 2 / 3 - OFFBEAT;
const NEAR: usize = BEAT / 16;

pub fn delay(swing: u8) -> usize {
    let swing = usize::from(swing.min(MAX_SWING));
    (FULL_DELAY * swing + usize::from(MAX_SWING) / 2) /
usize::from(MAX_SWING)
}
```

Sumar la mitad de 100 antes de dividir por 100 vuelve a redondear al tick más cercano: 50 son 8 ticks, un swing de 7:5; 70 son 11,2 y quedan en 11.

`swung` mueve cada comienzo y cada final que cae en un contratiempo o cerca de él:

```
pub fn swung(notes: &[NoteEvent], swing: u8, hits: bool) -> Vec<NoteEvent>
{
    let late = delay(swing);
    let played = |tick: usize| match tick % BEAT {
        offset if offset.abs_diff(OFFBEAT) < NEAR => tick + late,
        _ => tick,
    };
    notes
        .iter()
        .map(|note| {
            let start = played(note.start_tick());
            let end = match hits {
                true => note.end_tick(),
                false => played(note.end_tick()),
            };
            NoteEvent {
                velocity: note.velocity,
                ..NoteEvent::exact(note.pitch, start,
end.saturating_sub(start))
            }
        })
        .collect()
}
```

El primer brazo es una **guarda de match**: `offset` es una variable nueva que encaja con cualquier cosa, y el `if` que la sigue decide si el brazo se aplica. «Cerca» es menos de seis ticks a cada lado, porque las cuerdas de un acorde rasgueado empiezan con un tick de diferencia, 48, 49, 50, y todas tienen que moverse juntas. `abs_diff` es la distancia entre dos números, sea cual sea el mayor; `offset - OFFBEAT` sobre un `usize` menor que 48 se desbordaría hacia abajo y entraría en pánico en una compilación de depuración.

Un patrón de constante es distinto de un enlace a variable. Si el brazo fuera `OFFBEAT => ...`, compararía con el valor de la constante, 48. Si fuera un `offbeat => ...` en minúsculas, Rust lo leería como una variable nueva que encaja con *todo*, y el brazo `_` no se ejecutaría nunca. El compilador avisa de eso, y la convención de mayúsculas para las constantes mantiene separados los dos casos.

Una nota que termina en el contratiempo también termina más tarde, así que la primera corchea de verdad es larga, ligada a la segunda. Los golpes de batería son distintos: un tambor muestreado suena hasta el final por mucho que dure la nota, así que solo se mueven sus comienzos (`hits`). La nota nueva se construye con `NoteEvent::exact`, la misma función que usa la importación MIDI para las notas fuera de la cuadrícula: una corchea con swing en el paso 2 pasa a ser el paso 2 con un `nudge` de 16 ticks, que el hilo de audio ya sabe tocar tarde (El secuenciador). Las notas que no están en un contratiempo (semicorcheas, una corchea con puntillo y una semicorchea) quedan como están.

Dónde se aplica el swing

```
pub fn swings_in(time: TimeSignature) -> bool {
    time.unit <= 4
}
```

Solo las indicaciones de compás que se cuentan en negras o en notas más largas tienen pares de corcheas en un pulso. En 6/8 o 12/8 un pulso ya son tres corcheas, y en 5/8 o 7/8 la corchea es el pulso. `pattern_swing` devuelve 0 para un patrón en un compás así, y el control Swing aparece en gris cuando ninguna parte de la canción puede tener swing, con una pista que dice por qué. Una sección de la banda tocada en otro estilo puede traer el swing de ese estilo (`section_swing`).

Ningún comando nuevo para el hilo de audio

El hilo de audio se entera de cada edición por `diff`, que compara el proyecto antes y después, y compara las notas *tocadas* (Decirle al hilo de audio lo que ha cambiado). Así que cambiar el swing no necesita un comando propio: los patrones cuyas notas tocadas se movieron se envían de nuevo, y los demás se quedan como están. Un patrón que pasa a 6/8, y deja de tener swing, se detecta igual, y lo mismo deshacer, que pone en su sitio un proyecto anterior entero. Cuando nada tiene swing, los dos lados de la comparación son `Cow` prestados, y compararlos compara los slices sin copiar nada.

Recordado por estilo

Cada ritmo de la Biblioteca lleva su propio swing (`Rhythm::swing`), y un estilo toma el de su ritmo: Swing, Jazz ballad, Jazz waltz, Shuffle y Five-four swing a 100, Rock and roll a 50, el resto sin swing. Sus patrones de batería están escritos en corcheas iguales, como los escriben las partituras, y el swing hace el resto. Cuando cambias la cantidad, la aplicación la recuerda para ese estilo, entre una ejecución y otra, en `app/swing.rs`:

```
#[derive(Debug, Default)]
pub(super) struct SwingMemory {
    amounts: BTreeMap<String, u8>,
    /// Whether an amount changed since they were last stored.
    changed: bool,
}

/// What is stored: a newtype, so it has a type of its own in egui's
/// store.
#[derive(Debug, Clone, Default, Serialize, Deserialize)]
#[serde(default)]
struct Remembered(BTreeMap<String, u8>);
```

egui guarda los valores persistentes por un `Id` *y* un tipo. Envolver el mapa en un struct propio, un **newtype**, le da un tipo que nadie más usa, así que un `BTreeMap<String, u8>` guardado por otra parte de la aplicación nunca podría leerse como este. Un `BTreeMap` mantiene sus claves ordenadas, así que el archivo de ajustes enumera los estilos siempre en el mismo orden. `store_swings` se ejecuta una vez por fotograma y solo escribe cuando `changed` está activo.

Elegir un estilo le da primero a la canción el swing propio del estilo, en `lineup-core`, que no sabe nada de ajustes; luego la aplicación pone en su lugar la cantidad recordada, en el mismo paso de deshacer.

Soltura

Un buen músico nunca está exactamente en la cuadrícula. **Soltura**, en la barra de herramientas de la lista de reproducción, fija cuánto pueden desviarse las notas, de *no* a 40 ms (`MAX_LOOSE_TIMING`). Se guarda en el proyecto como `loose_timing`, en milisegundos, y el hilo de audio la convierte en frames una sola vez, cuando llega `Command::SetLooseTiming`.

El retraso de cada nota se calcula cuando empieza su paso:

```
fn delay(loose_frames: f64, channel: usize, song_step: usize, pitch: Note)
-> f64 {
    let onset = ((song_step as u64) << 8) | channel as u64;
    let together = scatter(onset);
    let apart = scatter(onset ^ (u64::from(pitch.midi()) << 40));
    loose_frames * (0.8 * together + 0.2 * apart)
}
```

Un pianista se adelanta o se atrasa más o menos lo mismo en todas las notas de un acorde, así que cuatro quintos del retraso son comunes a todas las notas que un canal empieza en un paso, y un quinto cambia de nota a nota. Los números salen de `scatter`, que mezcla los bits de una semilla con la función SplitMix64:

```
fn scatter(seed: u64) -> f64 {
    let mut x = seed.wrapping_add(0x9E37_79B9_7F4A_7C15);
    x = (x ^ (x >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
    x = (x ^ (x >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
    x ^= x >> 31;
    // The top 53 bits, as many as an `f64` holds exactly.
    (x >> 11) as f64 / (1u64 << 53) as f64
}
```

Parece aleatorio, pero la misma semilla da siempre el mismo número, así que una canción suena igual cada vez y una exportación WAV suena como la reproducción. `wrapping_mul` deja que la multiplicación dé la vuelta más allá de `u64::MAX` en vez de entrar en pánico en una compilación de depuración: aquí dar la vuelta es justo lo que se busca. `scatter` tampoco necesita estado, lo que le viene bien al hilo de audio: ningún generador aleatorio que guardar, bloquear o sembrar.

Una nota retrasada espera en una lista hasta su frame:

```
pub(crate) struct Pending {
    channel: usize,
    origin: Origin,
    pitch: Note,
    velocity: u8,
    gate: u32,
    /// The frame it starts on, counted on [`AudioThread::now`]'s clock.
    at: u64,
    /// How many frames after its step that is, for a trace.
    late: u32,
}
```

La lista se crea con sitio para 512 notas (`PENDING_NOTES`) en el hilo de la interfaz, y el hilo de audio nunca la deja crecer:

```
// A full list never grows (that would allocate): the note
// then starts on time.
if late > 0 && pending.len() < pending.capacity() {
```

En cada frame se empiezan las notas cuyo `at` ha llegado, y se sacan con `swap_remove`, que pone la última nota en el hueco en vez de desplazar toda la lista. El orden de las notas que esperan no importa, así que la forma más barata de quitarlas es la correcta.

¿Por qué solo tarde, nunca antes? Para empezar una nota antes, el hilo de audio tendría que mirar el paso siguiente antes de que empiece. Tarde entre 0 y 20 ms suena igual que antes o después 10 ms alrededor de una cuadrícula que va ella misma 10 ms tarde, y nadie oye una canción entera 10 ms tarde.

En archivos y en archivos MIDI

Cada una de estas funciones se guarda solo cuando se usa, así que un proyecto que no la usa todavía lo pueden abrir versiones anteriores (Guardar). `format_for` escribe el formato más antiguo que puede contener el proyecto:

Formato	Hace falta para
3	Una indicación de compás distinta de 4/4, o un patrón con la suya propia
5	Marcas de compás
7	Marcas de tempo, o soltura
10	Notas fuera de la cuadrícula, como las de una canción pasada a otro compás
11	Una indicación de compás agrupada, como 2+2+3/8
16	Swing

Al leer un archivo, cada marca de compás y de tempo pasa por `set_time_marker` o `set_tempo_marker`, que rechazan una que empieza demasiado tarde o cuyos compases no cabrían en un patrón, como si se hubiera añadido a mano.

Un Standard MIDI File guarda el tempo y el compás como *metaeventos* en su primera pista. Un tempo se escribe como microsegundos por pulso, en tres bytes, y un cambio recibe un evento en el paso donde cambia el tempo tocado, así que un cambio gradual se convierte en un evento por paso, que es como los archivos MIDI escriben un ritardando:

```
let tempo_event = |tempo: Tempo| {
    let micros = (60_000_000.0 / f64::from(tempo.bpm())).round() as u32;
    micros.min(0xFF_FFFF).to_be_bytes()[1..].to_vec()
};
// ...
for (step, pair) in tempos.windows(2).enumerate() {
    if pair[1] != pair[0] {
        let tick = (step as u32 + 1) * TICKS_PER_STEP;
        events.push((tick, TEMPO, tempo_event(pair[1])));
    }
}
```

`to_be_bytes` da los cuatro bytes del `u32`, el más significativo primero («big-endian», el orden que usan los archivos MIDI), y `[1..]` quita el primero, que siempre es 0 por debajo de `0xFF_FFFF`.

Un evento de indicación de compás lleva el número de arriba, el de abajo como potencia de dos (`unit.ilog2()`: 2 para una negra, 3 para una corchea), cuántos relojes MIDI dura un clic de metrónomo (24 por negra, así que 6 por paso), y un 8 fijo. Hay uno al principio y uno en cada cambio del compás tocado. MIDI no tiene sitio para una agrupación, así que un compás agrupado marca cada nota (`ungrouped().first_beat()`). Cómo se escribe un archivo MIDI entero está en MIDI.

Cómo se prueba

Cada tipo tiene a su lado tests que se leen como las reglas que comprueban:

- `meter.rs`: `a_steady_meter_counts_like_its_time_signature`, `a_marker_off_the_bar_line_leaves_a_short_bar`, `the_usual_grouping_is_written_plainly` (3+3/8 es 6/8), `a_grouping_is_saved_only_when_there_is_one`, `nonsense_does_not_divide_by_zero`.
- `tempo.rs`: `a_glide_moves_a_little_every_step` (el 90, 80, 70, 60 de más arriba), `a_glide_starts_from_where_the_last_one_got_to` (un ritardando cortado por un *a tempo*), `seconds_add_up_every_step`.
- `remeter.rs`: `every_note_keeps_its_place_in_the_bar`, `notes_between_steps_keep_their_exact_place`, `every_bar_lasts_as_long_as_before` (la duración de la canción en segundos, antes y después, con menos de un 0,2 % de diferencia), `a_tempo_put_in_another_meter_and_back_is_the_same_tempo`, `a_pattern_too_long_for_the_new_bars_is_cut`.

- `swing.rs`: `full_swing_plays_the_eighths_as_triplets`, `half_swing_is_seven_to_five`, `a_strum_on_the_offbeat_moves_as_a_whole`, `notes_off_the_eighths_stay`.
- `engine.rs` toca unas pocas notas y comprueba dónde empieza el sonido, frame a frame: `a_song_plays_each_step_at_its_own_tempo` (dos pasos a 120 BPM, luego dos a 60, que duran el doble) y `loose_timing_starts_notes_a_little_late` (cada nota dentro de los 960 frames, 20 ms a 48 kHz, después de su paso; no todas con el mismo retraso; y lo mismo cada vez).

Un test recorre la aplicación entera: `styles_tried_one_after_another_keep_the_songs_tempo` abre El Choclo y lo pasa por siete estilos y compases (entre ellos un tango vals en 3/4 y el cuadro Compás en 7/8), en modo patrón, en modo canción parado y en modo canción sonando. Después de cada uno, el tempo tiene que seguir igual cuando el compás no cambia, y ser el de la canción cada vez que vuelve a 4/4. Luego toca el principio de la canción con la traza de reproducción y comprueba que cada nota de la melodía empieza y dura lo mismo que antes, con menos de 2 ms de diferencia. Para que sea rápido en una compilación de depuración, solo toca doce segundos, a 8000 muestras por segundo: de sobra para medir notas.

Instrumentos: batería, sampler y Lineup Osc

Archivos: `crates/lineup-audio/src/instrument.rs`, `crates/lineup-audio/src/drums.rs`, `crates/lineup-audio/src/dsp.rs`, `crates/lineup-audio/src/sample.rs` y `crates/lineup-audio/src/timbre.rs`

Cada canal tiene un instrumento: la parte que convierte las notas en sonido. Este capítulo trata de los dos que Ferrous Lineup hace por sí mismo, el **sampler**, que toca un sonido grabado o sintetizado, y el **Lineup Osc**, un sintetizador pequeño. Los otros dos tipos tocan a través de otra cosa: una SoundFont (SoundFonts y General MIDI) o un sintetizador externo por MIDI (MIDI). Cómo se eligen los sonidos en la aplicación está en Sonidos, del manual.

Un enum, cuatro instrumentos

```
pub enum Instrument {
    Sampler(Sampler),
    Synth(Synth),
    SoundFont(SoundFontPlayer),
    MidiOut(MidiOut),
}
```

Un diseño habitual para «muchas clases de una misma cosa» es un trait, `trait Instrument`, con valores de tipo `Box<dyn Instrument>`. Es abierto: cualquiera podría añadir una clase sin tocar este archivo. Pero cada `Box::new` reserva memoria, y cada llamada pasa por un puntero a una tabla de funciones. Con cuatro clases, todas conocidas, un enum es más sencillo. El hilo de audio hace `match` sobre él, crear uno nunca reserva memoria, y cuando se añade una quinta clase el compilador enumera cada `match` que tiene que enterarse.

Cada método de `Instrument` es un `match` que pasa la llamada a la clase de dentro:

```
pub fn release_all(&mut self) {
    match self {
        Instrument::Sampler(sampler) => sampler.release_all(),
        Instrument::Synth(synth) => synth.release_all(),
        Instrument::SoundFont(player) => player.release_all(),
        Instrument::MidiOut(midi_out) => midi_out.release_all(),
    }
}
```

Lo que se le puede pedir a un instrumento:

Método	Qué hace
`note_on`	Empieza una nota: una altura, una velocidad y un **gate**, cuántos frames se sostiene
`note_on_from`	Lo mismo, para una nota de la canción, que recuerda de dónde viene (`Origin` , ver El secuenciador)
`note_off`	Suelta una nota sostenida con el gate `HELD`
`release_all`	Suelta todas las notas sostenidas, cuando se detiene la reproducción
`release_unless`	Suelta las notas de la canción que la canción ya no toca
`resume`	Empieza un sonido largo por la mitad, donde estaría si la reproducción hubiera empezado antes (solo el sampler)
`tempo_changed`	Estira las notas sostenidas después de un cambio de tempo (Tempo, compás, swing y sensación)
`next_sample`	El siguiente frame de sonido: la suma de todas las voces

Una nota de la canción sabe su longitud cuando empieza, así que recibe un gate en frames y termina sola; el hilo de audio nunca tiene que acordarse de pararla. Una tecla pulsada en un teclado MIDI no tiene una longitud conocida, así que recibe `HELD` , `u32::MAX` , que nunca se acaba: solo `note_off` la termina.

El instrumento de un canal lo construye `Instrument::for_channel` , en el hilo de la interfaz, a partir del `Channel` del proyecto:

```
Ok(match &channel.kind {
    // A drum sent over MIDI is its note of a General MIDI drum set.
    ChannelKind::Sampler(SampleSource::Builtin(drum)) => match
channel.sent_on {
        Some(number) => Instrument::MidiOut(

MidiOut::new(number).playing_as(Some(general_midi_drum(*drum))),
        ),
        None => {
            Instrument::Sampler(Sampler::new(Arc::new(drums::render(*drum,
sample_rate))))
        }
    },
    ChannelKind::Sampler(SampleSource::File(path)) => {
        Instrument::Sampler(Sampler::new(samples.get(path)?))
    }
    ChannelKind::Synth => Instrument::Synth(Synth::with_timbre(
        sample_rate,
        Timbre::for_program(channel.program),
    )),
    ChannelKind::MidiOut { channel: number } => {

Instrument::MidiOut(MidiOut::new(*number).with_program(channel.program))
    }
})
```

Aquí se sintetizan los tambores y se leen los archivos (a través de la `SampleCache`), un trabajo lento que puede reservar memoria: justo lo que el hilo de audio no debe hacer. El instrumento terminado se envía en un comando.

Una batería hecha con unas líneas de matemáticas

Ferrous Lineup no trae archivos de sonido. Sus trece tambores se calculan cuando se crea un canal de batería, cada uno en unas pocas líneas:

Tambor	Receta
Bombo	Una senoide cuya altura cae deprisa de 150 a 45 Hz, apagándose en medio segundo
Caja	Un tono corto de 185 Hz (el casco) más ruido con paso alto (la bordonera)
Palmas	Ruido con paso alto en tres ráfagas separadas 11 ms, como varias manos, y una cola
Charles	Ruido muy brillante, con paso alto a 7 kHz, que se va en una décima de segundo

Tambor	Receta
Charles abierto, ride, crash	«Metal» (ver abajo) y ruido, que resuenan cada vez más
Aro	Un clic de 1750 Hz, un cuerpo de 480 Hz y un chasquido de ruido
Tom grave y agudo	Como el bombo, más agudos y redondos, asentándose en 98 y 165 Hz
Cencerro	Dos ondas cuadradas a 540 y 800 Hz, a un poco menos de una quinta
Shaker	Ruido brillante que crece durante 15 ms y luego se apaga deprisa
Conga	Un tono de 320 Hz que baja un poco, y el golpe de la mano

Casi todos comparten una forma, el **decaimiento exponencial**:

```
fn decay(t: f32, tau: f32) -> f32 {
    (-t / tau).exp()
}
```

Vale 1 al principio y cae a un 37 % después de `tau` segundos, a un 13 % después del doble, y así. Las cosas golpeadas se apagan así.

Closures que recuerdan: `FnMut`

Todos los tambores se construyen con una sola función auxiliar:

```
fn generate(seconds: f32, sample_rate: u32, f: impl FnMut(f32) -> f32) ->
Vec<f32> {
    let frames = (seconds * sample_rate as f32) as usize;
    (0..frames)
        .map(|i| i as f32 / sample_rate as f32)
        .map(f)
        .collect()
}
```

Recibe un closure `f` que va de un tiempo en segundos a una muestra. El closure del bombo guarda una `phase` que avanza entre llamadas:

```
fn kick(sample_rate: u32) -> Vec<f32> {
    let mut phase = 0.0;
    generate(0.5, sample_rate, |t| {
        let frequency = 45.0 + 105.0 * decay(t, 0.04);
        phase += frequency / sample_rate as f32;
        (phase * TAU).sin() * decay(t, 0.15)
    })
}
```

El closure *cambia* `phase`, una variable que capturó de `kick`. Un closure que cambia lo que captura implementa el trait `FnMut`; `Fn` es para los closures que solo leen, y `FnOnce` para los que solo se pueden llamar una vez. `impl FnMut(f32) -> f32` en la lista de parámetros significa «cualquier closure con esa forma», y el compilador hace una copia de `generate` para el closure de cada tambor, como hace con los tipos genéricos.

¿Por qué ir sumando una fase en vez de calcular `sin(t × frecuencia × 2π)`? Porque la frecuencia cambia con el tiempo. La fase es la suma de todos los pasitos dados hasta ahora, así que la altura se desliza sin saltos; la fórmula directa saltaría de un lado a otro al cambiar la frecuencia.

Metal con seis ondas cuadradas

Los platillos usan un truco de las cajas de ritmos analógicas de principios de los 80: seis ondas cuadradas a frecuencias que no son múltiplos unas de otras, sumadas.

```
const METAL: [f32; 6] = [205.3, 304.4, 369.6, 522.7, 540.0, 800.0];

fn metal(t: f32, ratio: f32) -> f32 {
    METAL.iter().map(|&f| square(t, f * ratio)).sum::<f32>() / METAL.len()
as f32
}
```

Las notas cuyas frecuencias son múltiplos sencillos unas de otras se funden en un acorde. Estas no, así que juntas chocan y brillan como metal. Un filtro de paso alto se queda con la parte brillante, y la envolvente decide si es un charles cerrado, uno abierto, un ride o un crash.

Después, cada tambor se escala con `normalize` para que su muestra más fuerte sea 0,9, y dos tests recorren `Drum::ALL`: cada tambor llega a ese pico y dura más de mil frames (`every_drum_is_audible_and_in_range`), y cada uno termina en silencio en vez de cortarse (`every_drum_fades_out`).

Ruido y filtros

`dsp.rs` guarda las piezas pequeñas. El **ruido blanco** sale de xorshift, un generador de números aleatorios diminuto:

```
pub fn next_sample(&mut self) -> f32 {
    let mut x = self.state;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    self.state = x;
    (x as f32 / u32::MAX as f32) * 2.0 - 1.0
}
```

Tres desplazamientos y tres o exclusivos. No sirve para seguridad, pero suena igual que el ruido, nunca reserva memoria, no necesita ningún crate, y da los mismos tambores «aleatorios» cada vez, lo que permite probarlos. Su única trampa es el cero: xorshift desde 0 se queda en 0 para siempre, así que `Noise::new` empieza desde `seed.max(1)`.

Los filtros son de **un polo**, los más sencillos que hay. El de paso bajo mueve su salida una fracción del camino hacia la entrada en cada muestra, así que los cambios rápidos (las frecuencias altas) se suavizan:

```
pub fn process(&mut self, input: f32) -> f32 {
    self.state += (input - self.state) * self.coefficient;
    self.state
}
```

Es la misma aritmética que `Smoothed`, el volumen que se desliza en vez de saltar (Dos hilos). El de paso alto sigue en cambio los *cambios* de su entrada, así que una señal constante se apaga y los cambios rápidos pasan. Los tests dicen exactamente eso: una entrada constante atraviesa el paso bajo y el paso alto la elimina.

El sampler

```
pub struct Sampler {
    sample: Arc<SampleData>,
    /// Boxed so an `Instrument` stays small: commands carrying one are
    /// copied into the command queue, and a pointer is cheaper to copy
    /// than eight voices. The box is created here, on the UI thread.
    voices: Box<[SamplerVoice; SAMPLER_VOICES]>,
    notes_started: u64,
}
```

`Arc<T>` («atomically reference-counted») es un puntero que comparten varios dueños. Clonarlo no copia los datos, suma uno a un contador, y los datos se liberan cuando se va el último `Arc`. El contador se actualiza de forma atómica, así que es seguro entre hilos, y eso es lo que permite construir una muestra en el hilo de la interfaz y tocarla en el hilo de audio. El mismo archivo de sonido en tres canales está una sola vez en memoria.

Cada una de las ocho voces toca un golpe:

```
fn next_sample(&mut self, frames: &[f32]) -> f32 {
    let Some(position) = self.position else {
        return 0.0;
    };
    let index = position as usize;
    if index + 1 >= frames.len() {
        self.position = None;
        return frames.get(index).copied().unwrap_or(0.0) * self.gain;
    }
    // Linear interpolation between the two nearest frames.
    let fraction = (position - index as f64) as f32;
    let sample = frames[index] + (frames[index + 1] - frames[index]) *
fraction;
    self.position = Some(position + self.increment);
    // ...
}
```

`let ... else` encaja un patrón, o ejecuta el bloque `else`, que tiene que salir de la función. Mantiene el camino principal sin sangría: «si no suena, silencio; si no, sigue con `position`».

`position` es una fracción de frame, porque un sonido rara vez recorre sus frames exactamente a uno por frame de salida. Un WAV grabado a 44,1 kHz y tocado en una tarjeta de sonido de 48 kHz tiene que avanzar un poco más despacio, o sonaría más agudo y más corto. Y un sampler toca alturas tocando más deprisa o más despacio: una octava arriba es el doble de rápido, y cada semitono multiplica la velocidad por la

raíz duodécima de dos, la misma regla que usa `Note::frequency` (Clases de altura, notas y su escritura):

```
// Every semitone up plays the sample 2^(1/12) times faster.
let semitones = pitch.midi() as f64 - STEP_NOTE.midi() as f64;
let pitch_ratio = 2.0_f64.powf(semitones / 12.0);
let rate_ratio = self.sample.sample_rate() as f64 / output_rate as f64;
```

El `increment` de la voz es el producto de las dos razones. En `STEP_NOTE`, Do4, la nota que toca un botón de paso, la razón de altura es 1 y el sonido se toca como se grabó. Entre dos frames guardados la voz traza una recta, **interpolación lineal**: no es lo mejor que puede hacer un sampler, pero es barata y basta para la batería.

La velocidad escala el volumen, `velocity / 127`. Un sampler ignora el gate: como una caja de ritmos, toca su sonido entero cada vez, y como tiene voces, un golpe nuevo no corta el anterior.

Eso está bien para la batería y mal para una cosa: una grabación larga, como una canción entera importada como audio, seguiría sonando después de Detener. Así que `release_all` apaga en unos 15 ms las voces de un sonido de más de diez segundos, y deja resonar los golpes cortos:

```
pub fn release_all(&mut self) {
    let long = self.sample.frames().len() as u64
        > u64::from(self.sample.sample_rate()) * LONG_SAMPLE_SECONDS;
    if long {
        for voice in self.voices.iter_mut().filter(|voice|
!voice.is_idle()) {
            voice.fading.get_or_insert(1.0);
        }
    }
}
```

`release_unless`, para un clip que se quita o una pista que se silencia mientras suena, apaga los sonidos de la canción sea cual sea su longitud.

Empezar por la mitad

Una nota suena cuando la reproducción llega al paso en que empieza. Para un golpe de batería eso es todo, pero una grabación es una nota en el primer paso de una canción: empezada desde el compás 20, la canción sonaría sin ella. Así que cuando la reproducción empieza, salta o vuelve al comienzo del bucle, `resume_long_sounds`, en el motor, mira hacia atrás en la canción las notas que empezaron antes del cursor de reproducción, y le pide a cada instrumento que las retome con `resume`:

```
fn resume(
    &mut self,
    pitch: Note,
    velocity: u8,
    output_rate: u32,
    origin: Origin,
    elapsed: f64,
) -> bool {
    let increment = self.increment(pitch, output_rate);
    let position = elapsed * increment;
    if !self.is_long() || position + 1.0 >= self.sample.frames().len() as
f64 {
        return false;
    }
    self.start_at(position, increment, velocity, Some(origin));
    true
}
```

`elapsed` es cuántos frames de salida hace que empezó la nota, y una voz avanza `increment` frames del sonido por cada frame de salida, así que la voz empieza `elapsed * increment` frames adentro, tan lejos como habría llegado. Solo un sonido largo, de más de diez segundos, empieza así, y solo si dura tanto: un golpe de batería de hace dos compases ya terminó hace rato, y los demás instrumentos responden `false`. `resume_long_sounds` recorre cada trozo de un clip que empieza en el paso o antes, y cada vez que el trozo tocó cada nota:

```
for note in channel.notes.get(play.pattern) {
    // Every time the piece played the note before
    // `step`, counted from the piece's start, which is
    // below zero for a time in the part of the clip the
    // piece skips.
    let first = play.start as isize - play.skip as isize + note.start as
isize;
    let times = (first..play.end().min(step) as isize).step_by(length);
    for at in times.filter(|&at| at < step as isize) {
        let origin = Origin {
            pattern: play.pattern,
            pattern_step: note.start,
            song_step: at.max(0) as usize,
            over_song: false,
        };
        let elapsed = frames_between(timing, at, step);
        let started = channel.instrument.resume(
            origin,
            note.pitch,
            note.velocity,
            self.sample_rate,
            elapsed,
        );
        // ...
    }
}
```

`first` es donde cae la nota si el trozo empezara en su propio primer paso; es menor que cero cuando el clip deja fuera su comienzo (`skip`), y `step_by(length)` da cada repetición del patrón dentro del trozo. `frames_between` suma los frames de los pasos desde ahí hasta el cursor de reproducción, al tempo de la canción en cada paso, así que una grabación bajo un cambio de tempo también empieza en el lugar justo.

Cargar un archivo WAV

`SampleData::load_wav` lee archivos WAV con el crate `hound`. Un WAV guarda las muestras como números de coma flotante o como enteros de 8 a 32 bits, y los dos se convierten en flotantes entre -1 y 1:

```
let interleaved: Vec<f32> = match spec.sample_format {
    hound::SampleFormat::Float => reader
        .samples::<f32>()
        .collect::<Result<_, _>>()
        .map_err(SampleError::Wav)?,
    hound::SampleFormat::Int => {
        let full_scale = (1_i64 << (spec.bits_per_sample - 1)) as f32;
        reader
            .samples::<i32>()
            .map(|sample| sample.map(|s| s as f32 / full_scale))
            .collect::<Result<_, _>>()
            .map_err(SampleError::Wav)?
    }
};
```

`samples()` da un `Result` por muestra, porque un archivo puede estar cortado a la mitad. Recoger un iterador de `Result` en un `Result<Vec<_>, _>` da `Ok` con todos los valores si todos eran `Ok`, o el primer `Err`, y se detiene ahí. Los `_` dejan que el compilador complete los tipos. Después, `interleaved.chunks(channels)` toma un frame cada vez (izquierdo, derecho) y lo promedia, así que un archivo estéreo queda en mono.

`SampleError` implementa `Display` con mensajes traducidos, así que «no se pudo leer el archivo WAV» le llega al usuario en su idioma. El test `loads_and_mixes_to_mono` escribe un WAV estéreo real de 16 bits en la carpeta temporal del sistema con `hound::WavWriter` y lo vuelve a leer.

Voces

Una **voz** es lo necesario para hacer sonar una nota: su propio punto en el sonido, su propio oscilador, su propia envolvente. El Lineup Osc tiene 16 voces y un sampler 8. Cada nota nueva toma una:

```rust
trait Voice {
    /// Whether the voice is silent and free to play a new note.
    fn is_idle(&self) -> bool;
    /// When the voice's note started, as a count of notes started so far.
    fn started(&self) -> u64;
    /// Whether the voice's note has ended and it is only fading out.
    fn is_releasing(&self) -> bool;
}

fn pick_voice<V: Voice>(voices: &mut [V]) -> &mut V {
    let oldest = |releasing: bool| {
        voices
            .iter()
            .enumerate()
            .filter(|(_, voice)| !releasing || voice.is_releasing())
            .min_by_key(|(_, voice)| voice.started())
            .map(|(index, _)| index)
    };
    let index = voices
        .iter()
        .position(Voice::is_idle)
        .or_else(|| oldest(true))
        .or_else(|| oldest(false))
        .unwrap_or(0);
    &mut voices[index]
}
```

El orden es: una voz en silencio si la hay; si no, la voz más antigua de las que solo se están apagando; si no, la más antigua de todas. Tomar una voz ocupada es **robar una voz**, y todo sintetizador de hardware lo hace cuando se sostienen más notas de las voces que tiene. Una voz que se apaga va primero porque cortar una cola se nota mucho menos que cortar una nota todavía sostenida, como un bajo largo debajo de acordes rápidos (el test `voices_that_are_fading_out_are_stolen_first` toca justo eso).

`Voice` es un trait privado: `SamplerVoice` y `SynthVoice` son structs distintos, los dos lo implementan, y una sola función genérica sirve para ambos. `V: Voice` es la restricción de trait; la función solo puede llamar a lo que el trait promete.

¿Por qué buscar primero un índice y solo después devolver `&mut voices[index]`? Las búsquedas toman prestado el slice para leerlo (`iter()`). Una referencia mutable no puede salir de una búsqueda de solo lectura, así que el código encuentra la posición, deja que terminen los préstamos de lectura, y toma prestado de forma mutable una vez, al final.

`started` sale de un contador que sube con cada nota, así que «la más antigua» es «la menor». Un `u64` no se desborda: a mil notas por segundo tardaría 584 millones de años.

Un número fijo de voces, en un Box

```
voices: Box::new(std::array::from_fn(|_| {
    SynthVoice::new(sample_rate, timbre.cutoff)
})),
```

Las voces son un array de tamaño fijo, así que el hilo de audio nunca lo hace crecer y nunca reserva memoria. `std::array::from_fn` construye un array llamando al closure una vez por elemento; el closure recibe el índice, que aquí no hace falta, de ahí `|_|`. Las voces del sampler implementan `Default`, así que las construye `Box::default()`.

El array va en un `Box` por la cola de comandos hacia el hilo de audio. Cada hueco de la cola es tan grande como el `Command` más grande, y un comando que inserta un canal lleva un `Instrument`. Dieciséis voces pasan del kilobyte; un `Box` es un puntero de 8 bytes. Un test fija el tamaño:

```
let size = std::mem::size_of::<Command>();
assert!(size <= 80, "a command takes {size} bytes");
```

Meterlas en un `Box` significa que las voces viven en el montón, y liberarlas es liberar memoria. Por eso un instrumento con el que el hilo de audio ha terminado vuelve siempre al hilo de la interfaz por la cola de basura (Dos hilos): el box se crea y se libera en el hilo de la interfaz, y el hilo de audio solo mueve el puntero.

El Lineup Osc

El Lineup Osc toca un oscilador por voz, a través de un filtro de paso bajo suave, moldeado por una **envolvente** de volumen:

```
pub struct Synth {
    sample_rate: f32,
    voices: Box<[SynthVoice; SYNTH_VOICES]>,
    /// Boxed, like the voices, so an `Instrument` stays small.
    envelope: Box<EnvelopeRates>,
    notes_started: u64,
}
```

La onda

Un oscilador guarda una **fase**, de 0 a 1 a lo largo de un ciclo, y la onda dice cómo suena cada fase:

```
pub fn at(self, phase: f32) -> f32 {
    match self {
        // Ramps from -1 to 1.
        Wave::Saw => phase * 2.0 - 1.0,
        // 1 for the first half of the cycle, -1 for the second.
        Wave::Square => {
            if phase < 0.5 {
                1.0
            } else {
                -1.0
            }
        }
        // Up from -1 to 1 and back down.
        Wave::Triangle => 1.0 - 4.0 * (phase - 0.5).abs(),
        Wave::Sine => (phase * std::f32::consts::TAU).sin(),
    }
}
```

En cada frame la fase avanza `frequency / sample_rate` y vuelve a empezar con `fract()`. Una diente de sierra tiene todos los armónicos, así que es brillante y zumbona; una cuadrada solo los impares, que suena hueco; una triangular los tiene débiles y suena suave; una senoide no tiene ninguno y suena pura.

Una diente de sierra hecha así contiene frecuencias por encima de lo que puede representar la frecuencia de muestreo, y vuelven plegadas como tonos metálicos débiles, lo que se llama **aliasing**. El filtro de paso bajo esconde la mayor parte. Quitarlo de verdad (con correcciones «PolyBLEP», o con tablas de onda) sería un buen proyecto.

La envolvente: una máquina de estados

Una nota que solo se enciende y se apaga hace clic, y no se parece a nada real. Una envolvente moldea su volumen a lo largo del tiempo. La clásica tiene cuatro partes, **ADSR**:

```
 volumen
   1 ┤  /\
     │ /  \__________          A: ataque, subiendo al volumen completo
     │/    D    S     \        D: decaimiento, bajando hasta...
     │                 \       S: sostenido, mientras la nota se sostiene
   0 ┼──────────────────\___   R: liberación, apagándose al terminar la
nota
       A    fin del gate ^
```

Cada voz está en una **etapa**, y cada muestra puede moverla a la siguiente. El código con esta forma es una **máquina de estados**:

```
match self.stage {
    Stage::Idle => return 0.0,
    Stage::Attack => {
        self.level += envelope.attack_step;
        if self.level >= 1.0 {
            self.level = 1.0;
            self.stage = Stage::Decay;
        }
    }
    Stage::Decay => {
        let sustain = envelope.sustain;
        self.level = sustain + (self.level - sustain) *
envelope.decay_factor;
    }
    Stage::Release => {
        self.level *= envelope.release_factor;
        if self.level < SILENT {
            self.stage = Stage::Idle;
            return 0.0;
        }
    }
}

// Count down the gate while the note is held, then let go. A
// `HELD` gate is not counted: only `note_off` ends it.
if self.is_held() && self.gate != HELD {
    match self.gate.checked_sub(1) {
        Some(left) => self.gate = left,
        None => self.stage = Stage::Release,
    }
}
```

- El ataque sube en línea recta; el decaimiento y la liberación son **exponenciales**: cada frame conserva una fracción fija de la distancia que queda, como se apagan las cuerdas y las campanas. Las fracciones se calculan una vez por instrumento en `EnvelopeRates::new`, no en cada muestra: `decay_factor` es `e^(-1 / (segundos × frecuencia de muestreo))`.

- `checked_sub` devuelve `None` en vez de bajar de cero, así que se lee como «si no queda gate, suelta».
- No hay etapa `Sustain`. El decaimiento se acerca al nivel de sostenido y se queda ahí, así que una etapa propia no haría nada.
- `SILENT` es 0,0001, unos -80 dB. Un apagado exponencial nunca llega a cero, así que por debajo de ese valor la voz se da por callada y se libera.

`note_off` suelta las voces de una altura solo si se empezaron con el gate `HELD`. Una tecla que se suelta en un teclado MIDI no debe terminar la misma nota que la canción toca con su propio gate. `release_all` pasa todas las voces sostenidas a `Release`, que es como Detener deja que las notas largas se apaguen en vez de seguir sonando.

Una voz robada no se reinicia: `start` pone la altura, la ganancia y el gate nuevos, y deja el nivel, la fase y el filtro como estaban.

```
voice.stage = Stage::Attack;
voice.started = self.notes_started;
voice.origin = origin;
// The level, phase and filter carry on from where they were, so a
// stolen voice does not jump (and click).
```

El ataque sube entonces desde donde estuviera el nivel. Cada voz se baja además a `SYNTH_VOICE_GAIN`, 0,35, para que un acorde de cuatro notas no salga cuatro veces más fuerte.

Timbres: sonar como un instrumento General MIDI

El Lineup Osc nunca podrá sonar como un violín de verdad, pero puede sonar más a violín que a piano. Un canal con un programa General MIDI y sin SoundFont toca un **timbre** elegido según la familia del programa:

```rust
pub struct Timbre {
    pub wave: Wave,
    /// Seconds to reach full volume.
    pub attack: f32,
    /// How quickly the volume falls to the sustain level (time constant,
    /// in seconds).
    pub decay: f32,
    /// The volume a held note settles at, 0 to 1. 0 for a sound that dies
    /// away by itself, like a piano or a plucked string.
    pub sustain: f32,
    /// How quickly a note fades after it is let go (time constant).
    pub release: f32,
    /// The low-pass filter's cutoff, in Hz: lower is darker.
    pub cutoff: f32,
    /// How loud, compared with the Lineup Osc's own sound.
    pub gain: f32,
}
```

Dos pequeñas `const fn` dan nombre a las dos formas en que se comportan los instrumentos:

```rust
/// Struck or plucked: starts at once and dies away over `decay`.
const fn struck(wave: Wave, decay: f32, release: f32) -> Timbre {
    Timbre::new(wave, 0.003, decay, 0.0, release)
}

/// Bowed, blown or sung: swells in over `attack` and keeps going.
const fn held(wave: Wave, attack: f32, release: f32) -> Timbre {
    Timbre::new(wave, attack, 0.3, 0.85, release)
}
```

y `for_program` es un solo `match` sobre el número de programa, en el orden del propio General MIDI, donde cada familia es un rango de ocho números:

```rust
let timbre = match program.min(127) {
    // Pianos.
    0..=3 => Timbre::struck(Wave::Saw, 0.9, 0.25).cutoff(1_800.0),
    // ...
    // Brass.
    56..=63 => Timbre::held(Wave::Saw, 0.04, 0.1).cutoff(2_000.0),
    // Reeds: saxophones, oboe, bassoon, clarinet.
    64..=71 => Timbre::held(Wave::Square, 0.03, 0.08).cutoff(1_800.0),
    // Pipes: flutes, recorder, ocarina.
    72..=79 => Timbre::held(Wave::Triangle, 0.05, 0.1).cutoff(3_000.0),
    // ...
    // Sound effects have no tune to play; the Lineup Osc stands in.
    _ => Timbre::LINEUP_OSC,
};
```

Familia	Onda	Envolvente	Suena
Piano, guitarra, arpa, pizzicato	diente de sierra o triangular	empieza de golpe, se apaga aunque se sostenga	golpeado o pulsado
Cuerdas, metales	diente de sierra	entra creciendo, se mantiene	frotado o soplado
Órgano, acordeón, lengüetas	cuadrada	entera mientras se sostiene	hueco, de lengüeta
Flautas, voces	triangular	ataque suave, se mantiene	suave y redondo
Campanas, vibráfono, marimba	senoide	se apaga	puro

Los patrones de rango (`0..=3`), las alternativas (`24..=28 | 31`) y un `_` final hacen de toda la tabla una sola expresión, y el compilador comprueba que cada número de 0 a 255 cae en algún sitio. Al final, la ganancia de cada timbre se multiplica por la de su onda, para que una cuadrada, la onda más fuerte, no destaque en la mezcla. `Timbre::for_program(None)` es `LINEUP_OSC`, la diente de sierra brillante propia del sintetizador, así que un canal sin programa suena como siempre. Qué programa recibe cada canal está en SoundFonts y General MIDI.

Cómo se prueba

Los tests tocan un instrumento en un búfer y miran los números, ya que en un test nadie puede escuchar:

- `synth_plays_the_right_pitch` toca La4 durante un segundo y cuenta cuántas veces cambia de signo la salida: unas 880, dos por cada ciclo de 440 Hz.
- `synth_stops_after_the_gate_and_release` sostiene una nota una décima de segundo y comprueba que la salida es exactamente cero diez tiempos de liberación después, sin ninguna voz sonando.
- `synth_plays_chords`, `voices_are_stolen_when_all_are_busy` y `voices_that_are_fading_out_are_stolen_first` comprueban las voces.
- `sampler_resamples_to_the_output_rate` toca un segundo grabado a 24 kHz en una salida de 48 kHz y espera 48 000 frames de sonido, con un margen de diez; `sampler_plays_higher_notes_faster` espera la mitad una octava arriba; `sampler_hits_can_overlap` toca dos golpes a la vez.

- `held_notes_last_until_note_off` comprueba el gate `HELD`.
- En `timbre.rs`, `plucked_instruments_die_away_and_bowed_ones_keep_going` comprueba las familias, y `every_wave_stays_between_minus_one_and_one` las ondas.

SoundFonts y General MIDI

Archivos: `crates/lineup-core/src/general_midi.rs`, `crates/lineup-audio/src/soundfont.rs`, `crates/lineup-audio/src/instrument.rs` y `crates/lineup-app/src/app/sounds.rs`

Un canal dice qué instrumento toca con un número: 0 es un piano de cola, 40 un violín, 52 un coro. Los números vienen de **General MIDI**, una lista en la que están de acuerdo todos los sintetizadores. En Ferrous Lineup, tres cosas convierten ese número en sonido:

- el Lineup Osc elige una onda y una envolvente que se le acercan (Timbres);
- a un sintetizador externo se le pide por MIDI que lo toque, con un *cambio de programa* (MIDI);
- un **SoundFont**, un archivo de grabaciones de instrumentos reales, lo toca dentro de la aplicación.

Este capítulo trata de los números, de dónde saca uno cada canal, y del SoundFont. Cómo activar los sonidos y elegir instrumentos en la aplicación está en el capítulo Sonidos del manual.

General MIDI: 128 instrumentos por número

Un cable MIDI o un archivo MIDI llevan notas, no sonidos. Para decir qué sonido debe tocarlas, MIDI tiene el mensaje de cambio de programa: «a partir de ahora, el canal 3 toca el programa 40». Son dos bytes: `0xC2` (cambio de programa, `0xC0`, en el canal 3, que contado desde 0 es el 2) y `40`.

Por sí solo, 40 no significa nada. General MIDI, publicado en 1991, fijó una lista de 128 programas, para que el 40 sea un violín en cualquier sintetizador que la siga, desde un teléfono hasta un teclado de escenario. Los programas vienen en dieciséis familias de ocho: pianos del 0 al 7, percusión cromática (celesta, vibráfono, marimba) del 8 al 15, órganos del 16 al 23, guitarras del 24 al 31, bajos del 32 al 39, cuerdas del 40 al 47, y así hasta los efectos de sonido, del 120 al 127. El canal MIDI 10 está reservado a la batería: allí cada *nota* es un tambor, el 36 el bombo, el 38 la caja, el 42 el charles cerrado, sea cual sea el programa elegido. Los manuales cuentan los programas desde 1 («41 Violin»); los bytes cuentan desde 0.

`general_midi.rs`, en `lineup-core`, guarda la lista. Los programas que usan las demostraciones y los estilos tienen nombre, y cada programa tiene su nombre y su

familia:

```
pub const VIOLIN: u8 = 40;
pub const CELLO: u8 = 42;
pub const CHOIR_AAHS: u8 = 52;
```

```
/// The name of program `program`, such as "Violin" for 40.
pub fn name(program: u8) -> &'static str {
    NAMES[usize::from(program.min(LAST))]
}

/// The family program `program` belongs to, 0 (piano) to 15 (sound
effects).
pub fn family(program: u8) -> usize {
    usize::from(program.min(LAST)) / 8
}
```

`NAMES` es un `[&str; 128]` y `FAMILIES` un `[&str; 16]`, los dos en el orden de General MIDI. Como cada familia tiene ocho programas, «qué familia» es una división por 8, y los programas de la familia `f` son `f * 8..f * 8 + 8`. `.min(LAST)` impide que un número mayor que 127 indexe más allá del final: un `u8` puede valer hasta 255, y un archivo dañado podría traer uno. Los menús se construyen con esos mismos dos arrays, y los nombres pasan por la tabla de traducciones, así que el menú en español dice «Violín».

La batería integrada conserva sus propios sonidos, pero todo lo que habla General MIDI necesita cada tambor como una nota del canal 10. `general_midi_drum`, en `midi_file.rs`, es esa tabla:

```
/// The note General MIDI drum sets play for each built-in drum.
pub fn general_midi_drum(drum: Drum) -> Note {
    Note::new(match drum {
        Drum::Kick => 36,
        Drum::Snare => 38,
        Drum::Clap => 39,
        Drum::HiHat => 42,
        Drum::OpenHat => 46,
        Drum::Rim => 37,
        Drum::LowTom => 45,
        Drum::HighTom => 50,
        Drum::Ride => 51,
        Drum::Crash => 49,
        Drum::Cowbell => 56,
        Drum::Shaker => 70,
        Drum::Conga => 63,
    })
}
```

El 37 es el «Side Stick» de General MIDI, el golpe de aro; el 70 son las «Maracas», lo más parecido que tiene a un shaker; el 63 es «Open Hi Conga», la conga aguda abierta. `drum_name` va en sentido contrario para los 47 tambores del 35 al 81, y así el mezclador puede decir qué tambor toca una nota. Un test compara unos cuantos con la lista de General MIDI, y otro comprueba los nombres de los programas, de modo que una errata en cualquiera de las dos tablas hace fallar los tests.

El instrumento de un canal

`Channel` tiene un campo para él:

```
/// The General MIDI instrument the channel plays, 0 to 127 (see
/// [`general_midi`](crate::general_midi)), or `None` for none in
/// particular. MIDI files and the MIDI output send it to the synth as
/// a program change; the Lineup Osc picks a sound close to it.
#[serde(default, skip_serializing_if = "Option::is_none")]
pub program: Option<u8>,
```

`Option<u8>` porque un canal puede no tener ningún instrumento en particular. Entonces no se envía nada a un sintetizador, y el Lineup Osc conserva su diente de sierra brillante. Los dos atributos de serde mantienen legibles los archivos viejos: un archivo sin el campo se lee como `None` (`default`), y un canal sin instrumento no escribe el campo (`skip_serializing_if`). Solo un proyecto en el que algún canal tiene instrumento necesita el formato 9, así que un Ferrous Lineup más antiguo sigue abriendo el resto (El número de versión).

Un canal de batería no tiene programa: en el canal MIDI 10 son las notas las que eligen los tambores. Un sampler que toca un archivo WAV tampoco lo tiene, porque su sonido es el archivo.

De dónde sale el número

- **El rack y el mezclador.** Haz clic derecho en el nombre de un canal y abre **Instrumento**, o haz clic en el nombre del instrumento en el mezclador. Los dos muestran `instrument_choices`: «Ninguno (el sonido propio del Lineup Osc)», y luego las dieciséis familias, cada una un submenú con sus ocho programas, numerados desde 1 como en los manuales. La familia del programa actual lleva un punto delante, para encontrarla en una lista larga.
- **La banda.** El panel de la banda de la hoja de acordes da a cada músico un instrumento con las mismas familias (Cómo toca la banda una hoja de acordes).

- **La biblioteca.** Los canales **Teclados** y **Bajo** de un estilo los crea `synth_named`, que les da un piano y un bajo:

```
let program = match name {
    KEYS => Some(general_midi::ACOUSTIC_GRAND_PIANO),
    BASS => Some(general_midi::FINGER_BASS),
    _ => None,
};
```

- **Las demostraciones.** Cada voz de una demostración dice su instrumento: violines y un violonchelo en el Canon de Pachelbel, un bandoneón («Acordeón de tango» en General MIDI) en un tango (Cómo se escribe una demostración). Un test construye cada canción de demostración y comprueba que todos sus canales de sintetizador tienen programa.

- **Los archivos MIDI.** Al importar un archivo, cada canal MIDI toma como instrumento su primer cambio de programa (MIDI).

Adónde va el número

1. **La exportación MIDI** escribe un cambio de programa al principio de la pista de cada canal, antes de su primera nota, para que el sintetizador esté listo:

```
// The instrument first, so the synth is ready before the first note.
if let Some(program) = channel.program
    && midi_channels[index] != DRUM_CHANNEL
{
    let change = MidiMessage::ProgramChange {
        channel: midi_channels[index],
        program,
    };
    track.event(0, &change.bytes());
}
```

2. **La salida MIDI** lo envía justo antes de la primera nota del canal, y de nuevo después de cada parada, por si entretanto se cambió el sonido en el propio sintetizador (MIDI).

3. **El Lineup Osc** toca un `Timbre` según la familia del programa.

4. **El SoundFont** toca las grabaciones del programa, que es el resto de este capítulo.

Qué es un SoundFont

Un sampler toca una grabación a distintas velocidades (El sampler): al doble de velocidad suena una octava más aguda. Eso funciona para unas pocas notas, pero un piano grabado en el Do central y tocado dos octavas más arriba suena a juguete. El timbre de un instrumento real cambia a lo largo de su registro, y acelerado, su ataque se vuelve demasiado corto.

Un SoundFont (un archivo `.sf2`, un formato de las tarjetas de sonido de los años noventa) es toda una biblioteca de sampler en un solo archivo:

- **muestras**: grabaciones cortas, el do, el mi, el sol♯ de un piano y así cada pocas notas, cada una con la nota en la que se grabó. Los sonidos que duran, como un órgano o las cuerdas, tienen un **bucle**: una vez sonado el ataque, el centro de la grabación se repite mientras la tecla siga pulsada;
- **instrumentos**: qué muestra tocar para qué teclas y velocidades (una nota suave puede tocar la grabación de una nota suave), y cómo: la envolvente de volumen, un filtro, el vibrato;
- **presets**: lo que elige un cambio de programa MIDI, por **banco** y **programa**.

Un SoundFont General MIDI pone los 128 instrumentos de General MIDI en el banco 0, en su orden numerado, y un kit de batería en el banco 128, que el sintetizador toca en el canal MIDI 10. Así, el programa 0 es siempre un piano de cola, el 33 un bajo eléctrico tocado con los dedos, y la tecla 36 del canal 10 un bombo.

Tocar una nota es buscar la muestra cuyo rango de teclas y velocidades la contiene, y tocarla más rápido o más lento según la distancia entre las dos notas. Cada semitono es un factor de 2^(1/12), alrededor de 1,0595, así que una muestra de La4 (440 Hz) tocada para Si4, dos semitonos más arriba, va 2^(2/12) ≈ 1,122 veces más rápido: 494 Hz.

Quién lo lee y lo toca: rustysynth

Leer un archivo `.sf2` y tocarlo es mucho código: interpretar el archivo, los rangos de teclas, las envolventes, los filtros, los bucles, mezclar decenas de voces. El crate

`rustysynth` lo hace todo, en Rust puro sin ninguna biblioteca de C que instalar, con la licencia MIT, que un programa GPL puede usar. Su interfaz es pequeña; en resumen:

```
let sound_font = Arc::new(SoundFont::new(&mut file)?);
let mut synthesizer = Synthesizer::new(&sound_font, &settings)?;
synthesizer.process_midi_message(0, 0xC0, 33, 0); // channel 1: program 33
synthesizer.note_on(0, 40, 100); // E2, velocity 100
synthesizer.render(&mut left, &mut right); // the next block of sound
```

`soundfont::load` es la primera línea con los errores explicados: un archivo que no se puede abrir es un `SoundFontError::Io`, y uno que no es un SoundFont, o está dañado, es un `SoundFontError::Invalid`. Cada uno imprime un mensaje traducido, así que la barra de estado puede decir «no es un archivo SoundFont (...)» en el idioma de la aplicación.

Habla MIDI. Y la aplicación ya tiene un instrumento que habla MIDI: la **salida MIDI** (`MidiOut`), que lleva la cuenta de qué notas suenan y cuándo debe acabar cada una, y junta mensajes de nota activada, nota desactivada y cambio de programa en un buzón de salida para un sintetizador externo (MIDI). Así que un instrumento de SoundFont es una salida MIDI cuyos mensajes van a un sintetizador dentro de la aplicación.

Una salida MIDI que toca dentro de la aplicación

```
/// A channel's instrument played from a SoundFont.
pub struct SoundFontPlayer {
    /// Which notes sound and until when, as for an external synth; its
    /// messages go to the synthesizer once per block.
    notes: MidiOut,
    /// Boxed so an `Instrument` stays small.
    block: Box<Block>,
}

/// The synthesizer and the block of sound it rendered last.
struct Block {
    synthesizer: Synthesizer,
    sound_font: Arc<SoundFont>,
    /// [`GAIN`] or [`DRUM_GAIN`].
    gain: f32,
    left: [f32; BLOCK],
    right: [f32; BLOCK],
    /// The next frame of `left` and `right` to hand out. At `BLOCK`, a
    /// new block is due.
    read: usize,
}
```

El hilo de audio pide a cada instrumento una muestra cada vez. El sintetizador trabaja en bloques de `BLOCK`, 64 frames, así que el reproductor reparte el bloque que tiene, y cuando se le acaba le pasa los mensajes de los últimos 64 frames y le pide el bloque siguiente:

```
pub fn next_sample(&mut self) -> f32 {
    self.notes.next_sample();
    let block = &mut *self.block;
    if block.read == BLOCK {
        let synthesizer = &mut block.synthesizer;
        self.notes.flush(|message| play(synthesizer, message));
        synthesizer.render(&mut block.left, &mut block.right);
        block.read = 0;
    }
    let sample = (block.left[block.read] + block.right[block.read]) * 0.5
* block.gain;
    block.read += 1;
    sample
}
```

`self.notes.next_sample()` no hace sonido: el `next_sample` de una salida MIDI cuenta frames, para saber cuándo se cumple la duración de cada nota y enviar su nota desactivada. Una nota que empieza a mitad de un bloque se oye desde el siguiente: como mucho 64 frames tarde, 1,3 ms a 48 kHz, muy por debajo de lo que nadie oye.

`let block = &mut *self.block;` atraviesa el `Box` una sola vez, y así las líneas siguientes toman prestados `block.synthesizer` y `block.left` como campos separados. Dentro del `if`, `self.notes` y `block.synthesizer` están prestados a la vez, uno por `flush` y otro por el closure; Rust lo permite porque son campos distintos.

Los sonidos del SoundFont son estéreo, y cada canal de la aplicación es mono hasta que el mezclador lo coloca con su balance (El mezclador). Así que los dos lados se suman y se dividen por dos.

`flush` es el mismo método que entrega los mensajes de una salida MIDI al puerto MIDI, y `play` convierte cada `MidiMessage` en la llamada correspondiente del sintetizador:

```
/// Hands one MIDI message to the synthesizer.
fn play(synthesizer: &mut Synthesizer, message: MidiMessage) {
    match message {
        MidiMessage::NoteOn {
            channel,
            pitch,
            velocity,
        } => synthesizer.note_on(
            i32::from(channel),
            i32::from(pitch.midi()),
            i32::from(velocity),
        ),
        MidiMessage::NoteOff { channel, pitch } => {
            synthesizer.note_off(i32::from(channel),
i32::from(pitch.midi()))
        }
        MidiMessage::ProgramChange { channel, program } => {
            synthesizer.process_midi_message(i32::from(channel), 0xC0,
i32::from(program), 0)
        }
    }
}
```

`i32::from` ensancha un `u8` al `i32` que pide rustysynth; no puede fallar, y por eso es `from` y no `try_from`.

Todo lo que una salida MIDI ya hace bien viene gratis: notas que acaban al cumplirse su duración, teclas sostenidas desde el teclado, notas que se callan cuando se silencia su pista, duraciones que se estiran con un cambio de tempo. El motor también trata a las dos igual en una cosa más: MIDI tiene una sola nota de cada altura por canal, así que una nota que vuelve a empezar termina la que suena (`one_per_pitch` en `engine.rs`).

Un reproductor se crea para un programa:

```
pub fn new(sound_font: &Arc<SoundFont>, sample_rate: u32, program: u8) ->
Option<Self> {
    let notes = MidiOut::new(0).with_program(Some(program));
    Self::on(sound_font, sample_rate, notes, GAIN)
}
```

`with_program` hace que la salida MIDI envíe el cambio de programa antes de su primera nota, de modo que el sintetizador está en el instrumento correcto cuando llega la primera nota. Un tambor es una salida MIDI en el canal 10 que toca una sola nota, sea cual sea la nota de sus pasos (`playing_as`), como hace un tambor enviado a un sintetizador externo:

```
pub fn drum(sound_font: &Arc<SoundFont>, sample_rate: u32, drum: Note) ->
Option<Self> {
    let notes = MidiOut::new(DRUM_CHANNEL).playing_as(Some(drum));
    Self::on(sound_font, sample_rate, notes, DRUM_GAIN)
}
```

Un sintetizador por canal, un SoundFont para todos

Un sintetizador puede tocar dieciséis canales MIDI a la vez, así que uno solo podría tocar toda la banda. En cambio, cada canal de la aplicación tiene el suyo. Así su volumen, su balance, sus efectos, su luz de silencio y su medidor de nivel funcionan exactamente como en cualquier otro canal, porque el motor mezcla canales, no canales MIDI.

El SoundFont en sí ocupa decenas de megabytes, y no se copia. Se lee una vez en un `Arc<SoundFont>`, un puntero con *recuento de referencias atómico*: `Arc::clone` copia el puntero y suma uno a un contador, y la memoria se libera cuando desaparece el último clon. Cada sintetizador guarda un clon, y la aplicación otro. Un instrumento que el hilo de audio reemplaza vuelve por la cola de basura y se descarta en el hilo de la ventana (Liberar memoria en otro sitio), así que, aunque guarde el último clon, el SoundFont nunca se libera en el hilo de audio.

`Instrument` tiene una variante para él, y un constructor que dice qué canales toca el SoundFont:

```
pub fn from_sound_font(
    channel: &Channel,
    sample_rate: u32,
    sound_font: &Arc<SoundFont>,
    drums: bool,
) -> Option<Self> {
    let player = match &channel.kind {
        ChannelKind::Synth => SoundFontPlayer::new(sound_font,
sample_rate, channel.program?),
        ChannelKind::Sampler(SampleSource::Builtin(drum))
            if drums && channel.sent_on.is_none() =>
        {
            SoundFontPlayer::drum(sound_font, sample_rate,
general_midi_drum(*drum))
        }
        _ => None,
    };
    player.map(Instrument::SoundFont)
}
```

El `?` después de `channel.program` devuelve `None` de toda la función cuando el canal no tiene programa: eso es lo que mantiene en el Lineup Osc un canal en «Ninguno». Un tambor integrado toca el kit del SoundFont cuando el ajuste de la batería está activado, salvo que se esté enviando a un sintetizador externo (`sent_on`). Un archivo de muestra y una salida MIDI no le tocan al SoundFont. El `build_instrument` de la aplicación pregunta esto primero y recurre a `Instrument::for_channel` cuando la respuesta es `None`:

```
fn build_instrument(&mut self, channel: &Channel, sample_rate: u32) ->
Instrument {
    if let Some((sound_font, drums)) = self.sounds.in_use()
        && let Some(instrument) =
            Instrument::from_sound_font(channel, sample_rate, sound_font,
drums)
    {
        return instrument;
    }
    match Instrument::for_channel(channel, sample_rate, &mut self.samples)
{
        ...
    }
}
```

Todos los instrumentos se construyen ahí, para tocar y para exportar, así que **Archivo > Exportar WAV…** y la exportación de video también suenan con el SoundFont.

Los dos bloques de 64 muestras de un sintetizador ocupan medio kilobyte, y cada hueco de la cola hacia el hilo de audio es tan grande como el comando más grande. Por eso el `Block` va en un `Box`, igual que los demás instrumentos guardan sus voces en uno (Un número fijo de voces, en un `Box`); el test `commands_stay_small` detectaría un comando que creciera.

Nada se reserva en el hilo de audio

El hilo de audio nunca pide memoria (La regla de oro). Un sintetizador reserva todo lo que necesita al crearse: sus voces (`VOICES`, 128 por canal; una nota de piano puede usar dos, su muestra izquierda y la derecha), sus buffers, la tabla de presets. `Synthesizer::new` se ejecuta en el hilo de la ventana, dentro de `build_instrument`, y el instrumento terminado se envía al otro lado. Su `note_on` y su `render` solo reutilizan esas voces y esos buffers.

`on` prepara el sintetizador:

```
let mut settings =
SynthesizerSettings::new(i32::try_from(sample_rate).ok()?);
settings.block_size = BLOCK;
settings.maximum_polyphony = VOICES;
// Reverb and chorus would cost as much as the voices, once for
// every channel. The sounds are recorded with some room already.
settings.enable_reverb_and_chorus = false;
let synthesizer = Synthesizer::new(sound_font, &settings).ok()?;
```

El mezclador tiene una reverberación a la que envían todos los canales, así que la sala sigue ahí, una sola vez (El mezclador). `i32::try_from(sample_rate).ok()?` es la conversión cuidadosa: un `u32` mayor que `i32::MAX` no cabría, y `?` sobre el `Option` se rinde en lugar de dar la vuelta. El propio sintetizador rechaza frecuencias de muestreo por debajo de 16 kHz y por encima de 192 kHz, y entonces el canal se queda con su Lineup Osc.

Con volumen suficiente

rustysynth toca bajo, a medio volumen, porque espera que lo comparta toda una banda. El reproductor lo sube por `GAIN` (5) y la batería por `DRUM_GAIN` (7). Esos números salen de una medición, no del gusto: se tocaron las mismas notas con el Lineup Osc y con el SoundFont, y se compararon sus picos y sus niveles medios. Un piano real se apaga mientras el Lineup Osc sostiene sus notas, así que los dos nunca coinciden del todo, y otro SoundFont puede sonar más fuerte o más suave; el volumen de cada canal hace el resto.

La oferta, y la descarga

La aplicación no trae un SoundFont, porque uno bueno ocupa decenas de megabytes. Apenas la aplicación se abre con sus instrumentos en el Lineup Osc, antes de que nada suene, una ventana ofrece uno:



Descargar baja GeneralUser GS, un SoundFont General MIDI gratuito de S. Christian Collins, desde su propio repositorio, y lo guarda en la carpeta de datos de la aplicación: `~/.local/share/ferrouslineup/soundfonts` en Linux, la carpeta `Application Support` en macOS, `AppData\Roaming` en Windows. **Elegir un archivo…** toma cualquier otro `.sf2`. **Ahora no** se queda con el Lineup Osc. **Edición > Ajustes… > Audio** tiene las mismas opciones para más adelante: **Instrumentos del SoundFont**, **Batería del SoundFont**, **Elegir SoundFont…** y **Usar GeneralUser GS**.

Si GeneralUser GS ya está en esa carpeta, descargado por una ejecución o una versión anterior, no hay nada que descargar, y la ventana ofrece **Usarlo** en su lugar:



Se pregunta hasta que se responde

Un solo método decide si preguntar:

```
pub(super) fn offer_sound_font(&mut self) {
    let sounds = &mut self.sounds;
    if !sounds.settings.on && !sounds.settings.turned_down &&
sounds.general_user.is_some() {
        sounds.offer_open = true;
    }
}
```

Abre la ventana siempre que los instrumentos suenan con el Lineup Osc y la oferta no fue rechazada. No se pregunta nada mientras los sonidos están activados, aunque su SoundFont todavía se esté leyendo, ni en un sistema sin carpeta para datos de aplicaciones (ahí `general_user` es `None`).

Se llama en dos lugares. `LineupApp::new` lo llama una vez creada la aplicación, así que la pregunta es lo primero que muestra la ventana:

```
// Real instrument sounds are offered as soon as the window opens,
// before anything plays.
app.offer_sound_font();
```

y todas las formas de empezar a tocar (el botón ▶, **Espacio**, una escucha previa de la biblioteca, abrir una demostración) pasan por `set_playing`, que termina con

```
self.playing = true;
self.offer_sound_font();
```

así que una oferta cerrada sin responder vuelve al reproducir. La reproducción sigue mientras la ventana pregunta, así que el sonido nunca espera a una pregunta.

Una ejecución que no se cerró bien tiene su propia pregunta al arrancar: si recuperar el trabajo sin guardar (ver Una copia del trabajo sin guardar). Las dos se taparían, así que los sonidos esperan a esa:

```
// Unsaved work from a run that did not close cleanly is offered
// first.
if !self.sounds.offer_open || !self.recovery.pending.is_empty() {
    return;
}
```

Abrir la ventana no responde nada. Una ejecución que se cierra antes de responder vuelve a preguntar la próxima vez que se abre la aplicación, y lo mismo pasa si su SoundFont desapareció. Solo un «no» termina la oferta:

```
fn decline_sound_font(&mut self) {
    let sounds = &mut self.sounds;
    sounds.offer_open = false;
    sounds.settings.turned_down = true;
    sounds.turn_off();
}
```

Ahora no la llama, y también cerrar la ventana antes de una descarga y **Detener** durante una. Desmarcar **Instrumentos del SoundFont** en los ajustes también marca `turned_down`, y marcarlo lo borra. Un «sí» activa los sonidos y borra `turned_down`:

```
// Yes: GeneralUser GS, even in place of a chosen file that is
// gone, played from as soon as it is downloaded or read.
if (download || use_it)
    && let Some(path) = self.sounds.general_user.clone()
{
    let sounds = &mut self.sounds;
    sounds.settings.file = None;
    sounds.settings.turned_down = false;
    if download {
        sounds.settings.on = true;
        sounds.changed = true;
        sounds.start(path, true);
    } else {
        sounds.offer_open = false;
        self.set_sound_font(true);
    }
}
```

Descargar empieza la descarga y deja la ventana abierta para mostrarla. **Usarlo** cierra la ventana y pasa por `set_sound_font`, igual que marcar el ajuste, que encuentra el archivo en el disco y empieza a leerlo. En los dos casos, cuando el SoundFont está listo, `check_sounds` vuelve a dar a cada canal su instrumento y la barra de estado dice «Los instrumentos suenan desde GeneralUser GS». La ventana se dibuja en el orden `Foreground` de egui, encima de todas las demás, porque una demostración abre su hoja de acordes y la trae al frente en el mismo frame en que empieza a sonar.

Los ajustes son una pequeña struct:

```
#[derive(Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(default)]
struct Settings {
    /// Whether channels play from the SoundFont.
    on: bool,
    /// Whether the built-in drums do too.
    drums: bool,
    /// A SoundFont file chosen with "Choose SoundFont…", or `None` for
    /// GeneralUser GS.
    file: Option<PathBuf>,
    /// Whether the offer of real instrument sounds was turned down, with
    /// "Not now", or the sounds were turned off in Settings. Until then,
    /// the app offers them as it opens and on Play whenever they are off.
    turned_down: bool,
}
```

Se guardan entre ejecuciones por nombre, como `"sound-font"`, con el módulo `preferences` de la aplicación (Guardado por nombre, no por tipo):

```
settings: preferences::get(ctx, SETTINGS_NAME).unwrap_or_default(),
```

al arrancar la aplicación, y `preferences::set(ctx, SETTINGS_NAME, &self.sounds.settings)` cada vez que cambian. Guardados por nombre, sobreviven a una actualización de la aplicación. En la memoria propia de egui no sobrevivirían: una actualización desactivaría los sonidos y, con GeneralUser GS todavía en el disco, la aplicación volvería al Lineup Osc.

`#[serde(default)]` sobre la struct rellena cualquier campo que falte en los datos guardados con su valor de `Default`, de modo que se puede añadir un campo sin perder los ajustes de nadie, y un campo que la struct ya no tiene se salta. Por eso renombrar un campo olvida solo ese campo: un `turned_down` renombrado empieza en `false`, y se vuelve a preguntar a todos. `Default` está escrito a mano en lugar de derivado, porque `drums` empieza en `true`.

Descargar en un hilo propio

Una descarga de 30 MB tarda entre un segundo y un minuto, y leer un SoundFont también lleva un momento. Las dos cosas se hacen en un hilo propio, como la exportación WAV, y la ventana sigue dibujándose. El hilo y la ventana comparten un `Progress`:

```
/// How far a download is. Its thread writes it and the UI thread reads
/// it; atomics need no lock for that.
#[derive(Default)]
struct Progress {
    /// Bytes downloaded so far.
    done: AtomicU64,
    /// The size of the file, or 0 while it is not known.
    total: AtomicU64,
    /// Set by the UI thread to stop the download.
    cancel: AtomicBool,
}
```

Los *atómicos* son números que dos hilos pueden leer y escribir a la vez sin un cerrojo: la descarga guarda cuántos bytes lleva, la ventana lo lee para dibujar la barra, y **Detener** guarda `true` en `cancel`, que la descarga mira antes de cada lectura de hasta 64 KB. `Ordering::Relaxed` basta, porque ningún otro dato depende de estos números: una barra que va un frame atrasada no importa.



La petición la hace `ureq`, un cliente HTTP pequeño, con los certificados en los que confía el sistema operativo (así funciona detrás del proxy de una empresa que tenga los suyos) y con el proxy de `HTTPS_PROXY`, si lo hay. La variable de entorno `FERROUS_LINEUP_SOUNDFONT_URL` da otra dirección, como una copia en tu propia red. Un archivo de más de un gigabyte se rechaza: ningún SoundFont General MIDI es tan grande, así que es otra cosa.

El archivo se escribe como `GeneralUser-GS.sf2.part`, luego se lee como SoundFont, lo que comprueba que lo es, y solo entonces se renombra:

```
fn download_to(url: &str, path: &Path, progress: &Progress) ->
Result<Arc<SoundFont>, String> {
    let part = path.with_extension("sf2.part");
    let result = download_part(url, &part, progress).and_then(|()| {
        let sound_font = soundfont::load(&part).map_err(|err|
err.to_string())?;
        fs::rename(&part, path).map_err(|err| err.to_string())?;
        Ok(sound_font)
    });
    if result.is_err() {
        let _ = fs::remove_file(&part);
    }
    result
}
```

Una descarga cortada a medias, o una página web que llegó en lugar del archivo, nunca deja un `GeneralUser-GS.sf2` roto que fallaría cada vez que arranca la

aplicación. `and_then` ejecuta el segundo paso solo si el primero salió bien, y `let _ =` dice a propósito que no vale la pena informar de un archivo `.part` que no se puede borrar.

Una vez por frame, `check_sounds` guarda los ajustes si cambiaron y pregunta al hilo si ha terminado (`JoinHandle::is_finished`). Mientras no, le pide a egui que vuelva a dibujar dentro de una décima de segundo, para que la barra avance aunque el ratón no se mueva. Cuando termina, `join` entrega el `Arc<SoundFont>` , y cada canal necesita un instrumento nuevo hecho con él. La aplicación ya sabe hacerlo: cuando algunos canales empiezan o dejan de enviarse por MIDI, `send_routes` reconstruye los instrumentos cuya ruta cambió. Olvidar todas las rutas hace que los reconstruya todos en el frame siguiente:

```
/// Has every channel of the audio thread given its instrument again,
/// on the next frame: the SoundFont was turned on or off.
pub(super) fn instruments_changed(&mut self) {
    self.routes_sent = None;
}
```

Desactivar los sonidos hace lo mismo, y cada canal recupera su Lineup Osc o su tambor integrado. El SoundFont sigue cargado mientras los sonidos están desactivados, así que volver a activarlos es instantáneo.

Compilaciones de depuración

`cargo run` compila sin optimizaciones, y un sintetizador calcula cada muestra de cada voz: sin optimizar, una banda podría ser demasiado para el hilo de audio. Cargo puede optimizar una sola dependencia mientras el resto sigue compilando rápido, en el `Cargo.toml` del workspace:

```
[profile.dev.package.rustysynth]
opt-level = 3
```

Cómo se prueba

Los tests no deberían necesitar una descarga de 30 MB, y el repositorio no debería guardar ninguna. Así que el módulo de tests escribe un SoundFont byte a byte: una muestra, una onda senoidal de 440 Hz en bucle, dos instrumentos (uno toca la

muestra tal como se grabó, otro una octava más aguda) y tres presets (el programa 0, el programa 33 y un kit de batería en el banco 128).

Un archivo `.sf2` es un archivo RIFF, el mismo contenedor que un archivo WAV: un árbol de *chunks*, cada uno con un nombre de cuatro letras, un tamaño y sus bytes. Dos funciones pequeñas los construyen:

```
/// One chunk of a RIFF file: its name, its size and its bytes, padded
/// to an even length.
fn chunk(id: &[u8; 4], body: &[u8]) -> Vec<u8> {
    let mut bytes = id.to_vec();
    bytes.extend((body.len() as u32).to_le_bytes());
    bytes.extend(body);
    if body.len() % 2 == 1 {
        bytes.push(0);
    }
    bytes
}

/// A list of chunks: a chunk whose bytes are a type and other chunks.
fn list(kind: &[u8; 4], chunks: &[Vec<u8>]) -> Vec<u8> {
    let mut body = kind.to_vec();
    for each in chunks {
        body.extend(each);
    }
    chunk(b"LIST", &body)
}
```

El archivo es una lista de tres listas: `INFO` (una versión y un nombre), `sdta` (las muestras, como números de 16 bits) y `pdta` (los presets, los instrumentos y las cabeceras de las muestras, en nueve tablas de tamaño fijo). Luego los tests lo tocan y escuchan como lo haría un osciloscopio: una onda senoidal cruza el cero dos veces por periodo, así que contar los cruces en un cuarto de segundo, por dos, da su frecuencia.

```
#[test]
fn the_program_picks_the_instrument() {
    let mut player = SoundFontPlayer::new(&tiny_sound_font(), RATE,
33).unwrap();
    player.start(Note::new(A4), 100, HELD, None);
    let sound = render(&mut player, RATE as usize / 2);
    assert!(
        (frequency(&sound) - 880.0).abs() <= 8.0,
        "{}",
        frequency(&sound)
    );
}
```

El programa 33 es el instrumento una octava más agudo, así que el La4 sale a 880 Hz. Otros tests comprueban que un reproductor está en silencio hasta que empieza una nota, que el la3 sale a 220 Hz, que una nota empieza dentro de un bloque y acaba al cumplirse su duración, que las notas más fuertes suenan más fuerte, que un tambor toca su propia tecla sea cual sea la nota en la que está escrito, que se rechaza un archivo que no es un SoundFont, y qué canales toma el SoundFont (`channels_without_a_sound_in_the_sound_font_keep_theirs`).

Los tests de ventana no deben depender de si el ordenador que los ejecuta ha descargado GeneralUser GS. Así que cada aplicación de prueba lo busca en una ruta que nunca existe, `keep_general_user_in(no_sound_font())`, y arranca como si la oferta ya se hubiera rechazado. Cuatro tests deshacen eso.
`the_app_offers_real_instrument_sounds_as_it_opens` hace lo que hace `LineupApp::new` y comprueba que la ventana se abre antes de que nada suene, con **Descargar**, y que tras **Ahora no** no vuelve, ni tras reiniciar ni tras una actualización.
`play_offers_real_instrument_sounds_until_the_offer_is_answered` pulsa **Espacio** y comprueba que la ventana se abre mientras la canción suena, con **Descargar**. Vuelve a arrancar la aplicación sin responder y comprueba que reproducir pregunta otra vez. Luego hace clic en **Ahora no**, reproduce dos veces, y comprueba que no se pregunta nada, ni tras reiniciar ni tras una actualización.
`play_offers_a_sound_font_already_on_disk` pone un archivo vacío donde va GeneralUser GS, comprueba que la ventana ofrece **Usarlo**, hace clic y comprueba que el archivo se está leyendo.
`a_demo_asks_for_real_instrument_sounds_in_front_of_its_chord_sheet` comprueba que la ventana queda encima, y que sus botones reciben el clic.

El mezclador: niveles, balance, reverberación, efectos y orden de partitura

Archivos: `crates/lineup-core/src/mixer.rs`, `crates/lineup-core/src/effects.rs`, `crates/lineup-core/src/score_order.rs`, `crates/lineup-audio/src/reverb.rs`, `crates/lineup-audio/src/effects.rs`, `crates/lineup-audio/src/levels.rs`, `crates/lineup-app/src/app/mixer.rs` y `crates/lineup-app/src/meters.rs`

El mezclador (**F9**) es donde se equilibra la banda. Cada canal tiene una línea: su instrumento, un medidor de nivel, su volumen, dónde se ubica entre los altavoces (**Balance**), cuánto de él va a la sala (**Reverberación**), sus propios **Efectos** (un ecualizador, un compresor y un delay), **Silenciar** y **Solo**. Encima están el volumen **Principal**, con un medidor para cada altavoz, y la reverberación que comparten todos los canales. Las líneas aparecen como una partitura de director presenta los instrumentos, por secciones.



El trabajo se reparte como en todo el programa. `lineup-core` guarda los ajustes, que se guardan con la canción; `lineup-audio` los convierte en sonido en el hilo de audio; `lineup-app` dibuja la ventana. Cómo usar el mezclador está en el capítulo El mezclador del manual.

Qué se guarda

El proyecto tiene un `Mixer`:

```
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
pub struct Mixer {
    pub master: f32,
    pub reverb: ReverbSettings,
    pub solo: Option<usize>,
}
```

```
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
pub struct ReverbSettings {
    pub enabled: bool,
    /// Wet fraction, from 0 to 1.
    pub mix: f32,
    /// Approximate time to decay by 60 dB, in seconds.
    pub decay: f32,
    /// High frequency absorption, from 0 to 1.
    pub damping: f32,
    pub pre_delay_ms: f32,
}
```

y cada `Channel` tiene su propia mezcla: `volume` y `muted`, y tres campos más:

```
#[serde(default, skip_serializing_if = "is_centred")]
pub pan: f32,
#[serde(default = "full_send", skip_serializing_if = "is_full_send")]
pub reverb_send: f32,
#[serde(default, skip_serializing_if = "Effects::is_plain")]
pub effects: Effects,
```

`skip_serializing_if` deja un campo fuera del archivo mientras tiene el valor de un canal nuevo (en el centro, todo enviado a la reverberación, sin efectos), y `default` lo vuelve a poner cuando falta. Una canción cuyos canales son todos así se escribe exactamente igual que con un Ferrous Lineup sin balances ni efectos, así que una versión más antigua la sigue abriendo. Solo una canción que los usa necesita el formato 26 (balance y envío) o el 28 (efectos) (El número de versión). Al abrir se comprueban los rangos: un archivo editado a mano con un balance de −3, o un delay cuyos ecos no se apagarían nunca, se rechaza con un mensaje en lugar de tocar un canal más fuerte que el máximo.

El **Solo** es un número de canal, no una marca en cada canal. El proyecto responde «¿se oye este canal?» en un solo sitio, y así todo lo que suena (el hilo de audio, una

exportación, un archivo MIDI) está de acuerdo:

```
pub fn channel_muted(&self, channel: usize) -> bool {
    self.channels.get(channel).is_none_or(|state| state.muted)
        || self.mixer.solo.is_some_and(|solo| solo != channel)
}
```

Por eso poner un canal en solo llega al hilo de audio como un cambio de silencio en todos los demás (`diff` : la lista de cambios).

Restablecer mezcla devuelve el principal, la reverberación, el solo y el volumen, el balance, el envío y los efectos de cada canal a como están en una canción nueva. Los músicos de la banda recuperan sus volúmenes de siempre, y los canales que la banda deja callados siguen silenciados (`Project::reset_mix` , en `band.rs`).

El camino de una muestra

En cada frame, el hilo de audio lleva cada canal por los mismos pasos. Este es el corazón de la mezcla, en `engine.rs` :

```
let wet = self.reverb.wet();
let mut dry = [0.0; 2];
let mut send = 0.0;
for channel in &mut self.channels {
    let volume = channel.volume.next_value();
    let sample = channel.effects.process(channel.instrument.next_sample())
* volume;
    channel.peak = channel.peak.max(sample.abs());
    let amount = channel.reverb_send.next_value();
    let gains = lineup_core::Channel::pan_gains(channel.pan.next_value());
    let kept = sample * (1.0 - wet * amount);
    dry[0] += kept * gains[0];
    dry[1] += kept * gains[1];
    send += sample * amount;
}
// ...
let click = self.click.next_sample();
let stereo = self
    .reverb
    .next(dry, send, wet)
    .map(|sample| sample + click);
let volume = self.master_volume.next_value();
let stereo = stereo.map(|sample| sample * volume);
master_peak = [0, 1].map(|side|
master_peak[side].max(stereo[side].abs()));
// `tanh` gently rounds off peaks above 1.0 instead of clipping them hard.
let stereo = stereo.map(f32::tanh);
```

Por orden:

1. El instrumento hace una muestra, mono.
2. Los **efectos** del canal le dan forma: ecualizador, compresor, delay.
3. Su **volumen** la escala, y su **pico** se guarda para el medidor.
4. Su **balance** la reparte entre los dos altavoces, y su **envío** manda una parte a la reverberación.
5. La **reverberación** añade la sala a la suma de todos los canales.
6. El clic del metrónomo se suma después de la reverberación: la sala es de la música, y un clic con cola es más difícil de seguir.
7. El **volumen principal** escala el conjunto, sus picos van a los medidores del principal, y `tanh` redondea lo que pase del máximo (Mezclar un frame).

El volumen, el balance, el envío y el nivel húmedo de la reverberación son todos `Smoothed`: al moverlos en la ventana, se deslizan hasta el nuevo valor en unos milisegundos en lugar de saltar, porque un salto en mitad de una onda se oye como un clic. Los efectos van antes que el volumen, como en el canal de una mesa de mezclas, así que un compresor oye el instrumento tal como toca, diga lo que diga el fader.

El balance: dos altavoces, un canal

El balance es un número de −1 (todo a la izquierda) a 1 (todo a la derecha), con 0 en el centro. Se convierte en una ganancia para cada altavoz:

```
/// How loud it is in the left and the right speaker for its pan: both
/// full in the middle, and the far side fading out as it moves away.
pub fn pan_gains(pan: f32) -> [f32; 2] {
    [(1.0 - pan).min(1.0), (1.0 + pan).min(1.0)]
}
```

```
balance  -1.0  -0.5   0.0   0.5   1.0
izq.      1.0   1.0   1.0   0.5   0.0
der.      0.0   0.5   1.0   1.0   1.0
```

En el centro los dos altavoces tocan el canal a todo volumen, como suena un sonido mono en un equipo estéreo. Mover un canal hacia un lado baja el **otro** altavoz, como la perilla de balance de un equipo de música.

Esta es una de varias **leyes de panorama**. Las mesas de mezclas suelen usar en cambio una ley de *potencia constante*, en la que el centro toca cada altavoz a alrededor de 0,71 (−3 dB), para que un canal suene igual de fuerte esté donde esté. Con la ley que se usa aquí, un canal llevado del todo a la izquierda suena un poco más bajo que en el centro, porque solo lo toca un altavoz. La razón para mantenerla es que una canción en la que nada tiene balance toca cada canal exactamente a su volumen en los dos lados.

El deslizador escribe su valor como `I30`, `C` o `D100` en español (`L30` y `R100` en inglés), y lee lo que escribes de la misma manera: `pan_value` acepta `L30`, `R100`, `C`, un número simple de −100 a 100, y los españoles `I30` (izquierda) y `D30` (derecha). Un doble clic lo devuelve al centro. Un deslizador solo detecta arrastres, así que el doble clic se lee del puntero y se compara con el rectángulo del deslizador.

La reverberación: una sala hecha de retardos

Una reverberación imita una sala. Un sonido rebota en las paredes, el suelo y el techo y vuelve, más tarde y más suave, miles de veces, hasta que los ecos se funden en una cola que se apaga en uno o dos segundos. La reverberación del mezclador es una **reverberación de Schroeder**, el diseño digital más antiguo (Manfred Schroeder, 1962), hecho con dos tipos de retardo.

Un **filtro peine** es un retardo cuya salida vuelve a su entrada, un poco más suave cada vez: un sonido vuelve cada 31 ms, por ejemplo, más bajo en cada vuelta. Un solo peine suena como un aleteo metálico, pero cuatro en paralelo, con longitudes que nunca coinciden, dan un patrón denso de ecos. Un **filtro pasatodo** es un retardo que esparce cada eco en un racimo sin cambiar el tono general; tres seguidos convierten los ecos de los peines en una capa uniforme.

```
fn new(rate: u32, spread: usize) -> Self {
    let length =
        |samples| ((samples + spread) as f32 * rate as f32 /
48_000.0).round() as usize;
    Self {
        combs: [1493, 1601, 1747, 1867].map(|samples| Comb {
            delay: Delay::new(length(samples)),
            feedback: 0.0,
            filtered: 0.0,
        }),
        diffusers: [241, 503, 719].map(|samples|
Delay::new(length(samples))),
    }
}
```

Las longitudes se cuentan en muestras a 48 kHz (1493 muestras son 31 ms) y se escalan a la frecuencia real. Todas son números primos, así que ningún par comparte un factor y sus ecos casi nunca caen en el mismo instante, lo que sonaría como una nota que resuena. Cada lado de la sala estéreo es un `Side` con sus propios peines y pasatodos; el derecho se crea con `spread` 37, así que sus retardos son 37 muestras más largos y los dos lados difieren, lo que el oído percibe como amplitud.

`[...].map(...)` sobre un array da un array de la misma longitud, así que `combs` es un `[Comb; 4]` construido en una sola expresión, sin `Vec`.

Cuánto resuena la sala

El **Decaimiento** es el tiempo que tarda la cola en bajar 60 dB, a una milésima de su nivel: la medida habitual de una sala, llamada RT60. La realimentación de cada peine sale de ahí:

```
fn configure(&mut self, rate: u32, decay: f32) {
    for comb in &mut self.combs {
        comb.feedback =
            10.0_f32.powf(-3.0 * comb.delay.samples.len() as f32 / (rate
as f32 * decay));
    }
}
```

En `decay` segundos el sonido da `rate × decay / longitud` vueltas a un peine, y en cada vuelta se multiplica por `feedback`. Para que el total sea 10^{-3} (−60 dB), cada vuelta multiplica por 10^(−3 × longitud / (rate × decay)). Los peines más largos dan menos vueltas, así que reciben una realimentación más fuerte, y los cuatro se apagan a la vez.

La **Amortiguación** imita el aire y las paredes blandas de una sala real, que se tragan las frecuencias agudas antes que las graves. Dentro de cada peine, el eco pasa por un filtro pasabajos de un polo antes de volver a entrar:

```
fn next(&mut self, input: f32, damping: f32) -> f32 {
    let out = self.delay.read();
    self.filtered = flushed(out * (1.0 - damping) + self.filtered *
damping);
    self.delay.write(input + self.filtered * self.feedback);
    out
}
```

`filtered` se mueve solo una parte del camino hacia cada muestra nueva, lo que suaviza las oscilaciones rápidas: los agudos. Cada vuelta los suaviza un poco más, así que la cola se oscurece mientras se apaga, como en una sala de conciertos.

El **Predelay** es el hueco antes de que la sala responda, hasta 200 ms: en una sala grande la primera reflexión llega más tarde que en una habitación pequeña. Es un buffer circular más delante de los peines, de una quinta parte de segundo.

Nunca demasiado pequeña, nunca un estorbo

`flushed` convierte cualquier valor menor que 10^{-20} en exactamente 0. Una cola que se apaga se hace cada vez más pequeña sin llegar nunca a cero, y los números tan diminutos (**denormales**) hacen que muchos procesadores vayan decenas de veces más lentos. Una reverberación que se deja sonar en silencio podría entonces pasarse del tiempo del hilo de audio.

Toda la memoria de los retardos se reserva en `Reverb::new`, antes de que arranque el stream; `configure` solo cambia números, así que mover un deslizador nunca reserva memoria en el hilo de audio. Un ajuste fuera de rango se ignora en lugar de aplicarse. Desactivada, el nivel húmedo baja hasta cero y entonces `Reverb::next` devuelve el sonido seco enseguida, sin hacer ningún trabajo. Activada de nuevo, empieza desde el silencio (`clear`), no desde la cola que tenía cuando se apagó.

El envío: cuánto de cada canal está en la sala

Cada canal tiene un **envío**, de 0 a 1: cuánto de él va a la sala. La suma de todos los envíos es la única entrada de la reverberación, así que veinte canales cuestan una reverberación, no veinte, como con un envío en una mesa real. La reverberación devuelve solo la sala, que `next` suma al sonido seco:

```
[
    dry[0] + left.next(delayed, damping) * wet,
    dry[1] + right.next(delayed, damping) * wet,
]
```

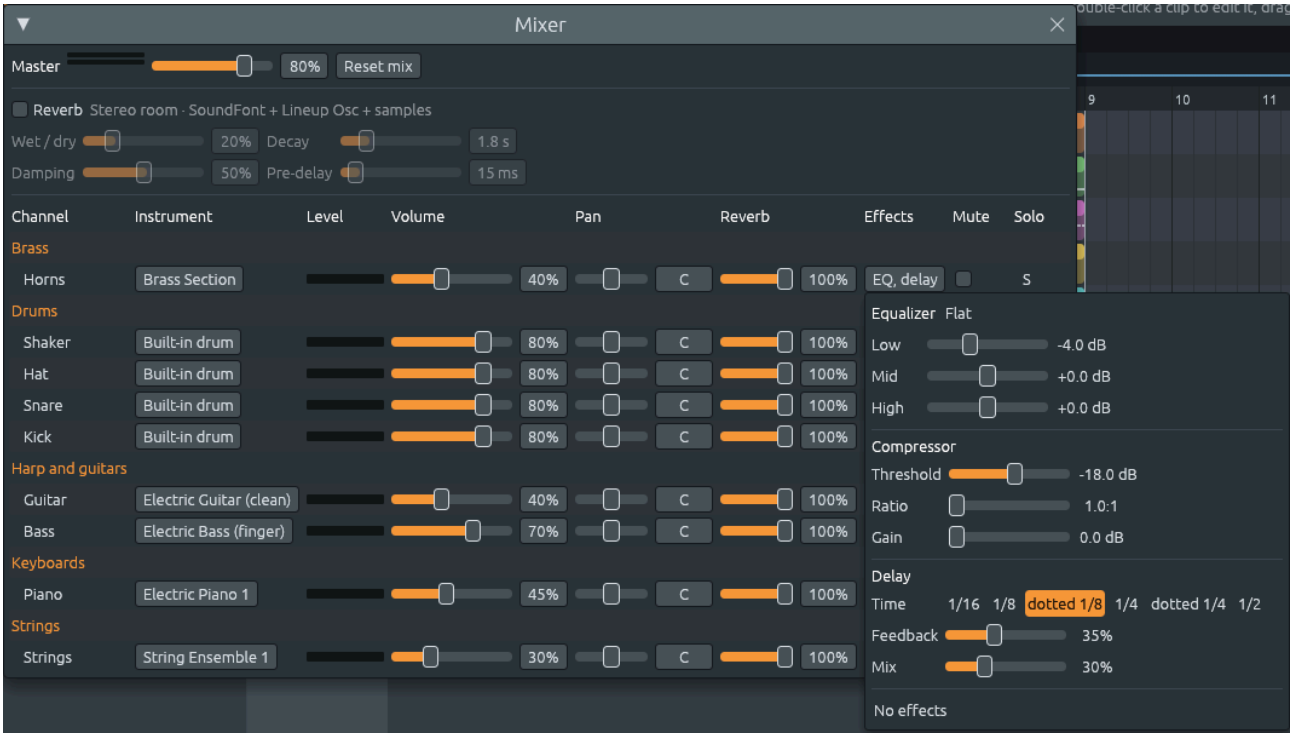
La línea que hay que mirar en la mezcla es `kept = sample * (1.0 - wet * amount)`: lo que entra en la sala se quita del sonido seco, en proporción al ajuste **Húmedo / seco**. Comprueba los dos extremos:

- **Envío 1** (todo, como en cada canal nuevo): `kept = sample × (1 - wet)`, y toda la muestra va a la sala. Es una mezcla húmedo / seco clásica de toda la canción.
- **Envío 0**: `kept = sample`, y nada va a la sala. El canal suena seco y cercano, como con la reverberación desactivada, mientras el resto de la banda sigue en la sala.

La sala es mono a la entrada y estéreo a la salida, así que un canal llevado del todo a la izquierda sigue teniendo su reverberación en los dos lados, como un guitarrista a la izquierda del escenario llena toda la sala.

Los efectos de cada canal

Una mesa de mezclas real tiene, en cada canal, unos cuantos efectos que dan forma a un instrumento antes de que se una a los demás. Cada línea del mezclador tiene tres, en el orden que tienen en la mayoría de las mesas: un **ecualizador**, un **compresor** y un **delay**.



El ecualizador: subir o bajar frecuencias

Todo sonido es una mezcla de frecuencias: la fundamental de una nota, y sus armónicos por encima. Un **ecualizador** (EQ) sube o baja rangos de ellas. No añade notas; cambia el color de las que hay. El EQ del mezclador tiene tres bandas, cada una de −12 dB a +12 dB:

Banda	Dónde	Forma	Qué vive allí
Graves	por debajo de 200 Hz	un estante	el peso de un bajo, un bombo, la mano izquierda de un piano
Medios	alrededor de 1 kHz	una campana, de una octava y media de ancho	el cuerpo de la mayoría de los instrumentos, y de las voces
Agudos	por encima de 4 kHz	un estante	el aire y el brillo: platillos, una voz con aire, un arco sobre una cuerda

Un **estante** (*shelf*) sube o baja todo lo que pasa de su frecuencia en la misma cantidad, como un escalón. Una **campana** sube o baja su frecuencia más que nada, y cada vez menos a cada lado. Los decibelios miden una proporción: +6 dB hace las ondas de una banda casi el doble de altas, −6 dB casi la mitad, 0 dB la deja como está.

Bajar es el uso más común. Cuando dos instrumentos tocan en el mismo registro, se tapan: los graves de un piano y el bajo se vuelven un solo sonido turbio. Quitar los graves al piano deja sitio al bajo, y cuando el bajo toca, el piano no suena más delgado.

En el hilo de audio cada banda es un **biquad**, un filtro de segundo orden, la pieza básica de la mayoría de los ecualizadores. Cada muestra de salida es una mezcla de las tres últimas muestras de entrada y las dos últimas de salida:

```
let output = b0 * input + b1 * self.inputs[0] + b2 * self.inputs[1]
    - a1 * self.outputs[0]
    - a2 * self.outputs[1];
```

Los cinco números `b0`, `b1`, `b2`, `a1` y `a2` deciden qué hace el filtro. `Biquad::set` los calcula para un estante o una campana, una frecuencia y una ganancia, con las fórmulas del *Audio EQ Cookbook* de Robert Bristow-Johnson, que usa casi todo programa de audio: una pendiente de 1 para los estantes y un Q de 1 para la campana. Los tres filtros van uno detrás de otro, graves, medios, agudos, y sus números solo se recalculan cuando cambian los ajustes, nunca en cada muestra. `set` conserva las muestras que el filtro recuerda, así que mover una banda mientras suena no hace clic. El mismo suelo de 10^{-20} que en la reverberación mantiene fuera los denormales.

El compresor: igualar lo fuerte que suena una parte

Algunas partes salen desiguales: una línea de bajo en la que unas notas saltan y otras se pierden. Un **compresor** baja un sonido cuando se vuelve fuerte:

- Por debajo de su **umbral** no hace nada.
- Por encima, baja el sonido en una cantidad que fija su **relación**. A 4:1, un sonido 8 dB por encima del umbral sale solo 2 dB por encima: cuatro decibelios de entrada por cada uno de salida. A 1:1 no baja nada, así que el compresor está apagado. A 20:1 casi nada pasa del umbral.
- Su **ganancia** (la «ganancia de compensación») sube toda la parte después, ya que las notas fuertes bajaron. La parte suena tan fuerte como antes, pero más pareja.

Esa regla es aritmética simple, así que vive en `lineup-core`, donde un test puede comprobarla sin ningún sonido:

```
pub fn reduction(&self, level: f32) -> f32 {
    let over = level - self.threshold;
    if over <= 0.0 || self.ratio <= 1.0 {
        0.0
    } else {
        over * (1.0 - 1.0 / self.ratio)
    }
}
```

A 4:1, 8 dB por encima del umbral se bajan 8 × (1 − ¼) = 6 dB, y salen 2 dB por encima.

Un compresor no baja cada onda. Sigue lo fuerte que ha sonado el sonido en los últimos milisegundos: baja deprisa cuando un sonido pasa del umbral (su **ataque**, 10 ms) y vuelve a subir más despacio cuando vuelve a estar por debajo (su **liberación**, 150 ms). Si siguiera cada onda, deformaría su forma y añadiría un zumbido. El `Follower` hace esto:

```
let peak = sample.abs();
let speed = if peak > self.level {
    self.attack
} else {
    self.release
};
self.level += (peak - self.level) * speed;
let level = (20.0 * self.level.log10()).max(FLOOR);
let decibels = compressor.gain - compressor.reduction(level);
10f32.powf(decibels / 20.0)
```

`attack` y `release` se calculan una vez a partir de sus tiempos, como `1 - e^(-1 / (segundos × frecuencia de muestreo))`: con ese número, el nivel recorre unos dos tercios del camino hacia una nueva intensidad en el tiempo dado. `20 × log10` convierte el nivel en decibelios, `reduction` dice cuánto bajar, y `10^(dB / 20)` convierte la respuesta de nuevo en un número por el que multiplicar la muestra. `.max(FLOOR)` mantiene el silencio, cuyo logaritmo es menos infinito, en −100 dB.

El delay: ecos a tiempo con la canción

Un **delay** vuelve a tocar el sonido un poco más tarde, y otra vez, cada eco más bajo que el anterior. Su **tiempo** es una figura de la canción, contada en pasos (semicorcheas):

```
pub const TIMES: [(u8, &'static str); 6] = [
    (1, "1/16"),
    (2, "1/8"),
    (3, "dotted 1/8"),
    (4, "1/4"),
    (6, "dotted 1/4"),
    (8, "1/2"),
];
```

Así los ecos caen en el ritmo de la canción, y siguen el tempo cuando cambia. La **corchea con puntillo** («1/8 con puntillo» en el menú) es la favorita: sus ecos caen tres semicorcheas después de cada nota, entre los pulsos, y una guitarra o una voz que toca corcheas suena como si tocara semicorcheas.

```
pub fn seconds(&self, tempo: Tempo) -> f32 {
    let step = tempo.seconds_per_beat() / crate::STEPS_PER_BEAT as f32;
    (f32::from(self.steps) * step).min(Delay::MOST_SECONDS)
}
```

Una corchea con puntillo a 120 BPM son 3 × 0,125 = 0,375 s. Un eco llega como mucho 2 segundos después del sonido: una blanca a 40 BPM serían 3 segundos, así que a tempos tan lentos los tiempos más largos se quedan en 2.

- La **Realimentación** dice cuánto de cada eco vuelve como el siguiente. Al 50 %, cada eco suena la mitad que el anterior; al 0 % hay uno solo. Se detiene en el 90 %, así que los ecos siempre se apagan.
- La **Mezcla** dice lo fuertes que son los ecos al lado del propio sonido. 0 % apaga el delay; 100 % hace el primer eco tan fuerte como la nota.

El delay recuerda el sonido que vuelve a tocar en un **buffer circular**: un array fijo que se escribe dando vueltas, donde la muestra de `delay_frames` atrás siempre va detrás de la que se está escribiendo:

```
let frames = line.len();
let read = (self.written + frames - self.delay_frames) % frames;
let echo = line.0[read];
line.0[self.written] = sample + echo * self.settings.delay.feedback;
self.written = (self.written + 1) % frames;
sample += echo * mix;
```

Lo que entra en la línea es el sonido más una parte del eco; esa parte vuelve otra vez `delay_frames` más tarde, y eso es la realimentación. Lo que sale es el sonido más el eco a la mezcla. Sumar `frames` antes del `%` impide que la resta baje de cero, lo que un `usize` no puede guardar.

Memoria hecha al otro lado

Dos segundos de sonido son 96 000 muestras a 48 kHz, una reserva grande, y el hilo de audio nunca reserva memoria. Así que el hilo de la ventana crea la `DelayLine` y la envía con los ajustes:

```
SetEffects {
    channel: usize,
    effects: lineup_core::Effects,
    line: Option<DelayLine>,
},
```

Envía una línea cada vez que envía ajustes con el delay activado, porque solo el hilo de audio sabe si ese canal ya tiene una. `ChannelEffects::set` se queda con la primera línea que recibe y devuelve cualquier otra, que vuelve por la cola de basura para liberarse del lado de la ventana:

```
let spare = match self.channels.get_mut(channel) {
    Some(state) => state.effects.set(effects, line, tempo),
    None => line,
};
if let Some(spare) = spare {
    self.throw_away(Garbage::DelayLine(spare));
}
```

Un canal conserva su línea una vez que la tiene. Cuando su delay se apaga y se vuelve a encender, `set` llena antes la línea de silencio, o volverían los ecos de la vez anterior. Cuando cambia el tempo, el motor llama a `set_tempo` en los efectos de cada canal, así que los ecos se mueven con él.

`process` mira primero si algún efecto está activado. Si ninguno lo está, devuelve la muestra intacta: un canal sin efectos suena exactamente como sonaría sin este código, y no cuesta nada.

Medidores de nivel: del hilo de audio a la pantalla

Un medidor muestra lo fuerte que suena un canal **ahora**. El hilo de audio lo sabe, porque hace el sonido; el hilo de la ventana dibuja el medidor. Van a velocidades distintas (el hilo de audio llena un bloque de unos cientos de muestras muchas veces por segundo, la ventana dibuja unos 60 frames por segundo), y el hilo de audio nunca debe esperar a la ventana, así que un `Mutex` queda descartado.

Picos en atómicos

El hilo de audio guarda la muestra más fuerte de cada canal durante un bloque (`channel.peak` más arriba). Al final del bloque sube un pico compartido hasta ella, si es más fuerte. Una vez por frame, el medidor toma el pico compartido y lo devuelve al silencio. Así no se pierde ningún pico, se alineen como se alineen bloques y frames, y ningún lado espera:

```
/// The peaks since the UI last took them, as the bits of `f32`s.
///
/// For numbers that are not negative, comparing their bits compares the
/// numbers, so `fetch_max` on the bits keeps the louder peak.
pub struct Levels {
    channels: [AtomicU32; MAX_CHANNELS],
    master: [AtomicU32; 2],
}
```

```
fn raise(slot: &AtomicU32, peak: f32) {
    // `peak > 0.0` is false for NaN too, which must not get in.
    if peak > 0.0 {
        slot.fetch_max(peak.to_bits(), Ordering::Relaxed);
    }
}

fn take(slot: &AtomicU32) -> f32 {
    f32::from_bits(slot.swap(0, Ordering::Relaxed))
}
```

La biblioteca estándar de Rust no tiene `AtomicF32`. Pero un `f32` son 32 bits, y `to_bits` y `from_bits` lo convierten en un `u32` y de vuelta sin cambiar un bit. Mejor aún: para números **no negativos**, el orden de los bits como enteros sin signo es el orden de los números, porque el exponente está en los bits altos y la mantisa debajo. Así que `fetch_max` sobre los bits, un «quédate con el mayor» atómico, se queda con el pico más fuerte. Los picos son valores absolutos, nunca negativos, y el `if` deja fuera los NaN (sus bits se compararían como mayores que cualquier número).

`swap(0, …)` lee el valor y pone 0 (los bits de 0.0) en su lugar en un solo paso, así que un pico que llegue entre una lectura y un reinicio no se puede perder. `Ordering::Relaxed` basta porque cada pico va solo: nada más en memoria tiene que verse en un orden determinado junto con él.

`std::array::from_fn(|_| AtomicU32::new(0))` construye los arrays, ya que un `AtomicU32` no es `Copy` y `[AtomicU32::new(0); N]` no compilaría.

Decibelios, y un medidor que cae despacio

El oído percibe la intensidad por proporciones, no por diferencias, así que el medidor se dibuja en **decibelios**, una escala de proporciones:

```
decibelios = 20 × log10(nivel)      1.0 → 0 dB (el máximo)
                                    0.5 → −6 dB
                                   0.25 → −12 dB
                                  0.004 → −48 dB (el suelo del medidor)
```

Cada 6 dB menos reducen el nivel a la mitad, en cualquier punto de la escala. Un medidor se ilumina desde −48 dB a la izquierda hasta 0 dB a la derecha: verde, ámbar desde −6 dB, rojo en el máximo y por encima. Los medidores del principal miden después del volumen principal y antes de `tanh`, así que el rojo en el principal significa que el redondeo está actuando. Al pasar el ratón por un medidor, dice su nivel en decibelios.

Un medidor que siguiera cada pico exactamente parpadearía demasiado rápido para leerlo. Así que **salta hacia arriba** a un pico nuevo enseguida y **cae** 24 dB por segundo:

```
pub fn update(&mut self, levels: &Levels, channels: usize, seconds: f32) {
    let fall = gain(-FALL_DB_PER_SECOND * seconds);
    self.channels.resize(channels, 0.0);
    for (index, shown) in self.channels.iter_mut().enumerate() {
        *shown = levels.take_channel(index).max(*shown * fall);
    }
    for (shown, peak) in self.master.iter_mut().zip(levels.take_master())
{
        *shown = peak.max(*shown * fall);
    }
}
```

`seconds` es el tiempo desde el frame anterior. Caer 24 dB por segundo es multiplicar por 10^(−24 × segundos / 20) en cada frame, sea cual sea la frecuencia de frames. Medio segundo después de un pico, el medidor lo muestra 12 dB más abajo, a una cuarta parte, más o menos.

egui solo dibuja un frame cuando pasa algo. Mientras un medidor sigue cayendo, el mezclador pide el siguiente frame enseguida (`request_repaint`); cuando todos los medidores están apagados, vuelve a mirar cada décima de segundo (`request_repaint_after`), lo que basta para captar las notas de un teclado MIDI tocado con la canción parada.

Orden de partitura: la batería junta

El rack de canales guarda los canales en el orden en que se añadieron. Un estilo que trae un aro, un bajo, un cencerro y una conga los añade al final, así que una lista en el orden del rack desperdiga la batería alrededor del piano y del bajo. Una partitura de director nunca hace eso: cada instrumento tiene su lugar, el mismo en todas las partituras, para que quien la lee encuentre las flautas arriba y los contrabajos abajo sin buscar. El mezclador lista los canales en ese orden, por secciones con un título cada una:

Sección	Instrumentos, de arriba abajo
Maderas	flautín, flautas, oboes, corno inglés, clarinetes, saxofones, fagotes
Metales	trompas, trompetas, trombones, tuba
Percusión	timbales; glockenspiel, xilófono, vibráfono, marimba, campanas tubulares; el resto
Batería	percusión de mano, luego la batería
Arpa y guitarras	arpa, guitarras, banjo, koto, bajo eléctrico
Teclados	celesta, piano, clavecín, órgano, acordeón y bandoneón
Sintetizadores	solistas, colchones y efectos de sintetizador; un Lineup Osc sin instrumento
Voces	una voz solista, luego el coro
Cuerdas	secciones de cuerda, violines, violas, violonchelos, contrabajos

Las trompas van encima de las trompetas aunque suenan más graves, porque se funden con las maderas y durante mucho tiempo se escribieron junto a ellas. Las cuerdas van al final, así que los contrabajos, la parte más grave de la orquesta, cierran la página; un coro va justo encima de ellas, como en un réquiem o un oratorio.

La batería sigue el **pentagrama de batería**, donde cada tambor tiene su línea o su espacio: los platillos en la línea superior y por encima (el crash el más alto, luego el charles, luego el ride), el tom agudo en el espacio superior, la caja en el tercer espacio, el tom de pie debajo y el bombo en el espacio inferior. Así que la sección **Batería** se lee Crash, Charles, Charles abierto, Ride, Tom agudo, Caja, Aro (un golpe de aro, escrito donde va la caja), Tom grave, Bombo. La percusión de mano (cencerro, shaker, palmas, conga) suele tener sus propios pentagramas encima de la batería, así que va primero.

Solo el mezclador cambia su orden. El rack de canales, el piano roll y el archivo guardado mantienen los canales como se añadieron, y nada suena distinto.

Dónde va un canal

`score_order.rs` está en `lineup-core`, sin código de interfaz, así que se prueba con un simple `cargo test`. Un `Section` es un enum en orden de partitura:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Section {
    Woodwinds,
    Brass,
    Percussion,
    Drums,
    HarpAndGuitars,
    Keyboards,
    Synths,
    Voices,
    Strings,
}
```

Derivar `PartialOrd` y `Ord` en un enum ordena sus variantes tal como están escritas: `Section::Woodwinds < Section::Brass`. El orden de la partitura es el orden del código fuente.

El mezclador no mira el nombre de un canal para ubicarlo, sino su instrumento, el programa General MIDI (SoundFonts y General MIDI). Un canal llamado «Melodía» que toca una flauta va con las maderas. Los números de programa ya vienen casi siempre por familias, pero no siempre como los quiere una partitura: General MIDI pone los timbales con las cuerdas y la celesta con las láminas, y la partitura pone los timbales los primeros de la percusión y la celesta con los teclados. Así que cada sección lista sus programas, de arriba abajo, y `place` encuentra la sección de un canal y su puesto en ella:

```
pub fn place(channel: &Channel) -> (Section, usize) {
    match &channel.kind {
        ChannelKind::Sampler(SampleSource::Builtin(drum)) => {
            let rank = DRUMS.iter().position(|d| d ==
drum).unwrap_or(DRUMS.len());
            (Section::Drums, rank)
        }
        // A sound file usually takes a drum's row in the rack.
        ChannelKind::Sampler(SampleSource::File(_)) => (Section::Drums,
DRUMS.len() + 1),
        ChannelKind::MidiOut {
            channel: DRUM_CHANNEL,
        } => (Section::Drums, DRUMS.len()),
        ChannelKind::Synth | ChannelKind::MidiOut { .. } => match
channel.program {
            Some(program) => program_place(program),
            // The Lineup Osc, or an outside synth with no instrument
named.
            None => (Section::Synths, usize::MAX),
        },
    }
}
```

`ChannelKind::MidiOut { channel: DRUM_CHANNEL }` solo encaja con canales de salida MIDI cuyo campo es igual a la constante `DRUM_CHANNEL` (9, que se muestra como 10): una `const` puede ir en un patrón como un número literal. El brazo siguiente, `MidiOut { .. }`, atrapa cualquier otro canal MIDI; `match` prueba los brazos desde arriba, así que el canal de batería nunca llega hasta ahí.

```
fn program_place(program: u8) -> (Section, usize) {
    Section::ALL
        .into_iter()
        .find_map(|section| {
            let rank = section.programs().iter().position(|&p| p ==
program)?;
            Some((section, rank))
        })
        .unwrap_or((Section::Synths, usize::MAX))
}
```

`find_map` ejecuta el closure en cada sección y se detiene en la primera que devuelve `Some`. Dentro, el `?` después de `position` devuelve `None` desde el closure cuando el programa no está en esa sección, lo que le dice a `find_map` que pruebe la siguiente. Un test comprueba que cada uno de los 128 programas está en exactamente una lista, y otro que cada tambor integrado tiene un lugar en la batería.

Ordenar y agrupar

```
pub fn score_order(channels: &[Channel]) -> Vec<(Section, Vec<usize>)> {
    let mut order: Vec<usize> = (0..channels.len()).collect();
    order.sort_by_key(|&index| place(&channels[index]));
    let mut sections: Vec<(Section, Vec<usize>)> = Vec::new();
    for index in order {
        let section = place(&channels[index]).0;
        match sections.last_mut() {
            Some((last, indices)) if *last == section =>
indices.push(index),
            _ => sections.push((section, vec![index])),
        }
    }
    sections
}
```

Ordena los **números** de los canales, no los canales: cada fila del mezclador tiene que saber qué canal es, para cambiar el volumen de ese canal y poner en solo el correcto. La clave es el par `(Section, usize)`, y las tuplas se comparan campo a campo, así que decide primero la sección y el puesto desempata. `sort_by_key` es un ordenamiento *estable*: los canales con la misma clave, como «Melodía», «Voz 2» y «Voz 3» tocando todos violines, conservan el orden del rack.

Luego el bucle recorre la lista ordenada y empieza un grupo nuevo cada vez que cambia la sección. `last_mut` da una referencia mutable al último grupo, si lo hay; el `if` después del patrón (una *guarda de match*) solo toma el primer brazo cuando ese grupo es de la misma sección.

La ventana

`mixer_window` dibuja a partir de copias: `mix` es una copia del `Mixer` del proyecto, y `levels` una copia de sus canales. Los deslizadores cambian las copias, y cuando la ventana termina la aplicación las compara con el proyecto y hace un solo `edit` con lo que haya cambiado, así que un arrastre es un paso de deshacer (Deshacer y rehacer). El volumen principal pasa en cambio por `set_master_volume`, que reabre el último paso si también era el volumen principal, así que moverlo varias veces es un solo paso (Varios clics, un paso).

```
for (section, indices) in score_order(&levels) {

ui.label(egui::RichText::new(tr(section.name())).color(theme::ACCENT));
    ui.end_row();
    for index in indices {
        let channel = &mut levels[index];
        // name, instrument, meter, volume, pan, send, effects, mute ...
        if ui
            .selectable_label(mix.solo == Some(index), tr("S"))
            // ...
            .clicked()
        {
            mix.solo = if mix.solo == Some(index) {
                None
            } else {
                Some(index)
            };
        }
        ui.end_row();
    }
}
```

La columna **Instrumento** es un menú para un canal de sintetizador o de salida MIDI, con las mismas opciones que el del rack: el rack y el mezclador llaman los dos a `channel_rack::instrument_choices`. Un tambor integrado no tiene instrumento que elegir, pero su menú puede enviarlo por MIDI (MIDI); la celda de un archivo de muestra es texto simple. El menú se dibuja dentro de closures que ya toman prestadas partes de la aplicación, así que no puede cambiar el proyecto mientras se dibuja. Guarda lo que elegiste en una lista, y la aplicación lo aplica cuando la ventana termina:

```
for action in picked {
    self.apply(action);
}
```

Es el mismo patrón que usan el rack de canales y la hoja de acordes: el código de la interfaz dice **qué** hiciste, como un valor (`RackAction::SetProgram { channel, program }`), y un solo sitio de la aplicación lo hace.

El botón **Efectos** dice qué efectos están activados («EQ, delay»), o «ninguno», y abre un menú con los tres. Ese menú se crea con `PopupCloseBehavior::CloseOnClickOutside`, así que sigue abierto mientras arrastras sus deslizadores, hasta que haces clic en otro sitio. Los deslizadores del ecualizador vuelven a 0 dB con un doble clic, como el balance vuelve al centro.

La ventana se ajusta a sus columnas, y se vuelve a ajustar cuando cambian los canales o sus instrumentos, porque un nombre de instrumento más corto puede estrechar

una columna (Ventanas del tamaño de lo que muestran).

Cómo se prueba

- `reverb.rs`: `bypass_preserves_samples_exactly` (desactivada, las muestras salen idénticas bit a bit); `impulse_has_a_stereo_tail_that_decays_and_can_be_cleared` a 22,05, 48 y 96 kHz (silencio durante el predelay, una cola distinta a izquierda y derecha, menos de una milésima de la energía del primer medio segundo pasado un segundo, y silencio después de `clear`); `a_tail_ends_in_zeros_not_denormals`.
- `effects.rs` de `lineup-audio` hace pasar ondas senoidales por los efectos y mide lo que sale: Graves a +6 dB sube 50 Hz en 6 dB y deja 5 kHz igual; un compresor a −20 dB y 4:1 baja 15 dB una onda al máximo, 20 dB por encima, y deja pasar una suave; un delay de corchea a 120 BPM con un 50 % de realimentación repite un clic a los 0,25 s a nivel completo, luego a la mitad, luego a un cuarto; una línea se guarda una vez y una segunda se devuelve.
- `effects.rs` de `lineup-core`: la aritmética de la reducción, el tiempo del delay a 120 BPM y su límite de 2 segundos, ajustes fuera de rango, el resumen del botón («EQ, comp»), y que unos efectos sin cambios se guardan como `()`.
- `render.rs` hace pasar un clic por todo el motor: `a_panned_channel_plays_more_on_its_side` con cinco balances, contra la tabla de arriba; `a_channel_sent_nowhere_plays_dry_in_a_room` (un envío de 0 suena tan fuerte como con la reverberación desactivada y no deja cola); `a_channel_with_a_delay_echoes_in_time_with_the_song`; `the_meters_see_each_channel_and_the_master` (un clic a volumen 0,5, a la izquierda, con el principal a 0,8: el medidor del canal marca 0,25, el izquierdo del principal 0,2 y el derecho 0).
- `levels.rs` y `meters.rs`: el pico más fuerte se guarda hasta que se toma y un NaN nunca entra; un medidor salta hacia arriba, cae 12 dB en medio segundo y se apaga por debajo del suelo.
- `score_order.rs`: cada programa y cada tambor tienen exactamente un lugar; la batería añadida después de la banda se junta con el resto de la batería; una orquesta se lee de las flautas a los contrabajos; el canal MIDI de batería y los archivos de sonido cuentan como batería; los canales en el mismo lugar conservan el orden del rack.
- En la aplicación: `the_mixer_pans_a_channel_and_sets_its_reverb_send`, `the_mixer_sets_a_channels_effects` (abre el menú, elige «dotted 1/4» y luego «No effects»), `a_click_on_an_instrument_in_the_mixer_changes_it` (y un Deshacer recupera el piano),

`the_mixer_keeps_the_drums_together_and_each_one_still_solos_alone`, y dos exportaciones: `an_exported_song_keeps_each_channel_where_the_mixer_put_it` y `an_exported_song_plays_each_channels_effects`.

- En `file.rs`: `pan_and_reverb_sends_need_format_twenty_six` y `effects_need_format_twenty_eight`, cada uno con un archivo editado a mano que debe rechazarse.

MIDI: mensajes, dispositivos, grabación y archivos

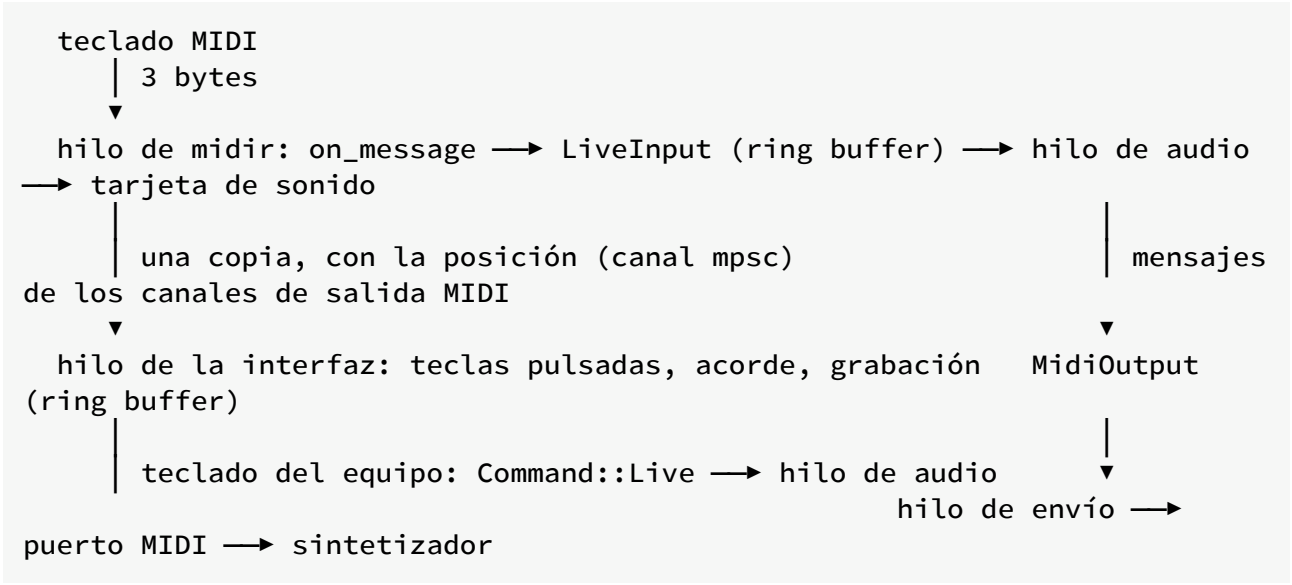
Archivos: `crates/lineup-core/src/midi.rs`, `crates/lineup-core/src/midi_file.rs`, `crates/lineup-core/src/midi_file/read.rs`, `crates/lineup-core/src/midi_file/import.rs`, `crates/lineup-core/src/record.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-audio/src/instrument.rs`, `crates/lineup-app/src/midi.rs`, `crates/lineup-app/src/app/live.rs` y `crates/lineup-app/src/channel_rack.rs`

MIDI es como se hablan los teclados, los sintetizadores y los programas de música. Es el mismo desde 1983: unos pocos bytes que dicen «esta tecla bajó, con esta fuerza» o «esta tecla subió». Ferrous Lineup lo habla en los dos sentidos:

- las teclas de un **teclado MIDI**, o del propio teclado del equipo dispuesto como un piano, tocan el *canal seleccionado*, el que muestra el piano roll, y la barra de estado nombra el acorde que forman;
- **grabar** convierte las teclas tocadas mientras un patrón se repite en notas de ese patrón;
- la **salida MIDI** envía canales a un sintetizador externo en lugar de a los altavoces: todos a la vez, o los que elijas;
- los **archivos MIDI** (`.mid`) se escriben a partir de una canción o de un patrón, y se leen de vuelta en un proyecto nuevo.

Cómo se usa todo esto está en las páginas del manual Teclados y sintetizadores MIDI, Abrir archivos MIDI y grabaciones y Exportar. Este capítulo sigue a una tecla en su camino: los bytes, los hilos que cruza, la nota en que se convierte y el archivo que la guarda.

Cuatro hilos se reparten el trabajo. Una tecla de un teclado MIDI hace este camino:



Cuatro ideas sostienen el diseño:

1. **Las teclas se saltan la interfaz.** La interfaz dibuja una vez por frame, y una tecla podría esperar allí un frame entero antes de que pase nada. Así que una tecla va del hilo MIDI directamente al hilo de audio, y suena dentro de un bloque de audio, unos pocos milisegundos. La interfaz recibe una copia, para mostrarla y grabarla.
2. **El hilo de audio sigue sin esperar nunca.** Enviar un mensaje MIDI es una llamada al sistema, que puede tardar lo que sea. Así que el hilo de audio deja los mensajes en una cola, y un hilo propio los envía.
3. **Un solo reloj para todo.** El hilo de audio publica por dónde va en el patrón, hasta una fracción de paso. Cada tecla se marca con esa posición al llegar, y grabar solo tiene que redondear.
4. **Un archivo se lee como datos que pueden estar mal.** Un archivo MIDI de internet puede estar cortado o dañado. Cada lectura puede fallar sin un panic, y la importación se queda con lo que pudo leer y lo dice.

Tres bytes por tecla

Un teclado MIDI dice casi todo con mensajes de tres bytes. El primero, el *byte de estado*, dice qué clase de mensaje es y en cuál de los 16 *canales*. Los otros dos son *bytes de datos*. Para una tecla que baja:

```
  1001 0000   0011 1100   0110 0100
  │    │      │           │
  │    │      │           └ velocidad 100: con qué fuerza se pulsó la
tecla
  │    │      └ nota 60: Do central (Do4)
  │    └ canal 0 (la gente lo llama canal 1)
  └ 1001 = note-on
```

Un byte de estado siempre tiene el bit superior a 1 (de 128 en adelante), y un byte de datos nunca (de 0 a 127). Por eso un número de nota MIDI y una velocidad nunca pasan de 127, y por eso `Note` usa los mismos números: Do4 es 60 en Ferrous Lineup porque es 60 en MIDI.

Hay muchas clases de mensaje: potenciómetros y pedales («control change»), pitch bends, tics de reloj para mantener las máquinas a tiempo. La aplicación necesita tres:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum MidiMessage {
    NoteOn {
        channel: u8,
        pitch: Note,
        /// How hard the key was struck, 1 to 127.
        velocity: u8,
    },
    NoteOff {
        channel: u8,
        pitch: Note,
    },
    /// From now on, `channel` plays General MIDI program `program`, 0 to
    /// 127. Only sent, never read: a keyboard choosing its own sound does
    /// not change the channel it plays.
    ProgramChange {
        channel: u8,
        program: u8,
    },
}

/// The kinds of message, as the top four bits of the status byte.
const NOTE_OFF: u8 = 0x80;
const NOTE_ON: u8 = 0x90;
const CONTROL_CHANGE: u8 = 0xB0;
const PROGRAM_CHANGE: u8 = 0xC0;
```

El byte de estado guarda dos cosas a la vez, así que el código lo divide con operaciones de bits, las mismas que usan los conjuntos de clases de altura: `status & 0xF0` se queda con los cuatro bits superiores, la clase, y `status & 0x0F` con los cuatro inferiores, el canal.

Leer un mensaje: patrones de slice

```
pub fn parse(bytes: &[u8]) -> Option<MidiMessage> {
    let [status, pitch, velocity, ..] = *bytes else {
        return None;
    };
    // Data bytes have the top bit clear; anything else is malformed.
    if pitch > 127 || velocity > 127 {
        return None;
    }
    let channel = status & 0x0F;
    let pitch = Note::new(pitch);
    match status & 0xF0 {
        NOTE_ON if velocity > 0 => Some(MidiMessage::NoteOn {
            channel,
            pitch,
            velocity,
        }),
        NOTE_ON | NOTE_OFF => Some(MidiMessage::NoteOff { channel, pitch
}),
        _ => None,
    }
}
```

`let [status, pitch, velocity, ..] = *bytes else` es un **patrón de slice**. Un slice puede tener cualquier longitud, así que un patrón para él describe una *forma*: «al menos tres elementos; llama a los tres primeros `status`, `pitch` y `velocity`, y `..` para lo que siga». Si el slice es más corto, el patrón no encaja y el bloque `else` retorna. Sin patrones de slice, esto sería una comprobación de longitud y tres indexaciones, cada una de las cuales podría entrar en pánico si la comprobación estuviera mal. El `*` encaja con el slice mismo, no con la referencia a él; se permite porque las partes que se sacan son `u8`, que son `Copy`.

El `match` usa dos cosas más. `NOTE_ON` y `NOTE_OFF` son constantes, y una constante en un patrón significa «igual a este valor», no «una variable nueva». Y `NOTE_ON if velocity > 0` tiene una **guarda de match**: el brazo solo encaja cuando además se cumple la condición. Un note-on con velocidad 0 cae al brazo siguiente y se convierte en un note-off. Muchos teclados envían de verdad eso en lugar de un note-off: una serie de mensajes de la misma clase puede omitir el byte de estado repetido («running status»), lo que ahorraba tiempo en los cables lentos de 1983.

`parse` solo lee notas. Los program changes solo se envían, y cualquier otro mensaje, desde un pedal de sustain hasta un tic de reloj, da `None`.

Escribir uno: dos o tres bytes

Una nota son tres bytes, un program change dos. `bytes` devuelve un valor pequeño que guarda tres y dice cuántos cuentan:

```
/// The bytes of one message: two or three of them, as a slice.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct MidiBytes {
    bytes: [u8; 3],
    len: usize,
}

impl std::ops::Deref for MidiBytes {
    type Target = [u8];

    fn deref(&self) -> &[u8] {
        &self.bytes[..self.len]
    }
}
```

`Deref` es el trait detrás del operador `*`. Implementarlo dice «un `MidiBytes` se puede usar como un `[u8]`»: `*bytes` da el slice, y también lo da pasar `&bytes` donde se espera un `&[u8]`, porque Rust sigue `Deref` por su cuenta ahí (la *coerción por deref*). Así, el código que envía puede pasar `&message.bytes()` a un puerto, un test puede comparar `*off.bytes()` con `[0x80, 60, 64]`, y no se reserva memoria: los tres bytes viven en el valor, en la pila.

```rust
pub fn bytes(self) -> MidiBytes {
    let (bytes, len) = match self {
        MidiMessage::NoteOn {
            channel,
            pitch,
            velocity,
        } => (
            [
                NOTE_ON | (channel & 0x0F),
                pitch.midi(),
                velocity.clamp(1, 127),
            ],
            3,
        ),
        // Velocity 64 is the usual "no particular speed" for a release.
        MidiMessage::NoteOff { channel, pitch } => {
            ([NOTE_OFF | (channel & 0x0F), pitch.midi(), 64], 3)
        }
        MidiMessage::ProgramChange { channel, program } => {
            ([PROGRAM_CHANGE | (channel & 0x0F), program & 0x7F, 0], 2)
        }
    };
    MidiBytes { bytes, len }
}
```

El `clamp(1, 127)` importa. Un note-on con velocidad 0 lo leería el sintetizador como un note-off, así que una nota tocada «lo más suave posible» no sonaría en absoluto. `& 0x0F` y `& 0x7F` evitan que un número de canal o de programa equivocado se derrame en los bits de otro campo.

Hay un mensaje que se escribe y nunca se lee: el control change 123, «All Notes Off». Le dice a un sintetizador que pare todas las notas de un canal:

```rust
pub fn all_notes_off(channel: u8) -> [u8; 3] {
    [CONTROL_CHANGE | (channel & 0x0F), 123, 0]
}
```

La aplicación lo envía en los 16 canales cada vez que suelta un sintetizador, porque un note-off perdido por el camino dejaría una nota sonando para siempre: cuando se elige otro puerto de salida, y cuando desaparece el hilo de audio que debía los note-offs (Dos hilos).

El camino de entrada de una tecla

La mayoría de las notas vienen del secuenciador, que vive en el hilo de audio y conoce la duración de cada nota antes de que empiece. Una tecla de un teclado es distinta en

dos cosas: viene de fuera, en un hilo que no es de Ferrous Lineup, y nadie sabe cuánto tiempo se mantendrá pulsada.

Por qué las teclas se saltan la interfaz

El camino obvio para una tecla sería el que siguen todos los demás cambios: la interfaz se entera y envía un `Command` al hilo de audio. Pero la interfaz solo se ejecuta una vez por frame. A 60 frames por segundo eso son hasta 17 ms de espera, y en un equipo lento mucho más, antes de que el hilo de audio se entere siquiera de la tecla. Un pianista lo nota.

El hilo de audio se ejecuta mucho más a menudo: una vez por *bloque*, cada pocos milisegundos. Así que una tecla de un teclado MIDI va directamente allí, y la interfaz recibe una copia después, solo para mostrarla y grabarla:

```
  hilo de midir ──► LiveInput ──► hilo de audio          (la toca)
        └──────────► canal mpsc ──► hilo de la interfaz    (la muestra y la
graba)
```

El teclado del equipo no puede hacer esto: egui lee el teclado, y egui se ejecuta en el hilo de la interfaz. Así que las teclas escritas toman el camino lento, como un `Command::Live(MidiMessage)`. En el hilo de audio las dos acaban en el mismo método, `play_live`.

Un productor por cola: `LiveInput`

La cola de comandos es un ring buffer del crate `rtrb` (Dos hilos), y su `Producer` pertenece al `AudioEngine`, en el hilo de la interfaz. rtrb es de *un productor y un consumidor*: un hilo mete, un hilo saca, y eso es lo que le permite funcionar sin cerrojos. El hilo de midir es un segundo hilo que quiere meter, así que necesita un ring buffer propio:

```
pub struct LiveInput {
    notes: rtrb::Producer<MidiMessage>,
    position: Arc<AtomicU64>,
}

impl LiveInput {
    /// Plays (or lets go of) a key on the live channel. Returns `false`
    /// if the queue was full and the key was lost.
    pub fn send(&mut self, message: MidiMessage) -> bool {
        self.notes.push(message).is_ok()
    }

    /// Where playback is, in steps (see [`Sequencer::position`]), or
    /// `None` when stopped. Recording uses it to place the key.
    pub fn position(&self) -> Option<f64> {
        load_position(&self.position)
    }
}
```

Como solo puede haber un productor, esto no puede ser un método de `AudioEngine` que cualquiera pueda llamar. Es un *valor*, y quien lo posee es el único productor. El motor lo crea al arrancar y lo guarda en un `Option` hasta que alguien lo pide:

```
pub fn take_live_input(&mut self) -> Option<LiveInput> {
    self.live_input.take()
}
```

`Option::take` saca el valor y deja `None` en su lugar, así que la segunda llamada no recibe nada. El sistema de tipos hace imposible romper la regla «un productor»: `LiveInput` no implementa `Clone`, así que no se puede copiar, solo mover. `MidiOutput`, la salida de los canales de salida MIDI, funciona igual desde el otro extremo: guarda el `Consumer` de la cola, y `take_midi_output` lo entrega una vez.

En el hilo de audio, el consumidor también es un `Option`, porque el hilo de audio que exporta una canción no tiene teclado:

```
while let Some(message) = self.live_input.as_mut().and_then(|keys|
keys.pop().ok()) {
    self.play_live(message);
}
```

Esa línea empaqueta tres pasos. `self.live_input` es un `Option<Consumer>`, y llamar a `pop` necesita un `&mut Consumer`. `as_mut()` convierte `&mut Option<Consumer>` en `Option<&mut Consumer>`, prestando lo de dentro en lugar de sacarlo. `and_then` ejecuta el closure solo si hay consumidor, y el `Option` del propio closure (de

`pop().ok()`: `Some(key)`, o `None` cuando la cola está vacía) es el resultado. Así que el bucle sigue mientras haya consumidor *y* tenga una tecla.

Dónde en el patrón: la posición

Grabar necesita saber *cuándo* llegó cada tecla, como un lugar en el patrón: «a mitad del paso 4». El secuenciador lo sabe, porque cuenta hacia atrás los frames hasta el siguiente paso (El secuenciador):

```
pub fn position(&self) -> f64 {
    if self.frames_until_next <= 0.0 {
        return self.next_step as f64;
    }
    let current = (self.next_step + self.length - 1) % self.length;
    let passed = 1.0 - self.frames_until_next / self.frames_per_step;
    current as f64 + passed.clamp(0.0, 1.0)
}
```

`next_step` es el paso a punto de dispararse, así que el que suena es el anterior. `next_step - 1` bajaría de cero en el paso 0, y un `usize` no puede (haría panic en una compilación de depuración). Sumar `length` primero y tomar el resto retrocede un paso *alrededor del bucle*: del paso 0 al último. `passed` es cuánto de ese paso ha pasado, de 0 a 1.

El hilo de audio la guarda después de cada bloque, como los bits del `f64` en un `AtomicU64`, ya que no hay float atómico (Dos hilos). «Parado» es `NO_POSITION = u64::MAX`; como float, esos bits son un NaN («no es un número»), que ninguna posición es nunca. `LiveInput` guarda un clon del mismo `Arc`, así que el hilo de midir puede leer la posición en el mismo momento en que llega una tecla. Para entonces tiene como mucho un bloque de antigüedad, una centésima de segundo; un paso a 120 BPM dura un octavo de segundo, y grabar redondea al paso más cercano de todos modos.

Una tecla en el hilo de audio

Una nota de la canción recibe un gate, el número de frames que se mantiene. Una tecla no tiene duración conocida, así que recibe `HELD`, un gate que nunca se agota: solo `note_off` la termina (Instrumentos). `play_live` las empieza y las termina:

```
fn play_live(&mut self, message: MidiMessage) {
    match message {
        MidiMessage::NoteOn {
            pitch, velocity, ..
        } => {
            // A key that is already down (on both keyboards at once,
            // say) comes up first.
            self.release_live(pitch);
            if let Some(channel) = self.live_channel
                && let Some(state) = self.channels.get_mut(channel)
            {
                state
                    .instrument
                    .note_on(pitch, velocity, HELD, self.sample_rate);
                self.live_keys[usize::from(pitch.midi())] = Some(channel);
            }
        }
        MidiMessage::NoteOff { pitch, .. } => self.release_live(pitch),
        MidiMessage::ProgramChange { .. } => {}
    }
}
```

Las teclas tocan el *canal en vivo*, que la interfaz fija con `Command::SetLiveChannel` cada vez que cambia el canal seleccionado. Pero la selección puede cambiar mientras una tecla está pulsada. Si la tecla subiera entonces en el canal nuevo, la nota del antiguo sonaría para siempre. Así que el hilo de audio recuerda, para cada una de las 128 notas, en qué canal bajó su tecla:

```
live_keys: [Option<usize>; 128],

fn release_live(&mut self, pitch: Note) {
    if let Some(channel) =
self.live_keys[usize::from(pitch.midi())].take()
        && let Some(state) = self.channels.get_mut(channel)
    {
        state.instrument.note_off(pitch);
    }
}
```

Un array de 128 `Option` es un mapa de nota a canal que nunca reserva memoria, y `take()` lee una entrada y la vacía de una vez. El `if let ... && let ...` es una **let chain**: los dos patrones tienen que encajar para que se ejecute el bloque.

Los canales se conocen por su posición en la lista, e insertar o quitar un canal mueve los que vienen detrás. Un «canal 2» recordado podría ser entonces otro canal, o ninguno. Así que antes de que cambie la lista, `release_live_keys` suelta todas las teclas pulsadas. El test `removing_channels_lets_go_of_their_keys` quita el canal 0 mientras hay una tecla pulsada en el canal 1, que entonces *pasa a ser* el canal 0.

Las notas en vivo suenan como una vista previa: también en un canal silenciado, y esté o no tocando el secuenciador. Silenciar detiene las notas de la canción en un canal, no la tecla que estás pulsando (`muting_a_channel_stops_its_notes_but_not_keys_held_down`). Detener la reproducción suelta todo, teclas incluidas.

midir: puertos y conexiones

midir es el crate que habla con la capa MIDI de cada sistema: ALSA en Linux, Core MIDI en macOS, Windows Multimedia en Windows. Está hecho como `cpal`: listar los puertos, abrir una conexión a uno, y la conexión queda abierta mientras se conserve el valor. Ferrous Lineup lo envuelve en un struct `Midi`, en `lineup-app/src/midi.rs`.

Si hay MIDI o no se averigua probando: crear un `MidiInput` falla en un sistema sin capa MIDI, y el error se guarda para mostrarlo en el menú. En Linux eso es un equipo sin el *secuenciador* de ALSA (el módulo del kernel `snd-seq`, que crea `/dev/snd/seq`), y la ayuda del menú lo dice.

Los puertos se vuelven a encontrar por un id que da midir, un `String` que no cambia mientras el dispositivo esté enchufado, porque la lista de puertos cambia a medida que los dispositivos van y vienen. El menú MIDI busca puertos cada vez que se abre (`refresh_ports`), y deja fuera los de Ferrous Lineup: en Linux, el puerto desde el que envía la aplicación aparece también como *entrada*, y elegirlo haría dar vueltas en círculo a cada nota.

```
fn is_own(name: &str) -> bool {
    name.starts_with(&format!("{CLIENT_NAME}:"))
}
```

Los dos puntos importan: un sintetizador llamado «Ferrous Lineupish Synth» no es la aplicación (el test `its_own_ports_are_left_out` lo comprueba).

Conectar consume, cerrar devuelve

```
let mut midi_in = MidiInput::new(CLIENT_NAME).map_err(|err|
err.to_string())?;
// Clock ticks, "I am still here" messages and system exclusive
// data are not notes. Leaving them out saves waking up for them.
midi_in.ignore(Ignore::All);
let found = midi_in
    .find_port_by_id(&port.id)
    .ok_or_else(|| crate::i18n::trf("{} is not connected any more", &
[&port.name]))?;
let link = InputLink {
    live: self.live.clone(),
    to_ui: self.to_ui.clone(),
    ctx: self.ctx.clone(),
    snap: Arc::clone(&self.snap),
    sounding: Vec::new(),
};
let connection = midi_in
    .connect(&found, "input", on_message, link)
    .map_err(|err| err.to_string())?;
```

`connect` toma `self`, no `&self`: el `MidiInput` se consume y vive dentro de la conexión desde entonces. `close` lo devuelve, con los datos. Es la propiedad usada como máquina de estados: un `MidiInput` conectado no se puede conectar dos veces, porque ya no lo tienes.

`on_message` es una función normal, y `link` son los datos que recibe cada vez, como `&mut InputLink`:

```
fn on_message(_microseconds: u64, bytes: &[u8], link: &mut InputLink) {
    let Some(message) = MidiMessage::parse(bytes) else {
        return;
    };
    let message = link.snapped(message);
    let mut position = None;
    if let Some(live) = &link.live {
        let mut live = lock(live);
        live.send(message);
        position = live.position();
    }
    // `send` fails only if the UI is gone, and then nobody needs the
copy.
    let _ = link.to_ui.send(Played { message, position });
    // egui only redraws when something happens. This is something, and
    // `request_repaint` is one of the few egui calls meant for other
threads.
    link.ctx.request_repaint();
}
```

Se ejecuta en el hilo de midir, así que `connect` exige que ella y sus datos sean `Send + 'static`, como el callback de audio. El tiempo que pasa midir se ignora: el reloj que importa es la posición del hilo de audio.

Por qué el `LiveInput` está detrás de un `Mutex`

`InputLink` guarda el `LiveInput` como `Arc<Mutex<LiveInput>>`, lo que parece raro para un valor que solo usa un hilo a la vez. La razón son los errores de midir. Si `connect` falla, el error devuelve el `MidiInput`, pero *no* los datos: `link` se perdería, y con él el único productor de la cola en vivo, para siempre. Ya no se podría conectar ningún teclado.

Así que el struct `Midi` guarda el `LiveInput` él mismo, detrás de un `Arc<Mutex<>>`, y cada enlace guarda un clon del `Arc`. Una conexión fallida solo pierde un clon. Solo existe una conexión a la vez, así que nunca se espera por el cerrojo, y el hilo de midir no es el hilo de audio, así que puede bloquear. El mismo `Arc` permite a la aplicación poner el `LiveInput` de un motor nuevo después de abrir otra vez la tarjeta de sonido (`reconnect_audio`), sin volver a conectar el teclado.

Cerrojos envenenados

```
fn lock<T>(mutex: &Mutex<T>) -> MutexGuard<'_, T> {
    mutex.lock().unwrap_or_else(PoisonError::into_inner)
}
```

Si un hilo entra en pánico mientras tiene un cerrojo, Rust marca el `Mutex` como **envenenado**, y cada `lock()` posterior devuelve un `Err`: los datos de dentro pueden estar a medio cambiar, y Rust no los entrega sin preguntar.
`PoisonError::into_inner` es la forma de decir «dámelos de todos modos». Aquí es seguro, porque un extremo de cola, una conexión o una tecla nunca quedan a medio cambiar. La función es genérica sobre `T`, así que sirve para todos los mutex del archivo.

Una copia para la interfaz: `mpsc`

La copia de cada tecla para la interfaz va por un canal `std::sync::mpsc` («varios productores, un consumidor»), de la biblioteca estándar:

```
let (to_ui, from_keyboard) = mpsc::channel();
// ...
pub fn played(&self) -> Vec<Played> {
    self.from_keyboard.try_iter().collect()
}
```

A diferencia de rtrb, reserva memoria y puede bloquear, lo que aquí no importa: ni el hilo de midir ni el de la interfaz son de tiempo real. Y el `Sender` se puede clonar, así que cada conexión nueva recibe el suyo. `try_iter` da los mensajes que esperan y se detiene cuando no hay más, sin esperar: lo correcto para una interfaz que no debe dejar de dibujar.

egui solo dibuja un frame cuando pasa algo: se mueve el ratón, se escribe una tecla. Una tecla de un teclado MIDI no es algo que egui vea, así que `on_message` llama a `ctx.request_repaint()`. `egui::Context` es un handle que se puede clonar y enviar a otros hilos, y `request_repaint` es una de las pocas llamadas de egui pensadas para usarse desde ellos.

Ajustar las teclas a la tonalidad

Ajustar las notas que tocas, en el piano roll, lleva cada tecla que tocas a la nota más cercana de la tonalidad en que está la canción en ese punto, así que no puede salir una nota equivocada (Escalas, modos y tonalidades). Para un teclado MIDI eso tiene que ocurrir en el hilo de midir, antes de que la tecla vaya al hilo de audio. La interfaz comparte la tonalidad a la que ajustar mediante un `Arc<Mutex<Option<Scale>>>` más, que fija en cada frame, y el enlace ajusta cada tecla con `tools::nearest_in_key` (Herramientas de edición como funciones simples).

La tonalidad de la canción puede cambiar mientras se mantiene una tecla: la reproducción cruza una marca de tonalidad. Entonces la subida de la tecla, ajustada a la tonalidad nueva, podría nombrar otra nota, y la nota que empezó al bajar no terminaría nunca. Así que el enlace recuerda, para cada tecla pulsada, la nota que toca:

```
MidiMessage::NoteOff {
    channel,
    pitch: key,
} => {
    let at = self.sounding.iter().position(|&(k, _)| k == key);
    let pitch = at.map_or(key, |at| self.sounding.remove(at).1);
    MidiMessage::NoteOff { channel, pitch }
}
```

`map_or(key, ...)` se lee «la nota que empezó esta tecla, o la tecla misma si no empezó ninguna que conozcamos». El teclado del equipo guarda los mismos pares para sus teclas, en `typing_down`.

El teclado del equipo

¿No hay un teclado MIDI a mano? **MIDI > Teclado del equipo como piano** convierte las teclas de letras en uno, dispuesto como las teclas negras de un piano, entre las blancas y por encima de ellas:

```
  teclas negras:  S  D     G  H  J      2  3     5  6  7     9  0
  teclas blancas: Z  X  C  V  B  N  M   Q  W  E  R  T  Y  U  I  O  P
  notas:          Do4 Re4 Mi4 Fa4 Sol4 La4 Si4  Do5 Re5 Mi5 Fa5 Sol5 La5
Si5 Do6 Re6 Mi6
```

La disposición es una tabla, `TYPING_KEYS: &[(Key, u8)]`, y `typing_note` busca una tecla en ella con `find`. Las teclas se leen una vez por frame. egui guarda los eventos de teclado de este frame en una lista, y el código elige los que quiere:

```
egui::Event::Key {
    key,
    physical_key,
    pressed,
    repeat: false,
    modifiers,
} => {
    // The *physical* key, where it is on the keyboard,
    // so the piano keeps its shape on any layout:
    // on a French keyboard, W is where Z is in English.
    let key = physical_key.unwrap_or(*key);
    // Ctrl+Z is undo, not a note. Key releases always
    // count, or a note could stay on.
    let shortcut = modifiers.command || modifiers.alt;
    (!*pressed || !shortcut).then_some((key, *pressed))
}
```

Cuatro detalles hacen que se comporte como un piano:

- **`repeat: false`** en el patrón. Una tecla mantenida hace que el sistema la repita («aaaaa» en un campo de texto). Un campo escrito como un valor en un patrón tiene que ser igual a ese valor, así que las repeticiones no encajan, y una tecla mantenida toca una vez.
- **`physical_key`**. `key` es la letra que la distribución del teclado imprime en la tecla; `physical_key` es dónde *está* la tecla. Algunos sistemas no la informan, así

que `unwrap_or` vuelve a la letra.

- **Los atajos no son notas.** Una tecla que baja con Ctrl (Cmd en un Mac) o Alt pulsado se salta. Una tecla que *sube* siempre cuenta, se pulse lo que se pulse, o una nota empezada antes de pulsar Ctrl no terminaría nunca. Los atajos de una sola tecla de la aplicación (Espacio, L, Inicio, Fin, Supr…) no están en las teclas del piano, así que siguen funcionando mientras está activo.
- **El foco.** Si la ventana pierde el teclado (haces clic en otra ventana), o se lo queda un campo de texto, un menú o un diálogo, los eventos de «subida» de las teclas van a otra parte. Así que todas las teclas del equipo se sueltan a la vez cuando eso ocurre.

Los eventos se leen dentro de `ctx.input(|input| ...)`, que mantiene un cerrojo sobre la entrada de egui mientras se ejecuta el closure. El closure solo reúne un `Vec` de `(key, pressed)`, y el bucle que toca las notas se ejecuta después, con el cerrojo ya liberado.

Soltar todas las teclas usa `std::mem::take`:

```
fn release_typing_keys(&mut self) {
    // `take` leaves an empty list behind, so the loop can call `&mut
self` methods.
    for (_, pitch) in std::mem::take(&mut self.live.typing_down) {
        self.play_typed(MidiMessage::NoteOff { channel: 0, pitch });
    }
}
```

Recorrer `&self.live.typing_down` prestaría `self` durante todo el bucle, y `play_typed` necesita `&mut self`. Sacar la lista primero deja una vacía en su lugar, y el bucle es dueño de la antigua, así que nunca se solapan.

`play_typed` envía la tecla al hilo de audio como `Command::Live`, y luego se la pasa a `key_played` con la posición, exactamente como llega una tecla de midir. Las teclas de armonía tocan sus acordes del mismo modo. A partir de ahí, la aplicación no distingue los teclados.

El canal seleccionado

Las teclas necesitan un canal que tocar, y el piano roll ya tiene uno: el canal que muestra. Una vez por frame, `live_input` avisa al hilo de audio si cambió:

```
let channel = self
    .piano_roll
    .selected_channel()
    .filter(|&channel| channel < self.project.channels().len());
if self.live.channel_sent != Some(channel) {
    self.send(Command::SetLiveChannel(channel));
    self.live.channel_sent = Some(channel);
}
```

`channel_sent` es un `Option<Option<usize>>`, y las dos capas significan algo. El `None` exterior es «aún no se envió nada», así que el primer frame, y el primero después de un proyecto nuevo, siempre envían. `Some(None)` es «enviado: ningún canal», que el hilo de audio entiende como «suelta todas las teclas e ignora las nuevas». Juntar las dos cosas en un solo `Option` haría imposible decir una de ellas.

Elegir otro canal también termina la toma que se esté grabando, y suelta todas las teclas pulsadas, así que una tecla nunca termina en un canal en el que no empezó.

Las teclas pulsadas

Cada tecla pulsada, de cualquiera de los dos teclados, se guarda en una lista de `Held`, en el orden en que bajaron:

```
struct Held {
    pitch: Note,
    velocity: u8,
    /// The channel it plays, and will be recorded into.
    channel: usize,
    /// Where playback was when it went down, if it is being recorded.
    recorded_from: Option<f64>,
}
```

Esta lista es lo que el resto de la aplicación entiende por «las teclas que pulsas». La barra de estado nombra su acorde («🎹 Am: La menor, acorde vi de Do mayor»), el piano roll las ilumina en su teclado, las teclas de armonía y el buscador de tonalidad las escuchan (Oír la armonía), y **El acompañamiento sigue los acordes** las convierte en el acorde de la banda (Cómo toca la banda una hoja de acordes). Grabar es un uso más.

Grabar: de una tecla a una nota

El botón ● junto a Reproducir solo *arma* la grabación. Las teclas se graban mientras se cumplen cuatro cosas a la vez:

```
pub(super) fn is_recording(&self) -> bool {
    self.live.recording
        && self.playing
        && self.mode == PlayMode::Pattern
        && self.piano_roll.selected_channel().is_some()
}
```

Armada, reproduciendo, en modo patrón (una toma va al patrón que se repite) y con un canal en el que grabar. Armarla pasa al modo patrón por ti, así que basta ● y luego Espacio.

Recordar dónde bajó una tecla

Cuando una tecla baja, `key_played` añade un `Held`:

```
self.live.held.retain(|held| held.pitch != pitch);
self.live.follow.key_down();
let recorded_from = played.position.filter(|_| self.is_recording());
self.live.held.push(Held {
    pitch,
    velocity,
    channel,
    recorded_from,
});
```

`retain` quita primero la misma altura si ya estaba pulsada (en el otro teclado, por ejemplo), así que cada altura está una sola vez en la lista.

`Option::filter` conserva un `Some` solo si el closure dice que sí, y si no lo convierte en `None`. Así que `recorded_from` es la posición cuando la tecla se está grabando, y `None` cuando solo se toca. El closure ignora su argumento (`|_|`): la pregunta no es sobre la posición, sino sobre si se está grabando.

Cuando la tecla sube, su `Held` sale de la lista, y si tiene posición de inicio y hay posición final, se convierte en una nota:

```
let held = self.live.held.remove(index);
if let (Some(from), Some(until)) = (held.recorded_from, played.position) {
    self.record(held, from, until);
}
```

Hacer match con una tupla de dos `Option` comprueba los dos a la vez: el bloque solo se ejecuta si los dos son `Some`.

Cuantizar: `recorded_note`

Nadie toca exactamente en el paso. Una tecla que bajó en 3,9 y subió en 6,2 quería ser una nota del paso 4 al 6. Mover los extremos de una nota a la cuadrícula se llama **cuantizar**, y es una función normal en `lineup-core`, sin teclado ni interfaz a la vista:

```
pub fn recorded_note(pitch: Note, velocity: u8, from: f64, until: f64,
length: usize) -> NoteEvent {
    let length = length.max(1);
    let until = if until < from {
        until + length as f64
    } else {
        until
    };
    let start = from.round();
    let end = until.round().max(start + 1.0);
    // `as usize` turns a float into an integer, dropping the fraction
    // (there is none left after `round`). Negative numbers become 0.
    let start_step = start as usize % length;
    let steps = ((end - start) as usize).min(length - start_step);
    NoteEvent {
        velocity,
        ..NoteEvent::new(pitch, start_step, steps)
    }
}
```

Línea a línea:

1. `length.max(1)` protege el `%` de abajo: un resto entre cero hace panic. Un patrón nunca está vacío, pero una función no debería depender de quien la llama para eso.
2. El patrón se repite. Si la tecla subió *antes* de donde bajó (bajó en 14,5, subió en 1,5), el bucle volvió a empezar en medio, y el final es en realidad 17,5, una vuelta más allá.
3. `round()` lleva cada extremo al paso más cercano. `f64::round` redondea las mitades alejándose de cero, así que 2,5 pasa a 3.
4. `.max(start + 1.0)`: una pulsación rápida redondearía los dos extremos al mismo paso, una nota de duración 0. Una nota dura al menos un paso.

5. Una tecla pulsada justo antes de que el bucle vuelva a empezar (en 15,7 en un patrón de 16 pasos) redondea a 16, que es el paso 0 de la vuelta siguiente: `% length` lo pliega de nuevo a 0. Ahí es donde lo quería quien toca.
6. Una nota no puede pasar del final del patrón, así que se corta ahí con `.min(length - start_step)`.

Las últimas líneas usan la **sintaxis de actualización de structs**: `..NoteEvent::new(...)` significa «todos los campos no nombrados, como en este valor», así que la nota recibe su velocidad, y `nudge` y `trim` a 0 de `new`. Una nota grabada cae en la cuadrícula.

Los tests se leen como esas reglas:

```
#[test]
fn ends_go_to_the_nearest_step() {
    assert_eq!(recorded(2.0, 4.0), (2, 2));
    assert_eq!(recorded(1.8, 4.3), (2, 2), "a little early, a little
late");
    assert_eq!(recorded(2.4, 4.6), (2, 3));
}

#[test]
fn just_before_the_loop_is_step_zero() {
    assert_eq!(recorded(15.7, 0.4), (0, 1));
    assert_eq!(recorded(15.6, 1.9), (0, 2));
}

#[test]
fn a_note_across_the_loop_end_is_cut_there() {
    assert_eq!(recorded(14.0, 2.0), (14, 2));
}
```

`recorded(from, until)` es una función auxiliar del módulo de tests que graba un Do4 en un patrón de 16 pasos y devuelve `(start, length)`. Poder escribir `15.7` en lugar de tener que tocar una tecla justo en ese momento es la razón de que la función viva en `lineup-core` y reciba números simples.

`start as usize` convierte un `f64` en un `usize`. Para los floats, `as` descarta la parte fraccionaria y **satura**: un número negativo pasa a 0, uno demasiado grande a `usize::MAX`, y NaN a 0. Nunca hace panic ni da la vuelta. Es distinto de `as` entre enteros, que se queda solo con los bits bajos: `300_i32 as u8` es 44.

Las posiciones dicen *en qué punto del bucle* estaba una tecla, no *en qué vuelta*. Una tecla mantenida más de un bucle entero parece igual que una corta: bajar en 2 y subir en 3 una vuelta después se lee como un paso de duración. Y una nota que cruza el final del bucle se corta ahí, en lugar de seguir en el paso 0. Las dos cosas quedan como ejercicios.

Un paso de deshacer por toma

`record` añade la nota al patrón seleccionado mediante `edit_pattern`, como cualquier otro cambio, así que se puede deshacer y el hilo de audio se entera (Decirle al hilo de audio lo que ha cambiado):

```
fn record(&mut self, held: Held, from: f64, until: f64) {
    let Some(length) =
self.project.pattern(self.pattern).map(Pattern::length) else {
        return;
    };
    let note = recorded_note(held.pitch, held.velocity, from, until,
length);
    self.edit_pattern("record notes", |pattern| {
        pattern.add_note(held.channel, note);
    });
}
```

`.map(Pattern::length)` pasa un método como la función de `map`: `Pattern::length` es una función que recibe un `&Pattern`, justo lo que entrega `map`. `add_note` se salta una nota con la misma altura y el mismo inicio que otra que ya está, así que tocar lo mismo en la vuelta siguiente no apila notas unas encima de otras.

Una toma puede tener docenas de notas, y Ctrl+Z debería deshacerlas todas, no de una en una. El historial ya mantiene un paso abierto mientras el ratón está pulsado, así que un arrastre es un solo paso (Deshacer y rehacer). Grabar se suma a esa regla:

```
pub(super) fn end_frame_for_undo(&mut self, ctx: &egui::Context) {
    let holding = ctx.input(|input| input.pointer.any_down());
    if !holding && !ctx.text_edit_focused() && !self.is_recording() {
        self.close_step();
    } else if self.step_unsynced {
        // ...
    }
}
```

Mientras se graba, el paso sigue abierto, y cada nota se une a él. La toma termina, y el paso con ella, cuando se detiene la reproducción, se desarma la grabación, cambia el modo o se elige otro canal.

Terminar una toma: antes de Detener

Cuando se detiene la reproducción, puede haber teclas aún pulsadas. Se graban como si hubieran subido en ese momento, con `finish_recording`:

```
pub(super) fn finish_recording(&mut self) {
    let until = self.position();
    for index in 0..self.live.held.len() {
        let held = self.live.held[index];
        if let (Some(from), Some(until)) = (held.recorded_from, until) {
            self.record(held, from, until);
        }
        self.live.held[index].recorded_from = None;
    }
}
```

Las teclas siguen en la lista (siguen pulsadas, y siguen sonando), pero su `recorded_from` se borra, así que su subida más tarde no graba nada. El bucle va por índice, no `for held in &self.live.held`, porque `record` necesita `&mut self` mientras la lista está prestada. `Held` es `Copy`, así que `let held = self.live.held[index];` lo copia, y el préstamo termina en esa línea.

El orden en `halt`, por donde pasa cualquier forma de detener, importa:

```
fn halt(&mut self) {
    if self.playing {
        // Before the audio thread forgets where it was.
        self.finish_recording();
    }
    self.playing = false;
    self.send(Command::Stop);
}
```

`finish_recording` necesita la posición, y `position()` da `None` en cuanto `self.playing` es falso, o en cuanto el hilo de audio se ha detenido y ha guardado `NO_POSITION`. Así que la toma se termina primero, mientras los dos aún dicen «reproduciendo», y solo después se envía Stop.

Latencia: lo que oyes ya pasó

Cuando pulsas una tecla «en el tiempo», la pulsas en el tiempo que *oíste*. Pero el sonido tarda en llegarte. El hilo de audio prepara cada bloque por adelantado y se lo da a la tarjeta de sonido, que lo toca un poco más tarde; una tarjeta suele guardar uno o dos bloques, así que con bloques de 512 frames a 48 kHz un sonido sale de 11 a 21 ms después de que el hilo de audio lo hiciera. Cuando oyes empezar el paso 4, el hilo de audio ya va así de adentrado en él, y una tecla pulsada justo entonces se marca así de tarde.

A 120 BPM un paso dura 125 ms, así que 21 ms son un sexto de paso, y redondear al paso más cercano lo oculta, salvo que ya llegaras un poco tarde. Un teclado MIDI

apenas añade nada: sus teclas se marcan en el hilo de midir en cuanto llegan. El teclado del equipo añade más: egui lee el teclado una vez por frame, así que una tecla escrita puede esperar hasta un frame antes de marcarse, 17 ms a 60 frames por segundo, y mucho más en un equipo que dibuja sus frames despacio.

La solución es restar la latencia a la posición de cada tecla antes de redondear. cpal puede decir cuánta es: cada llamada del callback de audio recibe un `OutputCallbackInfo` con dos marcas de tiempo, cuándo se ejecutó el callback y cuándo sonará su sonido. Es uno de los ejercicios.

Salida MIDI: un instrumento con bandeja de salida

Un canal de salida MIDI no suena. Su instrumento convierte las notas en mensajes MIDI para otro sintetizador. Es una clase más de `Instrument`, junto al sampler, el Lineup Osc y el reproductor de SoundFont, así que todo lo demás (el secuenciador, las teclas en vivo, el volumen, silenciar) funciona con él sin cambios.

El hilo de audio no debe enviar los mensajes él mismo. Enviar pasa por la capa MIDI del sistema operativo, una llamada al sistema que puede tardar lo que sea, y la regla de oro prohíbe esperar. Así que `MidiOut` junta sus mensajes en una *bandeja de salida*, un `Vec` con espacio reservado al crearlo en el hilo de la interfaz:

```
fn post(&mut self, message: MidiMessage) {
    let outbox = &mut self.state.outbox;
    if outbox.len() < outbox.capacity() {
        outbox.push(message);
    }
}
```

Comprobar `len() < capacity()` antes de meter hace que el `Vec` nunca crezca, así que nunca reserva memoria. El espacio es `OUTBOX_SIZE`, 256 mensajes para un bloque de pocos milisegundos; incluso un acorde de dieciséis notas que empieza mientras terminan otras dieciséis son solo 32. Si alguna vez se llenara, se perdería un mensaje antes que romper la regla.

```
pub struct MidiOut {
    /// The MIDI channel, 0 to 15.
    channel: u8,
    /// The note every note goes out as, for a drum.
    pitch: Option<Note>,
    /// The General MIDI program the synth should play, if any.
    program: Option<u8>,
    /// Whether the program change still has to be sent: before the first
    /// note, and again after every stop, in case the synth's sound was
    /// changed on the synth itself in between.
    program_due: bool,
    /// Frames played so far: the clock that says when a note-off is due.
    now: u64,
    /// Boxed so an `Instrument` stays small, as the voices of the others
are.
    state: Box<MidiOutState>,
}
```

Un sintetizador solo termina una nota cuando se le dice, así que `MidiOut` tiene que enviar un note-off cuando se agota el gate de cada nota. Guarda, para cada una de las 128 notas, un enum que dice si la nota suena y hasta cuándo:

```
enum Sounding {
    Off,
    /// Until this frame, when its note-off is due.
    Until(u64),
    /// Until `note_off`.
    Held,
}
```

`next_sample` no suena, pero cuenta los frames en `now`, que es el reloj de `Until`. Empezar una nota envía primero un program change cuando toca uno, y luego el note-on:

```
let pitch = self.pitch.unwrap_or(pitch);
let slot = usize::from(pitch.midi());
// MIDI has one of each note per channel. If this one is already
// sounding, end it first, or the synth would hear "on, on, off"
// and the new note would stop at the old one's note-off.
if self.state.notes[slot] != Sounding::Off {
    self.post(self.note_off_message(pitch));
}
```

`self.pitch.unwrap_or(pitch)` se lee «la nota del tambor si la hay, y si no la nota misma», y escribir `let pitch = ...` otra vez *oculta* el parámetro: desde esa línea, `pitch` es el valor nuevo, y el antiguo no se puede usar por error. `note_off` solo termina notas `Held`: si el patrón toca un Do4 largo y tú pulsas Do4 en el teclado, soltar tu tecla no debe terminar la nota del patrón.

Una vez por bloque, el hilo de audio pide a cada instrumento que entregue sus mensajes:

```
pub fn flush(&mut self, mut send: impl FnMut(MidiMessage)) {
    for index in 0..128 {
        if let Sounding::Until(end) = self.state.notes[index]
            && end <= self.now
        {
            self.post(self.note_off_message(Note::new(index as u8)));
            self.state.notes[index] = Sounding::Off;
        }
    }
    // `drain(..)` empties the `Vec` but keeps its memory for next time.
    for message in self.state.outbox.drain(..) {
        send(message);
    }
}
```

Los note-offs se comprueban una vez por bloque, no en cada frame, así que pueden llegar unos milisegundos tarde, lo que nadie oye al final de una nota. `drain(..)` saca todos los elementos del `Vec` en orden, y el `Vec` conserva su memoria para el bloque siguiente. `send` es un `impl FnMut`: al instrumento no le importa adónde van los mensajes. El hilo de audio pasa un closure que los pone en la cola `MidiOutput` (o, cuando está llena, en una reserva preparada de antemano, porque un note-off perdido deja una nota sonando para siempre; ver Dos hilos). Los tests pasan uno que los mete en un `Vec` normal para mirarlos.

El estado vive en un `Box`. 128 `Sounding` y 128 `Option<Origin>` ocupan unos pocos kilobytes, y un enum es tan grande como su variante más grande: sin el box, cada `Instrument`, y cada `Command` que lleva uno por el ring buffer, sería así de grande. Un test del motor mantiene un comando en 80 bytes o menos, y fallaría sin el box.

Un instrumento de SoundFont también es un `MidiOut`, cuyos mensajes van a un sintetizador dentro de la aplicación en lugar de a un puerto (SoundFonts y General MIDI).

El hilo de envío

Los mensajes que los canales de salida MIDI dejan en la cola `MidiOutput` los envía un hilo propio, llamado `midi-out`:

```
let sender = move || {
    while !messages.is_abandoned() {
        thread::sleep(SEND_INTERVAL);
        let mut output = lock(&output);
        while let Some(message) = messages.pop() {
            // With no port connected, the messages are dropped. A send
            // that fails (a cable pulled out) loses only that message.
            if let Some(connection) = output.as_mut() {
                let _ = connection.send(&message.bytes());
            }
        }
    }
    // ...
};
// If the thread cannot start, MIDI Out stays silent; nothing else breaks.
thread::Builder::new()
    .name("midi-out".to_string())
    .spawn(sender)
    .ok()
```

Duerme un milisegundo (`SEND_INTERVAL`), envía lo que esté esperando y vuelve a empezar. Un hilo que espera trabajo normalmente dormiría hasta que lo despertaran, con un `Condvar` . Pero despertar un hilo es una llamada al sistema, que el hilo de audio no debe hacer. Mirar mil veces por segundo no cuesta casi nada, y añade como mucho un milisegundo.

El cerrojo se toma después de dormir, no antes, así que el hilo de la interfaz nunca espera por un hilo dormido. El guard `output` se suelta al final de cada vuelta del bucle, lo que libera el cerrojo.

`is_abandoned()` es verdadero en cuanto el otro extremo de la cola ha desaparecido: se soltó el motor de audio, como al cerrar la aplicación o al abrir otra vez la tarjeta de sonido. Entonces el bucle termina, el hilo envía All Notes Off por las notas que el motor antiguo aún debía, y acaba. Nadie tiene que decirle que pare.

`thread::Builder` es la forma larga de empezar un hilo. Puede darle un nombre, que aparece en los depuradores y en herramientas como `top -H` . Y su `spawn` devuelve un `Result` donde `thread::spawn` haría panic si el sistema se negara. `.ok()` lo convierte en un `Option<JoinHandle>` : si el hilo no puede empezar, la salida MIDI se queda callada y todo lo demás funciona, que es mejor que un fallo al arrancar. Sin audio no hay hilo de envío, ya que las notas se hacen en el hilo de audio, y el menú MIDI dice **No se envía nada mientras el audio no esté disponible**.

Cambiar de puerto sin notas colgadas

La conexión al puerto de salida está detrás de un `Mutex` porque la usan dos hilos: el hilo de envío envía, y el hilo de la interfaz la sustituye cuando eliges otro puerto:

```
// Swap the connections while holding the lock, so the sending
// thread never sends half to one and half to the other.
let old = std::mem::replace(&mut *lock(&self.output), new);
if let Some(mut old) = old {
    // Notes that were playing there would hang: tell them all to stop.
    for channel in 0..16 {
        let _ = old.send(&all_notes_off(channel));
    }
    // `close` gives back the `MidiOutput`, which is not needed.
    old.close();
}
```

`&mut *lock(&self.output)` bloquea el mutex, sigue el guard hasta el `Option` de dentro (`*`) y lo presta de forma mutable, todo en una expresión; el guard se suelta al final de la sentencia, así que el cerrojo se mantiene justo lo necesario para el intercambio. El hilo de envío envía entonces al puerto antiguo o al nuevo, nunca la mitad a cada uno.

La entrada tiene su propia versión de esto: cuando eliges otro teclado MIDI, se sueltan todas las teclas pulsadas en el antiguo (`release_all_keys`), porque sus subidas no llegarían nunca.

Enviar canales por MIDI

Un canal de salida MIDI, añadido con **Añadir canal > Salida MIDI (sintetizador externo)**, es la forma clásica: un canal propio que envía sus notas al sintetizador. También es una trampa. Conecta un sintetizador, elígelo en la **Salida** del menú **MIDI**, abre una demostración, pulsa Reproducir… y si no se hiciera nada al respecto, el sintetizador no diría nada, porque ninguna demostración tiene un canal de salida MIDI. Así que hay dos formas más: todos los canales a la vez, y un canal desde su menú de instrumento.

Todos los canales a la vez: rutas

MIDI > Enviar todos los canales está activado desde el principio. Mientras haya una salida conectada, cada canal suena en el sintetizador en lugar de en los altavoces, en los mismos canales MIDI que escribe **Archivo > Exportar canción como MIDI…**, así

que una canción suena igual tocada en vivo que desde su archivo MIDI. Una *ruta* es por dónde salen las notas de un canal:

```
pub type Route = (u8, Option<Note>);

pub fn routes(channels: &[Channel]) -> Vec<Option<Route>> {
    channels
        .iter()
        .zip(midi_channels(channels))
        .map(|(channel, midi_channel)| match &channel.kind {
            _ if channel.sent_over_midi().is_some() => None,
            ChannelKind::Sampler(SampleSource::Builtin(drum)) => {
                Some((midi_channel, Some(general_midi_drum(*drum))))
            }
            _ => Some((midi_channel, None)),
        })
        .collect()
}
```

Una ruta es un canal MIDI, y la nota del tambor si la hay. `type` solo da un nombre corto a un tipo largo; un `Route` sigue siendo una tupla normal. `zip` recorre dos listas a la par: cada canal con su canal MIDI, de `midi_channels` (más abajo). El primer brazo, `_ if ...`, es un comodín con guarda: encaja con cualquier clase, pero solo para un canal que ya se envía por MIDI, que envía sus notas él mismo y no necesita ruta.

Mientras se envían todos los canales, cada sampler y sintetizador se *sustituye* sin más por un `MidiOut`, y se repone cuando se deja de enviar:

```
pub fn to_midi(self, channel: u8, pitch: Option<Note>, program:
Option<u8>) -> Self {
    match self {
        Instrument::MidiOut(_) => self,
        _ => Instrument::MidiOut(
            MidiOut::new(channel)
                .playing_as(pitch)
                .with_program(program),
        ),
    }
}
```

`to_midi` toma `self` por valor: el instrumento antiguo se *mueve* a la función y se suelta allí, y quien llama recibe el nuevo. Una salida MIDI ya envía sus notas, así que vuelve sin cambios.

Las rutas dependen del menú (una salida elegida, la casilla marcada) y del rack: añadir un sintetizador encima de otro mueve el de abajo al canal MIDI siguiente. En lugar de

perseguir cada lugar que puede cambiarlas, la aplicación las vuelve a calcular en cada frame y compara:

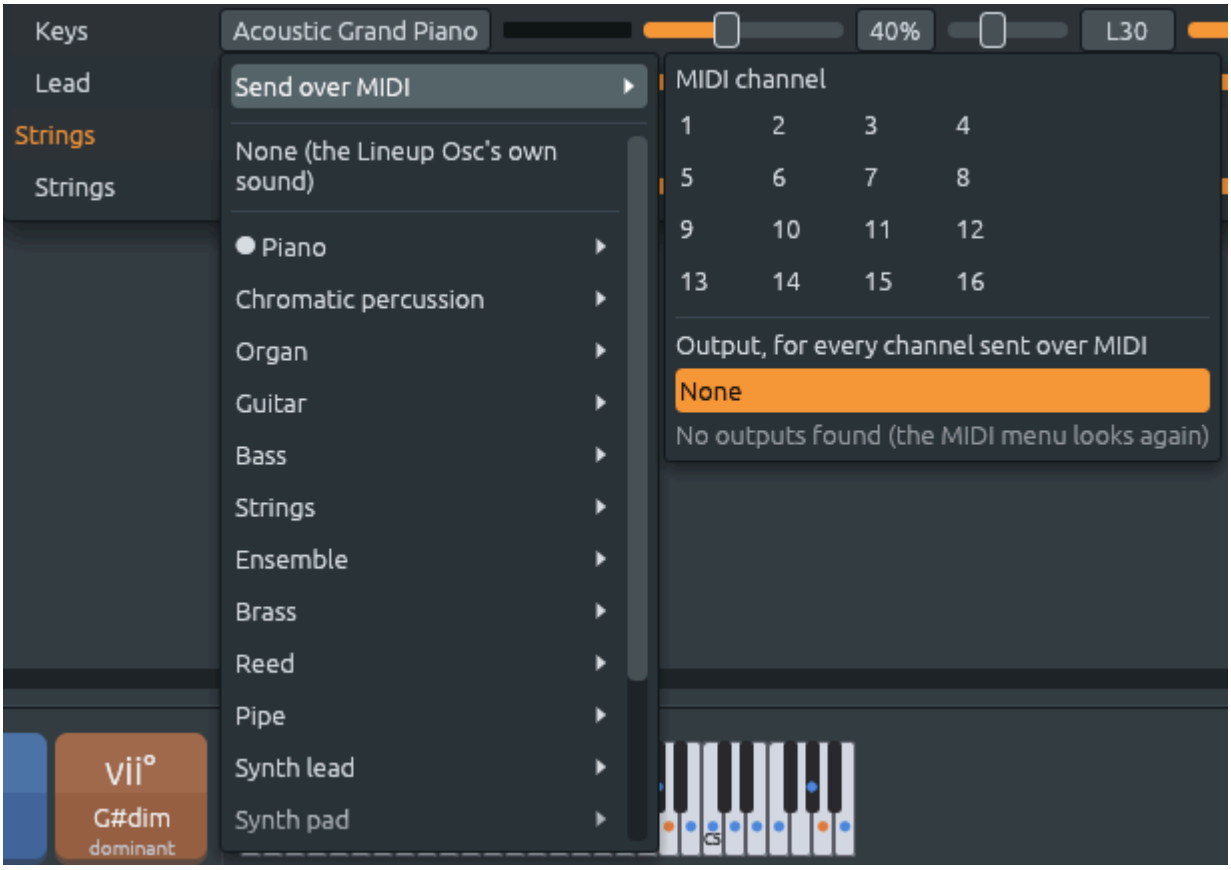
```
let wanted = if self.live.send_all && self.live.midi.output().is_some() {
    midi::routes(channels)
} else {
    vec![None; channels.len()]
};
if self.live.routes_sent.as_ref() == Some(&wanted) {
    return;
}
```

Calcular 32 rutas como mucho cuesta mucho menos que dibujar la ventana, y comparar dos `Vec` con `==` los compara elemento a elemento, porque las tuplas, `Option` y `Note` implementan `PartialEq`. Solo cuando difieren recibe un canal un instrumento nuevo, una salida MIDI o el suyo, mediante `Command::SetInstrument`, y solo los canales cuya ruta cambió.

`routes_sent` es un `Option<Vec<...>>`: `None` significa «no se sabe». Cuando se añaden, quitan o recargan canales, el hilo de audio recibe instrumentos normales para ellos, y los que se movieron conservan lo que tuvieran. Si alguno era una salida MIDI en lugar de su propio instrumento, la aplicación olvida lo que envió, y el frame siguiente vuelve a enviar todos los canales. Eso es raro (hace falta un cambio en el rack), y no puede salir mal.

Un canal desde su menú de instrumento

A menudo se quiere algo intermedio: el bajo en un sintetizador de hardware y el resto de la banda en la aplicación, o la batería en una caja de ritmos. Así que cada menú de instrumento, en el mezclador y en el rack de canales, empieza con **Enviar por MIDI**: un canal MIDI del 1 al 16, y la salida.



Un sintetizador enviado por MIDI **se convierte** en un canal de salida MIDI, y conserva su instrumento:

```
pub fn send_over_midi(&mut self, midi_channel: u8) {
    let midi_channel = midi_channel.min(15);
    match self.kind {
        ChannelKind::Sampler(SampleSource::Builtin(_)) => {
            self.sent_on = Some(midi_channel);
            return;
        }
        ChannelKind::Sampler(SampleSource::File(_)) => return,
        _ => {}
    }
    let plain = match self.kind {
        ChannelKind::Synth => self.name == Channel::synth().name,
        ChannelKind::MidiOut { channel } => self.name ==
Channel::midi_out(channel).name,
        _ => false,
    };
    if plain {
        self.name = Channel::midi_out(midi_channel).name;
    }
    self.kind = ChannelKind::MidiOut {
        channel: midi_channel,
    };
}
```

Un nombre que solo decía qué lo tocaba, «Lineup Osc» o «Salida MIDI 2», sigue el cambio y pasa a «Salida MIDI 3». Un nombre que le diste tú, como «Bajo», se queda. `play_here` hace lo contrario. Una muestra de un archivo no tiene ninguna nota que un sintetizador pueda tocar en su lugar, así que no se puede enviar.

Un **tambor** no cambia de clase. Recibe un campo, `sent_on`, con el canal MIDI, y sigue siendo un `Sampler` que toca su tambor incorporado. Eso le importa a la banda: los estilos encuentran el bombo, la caja y los charles por lo que son (`Sampler(Builtin(Drum::Kick))`), para escribir sus ritmos en ellos. Un bombo convertido en un canal de salida MIDI ya no se encontraría, y el estilo siguiente añadiría un segundo bombo. Por la misma razón, los estilos encuentran un «Bajo» o unos «Teclados» enviados por MIDI por su nombre, como el bajo y los teclados (La biblioteca). En el hilo de audio, `Instrument::for_channel` construye un `MidiOut` para un tambor con `sent_on`, que toca su propia nota de una batería General MIDI (Instrumentos), así que la parte de batería, la banda y los estilos quedan exactamente igual; solo el sonido se va a otra parte.

`Channel::sent_over_midi` responde «¿se envía, y en qué canal?» para las dos clases a la vez, así que el resto del código no tiene que conocer la diferencia.

El mezclador nombra el tambor que envía un canal, con la tabla de tambores General MIDI de `general_midi.rs`:

```
/// The drum General MIDI plays for `note` on channel 10, such as "Bass
/// Drum 1" for 36, if the drum set has one there.
pub fn drum_name(note: u8) -> Option<&'static str> {
    DRUMS
        .get(usize::from(note.checked_sub(FIRST_DRUM)?))
        .copied()
}
```

Aquí se juntan dos clases de «quizá». `checked_sub` devuelve `None` en lugar de dar la vuelta cuando la nota está por debajo de 35 (un `u8` no puede bajar de 0), y el `?` devuelve ese `None` de la función en el acto. `get` devuelve `None` más allá del final de la tabla, para las notas por encima de 81. `copied` convierte el `Option<&&str>` que da `get` en un `Option<&str>`.

El menú necesita la lista de salidas

Los menús de instrumento los dibujan funciones de `channel_rack.rs`, que conocen un canal y nada de las conexiones MIDI de la aplicación. Pasárselas a través del rack, el mezclador y el editor de estilos añadiría un parámetro a muchas funciones. En su

lugar, la aplicación pone la lista de salidas en la memoria de egui una vez por frame, y el menú la lee de ahí:

```
pub fn set_midi_outputs(ctx: &egui::Context, outputs: MidiOutputs) {
    ctx.data_mut(|data| data.insert_temp(midi_outputs_id(), outputs));
}
```

```
let outputs: MidiOutputs = ui
    .ctx()
    .data(|data| data.get_temp(midi_outputs_id()))
    .unwrap_or_default();
```

La memoria de egui puede guardar un valor de cualquier tipo que sea `Clone`, `Send`, `Sync` y `'static`, uno por `Id` y tipo. Los valores guardados con `insert_temp` no se guardan al cerrar la aplicación, lo que está bien para una lista de puertos que mañana puede ser otra. El menú no cambia la conexión él mismo: añade un `RackAction::ConnectMidiOutput`, que lleva a cabo la aplicación, como cualquier otro clic en el rack.

Un test de este menú, `the_mixer_sends_a_channel_over_midi_and_back`, muestra por qué un test tiene que preguntarle a la máquina qué tiene. Abre **Enviar por MIDI** y comprueba las salidas de la lista. Esperar **No se encontraron salidas** pasa en las máquinas de test de Linux y macOS, que no tienen salida MIDI, y falla en Windows, que siempre tiene una: su «Microsoft GS Wavetable Synth» incorporado. Así que el test lee las salidas que encontró la aplicación, y espera ese texto solo cuando no hay ninguna. Es la clase de diferencia que hay que pillar ejecutando los tests en tres sistemas (Tres sistemas e integración continua).

Qué canal MIDI usa cada canal

Un programa que lee un archivo MIDI, o un sintetizador que oye la aplicación, usa el canal MIDI para elegir el instrumento. General MIDI, el estándar que siguen la mayoría de los sintetizadores, reserva el **canal 10** (9, contando desde 0) para la batería, con una nota fija para cada tambor: 36 es un bombo, 38 una caja, 39 unas palmas, 42 un charles cerrado. Una sola función decide el canal MIDI de cada canal, para el archivo MIDI y para las rutas por igual:

```
pub fn midi_channels(channels: &[Channel]) -> Vec<u8> {
    let taken = |number: u8| {
        channels
            .iter()
            .any(|channel| channel.sent_over_midi() == Some(number))
    };
    let free: Vec<u8> = (0..16u8)
        .filter(|&number| number != DRUM_CHANNEL && !taken(number))
        .collect();
    let mut melodic = if free.is_empty() {
        (0..16u8).filter(|&number| number != DRUM_CHANNEL).collect()
    } else {
        free
    }
    .into_iter()
    .cycle();
    channels
        .iter()
        .map(|channel| match channel.kind {
            ChannelKind::Sampler(SampleSource::Builtin(_)) => {
                channel.sent_on.unwrap_or(DRUM_CHANNEL)
            }
            ChannelKind::MidiOut { channel } => channel,
            _ => melodic.next().unwrap_or(0),
        })
        .collect()
}
```

Los tambores incorporados comparten el canal 10, salvo que alguno se haya enviado a otro. Los canales enviados por MIDI conservan el canal que se les dio. Cada uno de los demás toma el siguiente canal *libre*: no el 10, ni uno en el que ya toque un canal enviado por MIDI, o los dos compartirían un sonido en el sintetizador y se cortarían las notas. `cycle` repite los canales libres para siempre, y el iterador se crea una vez, fuera del closure, que lo toma prestado de forma mutable: cada `next()` lo hace avanzar, así que los canales reciben 1, 2, 3… por turno, y si hay más canales que canales libres, empiezan a compartir.

Escribir un archivo MIDI

Un Standard MIDI File guarda los mismos mensajes, con sus tiempos. Guarda notas, no sonido: «Do4 empieza aquí, termina allí, en el canal 1». El programa que lo abre elige los instrumentos. Eso lo convierte en la lengua común de los programas de música, y en la forma de llevar una canción de Ferrous Lineup a otro estudio. `song_to_midi` escribe la canción tal como la toca la lista de reproducción, y `pattern_to_midi` un patrón, una vez.

El archivo es una lista de *chunks*. Cada uno empieza con cuatro letras y su longitud:

```
  MThd  longitud 6  formato 1  3 pistas  96 ticks por tiempo     la
cabecera
  MTrk  longitud …  nombre "Ferrous Lineup", tempo, compás, tonalidad, fin
  MTrk  longitud …  nombre "Lineup Osc", un program change, las notas, fin
  MTrk  longitud …  nombre "Kick", las notas del bombo, fin
```

Este es el **formato 1**: varias pistas tocadas a la vez. La primera pista solo guarda el tempo, el compás y la tonalidad, y cada canal del proyecto recibe una pista propia, con su nombre, que empieza con un program change cuando el canal tiene un instrumento General MIDI. Un canal sin notas, o uno que no se oye (silenciado, o callado por el solo de otro), no recibe pista.

Los números de un archivo MIDI son big-endian: el byte más significativo primero. Los enteros de Rust tienen un método para eso, que da un array de bytes:

```
let mut file = Vec::new();
file.extend_from_slice(b"MThd");
file.extend_from_slice(&6u32.to_be_bytes()); // the header's length
file.extend_from_slice(&1u16.to_be_bytes()); // format 1
file.extend_from_slice(&(tracks.len() as u16).to_be_bytes());
file.extend_from_slice(&TICKS_PER_BEAT.to_be_bytes());
```

`b"MThd"` es una **cadena de bytes**: un `&[u8; 4]` con los códigos ASCII de las letras, en lugar de un `&str`. `6u32` es el número 6 como `u32` (el tipo escrito tras el número), así que `to_be_bytes` da cuatro bytes: `[0, 0, 0, 6]`. Un `u16` da dos.

Ticks y tiempos delta

El tiempo en un archivo MIDI se cuenta en *ticks*, y la cabecera dice cuántos ticks hacen un tiempo. Ferrous Lineup escribe 96, los mismos ticks en los que coloca sus notas: un paso, una semicorchea, son 24 ticks, y el `nudge` y el `trim` de una nota cuentan los ticks que empieza después o termina antes de sus pasos (El proyecto). Así que una nota entre dos pasos se escribe exactamente donde suena.

```
pub const TICKS_PER_BEAT: u16 = 96;
pub const TICKS_PER_STEP: u32 = crate::pattern::TICKS_PER_STEP as u32;

// A check made while compiling: if someone changed one of the two
// numbers and not the other, the build would stop here.
const _: () = assert!(TICKS_PER_BEAT as u32 == TICKS_PER_STEP *
STEPS_PER_BEAT);
```

`const _: () = assert!(...)` es una constante sin nombre (`_`) cuyo valor se calcula al compilar. `assert!` funciona en un `const` desde Rust 1.57, y una aserción fallida ahí es un error de compilación, no un panic al ejecutar. No cuesta nada en el programa, y convierte «acuérdate de cambiar los dos» en algo que comprueba el compilador.

Dentro de una pista, cada evento se escribe tras un *tiempo delta*: el número de ticks desde el evento anterior, no desde el principio. La mayoría de los deltas son pequeños (0 para las notas de un acorde, 24 para el paso siguiente), así que se escriben en un formato donde los números pequeños ocupan un byte, la **cantidad de longitud variable**: siete bits del número por byte, el más significativo primero, con el bit superior a 1 en todos los bytes menos el último.

```
  200  = 1 1001000            ->  1000 0001  0100 1000   (0x81 0x48)
         │ └ últimos 7 bits             │         └ bit superior a 0: el
último byte
         └ el resto: 1                  └ bit superior a 1: viene más
  384  = 11 0000000           ->  0x83 0x00
```

```
fn write_variable_length(out: &mut Vec<u8>, value: u32) {
    let mut groups = [0u8; 5];
    let mut count = 0;
    let mut rest = value;
    loop {
        groups[count] = (rest & 0x7F) as u8;
        count += 1;
        rest >>= 7;
        if rest == 0 {
            break;
        }
    }
    for index in (0..count).rev() {
        let more = if index > 0 { 0x80 } else { 0 };
        out.push(groups[index] | more);
    }
}
```

El bucle corta el número en grupos de siete bits desde el extremo bajo, porque ese es el extremo al que llegan `& 0x7F` y `>>= 7`. El archivo los quiere desde el extremo alto, así que el segundo bucle recorre los grupos hacia atrás con `(0..count).rev()`. Cinco grupos bastan para cualquier `u32`, así que caben en un array, y no se reserva memoria. El test `variable_length_numbers` le da los ejemplos impresos en la especificación de los archivos MIDI.

Un `TrackWriter` guarda el tick del último evento, para que el resto del código pueda pensar en ticks absolutos:

```rust
#[derive(Default)]
struct TrackWriter {
    data: Vec<u8>,
    last_tick: u32,
}

impl TrackWriter {
    /// Writes the time since the last event. Events must come in order.
    fn delta(&mut self, tick: u32) {
        let delta = tick.saturating_sub(self.last_tick);
        write_variable_length(&mut self.data, delta);
        self.last_tick = self.last_tick.max(tick);
    }

    // ...

    /// Ends the track at `end` (or at its last event, if that is later)
    /// and wraps it in its chunk header.
    fn finish(mut self, end: u32) -> Vec<u8> {
        self.meta(end.max(self.last_tick), END_OF_TRACK, &[]);
        let mut chunk = Vec::with_capacity(8 + self.data.len());
        chunk.extend_from_slice(b"MTrk");
        chunk.extend_from_slice(&(self.data.len() as u32).to_be_bytes());
        chunk.extend(self.data);
        chunk
    }
}
```

`#[derive(Default)]` da `TrackWriter::default()`, un `Vec` vacío y el tick 0. `finish` toma `mut self`, no `&mut self`: consume el escritor, y sus datos se mueven al chunk sin copia. Un escritor terminado no se puede seguir escribiendo por error, porque ya no existe.

Metaeventos: nombres, tempo, compás y tonalidad

Los eventos que no son notas empiezan con `0xFF`, una clase y una longitud:

Bytes	Significa
`FF 03 len texto`	el nombre de la pista
`FF 51 03 tt tt tt`	el tempo, como microsegundos por tiempo
`FF 58 04 nn dd cc 08`	el compás: 4/4 es `04 02 18 08`
`FF 59 02 sf mi`	la armadura: sostenidos (negativos para bemoles), mayor o menor

Bytes	Significa
`FF 7F len datos`	datos para un solo programa (sequencer-specific)
`FF 2F 00`	el final de la pista

El tempo es el raro: no tiempos por minuto, sino cuánto dura un tiempo, en microsegundos. Una canción cuyo tempo cambia recibe un evento de tempo en cada paso donde cambia el tempo tocado, y los compases de la canción entran en cada cambio; las dos cosas están en Tempo, compás, swing y sensación.

La armadura dice cuántos sostenidos o bemoles, y si la tonalidad es menor. `Scale::key_signature` la calcula dando la vuelta al círculo de quintas (Escalas, modos y tonalidades). Una armadura solo conoce mayor y menor, así que un modo toma la armadura de la tonalidad mayor de la que viene: Re dórico no tiene ninguna, como Do mayor, y Mi dórico los dos sostenidos de Re mayor. Eso pierde algo: un archivo que dice «dos sostenidos, mayor» no puede decir si quiere decir Re mayor o Mi dórico. Así que junto a cada armadura el archivo recibe la tonalidad exacta, en un evento **sequencer-specific**, el único lugar que el estándar deja para los datos propios de un programa:

```
/// The start of our own event: 0x7D is the maker's number set aside for
/// programs that are not sold, then our name, so another program's
/// 0x7D event is never mistaken for ours.
const SCALE_EVENT_START: &[u8] = b"\x7DFerrous Lineup key ";

pub fn scale_event(scale: Scale) -> Vec<u8> {
    let mut data = SCALE_EVENT_START.to_vec();

data.extend_from_slice(ron::to_string(&scale).unwrap_or_default().as_bytes
());
    data
}

/// The key a sequencer-specific event holds, if [`scale_event`] wrote it.
pub fn scale_from_event(data: &[u8]) -> Option<Scale> {
    let key = data.strip_prefix(SCALE_EVENT_START)?;
    let text = std::str::from_utf8(key).ok()?;
    ron::from_str(text).ok()
}
```

La tonalidad se escribe como la escribe un archivo de proyecto, en RON, así que el evento dice `7D "Ferrous Lineup key (root:E,kind:Dorian)"`, y el código que guarda los proyectos la vuelve a leer (Guardar). `\x7D` en una cadena de bytes es el byte 0x7D. `strip_prefix` da el resto del slice después del comienzo, o `None` cuando el slice no empieza así; los `?` hacen que cada paso que puede fallar termine la función con `None`. Los demás programas se saltan el evento, y Ferrous Lineup recupera exactamente Mi dórico, Do♯ menor armónica o una serie dodecafónica. Una

serie o un conjunto no tienen armadura, así que para ellos este evento es todo lo que hay.

Notas, batería y orden

Cada nota se convierte en dos eventos, su note-on y su note-off. Las notas son las que toca la reproducción: `played_notes` las da con el swing de la canción, `song_steps_of` cada paso de la canción donde un trozo de la canción toca ese paso (una vez por repetición), y una nota que pasa del final de su patrón, o del trozo que se toca, se corta ahí, exactamente como la toca el hilo de audio:

```
let room = (length - note.start).min(play.end() - song_step);
let step = song_step as u32 * TICKS_PER_STEP;
let on = step + u32::from(note.nudge);
let off = if note.length <= room {
    on + note.ticks() as u32
} else {
    step + room as u32 * TICKS_PER_STEP
};
let pitch = drum.map_or(note.pitch, general_midi_drum);
```

`map_or(note.pitch, general_midi_drum)` da la altura propia de la nota para un canal melódico, y para un tambor incorporado, su nota General MIDI: los pasos de un tambor se escriben todos en Do4, que una batería General MIDI toca como un bongó. Pasar la función `general_midi_drum` misma, por su nombre, funciona porque su tipo, `fn(Drum) -> Note`, es lo que pide `map_or`.

Después los eventos se ordenan por tiempo. En el mismo tick, los note-offs tienen que ir antes que los note-ons: si una nota termina justo donde la misma nota vuelve a empezar, «on, off» terminaría la nota nueva enseguida, y solo «off, on» se oye dos veces.

```
events.sort_by_key(|&(tick, message)| (tick, matches!(message,
MidiMessage::NoteOn { .. })));
```

La clave es una tupla, `(tick, is_note_on)`, y las tuplas se comparan elemento a elemento. Para el mismo tick, `false < true`, así que ganan los note-offs. `sort_by_key` es un orden *estable*: los eventos con claves iguales conservan su orden, así que las notas de un acorde se quedan en el orden en que se escribieron. El test `a_repeated_note_ends_before_it_starts_again` comprueba el orden en el tick donde un Do4 termina y vuelve a empezar.

En la aplicación, **Archivo > Exportar canción como MIDI…** y **Exportar patrón como MIDI…** solo preguntan dónde guardar, y escriben los bytes con `std::fs::write`. El nombre de un patrón podría llevar una `/`, que un nombre de archivo no puede, así que el nombre que se ofrece para un patrón la sustituye.

Leer un archivo MIDI

Archivo > Importar archivo MIDI…, abrir un archivo `.mid` desde la línea de órdenes, o soltarlo en la ventana, convierte un archivo MIDI en un proyecto nuevo, listo para tocar. Los archivos MIDI son la forma más fácil de meter música real en la aplicación: casi cualquier pieza escrita se encuentra como uno, a menudo copiado de la partitura, a veces tocado en un teclado.

Leer es escribir recorrido al revés, con una diferencia que cambia cómo se escribe el código: **un archivo que leemos puede estar mal.** Puede estar cortado por una descarga rota, puede decir que un chunk es más largo de lo que es, puede no ser un archivo MIDI en absoluto. Sean cuales sean los bytes, el programa no debe fallar.

Un cursor que no puede caerse del final

El lector recorre los bytes con un pequeño *cursor*: un slice y una posición en él.

```
struct Bytes<'a> {
    data: &'a [u8],
    at: usize,
}

impl<'a> Bytes<'a> {
    // ...

    fn byte(&mut self) -> Option<u8> {
        let byte = *self.data.get(self.at)?;
        self.at += 1;
        Some(byte)
    }

    /// The next `count` bytes, or `None` if there are fewer left.
    fn take(&mut self, count: usize) -> Option<&'a [u8]> {
        let end = self.at.checked_add(count)?;
        let slice = self.data.get(self.at..end)?;
        self.at = end;
        Some(slice)
    }
}
```

Dos ideas de Rust hacen esto seguro:

- **`'a` es un lifetime.** `Bytes<'a>` toma prestados los bytes del archivo; no los copia. El lifetime dice «un `Bytes` no puede vivir más que los bytes a los que apunta», y el compilador lo comprueba. `take` devuelve `&'a [u8]`: un slice de los bytes *originales*, que sigue siendo válido aunque el cursor avance.
- **`get` en lugar de indexar.** `data[at]` entra en pánico cuando `at` pasa del final; `data.get(at)` devuelve `None`. El `?` que le sigue devuelve ese `None` de `byte` en el acto. `checked_add` hace lo mismo con una longitud tan grande que sumarla desbordaría. Así que cada lectura da un valor o `None`, y quien llama decide qué significa un valor que falta.

Un tiempo delta se lee de vuelta pegando los grupos de siete bits:

```
fn variable_length(&mut self) -> Option<u32> {
    let mut value = 0u32;
    for _ in 0..4 {
        let byte = self.byte()?;
        value = (value << 7) | u32::from(byte & 0x7F);
        if byte & 0x80 == 0 {
            return Some(value);
        }
    }
    None
}
```

`81 00` es `1 << 7 | 0` = 128. El estándar permite cuatro bytes como mucho; un quinto significaría que el archivo está dañado, así que tras cuatro el bucle se rinde.

La cabecera y los chunks

`read` comprueba `MThd`, lee el formato y la *división*, y luego lee cada chunk hasta el final del archivo:

```
while let Some(id) = file.take(4) {
    let Some(length) = file.u32() else {
        damaged = true;
        break;
    };
    let length = length as usize;
    // A chunk that claims to be longer than what is left of the file
    // was cut short: read what there is.
    let data = file.take(length.min(file.remaining()));
    let data = data.unwrap_or_default();
    if data.len() < length {
        damaged = true;
    }
    // Chunks of other kinds may be added by future versions of the
    // standard, or by one program for itself: they are skipped.
    if id == b"MTrk" {
        let (track, complete) = read_track(data);
        damaged |= !complete;
        tracks.push(track);
    }
}
```

La cabecera también dice cuántas pistas hay, pero algunos programas escriben mal ese número, así que se confía en los chunks mismos. Los archivos reales a menudo están un poco rotos, y todos los programas que los abren son indulgentes con ello. Este también: una pista cortada conserva los eventos leídos hasta ahí, un chunk más largo que el archivo se lee hasta donde llega, los chunks desconocidos se saltan. Pero el resultado lo dice: `MidiFile::damaged` se activa, y el aviso tras la importación termina con «El archivo está dañado: se leyó todo lo anterior al daño». `damaged |= !complete` es el «o» booleano escrito como asignación, como `+=`: una vez dañado, siempre dañado.

Dos cosas más que un lector encuentra por ahí:

- **Archivos RMID** (`.rmi`). Windows envolvía los archivos MIDI en RIFF, el contenedor de los archivos WAV: `RIFF`, un tamaño, `RMID`, y luego chunks. El archivo MIDI es el chunk llamado `data`. A diferencia de MIDI, RIFF escribe sus números con el byte menos significativo primero, así que `unwrap_riff` usa `u32::from_le_bytes` donde MIDI usa `u32::from_be_bytes`, y un chunk de longitud impar va seguido de un byte de relleno.
- **Ticks por segundo.** La división normalmente cuenta ticks por tiempo (480 y 960 son habituales). Con el bit superior a 1 cuenta en cambio ticks por fotograma de cine o video (24, 25, 29,97 o 30 fotogramas por segundo), como `Timing::PerSecond`. Un archivo así no tiene tiempos en absoluto; la importación cuenta su tiempo en tiempos de 120 BPM, para que suene a la velocidad correcta.

Running status

Una pista de una pieza de piano es casi todo note on, note off, note on... Para ahorrar espacio, un archivo puede omitir el byte de estado cuando es el mismo que el anterior. El lector siempre puede saberlo, porque un byte de estado tiene el bit superior a 1 y un byte de datos no:

```
let (status, first_data) = if first & 0x80 != 0 {
    (first, bytes.byte())
} else {
    (running.unwrap_or(0), Some(first))
};
```

Así que `90 3C 64 00 40 64` son dos note-ons: Do4 y Mi4, los dos con velocidad 100. Los program changes y el channel pressure tienen un byte de datos, el resto de los mensajes de canal dos. Un metaevento (`FF`) y un mensaje system exclusive (`F0` o `F7` , ajustes para los sintetizadores de un fabricante, que se saltan) llevan su propia longitud. Cualquier otro mensaje de sistema solo viaja por cables, nunca en archivos, así que encontrar uno significa que la pista está dañada.

Los eventos como enum

Cada evento que necesita la importación se convierte en una variante de un enum, con sus datos dentro:

```
pub enum EventKind {
    NoteOn { channel: u8, key: u8, velocity: u8 },
    NoteOff { channel: u8, key: u8 },
    Controller { channel: u8, number: u8, value: u8 },
    Program { channel: u8, program: u8 },
    Tempo(u32),
    TimeSignature { beats: u8, unit: u8 },
    KeySignature { sharps: i8, minor: bool },
    ExactKey(Scale),
    TrackName(String),
    InstrumentName(String),
}
```

Todo lo demás (pitch bend, aftertouch, letras) se lee y se descarta. Un note-on con velocidad 0 se convierte en un `NoteOff` , como en un cable. `meta_event` usa patrones de slice para comprobar que los datos de un metaevento tienen la forma correcta antes de fiarse de ellos:

```
// Sharps (negative for flats), then 0 for major or 1 for minor.
(KEY_SIGNATURE, &[sharps, mode]) => {
    let sharps = sharps as i8;
    ((-7..=7).contains(&sharps) && mode <=
1).then_some(EventKind::KeySignature {
        sharps,
        minor: mode == 1,
    })
}
```

Una armadura son exactamente dos bytes. `&[sharps, mode]` solo encaja con un slice de longitud dos y nombra sus bytes; una armadura de tres bytes, o una que dice tener 9 sostenidos, no es una armadura y se salta. `sharps as i8` lee los mismos ocho bits como un número con signo, así que el byte `0xFE` es -2: dos bemoles. `bool::then_some` da `Some(valor)` cuando el `bool` es verdadero y `None` si no.

Los nombres de pistas e instrumentos son texto, y el estándar no dice nada de cómo se escriben las letras que no son del inglés básico: los programas nuevos usan UTF-8, los antiguos Latin-1, un byte por letra. `text` prueba primero UTF-8, ya que un texto en Latin-1 casi nunca es UTF-8 válido, y si no, convierte cada byte en el `char` con ese número, que es exactamente Latin-1.

Importar: de los eventos a un proyecto

`import(&file, title)` convierte un `MidiFile` en un `Project` y un `Report` de lo que hizo. El informe se convierte en el aviso de la barra de estado, como *«Clementi: 2 canales, 812 notas, 38 compases, 4/4, 120 BPM, Do mayor»*.

En la aplicación, leer el archivo, leer el MIDI e importar son tres pasos que pueden fallar cada uno con un tipo de error distinto. Se encadenan en un solo `Result`:

```
let imported = std::fs::read(path)
    .map_err(|err| err.to_string())
    .and_then(|bytes| midi_file::read::read(&bytes).map_err(|err|
err.to_string()))
    .and_then(|file| {
        midi_file::import::import(&file, &title).map_err(|err|
err.to_string())
    });
```

`and_then` ejecuta el paso siguiente solo si el anterior tuvo éxito, y le pasa su valor. `and_then` conserva el tipo de error, así que cada paso convierte primero su propio error en un `String` con `map_err`. Un solo `match` al final muestra el aviso o el error.

Leer tarda unas milésimas de segundo, así que, a diferencia de importar una grabación, se hace en el acto en el hilo de la interfaz.

Los ticks del archivo en los nuestros

Un archivo cuenta en sus propios ticks, a menudo 480 o 960 por tiempo; un proyecto cuenta 96. `Clock` convierte unos en otros, al tick más cercano:

```
Timing::PerBeat(per_beat) => {
    let per_beat = u128::from(per_beat);
    (u128::from(tick) * u128::from(TICKS_PER_BEAT) + per_beat / 2) /
per_beat
}
```

Sumar la mitad del divisor antes de dividir redondea al entero más cercano en lugar de hacia abajo. La cuenta se hace en `u128` porque un archivo dañado puede decir que un tick está cerca de `u64::MAX`, y multiplicarlo por 96 desbordaría un `u64`; luego el resultado se limita, mucho más allá del final de cualquier canción.

Notas entre pasos

Un patrón cuenta el tiempo en pasos. Redondear cada nota al paso más cercano estropearía mucha música:

- **Los tresillos** dividen un tiempo en tres. Los tresillos de corcheas empiezan en 0, ⅓ y ⅔ de tiempo: ninguno es una semicorchea.
- **El swing** toca tarde la segunda corchea de cada tiempo.
- Un archivo **tocado en vivo** en un teclado tiene cada nota un poco adelantada o atrasada, y eso es gran parte de lo que lo hace sonar vivo.

Así que una nota guarda sus ticks exactos en `nudge` y `trim`. ¿Por qué 24 por paso? Porque 24 se divide entre 2, 3, 4, 6, 8 y 12. Un tresillo de corcheas dura 32 ticks, uno de semicorcheas 16: los dos números enteros, así que los tresillos caen *exactos*. `NoteEvent::exact` construye una nota a partir de sus ticks exactos:

```rust
pub fn exact(pitch: Note, start: usize, ticks: usize) -> Self {
    let end = start + ticks.max(1);
    let first = start / TICKS_PER_STEP;
    let last = end.div_ceil(TICKS_PER_STEP);
    NoteEvent {
        nudge: (start % TICKS_PER_STEP) as u8,
        trim: (last * TICKS_PER_STEP - end) as u8,
        ..NoteEvent::new(pitch, first, last - first)
    }
}
```

El tresillo del tick 32 está en el paso 1 (`32 / 24`), desplazado 8 ticks (`32 % 24`). `div_ceil` divide redondeando hacia arriba, así que `last` es el paso siguiente a aquel en que termina la nota. El piano roll dibuja una nota desplazada donde empieza de verdad, el hilo de audio la toca ahí (El secuenciador), y **Herramientas > Cuantizar a los pasos** lleva las notas a la cuadrícula cuando las quieres ahí.



Emparejar note on con note off

Un archivo no guarda notas. Guarda *tecla abajo* y *tecla arriba*, y la importación tiene que emparejarlas. Mientras se lee una pista, cada canal MIDI guarda las notas que aún suenan en cada tecla:

```
struct ChannelState {
    /// For each key, the notes sounding on it, oldest first. There is
    /// usually one at most, but a file can strike a key again before
    /// letting it go; each note off then ends the oldest note.
    open: Vec<Vec<Open>>,
    pedal: bool,
    notes: Vec<Played>,
}
```

Una tecla puede tener dos notas abiertas: dos voces de una fuga en la misma nota, o un canon donde las partes se solapan. Entonces cada subida termina la nota **más antigua** de esa tecla, la primera que entró es la primera que sale. Terminar la más nueva cortaría notas allí donde las partes se solapan: los cánones de las demostraciones no volverían como están escritos.

Dos cosas más que la importación hace como lo haría un piano:

- **El pedal de sustain** (controlador 64). Mientras está abajo, una tecla soltada sigue sonando (`held_by_pedal`); la nota termina cuando sube el pedal, o cuando se vuelve a tocar la misma tecla. Sin esto, una pieza de piano tocada con pedal suena seca y entrecortada, porque los dedos dejaron las teclas mucho antes de que el sonido debiera parar.
- **Una nota soltada en el mismo tick en que empezó** (una nota de adorno, un golpe de batería) dura igualmente un tick, para que se oiga.

Levantar el pedal divide en dos la lista de cada tecla:

```
let (held, sounding): (Vec<Open>, Vec<Open>) =
    open.drain(..).partition(|note| note.held_by_pedal);
*open = sounding;
for note in held {
    self.end(key, note, tick);
}
```

`partition` manda cada elemento a una de dos colecciones, según el closure, y conserva su orden. `drain(..)` vacía la lista a medida que avanza, así que las notas se mueven, no se copian, y `*open = sounding` vuelve a poner las que siguen sonando. Las teclas aún pulsadas cuando termina una pista se sueltan ahí.

Una nota tocada dos veces en el mismo tick por error es una sola nota: las notas se ordenan con `sort_by_key(|note| (note.start, note.pitch, std::cmp::Reverse(note.end)))`, y `dedup_by_key` se queda con la primera de cada `(start, pitch)`. `Reverse` da la vuelta a un orden, así que la más larga de las dos va primero y es la que se queda.

Pistas, canales y batería

Cada canal MIDI de cada pista que toca notas se convierte en un canal del rack. La mayoría de los archivos son de **formato 1**, con una pista por instrumento, así que cada pista se convierte en un canal, con el nombre de la pista («Violin I»), o con el de su instrumento General MIDI cuando la pista no tiene nombre. Un archivo de **formato 0** lo pone todo en una pista, con los instrumentos en canales MIDI distintos; ahí, cada canal MIDI se convierte en un canal del rack. Un rack tiene 32 canales; cuando un archivo tiene más partes de las que caben junto a su batería, sus partes se juntan por canal MIDI.

El **program change** que eligió el instrumento de cada parte se guarda como el `program` del canal (SoundFonts y General MIDI), y su **volumen** (controlador 7) fija el volumen del canal, para que un acompañamiento suave siga siéndolo. Los programas y los volúmenes a veces se fijan en otra pista que las notas, así que el primero que se fija en cada canal en cualquier parte del archivo es el de reserva:

```
EventKind::Program { channel, program } => {
    programs[usize::from(channel)].get_or_insert(program);
}
```

`Option::get_or_insert` pone el valor solo si aún no hay ninguno, así que gana el primer program change sin un `if`.

El canal MIDI 10 es el canal de batería: cada tecla es un sonido de batería. La importación da a cada tambor incorporado que usa el archivo un canal propio, como en un proyecto nuevo, así que la batería aparece en el rack de canales como un secuenciador por pasos, cada golpe de un paso. Los sonidos que no tienen los tambores incorporados van al más cercano (`general_midi_drum_sound`: un triángulo al ride, una pandereta al shaker, los bongós a la conga); un sonido sin nada cercano, como un silbato, se deja fuera, y el aviso lo cuenta.

Unos pocos archivos de internet tienen millones de notas, escritos como proeza, y harían el piano roll demasiado lento para usarlo. La importación se queda con las 50 000 primeras (`MAX_NOTES`). `keep_first_notes` ordena los ticks de inicio, toma el inicio de la última nota que cabe como corte, y usa `partition_point` (una búsqueda binaria del primer elemento que no pasa una prueba) para contar cuántas notas del tick de corte aún caben.

Tempo, compás y tonalidad

Los cambios de tempo, los compases y las armaduras se aplican a toda la canción, sea cual sea la pista que los lleve. Una función genérica reúne los eventos de una clase de todas las pistas:

```
fn song_events<T>(tracks: &[Track], pick: impl Fn(&EventKind) ->
Option<T>) -> Vec<(u64, T)> {
    let mut events: Vec<(u64, T)> = tracks
        .iter()
        .flat_map(|track| &track.events)
        .filter_map(|event| pick(&event.kind).map(|value| (event.tick,
value)))
        .collect();
    // A stable sort keeps events on the same tick in the order they were
    // read, so the last one written wins.
    events.sort_by_key(|&(tick, _)| tick);
    events
}
```

`pick` dice qué eventos conservar y qué sacar de ellos, así que la misma función sirve para los tempos (`u32`), los compases y las tonalidades. Cada clase va luego a un `BTreeMap` por paso, donde un evento posterior en el mismo paso sustituye a uno anterior, como en la reproducción. Cada cambio de tempo se convierte en una marca de tempo, cada compás en una marca de compás (Tempo, compás, swing y sensación), y `tidy` deja fuera las marcas que no cambian nada: los archivos a menudo repiten el compás al comienzo de cada sección. Un compás que la aplicación no puede contar, con compases que no son un número entero de semicorcheas, se reduce cuando se puede: 6/32 pasa a 3/16.

La tonalidad sale de las armaduras, tal como están escritas: dos bemoles y menor es Sol menor, y los sostenidos o bemoles de la armadura deciden la escritura, así que seis bemoles son Sol♭ mayor y seis sostenidos Fa♯ mayor. Cada armadura posterior que nombra otra tonalidad se convierte en una marca de tonalidad en la lista de reproducción. La tonalidad exacta de un archivo que escribió Ferrous Lineup gana a la armadura que tiene al lado. Dos casos se dejan a las notas:

- **Un archivo sin ninguna armadura.**
- **Un archivo que dice Do mayor pero no lo es.** Muchos programas escriben «sin sostenidos, mayor» en cada archivo, lo haya elegido alguien o no. Así que cuando todas las armaduras del archivo dicen Do mayor, la importación lo contrasta con las notas, y si otra tonalidad encaja *claramente* mejor (por `CLEARLY_BETTER`, 0,1 de correlación) se queda con esa, y el aviso dice «deducido de las notas».

Cómo deciden las notas una tonalidad, con perfiles de tonalidad y el método de Krumhansl-Schmuckler, está en Oír la armonía. La importación suma cuánto suena cada una de las doce clases de altura, y le pregunta a `key_fits` qué tonalidad encaja mejor.

Cortar la canción en patrones

Un patrón tiene como mucho 256 pasos (16 compases de 4/4), y una canción puede durar 200 compases. Así que la importación corta la canción en patrones en las **barras de compás**, de 16 compases como mucho cada uno, y los coloca uno tras otro en la lista de reproducción.

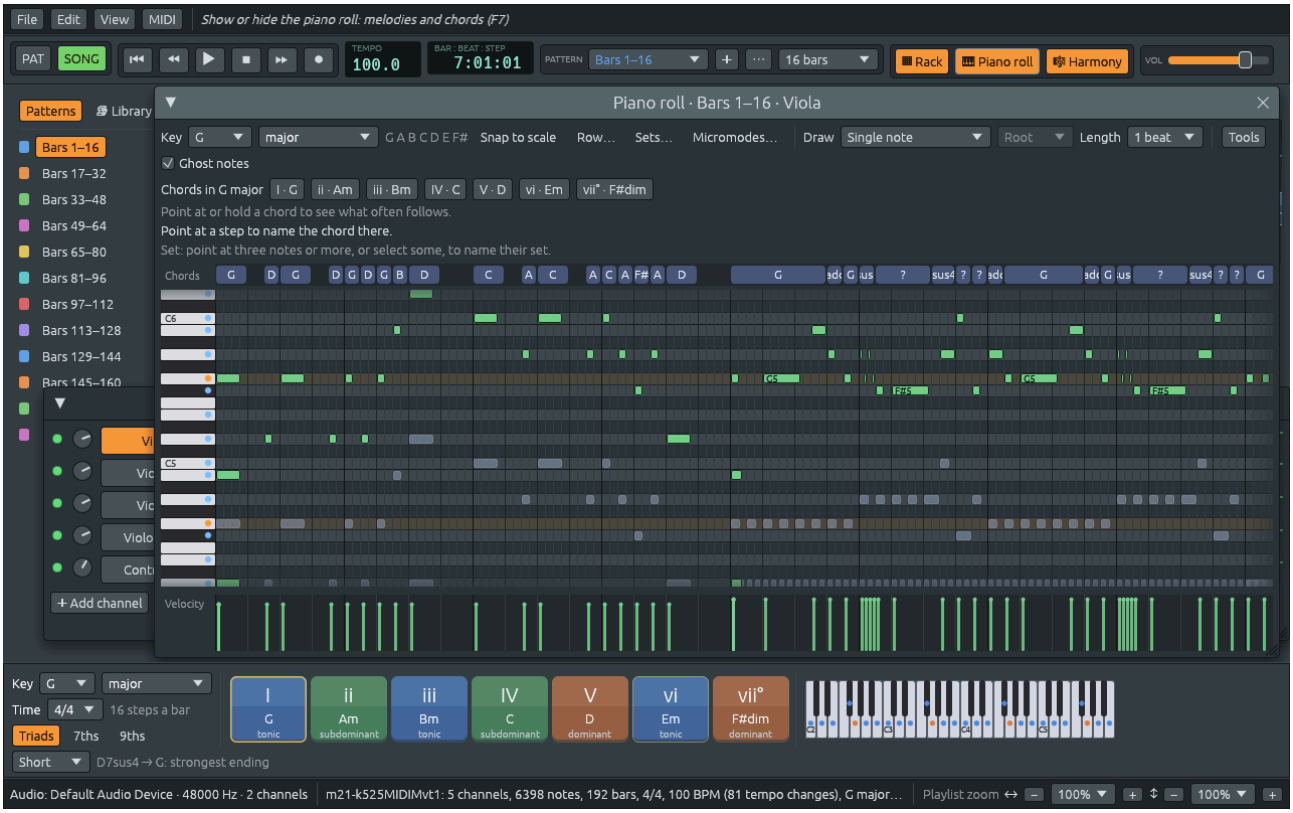
¿Dónde cortar? Una nota que suena a través del corte tiene que partirse en dos, lo que no afecta al sonido pero es feo de editar. Así que entre las barras de compás de la segunda mitad de cada tramo, `chunks` elige la que menos notas cruzan, la última de ellas. Para contarlas en todos los pasos a la vez usa un **array de diferencias**:

```
// How many notes cross each step's start: a difference array, +1
// where a note starts crossing and -1 after it stops.
let mut change = vec![0i64; length + 2];
for note in notes {
    let first = note.start / TICKS_PER_STEP + 1;
    let last = note.end.div_ceil(TICKS_PER_STEP).saturating_sub(1);
    if first <= last && first <= length {
        change[first] += 1;
        change[(last + 1).min(length + 1)] -= 1;
    }
}
let mut crossing = Vec::with_capacity(length + 1);
let mut count = 0;
for &delta in change.iter().take(length + 1) {
    count += delta;
    crossing.push(count);
}
```

Para cada nota, +1 en el primer límite de paso que cruza y -1 después del último; sumar los números desde la izquierda da, en cada paso, cuántas notas suenan a través de su comienzo. Es una pasada por las notas y otra por los pasos, en lugar de comprobar cada nota en cada paso. La elección es luego `(earliest..=last).rev().min_by_key(...)`: `min_by_key` devuelve el *primero* de varios mínimos iguales, y recorrer los compases hacia atrás hace que el primero sea el último compás.

`place_patterns` llena los patrones, partiendo las notas que cruzan un corte. Un tramo que vuelve **nota por nota** (una estrofa repetida, un da capo) toca otra vez el

mismo patrón en lugar de una copia, como construirías la canción a mano. Cada patrón lleva el nombre de sus compases, como «Compases 1–16».



Como las notas caen en patrones normales, todo lo que ya entiende de notas funciona con ellas: el carril de **Acordes** del piano roll nombra el acorde de cada paso, y las teclas de armonía iluminan el acorde que suena mientras se toca la canción.

Cómo se prueba

- `midi.rs`: `parse` con note-ons, note-offs, un note-on silencioso, y mensajes que debe ignorar (un pedal de sustain, un pitch bend, un tic de reloj, un mensaje demasiado corto, un byte de datos fuera de rango), y cada mensaje a bytes y de vuelta.
- `record.rs`: las reglas de cuantización, con posiciones inventadas.
- `engine.rs`: `keys_play_the_live_channel_until_they_come_up`, `a_key_comes_up_on_the_channel_it_went_down_on`, `a_key_up_leaves_the_sequencers_notes_alone`, `keys_from_a_midi_keyboard_skip_the_command_queue`, `removing_channels_lets_go_of_their_keys`, `midi_out_channels_send_their_notes`, `removing_a_midi_out_ends_its_notes`, y los dos tests de que un mensaje MIDI espera a que haya sitio en lugar de perderse, en orden.

- `instrument.rs`: note-offs tras el gate, una nota repetida, el program change antes de la primera nota y después de cada parada, un sintetizador o tambor enviado a MIDI que no suena y envía sus notas.
- `midi_file.rs`: los números de longitud variable de la especificación, y luego archivos enteros, leídos por un pequeño lector escrito en el módulo de tests: pistas, program changes, cambios de tempo, compases y armaduras, la tonalidad exacta, la batería en el canal 10, clips y repeticiones, una nota repetida, una sección tocada una vez, y los canales MIDI de cada clase de canal.
- `read.rs`: running status, metaeventos, system exclusive, un archivo RMID, ticks por segundo, y `a_file_cut_short_keeps_what_was_read`, que corta un archivo en *cada* longitud de 0 bytes al archivo entero y lee cada una sin un panic.
- `import.rs`: archivos pequeños hechos a mano para lo que una demostración nunca tiene: tresillos, el pedal de sustain, un archivo de formato 0, batería, tonalidades con y sin armadura, una canción larga cortada en barras de compás, un tramo que vuelve, un archivo sin notas.

La mejor prueba de una importación es una exportación. `every_demo_comes_back_note_for_note` exporta cada demostración de canción como archivo MIDI, la importa, exporta otra vez el resultado y compara los dos archivos evento a evento: cada note on con su tecla y su velocidad, cada note off, cada cambio de tempo, compás, armadura y program change, cada uno en el mismo tick. También comprueba que el proyecto importado tiene exactamente la tonalidad y las marcas de tonalidad de la demostración.

Aun así, un lector escrito por la misma persona que escribió el escritor puede compartir sus errores. Así que vale la pena abrir los archivos exportados en un programa que no sabe nada de Ferrous Lineup (la biblioteca `mido` de Python lee uno en pocas líneas), y archivos de otros programas en la aplicación.

Probar sin sintetizador

Un test puede comprobar los bytes que enviaría un instrumento, pero no que lleguen a un puerto. Para eso, Linux tiene el secuenciador de ALSA, y dos módulos del kernel que crean dispositivos MIDI de mentira: `snd-seq-dummy` (un puerto «Midi Through» que lo deja pasar todo) y `snd-virmidi` (puertos que parecen un teclado o un sintetizador USB). `aseqdump` imprime cada mensaje que llega a un puerto:

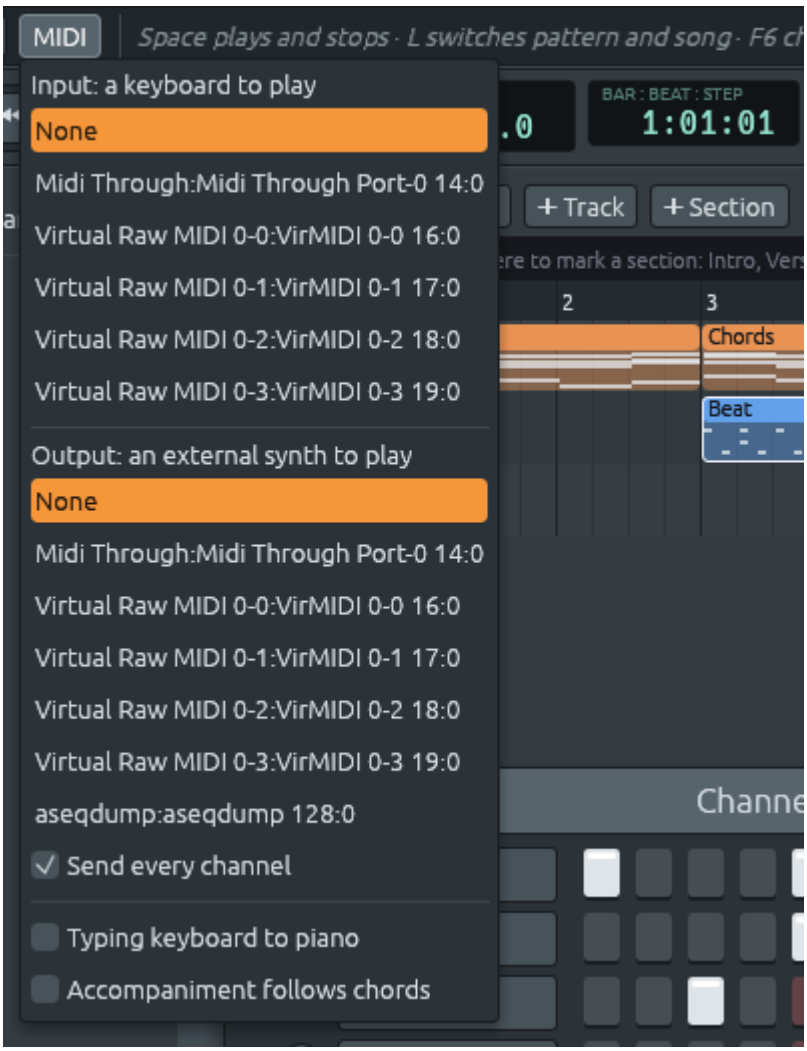
```
sudo modprobe snd-seq-dummy snd-virmidi
aseqdump -p "Midi Through"
```

Elige **Midi Through** en la **Salida** del menú **MIDI**, pulsa Reproducir, y cada nota aparece al enviarse:

```
129:0   Note on                 9, note 36, velocity 100
129:0   Note on                 0, note 60, velocity 100
129:0   Note off                9, note 36, velocity 64
```

`aseqdump` cuenta los canales desde 0, así que su 9 es el canal MIDI 10, la batería. Escribir bytes en el dispositivo en bruto de un puerto virtual toca una tecla en un teclado de mentira: elige **Virtual Raw MIDI 0-0** como entrada, y luego

```
printf '\x90\x45\x50' > /dev/snd/midiC0D0   # A4 down, velocity 80
printf '\x80\x45\x00' > /dev/snd/midiC0D0   # and up
```



Exportar: WAV, MIDI y video

Archivos: `crates/lineup-audio/src/render.rs`, `crates/lineup-app/src/app/export.rs`, `crates/lineup-video/`, `crates/lineup-app/src/video.rs` y `crates/lineup-app/src/app/video.rs`

El menú **Archivo** saca una canción de la aplicación de varias formas:

- **Exportar WAV…** escribe el sonido, tal como lo oyes cuando la canción suena.
- **Exportar video…** escribe un archivo MP4 con el sonido y las vistas que elijas siguiendo la música: la hoja de acordes, la partitura guía, la lista de reproducción, el piano roll, las teclas de armonía.
- **Exportar canción como MIDI…** y **Exportar patrón como MIDI…** escriben las notas, para otro programa de música. Cómo se hacen los bytes de un archivo MIDI está en MIDI.
- **Exportar hoja de acordes…** escribe los acordes como texto, y la partitura guía se imprime como PDF; ambas cosas están en los capítulos sobre la hoja de acordes y la partitura guía.

Cómo usarlas está en el capítulo Exportar del manual. Este capítulo trata de las dos exportaciones que tienen que *tocar* la canción: el archivo WAV y el video. Tres ideas las sostienen:

1. **El mismo código, más rápido que el tiempo real.** El hilo de audio es una función que llena un búfer. No sabe que quien le pide es una tarjeta de sonido. Puede pedirle un bucle, tan rápido como dé el ordenador, y escribir lo que sale en un archivo. No hay un segundo motor que mantener de acuerdo con el primero, así que una exportación suena exactamente como la reproducción.
2. **Una instantánea, en un hilo propio.** La exportación recibe su propia copia de la canción y sus propios instrumentos, y corre al lado de la ventana. Puedes seguir tocando y editando; la exportación sigue con la canción tal como estaba cuando la elegiste.
3. **Un archivo terminado o ninguno.** Toda exportación escribe en un archivo temporal y lo renombra al final, así que una exportación que falla o se detiene nunca deja medio archivo donde debería haber uno terminado.

El hilo de audio, sin tarjeta de sonido

Visto desde fuera, el hilo de audio es un método:

```
pub(crate) fn process<T>(&mut self, output: &mut [T])
where
    T: SizedSample + FromSample<f32>,
```

Llena `output` con los siguientes frames, en el tipo de muestra que pida la tarjeta de sonido. Con una tarjeta de sonido, cpal lo llama unos cientos de frames por vez, tan rápido como se reproducen. Nada en él espera al reloj: simplemente hace los siguientes frames. Así que un bucle que lo llama con un simple `Vec<f32>` obtiene la canción tan rápido como el procesador puede calcularla. Los tests del propio motor siempre han funcionado así: nunca abren una tarjeta de sonido.

`lineup-audio` envuelve esto en un `Renderer`:

```
pub struct Renderer {
    audio: AudioThread,
    sample_rate: u32,
    channels: usize,
    /// The audio thread is built with a command queue, but the renderer
    /// hands it commands directly. Keeping this end of the queue alive
    /// keeps the queue valid.
    _commands: rtrb::Producer<Command>,
    trash: rtrb::Consumer<Garbage>,
    /// Where playback is, which the audio thread stores after every
block.
    position: Arc<AtomicU64>,
    /// The trace being written, and its reader, once [`Renderer::trace`]
    /// has started one.
    trace: Option<(TraceReader, Trace)>,
}
```

`AudioThread` es `pub(crate)`: los otros crates no pueden construir uno, llamar a su `process` ni meterse dentro. Reciben el `Renderer`, cuyos métodos respetan las reglas del hilo de audio. Mantener la exportación *dentro* de `lineup-audio` es lo que permite que el motor siga siendo privado.

`AudioThread::new` quiere el extremo receptor de una cola de comandos, porque en el hilo de audio real así llegan los comandos. El renderer no tiene otro hilo del que recibir. Crea una cola con espacio para un comando:

```
let (commands, command_consumer) = rtrb::RingBuffer::new(1);
```

y nunca la usa. Los comandos entran directamente:

```
/// Applies a command at once: there is no other thread to hand it to.
/// Volume changes take effect at once too, without smoothing: the
/// commands come before the first sample is made.
pub fn send(&mut self, command: Command) {
    self.audio.apply(command);
    self.audio.settle_volumes();
    self.empty_trash();
    self.read_trace();
}
```

El campo que guarda el otro extremo de la cola es `_commands`. Un guion bajo al principio le dice al compilador (y al lector) que el campo no se lee nunca a propósito, así que no avisa «field `commands` is never read». El campo igualmente guarda el productor mientras vive el renderer, así que la cola del hilo de audio tiene sus dos extremos, como en la aplicación.

Los otros métodos son igual de cortos:

```
/// Fills `output` with the next frames, one sample per channel each.
pub fn render(&mut self, output: &mut [f32]) {
    self.audio.process(output);
    self.empty_trash();
    self.read_trace();
}
```

`read_trace` pertenece a la traza de reproducción, que usa el mismo renderer para anotar cada nota en lugar del sonido.

Dos detalles que solo aparecen fuera de línea

El suavizado. Los cambios de volumen se deslizan durante unos milisegundos (`Smoothed`), porque un salto brusco produce un clic audible. Con una tarjeta de sonido, ese deslizamiento ocurre mientras suena la música. En una exportación, fijar el volumen principal justo antes de la primera muestra haría que los primeros milisegundos del archivo se deslizaran del volumen por defecto al tuyo. Por eso `send` llama a `settle_volumes`, que pone cada valor suavizado en su objetivo de una vez:

```
pub(crate) fn settle_volumes(&mut self) {
    self.master_volume.settle();
    self.reverb.settle();
    for channel in &mut self.channels {
        channel.volume.settle();
        channel.pan.settle();
        channel.reverb_send.settle();
    }
}
```

```
/// Jumps straight to the target, with no smoothing.
pub fn settle(&mut self) {
    self.current = self.target;
}
```

El primer test del renderer, `renders_like_the_audio_thread`, detectaría que falta ese ajuste: fija el volumen principal y comprueba que el primerísimo frame de un clic tiene el nivel completo.

La basura. El hilo de audio real nunca libera memoria. Lo que tira (una lista vieja de notas, un canal eliminado) va a la cola de basura, y el hilo de la interfaz lo libera una vez por frame de la ventana. El renderer no es un hilo de tiempo real, así que puede liberar las cosas en el acto, y vacía la cola después de cada comando y de cada bloque:

```
/// Frees what the audio thread threw away. There is no real-time
/// thread here, so it can be done straight away.
fn empty_trash(&mut self) {
    while self.trash.pop().is_ok() {}
}
```

`pop` devuelve cada valor, y soltarlo lo libera. El cuerpo del bucle está vacío porque no hay nada más que hacer con ellos.

Poner la canción en el renderer

Una exportación construye su renderer con los mismos comandos que la aplicación envía al arrancar y después de abrir un archivo:

```
/// The whole song, ready for `renderer` to play from the start.
fn song_render_with(&mut self, mut renderer: Renderer) -> SongRender {
    let sample_rate = renderer.sample_rate();
    for command in self.project_commands(sample_rate, 1.0) {
        renderer.send(command);
    }

    // The song loops when it reaches its end, so silent steps are
    // added after it: enough for the tail, and one more, so the
    // export ends before the song would start again.
    // The whole song, every repeat included, even when song mode only
    // loops part of it.
    let song = self.project.arrangement();
    let song_length = song.length;
    // The silent steps play at the tempo the song ends at: the
    // audio thread plays steps past the end of the tempo list at
    // its last tempo.
    let tempos = self.project.played_tempos(&song);
    let end_tempo = tempos.last().copied().unwrap_or(self.project.tempo);
    let step_frames = frames_per_step(end_tempo, sample_rate);
    let reverb = self.project.mixer.reverb;
    let tail_seconds = if reverb.enabled {
        TAIL_SECONDS + f64::from(reverb.decay + reverb.pre_delay_ms /
1000.0)
    } else {
        TAIL_SECONDS
    };
    let tail_frames = tail_seconds * f64::from(sample_rate);
    let silent_steps = (tail_frames / step_frames).ceil() as usize + 1;
    let song_frames = self.project.played_seconds(&song) *
f64::from(sample_rate);
    renderer.send(Command::SetSong {
        plays: song.plays,
        length: song_length + silent_steps,
    });
    renderer.send(Command::SetSongTempos(tempos));
    renderer.send(Command::SetMode(PlayMode::Song));
    renderer.send(Command::Play);
    SongRender {
        renderer,
        frames: (song_frames + tail_frames) as usize,
    }
}
```

`project_commands` enumera el tempo y el tiempo suelto, el volumen principal y la reverberación, cada patrón (`InsertPattern`), cada canal (`InsertChannel` , con su instrumento y sus notas en todos los patrones, y luego su balance, su envío a la reverberación y sus efectos) y la canción. El segundo argumento es la velocidad de práctica. La reproducción puede hacer más lenta la canción para practicar, pero una exportación va siempre a la velocidad propia de la canción, así que pasa `1.0` .

`send_whole_project`, que habla con el hilo de audio real, llama a la misma función con la velocidad de práctica, así que ambos se construyen igual.

Lo que toca la exportación se deduce de esos comandos:

- **Todo lo que oyes**: la banda, los volúmenes del mezclador, el balance, la reverberación y los efectos. Los canales y pistas silenciados siguen callados, como cuando pulsas Reproducir. Con la SoundFont activada, suenan sus instrumentos.
- **La canción entera**, cada repetición y cada vuelta de un estribillo, aunque una región de bucle haga que el modo canción toque solo una parte: `arrangement()` es la canción entera, mientras que Reproducir usa `playback_arrangement()`.
- **Sin metrónomo.** `project_commands` nunca envía `Command::SetMetronome`, y un hilo de audio nuevo empieza con él apagado.

Los instrumentos se hacen para la frecuencia de muestreo de la exportación, 44.100 frames por segundo, sea cual sea la de tu tarjeta de sonido. La batería incorporada se sintetiza de nuevo a esa frecuencia, el Lineup Osc se hace para ella, y los samplers toman sus sonidos de la `SampleCache`, así que no se vuelve a leer ningún archivo.

El renderer tiene sus propios instrumentos y su propia copia de cada nota. Es una instantánea, como las copias del historial de deshacer: la exportación sigue con la canción tal como estaba, hagas lo que hagas en la ventana mientras tanto.

¿Cuánto dura una canción?

La duración de la canción se conoce en pasos; la del archivo, en frames. `played_seconds` suma cada paso al tempo vigente en él, así que una canción con cambios de tempo, ritardandos y calderones también obtiene la duración correcta. Después viene una **cola** de dos segundos, para dejar sonar las últimas notas. Con la reverberación activada, la cola espera además su decaimiento y su pre-delay, para que la sala se apague en lugar de cortarse.

La canción inicial tiene 128 pasos a 130 BPM. Hay 4 pasos por pulso, así que un paso dura 60 / 130 / 4 = 0,115 segundos, 5.088,46 frames a 44.100 (un paso no es un número entero de frames; el secuenciador conserva la fracción). Los 128 pasos ocupan 651.323 frames; con una cola de dos segundos, 88.200, el archivo tiene 739.523 frames, 16,77 segundos.

Pasos en silencio después del final

Hay una trampa. En modo canción, el hilo de audio *repite en bucle*: cuando el cursor llega al final de la canción, vuelve a empezar desde el compás 1. La cola terminaría con el primer acorde de la canción empezando otra vez. La solución es decirle a este renderer, y solo a este, que la canción es más larga de lo que es, con algunos pasos en silencio. Para la canción inicial son 88.200 / 5.088,46 = 17,33, redondeado hacia arriba a 18, más uno: 19 pasos en silencio.

¿Por qué el paso de más? Un paso no es un número entero de frames, así que el secuenciador empieza algunos pasos un frame antes o después de la fracción exacta. El paso de más asegura que el final del archivo esté siempre a un paso entero del bucle, se redondee como se redondee.

Los pasos en silencio suenan al tempo con el que termina la canción. El hilo de audio toca los pasos que quedan más allá del final de la lista de tempos a su último tempo, así que una canción que termina con un ritardando recibe una cola medida en sus lentos pasos finales, no en los más rápidos del principio.

Escribir un archivo WAV

Un archivo WAV es una cabecera corta (aquí, 44 bytes) que dice cómo se guarda el sonido, seguida de las muestras. La exportación escribe el tipo más común:

- **44.100 frames por segundo**, la frecuencia de un CD, que todo programa entiende,
- **2 canales** (estéreo), guardados **intercalados**: izquierdo, derecho, izquierdo, derecho…, exactamente como los búferes que cpal le da al hilo de audio,
- **16 bits por muestra**: cada muestra es un `i16`, un número entero de -32.768 a 32.767.

Eso da 44.100 × 2 × 2 = 176.400 bytes por segundo de sonido, unos 10,6 MB por minuto. El archivo de la canción inicial ocupa 44 + 739.523 × 4 = 2.958.136 bytes.

Escribir la cabecera y las muestras a mano sería fácil, pero el crate `hound` ya lo hace, y el sampler lo usa de todos modos para *leer* archivos WAV:

```rust
pub fn export_wav_cancellable(
    renderer: &mut Renderer,
    path: &Path,
    frames: usize,
    mut progress: impl FnMut(usize),
    cancel: &AtomicBool,
) -> Result<WavExport, hound::Error> {
    let spec = hound::WavSpec {
        channels: renderer.channels() as u16,
        sample_rate: renderer.sample_rate(),
        bits_per_sample: 16,
        sample_format: hound::SampleFormat::Int,
    };
    let mut temporary = PathBuf::from(path);
    temporary.as_mut_os_string().push(".tmp");

    let mut writer = hound::WavWriter::create(&temporary, spec)?;
    let mut block = vec![0.0_f32; BLOCK_FRAMES * renderer.channels()];
    let mut done = 0;
    while done < frames {
        if cancel.load(Ordering::Relaxed) {
            drop(writer);
            let _ = fs::remove_file(&temporary);
            return Ok(WavExport::Cancelled);
        }
        let count = (frames - done).min(BLOCK_FRAMES);
        let samples = &mut block[..count * renderer.channels()];
        renderer.render(samples);
        for &sample in samples.iter() {
            writer.write_sample(to_i16(sample))?;
        }
        done += count;
        progress(done);
    }
    writer.finalize()?;
    fs::rename(&temporary, path)?;
    Ok(WavExport::Written)
}
```

El sonido se hace en bloques de 4.096 frames (`BLOCK_FRAMES`), en un único búfer que se reutiliza para cada bloque. El último bloque suele ser más corto, así que `samples` es un slice con solo la parte necesaria. Después de cada bloque se llama a `progress` con el número de frames hechos. Su tipo es `impl FnMut(usize)`: cualquier closure que reciba un número y pueda cambiar lo que capturó. El closure de la aplicación guarda el número donde la barra de estado puede leerlo.

Cada `?` aquí devuelve un `hound::Error`. `fs::rename`, en cambio, devuelve un `std::io::Error`, y `?` funciona igual, porque `hound::Error` implementa `From<std::io::Error>`: el operador `?` convierte el error a la salida.

`export_wav` es un envoltorio fino para quien no puede cancelar. Pasa un indicador que nunca se activa y devuelve un simple `Result<(), hound::Error>`; lo usan los tests. La aplicación usa la versión que se puede cancelar.

Escribir en un archivo temporal y después renombrar

Las muestras van primero a `song.wav.tmp`, y solo cuando todo está escrito se renombra a `song.wav`. Así se guarda también el archivo del proyecto. Exportar encima de un `song.wav` anterior deja el archivo viejo entero hasta que el nuevo está completo: si el disco se llena o algo falla a mitad de camino, te queda tu exportación anterior, no la mitad de una nueva.

`as_mut_os_string().push(".tmp")` añade al nombre. `set_extension("tmp")` *reemplazaría* `.wav`, dando `song.tmp`, que podría ser el nombre del archivo de otra persona.

De `f32` a `i16`

```
/// Turns a sample between -1.0 and 1.0 into a 16-bit integer sample.
fn to_i16(sample: f32) -> i16 {
    (sample.clamp(-1.0, 1.0) * f32::from(i16::MAX)).round() as i16
}
```

`clamp` mantiene la muestra entre -1,0 y 1,0; después se escala a -32.767..32.767 y se redondea al entero más cercano. Una conversión con `as` de un número de coma flotante a un entero ya se detiene en los límites del entero en Rust (*satura*: `3e9 as i16` es 32.767), así que el `clamp` no hace falta por seguridad, pero dice lo que se quiere. `f32::from(i16::MAX)` es una conversión que nunca puede perder nada, así que es un `From`, no un `as`.

El hilo de audio ya mantiene la salida principal dentro de -1,0..1,0 con su limitador suave (`tanh`), así que en la práctica el `clamp` nunca recorta nada.

Un hilo propio

Renderizar lleva un rato. Con los sonidos incorporados, una compilación de release en un ordenador de cuatro núcleos toca el Boléro (catorce minutos, para una orquesta completa) a través de un renderer en unos veinte segundos, unas cuarenta veces más rápido de lo que suena; la canción inicial tarda una fracción de segundo. Una

compilación de depuración es muchas veces más lenta. Mientras tanto la ventana no debe congelarse, así que la exportación corre en un **hilo** propio:

```
let done = Arc::new(AtomicUsize::new(0));
let cancel = Arc::new(AtomicBool::new(false));
// The thread gets its own `Arc`s pointing to the same counter and
// flag, and its own copy of the path. `move` gives the closure
// ownership of them, and of the renderer: from now on they belong
// to the thread.
let worker = {
    let done = Arc::clone(&done);
    let cancel = Arc::clone(&cancel);
    let path = path.clone();
    thread::spawn(move || {
        match export_wav_cancellable(
            &mut renderer,
            &path,
            frames,
            |frames_done| done.store(frames_done, Ordering::Relaxed),
            &cancel,
        ) {
            Ok(WavExport::Written) => Ok(()),
            Ok(WavExport::Cancelled) => Err(CANCELLED.to_string()),
            Err(err) => Err(err.to_string()),
        }
    })
};
self.export = Some(Export::new(path, frames, done, Some(cancel), worker));
```

`std::thread::spawn` arranca un hilo nuevo que ejecuta un closure, y le pide dos cosas, las mismas dos que cpal le pide al callback de audio:

- `'static`: el closure no puede tomar prestado nada del código que lo arrancó. El hilo nuevo puede sobrevivir a la función que lo creó (sin duda sobrevive a este frame de la interfaz), así que una referencia prestada podría apuntar a algo que ya no existe. Por eso el closure lleva `move`: toma *la propiedad* de todo lo que usa (el renderer, la ruta, el contador, el indicador) en lugar de tomarlo prestado.
- `Send`: todo lo que se mueve a él tiene que poder pasarse a otro hilo sin peligro. El renderer cumple, porque todo lo que contiene cumple: los instrumentos, los `Arc` con las muestras, los extremos de los ring buffers. Si contuviera un `Rc` (un contador de referencias que no es seguro entre hilos), esta línea no compilaría.

El bloque `{ let done = Arc::clone(&done); ... }` es un truco habitual. Tanto la interfaz como el hilo necesitan el contador, así que cada uno recibe su propio `Arc` que apunta a él. Dentro del bloque, el `done` nuevo *oculta* (shadowing) al de fuera, y el closure `move` se lleva el de dentro, dejando el `done` de fuera para la interfaz. Lo mismo pasa con `cancel` y con `path`.

El progreso: un contador atómico

El hilo escribe cuántos frames están hechos en un `AtomicUsize`, y la barra de estado lo lee cuando dibuja la barra de progreso, como la ventana lee el cursor de reproducción. `Ordering::Relaxed` basta: ningún otro dato depende del número, y una barra un frame atrasada no importa. El archivo solo se da por bueno cuando el hilo ha terminado, y `join` (abajo) da la garantía completa.

Saber cuándo ha terminado

`spawn` devuelve un `JoinHandle`. La aplicación lo guarda, junto con el contador, en un `Export`:

```
pub(super) struct Export {
    path: PathBuf,
    /// How much there is to do: frames of sound, or pictures.
    frames: usize,
    /// How many are done. The export thread writes it and the UI
    /// thread reads it; an atomic needs no lock for that.
    done: Arc<AtomicUsize>,
    /// Set to ask the thread to stop, for exports that can be cancelled.
    cancel: Option<Arc<AtomicBool>>,
    /// The thread doing the work. Joining it gives back what its closure
    /// returned: here, whether the file was written.
    worker: JoinHandle<Result<(), String>>,
}
```

La exportación de video usa la misma estructura, contando imágenes en lugar de frames. `JoinHandle<Result<(), String>>` dice qué devuelve el closure del hilo: `Ok(())` si se escribió el archivo, o un mensaje. El error se convierte en `String` en el hilo de la exportación (`err.to_string()`), ya que un mensaje para la barra de estado es todo lo que la ventana necesita. Una exportación detenida también vuelve como `Err`, con la cadena especial `CANCELLED`.

En cada frame, la aplicación comprueba cómo va:

```
pub(super) fn check_export(&mut self, ctx: &egui::Context) {
    let Some(export) = &self.export else {
        return;
    };
    if !export.worker.is_finished() {
        ctx.request_repaint_after(Duration::from_millis(100));
        return;
    }
    // `take` moves the export out of the `Option` and leaves `None`,
    // so `join` can take ownership of the thread's handle.
    if let Some(export) = self.export.take() {
        let name = export.file_name();
        match export.worker.join() {
            Ok(Ok(())) => self.notice = Some(trf("Exported {}", &
[&name])),
            Ok(Err(err)) if err == CANCELLED => {
                self.notice = Some(trf("Stopped exporting {}", &[&name]));
            }
            Ok(Err(err)) => {
                self.message = Some(trf("Could not export {}: {}", &
[&name, &err]));
            }
            // `join` fails only if the thread panicked.
            Err(_) => self.message = Some(trf("Exporting {} failed", &
[&name])),
        }
    }
    // ...and closes the window, if a quit was waiting for the export.
}
```

Algunas cosas en las que fijarse:

- egui solo dibuja un frame nuevo cuando pasa algo: un movimiento del ratón, una tecla. Mientras corre una exportación puede que no pase nada, y la barra de progreso se quedaría quieta. `request_repaint_after(100 ms)` pide otro frame pronto, diez veces por segundo, hasta que la exportación termina.
- `is_finished()` pregunta sin esperar. `join()` *espera* a que el hilo termine, pero solo se llama cuando `is_finished()` es verdadero, así que vuelve en el acto.
- `join` toma el handle por valor: lo consume, ya que un hilo solo se puede unir una vez. El handle está dentro de `self.export`, un `Option`, y no se puede sacar algo de un campo que solo se tiene prestado. `Option::take` saca el valor y deja `None` en su lugar, así que la exportación desaparece de `self` y pasa a ser de la variable local `export`.
- El `match` tiene dos capas: `join` devuelve un `Result` que es `Err` solo si el hilo entró en pánico, y dentro de su `Ok` está el propio `Result` del closure. Los patrones se anidan, así que `Ok(Ok(()))` coincide con «el hilo terminó normalmente y el archivo se escribió». El brazo con `if err == CANCELLED` es

una *guarda de match*: solo coincide cuando se cumple la condición, y el brazo siguiente se queda con los demás errores.

Detener una exportación

Mientras corre una exportación, la barra de estado muestra una barra de progreso, **Exportando song.wav**, y un pequeño botón **Detener**:

```
if let Some(export) = &self.export {
    ui.separator();
    ui.add(
        egui::ProgressBar::new(export.progress())
            .desired_width(160.0)
            .text(trf("Exporting {}", &[&export.file_name()])),
    );
    if export.can_cancel() && ui.small_button(tr("Stop")).clicked() {
        export.cancel();
    }
}
```

`cancel` guarda `true` en el `AtomicBool`, y el hilo de la exportación lo comprueba antes de cada bloque, como muestra `export_wav_cancellable` más arriba. Una exportación detenida suelta el escritor, borra su archivo temporal y devuelve `WavExport::Cancelled`, así que nunca queda un WAV a medio escribir donde debería haber uno terminado. Primero se suelta el escritor, lo que cierra el archivo: Windows no borra un archivo que sigue abierto.

Cerrar la ventana mientras corre una exportación pregunta antes: **Una exportación aún se está ejecutando**, con **Detener y salir** y **Continuar exportando**. Funciona como la pregunta por los cambios sin guardar, que va primero si los hay: la aplicación cancela el cierre con `ViewportCommand::CancelClose`, y una ventana modal pregunta. Elegir **Detener y salir** activa el mismo indicador y recuerda que hay un cierre esperando; cuando `check_export` ve que el hilo terminó, cierra la ventana. Esperar al hilo le da tiempo para borrar su archivo temporal antes de que termine el proceso.

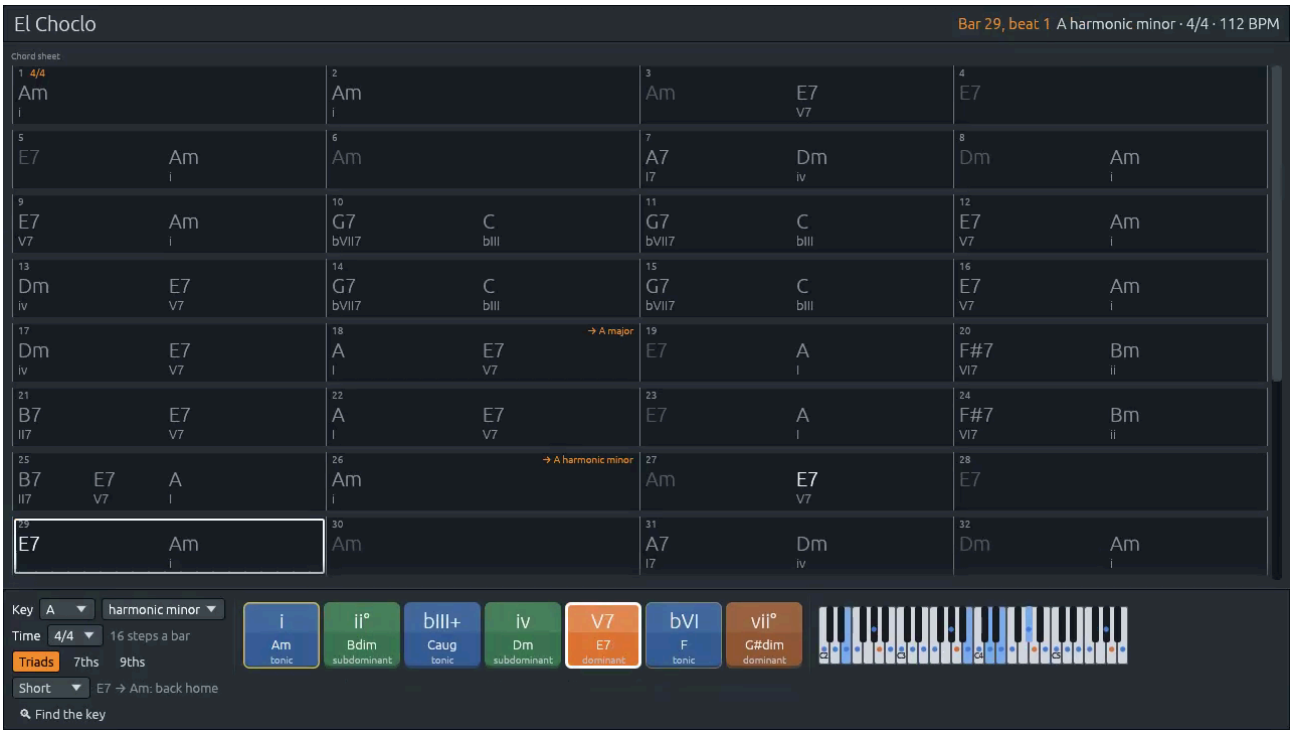
Archivos MIDI: se escriben al instante

Un archivo MIDI guarda solo las notas, unos pocos bytes cada una. Una canción larga entera ocupa unos pocos kilobytes, hechos en un momento, así que **Exportar canción como MIDI…** y **Exportar patrón como MIDI…** no necesitan hilo ni barra de progreso: preguntan dónde guardar y escriben los bytes con `std::fs::write`. Cómo `song_to_midi` y `pattern_to_midi` convierten las notas en esos bytes está en MIDI.

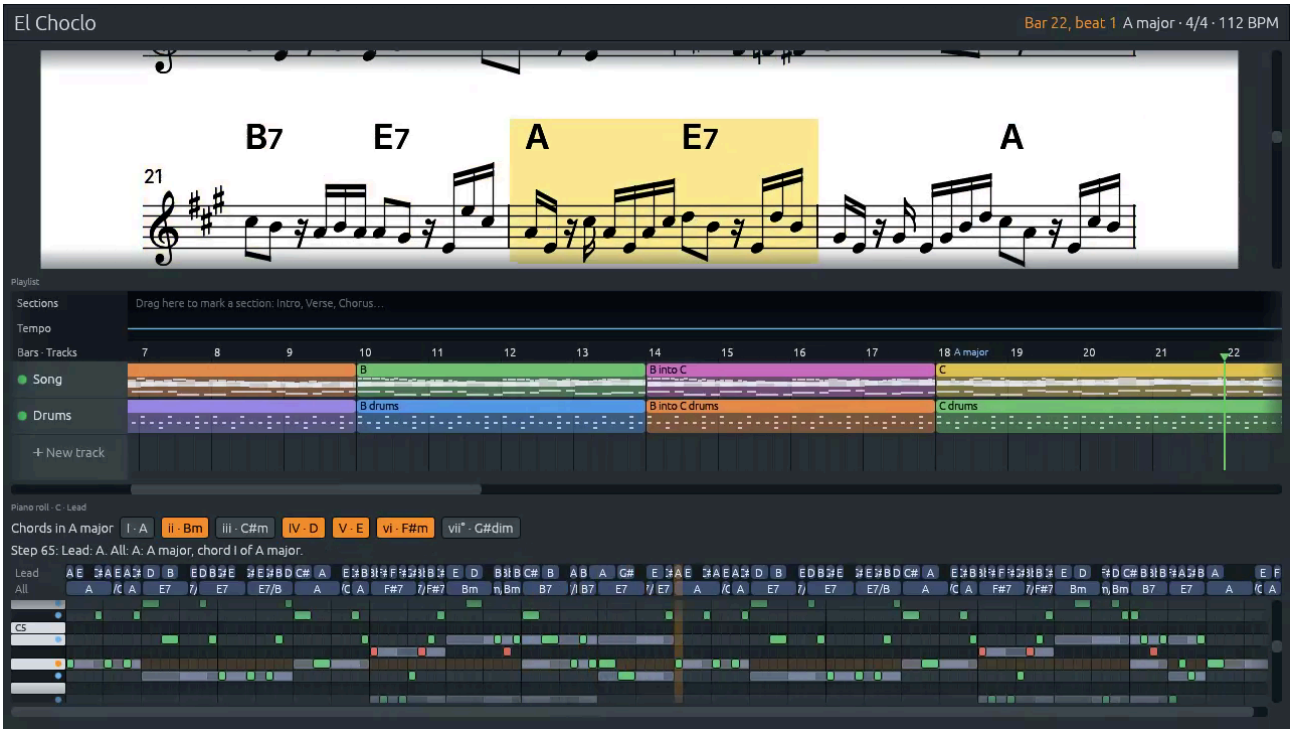
Toda exportación propone un nombre de archivo hecho a partir del del proyecto: `FileState::export_name` convierte `song.lineup` en `song.wav`, `song.mp4` o `song.mid`, o propone `Sin título.wav` para un proyecto que nunca se guardó. El archivo MIDI de un patrón lleva el nombre del patrón, con cada `/` o `\` reemplazado, ya que un nombre de archivo no puede contenerlos. Si escribes un nombre sin extensión, se añade la correcta.

Un video: la canción con las vistas acompañando

Un archivo WAV lleva la canción, pero no la armonía que ves mientras suena. **Exportar video…** guarda un archivo MP4 con el sonido y las vistas que elijas siguiendo la música: el compás que suena enmarcado en la hoja de acordes, el acorde iluminado en las teclas de armonía, el cursor avanzando por la lista de reproducción y el piano roll, y cada cambio de tonalidad mostrado cuando llega.



Un diálogo pregunta qué vistas mostrar y qué tamaño tienen las imágenes: **1280 × 720** (HD) o **1920 × 1080** (Full HD). Su vista previa muestra una imagen del video en el compás que elijas con **Vista previa en**. En ella, el cuadro de cada vista se puede arrastrar, redimensionar por su esquina y ampliar girando la rueda encima, del 50 % al 250 %. **Apilar** vuelve a poner las vistas una sobre otra.



Hacer un video son cuatro trabajos, y las secciones siguientes los toman uno por uno:

1. **Dibujar** las vistas como imágenes, sin ventana ni tarjeta gráfica.
2. **Comprimir** las imágenes como H.264 y el sonido como MP3.
3. **Poner ambos** en un archivo MP4.
4. **Mantener la imagen con el sonido**, con cambios de tempo y todo.

Nada que instalar

La forma habitual de hacer un video desde un programa es pasarle imágenes a FFmpeg. Pero FFmpeg es un programa aparte que Windows y macOS no traen, así que cada usuario tendría que instalarlo. En cambio, la aplicación compila dos codificadores muy conocidos a partir de su código fuente en C junto con ella misma, a través de dos crates:

Qué	Formato	Codificador	Crate
Imágenes	H.264, 30 por segundo	OpenH264 de Cisco (licencia BSD)	`openh264`
Sonido	MP3, 256 kbit/s estéreo	LAME (LGPL)	`mp3lame-encoder`
Ambos juntos	MP4	el nuestro, unos cientos de líneas	(`lineup-video`)

Un crate con C dentro tiene un `build.rs` que ejecuta un compilador de C antes de que Rust compile el resto, con el crate `cc`. La CI compila en Linux, Windows y los dos

tipos de Mac, así que demuestra que funciona en todas partes. Ambos codificadores, y el crate que pinta las imágenes, se optimizan incluso en las compilaciones de depuración, como el sintetizador de SoundFonts, por la razón que se da en Compilaciones de depuración.

H.264 y MP3 no son los formatos más nuevos. Se eligieron porque todo reproductor, teléfono, navegador y sitio de videos los lee.

El crate: `lineup-video`

El crate no sabe nada de canciones. Recibe imágenes y sonido y escribe un archivo. Sus únicas dependencias son los dos codificadores y `epaint`, la parte de egui que describe las formas. Tiene cuatro módulos:

- `canvas` pinta las formas de egui en una imagen en memoria.
- `h264` comprime cada imagen.
- `mp3` comprime el sonido.
- `mp4` escribe el archivo.

y un tipo que los usa a todos, `VideoWriter`. Le das una imagen, luego el sonido que suena mientras se muestra, y así hasta el final:

```
let mut writer = VideoWriter::create(&path, settings)?;
```

```
match scene.draw(time, position) {
    Some(picture) => writer.add_picture(picture)?,
    None => writer.repeat_picture(),
}
renderer.render(&mut sound);
writer.add_sound(&sound)?;
```

```
match record() {
    Ok(()) => writer.finish(),
    Err(err) => {
        writer.cancel();
        Err(err)
    }
}
```

`create` escribe en `nombre.mp4.tmp`, `finish` lo renombra y `cancel` lo borra, como con un archivo WAV. `cancel(self)` toma el escritor por valor, así que un escritor cancelado no se puede volver a usar: el compilador te lo impide.

Dentro de un archivo MP4

Un archivo MP4 es un árbol de **cajas** (boxes). Cada caja es una longitud de 4 bytes, un nombre de 4 letras y su contenido, que puede ser más cajas:

```
ftyp                "esto es un archivo MP4"
mdat                todas las imágenes y el sonido comprimidos, uno tras
otro
moov                el índice:
  mvhd                cuánto dura el conjunto
  trak                las imágenes:
    tkhd, mdia …        tamaño, reloj y las tablas de abajo
      avc1 + avcC         "H.264", y los conjuntos de parámetros del
codificador
      stts                cuánto tiempo se muestra cada imagen
      stss                qué imágenes son imágenes clave
      stsz, stco          qué tamaño tiene cada una y dónde está en mdat
  trak                el sonido:
    edts/elst           dónde empieza de verdad el sonido (ver abajo)
    mp4a + esds         "MP3"
    stts, stsz, stco    las mismas tablas para los frames MP3
```

Una imagen o un trozo de sonido en `mdat` es una *muestra* (sample), y las tablas de `moov` enumeran cada muestra de una pista. El formato es ISO/IEC 14496-12 y 14496-14; solo se escribe lo que necesita un video con una pista H.264 y una pista MP3.

Una caja cuyo tamaño va al final

Una caja empieza con su tamaño, pero el tamaño solo se conoce cuando su contenido está escrito. `write_box` deja sitio para él, deja que un closure escriba el contenido, y después lo rellena:

```
/// Writes a box: its size, its name, then whatever `contents` writes.
/// The size is only known afterwards, so it is filled in at the end.
fn write_box(out: &mut Vec<u8>, name: &[u8; 4], contents: impl FnOnce(&mut
Vec<u8>)) {
    let start = out.len();
    out.extend_from_slice(&[0; 4]);
    out.extend_from_slice(name);
    contents(out);
    let size = (out.len() - start) as u32;
    out[start..start + 4].copy_from_slice(&size.to_be_bytes());
}
```

Las cajas dentro de cajas son entonces closures dentro de closures, y el código tiene la forma del árbol:

```
write_box(out, b"trak", |out| {
    write_tkhd(out, 1, duration_ms, false, format.width, format.height);
    write_box(out, b"mdia", |out| {
        write_mdhd(out, format.timescale, self.video.duration());
        write_hdlr(out, b"vide", "Pictures");
```

`b"trak"` es una *cadena de bytes*: un `&[u8; 4]`, cuatro bytes, no un `&str`. El tipo del parámetro, `&[u8; 4]`, hace que el compilador compruebe que cada nombre tiene exactamente cuatro letras. `to_be_bytes` da los bytes de un número en *big-endian*, el más significativo primero, que es como MP4 (y MIDI) guardan los números.

El índice va al final

`moov` tiene que ir después de los datos. Hasta el final, nadie sabe qué tamaño tiene cada imagen ni dónde quedó. Así que `Mp4Writer` escribe `mdat` sobre la marcha, guarda una lista de tamaños y posiciones, y escribe `moov` en `finish`. Después vuelve atrás para rellenar la longitud de `mdat`:

```
// A size of 1 says the real size follows as 64 bits, so the file
// can grow past 4 GB. It is filled in at the end.
head.extend_from_slice(&1_u32.to_be_bytes());
head.extend_from_slice(b"mdat");
head.extend_from_slice(&0_u64.to_be_bytes());
```

```
self.file.seek(SeekFrom::Start(self.mdat_start + 8))?;
self.file
    .write_all(&(self.position - self.mdat_start).to_be_bytes())?;
self.file.seek(SeekFrom::Start(self.position))?;
```

Un video largo en Full HD puede pasar de 4 GB, así que el tamaño es de 64 bits. Por la misma razón, la tabla de dónde empieza cada muestra es `stco` (posiciones de 32 bits, que todo reproductor lee) mientras el archivo es lo bastante pequeño, y `co64` en cuanto una posición pasa de `u32::MAX`.

Cuánto se muestra cada imagen

`stts` guarda *tramos*: «300 imágenes de 1/30 s cada una» ocupa una entrada, no 300:

```
// How long the samples last, as runs of samples of the same length.
let mut runs: Vec<(u32, u32)> = Vec::new();
for &duration in &track.durations {
    match runs.last_mut() {
        Some((count, last)) if *last == duration => *count += 1,
        _ => runs.push((1, duration)),
    }
}
```

`last_mut` da una referencia mutable al último tramo, si lo hay. La guarda de match `if *last == duration` lo alarga cuando la nueva duración es la misma; si no, empieza un tramo nuevo.

El reloj de las imágenes va a 90.000 tics por segundo, el habitual en video, que se divide exactamente en 24, 25, 30 o 60 imágenes por segundo. A 30 por segundo, una imagen dura 3.000 tics. Cuando nada cambia en pantalla, no hay ninguna imagen nueva: `repeat_picture` hace que la última dure 3.000 tics más. Una pantalla donde solo se mueve un marco una vez por compás no cuesta casi nada.

Que el sonido empiece a tiempo

Un codificador MP3 pone 576 muestras de silencio antes del sonido, y un decodificador añade 529 más. Sin una corrección, la música llegaría 1.105 muestras tarde, 25 ms. Eso es casi una imagen, y basta para ver un acorde iluminarse antes de oírlo. La **lista de edición** (`elst`) de la pista de sonido le dice al reproductor dónde empieza el sonido de verdad:

```
// The edit list: play the track from `priming` on, for as long
// as the real sound lasts. The silence the encoder put first,
// and the padding after the last frame, are skipped.
write_box(out, b"edts", |out| {
    write_full_box(out, b"elst", 0, 0, |out| {
        out.extend_from_slice(&1_u32.to_be_bytes());
        out.extend_from_slice(&(duration_ms as u32).to_be_bytes());
        out.extend_from_slice(&format.priming.to_be_bytes());
        out.extend_from_slice(&1_u16.to_be_bytes()); // at normal speed
        out.extend_from_slice(&0_u16.to_be_bytes());
    });
});
```

Una edición: tocar durante lo que dura el sonido real, empezando 1.105 muestras dentro de la pista, a velocidad normal. Para comprobarlo, se hizo un video de prueba con clics en 1,0, 2,5 y 4,0 s y un destello blanco en las imágenes en los mismos momentos. Decodificado con FFmpeg, los clics están exactamente en 1,0, 2,5 y 4,0 s, y los destellos en las imágenes 30, 75 y 120.

Imágenes: H.264

Las imágenes llegan como RGBA, cuatro bytes por píxel. H.264 las quiere como **YUV**: brillo (Y) para cada píxel, y dos diferencias de color (U y V) para cada cuadrado de cuatro píxeles. El ojo ve el detalle en el brillo mucho mejor que en el color, así que solo con eso el tamaño se reduce a la mitad. Después de la primera imagen, el codificador solo guarda lo que cambió. Cada dos segundos hay una *imagen clave* que se sostiene sola, para que un reproductor pueda saltar allí y empezar a dibujar en el acto.

El codificador se configura para lo que muestra un video de la aplicación: una pantalla, con bordes nítidos, colores planos y texto:

```
let config = EncoderConfig::new()
    // A screen recording: sharp edges, flat colours, text.
    .usage_type(UsageType::ScreenContentRealTime)
    .profile(Profile::Baseline)
    .rate_control_mode(RateControlMode::Bitrate)
    .bitrate(BitRate::from_bps(bitrate(width, height)))
    .max_frame_rate(FrameRate::from_hz(frame_rate as f32))
    // Every picture is kept: dropping one to save room would
    // make the playhead stutter.
    .skip_frames(false)
    .complexity(Complexity::High)
    // A key picture every two seconds, so a player can jump.
    .intra_frame_period(IntraFramePeriod::from_num_frames(frame_rate * 2))

.vui(VuiConfig::bt709().matrix_coefficients(MatrixCoefficients::Smpte170M)
);
```

La tasa de bits es de 4,5 bits por píxel y por segundo: unos 4 Mbit/s a 1280×720 y 9 a 1920×1080, lo que mantiene el texto nítido y el archivo pequeño.

Lo que sale es una lista de *unidades NAL*: dos conjuntos de parámetros que describen el flujo (el tamaño de la imagen, el perfil), y después las imágenes. El codificador pone `0 0 1` o `0 0 0 1` antes de cada unidad para marcar dónde empieza. Un archivo MP4 quiere cada unidad después de su longitud, así que las marcas se quitan, con los patrones de slice con los que se leen los mensajes MIDI:

```
/// A NAL unit without the 0 0 1 or 0 0 0 1 the encoder puts before it to
/// mark where it starts.
fn without_start_code(nal: &[u8]) -> &[u8] {
    match nal {
        [0, 0, 0, 1, rest @ ..] | [0, 0, 1, rest @ ..] => rest,
        _ => nal,
    }
}
```

`rest @ ..` da un nombre al resto del slice, y `|` une dos patrones en un mismo brazo, ya que ambos ligan `rest`. El resultado toma prestado del argumento, así que no se copia nada.

El tipo de cada unidad está en los cinco bits bajos de su primer byte. Los conjuntos de parámetros se apartan, porque el archivo MP4 los guarda una sola vez, en su caja `avcC`; el resto son la imagen:

```
match header & 0x1F {
    SEQUENCE_PARAMETERS => self.sps = Some(nal.to_vec()),
    PICTURE_PARAMETERS => self.pps = Some(nal.to_vec()),
    kind => {
        picture.key |= kind == KEY_PICTURE;
        picture
            .bytes
            .extend_from_slice(&(nal.len() as u32).to_be_bytes());
        picture.bytes.extend_from_slice(nal);
    }
}
```

Un brazo de `match` puede compararse con constantes (`SEQUENCE_PARAMETERS` es `7`), y el último brazo, un simple nombre, atrapa cualquier otro valor y lo liga a `kind`.

Sonido: MP3

LAME recibe las muestras y devuelve *frames* MP3: cada uno contiene 1.152 muestras de cada canal y empieza con una cabecera de 4 bytes. El archivo MP4 guarda cada frame como una muestra de la pista de sonido, así que los frames se separan según llegan, leyendo la longitud de cada uno en su cabecera:

```rust
fn frame_length(bytes: &[u8]) -> Option<usize> {
    let [first, second, third, ..] = *bytes else {
        return None;
    };
    // Sync, MPEG-1 (11), Layer III (01).
    if first != 0xFF || second & 0xFE != 0xFA {
        return None;
    }
    const KBPS: [usize; 15] = [
        0, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320,
    ];
    const RATES: [usize; 3] = [44_100, 48_000, 32_000];
    let kbps = *KBPS.get(usize::from(third >> 4))?;
    let rate = *RATES.get(usize::from((third >> 2) & 0b11))?;
    let padding = usize::from((third >> 1) & 1);
    (kbps > 0).then(|| 144 * kbps * 1000 / rate + padding)
}
```

Los primeros once bits de la cabecera están todos a uno, para que un decodificador pueda encontrar un frame; luego vienen la versión y la capa, y en el tercer byte la tasa de bits y la frecuencia de muestreo, como posiciones en una tabla. A 256 kbit/s y 44,1 kHz un frame ocupa 144 × 256.000 / 44.100 = 835,9 bytes: 835, o 836 cuando el bit de relleno está activo, para que en promedio los frames salgan exactamente a la tasa de bits.

`let [first, second, third, ..] = *bytes else { return None; }` es un *let-else*: toma los tres primeros bytes si hay al menos tres, y si no, vuelve. `KBPS.get(...)?` devuelve `None` para un número más allá del final de la tabla, en lugar de entrar en pánico como haría `KBPS[...]`: los bytes podrían ser cualquier cosa.

El único `unsafe` del programa

LAME es C, y escribe su salida en la memoria que le damos. El crate quiere esa memoria como `&mut [MaybeUninit<u8>]`: bytes que todavía pueden contener cualquier cosa. Un `Vec` tiene espacio más allá de su final una vez que ha reservado algo, y `spare_capacity_mut` entrega ese espacio:

```
let frames = samples.len() / 2;
self.pending
    .reserve(mp3lame_encoder::max_required_buffer_size(frames));
let written = self
    .encoder
    .encode(InterleavedPcm(samples), spare(&mut self.pending))
    .map_err(|err| err.to_string())?;
// SAFETY: LAME wrote `written` bytes into the spare room.
unsafe { self.pending.set_len(self.pending.len() + written) };
Ok(self.cut())
```

Después hay que decirle al `Vec` que ahora es más largo. `set_len` es `unsafe`: Rust no puede comprobar que los bytes que añade al `Vec` se escribieron de verdad, y leer bytes que no se escribieron es comportamiento indefinido. Un bloque `unsafe` es donde el programador asume una promesa que el compilador no puede comprobar, y el comentario `// SAFETY:` dice por qué se cumple: LAME dijo que escribió `written` bytes allí. Los dos bloques de `mp3.rs` (este y el de `finish`) son el único código `unsafe` de todo el programa.

La otra forma sería un búfer de ceros que LAME sobrescribe. Es seguro y un poco más lento, ya que cada byte se escribe dos veces. Cualquiera de las dos sirve aquí; esta muestra para qué sirve `unsafe`.

Dibujar la aplicación sin ventana

Las vistas de un video son código egui normal. La hoja de acordes del video *es* la hoja de acordes de la aplicación, a través de pequeñas funciones `video` que llaman al mismo código de dibujo sin los menús. Pero egui no dibuja píxeles por sí mismo. En cada frame convierte la interfaz en una lista de **triángulos**, cada esquina con un color y un punto en una textura (la textura contiene las letras de la fuente). En la aplicación, `egui_glow` envía esos triángulos a la tarjeta gráfica.

La exportación no puede contar con una tarjeta gráfica, y corre en un hilo propio. Así que cada vista recibe un **panel**: un `egui::Context` propio, sin ventana, y un `Canvas` que pinta los triángulos con el procesador. En cada imagen del video, un panel ejecuta el código de la vista como eframe ejecuta el de la ventana, convierte lo que dibujó en triángulos y los pinta:

```
let ctx = self.ctx.clone();
let mut output = ctx.run_ui(input, &mut contents);
let shapes = std::mem::take(&mut output.shapes);
let primitives = ctx.tessellate(shapes, output.pixels_per_point);
changed |= self.canvas.paint(
    primitives,
    &mut output.textures_delta,
    output.pixels_per_point,
    background,
);
```

`textures_delta` es lo que egui añadió a sus texturas en ese frame: la imagen de la fuente crece a medida que hacen falta letras y tamaños nuevos, y el canvas guarda su propia copia, como haría una tarjeta gráfica.

Pintar un triángulo

Para pintar un triángulo, se visita cada píxel cercano y se pregunta «¿está este píxel dentro?». La prueba es la **función de arista**:

```
/// Twice the area of the triangle `from`, `to`, `p`: positive when `p`
/// is on the inside of the edge from `from` to `to`, zero on it.
fn edge(from: (i64, i64), to: (i64, i64), p: (i64, i64)) -> i64 {
    (to.0 - from.0) * (p.1 - from.1) - (to.1 - from.1) * (p.0 - from.0)
}
```

Para una arista de A a B, este producto vectorial es positivo de un lado y negativo del otro. Un píxel está dentro cuando las tres aristas coinciden. Los mismos tres números, divididos por su total, dicen lo cerca que está el píxel de cada esquina. Con eso se mezclan los colores y las coordenadas de textura de las esquinas, y así las letras toman su forma. Algunos detalles hacen que coincida con la pantalla:

- **Números enteros, en 1/256 de píxel.** Dos triángulos que comparten una arista no deben pintar ambos los píxeles que caen sobre ella, o una forma semitransparente recibe una costura más oscura. Con enteros no hay dudas de redondeo, y la *regla de arriba a la izquierda* resuelve los empates: un píxel justo sobre una arista pertenece al triángulo solo si esa arista es superior o izquierda.
- **Colores premultiplicados.** egui guarda los colores ya multiplicados por su opacidad, y mezcla con `nuevo + viejo × (1 − opacidad)`, como hace su shader de la tarjeta gráfica.
- **Avanzar, no multiplicar.** Moverse un píxel a la derecha cambia cada función de arista en una constante, así que el bucle interior suma en lugar de multiplicar. Para los triángulos simples de un solo color, la mayor parte de una interfaz, un camino rápido se salta la textura por completo.

Franjas de filas, en cada núcleo

La imagen se corta en franjas de filas, y cada franja se pinta en su propio núcleo:

```
self.pixels.fill(background.to_array());
let width = self.width;
let cores = thread::available_parallelism().map_or(1, |n| n.get().min(8));
let rows_per_band = self.height.div_ceil(cores).max(1);
let textures = &self.textures;
thread::scope(|scope| {
    for (band, pixels) in self.pixels.chunks_mut(rows_per_band *
width).enumerate() {
        let top = band * rows_per_band;
        let bottom = top + pixels.len() / width;
        scope.spawn(move || {
            let mut band = Band {
                pixels,
                width,
                top,
                bottom,
            };
            for clipped in primitives {
                if let Primitive::Mesh(mesh) = &clipped.primitive {
                    band.paint_mesh(mesh, clipped.clip_rect,
pixels_per_point, textures);
                }
            }
        });
    }
});
```

Estos hilos *toman prestado*: cada uno recibe un slice `&mut` de la imagen, y todos comparten `&self.textures` y los triángulos. `thread::spawn` no podría hacerlo, ya que su closure debe ser `'static`. `std::thread::scope` sí puede, porque no vuelve hasta que cada hilo creado dentro de él ha terminado, así que los préstamos no pueden sobrevivir a lo que apuntan. El compilador comprueba el resto: `chunks_mut` da slices que no se solapan, así que dos hilos nunca pueden escribir el mismo píxel.

Cada franja pinta todos los triángulos, pero solo sus propias filas de cada uno. Un triángulo que cruza dos franjas lo pintan a medias entre las dos.

Un cuadro para cada vista, y el zoom

El zoom funciona como el propio zoom de egui. El panel le dice a su contexto que hay más píxeles por punto, 1,4 veces más al 140 %, así que egui distribuye la vista en menos puntos y dibuja cada letra al tamaño mayor. Si la vista se dibujara pequeña y luego se estirara la imagen, las letras saldrían borrosas:

```
let ppp = self.pixels_per_point;
let size = vec2(self.canvas.width() as f32, self.canvas.height() as f32) /
ppp;
let mut input = egui::RawInput {
    screen_rect: Some(Rect::from_min_size(Pos2::ZERO, size)),
    time: Some(time),
    predicted_dt: 1.0 / frame_rate as f32,
    ..Default::default()
};
input
    .viewports
    .entry(egui::ViewportId::ROOT)
    .or_default()
    .native_pixels_per_point = Some(ppp);
```

`RawInput` es lo que eframe le da a egui en cada frame: el tamaño de la pantalla, el tiempo, el ratón y las teclas. Aquí es inventado, sin ratón ni teclas, y con el tiempo *del video*, así que lo que se mueve con el tiempo se mueve con el video. `..Default::default()` rellena cada campo que no se nombra.

Full HD distribuye las vistas exactamente como HD, 1280 por 720 puntos, a 1,5 píxeles por punto: el texto es más nítido, pero no cabe nada más. Cada panel pinta su propio canvas, y `Canvas::copy_from` copia los paneles en la imagen, en sus cuadros, bajo una línea de título con el nombre de la canción, el compás y el pulso, la tonalidad, el compás de la partitura y el tempo. Los cuadros pueden solaparse: la vista más abajo en la lista se copia la última, encima.

A tiempo

Dos detalles mantienen las vistas a tiempo:

- **Sin deslizamiento.** En la ventana, una vista se desplaza suavemente hasta el compás que suena, en aproximadamente medio segundo. En un video eso es medio segundo tarde, así que cada panel desactiva la animación de desplazamiento de egui y salta allí: `style.scroll_animation = egui::style::ScrollAnimation::none()`.
- **Ni una imagen tarde.** Una vista que se desplaza solo se mueve en el frame *siguiente*, y le pide uno a egui de inmediato (un retraso de redibujado de cero). Cuando eso pasa, el panel hace una segunda pasada dentro de la misma imagen, así que el compás aparece en su primer pulso. Un panel cuyo tamaño o zoom acaba de cambiar también se dibuja siempre dos veces, porque egui mide algunas cosas en su primer frame.

Imágenes que no cambiaron

Mientras suena una canción, la mayoría de las imágenes son iguales a la anterior: la hoja de acordes solo se mueve cuando cambia el compás. `Canvas::paint` guarda los triángulos de la última imagen, y pinta solo si los nuevos son distintos:

```
/// Whether two lists of shapes paint the same picture.
fn same(new: &[ClippedPrimitive], old: &[ClippedPrimitive]) -> bool {
    new.len() == old.len()
        && new.iter().zip(old).all(|(new, old)| {
            new.clip_rect == old.clip_rect
                && match (&new.primitive, &old.primitive) {
                    (Primitive::Mesh(new), Primitive::Mesh(old)) => new ==
old,
                    _ => false,
                }
        })
}
```

Comparar dos mallas con `==` compara cada esquina de cada triángulo, lo que es mucho más barato que pintarlas. Cuando ningún panel cambió y ningún cuadro se movió, `Scene::draw` devuelve `None`, el codificador no se ejecuta y `repeat_picture` hace que la última imagen dure más. Saltarse las imágenes sin cambios y pintar los triángulos simples por el camino rápido hacen que exportar un video sea varias veces más rápido que pintar cada imagen entera.

Ajustes que viven en un hilo

Los nombres de las notas, las traducciones y los números romanos en menor son ajustes por hilo, valores `thread_local!` que la ventana fija antes de dibujar cada frame. La exportación dibuja en un hilo propio, que empieza con los valores por defecto: el video de un usuario en español saldría en inglés y en letras. Así que la exportación lee los tres ajustes en el hilo de la interfaz, los mueve al closure (son pequeños valores `Copy`) y los fija lo primero en su propio hilo:

```
let numerals = MinorNumerals::current();
let language = crate::i18n::Language::current();
let naming = lineup_core::NoteNaming::current();
```

```
thread::spawn(move || {
    // The views are made here, on the export's own thread,
    // and write numerals, words and note names as the app does.
    numerals.set();
    language.set();
    naming.set();
    // ...then the scene, the writer and the pictures, below.
})
```

Mantener la imagen con el sonido

La canción la toca el mismo `Renderer` fuera de línea que una exportación WAV, 1/30 s cada vez: a 44.100 muestras por segundo, exactamente 1.470 muestras por imagen. Las imágenes tienen que mostrar *dónde está la música*, y con cambios de tempo, ritardandos y calderones eso no es una suma simple. Así que el renderer lo informa. Después de cada bloque de sonido, el hilo de audio guarda su posición en pasos, con una fracción (4,5 es la mitad del paso 4), como hace para la ventana, y `Renderer::position` la devuelve:

```
for picture in 0..pictures {
    if cancel.load(Ordering::Relaxed) {
        return Err(CANCELLED.to_string());
    }
    let time = picture as f64 / f64::from(FRAME_RATE);
    // Before the first sound, the song is about to
    // start; after it, where the sound so far ended.
    let position = if picture == 0 {
        Some(0.0)
    } else {
        renderer.position()
    };
    match scene.draw(time, position) {
        Some(picture) => writer.add_picture(picture)?,
        None => writer.repeat_picture(),
    }
    renderer.render(&mut sound);
    writer.add_sound(&sound)?;
    done.store(picture + 1, Ordering::Relaxed);
}
```

Cada imagen se dibuja *antes* del sonido que suena mientras se muestra, en la posición donde empieza ese sonido.

La posición cuenta los pasos *tal como suenan*: con una repetición, o un estribillo que la banda toca tres veces, ese no es el lugar en la lista de reproducción. `Scene::draw` la traduce con el arreglo de la canción, como hace la ventana con su cursor:

```
let at = position.and_then(|position| {
    let step = position.max(0.0) as usize;
    let pass = *self.song.pass_at(step)?;
    Some((pass.from + (step - pass.start), pass))
});
```

Una *pasada* (pass) es un tramo de la lista de reproducción tocado una vez: desde qué lugar de la lista (`from`), empezando en qué paso tocado (`start`). Cómo el arreglo despliega las repeticiones en pasadas está en Forma de la canción.

Para comprobarlo, se exportó El Choclo. A 112 BPM en 4/4, el compás 15 empieza a los 30,0 s, el compás 29 a los 60,0 s y el compás 43 a los 90,0 s, y la traza de reproducción, que toca a través del mismo renderer, tiene las primeras notas de esos compases exactamente en esos tiempos. Las imágenes del video en esos momentos muestran «Compás 15, pulso 1», «Compás 29, pulso 1» y «Compás 43, pulso 1».

Organizar las vistas

Dónde va cada cuadro y a qué tamaño se dibuja su vista es un `Layout`: un `Placement` para cada una de las cinco vistas.

```
/// Where a view goes in the picture, and how big it is drawn.
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
pub struct Placement {
    pub shown: bool,
    /// Its box: left, top, right and bottom, from 0 to 1 across the room
    /// under the title line.
    pub rect: [f32; 4],
    /// How big it is drawn: 1 as in the window, 2 twice as big.
    pub zoom: f32,
}
```

El cuadro se guarda en partes del espacio, de 0 a 1, no en píxeles, así que la misma distribución sirve para los dos tamaños, y para la vista previa del diálogo, que mide 640 puntos de ancho.

Las ubicaciones viven en un array, pero el código las lee por vista: `layout[View::ChordSheet]`. Eso funciona porque `Layout` implementa los traits de indexación, con un `View` como índice:

```
impl std::ops::Index<View> for Layout {
    type Output = Placement;
    fn index(&self, view: View) -> &Placement {
        &self.views[view.index()]
    }
}

impl std::ops::IndexMut<View> for Layout {
    fn index_mut(&mut self, view: View) -> &mut Placement {
        &mut self.views[view.index()]
    }
}
```

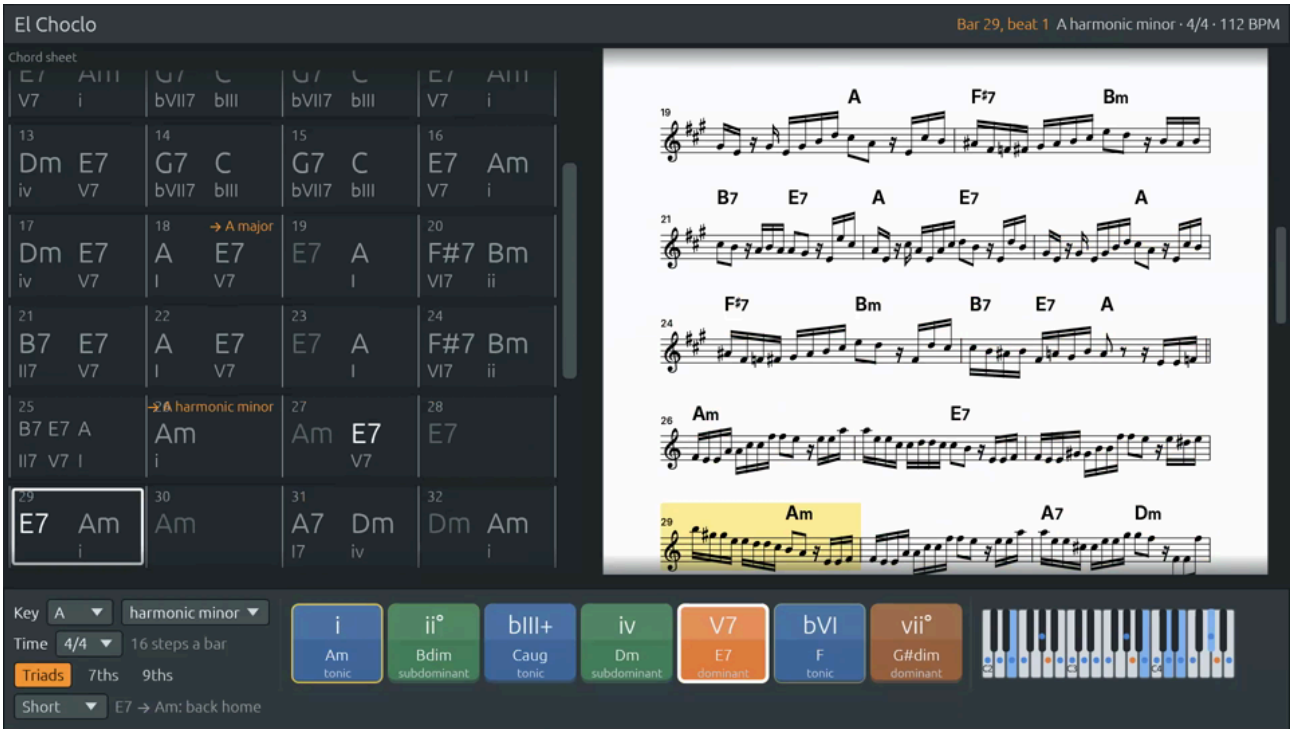
`Index` es lo que llama `layout[view]` para leer, e `IndexMut` lo que llama para cambiar, como en `layout[View::HarmonyKeys].shown = true`. `View` es un enum sin datos, así que `view as usize` da su posición en la declaración, que es lo que devuelve `View::index`. Nadie puede indexar el array con un número que no sea una vista.

Arrastrar un cuadro cambia su `rect` con `move_by`, su esquina con `resize_by`, y la rueda del ratón su zoom con `zoom_by`, manteniendo siempre el cuadro dentro del espacio y el zoom entre el 50 % y el 250 %. La rueda amplía por `(scroll / 500.0).exp()`: girarla hacia arriba y luego hacia abajo la misma cantidad multiplica por un número y después por su inverso, así que vuelve exactamente a donde estaba. **Apilar** llama a `Layout::stack`, que pone las teclas de armonía abajo, tan altas como necesitan, y reparte el espacio de arriba por igual entre las demás vistas mostradas.

El diálogo recuerda la distribución y el tamaño entre ejecuciones, con `get_persisted` e `insert_persisted` de egui. `Layout` y `Placement` derivan `Serialize` y `Deserialize`, y `#[serde(default)]` rellena cualquier campo que falte en lo guardado con su valor por defecto, así que los ajustes escritos por otra versión de la aplicación se siguen leyendo.



Aquí la hoja de acordes, ampliada al 140 %, está al lado de la partitura guía, con las teclas de armonía al 120 % debajo:



Cómo se prueba

Los tests del renderer tocan en un `Vec` y miran los números:

- `renders_like_the_audio_thread`: un clic hecho por el renderer es exactamente el clic que hace el hilo de audio, `tanh(0.5)` en los dos canales, y dura diez frames.
- `knows_where_playback_is`: a 120 BPM un paso dura 6.000 frames a 48 kHz, así que después de 9.000 frames la posición es 1,5 pasos.
- `exports_a_wav_file`: el archivo tiene el formato, la duración y la primera muestra correctos, `progress` se llamó después de cada bloque, y no queda ningún archivo `.tmp`.
- `a_cancelled_export_leaves_no_file`: después de detenerla no existen ni el archivo ni el temporal.

Los tests del escritor MP4 comprueban los bytes: `a_box_starts_with_its_size_and_name`, `samples_of_the_same_length_are_listed_as_runs`, y `writes_a_movie_the_tables_describe`, que vuelve a leer dónde quedó cada muestra y cuánto dura la película. Los tests de la escena dibujan imágenes y cuentan píxeles: `views_are_drawn_in_their_boxes` pone la hoja de acordes en la mitad derecha y comprueba que la mitad izquierda está vacía, y `zoomed_in_views_are_drawn_bigger` comprueba que una vista al 200 % tiene mucha más tinta que al 100 %. Dos tests de la

aplicación exportan un video entero de la canción inicial con todas las vistas, y detienen otro a mitad de camino para comprobar que no queda ningún archivo.

Lo que los tests no pueden comprobar es si un archivo se reproduce bien en todas partes. Para eso, abre una exportación en cualquier editor de audio: la forma de la canción se ve fácilmente en la onda. El archivo WAV de la canción inicial tiene exactamente 739.523 frames, el canal izquierdo y el derecho son iguales (todos sus canales están en el centro y la reverberación está apagada), los compases 1 y 2 tienen los acordes solos, los compases 3 a 7 el ritmo (el patrón Pulso), el compás 8 el relleno, y la cola está en silencio, sin ningún acorde de la canción que vuelva a empezar. Un video debería abrirse en cualquier reproductor, y saltar de un lado a otro debería empezar a dibujar en el acto, gracias a una imagen clave cada dos segundos.

Los ejercicios sobre exportar están en Ejercicios: el sonido.

La traza de reproducción

Archivos: `crates/lineup-audio/src/trace.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-app/src/app/trace.rs` y `crates/lineup-app/src/main.rs`

La pantalla muestra el proyecto; no muestra lo que el reproductor hizo con él. Una canción pasada a otro compás puede verse bien, cada compás redibujado en 3/4, y aun así tocar sus compases al ritmo de antes, porque las notas se apretaron y el tempo se hizo más lento, y lo uno compensa lo otro. Solo el sonido dice la verdad. Pero escuchar es lento, y dice que algo está mal, no qué.

Una **traza** es una lista de cada nota que toca el reproductor, anotada por el hilo de audio en el momento en que la toca. No se calcula de nuevo a partir del proyecto: es lo que les llega a los altavoces, después del swing, los cambios de tempo, los desplazamientos y la soltura, con la canción desplegada como la toca la lista de reproducción y sin las pistas silenciadas. Así que una traza puede leerse en lugar de escucharse, por ti o por un test. Muchos de los tests del programa leen trazas, y también las lee quien comprueba que una demostración toca lo que dice su partitura.

Probarla

Desde una terminal, con un archivo de canción o el nombre de una demostración:

```
ferrous-lineup --trace song.lineup --out song.tsv
ferrous-lineup --trace "Twelve-bar blues"
ferrous-lineup --demos
```

`--trace` toca la canción como lo hace **Archivo > Exportar WAV…**, de principio a fin, pero sin ventana, sin tarjeta de sonido y sin archivo de sonido: obtienes las notas de la exportación, frame a frame, en uno o unos pocos segundos. `--out` las escribe en un archivo; sin él van a la terminal. `--rate` toca a otra frecuencia de muestreo que los 44.100 de la exportación. `--demos` lista los nombres que entiende `--trace`, que son los nombres en inglés de las demostraciones («Twelve-bar blues» para la que la aplicación en español llama «Blues de doce compases»). Las mayúsculas no importan, y basta con parte de un nombre cuando solo una demostración lo tiene.

En Windows, usa `--out`: el programa ya compilado es un programa de ventana y no tiene terminal en la que escribir.

En la aplicación, **Edición > Ajustes… > Reproducción > Registro** tiene una casilla, **Escribir un registro mientras suena**. Mientras está marcada, cada reproducción, de Reproducir a Detener, añade una tabla a `playback-trace.tsv`, en la carpeta donde la aplicación guarda sus ajustes; el ajuste muestra la ruta completa. Empieza sin marcar cada vez que se abre la aplicación, para que una traza olvidada nunca llene el disco. La casilla está desactivada cuando no hay ninguna tarjeta de sonido abierta, ya que entonces no suena nada. El manual sugiere adjuntar este archivo al informar de un problema con la reproducción.

Leer la tabla

Las líneas que empiezan con `#` dicen qué contiene la tabla, y después enumeran el tempo y el compás de cada parte de la canción. Estas son las primeras de Für Elise («Para Elisa» en la aplicación en español):

```
# Ferrous Lineup playback trace: Für Elise
# Every note the player started, in the order they start. Times are in
seconds from the start of playback;
# steps are sixteenth notes of the loop played (the song, or the pattern
in pattern mode), counted from 0.
# sample rate: 44100 Hz
# tempo: 0.0000 s, step 0, bar 1: 144.00 BPM
# tempo: 123.7500 s, step 1188, bar 100: 216.00 BPM
# tempo: 130.0000 s, step 1278, bar 105: 144.00 BPM
# tempo: 156.2500 s, step 1530, bar 126: 142.80 BPM
# tempo: 156.3551 s, step 1531, bar 126: 141.60 BPM
...
# time: step 0, bar 1: 3/4
# time: step 1188, bar 100: 9/8
# time: step 1278, bar 105: 3/4
```

El tempo cambia como está escrito en los compases 100 y 105, y después baja deslizándose desde el compás 126: un ritardando, que el reproductor toca paso a paso, cada uno un poco más lento, así que la traza tiene una línea por cada paso.

Después viene una línea por nota, en el orden en que empiezan, con un tabulador entre columnas. Estas son las primeras notas del bajo y del charles en la demostración **Blues de doce compases**, sin algunas columnas:

```
start   length  bar beat into_beat step   steps bpm    name pitch note
velocity late_ms end
0.0000  0.4000  1   1    0.000     0.000  2.667 100.00 Bass 36    C2   104
0.0     gate
0.0000  0.1500  1   1    0.000     0.000  1.000 100.00 Hat  60    C4   96
0.0     gate
0.4000  0.0500  1   1    2.667     2.667  0.333 100.00 Hat  60    C4   96
100.0   gate
0.4000  0.2000  1   1    2.667     2.667  1.333 100.00 Bass 40    E2   90
100.0   gate
0.6000  0.4000  1   2    0.000     4.000  2.667 100.00 Bass 43    G2   96
0.0     gate
0.6000  0.1500  1   2    0.000     4.000  1.000 100.00 Hat  60    C4   96
0.0     gate
```

El swing está ahí mismo. A 100 BPM un pulso dura 0,6 segundos y tiene cuatro pasos. Escrita, la segunda corchea del pulso está en el paso 2, a la mitad. Tocada, llega 2,667 pasos dentro del pulso, 100 ms tarde: a dos tercios del camino, la tercera nota de un tresillo, que es lo que significa el swing completo. Y la nota del bajo anterior dura 2,667 pasos en lugar de 2, larga antes que corta. (Cómo el swing mueve las notas está en Swing.)

Columna	Qué dice
`start`, `length`	Cuándo empieza la nota y cuánto se sostiene, en segundos desde el inicio de la reproducción
`bar`, `beat`	Su compás y su pulso, contados desde 1, en el compás vigente
`into_beat`	Cuántos pasos después del inicio de su pulso empieza
`step`, `steps`	El paso en que empieza (contado desde 0, con la fracción con la que llega tarde) y cuántos pasos se sostiene
`bpm`, `time`	El tempo de su paso y el compás de su compás
`channel`, `name`, `program`, `instrument`	Su fila en el rack de canales (desde 1), el nombre del canal, su programa General MIDI y qué lo toca
`pitch`, `note`	El número de nota MIDI y su nombre, escrito en la tonalidad del momento
`velocity`	Con qué fuerza se toca, de 1 a 127
`late_ms`	Cuánto después de su paso empieza: su desplazamiento, el swing y la soltura
`end`	Por qué terminó: `gate` cuando se acabó su duración, `cut` cuando se cortó antes (Detener, un salto, su pista silenciada, su clip quitado), `again` cuando la misma nota

Columna	Qué dice
	volvió a empezar en una Salida MIDI o una SoundFont, que tocan una de cada nota a la vez
`pattern`	Su patrón y el paso del patrón del que viene

Tablas así están hechas para `grep` y `awk`. Las notas del bajo del compás 12, o cada nota más fuerte que 110:

```
ferrous-lineup --trace song.lineup | awk -F'\t' '$11 == "Bass" && $3 ==
12'
ferrous-lineup --trace song.lineup | awk -F'\t' '!/^#/ && $16+0 > 110'
```

Una hoja de cálculo también abre el archivo: dile que las columnas están separadas por tabuladores y que las líneas que empiezan con `#` son comentarios. La tabla está siempre en inglés y nombra las notas con letras, muestre lo que muestre la ventana, así que una traza se lee igual para todos.

Lo que una traza no dice

Una traza sigue las notas, no el sonido. Una muestra de batería suena hasta su final sea cual sea la duración de la nota, el release de un sintetizador sigue apagándose después de que la nota termina, y cuando todas las voces del Lineup Osc están ocupadas, la más vieja se toma para la nota nueva. Nada de eso está en la traza. Para un problema dentro de un instrumento, escuchar (o mirar las muestras) sigue siendo el camino.

Escribir en el hilo de audio

El hilo de audio tiene reglas estrictas: nunca debe esperar, nunca reservar memoria, nunca escribir en un archivo. Escribir una traza es justo lo que no puede hacer. Así que hace lo que ya hace con la Salida MIDI: entrega mensajes pequeños a un ring buffer, y otro hilo, que puede tomarse su tiempo, los lee.

Los mensajes son un enum `Copy`, así que meter uno copia unos pocos bytes y no reserva nada:

```rust
pub enum TraceEvent {
    /// Playback started on `step`: of the song, or of pattern `pattern`
    /// when `song` is false. After a count-in, when the count is over.
    Play {
        frame: u64,
        step: usize,
        song: bool,
        pattern: usize,
    },
    /// Playback jumped to `step` while playing.
    Jump { frame: u64, step: usize },
    /// Playback stopped.
    Stop { frame: u64 },
    /// A step started, at `tempo`.
    Step {
        frame: u64,
        step: usize,
        tempo: Tempo,
    },
    /// A note started on channel `channel` (its row in the channel rack),
    /// `late` frames after its step (nudge and loose timing), to be held
    /// for `gate` frames. `id` pairs it with its [`TraceEvent::NoteOff`].
    NoteOn {
        frame: u64,
        id: u32,
        channel: usize,
        origin: Origin,
        pitch: Note,
        velocity: u8,
        gate: u32,
        late: u32,
    },
    /// The note `id` ended, as `end` says.
    NoteOff { frame: u64, id: u32, end: End },
}
```

`frame` es el reloj propio del hilo de audio: los frames que ha tocado desde que arrancó. Cada nota recibe un `id`, para poder emparejar después su `NoteOff` con su `NoteOn`. Los ids cuentan hacia arriba con `wrapping_add(1)`, que vuelve a 0 después de `u32::MAX` en lugar de entrar en pánico en una compilación de depuración; cuatro mil millones de notas después, las primeras ya terminaron hace mucho.

El hilo de audio guarda el trazador como un `Option<Box<Tracer>>`. Cuando la traza está apagada es `None`, y cada lugar que anota es un `if let Some(trace) = &mut self.trace`, que no cuesta casi nada. El trazador se crea en otro hilo, donde se puede reservar su memoria:

```
/// A tracer for the audio thread and the reader of what it writes.
pub fn trace_queue() -> (Tracer, TraceReader) {
    let (events, consumer) = rtrb::RingBuffer::new(TRACE_QUEUE_SIZE);
    let lost = Arc::new(AtomicUsize::new(0));
    let tracer = Tracer {
        events,
        lost: Arc::clone(&lost),
        open: Vec::with_capacity(OPEN_NOTES),
        next_id: 0,
        next_end: u64::MAX,
    };
    let reader = TraceReader {
        events: consumer,
        lost,
    };
    (tracer, reader)
}
```

y viaja al hilo de audio dentro de un comando, igual que un instrumento: `Command::SetTrace(Some(Box::new(tracer)))`. La cola guarda 65.536 eventos (`1 << 16`), y la lista de notas que suenan se reserva para 4.096, así que nada crece en el hilo de audio. Un trazador viejo vuelve al hilo de la interfaz como basura, para liberarse allí.

Perdidos a propósito

Si el ring buffer está lleno, porque nadie lo leyó durante mucho tiempo, el evento no se mete y sube un contador de eventos perdidos:

```
fn push(&mut self, event: TraceEvent) {
    if self.events.push(event).is_err() {
        self.lost.fetch_add(1, Ordering::Relaxed);
    }
}
```

La cabecera de la tabla dice entonces cuántos eventos se perdieron, en lugar de que falten notas en silencio. Las colas que llevan la *música* hacen lo contrario: un note-off MIDI que encuentra su cola llena espera a que haya sitio, porque uno perdido dejaría una nota sonando para siempre en un sintetizador (ver Cuando la tarjeta de sonido desaparece). Una traza es un informe, no la música. Hacer esperar al hilo de audio por ella pondría el informe antes que el sonido.

Dónde empiezan las notas

Una nota de la canción empieza en uno de dos lugares: en `start_pattern_notes`, cuando llega su paso, o en `start_pending`, unos milisegundos después, para una nota desplazada o con soltura. Ambos se lo dicen al trazador en el frame en que la nota empieza de verdad, por eso una nota con swing muestra sus 100 ms:

```
if let Some(trace) = &mut self.trace {
    trace.note_on(
        self.now,
        note.channel,
        note.origin,
        note.pitch,
        note.velocity,
        note.gate,
        note.late,
        one_per_pitch(&channel.instrument),
    );
}
```

`one_per_pitch` dice si el instrumento es una Salida MIDI o una SoundFont, que tocan una de cada nota a la vez: en esos, una nota que vuelve a empezar termina la que suena, y la traza escribe `again` para ella.

Cuánto dura una nota

Cuando una nota empieza, su duración se convierte en un número de frames, el *gate*. Si el tempo cambia mientras suena, cada instrumento estira los gates de sus notas con la misma función, `stretched`, para que la nota siga terminando en su paso (ver Cada paso a su propio tempo). El trazador lleva su propia lista de las notas que suenan, `open`, y las estira con esa misma función:

```
pub(crate) fn tempo_changed(&mut self, now: u64, next_step: f64, stretch:
f64) {
    for note in &mut self.open {
        if note.end != u64::MAX {
            let left = note.end.saturating_sub(now) as f64;
            note.end = now + stretched(left, next_step, stretch) as u64;
        }
    }
    self.find_next_end();
}
```

Una nota con el gate que nunca se acaba, `HELD` (`u32::MAX`), no tiene un final que el trazador pueda saber, y lo guarda como `u64::MAX`.

El trazador también se entera de todo lo que termina notas antes de tiempo: Detener, un salto, un silenciado, una canción enviada mientras suena que ya no tiene un clip, un canal quitado. El caso de la canción cambiada muestra un pequeño detalle de Rust. El hilo de audio comprueba cada nota que suena con un closure, `still_played`, que toma prestada la canción. Lo necesitan tanto los instrumentos como el trazador:

```
for channel in &mut self.channels {
    channel.instrument.release_unless(still_played);
}
if let Some(trace) = &mut self.trace {
    trace.cut(self.now, None, still_played);
}
```

`still_played` se pasa por valor, una vez por cada canal y una vez más. Eso funciona porque un closure que solo captura referencias compartidas es `Copy`, como las propias referencias: pasarlo entrega una copia y el original sigue usable.

Solo cuando una nota termina

Para encontrar las notas cuya duración se acabó, el trazador tendría que mirar cada una en cada frame, 44.100 veces por segundo. En cambio, recuerda el final más temprano, `next_end`, y no hace nada hasta que llega ese frame:

```
/// Ends the notes whose length is over by frame `now`. Called once
/// a frame, before anything else happens on it.
pub(crate) fn frame(&mut self, now: u64) {
    if now < self.next_end {
        return;
    }
    // The fields are borrowed one by one: `open` to go through, the
    // queue and the counter to write to.
    let events = &mut self.events;
    let lost = &self.lost;
    self.open.retain(|note| {
        if note.end > now {
            return true;
        }
        let event = TraceEvent::NoteOff {
            frame: note.end,
            id: note.id,
            end: End::Gate,
        };
        if events.push(event).is_err() {
            lost.fetch_add(1, Ordering::Relaxed);
        }
        false
    });
    self.find_next_end();
}
```

`retain` conserva los elementos para los que el closure devuelve `true` y suelta los demás, en una sola pasada y en el mismo sitio. El closure necesita meter eventos en la cola mientras `retain` tiene `self.open` prestado como mutable. Llamar a `self.push` dentro de él tomaría prestado todo `self` por segunda vez, y el compilador lo rechaza. Tomar prestados antes los otros dos campos en variables locales (`events`, `lost`) da tres préstamos de tres campos distintos, que no se solapan.

Convertir los eventos en notas

`Trace::notes` lee los eventos después, en un hilo normal. Los divide en reproducciones (*runs*) en cada `Play`, y guarda los eventos `Step` de cada una en una lista, en el orden en que se tocaron. Para saber dónde está un frame, busca el último paso que empezó antes, con `partition_point`:

```
fn position(&self, frame: u64, sample_rate: u32) -> Option<(f64, f64,
Tempo)> {
    let index = self.steps.partition_point(|played| played.frame <=
frame);
    let played = self.steps.get(index.checked_sub(1)?)?;
    let length = match self.steps.get(index) {
        Some(next) => (next.frame - played.frame) as f64,
        None => frames_per_step(played.tempo, sample_rate),
    };
    let fraction = (frame - played.frame) as f64 / length.max(1.0);
    Some((
        played.step as f64 + fraction,
        played.count as f64 + fraction,
        played.tempo,
    ))
}
```

`partition_point` es una búsqueda binaria: dada una lista ordenada y una prueba que es verdadera para una primera parte y falsa para el resto, devuelve dónde empieza la parte falsa. Aquí es el primer paso que empezó después de `frame`; el anterior es el paso que se está tocando. La fracción es cuánto ha avanzado `frame` entre ese paso y el siguiente. `index.checked_sub(1)?` devuelve `None` para un frame anterior al primer paso, donde `index - 1` daría la vuelta por debajo de cero.

Cada paso tiene además un `count` de los pasos tocados antes que él, que sigue creciendo cuando un patrón vuelve a empezar o el cursor salta. La duración de una nota en pasos es la diferencia de esos contadores: se mide a lo largo de los pasos tocados de verdad, sean cuales sean los números que les da el bucle. Una nota sostenida a través del final de un bucle de un compás, del paso 14 al paso 2 de la vuelta siguiente, dura 4 pasos, no -12.

Un `NoteOff` encuentra su `NoteOn` por id en una pequeña lista de las notas aún abiertas en la reproducción. `swap_remove` lo saca moviendo el último elemento a su lugar, lo que es rápido y aquí está bien, ya que el orden de esa lista no importa.

Escribir la tabla

`write_table` escribe la cabecera y una línea por nota. Los nombres vienen de `Labels`: el título de la canción, el proyecto (para los nombres de los canales y patrones), los compases y las tonalidades, todo en los pasos del bucle que se tocó:

```
/// For a trace of song mode playing `arrangement`.
pub fn song(title: &'a str, project: &'a Project, arrangement:
&Arrangement) -> Self {
    Labels {
        title,
        project,
        meter: project.played_meter(arrangement),
        keys: project.played_keys(arrangement),
    }
}
```

`played_meter` y `played_keys` despliegan los compases y las tonalidades igual que el arreglo despliega los clips, así que una sección en 3/4 que suena dos veces está en 3/4 las dos veces. Cada nota se escribe en la tonalidad de su paso: un Fa♯ en Sol mayor, un Sol♭ en Re♭ mayor (en la tabla, siempre con letras: F♯, G♭).

El nombre de un canal o de un patrón podría contener un tabulador o un salto de línea, que partirían la tabla. `cell` los cambia por espacios:

```
/// Text for a table cell: no tabs or line breaks, which would split it.
fn cell(text: &str) -> String {
    text.replace(['\t', '\n', '\r'], " ")
}
```

`str::replace` recibe un *patrón*, y un array de caracteres lo es: coincide con cualquiera de ellos.

Una nota que seguía sonando cuando terminó la traza no tiene duración; la tabla escribe `?` para ella, y la cabecera dice cuántas hay.

Una traza mientras tocas

La traza del ajuste es un `LiveTrace`, que guarda la aplicación. Una vez por frame, `check_trace` mueve a él lo que entregó el hilo de audio. Cada reproducción se escribe como una tabla propia cuando termina, al Detener o cuando empieza otra (cambiar entre el modo canción y el modo patrón mientras suena empieza otra). `Trace::take_finished` saca las reproducciones terminadas y deja la que sigue en marcha.

Una tabla necesita los nombres del proyecto, pero puedes renombrar un canal mientras suena una canción. Así que en cada `Play`, `check_trace` guarda una copia del proyecto tal como estaba, y la tabla se escribe con los nombres que tenía esa reproducción:

```
for event in &live.trace.events[before..] {
    if let TraceEvent::Play { song, pattern, .. } = *event {
        live.playing.push_back(Playing {
            title: self
                .file
                .song_name()
                .unwrap_or_else(|| "Untitled".to_string()),
            project: self.project.clone(),
            song,
            pattern,
        });
    }
}
```

Un `VecDeque` es una cola que puede añadir por un extremo y sacar por el otro: las copias entran por detrás a medida que empiezan las reproducciones, y salen por delante a medida que se escriben sus tablas.

La ventana muestra los nombres de las notas en letras o en solfeo y las palabras en inglés o en español, pero la tabla no debe cambiar con ellos. Así que `write` fija ambos mientras escribe, con las funciones `with` de los ajustes por hilo:

```
let mut table = Vec::new();
lineup_core::i18n::Language::English.with(|| {
    lineup_core::NoteNaming::Letters.with(|| write_table(&mut table,
trace, &labels))
})?;
```

Desmarcar la casilla, o cerrar la aplicación, suelta el `LiveTrace`. Su `Drop` escribe la reproducción que sigue en marcha, para que no se pierdan las últimas notas. Si la tarjeta de sonido desaparece y vuelve, `resume_trace` escribe lo que dejó el hilo de audio viejo y le da un trazador nuevo al nuevo, y la traza sigue en el mismo archivo.

La línea de comandos

`main` mira su primer argumento antes de abrir una ventana (ver `main.rs`). Después de `--trace`, un pequeño bucle lee el resto:

```
while let Some(arg) = args.next() {
    match arg.to_str() {
        Some("--out" | "-o") => out = args.next().map(PathBuf::from),
        Some("--rate") => match args.next().and_then(|hz|
hz.to_str()?.parse().ok()) {
            Some(hz) if (8_000..=192_000).contains(&hz) => rate = hz,
            _ => {
                eprintln!("--rate takes a sample rate in Hz, such as
44100");
                return 2;
            }
        },
        _ if source.is_none() => source = Some(arg),
        _ => {
            eprintln!("{USAGE}");
            return 2;
        }
    }
}
```

`args` es un iterador, y el bucle le saca elementos de dos maneras: uno por vuelta con `while let`, y uno más con `args.next()` dentro del cuerpo, para el valor de una opción. Los brazos muestran tres tipos de patrón: `"--out" | "-o"` coincide con cualquiera de los dos textos, `Some(hz) if ...` añade una guarda, y `_ if source.is_none()` coincide con cualquier cosa, pero solo mientras no se haya dado una canción. La función devuelve el código de salida con el que termina el programa: 0 cuando se escribió la traza, 1 cuando no se pudo, 2 por un error en los argumentos. Los scripts lo comprueban.

`trace_table` construye la aplicación entera sin ventana ni tarjeta de sonido, como hacen los tests (`Audio::Unavailable`), abre el archivo o la demostración, y pide el mismo renderizado de la canción que hace la exportación WAV, con una traza activada:

```
pub(super) fn trace_song(&mut self, sample_rate: u32) -> Trace {
    let mut render = self.song_render_traced(sample_rate);
    render.renderer.skip(render.frames);
    render.renderer.take_trace().unwrap_or_default()
}
```

`skip` toca todos los frames y tira el sonido. La traza es la de la exportación, porque es el mismo camino de código (ver Exportar).

El nombre de una demostración se busca en tres intentos: exacto, después sin mirar las mayúsculas, después como parte de un nombre. El último intento puede encontrar ninguna, una o varias, y un patrón de slice dice cuál es el caso:

```
match matching.as_slice() {
    [demo] => Ok(*demo),
    [] => Err(format!(
        "There is no file or demo called \"{name}\" (--demos lists the
demos)."
    )),
    several => {
        let names: Vec<&str> = several.iter().map(|demo|
demo.name()).collect();
        Err(format!("\"{name}\" could be any of: {}.", names.join(", ")))
    }
}
```

Tests que leen en lugar de escuchar

Los tests usan las trazas como lo haría una persona. En `lineup-audio`, sobre un renderer pequeño:

- `notes_start_on_their_step_and_last_their_length`, y `a_nudged_note_starts_late_by_its_nudge`.
- `a_slower_tempo_stretches_a_held_note`: una nota que suena a través de un cambio a la mitad del tempo dura sus pasos, en más segundos.
- `stopping_and_muting_cut_the_notes_sounding`, y `a_midi_out_ends_a_note_that_starts_again`.
- `a_looping_pattern_counts_its_steps_on`: las duraciones se miden a lo largo de los pasos tocados, a través del bucle.

Y en la aplicación, sobre canciones enteras:

- `the_trace_plays_every_note_of_the_export`: cada nota que tiene el proyecto, clip a clip, está en la traza exactamente una vez, y todas terminan cuando se acaba su duración.
- `swing_plays_the_off_beats_late`: con swing, las únicas notas que empiezan tarde son las segundas corcheas de cada pulso.
- `a_tempo_marker_changes_the_seconds_not_the_steps`: una marca de tempo en el paso 16 cambia cuándo empiezan las notas que vienen después, en segundos, pero no sus pasos.
- `re_metering_keeps_each_bar_as_long_in_seconds`: una canción pasada de 4/4 a 3/4 empieza cada compás en el mismo segundo que antes. Es el error del principio de este capítulo, el que solo el sonido podía mostrar, escrito como un test.
- `a_trace_while_playing_writes_a_table_for_each_playback`.

Los tests de otros capítulos también usan trazas, como el que lleva El Choclo por siete estilos y comprueba que cada nota de la melodía sigue empezando y durando como antes (Tempo, compás, swing y sensación).

Los ejercicios sobre trazas están en Ejercicios: el sonido.

Ejercicios: el sonido

Estos ejercicios practican esta parte sobre el código real: el secuenciador, el tempo y el compás, los instrumentos, el mezclador, MIDI, las exportaciones y la traza. Cada uno dice qué **ya existe**, para que lo amplíes en lugar de volver a construirlo. Algunos son cosas que probar en la aplicación y escuchar; otros son piezas de Rust con un final claro; los que se llaman «Piénsalo» son preguntas, sin código que escribir.

Haz los de Rust en una rama propia. Ejecuta `cargo test` después de cada cambio, y `cargo fmt --all --check` y `cargo clippy --workspace --all-targets -- -D warnings` antes de dar uno por terminado: la CI ejecuta los tres. Varios ejercicios amplían el formato del archivo de proyecto; cada uno necesita el siguiente número de formato libre, así que si haces dos, el segundo toma el número siguiente al del primero.

Temporización, tempo y compás

1. Un mapa de compases dentro de un patrón (lineup-core, difícil)

La canción ya tiene un mapa de compases: `Meter` guarda la `TimeSignature` inicial y una lista ordenada de `TimeMarker`s, y `bar_at`, `time_at` y `click` la leen (ver Tempo, compás, swing y sensación). Un `Pattern` sigue teniendo un solo `time: Option<TimeSignature>`, así que una parte que cambia de compás en cada compás se corta en un patrón por cada tramo de compases.

Haz que un patrón tenga un mapa de compases propio, con cambios en las barras de compás. `Pattern::length` cuenta pasos, y un compás ya no es un número fijo de ellos, así que:

- ¿Qué código que le pregunta a `TimeSignature::steps_per_bar` o a `Meter::bar_at` por un patrón tendría que preguntarle al mapa del patrón: las herramientas de notas, `Project::pattern_clicks`, las líneas de compás del piano roll?
- ¿Qué deberían preguntar la lista de reproducción y el piano roll cuando un clip empieza a mitad de un patrón?
- ¿Cómo lo tendría en cuenta `format_for`, para que un patrón en un solo compás se escriba exactamente como ahora?

Empieza por `Meter` en lugar de inventar un segundo tipo. `bar_at` recorre los tramos entre marcas uno por uno. ¿Por qué no puede usar simplemente `partition_point` sobre las marcas, como otras listas ordenadas del programa? ¿Qué necesitaría guardar junto a cada marca para que eso fuera posible?

Practica: leer una estructura de datos existente antes de ampliarla, aritmética de compases, mantener los archivos compatibles.

2. El swing de oído (la aplicación, fácil)

Abre *Midnight Shuffle* y tócala con el control **Swing** en recto, 50 y 100. ¿Qué instrumento muestra más el cambio? ¿Qué ajuste suena a marcha, y cuál a shuffle?

Después, calcúlalo. A 96 BPM, ¿cuántos milisegundos después del pulso empieza la segunda corchea en recto, en 50 y en 100? (Un pulso son 96 ticks, y `swing::delay` dice cuántos ticks tarde llega la segunda corchea.) A 120 BPM la tercera nota de un tresillo llega 333 ms después del pulso: ¿por qué la respuesta para 100 no es 333 ms a 96 BPM? Comprueba tus respuestas con una traza: la columna `late_ms`.

Practica: convertir ticks en tiempo.

3. El swing en Rust (lineup-core, medio)

`swing.rs` retrasa la segunda corchea de cada pulso cuando las notas se *tocan*: lee `delay`, `swung`, `swings_in` y `Project::played_notes`.

- Escribe un test de que una nota sostenida desde la segunda semicorchea de un pulso (paso 1) hasta la segunda corchea termina 16 ticks más tarde con swing completo en un canal de piano, pero no en uno de batería. ¿Qué argumento de `swung` marca la diferencia, y por qué un golpe de batería no debería alargarse?
- Añade swing de semicorcheas: dentro de cada corchea, retrasa la segunda semicorchea de cada par en una cantidad menor, elegida por un segundo campo o un enum `SwingUnit`. ¿Qué ticks se mueven, y cuánto con swing completo? Escribe un test de que los pares de semicorcheas siguen durando lo mismo en total, y de que las notas escritas no cambian. ¿Qué debe decir `swings_in` de 6/8, donde las semicorcheas también van de dos en dos? ¿Qué más toca el ajuste nuevo: el archivo del proyecto, el control, la exportación MIDI?

Practica: aritmética con ticks, tests unitarios con números exactos, mantener separados tocar y editar.

4. Compases irregulares de oído (la aplicación, fácil)

- Abre **Ajde Jano** y, en el cuadro Compás, escribe `7` sobre `8` agrupado como `2+2+3`, y haz clic en **Usar 2+2+3/8**. La melodía conserva sus notas, pero la banda ahora se apoya en otras corcheas. ¿Qué pulsos de la melodía caen ahora entre las notas del bajo?
- En la Biblioteca, toca los ritmos **Kalamatianós** y **Rachenitsa** uno después del otro. Da palmas al compás: ¿dónde está el pulso largo en cada uno?
- Elige `5/8` en el cuadro Compás de una canción en 4/4, con **Batería** y **Partes de acordes y bajo** marcadas. ¿Qué ritmo recibe, y cómo escribe el cuadro el compás?
- Abre **Süsserl** y mira la regla mientras suena. Cuenta los compases de dos y los de tres en la primera parte: ¿coinciden con 2 2 3, 2 2 3, 2 2 3 3?

Practica: oír pulsos agrupados, leer los grupos de una `TimeSignature`.

5. Piénsalo: el swing

- En una canción con swing completo, escribe una corchea con puntillo y una semicorchea en el piano roll, y después dos corcheas. ¿Qué par se mueve? Muchos músicos de jazz leen las dos cosas igual, como un tresillo. ¿Qué necesitaría saber Ferrous Lineup para hacerlo, y por qué podría salir mal en una marcha?
- Pasa una canción con swing a 6/8 con el cuadro Compás. ¿Qué pasa con el control Swing, y con el sonido? Escribe el mismo groove en 12/8 a mano: ¿dónde van las notas largas y las cortas?
- `played_notes` devuelve un `Cow`. ¿Qué cambiaría si devolviera un `Vec<NoteEvent>`? Cuenta cuántas notas copiaría `diff` en cada edición para una canción de 20 patrones y 8 canales.

Instrumentos y SoundFonts

6. Un cencerro grave (lineup-audio y lineup-core, fácil)

Lee `drums::cowbell`: dos ondas cuadradas a 540 y 800 Hz (a algo menos de una quinta: 800 / 540 son unos 680 cents, y una quinta, 702), un filtro de paso alto a 400 Hz, uno de paso bajo a 3 kHz, y dos decaimientos.

Convierte las dos frecuencias en parámetros (`cowbell_at(sample_rate, low, high)`) y añade un `Drum::LowCowbell` una octava más grave. El compilador enumera cada `match` que hay que actualizar en `drums::render` , `Drum::name` y `general_midi_drum` ; añade también el tambor nuevo a `Drum::ALL` (y cambia su longitud), o los menús y los tests de batería no lo verán. ¿Como qué nota General MIDI debería enviarse? (`general_midi::drum_name` las enumera.)

Añadir un tambor cambia también qué proyectos pueden abrir las versiones anteriores. Lee `format_for` y `Drum::is_new_in_format_3` : ¿basta esa comprobación para un tambor añadido ahora, o el archivo necesita el siguiente número de formato?

Practica: enums, `match` exhaustivo, un cambio que cruza crates.

7. Un sintetizador legato (lineup-audio, más difícil)

El Lineup Osc toca cada nota en una voz propia. Dale un modo mono, *legato*: solo suena una voz, y una nota nueva cambia la altura de esa voz sin reiniciar la envolvente. Añade un tiempo de `glide` para que la altura se deslice de la nota vieja a la nueva (usa `dsp::Smoothed` para la frecuencia). ¿Cómo cambia `pick_voice` ? ¿Qué partes de la máquina de estados `Stage` se saltan? ¿Dónde viviría el ajuste, y cómo llega al hilo de audio?

Practica: máquinas de estados, reutilizar piezas pequeñas de DSP.

8. Un clic en los contratiempos (lineup-audio, medio)

El metrónomo marca los pulsos, más agudo en el primer pulso de cada compás: `Project::played_clicks` y `pattern_clicks` construyen un `Click` (`None` , `Beat` o `Bar`) para cada paso, `metronome::click_at` elige uno, y `ClickVoice` hace el sonido. Añade un ajuste de **subdivisión** (corcheas, tresillos o semicorcheas) que lo haga sonar también entre los pulsos, más suave que el pulso y conservando el acento del compás. Decide si la subdivisión va en `Click` o en un segundo ajuste enviado al hilo de audio. ¿Debería la cuenta previa quedarse en el pulso?

El clic no debe estar en la exportación WAV ni en la exportación MIDI. ¿De dónde sale cada una, y cuál se llevaría el clic por accidente si viviera en un canal?

Practica: un ajuste nuevo de punta a punta, sonido en tiempo real sin reservar memoria.

9. Otra SoundFont (la aplicación, fácil)

Descarga otra SoundFont General MIDI en formato `.sf2` (*FluidR3_GM*, o la pequeña *TimGM6mb*), elígela con **Elegir SoundFont…** y toca el Canon en Re. ¿Qué cuerdas prefieres? ¿Suena una SoundFont mucho más fuerte que la otra? ¿Dónde pondrías un ajuste de volumen para la SoundFont entera: en `Sounds`, en el hilo de audio, o en cada `SoundFontPlayer`?

Practica: escuchar, encontrar dónde va un ajuste.

10. Un instrumento para la Melodía (lineup-core, fácil)

Las series dodecafónicas y los micromodos escriben su melodía en un canal llamado Lead (Melodía, en la aplicación en español), que `library::apply::synth_named` crea sin programa General MIDI, así que se queda con el Lineup Osc mientras los Teclados reciben un piano y el Bajo un bajo. Elige un instrumento para ella en el rack y escucha. ¿Qué instrumento le darías a la Melodía por defecto, y dónde iría en `synth_named`? ¿Qué tests te dirían que las canciones, que eligen su propia Melodía, siguen conservando la suya?

Practica: leer un `match` sobre nombres, valores por defecto.

11. Piénsalo: los instrumentos

- Una SoundFont suena en estéreo, pero `SoundFontPlayer::next_sample` junta izquierda y derecha en una sola muestra antes del balance del canal. ¿Qué haría falta para mantener separados su lado izquierdo y su lado derecho, y qué debería hacer entonces el Balance: mover los dos lados, o bajar uno?
- rustysynth tiene una reverberación y un chorus propios, y `enable_reverb_and_chorus` los apaga. Enciéndelos y escucha. Cada canal tendría entonces su propia sala: ¿qué cuesta eso, comparado con la reverberación compartida del mezclador, y por qué las mesas de mezcla usan un envío?
- La exportación va a 44.100 frames por segundo aunque tu tarjeta de sonido vaya a 48.000. Mira cómo se hace la batería (`drums::render` recibe la frecuencia de muestreo) y cómo toca el sampler un sonido grabado a otra frecuencia. ¿Podría una exportación sonar distinta de lo que oyes?

El mezclador y los efectos

12. Una ley de balance de potencia constante (lineup-core, fácil)

`Channel::pan_gains` mantiene completo el lado cercano y baja el lado lejano en línea recta. Cámbiala por una ley de *potencia constante*: con `angle = (pan + 1) × π / 4`, las ganancias son `[angle.cos(), angle.sin()]`. ¿Qué tests fallan, y por qué? Mueve un canal de I100 a D100 con cada ley y escucha: ¿cuál suena igual de fuerte todo el recorrido, y cuál suena más fuerte en el centro?

Practica: trigonometría sobre muestras, tests que fijan un diseño.

13. Aritmética de medidores (lineup-app y lineup-audio, fácil)

- Un nivel de 0,1 son ¿cuántos decibelios? ¿Hasta dónde se ilumina el medidor? Calcúlalo, y después compruébalo con `lit` y `FLOOR_DB` en `meters.rs`.
- `levels::raise` compara `peak > 0.0` en lugar de `peak >= 0.0`. ¿Qué haría `fetch_max` con los bits de −0,0, o de un NaN? (Mira cómo los ordena `f32::to_bits` como enteros.)

Practica: decibelios, números de coma flotante como bits.

14. Delays y compresores de oído (la aplicación, fácil)

- Pon un delay de corchea con puntillo con un 30 % de mezcla en la Melodía de una demostración, y tócala a 60 BPM y a 140 BPM. ¿Cuántos milisegundos separan los ecos cada vez? Compruébalo con `Delay::seconds`.
- Pon un compresor a −30 dB y 10:1 en la batería. Después sube su **Ganancia** hasta que la batería suene tan fuerte como antes. ¿Qué pasó con la diferencia entre los golpes fuertes y los suaves? ¿Por qué una batería con un compresor fuerte suena «aplastada»?

Practica: escuchar lo que hace un efecto.

15. Una campana más ancha o más estrecha (lineup-core, lineup-audio y lineup-app, medio)

La banda central del ecualizador es una campana de ancho fijo, una Q de 1 (`Biquad::set` en `lineup-audio/src/effects.rs`). Añade un `width` a `Equalizer` y un deslizador para él, y úsalo en `Biquad::set`. ¿Qué dice el Cookbook que vale alpha para una Q dada? Una canción que fija el ancho necesita el siguiente número de formato libre: enséñaselo a `format_for`, y deja el campo fuera del archivo cuando tiene su valor por defecto.

Después escribe un test al estilo de los que hay: un seno en el centro de la campana se refuerza en la ganancia, y uno dos octavas más lejos se refuerza menos con una campana estrecha que con una ancha.

Practica: la fórmula de un filtro, un campo guardado nuevo de punta a punta.

16. Un envío de delay compartido (lineup-core, lineup-audio y lineup-app, difícil)

Cada canal ya tiene un delay propio entre sus efectos. Añade un segundo **envío** junto al de la reverberación, a un único delay que comparten todos los canales, para que varios canales hagan eco juntos. Esto atraviesa el archivo del proyecto, el hilo de audio y la ventana del mezclador.

1. En `Channel`, añade `delay_send: f32`, por defecto `0.0`, e inclúyelo en las comprobaciones de rango y en `has_plain_mix`. Dale el valor por defecto en cada constructor.
2. En `Mixer`, añade los ajustes propios del delay: el tiempo en pasos (para que siga al tempo, como `Delay`), la realimentación y la mezcla. Valídalos en `Mixer::valid`.
3. Añade `Command::SetSharedDelay` y `Command::SetDelaySend`, y un delay compartido que reserve su búfer antes de llegar al hilo de audio, nunca en `process`. La `DelayLine` del delay de canal muestra cómo. Sigue el modelo de la reverberación para el envío.
4. Añade un deslizador **Delay** a cada línea del mezclador, junto a **Reverberación**, y una pequeña sección del delay junto a los controles de la reverberación.
5. Enséñale a `format_for` que un envío de delay distinto de cero o unos ajustes del delay distintos de los de por defecto necesitan el siguiente número de formato libre, y deja los campos fuera del texto cuando tienen sus valores por defecto.
6. Prueba el camino del audio (un envío de 0 no hace eco, uno mayor que 0 sí) y el archivo (un archivo viejo se abre con envío 0; una canción con envío de delay se

escribe en el formato nuevo y se lee igual).

Terminado cuando el deslizador se oye en la aplicación y en una exportación WAV, el hilo de audio no reserva nada mientras corre, y una canción sin el delay se sigue escribiendo en el formato de antes.

Practica: todo lo de esta parte a la vez.

17. El orden de partitura (la aplicación y lineup-core, fácil)

- Abre *In the Mood* y después *Sing, Sing, Sing*, con el mezclador abierto. ¿En qué sección está el canal «Vientos» de cada canción? Mira la columna del instrumento para ver por qué.
- Abre *Danny Boy*, haz clic derecho en su Melodía en el rack de canales y elige **Instrumento > Flautas > 74 Flauta**. ¿A dónde se mueve en el mezclador?
- En `score_order.rs`, los saxofones están entre los clarinetes y los fagotes, como en la mayoría de las partituras de orquesta. En una partitura de big band los saxofones van primero, encima de las trompetas. ¿Qué cambiarías para tener el orden de big band, y qué test te diría que rompiste el de orquesta?

Practica: datos que deciden una distribución.

18. Piénsalo: la dinámica

- El `Follower` del compresor sube en 10 ms (`ATTACK_SECONDS`) en lugar de seguir cada onda. Escribe un test que pase un seno de 50 Hz por un compresor cuyo seguidor sea mucho más rápido, y mide cuánto se aleja la forma de la salida de un seno. ¿Qué te dice eso de los 10 ms?
- Muchas mesas permiten que un compresor escuche un canal mientras baja otro: un bajo que se agacha cada vez que suena el bombo («compresión side-chain»). Esboza lo que necesitarían los efectos de canal para eso. ¿La muestra de qué canal leería el seguidor, y en qué orden tendría que mezclar los canales el hilo de audio?

MIDI

19. Teclas de octava (lineup-app, fácil)

El teclado del equipo cubre de Do4 a Mi6. Añade dos teclas que lo bajen y lo suban una octava, como `-` y `=` (`Key::Minus` y `Key::Equals`), que todavía no tocan ninguna nota. Mantén el desplazamiento entre Do1 y Do7, y muéstralo en el menú MIDI, junto a **Teclado del equipo como piano**.

El caso difícil: mantén Z, pulsa `=`, y después suelta Z. ¿Qué nota debe terminar? Mira `typing_down`, que guarda la nota que tocó cada tecla, no solo la tecla. ¿Por qué se escribió así, si la nota podría volver a calcularse a partir de la tecla?

Practica: aritmética de `Note`, recordar lo que se hizo en lugar de volver a calcularlo.

20. Teclear más suave y más fuerte (lineup-app, fácil)

Las teclas tecleadas suenan con `DEFAULT_VELOCITY`, así que las tomas grabadas con el teclado del equipo tienen todas la misma velocidad, mientras que las de un teclado MIDI conservan la fuerza con que se tocó cada tecla. Añade un deslizador «Velocidad del teclado» al menú MIDI (`egui::Slider::new(&mut value, 1..=127)`), y haz que Mayúsculas toque a 127 mientras se mantiene pulsada.

Mayúsculas no cambia `physical_key`, así que el piano sigue funcionando. Pero mira el filtro de `typing_keyboard`: una tecla que bajó con Mayúsculas, ¿sube bien cuando se suelta antes Mayúsculas?

Practica: deslizadores de egui, modificadores del teclado.

21. Notas a través del bucle (lineup-core, medio)

`recorded_note` tiene dos límites. Una nota sostenida a través del final del bucle se corta ahí, y una tecla sostenida más de un bucle entero se graba como una nota corta.

- Haz que una tecla sostenida a través del final del bucle grabe **dos** notas: una hasta el final del patrón, y otra desde el paso 0 para el resto. La función devuelve entonces un `Vec<NoteEvent>` (o un array y una cuenta). Escribe antes los tests: (14, 2) en un patrón de 16 pasos da (14, 2) y (0, 2).
- Para las notas largas, la posición sola no basta. Haz que el hilo de audio cuente también cuántas veces ha dado la vuelta el bucle, o que publique el número de

pasos tocados desde Reproducir en lugar de la posición en el bucle. ¿Qué es más sencillo de compartir con un solo `AtomicU64`? ¿Cuál puede seguir usando `finish_recording`?

Practica: cambiar el tipo que devuelve una función y a quienes la llaman, tests como especificación.

22. Grabar en modo canción (los tres crates, más difícil)

Grabar pasa a modo patrón, porque una toma va al patrón que se repite. En modo canción, la posición es un paso de la *canción*, así que una tecla podría ir al clip que suena en ese momento, en `step - clip.start`.

Resuelve las preguntas antes de escribir código. Dos clips pueden sonar a la vez, en pistas distintas: ¿cuál se lleva la nota? ¿Y si no suena ningún clip ahí: debería crearse uno nuevo? Un clip puede sonar varias veces en la canción, y grabar en uno de ellos los cambia todos, porque son el mismo patrón. ¿Es eso lo que querrías? ¿Cómo lo resuelven otros estudios que conoces?

Practica: diseñar el comportamiento antes que el código.

23. Las otras piezas de un archivo de formato 2 (lineup-core, más difícil)

`midi_file::read::read` convierte los bytes de un archivo en eventos, y `midi_file::import::import` hace un proyecto con ellos. Faltan todavía dos rincones.

Un archivo de **formato 2** no es una canción sino varias piezas independientes, una por pista. `import` se queda solo con la primera pista. Añade una forma de elegir una pieza: enumerarlas en el aviso, preguntar en un diálogo, o importar cada una como un proyecto propio. ¿Qué deberían decir los tests sobre un archivo de formato 2 con tres piezas?

El otro es el meta evento de **desplazamiento SMPTE** (`FF 54`): cinco bytes que dicen dónde empieza una pista en un código de tiempo, para música escrita sobre una película. La temporización SMPTE de la cabecera (`Timing::PerSecond`) ya funciona; este evento se salta y `meta_event` lo descarta. Léelo y decide qué debería hacer con él un proyecto sin video. Prueba los dos casos con archivos pequeños construidos en el módulo de tests, como hacen los tests que hay.

Practica: analizar datos binarios con patrones de slice, diseñar una elección.

24. Compensar la latencia (lineup-audio y lineup-app, más difícil)

La latencia hace que cada tecla grabada llegue un poco tarde. cpal sabe cuánto: el callback de audio recibe un `&cpal::OutputCallbackInfo`, cuyo `timestamp()` tiene dos instantes, `callback` (ahora) y `playback` (cuándo se oirá este bloque). La diferencia entre ellos es la latencia de salida.

El callback ignora ese argumento (`_: &cpal::OutputCallbackInfo` en `build_stream`). Pásale la latencia al hilo de audio, publícala como un número de pasos en otro atómico, y réstala de la posición de cada tecla antes de que `recorded_note` la redondee. Una posición justo después del paso 0 menos la latencia debe dar la vuelta hasta el final del bucle, no bajar de cero.

Después piensa en la otra dirección. Los mensajes de la Salida MIDI salen cuando se hace su bloque, pero el sonido de los instrumentos incorporados de ese mismo bloque se oye un poco más tarde. Así que un sintetizador externo toca ligeramente *antes* que el resto. ¿Cómo podría el hilo de envío retrasar cada mensaje en la latencia? ¿Qué tendría que llevar cada mensaje?

Practica: leer la documentación de un crate (`cargo doc --open -p cpal`), el tiempo como dato.

25. Un test de note-off MIDI (lineup-audio, medio)

Demuestra que **Detener** envía note-offs para las notas que el secuenciador empezó en un canal de Salida MIDI. Lee `midi_out_channels_send_their_notes` y `stopping_releases_held_notes` en el módulo de tests del motor, que tiene las funciones auxiliares `add_midi_out`, `received` y `note_off`. Añade `stopping_releases_midi_out_notes`: una nota larga en el canal MIDI 3, **Reproducir**, renderiza un paso y comprueba el NoteOn; envía **Detener**, renderiza unos frames y comprueba el NoteOff. Haz la nota lo bastante larga para que no pueda haber terminado por su propio gate.

Ejecuta el test una vez sin su línea `Command::Stop`, para verlo fallar, y después vuelve a poner la línea. Un test que nunca has visto fallar puede no estar probando nada.

Practica: el arnés de tests del motor, tests de regresión.

26. La Salida MIDI desde los menús (la aplicación, fácil)

- Envía el Bajo de una demostración con banda por MIDI en el canal 2, y después aplica otro estilo desde la Biblioteca. ¿Cuántos canales de Bajo hay después? Lee `find_synth` en `library/apply.rs` para ver por qué.
- ¿Qué tambor General MIDI es la nota 60? Un tambor escrito en Do4 y enviado tal como está escrito, en lugar de como su propia nota, lo tocaría. Encuéntralo con `general_midi::drum_name`.
- `Channel::send_over_midi` limita el canal con `.min(15)` en lugar de devolver un error. ¿Dónde más se limita así un número que viene de un menú o de un archivo, y dónde se rechaza en cambio? ¿Qué es mejor para un número leído de un archivo?

Practica: leer código para explicar un comportamiento.

27. Piénsalo: MIDI

- La cola en vivo guarda 256 mensajes, y `LiveInput::send` devuelve `false` cuando está llena. `on_message` lo ignora. ¿Qué es peor perder, una tecla que baja o una que sube? ¿Podría un diseño hacer imposible perder una tecla que sube, por ejemplo compartiendo *qué teclas están bajadas* (128 bits) en lugar de enviar eventos?
- El hilo de envío se despierta mil veces por segundo, aunque ningún canal se envíe por MIDI. ¿Podría dormir más cuando no hay nada que enviar? ¿Qué costaría dormir más cuando sí lo hay?
- `release_live_keys` suelta todas las teclas cuando se inserta o se quita un canal. La otra forma sería renumerar `live_keys` para seguir a los canales. ¿Qué es más fácil de hacer bien? ¿Notaría alguna vez la diferencia quien toca?
- midir le da a `on_message` una marca de tiempo en microsegundos, que el código ignora en favor de la posición del hilo de audio. ¿Cuándo sería más precisa la marca de midir? ¿Por qué no se pueden comparar sin más los dos relojes?
- Un patrón tocado por un canal de Salida MIDI y exportado como archivo `.mid` envía las mismas notas en los dos casos. ¿Podrían la exportación y el hilo de audio compartir más código? ¿Qué necesitaría saber ese código sobre el tiempo?

Exportar

28. Exportar un patrón como audio (lineup-app, fácil)

Archivo > Exportar patrón como MIDI… escribe el patrón seleccionado como archivo MIDI. Añade lo mismo como *audio*: un elemento **Exportar patrón como WAV…**, que guarde el patrón seleccionado, tocado cuatro veces, como archivo WAV.

El plan evidente es el modo patrón: `SetMode(PlayMode::Pattern)`, `SelectPattern`, y renderizar cuatro veces la duración del patrón más la cola. Pero el modo patrón repite el patrón para siempre, así que la cola contendría el patrón empezando por quinta vez: el problema que la exportación resuelve con pasos en silencio, y el modo patrón no tiene una duración de canción que alargar. En su lugar, haz una *copia* del proyecto, reemplaza su lista de reproducción por cuatro clips del patrón en una pista, y exporta esa copia con el código que ya tienes. ¿Qué tiene que cambiar en `song_render_with` para que pueda renderizar un proyecto distinto de `self.project`?

Practica: trabajar sobre una copia, hacer una función más general.

29. Exportar la región de bucle (lineup-app, medio)

Las canciones pueden tener una región de bucle, y el modo canción toca solo esa parte. Añade **Archivo > Exportar región de bucle…**, que escriba solo la región en un archivo WAV. `Project::playback_arrangement()` ya devuelve exactamente la parte que toca el modo canción, con las repeticiones desplegadas. Haz que `song_render_with` reciba un `Arrangement` en lugar de llamar siempre a `project.arrangement()`, y calcula el número de frames a partir de la duración y los tempos de ese arreglo. ¿Cuál debería ser el nombre del archivo? ¿Qué debería hacer el elemento del menú cuando no hay región de bucle?

Practica: reutilizar un cálculo, pensar en una entrada vacía.

30. Stems: un archivo por canal (lineup-app, medio)

Los músicos a menudo quieren *stems*: la canción exportada una vez por canal, cada archivo con solo ese canal, para mezclarla en otro sitio. Añade **Archivo > Exportar stems…**, que pida una carpeta y escriba `Kick.wav`, `Clap.wav` y así sucesivamente.

Cada stem es una exportación con todos los demás canales silenciados (`Command::SetMuted`). Renderizarlos uno tras otro es lo más sencillo; renderizarlos en

varios hilos a la vez es más rápido en un ordenador con varios núcleos. El canvas de la exportación de video ya pinta franjas de filas con `std::thread::scope`: ¿podrían hacer lo mismo los stems, o necesitan `thread::spawn`, dado que la ventana no debe esperarlos? ¿Cómo debería mostrar una sola barra de progreso varias exportaciones? ¿Y qué pasa con dos canales con el mismo nombre, o con un nombre que lleva una `/`?

Practica: hilos, nombres de archivo a partir de texto del usuario.

31. Dos vistas en un video, y una comprobación de sincronía (la aplicación, medio)

1. Abre **El Choclo** y elige **Archivo > Exportar WAV…**; guarda `el-choclo.wav`.
2. Elige **Archivo > Exportar video…**, marca **Hoja de acordes** y **Piano roll**, ponlas una al lado de la otra en la vista previa, elige **1280 × 720**, y exporta `el-choclo.mp4`.
3. Si tienes `ffprobe` instalado, compara las duraciones:

   ```
   ffprobe -v error -show_entries format=duration -of default=nw=1:nk=1
   el-choclo.wav
   ffprobe -v error -show_entries format=duration -of default=nw=1:nk=1
   el-choclo.mp4
   ```

4. Mira un compás donde cambia el acorde. El compás enmarcado en la hoja de acordes, el cursor en el piano roll y el sonido deberían coincidir ahí. Comprueba un segundo punto, como el primer compás después de una repetición.

Terminado cuando las duraciones no difieren en más de una o dos imágenes, las dos vistas muestran el mismo compás en el mismo momento, y puedes decir por qué `start_video` dibuja cada imagen **antes** de renderizar el sonido que suena bajo ella.

Practica: comprobar un diseño con herramientas externas.

32. Piénsalo: exportar

- El contador de progreso usa `Ordering::Relaxed`. Supón que la ventana abriera el archivo en el momento en que el contador llega al total, sin esperar a `join`. ¿Qué podría encontrar? (Mira lo que hace `export_wav_cancellable` después de

su última llamada a `progress`.) ¿Lo arreglaría un `Ordering` más fuerte, o el problema está en otro sitio?
- El sonido de un video es MP3 a 256 kbit/s, mientras que el archivo WAV son las muestras exactas. ¿Podría el video llevar las muestras sin cambios? ¿Qué caja MP4 cambiaría, y qué reproductores dejarían de reproducirlo?
- `repeat_picture` hace que la última imagen dure más cuando nada cambió. ¿Qué saldría mal si la escena comparara *píxeles* en lugar de triángulos para decidirlo? ¿Cuánto costaría?

La traza de reproducción

33. La traza de un compás insertado y borrado (la aplicación y una terminal, fácil)

La hoja de acordes y la lista de reproducción tienen **Insertar un compás antes**, **Insertar un compás después** y **Eliminar este compás**; `--trace` escribe cada nota que toca el reproductor.

1. Escribe una traza de una demostración:

   ```
   cargo run --release -- --trace "El Choclo" --out before.tsv
   ```

2. Abre la misma demostración, guárdala como `inserted.lineup`, haz clic derecho en el compás 3 de la hoja de acordes y elige **Insertar un compás después**. Guarda, y escribe la traza del archivo guardado:

   ```
   cargo run --release -- --trace inserted.lineup --out after.tsv
   ```

3. Compara las dos tablas, sin las líneas `#`:

   ```
   diff <(grep -v '^#' before.tsv) <(grep -v '^#' after.tsv) | head -40
   ```

   Cada nota que empezaba en el compás 4 o después debería estar ahora un compás más tarde; las notas de los compases 1 a 3 no deberían cambiar.

4. Borra el compás que insertaste (es el compás 4), guarda, vuelve a escribir la traza y compárala con `before.tsv`.

Terminado cuando la traza con el compás insertado muestra el desplazamiento de un compás en `start`, `bar` y `step`, la del compás borrado coincide con la primera, y puedes señalar el código de `bars.rs` que mueve los clips, las marcas y las secciones.

Practica: comprobar una edición por lo que toca.

34. Leer trazas (una terminal, fácil)

- Escribe la traza de una demostración con un ritardando (una marca de tempo que se desliza) y lee las líneas `# tempo`. ¿Cuántas hay, y por qué una por paso?
- Sube la **Soltura** en una canción, escribe su traza y usa `awk` para encontrar la nota más tardía (la columna `late_ms`). ¿Llega alguna vez más tarde de lo que dice el ajuste?
- Con **Escribir un registro mientras suena** activado en la aplicación, toca un patrón, silencia su canal en mitad de una nota larga y detén. Busca la nota en el archivo: ¿qué dice su columna `end`?

Practica: `awk` sobre tablas, leer un informe.

35. Piénsalo: el trazador

- El trazador busca las notas cuya duración se acabó al principio de cada frame, antes de que el tempo pueda cambiar en ese frame. ¿Qué saldría mal con una nota que termina en el mismo frame que un cambio de tempo, si mirara después?
- Una traza pierde eventos cuando su cola está llena, y dice cuántos. ¿Podría en cambio conservar algunos eventos que importan más (los `NoteOff`, por ejemplo) y descartar antes otros? ¿Con qué tendría que lidiar entonces el lector?

Clases de altura, notas y su escritura

Archivos: `crates/lineup-core/src/pitch.rs` y `crates/lineup-core/src/harmony/pitch_set.rs`

Esta parte del libro trata de la teoría musical que hay dentro del programa: escalas, acordes, números romanos, conjuntos, series y las reglas de la conducción de voces, cada una en unos pocos tipos pequeños de Rust con métodos, probados como cualquier otro código. La teoría en sí se enseña en los capítulos de Aprender armonía del manual de usuario; cada capítulo de aquí cuenta la idea musical en pocas líneas y luego muestra cómo la guarda el código.

Todo lo demás se construye sobre los tipos de este capítulo: una nota, las doce teclas de una octava, el nombre con que se escribe una tecla y un conjunto de teclas.

La música, en pocas palabras

Un piano tiene 88 teclas, pero solo doce notas distintas, que se repiten en cada octava. La distancia de una tecla a la siguiente, blanca o negra, es un **semitono**:

```
Do Do♯ Re Re♯ Mi Fa Fa♯ Sol Sol♯ La La♯ Si Do
0  1   2  3   4  5  6   7   8    9  10  11 12
```

El nombre de una nota sin su octava es su **clase de altura**: todos los Do del piano tienen la clase de altura Do. La armonía trabaja sobre todo con clases de altura; un acorde de Do mayor es Do, Mi y Sol, en cualquier octava y en cualquier orden. La octava se escribe después del nombre: **Do4** es el Do central y **La4** el La con el que afina una orquesta, 440 Hz.

Cada tecla negra tiene dos nombres: la tecla entre Do y Re es Do♯ y también Re♭. Qué nombre es el correcto depende de la música que la rodea, y la regla es que una escala usa cada uno de los siete nombres una vez. Fa mayor tiene un Si♭, nunca un La♯, porque ya tiene un La. La séptima nota de Fa♯ mayor es Mi♯, la tecla que suele llamarse Fa, porque Fa♯ ya usa el nombre Fa. De vez en cuando una nota necesita dos signos: la séptima de Sol♯ menor armónica es Fa𝄪 (Fa doble sostenido, la tecla que suele llamarse Sol), y la séptima de un acorde de séptima disminuida sobre Do es Si𝄫 (Si doble bemol, que suena como La). El capítulo Notas, intervalos y escalas del manual lo cuenta con más calma.

Así que el código necesita dos cosas distintas: qué **tecla** del piano es una nota, y cómo se **llama**. Son tipos distintos.

Doce clases de altura: un enum

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash,
Serialize, Deserialize)]
pub enum PitchClass {
    C,
    CSharp,
    D,
    DSharp,
    E,
    F,
    FSharp,
    G,
    GSharp,
    A,
    ASharp,
    B,
}
```

Hay exactamente doce clases de altura, así que un enum encaja: un `PitchClass` nunca puede ser 13 ni -1. Las variantes no llevan datos (un enum **sin campos**), y Rust las numera desde 0 en el orden en que están escritas. La palabra clave `as` convierte un enum así en su número, que es la cantidad de semitonos por encima de Do. En el código las variantes llevan las letras con que se nombran las notas en inglés: `C` es Do, `CSharp` es Do♯, y así hasta `B`, que es Si.

```
pub fn semitone(self) -> u8 {
    self as u8
}
```

Ir en la otra dirección pide cuidado, porque no todo número es una clase de altura. `from_semitone` primero da la vuelta y luego busca la respuesta en el array `ALL`, así que no puede fallar:

```
pub fn from_semitone(semitone: u8) -> Self {
    Self::ALL[(semitone % 12) as usize]
}
```

`transpose` suma semitonos y da la vuelta a la octava. Recibe un `i32` para poder bajar también. Con un número negativo, `%` da un resto negativo (`-1 % 12` es `-1` en Rust), así que usa `rem_euclid`, que siempre da de 0 a 11:

```rust
pub fn transpose(self, semitones: i32) -> Self {
    let wrapped = (self.semitone() as i32 + semitones).rem_euclid(12);
    Self::from_semitone(wrapped as u8)
}
```

`semitones_up_to` responde la pregunta contraria: cuánto hay que subir de una clase de altura a otra. De La hasta Do son 3 semitonos (La♯, Si, Do), y de Do hasta La son 9; sumar 12 antes de restar evita que el `u8` baje de cero:

```rust
pub fn semitones_up_to(self, other: PitchClass) -> u8 {
    (other.semitone() + 12 - self.semitone()) % 12
}
```

Un match sobre dos valores a la vez

El nombre de todos los días de una clase de altura depende de dos cosas: cuál es, y si las teclas negras se escriben como sostenidos o como bemoles. `match` puede mirar las dos a la vez si se ponen en una **tupla**:

```rust
pub fn name(self, spelling: Spelling) -> &'static str {
    use PitchClass::*;
    use Spelling::*;
    match (self, spelling) {
        (C, _) => "C",
        (CSharp, Sharps) => "C#",
        (CSharp, Flats) => "Db",
        (D, _) => "D",
        // ...
        (B, _) => "B",
    }
}
```

- `use PitchClass::*;` dentro de la función le permite escribir `C` en lugar de `PitchClass::C`, sin repartir esos nombres cortos por todo el archivo.
- `_` encaja con cualquier cosa: una tecla blanca tiene el mismo nombre de las dos maneras.
- El compilador comprueba que las ramas cubran las 24 combinaciones. Si falta `(GSharp, Flats)`, se niega a compilar y dice qué caso falta.
- `&'static str` es una cadena prestada que vive tanto como el programa: los literales de cadena se guardan dentro del propio programa.

`Spelling` es un enum de dos variantes, `Sharps` (el `#[default]`) y `Flats`. Cuál usa cada tonalidad, y la elección más fina de una letra para cada nota, está en Escalas, modos y tonalidades.

`is_black_key` usa la macro `matches!`, una forma corta de preguntar «¿encaja este valor con este patrón?» como un `bool`. La `|` significa «o»:

```
pub fn is_black_key(self) -> bool {
    use PitchClass::*;
    matches!(self, CSharp | DSharp | FSharp | GSharp | ASharp)
}
```

`PitchClass` también implementa `Display`, así que `{}` en `format!` lo escribe con sostenidos, con letras o en solfeo según los ajustes (Dos idiomas: traducciones, y letras o solfeo).

Una nota: un número MIDI con un tipo propio

Una nota con su octava se guarda como su número de nota MIDI, de 0 a 127, envuelto en un struct propio:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash,
Serialize, Deserialize)]
#[serde(from = "u8", into = "u8")]
pub struct Note(u8);
```

Un struct de un solo campo que se usa para dar a un valor un tipo propio es un **newtype**. Un `Note` es un `u8` en memoria, pero el compilador no deja pasar una nota donde se espera una velocidad (también un `u8`), ni al revés. El campo es privado, así que las únicas formas de crear una nota son las que ofrece `Note`, y la mantienen dentro del rango:

```
pub fn new(midi: u8) -> Self {
    Note(midi.min(Self::MAX.0))
}
```

Lo mismo vale para los archivos. `#[serde(from = "u8", into = "u8")]` le dice a serde que guarde una nota como su número y que la lea a través de `From<u8>`, que llama a `Note::new`, así que ni siquiera un archivo de proyecto editado a mano puede tener una nota por encima de 127:

```
impl From<u8> for Note {
    fn from(midi: u8) -> Self {
        Note::new(midi)
    }
}
```

Algunas notas tienen nombre en el código: `Note::C4` es el Do central (60), `Note::A4` el La de afinación (69), `Note::MIN` y `Note::MAX` los extremos del rango.

Octavas

La clase de altura es el número módulo 12, y la octava es el número dividido entre 12, menos uno, porque la nota MIDI 60 es el Do central y el Do central se llama Do4:

```
pub fn octave(self) -> i8 {
    (self.0 / 12) as i8 - 1
}
```

Así, la nota MIDI más grave es Do-1 y la más aguda Sol9. La octava va con el nombre de la nota, no con la tecla: cambia entre Si y Do, así que la tecla un semitono por encima de Si3 es Do4.

Mover una nota: `Option` para «puede que no exista»

Una clase de altura siempre tiene otra siete semitonos más arriba; una nota no. Siete semitonos por encima de Sol9 está más allá del final de MIDI. `transpose` devuelve un `Option<Note>`, y `None` cuando la nota nueva quedaría fuera del rango:

```
pub fn transpose(self, semitones: i32) -> Option<Note> {
    let midi = self.0 as i32 + semitones;
    (0..=Self::MAX.0 as i32)
        .contains(&midi)
        .then_some(Note(midi as u8))
}
```

`contains` pregunta si un rango contiene un número, y `then_some` convierte un `bool` en un `Option`: `Some` del valor si es verdadero, `None` si es falso. Quien llama decide qué significa una nota imposible: un acorde construido cerca del extremo agudo del teclado deja fuera sus notas más altas, y la herramienta de transportar se niega a mover una selección entera si una de sus notas se saldría del teclado, así que no se pierde ninguna nota (Herramientas de edición como funciones simples).

La frecuencia

Ferrous Lineup afina como se afinan los pianos, en **temperamento igual de doce tonos**: cada semitono multiplica la frecuencia por el mismo número, la raíz

duodécima de dos (alrededor de 1,0595), así que doce semitonos la duplican exactamente. La4 es 440 Hz:

```
pub fn frequency(self) -> f32 {
    let semitones_from_a4 = self.0 as f32 - Self::A4.0 as f32;
    440.0 * 2.0_f32.powf(semitones_from_a4 / 12.0)
}
```

El Do central está nueve semitonos por debajo de La4, así que su frecuencia es 440 · 2^(-9/12), unos 261,63 Hz. El sintetizador toca esta frecuencia, y un sampler toca una grabación más deprisa o más despacio con la misma regla (Instrumentos: batería, sampler y Lineup Osc).

Letras y nombres

Cómo se llama una nota es una letra y algunos sostenidos o bemoles. Las letras son un enum propio, las siete teclas blancas. En el código son las letras de la nomenclatura inglesa, de `C` (Do) a `B` (Si); en solfeo cada letra es una sílaba, que da `Letter::solfege`.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Letter {
    C,
    D,
    E,
    F,
    G,
    A,
    B,
}
```

Las letras dan la vuelta de siete en siete como las clases de altura dan la vuelta de doce en doce, así que tienen el mismo tipo de métodos: el semitono por encima de Do de su tecla blanca, leído de una tabla, y subir o bajar con `rem_euclid`:

```
pub fn semitone(self) -> u8 {
    [0, 2, 4, 5, 7, 9, 11][self as usize]
}

pub fn up(self, steps: i32) -> Letter {
    Self::ALL[(self as i32 + steps).rem_euclid(7) as usize]
}
```

Dos letras por encima de La está Do (La, Si, Do), y `steps_up_to` cuenta en el otro sentido: de La a Do hay 2.

Un nombre es una letra y una cuenta de signos, positiva para los sostenidos y negativa para los bemoles:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct NoteName {
    pub letter: Letter,
    /// Sharps if above zero, flats if below: 1 is "#", -2 is "bb".
    pub accidental: i8,
}
```

Un solo `i8` guarda las dos direcciones, así que un nombre nunca puede ser «un sostenido y un bemol» a la vez, y Fa𝄪 es simplemente `NoteName::new(Letter::F, 2)`. La tecla del piano que nombra es la tecla blanca de su letra movida por los signos:

```
pub fn pitch_class(self) -> PitchClass {

PitchClass::from_semitone(self.letter.semitone()).transpose(i32::from(self
.accidental))
}
```

Un nombre para una tecla, con una letra dada

Casi siempre el código sabe la tecla y la letra que quiere, y necesita los signos. Si con la letra de Do es Do♭ (un bemol); Fa con la letra de Mi es Mi♯ (un sostenido). `with_letter` cuenta cuánto sube la clase de altura desde la tecla blanca de la letra, y toma el camino más corto: hasta cinco semitonos hacia arriba son otros tantos sostenidos, y seis o más son bemoles hacia abajo:

```
pub fn with_letter(pitch_class: PitchClass, letter: Letter) -> Self {
    let up = (pitch_class.semitone() + 12 - letter.semitone()) % 12;
    // Up to five semitones up is sharps; six or more is flats below.
    let accidental = if up <= 5 { up as i8 } else { up as i8 - 12 };
    NoteName::new(letter, accidental)
}
```

Sol con la letra de Fa está dos arriba, Fa𝄪; La con la letra de Si está diez arriba, es decir, dos abajo: Si𝄫.

Sobre él se construyen dos nombres más sencillos. `plain` toma la letra del nombre de todos los días, así que da Do♯ o Re♭ para la tecla negra y su propia letra a las teclas blancas. `chromatic` es el nombre que se usa cuando no hay tonalidad que seguir,

como en la música dodecafónica: Do, Do♯, Re, Mi♭, Mi, Fa, Fa♯, Sol, La♭, La, Si♭, Si, con sostenidos para dos teclas negras y bemoles para las otras tres:

```
pub fn chromatic(pitch_class: PitchClass) -> Self {
    let spelling = match pitch_class {
        PitchClass::CSharp | PitchClass::FSharp => Spelling::Sharps,
        _ => Spelling::Flats,
    };
    NoteName::plain(pitch_class, spelling)
}
```

Elegir la letra es trabajo de la tonalidad: cada nota de una tonalidad toma la letra siguiente, y una nota de fuera de la tonalidad toma la letra que los músicos usan más a menudo para ella. Eso es `Scale::spell`, en Escalas, modos y tonalidades.

Los intervalos también cuentan letras

Un intervalo tiene un tamaño en semitonos y un tamaño en letras. Una tercera mayor son cuatro semitonos y dos letras: sobre Do es Mi, sobre La es Do♯ (no Re♭, que está tres letras más arriba), y sobre Mi♭ es Sol. `above` mueve la tecla los semitonos, la letra las letras, y deja que `with_letter` encuentre los signos:

```
pub fn above(self, semitones: u8, letters: u8) -> NoteName {
    let pitch_class = self.pitch_class().transpose(i32::from(semitones));
    NoteName::with_letter(pitch_class, self.letter.up(i32::from(letters)))
}
```

La séptima disminuida son nueve semitonos y seis letras: sobre Si es La♭, y sobre Do es Si𝄫. Así se escribe cada acorde a partir de su fundamental (Acordes: construirlos, nombrarlos e invertirlos), y así se lleva una hoja de acordes a otra tonalidad, letra por letra, como la transporta un músico.

Un símbolo de acorde transportado a veces cae en un nombre con dos signos. Llevar una hoja de acordes de Mi menor a Sol♯ menor sube todo cuatro semitonos y dos letras, así que el D♯°7 de Mi menor pasa a ser F𝄪°7. En una partitura eso es lo correcto, pero un símbolo de acorde se lee de un vistazo, así que los símbolos toman el nombre más sencillo de la misma tecla:

```
pub fn simplified(self) -> NoteName {
    if self.accidental.abs() < 2 {
        return self;
    }
    let spelling = if self.accidental > 0 {
        Spelling::Sharps
    } else {
        Spelling::Flats
    };
    NoteName::plain(self.pitch_class(), spelling)
}
```

Mi𝄫 pasa a ser Re, Fa𝄪 pasa a ser Sol, y el D♯°7 de antes, G°7.

La octava de una nota escrita

El nombre propio de una nota, `Note::name`, usa el nombre de todos los días, así que el Do central es siempre «Do4». Pero cuando una tonalidad escribe el Do central como Si♯, hay que escribirlo «Si♯3»: el número de octava va con la letra, y Si♯3 es el Si de la octava de abajo, subido un semitono. Del mismo modo, el Si por debajo del Do central escrito como Do♭ es «Do♭4». `name_as` quita la letra y los signos para encontrar el Do del comienzo de la octava de la letra, y numera la octava desde ahí:

```
pub fn name_as(self, name: NoteName) -> String {
    let natural =
        i32::from(self.0) - i32::from(name.letter.semitone()) -
i32::from(name.accidental);
    format!("{name}{}", natural.div_euclid(12) - 1)
}
```

Para el Do central escrito Si♯, eso es 60 - 11 - 1 = 48, el número MIDI de Do3, así que el nombre es Si♯3. La comprobación de la conducción de voces y el entrenamiento auditivo escriben las notas así.

Escribir y leer un nombre

Escribir un nombre repite el signo tantas veces como dice la cuenta:

```
pub fn format(self, naming: NoteNaming) -> String {
    let sign = if self.accidental > 0 { "#" } else { "b" };
    let accidentals = sign.repeat(self.accidental.unsigned_abs() as
usize);
    match naming {
        NoteNaming::Letters => format!("{}{accidentals}",
self.letter.as_char()),
        NoteNaming::Solfege => format!("{}{accidentals}",
self.letter.solfege()),
    }
}
```

`unsigned_abs` convierte -2 en 2 como número sin signo, lo que nunca desborda (`abs` sobre un `i8` desbordaría con -128). Sin signos, `repeat(0)` da una cadena vacía.

Leer es el camino inverso, y lo que la aplicación escribe tiene que poder leerlo de vuelta. En Sol♯ menor armónica el acorde sobre la séptima nota es F𝄪°7, que se guarda como `F##dim7`, así que `read` toma hasta dos signos, los dos en la misma dirección. Lee un nombre al comienzo de un texto y devuelve el resto, así que `"F#m7"` da Fa♯ y `"m7"`:

```
let sign = |c| match c {
    '#' | '♯' => 1,
    'b' | '♭' => -1,
    _ => 0,
};
let mut accidental = 0;
for _ in 0..2 {
    let mut chars = rest.chars();
    let step = chars.next().map_or(0, sign);
    if step == 0 || (accidental != 0 && step != accidental) {
        break;
    }
    accidental += step;
    rest = chars.as_str();
}
```

- `sign` es un closure guardado en una variable. El tipo de su argumento no está escrito: el compilador lo deduce de cómo se usa el closure, con caracteres.
- `chars()` recorre el texto carácter a carácter, y `as_str()` devuelve lo que aún no se ha recorrido. Es la forma segura de pasar por encima de «♯», que ocupa tres bytes en UTF-8, donde cortar la cadena por una cuenta de bytes podría partir un carácter por la mitad.
- `map_or(0, sign)` da 0 cuando el texto se terminó, y el signo del carácter siguiente en otro caso.
- El bucle se detiene en un carácter que no es un signo, en un signo en la otra dirección, y después de dos signos. Así, `"Ebbb"` lee Mi𝄫 y deja `"b"`, y `"F#b"` lee

Fa♯ y deja `"b"`.

La letra tiene que ser mayúscula. `Letter::from_char` rechaza `'c'`, porque en un símbolo de acorde una «b» minúscula es un bemol: «Bb» es B♭ (Si♭), no dos letras B. La misma lectura en solfeo, «Sib» para Si♭, y cómo lee la aplicación un nombre escrito de cualquiera de las dos maneras, están en Dos idiomas: traducciones, y letras o solfeo.

Un conjunto de clases de altura en doce bits

Los acordes y las escalas son conjuntos de clases de altura. Un conjunto de doce cosas como mucho cabe en los bits de un entero: el bit 0 significa «Do está en el conjunto», el bit 1 «Do♯ está en el conjunto», y así hasta el bit 11 para Si.

```
pub struct PitchSet(u16);
```

```
pub fn insert(&mut self, pitch_class: PitchClass) {
    self.0 |= 1 << pitch_class.semitone();
}

pub fn contains(self, pitch_class: PitchClass) -> bool {
    self.0 & (1 << pitch_class.semitone()) != 0
}
```

`1 << n` es el número con solo el bit `n` encendido (1 desplazado `n` posiciones a la izquierda). `|=` enciende ese bit, y `&` deja solo los bits que comparten dos números, así que `x & (1 << n) != 0` pregunta si el bit `n` está encendido. Un acorde de Do mayor, Do, Mi y Sol, son los bits 0, 4 y 7:

```
bit:      11   10   9    8    7    6    5    4    3    2    1    0
          Si   La♯  La   Sol♯ Sol  Fa♯  Fa   Mi   Re♯  Re   Do♯  Do
Do mayor: 0    0    0    0    1    0    0    1    0    0    0    1
          = 0b0000_1001_0001
```

Otras preguntas sobre conjuntos son una instrucción cada una. `count_ones()` cuenta los miembros, y `a & !b == 0` pregunta si cada miembro de `a` está también en `b` (`!` invierte cada bit, así que `a & !b` es lo que `a` tiene y a `b` le falta):

```
pub fn is_subset_of(self, other: PitchSet) -> bool {
    self.0 & !other.0 == 0
}
```

Así pregunta el código si un acorde pertenece a una tonalidad: el La, Do y Mi de La menor están todos en la escala de Do mayor; el Sol♯ de Mi mayor no. Y como un conjunto es un número pequeño, comparar dos acordes es un solo `==`, y un conjunto puede ser la clave de un `HashMap`.

`iter` da los miembros desde Do hacia arriba, quedándose con las clases de altura que el conjunto contiene:

```
pub fn iter(self) -> impl Iterator<Item = PitchClass> {
    PitchClass::ALL
        .into_iter()
        .filter(move |&pitch_class| self.contains(pitch_class))
}
```

`move` le da al closure su propia copia de `self` (un `PitchSet` es `Copy`), porque el iterador se devuelve y vive más que la llamada.

Ver un acorde desde su fundamental

El truco para nombrar acordes es mover un conjunto entero hacia arriba o hacia abajo. En bits, eso es una **rotación** de los doce bits: los que pasan de Si vuelven a entrar por Do.

```
pub fn transpose(self, semitones: u8) -> PitchSet {
    let shift = (semitones % 12) as u32;
    let bits = self.0 as u32;
    // Rotate the 12 bits left by `shift`: bits that go past B come
    // back in at C.
    let rotated = (bits << shift) | (bits >> (12 - shift));
    PitchSet(rotated as u16 & ALL_BITS)
}
```

`bits << shift` sube cada miembro; los que pasaron del bit 11 los quita la máscara `ALL_BITS` (doce unos). `bits >> (12 - shift)` vuelve a meter exactamente esos por abajo. Do mayor subido 9 tiene La (bit 9), Do♯ (bit 13, que vuelve a entrar como bit 1) y Mi (bit 16, que vuelve a entrar como bit 4): La mayor.

`relative_to` baja un conjunto hasta que una fundamental dada se convierte en Do:

```
pub fn relative_to(self, root: PitchClass) -> PitchSet {
    self.transpose(12 - root.semitone())
}
```

Mi menor, Mi Sol Si, visto desde Mi es Do Re♯ Sol: los semitonos 0, 3 y 7, la forma de cualquier tríada menor. Visto desde su fundamental, cada acorde de un mismo tipo es el mismo número, sea cual sea su fundamental. Acordes: construirlos, nombrarlos e invertirlos nombra un acorde buscando la fundamental desde la que su conjunto coincide con una forma conocida.

Hacer que `collect` construya un `PitchSet`

`collect()` puede construir cualquier tipo que implemente el trait `FromIterator`; `Vec`, `String` y `HashMap` lo hacen. Implementarlo para `PitchSet` permite que cualquier código convierta notas en un conjunto en una línea, como `notes.map(|note| note.pitch_class()).collect::<PitchSet>()`:

```
impl FromIterator<PitchClass> for PitchSet {
    fn from_iter<I: IntoIterator<Item = PitchClass>>(iter: I) -> Self {
        let mut set = PitchSet::EMPTY;
        for pitch_class in iter {
            set.insert(pitch_class);
        }
        set
    }
}
```

`from_semitones(&[0, 4, 7])` construye un conjunto a partir de números de la misma manera, convirtiendo cada uno en una clase de altura y recogiéndolos.

En un archivo

Un archivo de proyecto guarda un conjunto como sus semitonos por encima de Do, por ejemplo `"0 3 6 9"`, que es más fácil de leer y de escribir a mano que el número 585. El atributo `#[serde(try_from = "String", into = "String")]` lo guarda a través de un `String` y lo lee a través de `TryFrom<String>`, que rechaza cualquier cosa que no sea un número de 0 a 11. El mismo par de conversiones, y por qué la de lectura es `try_from`, está en Series dodecafónicas y sus reglas.

Cómo se prueba

Los tests de `pitch.rs` y `pitch_set.rs` se leen como las reglas de este capítulo. Nombres y octavas, en los dos extremos del rango:

```
assert_eq!(Note::new(61).to_string(), "C#4");
assert_eq!(Note::new(61).name(Spelling::Flats), "Db4");
assert_eq!(Note::MIN.to_string(), "C-1");
assert_eq!(Note::MAX.to_string(), "G9");
```

Nombres con cualquier letra, y los nombres cromáticos:

```
assert_eq!(name(F, Letter::E), "E#");
assert_eq!(name(B, Letter::C), "Cb");
assert_eq!(name(G, Letter::F), "F##");
assert_eq!(name(A, Letter::B), "Bbb");
```

```
assert_eq!(chromatic.join(" "), "C C# D Eb E F F# G Ab A Bb B");
```

Intervalos contados en letras, y la octava de una nota escrita:

```
// A major third above A is C#, not Db.
assert_eq!(a.above(4, 2).to_string(), "C#");
```

```
assert_eq!(c4.name_as(NoteName::new(Letter::B, 1)), "B#3");
assert_eq!(Note::new(59).name_as(NoteName::new(Letter::C, -1)), "Cb4");
```

Y la lectura, incluido lo que no se debe leer:

```
assert_eq!(read("F##dim7"), Some(("F##".to_string(), "dim7")));
assert_eq!(read("Ebbb"), Some(("Ebb".to_string(), "b")), "two at most");
assert_eq!(read("F#b"), Some(("F#".to_string(), "b")), "not mixed");
assert_eq!(read("c"), None, "lower case is a flat, not a note");
```

Los tests de los conjuntos comprueban que transportar da la vuelta (Do mayor subido 9 es La mayor, subido 12 es él mismo) y que Mi menor visto desde Mi es el conjunto `0 3 7`.

Los ejercicios sobre estos tipos están en Ejercicios: la teoría musical en código.

Escalas, modos y tonalidades

Archivos: `crates/lineup-core/src/harmony/scale.rs`, `crates/lineup-core/src/key_map.rs`, `crates/lineup-core/src/tools.rs` y `crates/lineup-core/src/project.rs`

La tonalidad es lo que mira casi toda la aplicación antes de mostrar algo: el piano roll ilumina las filas de la tonalidad, las teclas de armonía tocan sus acordes, la hoja de acordes nombra los acordes en ella, la banda toca en ella, y el nombre de cada nota depende de ella. Este capítulo trata del tipo `Scale` que guarda una tonalidad, de cómo escribe una tonalidad cada una de sus notas, y de qué pasa cuando la canción cambia de tonalidad, entera o a mitad de camino.

La música, en pocas palabras

Una **escala** es una elección de algunas de las doce clases de altura, contadas hacia arriba desde una **tónica**. La escala mayor son las teclas blancas desde Do, Do Re Mi Fa Sol La Si: en semitonos por encima de la tónica, 0 2 4 5 7 9 11. El mismo dibujo empezado en otra nota da la escala mayor de esa nota: desde Re es Re Mi Fa♯ Sol La Si Do♯. Una escala sobre una tónica concreta, usada como casa de una pieza, es una **tonalidad**: Re mayor.

Las escalas que conoce Ferrous Lineup, en semitonos por encima de la tónica:

Escala	Semitonos
mayor	0 2 4 5 7 9 11
menor (natural)	0 2 3 5 7 8 10
menor armónica	0 2 3 5 7 8 11
menor melódica	0 2 3 5 7 9 11
dórico	0 2 3 5 7 9 10
frigio	0 1 3 5 7 8 10
lidio	0 2 4 6 7 9 11
mixolidio	0 2 4 5 7 9 10
locrio	0 1 3 5 6 8 10
pentatónica mayor	0 2 4 7 9
pentatónica menor	0 3 5 7 10

Escala	Semitonos
blues	0 3 5 6 7 10
cromática	las doce

La menor armónica sube la séptima nota de la escala menor para que se apoye en la tónica, un semitono más arriba; la menor melódica sube también la sexta. (La menor melódica clásica vuelve a la menor natural al bajar; la aplicación usa la forma ascendente en los dos sentidos, como hacen los músicos de jazz.) Los **modos**, de dórico a locrio, son la escala mayor empezada en otra de sus notas: Re dórico son las teclas blancas desde Re. Así que cada modo comparte sus notas, y su armadura, con una tonalidad mayor: su **relativa mayor**. La relativa mayor de La menor es Do mayor, tres semitonos más arriba.

La **armadura** son los sostenidos o bemoles que se escriben al principio de cada línea de una partitura. Cada tonalidad mayor tiene la suya, y subir una quinta añade un sostenido: Do mayor no tiene ninguno, Sol mayor uno, Re mayor dos. Bajar una quinta añade un bemol: Fa mayor tiene uno, Si♭ mayor dos. Ese círculo de tonalidades a una quinta de distancia es el **círculo de quintas**. El capítulo Tonalidades y modos del manual cuenta todo esto con sonidos para probar.

Clases de escala: un enum con datos

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub enum ScaleKind {
    Major,
    NaturalMinor,
    HarmonicMinor,
    MelodicMinor,
    Dorian,
    Phrygian,
    Lydian,
    Mixolydian,
    Locrian,
    MajorPentatonic,
    MinorPentatonic,
    Blues,
    Chromatic,
    Row(ToneRow),
    Set(PitchSet),
}
```

Las trece escalas de la música tonal son variantes simples. Las dos últimas llevan sus notas consigo: una serie dodecafónica y un conjunto de clases de altura también

pueden ser la «tonalidad» de una canción, para música que no tiene tonalidad en el sentido habitual. Tienen capítulos propios (Series dodecafónicas y sus reglas y Conjuntos de clases de altura, y una armonía con cuatro nombres); aquí aparecen sobre todo como los casos distintos. Un método los separa:

```
pub fn is_tonal(self) -> bool {
    !matches!(self, ScaleKind::Row(_) | ScaleKind::Set(_))
}
```

`Row(_)` encaja con un `Row` sea cual sea la serie que lleva.

Las notas de una escala se listan como semitonos por encima de la tónica. Para las escalas tonales esas listas no cambian nunca, así que están escritas dentro del programa y se prestan como `&'static [u8]`, slices que pueden tener cualquier longitud, cinco para una escala pentatónica, siete para la escala mayor:

```
fn tonal_intervals(self) -> &'static [u8] {
    match self {
        ScaleKind::Major => &[0, 2, 4, 5, 7, 9, 11],
        ScaleKind::NaturalMinor => &[0, 2, 3, 5, 7, 8, 10],
        // ...
        ScaleKind::Chromatic | ScaleKind::Row(_) | ScaleKind::Set(_) => {
            &[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
        }
    }
}
```

Una serie o un conjunto tiene sus propias notas, que solo existen dentro del valor, así que nada `'static` puede apuntar a ellas. Por eso el `intervals` público devuelve un `Vec<u8>` propio, hecho a partir de la serie, del conjunto o de la tabla:

```
pub fn intervals(self) -> Vec<u8> {
    match self {
        ScaleKind::Row(row) => row.semitones().to_vec(),
        ScaleKind::Set(set) => set.semitones(),
        kind => kind.tonal_intervals().to_vec(),
    }
}
```

El patrón de la última rama es un nombre simple, `kind`, que encaja con cualquier cosa y la liga, así que todas las escalas tonales van ahí.

Una tonalidad: una tónica y una clase

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub struct Scale {
    pub root: PitchClass,
    pub kind: ScaleKind,
    #[serde(default, skip_serializing_if = "std::ops::Not::not")]
    respelled: bool,
}
```

`respelled` es para las pocas tonalidades que se pueden escribir de dos maneras, más abajo. La mayoría de los métodos de `Scale` son cadenas cortas de iteradores. Las clases de altura de la escala son la tónica subida por cada intervalo:

```
pub fn pitch_classes(self) -> impl Iterator<Item = PitchClass> {
    self.kind
        .intervals()
        .into_iter()
        .map(move |interval| self.root.transpose(interval as i32))
}
```

`impl Iterator<Item = PitchClass>` dice «algún iterador de clases de altura» sin nombrar su tipo real, que incluye un closure, y los closures tienen tipos que nadie puede escribir. `into_iter()` sobre el `Vec` da sus números por valor, y `move` le da al closure su propia copia de `self` (un `Scale` es `Copy`), porque el iterador vive más que la llamada.

El **grado** de una nota es su lugar en la escala. `position` lo encuentra, contando desde 0, donde los músicos cuentan desde 1:

```
pub fn degree_of(self, pitch_class: PitchClass) -> Option<usize> {
    self.pitch_classes().position(|p| p == pitch_class)
}
```

En La menor, Mi es el grado 4 (los músicos dicen «el quinto»), y Fa♯ no tiene grado. `pitch_set` reúne las mismas clases de altura en un `PitchSet`, así que el test de que cada modo comparte sus notas con su relativa mayor es una comparación de tres números:

```
let c_major = Scale::default().pitch_set();
assert_eq!(Scale::new(D, ScaleKind::Dorian).pitch_set(), c_major);
assert_eq!(Scale::new(G, ScaleKind::Mixolydian).pitch_set(), c_major);
assert_eq!(Scale::new(A, ScaleKind::NaturalMinor).pitch_set(), c_major);
```

`Scale::default()` es Do mayor, e `is_heptatonic` dice si una escala tiene siete notas. Los acordes apilados por terceras y los números romanos solo tienen sentido en esas, así que muchos métodos empiezan preguntándolo (Acordes: construirlos, nombrarlos e invertirlos).

Sostenidos o bemoles: cómo se escribe una tonalidad

Cada tonalidad menor y cada modo comparte su armadura con su relativa mayor, así que la pregunta «¿sostenidos o bemoles?» la responde esa tonalidad mayor. `relative_major` dice cuánto más arriba está:

```
pub(crate) fn relative_major(self) -> u8 {
    match self {
        ScaleKind::Major
        | ScaleKind::MajorPentatonic
        | ScaleKind::Chromatic
        | ScaleKind::Row(_)
        | ScaleKind::Set(_) => 0,
        ScaleKind::NaturalMinor
        | ScaleKind::HarmonicMinor
        | ScaleKind::MelodicMinor
        | ScaleKind::MinorPentatonic
        | ScaleKind::Blues => 3,
        ScaleKind::Dorian => 10,
        ScaleKind::Phrygian => 8,
        ScaleKind::Lydian => 7,
        ScaleKind::Mixolydian => 5,
        ScaleKind::Locrian => 1,
    }
}
```

Re dórico son las teclas blancas desde Re, y Do está diez semitonos por encima de Re; el Do de Mi frigio está ocho por encima de Mi, y así sucesivamente. `pub(crate)` hace que el método se vea solo dentro de `lineup-core`: el resto del programa hace las preguntas construidas sobre él.

La escritura habitual depende entonces de esa tonalidad mayor. Las tonalidades cuya armadura tiene bemoles son Fa, Si♭, Mi♭, La♭ y Re♭ mayor, y sus menores y modos:

```
fn usual_spelling(self) -> Spelling {
    use PitchClass::*;
    match self.root.transpose(self.kind.relative_major() as i32) {
        F | ASharp | DSharp | GSharp | CSharp => Spelling::Flats,
        // F# major and Gb major are the same keys of the piano, and
        // the tonic's name decides between them. Their modes are
        // named with sharps (G# Dorian, B Lydian), except F Locrian,
        // whose notes are those of Gb major: F Gb Ab Bb Cb Db Eb.
        FSharp if self.root == F => Spelling::Flats,
        _ => Spelling::Sharps,
    }
}
```

La rama del medio tiene una **guarda de match**, `if self.root == F`: la rama se toma solo si el patrón encaja y la condición se cumple. Fa locrio es el único modo sobre esas teclas cuya tónica es una tecla blanca con nombre propio; escrito con sostenidos sería Mi♯ locrio, así que toma los bemoles de Sol♭ mayor.

La armadura, alrededor del círculo de quintas

Una armadura es un número: cuántos sostenidos, o cuántos bemoles como número negativo. Dando la vuelta al círculo de quintas, la tonalidad mayor que está `n` quintas por encima de Do tiene `n` sostenidos. `key_signature` encuentra `n` probando cada paso del círculo hasta caer en la relativa mayor:

```
pub fn key_signature(self) -> Option<(i8, bool)> {
    if !self.kind.is_tonal() || self.kind == ScaleKind::Chromatic {
        return None;
    }
    let major = self.root.transpose(self.kind.relative_major() as i32);
    // Going round the circle of fifths: each fifth up adds a sharp.
    let fifths = (0..12i8)
        .find(|&n| PitchClass::C.transpose(7 * i32::from(n)) == major)
        .unwrap_or(0);
    let sharps = match self.spelling() {
        Spelling::Flats => fifths - 12,
        Spelling::Sharps => fifths,
    };
    let minor = self.kind.relative_major() == 3;
    Some((sharps, minor))
}
```

Doce quintas hacia arriba vuelven a Do, así que una tonalidad con bemoles es el mismo número de quintas contado al revés: Fa mayor está once quintas arriba, que es una quinta abajo, así que 11 - 12 = -1, un bemol. Re♭ mayor está siete quintas arriba y cinco abajo: cinco bemoles. El segundo valor dice si la tonalidad es menor,

algo que también guarda el evento de armadura de un archivo MIDI (MIDI: mensajes, dispositivos, grabación y archivos). Una serie, un conjunto o la escala cromática no tienen armadura, de ahí el `Option`.

El rango `0..12i8` lleva su tipo escrito en el número, `12i8`, así que `n` es un `i8` como el resultado. `find` se detiene en el primer `n` que sirve; siempre hay uno, y `unwrap_or(0)` solo evita que el compilador pregunte qué pasa si no.

Tonalidades que se pueden escribir de dos maneras

Las tonalidades de seis sostenidos y de seis bemoles son las mismas teclas del piano: Fa♯ mayor y Sol♭ mayor. A su alrededor, Do♯ mayor (siete sostenidos) es Re♭ mayor (cinco bemoles), y Si mayor (cinco sostenidos) es Do♭ mayor (siete bemoles). Los compositores eligen el nombre que va con la música que la rodea; Debussy escribe el centro de *Golliwogg's Cakewalk* en Sol♭ mayor, una tonalidad de bemoles como el Mi♭ mayor del que viene. Cualquier otra tonalidad necesitaría ocho signos o más escrita al revés: Re♯ mayor necesita nueve sostenidos, así que siempre es Mi♭ mayor.

Así que un `Scale` puede estar **reescrito**: las mismas notas, el otro nombre. `other_spelling` da esa tonalidad cuando existe:

```
pub fn other_spelling(self) -> Option<Scale> {
    use PitchClass::*;
    let key = self.as_key();
    let major = key.root.transpose(key.kind.relative_major() as i32);
    let either_way = key.kind.is_tonal()
        && key.kind != ScaleKind::Chromatic
        && matches!(major, FSharp | CSharp | B);
    either_way.then_some(Scale {
        respelled: !self.respelled,
        ..self
    })
}
```

`Scale { respelled: !self.respelled, ..self }` es la **sintaxis de actualización de structs**: una copia de `self` con un campo cambiado. `spelling()` invierte la respuesta habitual cuando la tonalidad está reescrita, y la tónica toma la letra siguiente: Fa♯ pasa a ser Sol♭, Si pasa a ser Do♭.

El campo es privado, así que las únicas formas de cambiarlo son los métodos, que comprueban que la tonalidad se puede escribir al revés. `Scale::written(tonic, kind)` construye una tonalidad a partir de cómo se escribe su tónica, que es lo que usan los menús **Tonalidad** y la línea de la tonalidad de una hoja de acordes escrita como texto: Sol♭ con «mayor» da Fa♯ mayor reescrito, y Sol♭ con «menor», una

tonalidad que necesitaría un doble bemol en su armadura, da el Fa♯ menor de siempre.

En un archivo, el atributo `#[serde(default, skip_serializing_if = "std::ops::Not::not")]` deja fuera el campo cuando es `false`, que es casi siempre, y lee uno que falta como `false`. `skip_serializing_if` recibe la ruta de una función que toma `&bool` y devuelve `true` para saltarlo; `std::ops::Not::not` es el operador `!` como función, así que el campo se salta cuando `!respelled`. Un archivo guardado con una tonalidad reescrita y abierto por una versión de la aplicación que no conoce el campo recibe la tonalidad con su nombre habitual, con las mismas notas.

Una letra para cada nota

`spelling()` solo dice cómo se escriben las teclas negras de la tónica. Las notas de una tonalidad necesitan más: Re menor armónica tiene un Si♭ y un Do♯, y Fa♯ mayor tiene un Mi♯, que no es ni el nombre sostenido ni el bemol de ninguna tecla negra. La regla es la de Clases de altura, notas y su escritura: una escala de siete notas usa cada letra una vez. `spell` le da a una clase de altura la letra de su lugar en la escala, contada hacia arriba desde la letra de la tónica, y deja que `NoteName::with_letter` encuentre los sostenidos o bemoles:

```
pub fn spell(self, pitch_class: PitchClass) -> NoteName {
    if let Some(scale) = self.set_as_scale() {
        return scale.spell(pitch_class);
    }
    if !self.kind.is_tonal() {
        return NoteName::chromatic(pitch_class);
    }
    let name = NoteName::with_letter(
        pitch_class,
        self.tonic_name()
            .letter
            .up(i32::from(self.letters_above_tonic(pitch_class))),
    );
    if self.has_letter_for(pitch_class) {
        name
    } else {
        name.simplified()
    }
}
```

Cuántas letras por encima de la tónica va una nota es su lugar en la escala para una escala de siete notas. Las escalas pentatónicas y la de blues se saltan letras, así que tienen una tabla propia:

```
fn letter_steps(self) -> &'static [u8] {
    match self {
        ScaleKind::MajorPentatonic => &[0, 1, 2, 4, 5],
        ScaleKind::MinorPentatonic => &[0, 2, 3, 4, 6],
        ScaleKind::Blues => &[0, 2, 3, 4, 4, 6],
        ScaleKind::Chromatic | ScaleKind::Row(_) | ScaleKind::Set(_) =>
&CHROMATIC_LETTERS,
        _ => &[0, 1, 2, 3, 4, 5, 6],
    }
}
```

Las dos entradas de la cuarta letra de la escala de blues son su «blue note», una quinta bemol junto a la quinta: La blues es La Do Re Mi♭ Mi Sol, no La Do Re Re♯ Mi Sol.

Notas de fuera de la tonalidad

Una nota de fuera de la escala no tiene lugar en ella. Se escribe como más a menudo la encuentran los músicos, contada desde la tónica: una segunda, tercera, sexta o séptima bemol, una cuarta sostenida, y la sexta y la séptima mayores, que en una tonalidad menor son las notas subidas de la menor melódica y la armónica. Eso es una tabla de letras según los semitonos por encima de la tónica:

```
const CHROMATIC_LETTERS: [u8; 12] = [0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6];
```

Un semitono por encima de la tónica es una letra más arriba (una segunda bemol), seis semitonos son tres letras (una cuarta sostenida), once son seis letras (la séptima mayor). Así, en Do mayor las teclas negras de fuera de la tonalidad son Re♭, Mi♭, Fa♯, La♭ y Si♭, y en Re menor Do♯ es Do♯.

`letters_above_tonic` elige entre el lugar en la escala y esta tabla:

```
fn letters_above_tonic(self, pitch_class: PitchClass) -> u8 {
    let semitones = self.root.semitones_up_to(pitch_class);
    if self.has_letter_for(pitch_class)
        && let Some(place) = self
            .kind
            .tonal_intervals()
            .iter()
            .position(|&interval| interval == semitones)
    {
        return self.kind.letter_steps()[place];
    }
    CHROMATIC_LETTERS[semitones as usize]
}
```

El `if` une una condición y un `let` con `&&`. Es una **cadena de `let`** (*let chain*): el bloque se ejecuta solo si la condición es verdadera y el patrón encaja, y `place` queda ligado dentro. Sin ella, el código necesitaría un `if` dentro de otro `if`, o un `match` sobre una tupla.

La tabla puede pedir dos signos. En Re♭ mayor la nota un semitono por encima de la tónica, Re, es una segunda bemol, la letra de Mi: Mi𝄫. Ningún músico escribe eso para una nota de fuera de la tonalidad, así que `spell` toma el nombre más sencillo, Re, para cualquier nota a la que la escala no da letra. Dentro de la escala los dos signos se quedan, porque la tonalidad los necesita: la séptima nota de Sol♯ menor armónica es Fa𝄪.

Un conjunto de clases de altura cuyas notas son exactamente una escala tonal sobre su tónica se escribe como esa escala (`set_as_scale`), así que Mi mayor mostrado como conjunto conserva su Sol♯, Do♯ y Re♯. Una serie o cualquier otro conjunto no tiene letras, así que sus notas toman los nombres cromáticos.

Nombres con octava

`note_name` junta el nombre escrito con el número de octava correcto, a través de `Note::name_as`:

```
pub fn note_name(self, note: Note) -> String {
    note.name_as(self.spell(note.pitch_class()))
}
```

Do♯ menor armónica sube su séptima, Si, a Si♯, así que en esa tonalidad la nota MIDI 60, el Do central, es «Si♯3». La comprobación de la conducción de voces y la vista del movimiento de las partes nombran así las notas de las que hablan (Conducción de voces y movimiento de las partes).

Cambiar de tonalidad: mover la canción

Los menús **Tonalidad**, a la izquierda de las teclas de armonía y arriba en el piano roll, fijan la tonalidad de toda la canción, la que usan las teclas de armonía, las sugerencias y la banda. Así que también mueven la canción, como una banda toca la misma canción en otra tonalidad cuando se lo piden: elige Re y los acordes, el bajo y la melodía suben un tono; elige menor y cada nota pasa al mismo lugar de la escala menor. La batería se queda como está, y **Ctrl+Z** lo devuelve todo de una vez.

Como con cualquier herramienta de edición, la parte musical es una función simple que no cambia nada por sí misma. Responde una pregunta: ¿adónde va esta nota cuando la tonalidad cambia de `from` a `to`?

```
pub fn change_key(note: Note, from: Scale, to: Scale) -> Note {
    let from = from_key(from, to);
    if !from.kind.notes_follow(to.kind) {
        return note;
    }
    let up = from.root.semitones_up_to(to.root) as i32;
    let shift = if up > 6 { up - 12 } else { up };
    let Some(moved) = note.transpose(shift) else {
        return note;
    };
    // The chromatic scale holds every note, so it has no degrees to
    // match with a key's: into it or out of it, notes only take the new
    // root. (Out of it, every note would otherwise be "of the old scale"
    // and be pulled into the new key, losing every sharp and flat.)
    if from.kind == ScaleKind::Chromatic || to.kind ==
ScaleKind::Chromatic {
        return moved;
    }
    // The old scale moved to the new root, so `moved` can be compared
    // with it degree by degree.
    let old = Scale::new(to.root, from.kind);
    let Some(degree) = old.degree_of(moved.pitch_class()) else {
        return moved;
    };
    let (old_steps, new_steps) = (from.kind.intervals(),
to.kind.intervals());
    if old_steps.len() == new_steps.len() {
        let offset = new_steps[degree] as i32 - old_steps[degree] as i32;
        moved.transpose(offset).unwrap_or(moved)
    } else {
        nearest_in_key(moved, to)
    }
}
```

Trabaja en dos movimientos.

1. **Una tónica nueva mueve cada nota la misma distancia.** De Do a Re son dos semitonos hacia arriba. De Do a Sol podrían ser siete hacia arriba o cinco hacia abajo; `shift` toma el más cercano, así que la canción se queda más o menos donde estaba en el teclado en lugar de saltar una quinta hacia arriba.

2. **Un modo nuevo lleva cada nota al mismo grado.** Después del primer movimiento, la nota se compara con la clase de escala *vieja* sobre la tónica *nueva*. Si es la tercera nota de Re mayor (Fa♯), pasa a ser la tercera nota de Re menor (Fa). La diferencia entre las dos listas de intervalos en el mismo lugar es cuánto se mueve ese grado:

grado	1	2	3	4	5	6	7
mayor	0	2	4	5	7	9	11
menor natural	0	2	3	5	7	8	10
se mueve	0	0	-1	0	0	-1	-1

Así, de Do mayor a Do menor bajan Mi, La y Si, y quedan Do, Re, Fa y Sol.

Algunas notas necesitan una regla propia:

- Una **nota de fuera de la escala vieja**, como una nota de un acorde prestado o de una parte escrita en otra tonalidad, no tiene grado, así que el `let ... else` la devuelve solo con el primer movimiento. Conserva su distancia a la tónica, y una modulación sigue siendo una modulación.
- **Escalas de tamaños distintos**, como la mayor (siete notas) y la pentatónica mayor (cinco), no tienen grados que se correspondan. Una nota que no está en la escala nueva va a la más cercana que sí está, con `nearest_in_key` (Herramientas de edición como funciones simples).
- La **escala cromática** tiene todas las notas, así que entrar en ella o salir de ella solo mueve la tónica.
- Entre una tonalidad y una **serie** o un **conjunto** no hay con qué mover, y `notes_follow` lo dice: solo cambia lo que ilumina el piano roll. Un conjunto cuyas notas son exactamente una tonalidad, como Do mayor mostrado como el conjunto 7-35, sigue moviéndose como esa tonalidad (`from_key`).

`let Some(moved) = ... else { return note; };` es un **let-else**: si el patrón no encaja, el bloque `else` tiene que salir de la función. Aquí cubre «la nota se saldría del rango MIDI» y «la nota no está en la escala».

El proyecto entero

`Project::change_key` pasa esa función por cada nota que debe moverse. Su bucle central:

```
let pitched: Vec<bool> =
self.channels.iter().map(Channel::is_pitched).collect();
for (index, pattern) in self.patterns.iter_mut().enumerate() {
    let (old, new) = (before[index], after[index]);
    for (channel, _) in pitched.iter().enumerate().filter(|(_, p)| **p) {
        if pattern.is_step_row(channel) {
            continue;
        }
        let notes: Vec<NoteEvent> = pattern
            .notes(channel)
            .iter()
            .map(|note| NoteEvent {
                pitch: tools::change_key(note.pitch, old, new),
                ..*note
            })
            .collect();
        pattern.set_notes(channel, notes);
    }
}
```

- `pitched` se recoge antes del bucle porque el bucle presta `self.patterns` de forma mutable, y Rust no deja que el código mire `self.channels` a través de `self` al mismo tiempo. Un `Vec<bool>` pequeño leído antes termina ese préstamo antes.
- `Channel::is_pitched` es falso para la batería incorporada y para un canal MIDI Out en el canal de batería de General MIDI: la batería no se mueve.
- Una fila de pasos del secuenciador (cada nota un Do4 de un paso) es un sonido tocado a tiempo, no una melodía, así que se queda como está.
- `before[index]` y `after[index]` son las tonalidades en que se toca cada patrón antes y después del cambio, que no siempre son la tonalidad de la canción, como muestra la sección siguiente.

La aplicación la llama a través de `edit`, que guarda el proyecto como estaba antes (Deshacer y rehacer), así que un solo **Ctrl+Z** devuelve la tonalidad y todas las notas:

```
self.edit("change key", |project| project.change_key(scale));
```

La hoja de acordes y las partes de la banda se mueven con las notas; están en La hoja de acordes: modelo, distribución y texto y Cómo toca la banda una hoja de acordes.

Cambios de tonalidad a mitad de camino: marcas de tonalidad

Muchas piezas no se quedan en una tonalidad. El Rondó alla turca de Mozart empieza en La menor, pasa a La mayor en el compás 25, donde la armadura gana tres sostenidos, y vuelve a La menor en el compás 65. Pasar a una tonalidad nueva es una **modulación**. Tres sostenidos son también la armadura de Fa♯ menor, la relativa menor de La mayor, y la parte en La mayor pasa por Fa♯ menor sin cambiar de armadura; en qué tonalidad está la música se oye por dónde vuelve a casa, no por la armadura.

Un cambio de tonalidad es una marca más en la lista de reproducción, junto a las marcas de compás y de tempo:

```
pub struct KeyMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub scale: Scale,
}
```

Una marca de tonalidad nunca mueve una nota. Cambia cómo se llama la tonalidad, qué filas ilumina el piano roll, qué acordes ofrecen las teclas de armonía y qué acordes cuentan como «de la tonalidad». Esa es la diferencia con los menús Tonalidad, que mueven la canción.

`KeyMap` responde «¿en qué tonalidad está el paso 300?»: la tonalidad propia de la canción hasta la primera marca, y luego la de cada marca por turno. Al construirlo, ordena las marcas y se queda con una por paso:

```
pub fn new(first: Scale, markers: impl IntoIterator<Item = KeyMarker>) ->
Self {
    let mut markers: Vec<KeyMarker> = markers
        .into_iter()
        .filter(|marker| marker.start > 0)
        .collect();
    markers.sort_by_key(|marker| marker.start);
    markers.dedup_by_key(|marker| marker.start);
    KeyMap { first, markers }
}
```

`dedup_by_key` quita un elemento cuando tiene la misma clave que el anterior, así que después de ordenar se queda con la primera marca de cada paso. Buscar un paso toma la última marca que empieza en él o antes:

```
pub fn marker_at(&self, step: usize) -> Option<&KeyMarker> {
    self.markers
        .iter()
        .take_while(|marker| marker.start <= step)
        .last()
}
```

`take_while` se detiene en la primera marca posterior a `step`, y `last` da la anterior, o `None` si el paso está antes de todas las marcas. Un `Project` construye su mapa con `key_map()`, y `key_at(step)` le pregunta.

Añadir una marca mantiene la lista ordenada con `binary_search_by_key`, que da `Ok(index)` si ya hay una marca que empieza en ese paso y `Err(index)`, donde iría, si no:

```
let markers = &mut self.playlist.key_markers;
match markers.binary_search_by_key(&marker.start, |m| m.start) {
    Ok(index) => markers[index] = marker,
    Err(index) => markers.insert(index, marker),
}
```

Una marca en el paso 0 no se añade: cambia la tonalidad con la que empieza la canción.

La tonalidad de un patrón

Un patrón se escribe una vez y se toca dondequiera que estén sus clips, y una canción puede tocarlo en tonalidades distintas. El piano roll necesita una tonalidad para iluminar sus notas, así que un patrón toma la tonalidad donde empieza su primer clip:

```
pub fn pattern_key_at(&self, pattern: usize, step: usize) -> Scale {
    self.playlist
        .clips
        .iter()
        .filter(|clip| clip.pattern == pattern)
        .map(|clip| clip.start)
        .min()
        .map_or(self.scale, |start| self.key_at(start + step))
}
```

`pattern_key(p)` es esto en el paso 0. El piano roll dice *Este patrón: La mayor* junto a los menús Tonalidad, y las teclas de armonía dicen *Ahora en La mayor* cuando el cursor de reproducción está en una tonalidad que no es la propia de la canción.

Cuando la canción se mueve, las marcas se mueven

Cuando los menús Tonalidad mueven una canción con cambios de tonalidad, cada cambio tiene que moverse el mismo intervalo, o la marcha del Rondó se quedaría en La mayor mientras el resto iba a Do menor. `moved_key` mueve una tonalidad como se movió la de la canción:

```
pub fn moved_key(key: Scale, from: Scale, to: Scale) -> Scale {
    let from = crate::tools::from_key(from, to);
    if !from.kind.notes_follow(to.kind) {
        return key;
    }
    let up = from.root.semitones_up_to(to.root);
    let root = key.root.transpose(up as i32);
    let kind = if key.kind == from.kind && from.kind.is_tonal() &&
to.kind.is_tonal() {
        to.kind
    } else {
        key.kind
    };
    if root == key.root && kind == key.kind {
        return key;
    }
    Scale::new(root, kind)
}
```

La tónica se mueve los mismos semitonos. Una tonalidad en el modo viejo de la canción toma el modo nuevo: una canción en Do mayor con un pasaje en Sol mayor, llevada a Do dórico, tiene ese pasaje en Sol dórico. Una tonalidad en otro modo conserva el suyo: en una canción en La menor subida una tercera menor a Do menor, la marcha en La mayor pasa a Do mayor y el episodio en Fa♯ menor a La menor, como se leería una partitura transportada. Después `change_key` mueve cada patrón de la tonalidad en que se tocaba a la tonalidad en que se toca ahora, que es lo que guardaban `before` y `after` más arriba.

Los archivos MIDI también guardan los cambios de tonalidad, como eventos de armadura; ver MIDI: mensajes, dispositivos, grabación y archivos. Una marca también la puede escribir la propia aplicación, a partir de las tonalidades que oye en las notas (Oír la armonía: acordes y tonalidades a partir de las notas).

Ajustar a la escala

El botón **Ajustar a la escala** del piano roll hace que una nota que dibujas o arrastras caiga en la nota más cercana de la tonalidad, y su menú de clic derecho añade

Ajustar las notas que tocas, para el teclado del equipo y un teclado MIDI. Los dos se reducen a `tools::nearest_in_key(note, scale)`, en Herramientas de edición como funciones simples; la tonalidad a la que se ajusta es la de ese punto de la canción, después de las marcas de tonalidad. Cómo se ajustan las teclas de un teclado MIDI en el hilo de MIDI está en MIDI: mensajes, dispositivos, grabación y archivos.

Cómo se prueba

Los tests de `scale.rs` enuncian las reglas para todas las tonalidades a la vez, no solo para las que se le ocurrieron a alguien. Cada escala de siete notas, sobre cada tónica, escrita de cualquiera de las dos maneras, usa siete letras distintas, y cada nombre sigue nombrando la tecla correcta del piano:

```
let mut letters: Vec<_> = scale
    .pitch_classes()
    .map(|p| scale.spell(p).letter)
    .collect();
letters.sort();
letters.dedup();
assert_eq!(letters.len(), 7, "{}", scale.name());
```

Los signos con que se escriben las notas de una tonalidad suman su armadura, para cada modo sobre cada tónica, así que un pentagrama nunca necesita una alteración para una nota de la tonalidad:

```
let (signature, _) = scale.key_signature().unwrap();
let written: i8 = scale
    .pitch_classes()
    .map(|note| scale.spell(note).accidental)
    .sum();
assert_eq!(written, signature, "{}", scale.name());
```

Y las armaduras de las doce tonalidades mayores, desde Do hacia arriba, son el círculo de quintas:

```
// Sharps count up, flats count down, round the circle of fifths:
// C, Db, D, Eb, E, F, F#, G, Ab, A, Bb, B.
let major = [0, -5, 2, -3, 4, -1, 6, 1, -4, 3, -2, 5];
```

`tools.rs` comprueba los movimientos de un cambio de tonalidad: de Do a Sol baja una cuarta, de Do mayor a Do menor bajan Mi, La y Si, una nota de fuera de la escala solo toma la tónica nueva, y un D7 llevado a la escala cromática y de vuelta sigue

siendo un D7. `key_map.rs` comprueba que la marcha y el episodio del Rondó caen en Do mayor y La menor cuando la canción sube una tercera menor.

Los ejercicios sobre tonalidades están en Ejercicios: la teoría musical en código.

Acordes: construirlos, nombrarlos e invertirlos

Archivos: `crates/lineup-core/src/harmony/chord.rs`, `crates/lineup-core/src/harmony/symbol.rs`, `crates/lineup-core/src/harmony/analysis.rs` y `crates/lineup-core/src/harmony/scale.rs`

Los acordes están por todas partes en Ferrous Lineup: el piano roll los coloca con un clic y nombra lo que esté sonando, la hoja de acordes los lee a medida que los escribes, las teclas de armonía los tocan y la banda los convierte en partes. Este capítulo trata de los tipos que hay detrás de todo eso: la clase de un acorde, un acorde, un símbolo de acorde como lo escribe una partitura guía, y el análisis que nombra un grupo de notas.

La música, en pocas palabras

Un **acorde** son varias notas tocadas juntas. La mayoría de los acordes se construyen con **terceras**: una tercera mayor son cuatro semitonos, una tercera menor tres.

Acorde	Desde la fundamental	Sobre Do	Suena
mayor	tercera mayor + tercera menor (0 4 7)	Do Mi Sol	luminoso, estable
menor	tercera menor + tercera mayor (0 3 7)	Do Mi♭ Sol	más oscuro
disminuido	dos terceras menores (0 3 6)	Do Mi♭ Sol♭	tenso
aumentado	dos terceras mayores (0 4 8)	Do Mi Sol♯	inestable, flotante
séptima de dominante	tríada mayor + tercera menor (0 4 7 10)	Do Mi Sol Si♭	quiere seguir
séptima mayor	tríada mayor + tercera mayor (0 4 7 11)	Do Mi Sol Si	suave, jazzero
menor séptima	tríada menor + tercera menor (0 3 7 10)	Do Mi♭ Sol Si♭	cálido

Seguir subiendo por terceras más allá de la octava da los acordes **extendidos** del jazz, el soul y el funk. Sus nombres cuentan las notas de la escala desde la fundamental: la novena es la segunda una octava más arriba (14 = 12 + 2 semitonos), la oncena la cuarta (17 = 12

- 5), la decimotercera la sexta (21 = 12 + 9). C13 es Do Mi Sol Si♭ Re La.

La nota más grave es la **fundamental** cuando el acorde está en **estado fundamental**. Tocado como Mi Sol Do, con Mi abajo, sigue siendo Do mayor, en su primera **inversión**. Un **símbolo de acorde**, como lo escribe una partitura guía, pone el bajo después de una barra: C/E. El capítulo Los acordes y sus nombres del manual lo tiene todo, con qué escuchar.

Clases de acorde: una tabla en un `match`

Un acorde sin su fundamental es una **clase** de acorde: «menor séptima», empiece en la nota que empiece. Hay treinta, una variante del enum cada una:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum ChordQuality {
    Major,
    Minor,
    Diminished,
    Augmented,
    Sus2,
    Sus4,
    Major6,
    Minor6,
    Dominant7,
    Major7,
    Minor7,
    // ...
    Dominant13,
    Major13,
}
```

Todo lo que Ferrous Lineup sabe de una clase es un `match` sobre ella, un método por dato. Las notas, en semitonos por encima de la fundamental:

```
pub fn intervals(self) -> &'static [u8] {
    use ChordQuality::*;
    match self {
        Major => &[0, 4, 7],
        Minor => &[0, 3, 7],
        Diminished => &[0, 3, 6],
        // ...
        Dominant9 => &[0, 4, 7, 10, 14],
        // ...
        // Players leave the eleventh out of thirteenth chords, because
        // it clashes with the third; so does Ferrous Lineup.
        Dominant13 => &[0, 4, 7, 10, 14, 21],
        Major13 => &[0, 4, 7, 11, 14, 21],
    }
}
```

La novena se escribe 14, no 2. Como clase de altura es lo mismo, pero un acorde colocado en el piano roll se construye con estos números, y una novena un tono por encima de la fundamental sonaría a racimo, no a acorde de novena.

Los otros datos son métodos de la misma forma: `symbol` (el «m7» de «Am7»), `name` («minor seventh»), `tones` (qué nota del acorde es cada intervalo, más abajo), `is_minor`, `is_dominant` y `numeral_suffix` (el «°» de «vii°»). Escribir el conocimiento como ramas de `match` tiene una gran ventaja frente a, por ejemplo, una lista de structs: añade una variante y el compilador nombra cada método que todavía no la trata.

`numeral_suffix` termina con un **patrón que liga**:

```
match self {
    Major | Minor => "",
    Diminished => "°",
    // ...
    other => other.symbol(),
}
```

`other` encaja con todo lo que queda, como `_`, pero le da un nombre para usarlo. `family`, que clasifica una clase en los tres tipos de tríada, añade una guarda a uno:

```
pub fn family(self) -> ChordFamily {
    use ChordQuality::*;
    match self {
        Diminished | HalfDiminished7 | Diminished7 =>
ChordFamily::Diminished,
        quality if quality.is_minor() => ChordFamily::Minor,
        _ => ChordFamily::Major,
    }
}
```

Las tonalidades usan la familia para emparejar un acorde con sus propios acordes: G7 es el acorde mayor sobre Sol, Bm7b5 el disminuido sobre Si (Números romanos, las teclas de armonía y las funciones de los acordes).

¿Qué clase es esta?

Las notas de una clase sobre Do son un `PitchSet`, y encontrar la clase de un grupo de notas visto desde su fundamental es una búsqueda:

```
pub fn identify(relative: PitchSet) -> Option<ChordQuality> {
    Self::ALL
        .into_iter()
        .find(|quality| quality.pitch_set() == relative)
}
```

`find` devuelve la primera clase que encaja. Así que el orden de `ChordQuality::ALL` importa, y pone primero los acordes más sencillos y comunes. Para que `identify` encuentre todas las clases, no puede haber dos con las mismas notas, y un test se asegura de ello:

```
#[test]
fn no_two_qualities_have_the_same_notes() {
    // Otherwise `identify` could never find the second one.
    for (i, a) in ChordQuality::ALL.iter().enumerate() {
        for b in &ChordQuality::ALL[i + 1..] {
            assert_ne!(a.pitch_set(), b.pitch_set(), "{a:?} and {b:?}");
        }
    }
}
```

`&ChordQuality::ALL[i + 1..]` es el slice del array desde `i + 1` hasta el final, así que cada par se compara una vez.

Un acorde: una fundamental y una clase

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct Chord {
    pub root: PitchClass,
    pub quality: ChordQuality,
}
```

Sus clases de altura son las de la clase subidas hasta la fundamental, y sus notas se apilan desde una nota fundamental dada, dejando fuera las que se saldrían por arriba de MIDI:

```
pub fn pitch_set(self) -> PitchSet {
    self.quality.pitch_set().transpose(self.root.semitone())
}

pub fn notes_from(self, root_note: Note) -> Vec<Note> {
    self.quality
        .intervals()
        .iter()
        .filter_map(|&interval| root_note.transpose(interval as i32))
        .collect()
}
```

Los acordes de una tonalidad: apilar terceras

Para construir el acorde sobre una nota de una tonalidad, se toma una nota de la escala sí y otra no: la nota, la que está dos lugares más arriba, la que está cuatro (y seis para un acorde de séptima). Eso es **apilar terceras**. En Do mayor, empezar en Re da Re, Fa y La: Re menor.

```
pub fn stack_thirds(self, note: Note, count: usize) -> Option<Vec<Note>> {
    if !self.is_heptatonic() {
        return None;
    }
    let intervals = self.kind.tonal_intervals();
    let degree = self.degree_of(note.pitch_class())?;
    let notes = (0..count)
        .map(|i| {
            let index = degree + 2 * i;
            // Going past the seventh note wraps into the next octave.
            let octave = (index / 7) as i32;
            let above_root = intervals[index % 7] as i32 + 12 * octave;
            above_root - intervals[degree] as i32
        })
        .filter_map(|semitones| note.transpose(semitones))
        .collect();
    Some(notes)
}
```

Se lee como una cadena: los números `0..count` se vuelven lugares de la escala, los lugares se vuelven semitonos por encima de `note`, y `filter_map` convierte los semitonos en notas, descartando las que saldrían del rango MIDI. El `?` después de `degree_of` devuelve `None` enseguida cuando la nota no está en la escala.

`Scale::triad(degree)` y `seventh(degree)` apilan tres o cuatro notas sobre un grado y nombran el resultado con `identify`. Para los siete grados de Do mayor, es la tabla que aprende todo compositor de canciones:

Grado	1	2	3	4	5	6	7
Tríada	C	Dm	Em	F	G	Am	Bdim
Número	I	ii	iii	IV	V	vi	vii°
Séptima	Cmaj7	Dm7	Em7	Fmaj7	G7	Am7	Bm7b5

Cambia la tonalidad y la tabla la sigue, porque se calcula, no está escrita. `ninth(degree)` apila cinco notas, y devuelve `None` en los grados donde la novena queda solo un semitono por encima de la fundamental una octava más arriba (Mi Sol Si Re Fa en Do mayor), que choca, y ningún acorde con nombre encaja.

Sellos de acorde

Lo que coloca un clic en el piano roll es un **sello**, un enum cuyas variantes llevan lo que necesitan:

```
pub fn notes_at(self, note: Note, scale: Scale) -> Vec<Note> {
    match self {
        ChordStamp::Single => vec![note],
        ChordStamp::ScaleTriad => scale.stack_thirds(note,
3).unwrap_or_else(|| vec![note]),
        ChordStamp::ScaleSeventh => scale.stack_thirds(note,
4).unwrap_or_else(|| vec![note]),
        ChordStamp::Fixed(quality) => Chord::new(note.pitch_class(),
quality).notes_from(note),
        ChordStamp::Set(set) => set
            .semitones()
            .into_iter()
            .filter_map(|up| note.transpose(i32::from(up)))
            .collect(),
    }
}
```

Los sellos «Acorde de la tonalidad» son donde el piano roll enseña armonía: haz clic en cualquier fila iluminada y obtienes el acorde que la tonalidad le da a esa nota, sin saber antes su nombre. El nombre aparece luego en la franja de acordes sobre la cuadrícula. Un sello `Set` coloca cualquier grupo de notas elegido en el explorador de conjuntos (Conjuntos de clases de altura, y una armonía con cuatro nombres).

El menú **Dibujar** lista todos los sellos. `all()` encadena dos iteradores: los tres sellos especiales, y luego un sello `Fixed` por clase:

```
pub fn all() -> impl Iterator<Item = ChordStamp> {
    [
        ChordStamp::Single,
        ChordStamp::ScaleTriad,
        ChordStamp::ScaleSeventh,
    ]
    .into_iter()
    .chain(ChordQuality::ALL.into_iter().map(ChordStamp::Fixed))
}
```

`map(ChordStamp::Fixed)` pasa la propia variante como función: una variante de tupla como `Fixed` *es* una función de `ChordQuality` a `ChordStamp`.

Junto al menú Dibujar, un segundo menú elige la **inversión** de un acorde sellado. Invertir sube la nota más grave una octava (`tools::inversion`, en Herramientas de edición como funciones simples), lo que alejaría el acorde de donde hiciste clic, así que el piano roll lo baja una octava y la nota en la que hiciste clic sigue siendo la fundamental:

```
fn stamp_notes(&self, pitch: Note, scale: Scale) -> Vec<Note> {
    let notes = tools::inversion(&self.stamp.notes_at(pitch, scale),
self.inversion);
    if notes.contains(&pitch) {
        return notes;
    }
    notes
        .iter()
        .map(|note| note.transpose(-12))
        .collect::<Option<Vec<Note>>>()
        .unwrap_or(notes)
}
```

Recoger un iterador de `Option`s en un `Option<Vec<_>>` da `None` en cuanto un elemento es `None`. Aquí el tipo se escribe con un **turbofish**, `collect::<...>()`, porque `unwrap_or` se llama sobre el resultado enseguida, y Rust tiene que saber el tipo del resultado para encontrar ese método. Si una nota se saliera por abajo de MIDI, el acorde se queda como estaba. El contorno que sigue al puntero se dibuja con la misma función, así que lo que ves es lo que coloca un clic.

Escribir un acorde

C♯7 y D♭7 se tocan en las mismas teclas, pero se escriben distinto. Un acorde se escribe hacia arriba desde su fundamental por letras: la tercera está dos letras más arriba, la quinta cuatro, la séptima seis.

Acorde	Fundamental	Tercera	Quinta	Séptima
C♯7	Do♯	Mi♯	Sol♯	Si
D♭7	Re♭	Fa	La♭	Do♭

La séptima de D♭7 es Do♭, no Si: seis letras por encima de Re está Do. Así que cada clase dice también qué nota del acorde es cada intervalo, contada como cuentan los músicos:

```
pub fn tones(self) -> &'static [u8] {
    use ChordQuality::*;
    match self {
        Major | Minor | Diminished | Augmented => &[1, 3, 5],
        Sus2 => &[1, 2, 5],
        Sus4 => &[1, 4, 5],
        // ...
        Dominant13 | Major13 => &[1, 3, 5, 7, 9, 13],
    }
}
```

y `spell` recorre los intervalos y las notas juntos con `zip`, pidiendo a `NoteName::above` cada nota: tantos semitonos, `tone - 1` letras más arriba:

```
pub fn spell(self, root: NoteName) -> Vec<NoteName> {
    self.quality
        .intervals()
        .iter()
        .zip(self.quality.tones())
        .map(|(&interval, &tone)| root.above(interval % 12, tone - 1))
        .collect()
}
```

Eso da La mayor como La Do♯ Mi, la séptima disminuida de Do como Do Mi♭ Sol♭ Si𝄫 (su séptima es una séptima, aunque suene como una sexta), y E7♯9 como Mi Sol♯ Si Re Fa𝄪: la novena aumentada es una novena, una letra por encima de la fundamental, así que es una clase de Fa, no Sol.

Un nombre para la fundamental

Escribir necesita el nombre de la fundamental para empezar. En una tonalidad, la tonalidad lo da: `Scale::spell_root` escribe una fundamental de la escala como la escribe la escala, y una de fuera de la escala desde la tónica, como hace `Scale::spell` (Mi♭ mayor en Do mayor, el acorde prestado de Do menor). Una regla más mantiene legibles los nombres de acorde: si esa fundamental escribiría el acorde con dos sostenidos o bemoles en una nota, se usa el otro nombre de la fundamental. En Re♭ mayor, el acorde un semitono por encima de la tónica se escribe Re mayor, no Mi𝄫 mayor.

```
let doubled = |root: NoteName| chord.spell(root).iter().any(|n|
n.accidental.abs() >= 2);
if !doubled(strict) {
    return strict;
}
let other = chord.root_name();
if doubled(other) {
    strict.simplified()
} else {
    other
}
```

Sin tonalidad, como con una serie dodecafónica, `Chord::root_name` elige el nombre sostenido o el bemol de la fundamental, el que escriba el acorde entero con menos signos. Do♯ mayor es Do♯ Mi♯ Sol♯, tres sostenidos; Re♭ mayor es Re♭ Fa La♭, dos bemoles; así que es Re♭.

```
let usual = NoteName::chromatic(self.root);
[Spelling::Sharps, Spelling::Flats]
    .into_iter()
    .map(|spelling| NoteName::plain(self.root, spelling))
    .fold(usual, |best, other| {
        if accidentals(other) < accidentals(best) {
            other
        } else {
            best
        }
    })
```

`fold` lleva un valor a lo largo del iterador: empieza con `usual` y, para cada elemento, el closure devuelve el valor que sigue. Aquí se queda con el nombre que tenga menos signos, y en caso de empate con el que ya tiene, el nombre habitual de la tecla negra.

`Scale::chord_name(chord)` y `Scale::chord_notes(chord)` lo juntan todo: el acorde sobre la tercera nota de La mayor es «C♯m», y en Re menor las notas de La mayor son La Do♯ Mi.

Símbolos de acorde: leer una partitura guía

Una canción que quieres que toque la banda suele llegar como **partitura guía** (*lead sheet*), y una partitura guía nombra sus acordes: «C | G/B | Am | Am/G». La hoja de acordes lee exactamente eso (La hoja de acordes: modelo, distribución y texto), a través de `ChordSymbol`:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct ChordSymbol {
    /// The root, spelled as written: Db7 keeps its Db, it is not C#7.
    pub root: NoteName,
    pub quality: ChordQuality,
    /// The note under the chord, after the slash: the B of "G/B". `None`
    /// when the root is in the bass, as usual.
    pub bass: Option<NoteName>,
}
```

Un símbolo guarda su fundamental como está escrita, un `NoteName`, así que una hoja en Re♭ se queda en Re♭. Para tocarlo, `chord()` lo convierte en un `Chord`, una clase de altura y una clase de acorde.

Muchas maneras de escribir un acorde

Los músicos escriben la misma clase de varias maneras: un menor séptima es Cm7, Cmin7 o C-7; un séptima mayor Cmaj7, CM7 o CΔ7; un semidisminuido Cm7b5 o Cø. Se leen todas, desde una tabla de tuplas:

```
const SPELLINGS: [(ChordQuality, &[&str]); 30] = {
    use ChordQuality::*;
    [
        (Major, &["", "maj", "M"]),
        (Minor, &["m", "min", "mi", "-"]),
        (Diminished, &["dim", "°", "o"]),
        // ...
        (Major13, &["maj13", "M13", "Δ13"]),
    ]
};
```

Cada entrada empareja una clase con un slice de cadenas, `&[&str]`. Los slices pueden tener longitudes distintas, así que una clase puede tener una escritura y otra seis en el mismo array. El bloque `{ use ChordQuality::*; [...] }` es una expresión cuyo valor es su última línea, así que el `use` solo alcanza a la tabla. Leer una clase limpia el texto y busca:

```
fn read_quality(text: &str) -> Option<ChordQuality> {
    let plain: String = text
        .chars()
        .filter(|c| !matches!(c, '(' | ')' | ' '))
        .map(|c| match c {
            '♭' => 'b',
            '♯' => '#',
            other => other,
        })
        .collect();
    SPELLINGS
        .iter()
        .find(|(_, spellings)| spellings.contains(&plain.as_str()))
        .map(|(quality, _)| *quality)
}
```

Los paréntesis y los espacios se quitan, así que «C7(b9)» se lee como «C7b9», y ♭ y ♯ funcionan como `b` y `#`. Las mayúsculas y minúsculas importan en un lugar, uno importante: CM7 es un séptima mayor (Do Mi Sol Si) y Cm7 un menor séptima (Do Mi♭ Sol Si♭). Cuando Ferrous Lineup escribe un acorde de vuelta, usa una manera para cada uno, la que da `symbol`: Cmaj7, Cm7, Cm7b5, Cdim7. Un test escribe cada clase sobre cada fundamental, con y sin bajo, y la vuelve a leer, así que la tabla y `symbol` no pueden separarse.

Acordes con barra, y el acorde de sexta y novena

Una barra pone otra nota en el bajo. Hay dos tipos:

- **Inversiones**: la nota del bajo está en el acorde. C/E es Do mayor con su tercera en el bajo, C/G con su quinta, G7/F con la séptima de G7. `ChordSymbol::bass_in_chord` dice cuál: 3, 5 o 7.
- **Un acorde sobre una nota del bajo**: el bajo no está en el acorde. F/G es Fa mayor sobre un Sol, un sonido abierto que los músicos de pop y gospel usan a menudo en lugar de G11. `bass_in_chord` dice `None`.

Una barra no es una nota del bajo: la de C6/9, el acorde de sexta y novena. Así que solo lo que sigue a la **última** barra, y solo si es un nombre de nota, es un bajo:

```
fn split_bass(rest: &str) -> (&str, Option<NoteName>) {
    match rest.rsplit_once('/') {
        Some((before, after)) => match NoteName::parse(after.trim()) {
            Some(bass) => (before, Some(bass)),
            None => (rest, None),
        },
        None => (rest, None),
    }
}
```

`rsplit_once('/')` parte el texto por su última barra en las partes de antes y de después, o da `None` cuando no hay ninguna. «C6/9» es un acorde de sexta y novena, y «C6/9/E» un acorde de sexta y novena sobre Mi.

Leer un símbolo entero

`ChordSymbol::parse` lee la fundamental con `NoteName::read_either`, en letras o en solfeo, quedándose con la lectura cuyo resto es un acorde, así que «Faug» es Fa aumentado y «Lam7» es La menor séptima:

```
let read = NoteName::read_either(text, |rest| read_rest(rest).is_some());
let Some((root, rest)) = read else {
    return Err(match NoteName::read_either(text, |_| true) {
        None => trf(
            "\"{}\" does not start with a note: chords start with a letter
from A \
             to G, like C, F#m or Bb7, or with a syllable, like Do, Fa#m
or Sib7.",
            &[&text],
        ),
        Some((_, rest)) => trf(
            "\"{}\": \"{}\" is not a kind of chord Ferrous Lineup knows, \
             such as m, 7, maj7, m7b5, sus4 or 13.",
            &[&text, &split_bass(rest).0],
        ),
    });
};
```

El error dice qué salió mal, para que la hoja de acordes pueda mostrarlo junto al compás donde se escribió: o el texto no empieza con una nota, o lo que sigue a la nota no es ningún acorde que Ferrous Lineup conozca.

`ChordSymbol` implementa además el trait estándar `FromStr`, «se puede leer de un texto», llamando a `parse`. Con él, `str::parse` funciona para acordes igual que para números:

```rust
impl FromStr for ChordSymbol {
    type Err = String;

    fn from_str(text: &str) -> Result<Self, Self::Err> {
        ChordSymbol::parse(text)
    }
}
```

así que `let chord: ChordSymbol = "Am7".parse()?;` lee un acorde.

Llevar una hoja de acordes a otra tonalidad

Para llevar una canción a otra tonalidad, un músico mueve cada acorde la misma distancia, contada en letras además de en semitonos. De Mi menor a Re menor cada letra baja una:

Mi menor	F♯m7b5	B7	Em
Re menor	Em7b5	A7	Dm
Do menor	Dm7b5	G7	Cm
Si♭ menor	Cm7b5	F7	B♭m
Sol♯ menor	A♯m7b5	D♯7	G♯m

Es un **ii–V–i**, la forma más común de llegar a una tonalidad menor: en cada tonalidad el ii está una letra por encima de la tónica y el V cuatro letras por encima. Contar por letras lo mantiene así: en Sol♯ menor el V es D♯7, nunca E♭7. `transposed(from, to)` lo hace, moviendo la fundamental y el bajo con `NoteName::above` y tomando el nombre más sencillo para una fundamental que necesitaría dos signos, porque los símbolos de acorde nunca los llevan (Clases de altura, notas y su escritura). Un test comprueba el ii–V–i en las doce tonalidades, escritas con sostenidos y con bemoles.

Nombrar lo que suena

El piano roll nombra lo que suena en cada paso: nada, una nota, dos notas, un acorde que conoce, o un grupo sin nombre de acorde. Cada respuesta necesita datos distintos, que es trabajo para un enum cuyas variantes llevan datos:

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum Harmony {
    /// No notes at all.
    Silence,
    /// One pitch class, maybe in several octaves.
    Single(PitchClass),
    /// Two pitch classes. `low` is the one in the bass, and `semitones`
    /// counts up from it to the other.
    Interval {
        low: PitchClass,
        high: PitchClass,
        semitones: u8,
    },
    /// A chord we know. When `bass` is not the chord's root, the chord is
    /// an inversion, written with a slash: C/E is C major with E in the
bass.
    Chord { chord: Chord, bass: PitchClass },
    // ...
    /// Three or more pitch classes that are not a chord we know, named
    /// by their set class, such as "4-23" for a stack of fourths.
    Unknown(PitchSet),
}
```

Una variante más, `Polychord`, son dos acordes a la vez, como el Mi♭7 sobre Fa♭ mayor que abre los Augurios primaverales de La consagración de la primavera. Solo la escucha de acordes los encuentra, por cómo se agrupan las notas (Oír la armonía: acordes y tonalidades a partir de las notas); las notas solas no pueden decir cuáles pertenecen a cada acorde.

`analyze` recibe cualquier conjunto de notas y devuelve un `Harmony`:

```
pub fn analyze(notes: impl IntoIterator<Item = Note>) -> Harmony {
    let mut pitch_set = PitchSet::EMPTY;
    let mut lowest: Option<Note> = None;
    for note in notes {
        pitch_set.insert(note.pitch_class());
        lowest = Some(match lowest {
            Some(low) => low.min(note),
            None => note,
        });
    }

    let Some(lowest) = lowest else {
        return Harmony::Silence;
    };
    let bass = lowest.pitch_class();

    match pitch_set.len() {
        1 => Harmony::Single(bass),
        2 => {
            let high = pitch_set.iter().find(|&p| p !=
bass).unwrap_or(bass);
            Harmony::Interval {
                low: bass,
                high,
                semitones: bass.semitones_up_to(high),
            }
        }
        _ => match find_chord(pitch_set, bass) {
            Some(chord) => Harmony::Chord { chord, bass },
            None => Harmony::Unknown(pitch_set),
        },
    }
}
```

`impl IntoIterator<Item = Note>` en un *argumento* significa «cualquier cosa que se pueda recorrer y dé notas»: un `Vec<Note>`, un array, o un iterador directamente desde un patrón (`sounding_at`), sin recoger nada por el camino. Un solo bucle construye el conjunto y encuentra la nota más grave, y un `match` sobre el número de clases de altura elige el tipo de respuesta.

Encontrar el acorde

Cualquier nota de un acorde podría ser su fundamental. `find_chord` prueba cada una, empezando por el bajo, porque los acordes se tocan casi siempre con la fundamental abajo:

```
fn find_chord(pitch_set: PitchSet, bass: PitchClass) -> Option<Chord> {
    let roots = || {
        (0..12)
            .map(move |semitones| bass.transpose(semitones))
            .filter(move |&root| pitch_set.contains(root))
    };

    let exact = roots().find_map(|root| {
        ChordQuality::identify(pitch_set.relative_to(root)).map(|quality|
Chord::new(root, quality))
    });

    exact.or_else(|| {
        roots().find_map(|root| {
            let mut with_fifth = pitch_set;
            with_fifth.insert(root.transpose(7));
            if with_fifth == pitch_set {
                return None; // the fifth was already there
            }
            ChordQuality::identify(with_fifth.relative_to(root))
                .filter(|quality| quality.intervals().len() >= 4)
                .map(|quality| Chord::new(root, quality))
        })
    })
}
```

- `roots` es un **closure que fabrica un iterador**. Un iterador se gasta a medida que avanza, así que para buscar dos veces el código llama otra vez a `roots()` para tener uno nuevo.
- `find_map` es `find` y `map` en uno: llama al closure con cada elemento y devuelve el primer `Some`.
- `or_else` sobre un `Option` ejecuta su closure solo si la opción es `None`. La segunda búsqueda, que permite que falte la quinta, solo se ejecuta cuando la exacta no encontró nada. Los músicos suelen dejar fuera la quinta de los acordes de séptima, y Do, Mi y Si♭ se sigue oyendo como C7.
- `Option::filter` conserva un `Some` solo si el closure está de acuerdo. Evita que «Do y Mi, más un Sol adivinado» se llame acorde de Do mayor completo: adivinar solo se permite en acordes de cuatro notas o más.

Empezar por el bajo resuelve ambigüedades reales. Do Mi Sol La es C6 y también Am7: sobre un Do es C6, sobre un La es Am7. Do Re Sol es Csus2 y también Gsus4. Los acordes grandes se solapan todo el tiempo: Am11 y Cmaj13 son las mismas seis clases de altura, La Si Do Re Mi Sol, y el bajo les da nombre.

```
// C E G A is both C6 and Am7: the lowest note decides.
assert_eq!(label(&[60, 64, 67, 69]).unwrap(), "C6");
assert_eq!(label(&[57, 60, 64, 67]).unwrap(), "Am7");
```

Etiquetas: guardas de match y patrones anidados

La franja de acordes sobre el piano roll muestra una etiqueta corta para cada paso:

```
pub fn label(&self, spelling: Spelling) -> Option<String> {
    match *self {
        Harmony::Silence => None,
        Harmony::Single(pitch_class) =>
Some(pitch_class.name(spelling).to_string()),
        // A root and its fifth is a "power chord", written C5.
        Harmony::Interval {
            low, semitones: 7, ..
        } => Some(format!("{}5", low.name(spelling))),
        // A dash, not a plus: "C+" would read as C augmented.
        Harmony::Interval { low, high, .. } => {
            Some(format!("{}-{}", low.name(spelling),
high.name(spelling)))
        }
        Harmony::Chord { chord, bass } if bass != chord.root => {
            Some(format!("{}/{}", chord.name(spelling),
bass.name(spelling)))
        }
        Harmony::Chord { chord, .. } => Some(chord.name(spelling)),
        // ...
        Harmony::Unknown(set) => Some(set.set_class().name()),
    }
}
```

- `semitones: 7` dentro de un patrón encaja solo con ese valor. Los otros intervalos pasan a la rama siguiente.
- `..` significa «y los demás campos, sean cuales sean».
- `if bass != chord.root` es una **guarda de match**: las inversiones llevan barra, los acordes en estado fundamental no.
- Las ramas se prueban desde arriba, así que la más específica va primero. Si el `Harmony::Chord { chord, .. }` simple fuera primero, la rama de la barra nunca encajaría.
- `match *self` hace el match sobre el valor que hay detrás de la referencia. Funciona porque todos los campos son `Copy`, así que los patrones los copian.

Un grupo sin nombre de acorde recibe el nombre de su **clase de conjunto**, como 3-1 para tres semitonos vecinos (Conjuntos de clases de altura, y una armonía con cuatro nombres).

`label` nombra las teclas negras con sostenidos o bemoles simples. `label_in(scale)` escribe la misma respuesta en una tonalidad, con `Scale::spell` y `Scale::spell_root`: E♭ en lugar de D♯ en Do mayor, A/C♯ en Re menor, E♯dim en Fa♯ mayor.

`describe(scale)` escribe una frase para la línea del piano roll sobre el paso bajo el puntero, y para la barra de estado mientras mantienes teclas pulsadas, como «Am: La menor, acorde vi de Do mayor.» o «C/E: Do mayor con Mi en el bajo, acorde I⁶ de Do mayor.» Dice si un acorde es de la tonalidad, **prestado** de la tonalidad homónima, o una **dominante secundaria**; esas palabras se explican en Números romanos, las teclas de armonía y las funciones de los acordes.

Nombrar un intervalo

Dos notas son un **intervalo**, y un intervalo tiene dos nombres. Su tamaño en semitonos da el nombre de todos los días, una tabla de doce (`interval_name`: 3 es una «tercera menor», 6 un «tritono»). Pero los músicos lo nombran por cómo está escrito: las letras dan el número, los semitonos la especie. Do a Mi es una tercera (Do, Re, Mi), y también lo son Do a Mi♭ y Do♯ a Mi; una tercera de cuatro semitonos es mayor, de tres menor. Así que Do a Re♯ es una segunda aumentada, aunque suene como la tercera menor Do a Mi♭.

```
let letters = usize::from(low.letter.steps_up_to(high.letter));
let semitones = low.pitch_class().semitones_up_to(high.pitch_class()) as
i8;
// How far it is from the major or perfect interval, from -6 to 5.
let off = (semitones - MAJOR[letters] + 18).rem_euclid(12) - 6;
let perfect = matches!(letters, 0 | 3 | 4);
let quality = match (perfect, off) {
    (true, 0) => "perfect",
    (false, 0) => "major",
    (false, -1) => "minor",
    (_, 1) => "augmented",
    (true, -1) | (false, -2) => "diminished",
    (_, 2) => "doubly augmented",
    _ => "doubly diminished",
};
```

Los unísonos, las cuartas y las quintas son «justos» y no tienen mayor ni menor; un semitono menos que justo es disminuido, mientras que para los demás disminuido es uno menos que menor. Sumar 18 y tomar `rem_euclid(12)` antes de restar 6 lleva cualquier diferencia al rango de -6 a 5, así que una séptima que pasa de la octava se sigue comparando bien.

En una tonalidad, las dos notas toman los nombres de la tonalidad, así que en La menor armónica Fa a Sol♯ es de verdad una segunda aumentada, el famoso hueco de esa escala. Una nota de fuera de la tonalidad que formaría un intervalo raro se nombra en cambio de la manera habitual para el tamaño del intervalo: en Do mayor, Do y Mi♭ forman una tercera menor.

Inversiones

Un símbolo de acorde escribe una inversión con una barra. Un número romano la escribe con las cifras del **bajo cifrado**, la abreviatura de los teclistas barrocos, que son los intervalos sobre el bajo:

Acorde	Nota más grave	Símbolo	Número en Do mayor
Do mayor	Do, la fundamental	C	I
Do mayor	Mi, la tercera	C/E	I^6 (primera inversión)
Do mayor	Sol, la quinta	C/G	I^6_4 (segunda inversión)
G7	Sol, la fundamental	G7	V7
G7	Si, la tercera	G7/B	V^6_5 (primera inversión)
G7	Re, la quinta	G7/D	V^4_3 (segunda inversión)
G7	Fa, la séptima	G7/F	V^4_2 (tercera inversión)

C/E tiene Do una sexta por encima de Mi y Sol una tercera por encima, «6/3», abreviado ^{6}. Un acorde de séptima tiene tres intervalos sobre su bajo; las cifras conservan los dos que distinguen sus inversiones, y ocupan el lugar del 7: G7/B es V^6_5, no $V7^6_5$.

`Chord::tone_of` encuentra qué nota del acorde es una altura, recorriendo juntos los intervalos y sus números:

```
pub fn tone_of(self, pitch_class: PitchClass) -> Option<u8> {
    let semitones = self.root.semitones_up_to(pitch_class);
    self.quality
        .intervals()
        .iter()
        .zip(self.quality.tones())
        .find(|(interval, _)| **interval % 12 == semitones)
        .map(|(_, &tone)| tone)
}
```

e `inversion_figures` lo convierte en cifras con un solo `match` sobre una tupla:

```
match (self.tone_of(bass), triad, seventh) {
    (Some(3), true, _) => "⁶",
    (Some(5), true, _) => "⁶₄",
    (Some(3), _, true) => "⁶₅",
    (Some(5), _, true) => "⁴₃",
    (Some(7), _, true) => "⁴₂",
    _ => "",
}
```

Los acordes sus, de sexta y de novena no tienen cifras; se quedan con el nombre con barra. Cómo van las cifras sobre un número romano, y qué cambia una inversión sobre adónde va un acorde después (la cuarta y sexta cadencial), está en Números romanos, las teclas de armonía y las funciones de los acordes.

Una pista, o todas a la vez

La franja de acordes del piano roll tiene dos filas. La primera nombra los acordes de la pista que se está editando, paso a paso, con `analyze`. El piano de la banda puede tocar G7 como Si Re Fa Sol, así que por sí solo es G7/B. La segunda fila, **Todo**, nombra lo que toca la banda entera ahí, pulso a pulso: si el bajo toca Fa bajo ese G7, la banda toca G7/F, V^4_2. Nombrar la banda entera es más difícil que nombrar una pista, porque la melodía y la línea del bajo tienen notas de paso que no están en el acorde, y un arpegio toca su acorde nota a nota. Ese es el trabajo de la escucha de acordes, en Oír la armonía: acordes y tonalidades a partir de las notas.

Cómo se prueba

Los tests se leen como un libro de armonía. Las notas de cada clase son distintas, cada clase tiene un número de nota para cada intervalo, y los acordes se escriben por sus letras:

```
assert_eq!(
    spelled(name(Letter::C, 0), ChordQuality::Diminished7),
    "C Eb Gb Bbb"
);
assert_eq!(
    spelled(name(Letter::E, 0), ChordQuality::Dominant7Sharp9),
    "E G# B D F##"
);
```

El análisis nombra disposiciones reales, en cualquier orden y octava, y sin su quinta:

```
assert_eq!(label(&[48, 64, 67, 69, 74]).unwrap(), "C6/9");
assert_eq!(label(&[55, 59, 62, 65, 68]).unwrap(), "G7b9");
// E, G#, B, D and G: the "Hendrix chord".
assert_eq!(label(&[52, 56, 59, 62, 67]).unwrap(), "E7#9");
```

```
assert_eq!(label(&[48, 64, 70]).unwrap(), "C7");
```

Las descripciones usan la tonalidad, inversiones incluidas:

```
assert_eq!(describe(&[57, 60, 64]), "Am: A minor, chord vi of C major.");
```

Y los símbolos de acorde se leen en todas sus maneras de escribirlos, de «C» y «Am7» a «C7(b9)», «Bø» y «Cm(maj7)», y se llevan a las doce tonalidades por sus letras.

Los ejercicios sobre acordes están en Ejercicios: la teoría musical en código.

Números romanos, las teclas de armonía y las funciones de los acordes

Archivos: `crates/lineup-core/src/harmony/scale.rs`, `crates/lineup-core/src/harmony/keys.rs`, `crates/lineup-core/src/harmony/progression.rs`, `crates/lineup-core/src/library/numeral.rs`, `crates/lineup-core/src/chord_sheet/function.rs` y `crates/lineup-app/src/harmony_keys.rs`

El nombre de un acorde dice qué es; su **número romano** dice qué es *en la tonalidad*. Los números son la manera en que Ferrous Lineup enseña armonía: rotulan las teclas de armonía, van debajo de cada acorde de la hoja de acordes, explican las sugerencias sobre lo que podría venir después, y la biblioteca escribe sus progresiones con ellos. Este capítulo trata de escribir y leer números romanos, de las teclas de armonía, de las tablas de movimientos frecuentes, y de cómo averigua la hoja de acordes qué está haciendo ahí un acorde de fuera de la tonalidad.

La música, en pocas palabras

Una tonalidad tiene siete acordes, uno sobre cada nota de su escala, y los músicos los numeran con números romanos: en mayúscula para un acorde mayor, en minúscula para uno menor, ° para disminuido, + para aumentado, ø para semidisminuido.

```
I    ii    iii   IV    V     vi    vii°
C    Dm    Em    F     G     Am    Bdim
```

Los números permiten hablar de una progresión en todas las tonalidades a la vez: I vi IV V es C Am F G en Do mayor y G Em C D en Sol mayor.

Cada acorde hace uno de tres trabajos en su tonalidad, su **función**:

Función	Acordes	Se siente como
Tónica	I, iii, vi	casa: la música puede descansar aquí
Subdominante	ii, IV	salir de casa, camino de la dominante
Dominante	V, vii°	tensión que tira de vuelta a casa, al I

El tirón de la dominante viene de su tercera, Si en Sol mayor, un semitono por debajo de Do: la **sensible**, que quiere subir a la tónica. En una tonalidad menor la escala no tiene sensible (en La menor, Sol está un tono entero por debajo de La), así que la

música en menor la sube a Sol♯, que es lo que es la escala menor armónica, y su V es mayor: E, no Em.

Unos pocos movimientos entre los acordes son la columna vertebral de casi toda la música:

Movimiento	En Do	Por qué funciona
V a I	G a C	el final más fuerte: la sensible sube a la tónica
IV a V	F a G	un paso hacia arriba, de alejarse a querer volver a casa
ii a V	Dm a G	la fundamental baja una quinta; ii V I está en todas partes, del jazz al pop
IV a I	F a C	la cadencia plagal, el suave «amén»
V a vi	G a Am	la cadencia rota: esperas C y llega su relativa menor

Una fundamental que **baja una quinta** (Sol a Do, Re a Sol, La a Re) es el movimiento más fuerte que hay, y muchas de las sugerencias son de ese tipo.

Las canciones usan también acordes de fuera de la tonalidad, y la armonía nombra lo que hacen. Un acorde **prestado** viene de la tonalidad menor sobre la misma tónica, como Fm en Do mayor. Una **dominante secundaria** es la dominante de otro acorde de la tonalidad: D7 es el V de Sol, escrito V7/V, y por un momento hace que Sol suene a casa. La **napolitana** es un acorde mayor sobre el segundo grado rebajado, Re♭ en Do. Un **acorde pivote** pertenece a dos tonalidades, y la música cambia de tonalidad a través de él. Los capítulos Qué hacen los acordes: números y funciones y Acordes de fuera de la tonalidad del manual enseñan todo esto con ejemplos para tocar.

Escribir un número romano

Un número romano tiene tres partes: un sostenido o un bemol delante, el número, y lo que lo sigue, el «7» de V7 o el «°» de vii°. Las tres salen de una función:

```
fn numeral_on(self, letters: u8, chord: Chord) -> Option<String> {
    const NUMERALS: [&str; 7] = ["I", "II", "III", "IV", "V", "VI",
"VII"];
    if !self.is_heptatonic() {
        return None;
    }
    let letters = usize::from(letters);
    let unaltered =
        self.unaltered_degree(letters, chord.quality,
MinorNumerals::current()) as i8;
    let semitones = self.root.semitones_up_to(chord.root) as i8;
    // How far the root is from the note the numeral names without an
    // accidental, from -6 to 5.
    let shift = (semitones - unaltered + 18).rem_euclid(12) - 6;
    let accidentals = if shift < 0 { "b" } else { "#"
}.repeat(shift.unsigned_abs() as usize);
    let numeral = if chord.quality.is_minor() {
        NUMERALS[letters].to_lowercase()
    } else {
        NUMERALS[letters].to_string()
    };
    Some(format!(
        "{accidentals}{numeral}{}",
        chord.quality.numeral_suffix()
    ))
}
```

- `letters` es el grado, cuántas letras por encima de la tónica está la fundamental. Elige el número: cuatro letras arriba es V.
- `unaltered` es la nota que nombra el número sin alteración, en semitonos por encima de la tónica. En una tonalidad mayor, la nota de la escala mayor: V nombra la nota siete semitonos arriba. (En una tonalidad menor puede ser otra, como muestra la sección siguiente.)
- `shift` es cuánto se aparta de esa nota la fundamental del acorde. La diferencia entre dos clases de altura solo se conoce salvo octavas, así que sumar 18, tomar el resto entre 12 y restar 6 da el camino más corto, de -6 a 5: una fundamental un semitono por debajo da -1, un bemol, no +11.
- La mayúscula o minúscula sale de la clase de acorde, y el final de `numeral_suffix` (Acordes: construirlos, nombrarlos e invertirlos).

Así, en Do mayor, Si♭ mayor es ♭VII: el VII sin alteración nombra Si, once semitonos por encima de Do; Si♭ está a diez, uno por debajo, así que lleva un bemol.

Sobre qué grado está un acorde se puede contar de dos maneras, y se usan las dos. `roman_numeral(chord)` cuenta desde el sonido: la letra de la fundamental en la escritura propia de la tonalidad (Escalas, modos y tonalidades). `spelled_numeral(root, chord)` cuenta desde la fundamental como se **escribió**, así que un acorde escrito en la hoja de acordes conserva su letra. En Do mayor, C♯°7, una

tónica subida que lleva a Dm, es ♯i°7, y D♭7 es ♭II7. Por el sonido solo, los dos estarían sobre el segundo grado.

```
pub fn spelled_numeral(self, root: NoteName, chord: Chord) ->
Option<String> {
    self.numeral_on(self.tonic_name().letter.steps_up_to(root.letter),
chord)
}
```

Cifras

Las cifras de una inversión (Acordes: construirlos, nombrarlos e invertirlos) van después del número, en el lugar del 7 de un acorde de séptima: V7 con su tercera en el bajo es V^6_5. Una dominante secundaria conserva su destino después de las cifras: V^6_5/vi.

```
pub fn with_figures(numeral: &str, figures: &str) -> String {
    if figures.is_empty() {
        return numeral.to_string();
    }
    // A secondary dominant's figures go on its own numeral, before the
    // slash: V⁶₅/vi.
    let (own, target) = match numeral.split_once('/') {
        Some((own, target)) => (own, format!("/{target}")),
        None => (numeral, String::new()),
    };
    let own = own.strip_suffix('7').unwrap_or(own);
    format!("{own}{figures}{target}")
}
```

`split_once('/')` parte en la primera barra; `strip_suffix('7')` da el texto sin un 7 final, o `None` si no lo tiene.

Leer un número romano

La biblioteca guarda sus progresiones como números romanos, y la hoja de acordes los acepta escritos. `numeral::read` desarma un número:

```
let mut shift: i32 = 0;
let mut rest = text;
loop {
    if let Some(after) = rest.strip_prefix('b') {
        shift -= 1;
        rest = after;
    } else if let Some(after) = rest.strip_prefix('#') {
        shift += 1;
        rest = after;
    } else {
        break;
    }
}
// The numeral, in either case. `get(..len)` gives `None` instead of
// panicking when `rest` is shorter than the numeral.
let (numeral, degree) = NUMERALS.iter().find(|(numeral, _)| {
    rest.get(..numeral.len())
        .is_some_and(|start| start.eq_ignore_ascii_case(numeral))
})?;
```

`strip_prefix` da el resto del texto cuando empieza con el prefijo. Cortar una cadena con `&rest[..n]` provoca un panic cuando `n` pasa del final; `get(..n)` da `None` en su lugar, así que un texto corto simplemente no es ese número. La tabla pone III antes que II e I, y VII antes que VI y V, así que un número más largo se prueba antes que uno más corto con el que empieza. Un número con mayúsculas y minúsculas mezcladas, como «Iv», se rechaza, porque es su mayúscula o minúscula lo que dice mayor o menor.

`parse_in(text, key, numerals)` convierte luego el grado y la alteración en semitonos por encima de la tónica, con el mismo `unaltered_degree` que usa la escritura, así que escribir un número y leerlo de vuelta nunca pueden discrepar. Un test lo comprueba con 10.368 números: las dos maneras de contar, cuatro clases de tonalidad, doce tónicas, un acorde sobre cada una de las doce notas y nueve clases de acorde.

Tonalidades menores: bIII o III

En La menor, el acorde de Do mayor tiene dos nombres. Los libros de pop y de jazz lo llaman **♭III**; los libros de armonía clásica lo llaman **III**. Los dos tienen razón, y **Edición > Ajustes… > Armonía > Números romanos en tonalidades menores** deja elegir.

Contados desde la escala mayor, los números se miden contra la escala mayor sobre la misma tónica, La mayor, cuya tercera es Do♯. Do está un semitono por debajo, así que su acorde es ♭III; Fa es ♭VI y Sol ♭VII. Una sola tabla de progresiones

funciona entonces en todas las tonalidades y modos: ♭VI ♭VII i es la misma progresión en La menor y en La dórico, y una canción en Do mayor que toma prestados La♭ y Si♭ de Do menor los escribe ♭VI y ♭VII también. La biblioteca escribe siempre sus progresiones así.

Contados desde la escala menor, los números se miden contra la escala propia de la tonalidad. Do es simplemente su tercera nota, así que su acorde es III.

Solo difieren el tercer, el sexto y el séptimo grado, las notas que cambian entre La mayor y La menor. Pero la sexta y la séptima nota de una tonalidad menor además se mueven: la menor armónica sube Sol a Sol♯, la melódica sube también Fa a Fa♯. Así que un número sin alteración en esos grados tiene que nombrar una de dos notas, y la regla que usan los libros clásicos es: **un número sin alteración nombra la forma cuyo acorde pertenece a la tonalidad**. VII es Sol mayor, de la menor natural; vii° es G♯°, el acorde de sensible de la menor armónica, ya que G° (Sol Si♭ Re♭) no es ningún acorde de La menor.

```
                    desde la escala mayor          desde la escala menor
menor natural   i ii° bIII  iv v  bVI bVII     i ii° III  iv v  VI  VII
menor armónica  i ii° bIII+ iv V  bVI vii°     i ii° III+ iv V  VI  vii°
menor melódica  i ii  bIII+ IV V  vi° vii°     i ii  III+ IV V  vi° vii°
```

Esa regla es `unaltered_degree`:

```
pub(crate) fn unaltered_degree(
    self,
    letters: usize,
    quality: ChordQuality,
    numerals: MinorNumerals,
) -> u8 {
    use ScaleKind::{HarmonicMinor, MelodicMinor, NaturalMinor};
    if numerals == MinorNumerals::FromMajor || !self.is_minor_key() {
        return ScaleKind::Major.tonal_intervals()[letters];
    }
    let natural = NaturalMinor.tonal_intervals()[letters];
    if letters < 5 {
        return natural;
    }
    // Whether the chord on `semitones` is in one of these scales,
    // counted from C, since only the distance from the tonic counts.
    let of_key = |semitones: u8, kinds: &[ScaleKind]| {
        let chord =
Chord::new(PitchClass::C.transpose(i32::from(semitones)), quality);
        kinds
            .iter()
            .any(|&kind| Scale::new(PitchClass::C,
kind).is_diatonic(chord))
    };
    if !of_key(natural, &[NaturalMinor]) && of_key(natural + 1, &
[HarmonicMinor, MelodicMinor])
    {
        natural + 1
    } else {
        natural
    }
}
```

Para los cinco primeros grados la respuesta es la nota de la menor natural. Para el sexto y el séptimo, la función hace la pregunta de la regla: ¿el acorde sobre la nota natural es un acorde de la menor natural? Si no lo es, y el acorde un semitono más arriba está en la menor armónica o en la melódica, la nota subida es la que se quiere decir. `of_key` construye el acorde sobre Do, porque solo importa su distancia a la tónica.

El ajuste cambia cómo se **escriben** y se **leen** los números, nunca un acorde. Una canción guarda acordes, no números, así que es la misma elijas lo que elijas. Las tonalidades mayores y los modos se escriben igual de las dos maneras.

Un ajuste que toda función puede ver, uno por hilo

Los números se escriben en muchos lugares, las teclas de armonía, la hoja de acordes, el piano roll, las sugerencias, una exportación de video, y todos terminan en

`numeral_on`. Pasar el ajuste como parámetro cambiaría la firma de decenas de funciones por el camino, para una elección que cambia de vez en cuando. Así que el ajuste se guarda donde `numeral_on` puede leerlo:

```
thread_local! {
    /// How the numerals written on this thread count a minor key's
    /// degrees. The window's thread sets it from the settings each frame;
    /// each thread has its own, so tests running side by side never see
    /// each other's.
    static MINOR_NUMERALS: Cell<MinorNumerals> = const {
Cell::new(MinorNumerals::FromMajor) };
}

impl MinorNumerals {
    /// How numerals are written on this thread.
    pub fn current() -> MinorNumerals {
        MINOR_NUMERALS.with(Cell::get)
    }

    /// Writes numerals this way on this thread from now on.
    pub fn set(self) {
        MINOR_NUMERALS.with(|numerals| numerals.set(self));
    }
}
```

El hilo de la ventana lo fija antes de dibujar cada frame:

```
// Every view writes numerals as the settings say.
settings::minor_numerals(ctx).set();
```

Dos piezas de Rust lo hacen seguro:

- **`thread_local!`** le da a cada hilo su propia copia del valor. `cargo test` ejecuta los tests en muchos hilos a la vez, y un test que escribe números desde la escala menor no debe cambiar lo que ve un test que corre a su lado. Con un `static` normal lo cambiaría, y el otro test fallaría de vez en cuando, según los tiempos.
- **`Cell`** permite cambiar un valor a través de una referencia compartida. A un valor de hilo solo se llega con `&`, así que un valor simple nunca se podría fijar. `Cell` funciona metiendo y sacando valores enteros (`get`, `set`), sin entregar nunca una referencia a lo de dentro, así que nada se puede leer mientras cambia.

Como el valor pertenece a un hilo, un hilo nuevo empieza con el valor por defecto. Una exportación de video dibuja sus vistas en un hilo propio, así que se lleva el ajuste, junto con el idioma y la nomenclatura de notas (Exportar: WAV, MIDI y video):

```
let numerals = MinorNumerals::current();
```

y llama a `numerals.set()` lo primero en el hilo nuevo.

Devolverlo, incluso después de un panic

Los tests escriben los números de una manera durante un rato, y luego vuelven:

```
pub fn with<T>(self, f: impl FnOnce() -> T) -> T {
    struct Restore(MinorNumerals);
    impl Drop for Restore {
        fn drop(&mut self) {
            self.0.set();
        }
    }
    let _restore = Restore(MinorNumerals::current());
    self.set();
    f()
}
```

`Restore` guarda el valor viejo, y su `Drop` lo devuelve. Rust suelta una variable local cuando sale de su ámbito, y también mientras un **panic** se desenrolla a través de la función. Un `assert!` que falla dentro de `f` es un panic, así que incluso entonces vuelve el ajuste viejo, y el siguiente test de ese hilo empieza limpio. Limpiar en un `Drop` así se llama una **guarda**; es como el `MutexGuard` de la biblioteca estándar libera un mutex.

La variable se llama `_restore`, no `_`. Un `_` solo no liga nada, así que la guarda se soltaría enseguida, en la misma línea, antes de `self.set()`, y nada devolvería el valor viejo después de `f`.

Las sugerencias son frases escritas una vez, con los números como los escribe la biblioteca: «I to IV: moving away from home». Cuando los números se cuentan desde la escala menor, `numeral::written_in` reescribe los números dentro de esa frase. Casi siempre no cambia nada, así que devuelve un `Cow<str>`: el texto original prestado, o un `String` nuevo solo cuando tuvo que reescribir algo.

Las teclas de armonía

Las teclas de armonía de la parte de abajo de la ventana son una tecla grande para cada acorde de la tonalidad, coloreadas por función, que se tocan manteniéndolas

pulsadas. El modelo es teoría pura, así que vive en `lineup-core`, con tests, y no sabe nada de egui:

```
pub fn chord_keys(scale: Scale, size: ChordSize) -> Vec<ChordKey> {
    let scale = chord_scale(scale);
    let leading_tone = Scale::new(scale.root, ScaleKind::HarmonicMinor);
    (0..7)
        .filter_map(|degree| {
            let from = if scale.kind == ScaleKind::NaturalMinor && degree
== 4 {
                leading_tone
            } else {
                scale
            };
            let chord = chord_on(from, degree, size)?;
            Some(ChordKey {
                degree,
                chord,
                numeral: scale.roman_numeral(chord)?,
                function: has_functions(scale.kind).then(||
Function::of(degree)),
            })
        })
        .collect()
}
```

- `filter_map` ejecuta el closure con cada grado y se queda con los `Some`, y dentro del closure `?` convierte un `None` en «deja este fuera».
- En una tonalidad menor natural, el acorde sobre V sale de la menor armónica: mayor, con la sensible.
- `chord_scale` le da a una escala pentatónica o de blues, que no tiene siete acordes propios, la escala mayor o menor sobre la misma tónica.
- `bool::then` convierte `true` en `Some` del valor del closure y `false` en `None`. Los modos no tienen función: no tienen una dominante que tire hacia casa como lo hace V. En Do locrio el acorde sobre la quinta nota es Sol♭ mayor, que no tira hacia Do en absoluto.

`Function::of` es la tabla de trabajos por grado:

```
pub fn of(degree: usize) -> Function {
    match degree % 7 {
        1 | 3 => Function::Subdominant,
        4 | 6 => Function::Dominant,
        _ => Function::Tonic,
    }
}
```

`ChordSize` elige tríadas, séptimas o novenas. En dos grados de una tonalidad mayor, iii y vii, la novena cae un semitono por encima de la fundamental, una octava más arriba; eso choca, y los músicos la dejan fuera, así que `Scale::ninth` da `None` ahí y la tecla vuelve al acorde de séptima:

```
ChordSize::Ninths => scale.ninth(degree).or_else(||
scale.seventh(degree)),
```

`Option::or_else` prueba el closure solo cuando el primer valor es `None`: «si eso no funcionó, prueba esto».

Tocar una tecla

Pulsar C y luego F podría tocar Do Mi Sol y luego Fa La Do, saltando todas las notas hacia arriba. Un pianista mantendría el Do y movería los otros dos dedos un paso cada uno, a Do Fa La. `keys::play` dispone cada acorde cerca del anterior (la disposición en sí es la de la banda, en Cómo toca la banda una hoja de acordes) y añade la fundamental debajo, como lo haría una mano izquierda:

```
pub fn play(chord: Chord, previous: Option<&[Note]>) -> (Vec<Note>,
Vec<Note>) {
    let voicing = voice(chord, previous);
    let mut notes = voicing.clone();
    if let Some(&lowest) = voicing.first() {
        // The root below the lowest note: an octave below if the lowest
        // note is the root itself.
        let down = match chord.root.semitones_up_to(lowest.pitch_class())
{
            0 => 12,
            gap => gap,
        };
        if let Some(root) = lowest.transpose(-i32::from(down)) {
            notes.insert(0, root);
        }
    }
    (voicing, notes)
}
```

Devuelve una tupla de dos vectores: la disposición, para recordarla la próxima vez, y las notas que hay que tocar. Quien llama la desarma con `let (voicing, notes) = ...`.

Una tecla de armonía es una tecla, no un botón: su acorde empieza cuando baja el botón del ratón y se detiene cuando sube, y las teclas de número 1 a 7 también las

tocan. Quién mantiene pulsada una tecla es un enum cuya variante lleva la tecla de número:

```
enum Holder {
    Mouse,
    /// A number key on the computer keyboard, 1 to 7.
    Digit(Key),
}

/// The harmony key that is down, and the notes it started.
#[derive(Debug, Clone, PartialEq)]
struct Down {
    key: usize,
    holder: Holder,
    notes: Vec<Note>,
}
```

Solo quien empezó un acorde puede detenerlo, y `Down` guarda las notas que empezó, así que se detienen las notas correctas aunque la tonalidad o el tamaño de acorde hayan cambiado mientras se mantenía. Las notas van por el mismo camino que las del teclado del equipo (MIDI: mensajes, dispositivos, grabación y archivos), así que suenan en el canal seleccionado, se nombran en la barra de estado y se graban, sin que ese código sepa que existen las teclas de armonía.

Qué tecla se ilumina

Mientras suena la música, la tecla del acorde que oyes se ilumina y se enmarca en blanco, la marca más visible de la barra. Ganan las teclas pulsadas; si no, el acorde es el que se oye durante el pulso en el cursor de reproducción, según la escucha de acordes (Oír la armonía: acordes y tonalidades a partir de las notas), así que un arpegio ilumina su acorde y dos acordes a la vez iluminan dos teclas. Un acorde ilumina una tecla cuando tiene la misma fundamental y el mismo tipo de tríada que el acorde de la tonalidad ahí:

```
pub fn key_for(keys: &[ChordKey], chord: Chord) -> Option<usize> {
    keys.iter().position(|key| {
        key.chord.root == chord.root && key.chord.quality.family() ==
chord.quality.family()
    })
}
```

G7 ilumina el V en Do mayor. D7 no ilumina nada: el acorde de la tonalidad sobre Re es menor, y D7 viene de fuera de la tonalidad.

Qué viene después: el conocimiento como datos

Después de un acorde, las teclas que suelen venir a continuación reciben un contorno amarillo fino, la más habitual un poco más brillante, y una línea de sugerencia dice por qué. `next_chords(scale, current)` da esos acordes, el más habitual primero. El conocimiento no está en el código sino en tablas: cada acorde se encuentra por su fundamental (semitonos por encima de la tónica) y su tipo de tríada, y lista los movimientos que suelen seguirlo.

```
struct Move {
    to: u8,
    short: &'static str,
    reason: &'static str,
}

const fn go(to: u8, short: &'static str, reason: &'static str) -> Move {
    Move { to, short, reason }
}

struct Slot {
    root: u8,
    family: ChordFamily,
    moves: &'static [Move],
}
```

```
const MAJOR: &[Slot] = &[
    slot(
        0,
        Major,
        &[
            go(
                5,
                "away from home",
                "I to IV: moving away from home, toward the dominant.",
            ),
            go(
                7,
                "tension, wants to go home",
                "I to V: tension that wants to come back to I.",
            ),
            // ...
        ],
    ),
    // ...
];
```

Una constante solo se puede construir con código que el compilador pueda ejecutar mientras compila, y una `const fn` es código así, de modo que los diminutos `go` y `slot` dejan cada movimiento en una o dos líneas. `MAJOR` es un slice de entradas, así

que tiene tantas como necesite una tonalidad, incluidos los prestados iv, ♭III, ♭VI y ♭VII. Todo está dentro del programa. Cambiar lo que sugiere la aplicación es editar una tabla, y los tests comprueban la tabla: cada grado de tres clases de tonalidad tiene sugerencias, y ningún acorde se sugiere a sí mismo.

Cada movimiento lleva su razón dos veces, en una frase y en pocas palabras. El nivel de la línea de sugerencias, **Detallado**, **Breve** o **Mínimo**, elige una:

```
pub fn why(&self, level: HintLevel) -> Option<&str> {
    match level {
        HintLevel::Detailed => Some(&self.reason),
        HintLevel::Short => Some(&self.short),
        HintLevel::Minimal => None,
    }
}
```

Una `Suggestion` guarda sus razones como `Cow<'static, str>`, «clonar al escribir» (*clone on write*): casi siempre un `&'static str` prestado de la tabla, sin reservar memoria, y un `String` propio solo cuando `written_in` tuvo que reescribir sus números.

Las tonalidades menores encuentran sus acordes en otra tabla, la menor natural con el V mayor y el vii° de sensible de la menor armónica:

```
const MINOR_CHORDS: [(u8, ChordQuality); 8] = [
    (0, ChordQuality::Minor),
    (2, ChordQuality::Diminished),
    (3, ChordQuality::Major),
    (5, ChordQuality::Minor),
    (7, ChordQuality::Major),
    (8, ChordQuality::Major),
    (10, ChordQuality::Major),
    (11, ChordQuality::Diminished),
];
```

Primero, las dominantes secundarias

Un acorde de fuera de la tonalidad que suena a dominante es casi siempre la dominante del acorde una quinta más abajo, así que `next_chords` lo comprueba antes de mirar las tablas:

```
let dominant_kind = current.quality == ChordQuality::Major ||
current.quality.is_dominant();
let target = (root + 5) % 12;
if dominant_kind
    && !scale.is_chord_of_key(current)
    && target != 0
    && let Some(suggestion) = suggest(target, SECONDARY_DOMINANT_SHORT,
SECONDARY_DOMINANT)
    && suggestion.chord.quality != ChordQuality::Diminished
{
    return vec![suggestion];
}
```

Lee la cadena de `let` como la regla que es: el acorde suena a dominante, no es uno de los acordes propios de la tonalidad, el acorde una quinta más abajo no es la tónica (en una tonalidad menor, un acorde mayor sobre V es la dominante propia de la tonalidad), hay un acorde de la tonalidad ahí, y no es disminuido, que es demasiado inestable para sonar a casa ni siquiera un momento. Una quinta hacia abajo y una cuarta hacia arriba son la misma clase de altura, así que el destino es `(root + 5) % 12`. En Do mayor, D mayor sugiere G, A7 Dm, E Am, y C7, cuyo Si♭ hace que ya no sea el acorde de tónica, F.

El bajo importa: la cuarta y sexta cadencial

C/G, el acorde de tónica con su quinta en el bajo, suena inestable: sobre Sol, el Do y el Mi quieren bajar un paso a Si y Re, lo que forma G. Así que en una cadencia C/G sigue hacia V, y se llama **cuarta y sexta cadencial** (I^6_4). `next_chords_over` recibe también el bajo, y trata ese caso antes de pasarle todo lo demás a `next_chords`:

```
let six_four = current == Chord::new(scale.root, tonic) &&
current.tone_of(bass) == Some(5);
if !six_four {
    return next_chords(scale, current);
}
```

Qué hace un acorde: las funciones de la hoja de acordes

La hoja de acordes escribe un número romano bajo cada acorde, y para un acorde de fuera de la tonalidad, qué está haciendo ahí. Eso depende de algo más que el acorde y la tonalidad: un acorde secundario se nombra por adónde va, y un pivote por los

acordes que rodean un cambio de tonalidad. Así que `Functions::read` lee la hoja entera de una pasada, y la hoja le pregunta por cada acorde. Lo que hace un acorde es un `Role`, un enum cuyas variantes llevan lo que necesitan:

```
pub enum Role {
    /// One of the key's own chords, or a chord outside it that none of
    /// the others names.
    Plain,
    /// Taken from the parallel key, `from`.
    Borrowed { from: Scale },
    /// The dominant of `of`, one of the key's chords.
    SecondaryDominant { of: Chord },
    /// The chord on the leading tone of `of`, one of the key's chords.
    SecondaryLeadingTone { of: Chord },
    /// A major chord on the lowered second degree.
    Neapolitan,
}
```

Un `match` saca los datos donde se usan, así que `of` solo existe en las ramas donde significa algo:

```
match self.role {
    Role::SecondaryDominant { of } => Some(with_figures(
        &format!(
            "V{}/{}",
            chord.quality.numeral_suffix(),
            self.target_numeral(of)?
        ),
        figures,
    )),
    // ...
    Role::Neapolitan => Some(with_figures("N", figures)),
    Role::Plain | Role::Borrowed { .. } => figured(self.key, self.symbol),
}
```

`Role::Borrowed { .. }` encaja con un acorde prestado sea cual sea la tonalidad de la que viene.

Los métodos de la tonalidad

Las preguntas detrás de los papeles son métodos de `Scale`:

- `parallel()`: la tonalidad con la misma tónica y el otro modo, Do menor para Do mayor.
- `is_chord_of_key(chord)`: de la escala, o en una tonalidad menor el V o el vii° con la séptima subida de la menor armónica. E7 y G♯dim7 pertenecen a La

menor.

- `borrowed_from(chord)`: la tonalidad homónima, si el acorde no es de la tonalidad pero sí de esa. Una tonalidad mayor también toma prestado de la menor armónica: B♭ y Bdim7 en Do mayor vienen los dos de Do menor.
- `secondary_dominant_target(chord)`: para un acorde mayor o de séptima de dominante de fuera de la tonalidad, el acorde mayor o menor de la tonalidad (no la tónica) una quinta más abajo: E7 en Do mayor es el V7 de Am.
- `secondary_leading_tone_target(chord)`: para un acorde disminuido, el acorde un semitono más arriba: F♯°7 en Do mayor lleva a G.

Qué papel va primero

Algunos acordes podrían ser dos cosas. En La menor, La mayor es la dominante de Dm y también el acorde de tónica mayor prestado de La mayor; Re mayor es el IV de la menor melódica y la dominante de Sol. `role` decide por adónde va el acorde:

```
let goes_to = |of: Chord| next.is_some_and(|next| arrives(next, of));
let dominant = key.secondary_dominant_target(chord);
let leading = key.secondary_leading_tone_target(chord);
let secondary = match (dominant, leading) {
    (Some(of), _) => Some(Role::SecondaryDominant { of }),
    // A leading-tone chord is named after where it goes: F#ø7 going
    // on to B7 in C major is the ii of E minor, not the vii of G.
    (_, Some(of)) if goes_to(of) => Some(Role::SecondaryLeadingTone { of
}),
    _ => None,
};
// A chord that is both, such as A major in A minor (the major tonic,
// or the V of Dm), is a secondary dominant where it goes to the
// chord it leads to, and a borrowed chord anywhere else.
if let Some(of) = dominant.or(leading)
    && goes_to(of)
{
    return secondary.unwrap_or(Role::Plain);
}
// So is a chord of melodic minor, such as D major in A minor: the V
// of G going to G, and the key's own IV anywhere else.
if in_minor(key, chord, &[MelodicMinor]) {
    return Role::Plain;
}
if let Some(from) = borrowed_from(key, symbol) {
    return Role::Borrowed { from };
}
if is_neapolitan(key, symbol) {
    return Role::Neapolitan;
}
secondary.unwrap_or(Role::Plain)
```

Antes de esto, `role` ya ha despachado los acordes que no necesitan más nombre que su número: todos los acordes de un modo (Mi mayor en Mi frigio es el acorde de tónica del flamenco, no la dominante de La menor), todos los acordes de la tonalidad, y en una tonalidad menor todos los acordes de su menor natural o armónica. Un acorde «va a» su destino cuando el acorde siguiente está sobre esa fundamental, o es un acorde de cuarta y sexta sobre ella, que retiene el destino un momento. `is_some_and` responde `false` para `None`, así que el último acorde, sin nada después, no va a ninguna parte.

Un acorde prestado tiene que estar **escrito** como lo escribe la otra tonalidad, no solo sonar como él. Sol aumentado (Sol Si Re♯) se toca en las mismas teclas que Mi♭ aumentado (Mi♭ Sol Si), un acorde de Do menor armónica. Pero su Re♯ sube a Mi, donde un Mi♭ bajaría: no es ninguna nota de Do menor. Así que `borrowed_from` comprueba que cada nota del acorde está escrita en la letra que le dan las escalas de la tonalidad, contando hacia arriba desde la tónica. Del mismo modo la napolitana tiene que estar escrita sobre la segunda letra: Re♭ mayor en Do lo es; Do♯ mayor, las mismas teclas, es un acorde cromático sobre la tónica.

Encontrar el pivote

Una **modulación** es un paso a una tonalidad nueva en la que la música se queda. En Do mayor Am es vi; en Sol mayor es ii. En C, Am, D7, G el oído oye Am como vi en Do; luego llega D7, que Do mayor no tiene, y el oído entiende que Am era en realidad el ii de Sol. Am es el **acorde pivote**, y la hoja de acordes escribe sus dos números, primero el de la tonalidad vieja: `vi = ii`.

`modulation` mira los acordes alrededor de cada cambio de tonalidad, con dos closures que dicen qué es un acorde para las dos tonalidades:

```
let new_only = |chord: &Function| {
    let chord = chord.symbol.chord();
    belongs(to, chord) && !heard_in(from, chord)
};
let common = |chord: &Function| {
    let chord = chord.symbol.chord();
    belongs(to, chord) && heard_in(from, chord)
};
```

`heard_in` cuenta los acordes prestados de la tonalidad vieja como propios, por eso Fm, prestado en Do mayor, puede ser el pivote hacia La♭ mayor, donde es vi. La **llegada** es el primer acorde que pertenece solo a la tonalidad nueva (D7 arriba). Se busca primero justo antes de la marca de tonalidad, porque los músicos marcan una tonalidad nueva donde sienten que llegó, lo que puede ser uno o dos compases

tarde, y luego después de ella. El pivote es el acorde anterior a la llegada, si pertenece a las dos:

```
change.pivot = arrival
    .checked_sub(1)
    .map(|before| range[before])
    .filter(|before| common(before))
    .map(|before| (before.start, before.symbol));
```

`checked_sub(1)` da `None` cuando la llegada es el primer acorde, así que ningún índice baja nunca de cero, y un cambio que empieza en su llegada no tiene pivote. No todo cambio de tonalidad tiene uno: una **modulación directa** va derecho a un acorde de la tonalidad nueva, como hace una canción cuando sube un semitono para el último estribillo, y un **cambio de modo**, de Do mayor a Do menor, conserva la tónica y no necesita ningún acorde que lleve hasta ahí.

Cómo se prueba

Los tests son hojas de acordes pequeñas al estilo de un libro de armonía, cada una comprobando los números y lo que dicen las descripciones:

- C Am A♭ Fm G7 C: ♭VI e iv son prestados de Do menor.
- C E7 Am D7/F♯ G7 C A7 Dm: V7/vi, V$^{6}_{5}$/V y V7/ii, cada uno yendo a su acorde. E7 yendo a F sigue siendo V7/vi, y dice que va a otro lado.
- F♯dim7, G♯dim y C♯m7♭5: vii°7/V, vii°/vi y viiø7/ii. En Autumn Leaves leído en Do mayor, F♯m7♭5 yendo a B7 conserva ♯ivø7.
- D♭/F en Do menor es N^{6}; C♯ en Do mayor no es una napolitana.
- En La menor, B7 es V7/V, o V7/v yendo a Em; A7 yendo a Dm es V7/iv; La mayor al final es una tercera de picardía.
- Sol aumentado en Do mayor no es prestado; Bdim7, cuyo La♭ es de Do menor, sí.
- C, Am, D7, G hacia Sol mayor, con la marca en el Am, en el D7 o en el G: Am es el pivote cada vez.

En `scale.rs` se comprueban los números de las tonalidades menores de las dos maneras para la menor natural, la armónica y la melódica, y los 10.368 números se escriben y se vuelven a leer. En `keys.rs` y `progression.rs`, las teclas de armonía de cada clase de tonalidad y las sugerencias después de cada grado; por ejemplo, en Do mayor:

```
// D major is V of G; A7 is V of Dm; E is V of Am; C7 is V of F.
```

Los ejercicios sobre números romanos y funciones están en Ejercicios: la teoría musical en código.

Herramientas de edición como funciones simples

Archivos: `crates/lineup-core/src/tools.rs` y `crates/lineup-app/src/piano_roll.rs`

El menú **Herramientas** del piano roll cambia muchas notas a la vez: las sube un semitono, una octava o una nota de la tonalidad, las adelanta o las atrasa, las pone en la cuadrícula o dentro de la tonalidad, invierte acordes y los convierte en arpegios. Cada herramienta es una función de `lineup_core::tools` que recibe unas notas y devuelve otras nuevas. Ninguna sabe nada de ventanas, selecciones ni deshacer, y eso es lo que las hace fáciles de leer y de probar. Este capítulo las recorre una por una; cómo eligen una herramienta el menú y las flechas está en Los editores: rack de canales, piano roll y lista de reproducción.

La música, en pocas palabras

Transportar mueve la música hacia arriba o hacia abajo sin cambiar su forma: cada intervalo sigue igual, así que una melodía subida un tono es la misma melodía, más aguda. Do, Mi, Sol subido un tono es Re, Fa♯, La: sigue siendo un acorde mayor.

Mover notas **dentro de la tonalidad** es otra cosa. Cada nota va a la nota siguiente de la *escala*, no un número fijo de semitonos más arriba, así que los intervalos cambian para encajar en la tonalidad. En Do mayor, Do, Mi, Sol subido una nota de la tonalidad es Re, Fa, La: el acorde del grado siguiente, Re menor, no Re mayor. Los compositores lo han usado siempre: una figura corta repetida una nota más arriba cada vez, sin salir de la tonalidad, es una **secuencia** (también llamada progresión o marcha), y la música barroca está llena de ellas. Copia una frase, pégala más adelante y súbela una nota de la tonalidad, y habrás escrito una.

Cuantizar pone las notas en la cuadrícula. Nadie toca exactamente sobre el pulso, y un archivo MIDI de una interpretación guarda cada pequeña diferencia; cuantizar deja las notas tan exactas como una caja de ritmos.

Una **inversión** de un acorde tiene las mismas notas con otra en el bajo, como explica Acordes: construirlos, nombrarlos e invertirlos. Un **arpegio** toca las notas de un acorde una detrás de otra, como un guitarrista que puntea un acorde o un arpa, en lugar de todas a la vez.

Notas que entran, notas que salen

Todas las herramientas tienen la misma forma. Toman prestadas las notas como un slice, `&[NoteEvent]`, y devuelven otras nuevas; no cambian nada por sí mismas. El módulo lo dice al principio:

```
//! Each tool is a plain function: it takes some notes and returns new
//! ones, and changes nothing itself. The piano roll hands the result to
//! [`Pattern::replace_notes`](crate::Pattern::replace_notes), through
//! the app's `edit`, so every tool can be undone like any other change.
```

El piano roll elige la función de cada herramienta, con un nombre para el historial de deshacer:

```
let (label, new) = match tool {
    // ...
    Tool::Transpose(semitones) => {
        ("transpose notes", tools::transpose(&targets, semitones))
    }
    Tool::InKey(steps) => (
        "move notes in key",
        tools::transpose_in_key(&targets, scale, steps),
    ),
    Tool::Shift(steps) => ("move notes", tools::shift(&targets, steps,
length)),
    Tool::Quantize => ("quantize notes", Some(tools::quantize(&targets,
length))),
    Tool::SnapToKey => (
        "snap notes to key",
        Some(tools::snap_to_scale(&targets, scale)),
    ),
    Tool::Invert { up } => ("invert chords", tools::invert(&targets, up)),
    // ...
};
```

`targets` es la selección, o todas las notas del canal si no hay nada seleccionado. Las notas nuevas reemplazan a las viejas en una sola acción `ReplaceNotes`, que es un paso de deshacer; cómo funciona eso, y por qué la selección guarda notas y no posiciones, está en Los editores: rack de canales, piano roll y lista de reproducción. Tener las herramientas fuera de la aplicación también permite que otras partes usen las mismas funciones: las pruebas de las series dodecafónicas agrupan en acordes una serie escrita con `tools::chords`, y el teclado MIDI ajusta las teclas con `tools::nearest_in_key`.

La mayoría de las herramientas construyen cada nota nueva a partir de una vieja con la **sintaxis de actualización de structs**:

```
Some(NoteEvent {
    pitch: note.pitch.transpose(semitones)?,
    ..*note
})
```

`..*note` significa «todos los demás campos, de `*note`»: el inicio, la duración, la velocidad, el `nudge` y el `trim` quedan como estaban, y solo la altura es nueva. `NoteEvent` es `Copy`, así que `*note` copia la nota vieja del slice sin quitarla de allí. Si algún día se añade un campo a `NoteEvent`, todas las herramientas lo conservan sin cambiar nada.

Agrupar notas en acordes: `chords`

Invertir y arpegiar trabajan con acordes, así que la primera herramienta los busca. Aquí un acorde es simplemente las notas que empiezan en el mismo paso:

```
pub fn chords(notes: &[NoteEvent]) -> Vec<Vec<NoteEvent>> {
    let mut sorted = notes.to_vec();
    sorted.sort_by_key(|note| (note.start, note.pitch));
    // `chunk_by` cuts the sorted list wherever the start changes.
    sorted
        .chunk_by(|a, b| a.start == b.start)
        .map(<[NoteEvent]>::to_vec)
        .collect()
}
```

Ordenar por la tupla `(note.start, note.pitch)` ordena por el inicio, y por la altura entre las notas que empiezan juntas, así que cada acorde sale con la nota más grave primero. Después `chunk_by` corta la lista en tramos, como hace el carril de acordes (ver El carril de acordes: `chunk_by`, y luego medir).

Cada tramo es un slice que toma prestado de `sorted`, y `sorted` se libera cuando la función termina. Devolver los slices los dejaría apuntando a memoria liberada, cosa que el comprobador de préstamos rechaza, así que cada uno se copia a un `Vec` propio. `<[NoteEvent]>::to_vec` es el método de slice `to_vec` nombrado como función, para que `map` lo reciba directamente en lugar de `|run| run.to_vec()`. Los ángulos hacen falta porque `[NoteEvent]` es un tipo, no un nombre: `[NoteEvent]::to_vec` no se podría analizar.

Cuando una herramienta no puede: recolectar `Option`s

Transportar un acorde cerca de lo más alto del teclado puede llevar una nota más allá de la nota MIDI 127. Dejar esa nota donde está rompería el acorde; quitarla perdería una nota. Así que la herramienta no hace ninguna de las dos cosas: rechaza el movimiento entero.

```
pub fn transpose(notes: &[NoteEvent], semitones: i32) ->
Option<Vec<NoteEvent>> {
    // Collecting an iterator of `Option`s into `Option<Vec<_>>` gives
    // `None` as soon as one item is `None`, and the whole `Vec`
otherwise.
    notes
        .iter()
        .map(|note| {
            Some(NoteEvent {
                pitch: note.pitch.transpose(semitones)?,
                ..*note
            })
        })
        .collect()
}
```

`Note::transpose` devuelve `None` para una nota fuera del teclado (ver Clases de altura, notas y su escritura). El `?` dentro del closure devuelve `None` desde el *closure*, no desde `transpose`, así que el iterador produce un `Option<NoteEvent>` por cada nota. `collect` sabe construir muchas clases de colección, y el tipo de retorno pide `Option<Vec<NoteEvent>>`: esa colección se detiene en el primer `None` y es `None` ella misma, o guarda todas las notas. `Result` funciona igual: recolectado en un `Result<Vec<_>, E>`, se detiene en el primer error.

El piano roll no hace nada cuando una herramienta devuelve `None`, así que pulsar ↑ sobre un acorde con una nota en la fila de arriba simplemente no lo mueve:

```
let Some(new) = new.filter(|new| fits_view(new)) else {
    return;
};
```

`Option::filter` convierte `Some` en `None` cuando la prueba falla; aquí la prueba es que todas las notas quepan en las filas que muestra el piano roll, que son menos que las 128 notas de MIDI.

Moverse en el tiempo: `checked_add_signed` y `then_some`

← y → mueven las notas un paso antes o después, y con Mayús cuatro pasos, un pulso de 4/4. Los inicios son `usize`, que no puede ser negativo, mientras que el movimiento puede ir en los dos sentidos, así que es un `isize`:

```
pub fn shift(notes: &[NoteEvent], steps: isize, pattern_length: usize) ->
Option<Vec<NoteEvent>> {
    notes
        .iter()
        .map(|note| {
            let start = note.start.checked_add_signed(steps)?;
            (start < pattern_length).then_some(NoteEvent { start, ..*note
})
        })
        .collect()
}
```

`usize::checked_add_signed` suma un número con signo a uno sin signo y devuelve `None` si el resultado quedaría por debajo de cero, así que una nota en el paso 0 no puede adelantarse. `then_some` convierte el `bool` en un `Option`: `Some` de la nota movida si todavía empieza dentro del patrón, `None` si empezaría en su final o después. En los dos casos la selección entera se queda donde estaba, con el mismo `collect` de antes.

De vuelta a la cuadrícula: `quantize`

Cuantizar a los pasos lleva el inicio y el final de cada nota al paso más cercano. Una nota guarda su lugar exacto como un paso más un `nudge` y un `trim`, en ticks, 24 por paso (ver El proyecto: patrones, canales, pistas, clips y marcas), y el redondeo está en la propia nota:

```
/// The note moved to the nearest steps, both its start and its end:
/// what *quantizing* does. A note shorter than a step keeps one.
pub fn quantized(self) -> NoteEvent {
    let nearest = |tick: usize| (tick + TICKS_PER_STEP / 2) /
TICKS_PER_STEP;
    let start = nearest(self.start_tick());
    let end = nearest(self.end_tick()).max(start + 1);
    NoteEvent {
        velocity: self.velocity,
        ..NoteEvent::new(self.pitch, start, end - start)
    }
}
```

La división entera redondea hacia abajo, así que sumar antes medio paso redondea al paso más cercano. `NoteEvent::new` crea una nota en la cuadrícula, sin `nudge` ni `trim`, y la actualización de struct lo toma todo de ella menos la velocidad. La herramienta añade una regla: una nota que ahora empezaría en el final del patrón empieza en su último paso.

```
pub fn quantize(notes: &[NoteEvent], pattern_length: usize) ->
Vec<NoteEvent> {
    let last = pattern_length.saturating_sub(1);
    notes
        .iter()
        .map(|note| {
            let quantized = note.quantized();
            let start = quantized.start.min(last);
            NoteEvent {
                start,
                length: quantized.end() - start,
                ..quantized
            }
        })
        .collect()
}
```

Cuantizar es una elección, no algo que la aplicación haga por su cuenta, porque puede cambiar la música. Los tresillos de corcheas empiezan cada 32 ticks, a un paso y un tercio de distancia, y ninguna cuadrícula de semicorcheas los contiene: cuantizados, caen en los pasos 0, 1 y 3 y ya no son regulares. Por eso las notas importadas de un archivo MIDI conservan su `nudge` hasta que lo pidas. (Las notas que grabas se ponen en la cuadrícula al grabarlas, con una función propia: ver MIDI: mensajes, dispositivos, grabación y archivos.)

Quedarse en la tonalidad

Dos herramientas trabajan con la tonalidad: una mueve las notas una nota de la tonalidad hacia arriba o hacia abajo, la otra lleva dentro de la tonalidad todas las notas que están fuera. Las dos usan la tonalidad en la que está la canción donde suena el patrón, después de las marcas de tonalidad (ver Escalas, modos y tonalidades).

Una nota de la tonalidad más arriba: `find_map`

La nota siguiente de la tonalidad por encima de una nota es la primera nota, un semitono más arriba, luego dos, y así, que esté en la escala:

```
pub fn step_in_key(note: Note, scale: Scale, up: bool) -> Option<Note> {
    let direction = if up { 1 } else { -1 };
    (1..=12).find_map(|distance| {
        note.transpose(direction * distance)
            .filter(|candidate| scale.contains(candidate.pitch_class()))
    })
}
```

`find_map` llama al closure para cada distancia y se detiene en el primer `Some`. El closure transporta, cosa que puede fallar en el borde del teclado, y `filter` se queda con la nota solo si la escala la contiene. Doce semitonos alcanzan todas las clases de altura, así que si la escala tiene alguna nota, se encuentra una: en el peor caso la misma nota una octava más allá, cuando la escala no tiene otra. Un rango es perezoso, y `find_map` se detiene en el primer acierto, así que un paso en Do mayor prueba una o dos distancias, no doce.

Mover varias notas de la tonalidad es ese paso, repetido:

```
pub fn transpose_in_key(notes: &[NoteEvent], scale: Scale, steps: i32) ->
Option<Vec<NoteEvent>> {
    notes
        .iter()
        .map(|note| {
            let mut pitch = note.pitch;
            for _ in 0..steps.unsigned_abs() {
                pitch = step_in_key(pitch, scale, steps > 0)?;
            }
            Some(NoteEvent { pitch, ..*note })
        })
        .collect()
}
```

`unsigned_abs` da el tamaño de `steps` como número sin signo, así que el bucle se repite tantas veces bajando como subiendo. Una nota de fuera de la tonalidad cae en la nota de la tonalidad más cercana en la dirección del movimiento: Do♯ subido un paso en Do mayor es Re. **Alt+**↑ y **Alt+**↓ usan esta herramienta.

La nota de la tonalidad más cercana: ajustar

Ajustar mira en los dos sentidos a la vez:

```
pub fn nearest_in_key(note: Note, scale: Scale) -> Note {
    if scale.contains(note.pitch_class()) {
        return note;
    }
    (1..=6)
        .find_map(|distance| {
            [-distance, distance].into_iter().find_map(|semitones| {
                note.transpose(semitones)
                    .filter(|candidate|
scale.contains(candidate.pitch_class()))
            })
        })
        .unwrap_or(note)
}
```

Para cada distancia, el `find_map` de dentro prueba la nota de abajo antes que la de arriba, así que cuando dos notas de la tonalidad están igual de cerca, gana la más grave: en Do mayor, Do♯ se ajusta a Do, Fa♯ a Fa y Si♭ a La. Hace falta una regla, porque ninguna de las dos es más correcta, y esta es fácil de recordar. Seis semitonos hacia arriba o hacia abajo alcanzan las doce clases de altura, así que cualquier escala con alguna nota se encuentra; solo en los extremos del teclado pueden fallar los dos intentos, y `unwrap_or(note)` deja entonces la nota donde está. Ajustar nunca rechaza como lo hace un movimiento; en el peor caso una nota se queda quieta.

`snap_to_scale` la aplica a muchas notas, para **Ajustar a la tonalidad** del menú Herramientas:

```
pub fn snap_to_scale(notes: &[NoteEvent], scale: Scale) -> Vec<NoteEvent>
{
    notes
        .iter()
        .map(|note| NoteEvent {
            pitch: nearest_in_key(note.pitch, scale),
            ..*note
        })
        .collect()
}
```

Las mismas dos funciones hacen el botón **Ajustar a la escala** del piano roll: una nota que dibujas cae en `nearest_in_key` de la fila en la que hiciste clic, y un grupo que arrastras se ajusta con `snap_to_scale` una vez que se ha movido (ver Mover un grupo). Una tecla tocada en el teclado de escribir o en un teclado MIDI se ajusta del mismo modo (ver MIDI: mensajes, dispositivos, grabación y archivos).

Inversiones

Invertir los acordes hacia arriba sube la nota más grave de cada acorde por encima de la más aguda; **Invertir los acordes hacia abajo** baja la más aguda por debajo de la más grave. Las dos mueven una nota por octavas hasta que pasa el otro extremo del acorde:

```
/// `note` moved up by octaves until it is above `ceiling`.
fn above(note: Note, ceiling: Note) -> Option<Note> {
    let mut note = note;
    while note <= ceiling {
        note = note.transpose(12)?;
    }
    Some(note)
}
```

«Hasta que esté por encima», no «una octava más arriba»: en Do, Mi, Sol, Do, el Do grave tiene que subir dos octavas para quedar por encima del Do agudo. El `?` se rinde si la nota saldría del teclado.

```
pub fn invert(notes: &[NoteEvent], up: bool) -> Option<Vec<NoteEvent>> {
    let mut result = Vec::with_capacity(notes.len());
    for mut chord in chords(notes) {
        if let (Some(&lowest), Some(&highest)) = (chord.first(),
chord.last())
            && chord.len() >= 2
        {
            if up {
                chord[0].pitch = above(lowest.pitch, highest.pitch)?;
            } else {
                let last = chord.len() - 1;
                chord[last].pitch = below(highest.pitch, lowest.pitch)?;
            }
            chord.sort_by_key(|note| note.pitch);
        }
        result.extend(chord);
    }
    Some(result)
}
```

`chords` devuelve cada acorde con la nota más grave primero, así que `first` y `last` son el bajo y la nota de arriba. Devuelven `Option`s, porque un `Vec` vacío no tiene ninguna de las dos, y hacer match sobre una tupla de ambas las toma en un solo patrón. El `&` de `Some(&lowest)` copia la nota de la referencia que devuelve `first`, así que `lowest` y `highest` son `NoteEvent`s normales y no préstamos de `chord`. El `&&` forma una **cadena de let** (*let chain*): el cuerpo se ejecuta solo si existen las dos notas *y* el acorde tiene dos notas o más, así que una nota sola se deja como está. `for`

`mut chord` hace de cada acorde una variable que se puede cambiar, sin un `let mut` dentro del bucle.

`inversion`, una hermana para una lista de alturas, da un acorde en cualquier inversión de una vez. El sello de acordes del piano roll la usa para el menú de inversión que está junto a **Dibujar** (ver Acordes: construirlos, nombrarlos e invertirlos):

```
pub fn inversion(pitches: &[Note], times: usize) -> Vec<Note> {
    let mut notes = pitches.to_vec();
    notes.sort();
    for _ in 0..times % notes.len().max(1) {
        let (Some(&lowest), Some(&highest)) = (notes.first(),
notes.last()) else {
            break;
        };
        let Some(moved) = above(lowest, highest) else {
            break; // no room above: leave the chord as it is
        };
        notes.remove(0);
        notes.push(moved);
    }
    notes
}
```

Un acorde de tres notas tiene tres posiciones, así que girarlo tres veces vuelve al estado fundamental: `times % notes.len()` se salta las vueltas completas. `.max(1)` evita que una lista vacía divida por cero, lo que provocaría un pánico. El `let ... else` toma los dos extremos o sale del bucle. A diferencia de `invert`, esta no rechaza cuando una nota no puede subir más: un sello cerca de lo más alto del teclado sigue dibujando un acorde, en la última inversión que cabe.

Arpegios: `cycle`, `step_by` y `zip`

Arpegiar los acordes toca las notas de cada acorde una detrás de otra, mientras duraba el acorde. El menú Herramientas elige el orden (**arriba**, **abajo**, **arriba y abajo**) y cuánto dura cada nota (**1 paso**, **2 pasos**, **1 pulso**). El orden es un enum:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default)]
pub enum ArpeggioOrder {
    #[default]
    Up,
    Down,
    UpDown,
}
```

`#[derive(Default)]` en un enum necesita que se le diga qué variante es la predeterminada, con `#[default]`. Cada orden sabe qué nota del acorde viene después, contando desde la más grave, para siempre:

```
fn indices(self, count: usize) -> impl Iterator<Item = usize> {
    let once: Vec<usize> = match self {
        ArpeggioOrder::Up => (0..count).collect(),
        ArpeggioOrder::Down => (0..count).rev().collect(),
        ArpeggioOrder::UpDown => (0..count)
            .chain((1..count.saturating_sub(1)).rev())
            .collect(),
    };
    // `cycle` repeats an iterator forever. It is lazy, so "forever"
    // costs nothing: the caller takes only as many as it needs.
    once.into_iter().cycle()
}
```

Para tres notas, arriba y abajo es 0, 1, 2, 1, y luego otra vez desde 0: la nota de arriba y la de abajo se tocan una vez en cada vuelta, nunca dos veces seguidas. `saturating_sub` se detiene en cero en lugar de pasar por debajo, así que `count - 1` no puede desbordar el `usize` para un acorde vacío. El tipo de retorno `impl Iterator<Item = usize>` dice «algún iterador de `usize`» sin nombrar su tipo, que aquí sería `Cycle<std::vec::IntoIter<usize>>`; a quien lo llama solo le hace falta saber que puede recorrerlo.

```rust
pub fn arpeggiate(notes: &[NoteEvent], order: ArpeggioOrder, rate: usize)
-> Vec<NoteEvent> {
    let rate = rate.max(1);
    let chords = chords(notes);
    let mut result = Vec::new();
    for (index, chord) in chords.iter().enumerate() {
        if chord.len() < 2 {
            result.extend(chord);
            continue;
        }
        let start = chord[0].start;
        let mut end =
chord.iter().map(NoteEvent::end).max().unwrap_or(start + 1);
        if let Some(next) = chords.get(index + 1) {
            end = end.min(next[0].start);
        }
        // `step_by` takes every `rate`-th step; `zip` pairs each of those
        // steps with the next note of the order, and stops when the steps
        // run out (the order never does).
        for (step, pick) in
(start..end).step_by(rate).zip(order.indices(chord.len())) {
            result.push(NoteEvent {
                start: step,
                length: rate.min(end - step),
                ..chord[pick]
            });
        }
    }
    result
}
```

El arpegio dura hasta que termina la nota más larga del acorde, pero se detiene donde empieza el siguiente acorde o nota sola, así que nunca suena encima. `map(NoteEvent::end)` pasa el método como función, como `to_vec` más arriba; `max` devuelve un `Option`, porque una lista vacía no tiene elemento mayor. `zip` se detiene cuando se acaba cualquiera de los dos lados, y el orden, que no termina nunca, no se acaba, así que los pasos deciden cuántas notas hay. Cada nota se construye `..chord[pick]`, a partir de la nota del acorde que toca, así que conserva la velocidad de esa nota: una nota de arriba fuerte sigue siendo fuerte en el arpegio. La última nota se corta si el acorde termina en medio de ella (`rate.min(end - step)`).

`result.extend(chord)` añade una nota sola tal como está. `chord` es un `&Vec<NoteEvent>`, y un `Vec` de un tipo `Copy` se puede extender a partir de referencias, copiando cada elemento.

Oír lo que se movió

Después de una herramienta que cambia alturas, el piano roll toca el primer acorde del resultado, para que oigas adónde fue sin poner en marcha la canción:

```
if matches!(
    tool,
    Tool::Transpose(_) | Tool::InKey(_) | Tool::Invert { .. }
) && let Some(first) = tools::chords(&new).into_iter().next()
{
    actions.push(RollAction::Preview {
        channel: self.channel,
        pitches: first.iter().map(|note| note.pitch).collect(),
    });
}
```

`matches!` pregunta si un valor encaja en un patrón y da un `bool`; `_` y `{ .. }` aceptan cualquier valor dentro. Es otra vez una cadena de let: la escucha previa se envía solo para esas herramientas *y* si hay un primer acorde. `Preview` es una acción más que la aplicación atiende después de dibujar.

Cómo se prueba

Como las herramientas son funciones simples, las pruebas también lo son: unas notas entran, salen las notas esperadas, sin ventana. Una función auxiliar crea una nota a partir de un número MIDI, un inicio y una duración:

```
fn note(midi: u8, start: usize, length: usize) -> NoteEvent {
    NoteEvent::new(Note::new(midi), start, length)
}
```

y las pruebas se leen como la música:

```
#[test]
fn moving_in_key_changes_the_chord() {
    // C major one step up in C major is D minor: D, F, A.
    let c_major = [note(60, 0, 4), note(64, 0, 4), note(67, 0, 4)];
    let up = transpose_in_key(&c_major, C_MAJOR, 1).unwrap();
    assert_eq!(pitches(&up), [62, 65, 69]);
    // Two steps down from C, E, G: A, C, E (A minor).
    let down = transpose_in_key(&c_major, C_MAJOR, -2).unwrap();
    assert_eq!(pitches(&down), [57, 60, 64]);
    // ...
}
```

Las demás comprueban que una sola nota demasiado aguda estropea todo el transporte, que un desplazamiento se queda dentro del patrón, que los tresillos de corcheas se cuantizan a los pasos 0, 1 y 3, que ajustar mueve solo las notas de fuera de la tonalidad (y que en un empate gana la más grave), que invertir tres veces vuelve al estado fundamental, y que un arpegio se detiene en el acorde siguiente y conserva la velocidad de cada nota. Las pruebas de `change_key`, la función que usan los menús **Tonalidad**, que está en el mismo archivo, están en Escalas, modos y tonalidades.

Los ejercicios sobre las herramientas están en Ejercicios: la teoría musical en código.

Oír la armonía: acordes y tonalidades a partir de las notas

Archivos: `crates/lineup-core/src/harmony/hearing.rs`, `crates/lineup-core/src/harmony/key_finding.rs`, `crates/lineup-core/src/library/listen.rs`, `crates/lineup-core/src/project.rs`, `crates/lineup-app/src/app/key_finder.rs` y `crates/lineup-core/tests/chord_reading.rs`

Un patrón guarda notas, no acordes, y un archivo MIDI importado o una canción que tocaste no tienen acordes escritos en ningún sitio. Aun así, muchas partes de la aplicación necesitan la armonía: **Escribir los acordes que toca la canción** en la hoja de acordes, la fila **Todo** del carril de acordes del piano roll, la tecla de armonía que se ilumina mientras suena la canción, la comprobación de la conducción de voces, un estilo que vuelve a tocar una canción entera con otro ritmo, y la tonalidad de un archivo MIDI sin armaduras. Todas escuchan las notas como lo hace un músico de oído. Este capítulo trata de cómo: primero los acordes, después las tonalidades.

`analyze`, en Acordes: construirlos, nombrarlos e invertirlos, nombra las notas que suenan en un momento, y eso está bien para un acorde sostenido en un teclado. La música que se mueve necesita más.

La música, en pocas palabras

Nombrar las notas de un solo momento falla de cinco maneras comunes:

- Un **arpegio** toca su acorde una nota cada vez: Do, luego Mi, luego Sol. En cada momento suena una sola nota, y una nota no nombra ningún acorde.
- Una **melodía** pasa por notas que no son del acorde. Sobre un acorde de Do mayor, una melodía en Re hace que el momento sea Do, Re, Mi, Sol, que se lee como Cadd9 aunque la armonía sea un Do mayor sin más.
- Un **bajo caminante** (*walking bass*) va de un acorde a otro por grados: Do, Si♭, La camino de C a A7. El Si♭ debajo de Do, Mi y Sol forma C7 por un momento, aunque nadie toque C7.
- Una orquesta puede tocar **dos acordes a la vez**, uno sobre otro: un **poliacorde**. En los Augurios primaverales de *La consagración de la primavera* de Stravinsky, las cuerdas graves tocan Fa♭ mayor (Fa♭, La♭, Do♭) mientras arriba suena E♭7 (Mi♭, Sol, Si♭, Re♭). En *Petrushka* los clarinetes tocan Do mayor sobre Fa♯ mayor. Ningún cifrado de un solo acorde contiene ninguno de los dos.

- Algunas armonías **no son superposiciones de terceras**. La feria de *Petrushka* empieza con una figura de la flauta en Re, Mi, Sol y La, que se apilan por cuartas: Mi, La, Re, Sol. Llamar acorde a eso le impone un nombre.

Una **tonalidad** se oye del mismo modo, a lo largo del tiempo. En Do mayor, Do, Mi y Sol vuelven más que ninguna, luego las demás teclas blancas, y las negras casi nunca. La menor usa las mismas teclas blancas, pero mandan La, Do y Mi, y Sol♯ aparece en cada cadencia. Una pieza **modula** cuando pasa a una tonalidad nueva y se queda en ella; un solo acorde de otra tonalidad, como D7 llevando a G en una canción en Do mayor (una *dominante secundaria*), solo *toniciza* Sol un momento y no es un cambio de tonalidad.

Escuchar un tramo, no un momento

La escucha de acordes escucha un **tramo** de música, un pulso, medio compás o un compás, y pesa cada nota según cuánto suena en él. Una nota escuchada es un `HeardNote`, con tiempos en pasos desde el inicio del tramo:

```
pub struct HeardNote {
    pub pitch: Note,
    /// When it starts, in steps from the start of the stretch: below 0
    /// for a note that started before it.
    pub start: f32,
    /// When it ends, in steps from the start of the stretch: beyond its
    /// length for a note that goes on after it.
    pub end: f32,
    /// Which part plays it, such as a channel: an instrument's notes are
    /// a layer of their own when two chords sound at once.
    pub part: usize,
}
```

Los tiempos son `f32`, no pasos enteros, porque una nota puede empezar entre dos pasos (su `nudge`, ver El proyecto: patrones, canales, pistas, clips y marcas). Las notas también pueden quedar del todo fuera del tramo. Esas no pesan nada, pero cuentan qué hace cada parte alrededor, como muestra la sección siguiente.
`Project::heard_notes` reúne las notas de todos los canales que tocan notas (los sintetizadores y los canales MIDI Out que no están silenciados; la batería y los samplers no) entre dos pasos de la canción.

Sumar cuánto suena cada una de las doce clases de altura da un **perfil**:

```
struct Profile {
    weight: [f32; 12],
    total: f32,
    /// How long the pitch class heard longest sounds.
    loudest: f32,
}
```

Un arpegio de Do mayor a lo largo de un compás es Do, Mi y Sol sonando cada uno un tercio del compás: las mismas proporciones que un acorde sostenido, así que el arpegio se oye como su acorde. Esa sola idea resuelve el primer problema de la lista. Los demás necesitan saber qué hace cada parte.

Qué hace cada parte

Cuánto pesa una nota depende de la parte que la toca. `weights` distingue cuatro papeles:

Papel	Cómo se reconoce	Cuánto pesan sus notas
Acordes	en ella suenan juntas tres notas distintas o más, o toca acordes nota a nota	entero
Línea de bajo	una nota cada vez, todas por debajo de los acordes	entero si los acordes tienen la nota; 0,15 si no, una nota de paso
Melodía	cualquier otra parte, cuando alguna parte toca acordes	un cuarto
Capa	una parte que sería una melodía, pero cuyas notas son tres o más de una misma tríada o acorde de séptima, ninguna de ellas de los acordes	entero

El final de `weights` aplica la tabla, nota por nota:

```
let weight = if tune {
    w * TUNE
} else if lowest_chord.is_some()
    && below
    && !chord_notes.contains(note.pitch.pitch_class())
{
    w * PASSING_BASS
} else {
    w
};
```

`w` es cuánto suena la nota dentro del tramo. Cuando ninguna parte toca acordes, como en una orquesta donde cada instrumento toca una línea, `lowest_chord` es `None`, nadie es una melodía y cada línea cuenta entera.

La fila de la **capa** está ahí por *La consagración de la primavera*. Su ostinato Re♭, Si♭, Mi♭, Si♭ es una sola línea, pero deletrea E♭7 contra los arpegios de Do mayor y Mi menor que suenan debajo: una armonía propia, no una melodía.

Dos preguntas más necesitan los compases de alrededor, porque un solo pulso no puede contestarlas:

- ¿Una parte **mueve sus acordes junto con una melodía**, todas las voces en el mismo sentido por grado o por tercera, como en las últimas exposiciones del *Bolero* de Ravel? Entonces sus acordes colorean la melodía y la armonía es lo que pasa debajo, así que la parte cuenta como melodía. Eso es `moves_in_parallel`; dos acordes no bastan para mostrar un movimiento paralelo, pero una frase sí.
- ¿Una parte **toca acordes nota a nota**, como un bajo de Alberti o los tresillos de la sonata *Claro de luna*? Notas cortas, casi todas saltos de tercera menor o más, sobre tres clases de altura por lo menos: `breaks_chords`. Entonces la parte toca acordes tanto como una que los golpea enteros.

Una puntuación para cada acorde

Cada acorde, doce fundamentales por cada clase de acorde que conoce la aplicación, recibe una puntuación según lo bien que explica el perfil:

```
fn chord_fit_with(chord: Chord, profile: &Profile, also: PitchSet) ->
(f32, bool) {
    if profile.total <= 0.0 {
        return (f32::NEG_INFINITY, false);
    }
    let explained = PitchSet::from_bits(chord.pitch_set().bits() |
also.bits());
    let inside: f32 = explained.iter().map(|p| profile.of(p)).sum();
    let outside = profile.total - inside;
    let full = PRESENT * profile.loudest;
    let mut missing = 0.0;
    let mut core = true;
    for (&interval, &tone) in
chord.quality.intervals().iter().zip(chord.quality.tones()) {
        let pitch_class = chord.root.transpose(i32::from(interval));
        let presence = (profile.of(pitch_class) / full).min(1.0);
        let loss = if tone == 5 && interval % 12 == 7 {
            MISSING_FIFTH
        } else {
            MISSING
        };
        missing += loss * (1.0 - presence);
        if matches!(tone, 1..=4) && presence < 0.5 {
            core = false;
        }
    }
    let extra = chord.quality.intervals().len().saturating_sub(3) as f32;
    let fit = (inside - OUTSIDE * outside) / profile.total - missing -
SIZE * extra;
    (fit, core)
}
```

- `inside` es cuánto del sonido explica el acorde, y `outside` lo que deja fuera, que cuenta una vez y media (`OUTSIDE = 1.5`): una nota equivocada es una prueba más fuerte que una acertada. Dividido por el total, un acorde que lo explica todo puntúa 1.
- `missing` es una pequeña pérdida por cada nota del acorde que apenas se oye: 0,15 por una nota que falta y solo 0,05 por una quinta justa que falta, porque los músicos la omiten sin problema. Una nota que suena al menos el 35% de lo que suena la más fuerte (`PRESENT`) cuenta como presente del todo.
- `extra` cuenta las notas más allá de una tríada, 0,06 cada una. Un acorde de decimotercera contiene casi todas las notas de una escala, así que sin este coste lo explicaría todo; gana solo cuando sus notas suenan de verdad.
- `core` dice si la fundamental y la tercera (o la nota que hace sus veces en un acorde sus) se oyen de verdad. Un acorde sin ellas se descarta.

`PitchSet` guarda un conjunto de clases de altura como los bits de un solo `u16` (ver Clases de altura, notas y su escritura). `|` sobre los bits es la unión de dos conjuntos,

así que `explained` es «las notas del acorde y las de `also`» en una sola instrucción. `also` está vacío para un acorde solo; en un poliacorde guarda las notas del otro acorde.

El bajo da el último toque:

```
fn bass_fit(chord: Chord, bass: Option<PitchClass>) -> f32 {
    match bass.map(|bass| (bass, chord.tone_of(bass))) {
        Some((bass, _)) if bass == chord.root => BASS_ROOT,
        Some((_, Some(3 | 5 | 7))) | None => 0.0,
        Some(_) => -BASS_OUTSIDE,
    }
}
```

Una fundamental en el bajo gana 0,06; una tercera, quinta o séptima es una inversión y no cuesta nada; cualquier otra nota cuesta 0,06. El brazo del medio muestra dos herramientas de los patrones: `3 | 5 | 7` dentro de `Some(...)` encaja con cualquiera de los tres valores, y `| None` une al brazo un segundo patrón completo.

Un bajo caminante, paso a paso

Un piano sostiene Do, Mi y Sol durante dos pulsos (ocho pasos) mientras el bajo camina Do y luego Si♭, bajando hacia La para el acorde siguiente. El bajo toca por debajo de los acordes y Si♭ no es una de sus notas, así que pesa 4 × 0,15 = 0,6. El perfil es Do 12 (piano y bajo), Mi 8, Sol 8 y Si♭ 0,6, 28,6 en total; `loudest` queda limitado a la duración del tramo, 8, para que una nota doblada en dos partes no haga parecer débiles a las demás.

- **C** lo explica todo menos el Si♭: (28 − 1,5 × 0,6) / 28,6 = 0,948, sin nada que falte, más 0,06 por tener su fundamental en el bajo: **1,008**.
- **C7** lo explica todo, 1,0. Pero su Si♭ suena 0,6 de los 2,8 que cuentan como presencia completa, lo que cuesta 0,15 × (1 − 0,21) = 0,12, y su cuarta nota cuesta 0,06. Con el premio del bajo: **0,882**.

Gana C, como diría un cifrado, y C7 ni siquiera se guarda como segunda lectura (para eso tendría que quedar a 0,12 o menos del mejor). Si la nota de paso pesara 0,35, C seguiría ganando, por 0,941 a 0,925, pero C7 aparecería como alternativa cercana.

Quedarse con los tres mejores

Solo los tres mejores acordes se convierten en lecturas, así que no hace falta ordenarlos todos. Un vector corto se mantiene ordenado a medida que llegan los

acordes:

```
let at = found.partition_point(|&(_, other): &(Chord, f32)| other >= fit);
if at < MOST_READINGS {
    found.insert(at, (chord, fit));
    found.truncate(MOST_READINGS);
}
```

`partition_point` hace una búsqueda binaria del primer lugar donde la condición pasa a ser falsa: el primer acorde que encaja peor que el nuevo. Un acorde que encaja exactamente igual va detrás de los que ya están, así que el resultado es el mismo que el de un ordenamiento estable, con mucho menos trabajo.

Dos acordes a la vez

Si ningún acorde solo encaja bien (una puntuación por debajo de 0,7, `ONE_CHORD`), la escucha prueba con dos. `best_polychord` divide las notas en una capa inferior y otra superior de todas las maneras que tienen sentido:

- **por registro**: todo lo que está por debajo de una altura es la capa inferior. Se salta una división que cortaría una línea, porque una línea que va de La hasta Sol es un arpegio de A7, no la parte de arriba de Sol mayor;
- **por instrumento**: las partes ordenadas de la que suena más grave a la más aguda, cortadas entre dos cualesquiera.

Cada capa tiene que formar una tríada mayor o menor, o un acorde de séptima mayor, menor o de dominante, con tres de sus notas oídas y su fundamental entre ellas. Dos acordes más grandes a la vez explicarían cualquier nota. Después la pareja tiene que pasar cuatro pruebas:

1. Las fundamentales son distintas, y el acorde de arriba añade al menos dos notas propias. C sobre La menor es solo Am7.
2. Entre los dos acordes no forman **una sola tonalidad**. Fa mayor sobre Sol mayor usa solo las notas de Do mayor: eso es G con oncena, una sola armonía en Do, no dos tonalidades a la vez.
3. Los dos acordes **suenan juntos** durante al menos el 80% del tiempo en que suena algo en el tramo (`covered`). Do mayor y luego G7 es una progresión, no un poliacorde. El oído sostiene las notas de un arpegio, o de un acorde en staccato golpeado una y otra vez, así que un hueco de dos pasos o menos cuenta como sonando.
4. La pareja, menos 0,2 por necesitar dos acordes (`POLYCHORD`), puntúa mejor que el mejor acorde solo.

La segunda prueba son unas pocas líneas:

```
fn in_one_key(notes: PitchSet) -> bool {
    let major = PitchSet::from_semitones(&[0, 2, 4, 5, 7, 9, 11]);
    (0..12).any(|tonic| notes.is_subset_of(major.transpose(tonic)))
}
```

`transpose` rota los bits, así que esto prueba las doce escalas mayores, y con ellas sus modos. ¿Por qué solo escalas mayores? Porque el propio acorde de los Augurios encaja en una escala **menor armónica**. Sus siete notas, Fa♭, Sol, La♭, Si♭, Do♭, Re♭ y Mi♭, son exactamente La♭ menor armónica. Si la prueba contara la menor armónica como una tonalidad, el acorde habría quedado descartado.

El acorde de Petrushka pasa del mismo modo. Do mayor y Fa♯ mayor están a un tritono y no comparten ninguna nota, y entre los dos tienen seis notas, Do, Do♯, Mi, Fa♯, Sol y La♯, que ninguna escala mayor contiene. Las dos tríadas salen de la escala **octatónica**, Do, Re♭, Mi♭, Mi, Fa♯, Sol, La, Si♭, que alterna semitonos y tonos.

Un conjunto cuando ningún acorde encaja

Por último, las notas se puntúan como un **conjunto de clases de altura** (ver Conjuntos de clases de altura, y una armonía con cuatro nombres). Un conjunto explica todas las notas que se oyen lo bastante como para contar, así que siempre encaja, pero paga 0,35 por renunciar a un nombre de acorde. Una superposición de cuartas paga solo 0,1, porque desde Debussy es una armonía por derecho propio, tan clara al oído como una tríada:

```
/// The sets that are stacks of fourths: C F Bb (3-9, the same notes as a
/// sus chord), C F Bb Eb (4-23) and C F Bb Eb Ab (5-35, the pentatonic
/// scale), as their prime forms.
const QUARTAL_SETS: [&[u8]; 3] = [&[0, 2, 7], &[0, 2, 5, 7], &[0, 2, 4, 7,
9]];
```

El Re, Mi, Sol y La de la flauta al principio de la feria de *Petrushka* es 4-23, así que se nombra como ese conjunto, en lugar de con un acorde forzado como D6sus2.

La mejor lectura, y las que le siguen de cerca

Un `Hearing` guarda todas las lecturas que quedan a 0,12 o menos de la mejor (`CLOSE`), tres como máximo:

```rust
pub struct Hearing {
    readings: Vec<Reading>,
    /// The pitch classes heard enough to be part of the harmony, without
    /// the passing notes.
    notes: PitchSet,
    /// How long each pitch class sounds, weighed.
    profile: [f32; 12],
}
```

`best()` es la primera lectura, o `Harmony::Silence` si no hay ninguna, y `alternatives()` el resto:

```rust
pub fn alternatives(&self) -> &[Reading] {
    self.readings.get(1..).unwrap_or_default()
}
```

`get(1..)` devuelve `None` para un vector vacío, donde `&self.readings[1..]` provocaría un pánico, y `unwrap_or_default` convierte ese `None` en un slice vacío, el valor por defecto de `&[Reading]`.

Los campos son privados, así que un `Hearing` solo se puede crear escuchando: nada fuera del módulo puede construir uno con las lecturas desordenadas. La aplicación muestra las lecturas como muestra sus demás indicaciones, la más probable más brillante y las otras más tenues:

- Junto a la sugerencia de las teclas de armonía, una línea dice qué se escucha cuando no es simplemente un acorde: «Se escucha: Eb7|Fb, dos acordes a la vez», «Se escuchan las notas 4-23», o un acorde seguido de «o» y sus rivales cercanos. En las teclas de armonía, cada acorde de un poliacorde ilumina su tecla.
- La línea de estado del piano roll, para el paso que está bajo el puntero, añade las otras lecturas entre paréntesis: «(o C6)».

Una última regla protege a una melodía sola. *Para Elisa* empieza con Mi, Re♯, Mi, Re♯, Mi, Si, Re, Do y nada debajo: una melodía, no un acorde, aunque sumadas sus notas se ganarían algún nombre de acorde. Así que cuando una sola línea suena sola y su mejor lectura encaja menos de 0,85 (`ALONE`), la escucha no nombra ninguna armonía, y sigue el acorde de antes.

Cortar un compás en acordes

Escribir los acordes que toca la canción llena la hoja de acordes a partir de las notas. También se ejecuta sola cuando llega un estilo de banda a una canción con la hoja vacía, porque la banda toca a partir de la hoja. Recorre la canción compás a compás y pregunta: ¿dónde hay que cortar este compás? Un compás de 4/4 tiene líneas de pulso en 0, 1, 2, 3 y 4, y dos cualesquiera de ellas forman un tramo: diez tramos, desde un solo pulso hasta el compás entero. La mejor manera de cortar se encuentra con **programación dinámica**:

```
// best[i]: the best score for the beats before line i, where its
// last stretch starts, and how that stretch is heard.
let mut best: Vec<(f32, usize, Option<Hearing>)> = vec![(0.0, 0, None)];
for i in 1..lines.len() {
    let mut here = (f32::NEG_INFINITY, 0, None);
    for j in 0..i {
        let hearing = heard(lines[j], lines[i]);
        let cut = match lines[j] {
            line if line == bar.start => 0.0,
            line if line == half => CUT_HALF_BAR,
            _ => CUT_BEAT,
        };
        let total = best[j].0 + worth(&hearing) * (i - j) as f32 - cut;
        if total > here.0 {
            here = (total, j, Some(hearing));
        }
    }
    best.push(here);
}
```

La idea: la mejor manera de cortar los primeros `i` pulsos termina con algún último tramo, de la línea `j` a la línea `i`. Sea cual sea `j`, los cortes anteriores tienen que ser la mejor manera de cortar los primeros `j` pulsos, o habría una respuesta mejor. Así que `best[i]` se calcula a partir de los `best[j]` anteriores, que ya se conocen, probando cada `j`. Un compás de cuatro pulsos se puede cortar solo de $2^3 = 8$ maneras, pero el mismo tramo aparece en muchas de ellas, y así cada uno de los diez tramos se escucha una sola vez. El oyente de la biblioteca, más abajo, corta un patrón entero del mismo modo, donde probar todas las maneras sería imposible: un patrón de 64 pulsos se puede cortar de 2^{63} maneras, y el bucle hace unas $64 \times 64 / 2$ preguntas.

Cada tramo gana su puntuación una vez por cada pulso que cubre (`worth`), y cada corte cuesta un poco: 0,08 a mitad de compás, 0,12 en cualquier otro pulso. Así un compás se divide solo donde dos acordes encajan de verdad mejor que uno, y con más facilidad a la mitad que en un pulso débil, como cambian los acordes en la música real. `f32::NEG_INFINITY` es menor que cualquier otro número de coma

flotante, así que el primer total real siempre lo supera, sin un caso especial. Recorrer los `j` hacia atrás desde el final da los cortes, el último primero, y `reverse` los pone en orden.

`worth` da a un silencio la puntuación completa, y a una línea sola un poco menos:

```
fn worth(hearing: &Hearing) -> f32 {
    match hearing.best() {
        Harmony::Silence => 1.0,
        Harmony::Single(_) | Harmony::Interval { .. } => 0.8,
        _ => hearing.fit(),
    }
}
```

Un pulso de silencio, o de una melodía sola, se une entonces al tramo que lo rodea en lugar de quedar cortado aparte, y no escribe nada: sigue el acorde de antes.

Un poliacorde también **continúa** mientras continúan sus notas. En la habitación de Petrushka, la mitad de Fa♯ mayor del acorde descansa un compás mientras sigue Do mayor. Escuchado solo, ese compás es C. Pero Do mayor y Fa♯ mayor siguen explicando casi todo, así que donde ninguna otra cosa encaja bien la hoja mantiene C|F♯ en lugar de saltar a C y volver (`Hearing::goes_on`).

Escuchar rápido

Escribir la hoja escucha diez tramos de cada compás de 4/4, y las pruebas escriben las hojas de todas las demostraciones, así que la velocidad importa. Tres cosas la mantienen rápida.

Hacer una sola vez el trabajo compartido. Qué partes se mueven en paralelo, y cuáles tocan acordes nota a nota, depende del compás y de los compases de alrededor, no del tramo. Un pequeño struct lo calcula una vez y escucha los tramos a partir de ahí:

```
pub struct Listening<'a> {
    notes: &'a [HeardNote],
    /// The parts that move their chords along with a tune.
    planing: Vec<usize>,
    /// The parts that play chords one note at a time.
    broken: Vec<usize>,
}
```

`'a` es un **tiempo de vida** (*lifetime*). `Listening` toma prestadas las notas en lugar de copiarlas, y `'a` le dice al compilador que un `Listening` no puede vivir más que el

slice que toma prestado. `hear(start, end)` elige entonces las notas de un tramo y las vuelve a medir desde su inicio. El piano roll hace lo mismo para su fila Todo: reúne una vez las notas de cada compás y escucha sus pulsos a partir de ellas.

No hacer por cada nota una pregunta que se puede hacer por cada parte. Que una parte sea una melodía depende de todas sus notas. Preguntarlo otra vez para cada nota del tramo es un trabajo de orden n^2; `weights` pregunta una vez por parte, guarda las respuestas en `roles` y busca allí la parte de cada nota.

No escuchar dos veces el mismo compás. Las canciones se repiten. Un compás tocado otra vez exactamente como antes, entre los mismos compases, se escucha igual, así que `write_heard_chords` guarda un `HashMap` que va de las líneas de pulso y las notas de cada compás a cómo se cortó:

```
type HeardBars = HashMap<(Vec<usize>, Vec<(i64, i64, u8, usize)>),
Vec<(usize, Hearing)>>;
```

Las notas entran en la clave como números enteros, `(start, end, pitch, part)`, porque un `f32` no puede ser clave: no tiene `Hash` ni `Eq`, porque `NaN` no es igual a sí mismo. `type` da un nombre corto al tipo largo; es el mismo tipo, no uno nuevo.

Para encontrar lo que es lento, hay que medir. `valgrind --tool=callgrind` ejecuta un programa sobre un procesador simulado y cuenta cada instrucción, por función y por línea:

```
CARGO_PROFILE_RELEASE_DEBUG=true cargo test --release -p lineup-core --no-
run
valgrind --tool=callgrind ./target/release/deps/<test binary> --ignored
one
callgrind_annotate --inclusive=yes callgrind.out.<pid>
```

Señala la función, y la línea, en que se va el tiempo; una pregunta hecha por cada nota que podría hacerse por cada parte es el tipo de cosa que encuentra.

Escribirlo

Cada tramo se convierte en una marca de acorde donde cambia la armonía, y otra vez donde empieza una sección, como un cifrado vuelve a escribir el acorde al principio de cada parte. Un acorde se escribe en la tonalidad de la canción en el lugar donde suena, con la nota más grave de la banda después de la barra: acordes de piano con Do, Mi, La sobre un bajo que toca La son Am, no Am/C. Un poliacorde se escribe `superior|inferior`, `Eb7|Fb`, y un conjunto por su nombre, `4-23 T2` (ver La hoja de acordes: modelo, distribución y texto). En una tonalidad que es un conjunto o una

serie dodecafónica, donde los acordes no tienen función, cada pulso se escribe en cambio como el conjunto que forman sus notas, o como un micromodo (ver Micromodos).

Un segundo oído, para los patrones de la propia banda

La biblioteca también escucha acordes, en una situación más sencilla. Para tocar una canción con un ritmo nuevo, o para que la banda siga los acordes de un patrón, necesita los acordes de un **canal de acordes** sobre una **línea de bajo**, en pasos enteros: los patrones que escribe un estilo. `library/listen.rs` lo hace con las mismas dos ideas, una puntuación para cada acorde y programación dinámica para los cortes, ajustadas para que todos los estilos de la biblioteca, toquen como toquen sus acordes, se escuchen como su cifrado.

Sus puntuaciones necesitan saber «cuántos pasos suena cada clase de altura entre el paso `a` y el paso `b`», para miles de tramos. Así que suma una vez, desde el principio:

```
struct Heard {
    sums: Vec<[f32; 12]>,
    busy: Vec<f32>,
    quiet: Vec<f32>,
    /// The pitch class of the lowest bass note that starts at each step,
    /// if any.
    bass: Vec<Option<PitchClass>>,
}
```

`sums[s][pc]` es en cuántos de los pasos *anteriores* a `s` suena la clase de altura `pc`. El tiempo entre `a` y `b` es entonces una sola resta, sea cual sea la longitud del tramo:

```
fn between(&self, start: usize, end: usize) -> Stretch {
    let mut heard = [0.0; 12];
    for (pc, time) in heard.iter_mut().enumerate() {
        *time = self.sums[end][pc] - self.sums[start][pc];
    }
    // ...
}
```

Esto es una **suma de prefijos** (*prefix sum*), y aparece siempre que un programa pregunta muchas veces «cuánto hay en este rango». Los arrays de tamaño fijo son `Copy` cuando lo son sus elementos, así que al construir las sumas `let mut next = heard.sums[heard.sums.len() - 1];` copia la última fila, y sumar a `next` deja intacta la fila del vector.

Después de los cortes se unen los vecinos. Dos tramos con el mismo acorde se vuelven uno, y lo mismo un C y un C7 un pulso después, porque un arpegio muestra sus notas una a una:

```
match joined.last_mut() {
    Some(last) if last.chord == stretch.chord => last.length +=
stretch.length,
    // An arpeggio shows its notes one by one: a C that turns into
    // C7 a beat later is one C7 still unfolding. A change on a
    // bar line or half way through the bar is taken as meant.
    Some(last) if !half_bar(stretch.start) && grows(last.chord,
stretch.chord) => {
        last.chord = bigger(last.chord, stretch.chord);
        last.length += stretch.length;
    }
    _ => joined.push(stretch),
}
```

`last_mut()` da un `Option<&mut HeardChord>`, una referencia a través de la cual se cambia el último tramo, así que crece en su sitio. Los brazos se prueban en orden con sus **guardas** (el `if` después de un patrón); una guarda falsa pasa al brazo siguiente, y `_` recoge el resto. Qué hace la banda con los acordes que oye está en La biblioteca: estilos y ritmos como datos y en Cómo toca la banda una hoja de acordes.

Encontrar la tonalidad

La segunda mitad del capítulo es la tonalidad. 🔍 **Buscar la tonalidad**, debajo de las teclas de armonía, muestra la tonalidad en la que parece estar la música, con sus rivales más cercanas al lado, más tenues, y qué hace el acorde que suena en cada una. Es un experimento, así que está desactivado hasta que lo actives. El trabajo está en `harmony/key_finding.rs`.

Cómo suena una tonalidad: los perfiles

Carol Krumhansl y Edward Kessler (1982) tocaban a los oyentes una escala y después una sola nota, y les preguntaban cuánto encajaba la nota. Las respuestas forman un **perfil de tonalidad**: doce números, cuánto pertenece a la tonalidad cada nota por encima de la tónica. Estudios posteriores contaron notas en partituras reales. Ferrous Lineup usa los perfiles de Albrecht y Shanahan (2013), contados sobre 490 obras clásicas, que de los probados fueron los que mejor nombraron las tonalidades de las demostraciones:

nota sobre la tónica	1	♭2	2	♭3	3	4	♯4	5	♭6
mayor	.24	.01	.11	.01	.14	.09	.02	.21	.01
menor	.22	.01	.10	.12	.02	.10	.01	.21	.06

La tónica y la quinta destacan en los dos; la tercera distingue el mayor (3) del menor (♭3). En menor, ♭7 y 7 están cerca: aparecen tanto la séptima natural de la escala como la elevada de la menor armónica.

Comparar: el método de Krumhansl-Schmuckler

Para encontrar la tonalidad de una música:

1. Se suma cuánto suena cada una de las doce clases de altura, en cualquier octava: el **croma** de la música, empezando por Do. (Son los mismos doce números que el perfil de la escucha de acordes.)
2. Se gira cada perfil para que empiece en cada una de las doce tónicas: 24 tonalidades.
3. Se compara el croma con cada una de las 24 mediante su **correlación**: 1 cuando suben y bajan juntos, 0 cuando no tienen relación, −1 cuando son opuestos.
4. Gana la tonalidad con la correlación más alta.

La correlación son dos pasos: llevar las dos listas a una media de 0 y una longitud de 1, y después hacer su producto escalar.

```
fn normalized(values: &Chroma) -> Option<Chroma> {
    let mean = values.iter().sum::<f32>() / 12.0;
    let centered = values.map(|v| v - mean);
    let length = centered.iter().map(|v| v * v).sum::<f32>().sqrt();
    (length > 1e-9).then(|| centered.map(|v| v / length))
}
```

`Chroma` es `[f32; 12]`, y los arrays tienen un `map` propio que devuelve un array del mismo tamaño, así que no se crea ningún `Vec`. `Option` se ocupa del silencio: cuando no suena nada, los doce números son iguales, la longitud es 0 y no hay tonalidad que encontrar, así que la función devuelve `None` en lugar de dividir por cero. `then` es el primo perezoso de `then_some`: ejecuta el closure solo cuando se cumple la condición, así que con silencio la división nunca ocurre.

Girar un perfil hacia otra tónica usa `std::array::from_fn`, que construye un array llamando a un closure para cada índice:

```
// The profile counts from the tonic; turn it to count from C.
let turned: Chroma = std::array::from_fn(|c| profile[(c + 12 - tonic) %
12]);
```

`key_fits` devuelve las 24 tonalidades con su correlación, la mejor primero. La importación MIDI la usa para comprobar la armadura de un archivo contra sus notas (ver MIDI: mensajes, dispositivos, grabación y archivos).

Seguro o no seguro

Las correlaciones solas no dicen lo segura que es la respuesta. Do mayor y La menor tienen las mismas notas, así que una melodía en las teclas blancas puede encajar en las dos casi igual de bien. `ranked` convierte las 24 correlaciones en **partes** que suman 1, un *softmax*:

```
let best = fits.iter().copied().fold(f32::MIN, f32::max);
let weights: Vec<f32> = fits
    .iter()
    .map(|&fit| ((fit - best) * SHARE_SHARPNESS).exp())
    .collect();
let total: f32 = weights.iter().sum();
```

Con `SHARE_SHARPNESS` en 12, un encaje mejor por 0,05 recibe una parte e^(12 × 0,05) ≈ 1,8 veces mayor. Restar antes `best` deja todos los exponentes en 0 o por debajo, así que `exp` nunca puede desbordarse. Una tonalidad es **clara** cuando su parte es al menos el doble de la segunda:

```
fn clear(guesses: &[KeyGuess]) -> bool {
    match guesses {
        [] => false,
        [_] => true,
        [best, second, ..] => best.share >= CLEAR_RATIO * second.share,
    }
}
```

Estos son **patrones de slice**: `[]` encaja con un slice vacío, `[_]` con uno de exactamente un elemento, y `[best, second, ..]` nombra los dos primeros de uno más largo, con `..` para el resto. Cuando la tonalidad no es clara, las siguientes cuya parte es al menos el 40% de la mejor son sus **alternativas**, dos como máximo, y la aplicación las muestra más tenues.

Mi dórico tiene las mismas notas que La menor y Sol mayor, así que los perfiles, que solo conocen el mayor y el menor, no pueden elegir su nota de reposo con confianza, y la aplicación lo dice: La menor, o Mi menor, o La mayor. La tónica real está entre las alternativas, que es la respuesta honesta.

Dónde cambia la tonalidad

Una canción que modula no tiene un solo croma. Así que la música se corta en compases, y cada compás recibe una puntuación para cada tonalidad. Un solo compás es poco, sin embargo: un compás que solo tiene D7 encaja mejor en Sol mayor que en Do mayor, incluso en mitad de una canción en Do mayor. Así que cada compás se puntúa junto con los de alrededor: él mismo cuenta tres veces, sus vecinos dos, los compases a dos de distancia una.

Después, un cambio de tonalidad **cuesta** una penalización, 1,5 en unidades de correlación, y el **algoritmo de Viterbi** encuentra la secuencia de tonalidades con la mejor puntuación total. Para cada compás y cada tonalidad guarda solo la mejor manera de llegar allí: quedarse en esa tonalidad, o cambiar desde la mejor tonalidad hasta el momento pagando la penalización.

```
for (i, here) in windowed.iter().enumerate().skip(1) {
    let (leader, leader_score) = best_of(&best);
    let mut next = vec![0.0; count];
    for k in 0..count {
        let (previous, total) = if best[k] >= leader_score - CHANGE_COST {
            (k, best[k])
        } else {
            (leader, leader_score - CHANGE_COST)
        };
        from[i][k] = previous;
        next[k] = total + here[k];
    }
    best = next;
}
```

`from[i][k]` recuerda de dónde venía el mejor camino hacia la tonalidad `k` en el compás `i`, y al final el código lo recorre hacia atrás para leer el camino. Es otra vez programación dinámica, sobre tonalidades en lugar de cortes. Una tonalidad nueva gana solo cuando encaja mejor durante el tiempo suficiente para pagar la penalización, que es exactamente «un cambio solo cuando la tonalidad nueva se mantiene»: un acorde prestado no es una modulación.

Puntuar junto con los compases vecinos difumina *dónde* ocurre un cambio en uno o dos compases. Así que cada cambio se mueve después, dentro de tres compases, al compás donde las notas propias de los compases pasan de encajar en la tonalidad

vieja a encajar en la nueva. En la Habanera de *Carmen* cae en el compás 10, donde el Re menor de Bizet pasa a Re mayor.

`Project::heard_keys` hace esto para la canción entera: cuentan los sintetizadores y los canales MIDI Out que no están silenciados, la batería y los samplers (como una grabación importada) no, y cada nota cuenta tanto como suena en cada compás. Devuelve `KeySpan`s, cada uno con un inicio, un final y todas las tonalidades ordenadas. **Marcar en la lista de reproducción**, junto a la tonalidad escuchada, los convierte en marcas de tonalidad (ver Escalas, modos y tonalidades): ninguna nota se mueve, y Deshacer las quita. Es la manera de darle a un archivo MIDI importado sin armaduras los cambios de tonalidad que tiene de verdad.

Mientras tocas

Para los acordes tocados en un teclado MIDI o en el teclado de escribir no hay partitura por delante en la que mirar, solo los compases tocados hasta ahora. `KeyListener` guarda un croma del pasado que se va apagando:

```
/// A bar ended: it joins the past, and the key may change.
pub fn end_bar(&mut self) {
    if normalized(&self.bar).is_none() {
        return;
    }
    let heard = self.heard();
    self.past = heard.map(|v| v * LISTENER_MEMORY);
    self.bar = [0.0; 12];
    // ...
}
```

En cada compás, todo lo oído antes cuenta 0,84 veces, así que un compás de hace cuatro compases cuenta más o menos la mitad ($0{,}84^4 \approx 0{,}5$). Una tonalidad nueva toma el mando solo después de ir delante dos compases seguidos. El oyente tampoco empieza de cero: `KeyListener::starting_in` llena el pasado con música con la forma de la tonalidad de alrededor, la que se escucha en las notas de la canción en ese punto o la del proyecto, así que los primeros acordes que tocas se oyen a su luz. Toca C, F y G7 sobre una canción en Mi dórico y primero se oyen contra La menor; unos compases después el oyente se decide por Do mayor, clara.

La aplicación muestra la tonalidad de la fuente más inmediata que tiene (ver `key_finder.rs`): lo que tocas, si tocaste en los últimos cuatro compases; si no, las tonalidades oídas en una grabación importada, o en las notas de la canción en el cabezal; en el modo de patrón, las notas del patrón seleccionado. La línea dice cuál: «Suena en lo que tocas:», «en las notas de aquí», «en este patrón». Cómo se oyen las tonalidades de una grabación está en Importar una grabación (experimental).

Qué es un acorde en cada tonalidad

El mismo acorde hace un trabajo distinto en cada tonalidad a la que podría pertenecer, y por eso importan las alternativas. `read_chord` lo calcula con lo que la aplicación ya sabe de las tonalidades: el número romano, la función, un acorde de séptima de dominante oído como la dominante del acorde hacia el que tira, y los acordes prestados de la tonalidad homónima (ver Números romanos, las teclas de armonía y las funciones de los acordes). G7 se lee:

tonalidad	lectura	qué hace
Do mayor	V7, dominante	tira hacia Do, la tónica
La menor	♭VII7 = V7/♭III	la dominante de Do, la relativa mayor
Sol mayor	I7 = V7/IV	el acorde de tónica con séptima menor, que tira hacia Do, el IV

Los números romanos se cuentan sobre la escala mayor de la tónica, como en todo Ferrous Lineup, así que Sol en La menor es ♭VII.

Qué tan bien escucha

Las pruebas de `tests/chord_reading.rs` comprueban compases cuya armonía se conoce por las partituras: el acorde de los Augurios, el acorde de Petrushka en la habitación de Petrushka, y las cuartas del principio de la feria. Un informe, que es una prueba ignorada, compara pulso a pulso los acordes oídos en las notas con el cifrado de cada demostración que tiene uno:

```
cargo test --release -p lineup-core --test chord_reading -- --ignored --
nocapture
```

`tests/key_finding.rs` tiene un informe del mismo tipo para las tonalidades, contra las tonalidades y las marcas de tonalidad que las demostraciones tienen de sus partituras:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored
--nocapture
```

Medido el 30 de octubre de 2026:

Acordes, 330 demostraciones con cifrado, 60.322 pulsos	
Exactamente el acorde del cifrado	84,0%
Un acorde compatible (las mismas notas, o la misma fundamental con notas añadidas u omitidas)	91,1%
La fundamental correcta	92,1%
Tonalidades, a partir de las notas	
Compases en la tonalidad correcta	87,0% de 18.828
Cambios de tonalidad de las partituras encontrados a un compás o menos	132 de 315
Compases mostrados como claros que son correctos	13.900 de 16.153 (86%)

Cada informe también lista las demostraciones en las que más se equivoca: el sitio donde mirar después. En las tonalidades, la mayoría de los fallos son de tres tipos:

- **Visitas cortas.** Un cambio que dura de cuatro a ocho compases, a menudo hacia la dominante, es difícil de distinguir de un pasaje que solo la toniciza, y muchos músicos leerían esos compases también como tonicizaciones.
- **Tonalidades relativas.** Las mismas notas, otro reposo: una canción en Si♭ mayor puede salir como Sol menor.
- **Modos**, como el Mi dórico de antes.

Algunos «cambios que no están» son reales, también: las partituras de las que salen las demostraciones marcan solo los cambios de tonalidad más grandes.

Cómo se prueba

Las pruebas unitarias de `hearing.rs` construyen unas pocas notas a mano y comprueban qué se escucha. Un arpegio, una nota por paso, es su acorde:

```rust
#[test]
fn an_arpeggio_is_its_chord() {
    // A minor broken up, one note per step: no two notes ever sound
    // together.
    let notes = [
        held(45, 0.0, 1.0),
        held(52, 1.0, 2.0),
        held(57, 2.0, 3.0),
        held(60, 3.0, 4.0),
    ];
    assert_eq!(chord_of(&hear(&notes, 4.0)).unwrap(), "Am");
}
```

`held` crea una nota de la parte 0, y `chord_of` nombra la mejor lectura. `unwrap` está bien en una prueba: si no hay nombre, la prueba debe fallar ahí mismo. Las demás comprueban que un acorde sostenido es él mismo, que las notas de paso de una melodía dejan el acorde como está, que el acorde de los Augurios es E♭7 sobre F♭ y el de Petrushka dos arpegios a un tritono, que una superposición de cuartas es un conjunto, y que nada oído es silencio. `key_finding.rs` comprueba que una cadencia nombra su tonalidad, que las partes suman uno, que la escala pentatónica sola no es clara, que un compás prestado no es una modulación y que el oyente cambia de tonalidad cuando la nueva se mantiene; `tests/key_finding.rs` comprueba que la mayoría de los compases de las demostraciones se oyen en su tonalidad y que las modulaciones claras, como la de la Habanera, se encuentran en su compás.

Los ejercicios sobre la escucha están en Ejercicios: la teoría musical en código.

Conjuntos de clases de altura, y una armonía con cuatro nombres

Archivos: `crates/lineup-core/src/harmony/set_class.rs`, `crates/lineup-core/src/harmony/pitch_set.rs`, `crates/lineup-core/src/chord_sheet/notation.rs`, `crates/lineup-app/src/twelve_tone.rs` y `crates/lineup-app/src/chord_sheet/notation.rs`

Los nombres de acordes sirven para los acordes de la música tonal. Mucha música del siglo XX usa grupos de notas que no tienen nombre de acorde, como Do, Do♯ y Fa♯. La **teoría de conjuntos de clases de altura**, desarrollada por Milton Babbitt, Allen Forte, John Rahn y otros, da un nombre a cualquier grupo de notas. En Ferrous Lineup un conjunto es una manera más de *escribir* una armonía, junto a los cifrados, los números romanos y los micromodos: pasar a conjuntos nunca mueve una nota. Este capítulo trata de la teoría, de las pocas líneas de aritmética de bits que la calculan, y de los lugares donde la usa la aplicación: el explorador de conjuntos, una tonalidad mostrada como conjunto, y las cuatro notaciones de la hoja de acordes.

La música, en pocas palabras

Las clases de altura como números. Do es 0, Do♯ es 1, y así hasta Si, 11. Las octavas y la escritura no cuentan: todo Do es 0, y Do♯ y Re♭ son los dos 1. Un **conjunto** es una elección de algunas de las doce; hay $2^{12} = 4096$, desde el conjunto vacío hasta las doce.

Mover un conjunto. *Transportar* (T*n*) sube cada nota *n* semitonos: Do Mi Sol subido 3 es Mi♭ Sol Si♭. *Invertir* (I) da vuelta cada intervalo: Do Mi Sol (sube 4, luego sube 3) se vuelve Do La♭ Fa (baja 4, luego baja 3), que es Fa menor. Dada vuelta, una tríada mayor es una tríada menor.

Clases de conjuntos. Un conjunto junto con todas sus transposiciones e inversiones es una **clase de conjuntos**. Las 24 tríadas mayores y menores son una sola clase. Los 4096 conjuntos se reparten en **224 clases**.

La **forma normal** es la manera más compacta de escribir un conjunto. Escrito alrededor de un círculo, un conjunto puede empezar en cualquiera de sus notas; se empieza en cada una por turno y se mide la distancia de la primera nota a la última. Tomemos Do mayor con séptima mayor, Do Mi Sol Si:

empezando en	notas	distancia	hasta la penúltima
Do	Do Mi Sol Si	11	
Mi	Mi Sol Si Do	8	7
Sol	Sol Si Do Mi	9	
Si	Si Do Mi Sol	8	5

Mi y Si empatan con una distancia de 8, así que se compara la distancia hasta la penúltima nota: de Si a Mi hay 5, de Mi a Si hay 7. La forma normal es **Si Do Mi Sol**. (Este desempate, comparando desde el final, es el de John Rahn; Allen Forte comparaba desde el principio, y los dos difieren en seis clases. Ferrous Lineup usa el de Rahn, como hoy la mayoría de los libros.)

La **forma prima** va un paso más allá: se toman las formas normales del conjunto *y de su inversión*, se mueven las dos para que empiecen en 0, y se queda la más compacta. Para Cmaj7 es **[0158]**: Si Do Mi Sol bajado para empezar en 0 es 0 1 5 8. Todo acorde de séptima mayor, sobre cualquier fundamental, tiene la forma prima [0158].

Los **números de Forte** ordenan las clases por tamaño y después según la lista de Forte: [0158] es **4-20**, la vigésima clase de cuatro notas. Las tríadas son 3-11, el acorde de séptima disminuida 4-28, la escala mayor 7-35, la escala de tonos enteros 6-35. Los libros sobre la música del siglo XX usan estos números en todas partes.

El vector interválico. Cada par de notas de un conjunto forma un intervalo. Contarlos por tamaño da seis números: cuántas segundas menores, segundas mayores, terceras menores, terceras mayores, cuartas y tritonos. Una quinta cuenta como cuarta (es una cuarta dada vuelta) y una séptima mayor como segunda menor, así que seis tamaños bastan. El vector es un buen resumen de cómo *suena* un conjunto. Para Do Mi Sol Si:

```
Do–Mi tercera mayor   Do–Sol cuarta   Do–Si segunda menor
Mi–Sol tercera menor  Mi–Si cuarta    Sol–Si tercera mayor
<1 0 1 2 2 0>, escrito <101220>
```

De ahí salen dos curiosidades:

- Las clases en **relación Z** tienen el mismo vector interválico sin ser la misma clase. 4-Z15 [0146] y 4-Z29 [0137] tienen cada una un intervalo de cada tamaño, `<111111>` (son los dos **tetracordios de todos los intervalos**), y sin embargo ninguna se puede mover sobre la otra. La **Z** de sus nombres las señala.
- **Complementos**: las notas que deja afuera un conjunto forman otro conjunto. Las teclas blancas (7-35) dejan afuera las negras, la escala pentatónica (5-35). Un conjunto de seis notas y su complemento tienen siempre el mismo vector interválico.

La teoría de conjuntos en doce bits

Un `PitchSet` son doce bits de un `u16`, del bit 0 para Do al bit 11 para Si, y transportar es rotar los bits (ver Clases de altura, notas y su escritura). El resto de la teoría de conjuntos encaja igual de bien en esos bits.

El complemento es `!`. El «no» invierte cada bit, y `from_bits` se queda con los doce que significan notas:

```
pub fn complement(self) -> PitchSet {
    PitchSet::from_bits(!self.bits())
}
```

Invertir convierte cada clase de altura *p* en 12 − *p*, lo que refleja el conjunto alrededor de Do:

```
pub fn invert(self) -> PitchSet {
    self.iter()
        .map(|pitch_class| PitchClass::from_semitone(12 -
pitch_class.semitone()))
        .collect()
}
```

La forma más compacta es el número más pequeño. Esta es la parte ingeniosa. Se baja un conjunto para que su primera nota sea Do, el bit 0. Su última nota es ahora su bit más alto, así que una distancia menor significa un bit más alto menor, y eso significa un número menor. En un empate, el siguiente bit hacia abajo es la penúltima nota, que es exactamente el desempate de Rahn. Comparar dos números *es* la regla de Rahn:

empezando en	bajado a Do	bits	número
Do	0 4 7 11	`1000_1001_0001`	2193
Mi	0 3 7 8	`0001_1000_1001`	393
Sol	0 4 5 9	`0010_0011_0001`	561
Si	0 1 5 8	`0001_0010_0011`	291

Así que la forma normal es «el comienzo que da el número más pequeño», y la forma prima «el número más pequeño de las 24 transposiciones e inversiones»:

```
pub fn normal_form(self) -> Vec<PitchClass> {
    let Some(start) = self
        .iter()
        .min_by_key(|start| self.relative_to(*start).bits())
    else {
        return Vec::new();
    };
    self.relative_to(start)
        .iter()
        .map(|pitch_class| pitch_class.transpose(start.semitone() as i32))
        .collect()
}

pub fn prime_form(self) -> PitchSet {
    let inverted = self.invert();
    (0..12)
        .flat_map(|n| [self.transpose(n), inverted.transpose(n)])
        .min_by_key(|set| set.bits())
        .unwrap_or(self)
}
```

`min_by_key` recorre un iterador y se queda con el elemento de clave más pequeña. En `prime_form`, `flat_map` convierte cada `n` en un array de dos conjuntos, así que el iterador da 24 conjuntos, y el más pequeño es la forma prima. Sin ordenar nada y sin listas de intervalos: unas pocas comparaciones de enteros. `relative_to(start)` baja el conjunto hasta que `start` es Do.

Cómo se mueve un conjunto sobre otro, como T*n* o T*n*I, es una búsqueda entre los mismos 24 movimientos:

```
pub fn moved_to(self, other: PitchSet) -> Option<(u8, bool)> {
    let inverted = self.invert();
    (0..12)
        .find(|&n| self.transpose(n) == other)
        .map(|n| (n, false))
        .or_else(|| {
            (0..12)
                .find(|&n| inverted.transpose(n) == other)
                .map(|n| (n, true))
        })
}
```

`or_else` ejecuta su closure solo cuando la primera búsqueda no encontró nada, así que siempre se prefiere una transposición a una inversión. Una clase es **simétrica** cuando invertir su forma prima da una transposición de ella, como pasa con la tríada disminuida: dada vuelta, es el mismo conjunto movido.

El vector interválico es el único lugar que cuenta pares de la manera sencilla, un bucle dentro de otro. Con doce notas como mucho, son 66 pares como mucho:

```
pub fn interval_vector(self) -> [u8; 6] {
    let notes = self.semitones();
    let mut vector = [0; 6];
    for (i, low) in notes.iter().enumerate() {
        for high in &notes[i + 1..] {
            let interval = high - low;
            let class = interval.min(12 - interval);
            vector[class as usize - 1] += 1;
        }
    }
    vector
}
```

`interval.min(12 - interval)` junta un intervalo con su inversión: 7, una quinta, se vuelve 5, una cuarta.

224 clases en una tabla

La forma prima se puede calcular, pero el *número* de Forte no: es solo un lugar en la lista de Forte. Así que `set_class.rs` guarda la lista, un array constante de 224 entradas:

```
const SET_CLASSES: [SetClass; 224] = [
    SetClass::new(0, 1, false, 0),
    SetClass::new(1, 1, false, 0b000000000001),
    SetClass::new(2, 1, false, 0b000000000011),
    // ...
    SetClass::new(4, 20, false, 0b000100100011),
    // ...
    SetClass::new(12, 1, false, 0b111111111111),
];
```

`SetClass::new` y `PitchSet::from_bits` son `const fn`, así que la tabla entera se construye al compilar y se guarda con el programa; buscar un conjunto no cuesta nada al arrancar. El `false` o `true` dice si la clase lleva una Z en su nombre. `0b` escribe un número en binario, así que la forma prima [0158] se puede leer en los bits.

Una tabla de 224 líneas de números es fácil de equivocar, así que las pruebas la comprueban contra la teoría en lugar de confiar en ella:

- cada entrada es su propia forma prima, y las entradas están en orden de tamaño y de número;
- hay 1, 1, 6, 12, 29, 38, 50, 38, 29, 12, 6, 1 y 1 clases de 0 a 12 notas, las cantidades que da cualquier libro;

- se buscan **los 4096 conjuntos**, y cada uno cae en una clase de su propio tamaño;
- una clase lleva Z exactamente cuando otra clase de su tamaño tiene el mismo vector interválico;
- para 5-20, donde Forte y Rahn ordenan las notas de otra manera, la forma prima es la de Rahn.

Gracias a la tercera prueba, la búsqueda puede decir lo que espera en lugar de ocuparse de un «no encontrado» que no puede ocurrir:

```
pub fn of(set: PitchSet) -> &'static SetClass {
    let prime = set.prime_form();
    SET_CLASSES
        .iter()
        .find(|class| class.prime == prime)
        .expect("every prime form is in the table")
}
```

`expect` provoca un pánico con ese mensaje si el `Option` es `None`. Usarlo es una promesa, y la prueba es lo que la mantiene. La función devuelve `&'static SetClass`: un préstamo de una entrada de la tabla constante, que vive tanto como el programa, así que quien la llama puede guardarla todo el tiempo que quiera.

La clase escribe sus propios nombres. `vector_text` escribe las seis cantidades como dígitos, `<001110>`, como lo hacen los libros, hasta que una llega a 10: las doce notas tienen doce de cada intervalo menos del tritono, y eso se escribe con espacios, `<12 12 12 12 12 6>`, para que no se pueda leer mal. Algunas clases tienen además un nombre conocido, de una lista corta de ejemplos: «tríada mayor o menor», «escala de tonos enteros», «escala octatónica (disminuida)», «acorde de Petrushka».

El explorador de conjuntos

Conjuntos…, en la barra de herramientas del piano roll, abre la ventana **Conjuntos de clases de alturas**: todas las clases del tamaño elegido (de 2 a 10 notas), cada una con su número de Forte, su forma prima, su vector interválico y su nombre conocido, como «3-11 [037] <001110> tríada mayor o menor». El cuadro de búsqueda encuentra una clase por número o por nombre («4-Z15», «tríada», «octatónica»), y con dos o más notas seleccionadas en el piano roll, **Mostrarlo** salta a su clase. Al elegir una clase se muestra su pareja Z si la tiene, la clase de su complemento, y si es simétrica. **Transportarlo** e **Invertido (I)** eligen un miembro de la clase: T3 es el conjunto tres semitonos más arriba, T3I su inversión subida tres. Después:

- ▶ **Escucharlo** toca el conjunto con el instrumento seleccionado;
- **Dibujar con él** convierte el conjunto en el sello de acordes del piano roll, así que cada clic pone el conjunto entero sobre la nota pulsada (`ChordStamp::Set`, junto a los sellos de Acordes: construirlos, nombrarlos e invertirlos);
- **Usar como tonalidad** convierte el conjunto en la tonalidad de la canción: el piano roll resalta sus notas y las teclas de armonía tocan acordes apilados con ellas.

Un conjunto de tres notas es además uno de los micromodos de Kröpfl, y la ventana enlaza con él (ver Micromodos).

El piano roll nombra conjuntos también sin la ventana. Debajo de la barra de herramientas, junto al acorde del paso que está bajo el puntero, muestra el conjunto de ese paso cuando suenan tres notas o más, y el de las notas seleccionadas cuando hay dos o más:

```
Conjunto: 3-11 · forma normal Sol♯ Si Re♯ · forma prima [037] · intervalos
<001110> · tríada mayor o menor
```

Una tonalidad como conjunto: las mismas notas, otro nombre

Elegir **conjunto de clases de alturas** en un menú Tonalidad podría significar dos cosas: tocar en otra escala, o nombrar de otra manera las mismas notas. Ferrous Lineup toma el segundo sentido, porque un conjunto es una notación. Do mayor ya *es* un conjunto de clases de altura, Do Re Mi Fa Sol La Si, la clase 7-35. Así que pasar la tonalidad de Do mayor a «conjunto de clases de alturas» deja quietas todas las notas de la canción y solo muestra la misma armonía en notación de conjuntos:

Tecla de armonía	Notas	Clase
C	Do Mi Sol	3-11
Dm	Re Fa La	3-11
Em	Mi Sol Si	3-11
F	Fa La Do	3-11
G	Sol Si Re	3-11
Am	La Do Mi	3-11
Bdim	Si Re Fa	3-10

La tabla es una lección en sí misma. Las tríadas mayores y menores son la **misma** clase, 3-11, porque una tríada menor es una tríada mayor dada vuelta, y una clase cuenta un conjunto y su inversión como uno solo. La tríada disminuida es otra clase, 3-10.

Cambiar de escala es distinto: de mayor a menor, Mi, La y Si pasan a Mi♭, La♭ y Si♭, y la canción se mueve (ver Escalas, modos y tonalidades). Un conjunto que eliges en el explorador con **Usar como tonalidad** puede ser cualquier conjunto.

`set_key`, en la aplicación, elige el conjunto al que pasa el menú:

```
fn set_key(ctx: &egui::Context, scale: Scale, root: PitchClass) -> Scale {
    let up = scale.root.semitones_up_to(root);
    if let Some(group) = scale
        .tone_row()
        .and_then(|row| row.segments(3).first().copied())
    {
        return Scale::set(root, group.transpose(up));
    }
    if scale.kind.is_tonal() && scale.kind != ScaleKind::Chromatic {
        let set = Scale::set(root, scale.pitch_set().transpose(up));
        // Gb major as a set still writes its notes Gb Ab Bb Cb...
        return if up == 0 {
            set.with_spelling(scale.spelling())
        } else {
            set
        };
    }
    let set = remembered_set(ctx);
    let start = set.normal_form().first().copied().unwrap_or(root);
    Scale::set(root, set.transpose(start.semitones_up_to(root)))
}
```

Desde una tonalidad es la escala de la propia tonalidad; desde una serie dodecafónica, las tres primeras notas de la serie, que son lo que tocaban las teclas de armonía; desde la escala cromática, que tiene todas las notas y no dice nada de la armonía, el último conjunto usado. Un conjunto cuyas notas son exactamente una escala sobre su fundamental se escribe como esa escala, y sigue como esa tonalidad cuando la tonalidad vuelve a cambiar (`set_as_scale` y `as_key`, en Escalas, modos y tonalidades).

Una armonía, cuatro nombres

La hoja de acordes aplica la misma regla a cada armonía. Su fila superior tiene un menú **Mostrar**: **Como está escrito**, **Acordes**, **Conjuntos de clases de alturas** o

Micromodos. Las notas siguen siendo las mismas, solo cambian sus nombres, y nada de la canción se toca:

Escrito	Acordes	Conjuntos	Micromodos
C	C	3-11 T7I	Acorde menor-mayor T7I
Dm	Dm	3-11 T2	Acorde menor-mayor T2
G	G	3-11 T2I	Acorde menor-mayor T2I
Am	Am	3-11 T9	Acorde menor-mayor T9
Bdim	Bdim	3-10 T11	Acorde disminuido T11
G7	G7	4-27 T7I	4-27 T7I
3-11 T2	Dm	3-11 T2	Acorde menor-mayor T2
Menor 3	Menor 3	3-3	Menor 3

Cómo leerla:

- **T*n* y T*n*I.** Cada clase tiene una forma prima que empieza en Do. Para 3-11 es [0, 3, 7], Do Mi♭ Sol, una tríada menor. **T9** significa «la forma prima subida 9 semitonos»: La Do Mi, La menor. **T*n*I** significa «dada vuelta primero, y después subida *n*». Dada vuelta, [0, 3, 7] es 0, −3 y −7: Do, La y Fa. Subido 7, eso es Sol, Mi y Do, Do mayor. Así que todo acorde mayor lleva una I en su nombre, y ningún acorde menor la lleva. Dm (T2) y G (T2I) comparten el número 2 pero no las notas. La forma prima misma, T0, se escribe sin T: Menor 3 es Do Re♭ Mi, 3-3 sobre Do.
- **Un micromodo tiene siempre tres notas.** Los doce micromodos de Kröpfl son las doce clases de tres notas, de 3-1 a 3-12, con los nombres en castellano que él les dio. El micromodo 3-11 es el **Acorde menor-mayor**, porque la tríada mayor y la menor son una sola clase. G7 tiene cuatro notas, así que no es un micromodo, y la vista Micromodos escribe su conjunto.
- **Acordes** vuelve a convertir un conjunto en acorde **cuando sus notas forman uno**: el conjunto 3-11 T2 es Re Fa La, así que la vista escribe Dm, con la fundamental escrita como la escribe la tonalidad. Un conjunto que no forma ningún acorde que conozca la aplicación, como 3-1 o Menor 3, conserva su nombre. **N.C.** (sin acorde) queda como N.C. en todas las vistas, y una forma de serie (P0, RI7) queda como forma de serie: una serie es una línea de doce notas, no un grupo.

Un poliacorde son dos grupos de notas, uno sobre otro, y juntarlos en un solo conjunto perdería lo que lo hace un poliacorde. Así que cada acorde se nombra por separado, y se escriben el de arriba primero, como se escribe el poliacorde:

```
Eb7|Fb   →   4-27 T3I|3-11 T11I
```

En Rust: un solo `match` decide

`Notation` es un enum de cuatro variantes, con `#[default]` en `Written` para que la hoja empiece como siempre. Renombrar una armonía es un solo `match` sobre una **tupla** de la notación y la armonía. Rust prueba los brazos de arriba abajo, así que su orden es la regla: primero lo que nunca cambia, después la vista Acordes, después dos acordes a la vez, y después todo lo demás.

```
pub fn rename(self, harmony: HarmonyName, key: Scale) -> Vec<HarmonyName>
{
    let group = |set: PitchSet| self.name_group(set, key);
    match (self, harmony) {
        (Notation::Written, _)
        | (_, HarmonyName::NoChord)
        | (_, HarmonyName::Row(_))
        | (Notation::Chords, HarmonyName::Chord(_) |
HarmonyName::Polychord(..)) => {
            vec![harmony]
        }
        (Notation::Chords, HarmonyName::Set(set) |
HarmonyName::Micromode(set)) => {
            match chord_of(set) {
                Some(chord) => vec![HarmonyName::Chord(ChordSymbol::new(
                    key.spell_root(chord),
                    chord.quality,
                ))],
                None => vec![harmony],
            }
        }
        (_, HarmonyName::Polychord(upper, lower)) => {
            // The lower chord with its bass, then the upper chord.
            let lower = notes_of(lower);
            vec![group(lower), group(upper.chord().pitch_set())]
        }
        (_, HarmonyName::Chord(symbol)) => vec![group(notes_of(symbol))],
        (_, HarmonyName::Set(set) | HarmonyName::Micromode(set)) => vec!
[group(set)],
    }
}
```

El primer brazo usa **patrones «o»** en dos niveles: `A | B | C` entre tuplas enteras, y `Chord(_) | Polychord(..)` dentro de una. Un patrón «o» que da un nombre, como `Set(set) | Micromode(set)`, tiene que darlo en todas las alternativas, con el mismo tipo. La función devuelve un `Vec` porque un poliacorde se vuelve dos nombres y todo lo demás uno.

`chord_of` encuentra el acorde que forma un conjunto. Recorre las clases de acorde de la más sencilla, la tríada mayor, a la más compleja, y para cada una las notas del conjunto, en forma normal, como fundamentales posibles:

```
pub fn chord_of(set: PitchSet) -> Option<Chord> {
    let roots = set.normal_form();
    ChordQuality::ALL.into_iter().find_map(|quality| {
        roots
            .iter()
            .find(|&&root| ChordQuality::identify(set.relative_to(root))
== Some(quality))
            .map(|&root| Chord::new(root, quality))
    })
}
```

`identify` dice qué clase de acorde forman los intervalos sobre una fundamental, si forman alguna. Como las clases de acorde son el bucle de fuera, gana la que está primero en la lista, sea cual sea su fundamental: Do Mi Sol La es a la vez Am7 y C6, y se encuentra como Am7, porque los acordes de séptima van antes que los de sexta en la lista. `&&root` en el patrón del closure desarma una referencia a una referencia: `iter` da `&PitchClass`, y `find` le pasa a su closure una referencia a eso.

Leer las propias notas

Renombrar las armonías de la hoja es la mitad de lo que hace un analista con los conjuntos. La música sin cifrado, un concierto de Webern o una página de *La consagración de la primavera*, se corta en conjuntos **a partir de sus notas**, de dos maneras:

- **verticalmente**, las notas que suenan juntas, como se oye un acorde apilado o un cluster;
- **horizontalmente**, las notas una detrás de otra en una línea, como se corta una serie dodecafónica en grupos de tres o de seis.

Con **Conjuntos de clases de alturas** o **Micromodos** a la vista, un segundo menú elige cuál: **Las armonías de la hoja** (renombradas, como antes), **Notas a la vez (vertical)** o **Notas en línea (serial)**. Al leer notas, la hoja muestra los grupos que encuentra, cada uno donde empieza, y deja en paz lo escrito: escribir en un compás sigue escribiendo su armonía, y **Como está escrito** la muestra otra vez. Dos elecciones más dicen cuántas notas forman un grupo (de 2 a 12, o **Todas las notas** que suenan a la vez; un micromodo tiene siempre tres) y de quién son las notas que se leen. **Automático** lee la melodía para una línea (la pista de la melodía, Lead, o si no la más aguda, como la elige la partitura guía), porque una línea a través de todos los instrumentos a la vez mezclaría la melodía con los acordes; para las notas a la vez lee todas las pistas, porque un acorde suele estar repartido entre instrumentos.

Notas a la vez

Donde empieza una nota, `together` toma todas las notas que suenan en ese momento, de la más grave hacia arriba, se queda con una de cada clase de altura (una duplicación a la octava es la misma nota en un conjunto) y las corta en grupos del tamaño elegido. Dos reglas pequeñas hacen que los grupos tengan sentido musical:

- una sola nota que sobra arriba se une al grupo de debajo: siete notas de tres en tres son 3 + 4, no 3 + 3 + 1, porque una nota no es una armonía;
- un momento en el que suenan menos de dos notas distintas se salta.

```
fn chunks(notes: Vec<Note>, size: Option<usize>) -> Vec<NoteGroup> {
    if notes.is_empty() {
        return Vec::new();
    }
    let Some(size) = size else {
        return vec![NoteGroup { notes }];
    };
    let mut groups: Vec<NoteGroup> = notes
        .chunks(size.max(1))
        .map(|chunk| NoteGroup {
            notes: chunk.to_vec(),
        })
        .collect();
    if groups.len() > 1 && groups.last().is_some_and(|last|
last.notes.len() == 1) {
        let left = groups.pop().map(|last|
last.notes).unwrap_or_default();
        if let Some(under) = groups.last_mut() {
            under.notes.extend(left);
        }
    }
    groups
}
```

`chunks` sobre un slice lo corta en trozos de `size`, el último quizá más corto. `pop` saca el último grupo, y `last_mut` llega al que ahora es el último, para añadirle la nota que sobra.

El acorde de los Augurios muestra por qué importa la primera regla. De abajo arriba las cuerdas tocan Fa♭ La♭ Do♭, luego Sol Si♭ Re♭, y Mi♭ arriba del todo. De tres en tres eso es Fa♭ La♭ Do♭ | Sol Si♭ Re♭ | Mi♭, y el Mi♭ solitario se une al grupo de debajo: Sol Si♭ Re♭ Mi♭, que es E♭7. La hoja escribe **4-27 T3I|3-11 T11I**, los dos acordes que Stravinsky superpuso, encontrados solo a partir de las notas. Con **Todas las notas**, las siete notas son un solo conjunto, **7-32 T7**: la clase de la escala menor armónica, y las notas son exactamente La♭ menor armónica. Dos maneras de mirar un mismo acorde: dos acordes conocidos apilados, o una escala sonando entera a la vez.

El acorde de Petrushka funciona igual. Do Mi Sol debajo de Fa♯ La♯ Do♯, de tres en tres, es **3-11 T1I|3-11 T7I**, Fa♯ mayor sobre Do mayor, y las seis notas juntas son **6-30 T1I**, la clase a la que a menudo se llama el acorde de Petrushka.

Notas en línea

`in_line` toma las notas una detrás de otra, en el orden en que empiezan (las que suenan juntas, de la más grave hacia arriba), y cada tantas notas forman un grupo. Aquí importan dos reglas:

- una nota repetida justo después de sí misma, a la misma altura, cuenta una vez: es la misma nota tocada otra vez;
- cualquier otra repetición cuenta: la misma nota en otra octava, o que vuelve más tarde, es una nota nueva de la línea.

La serie del Concierto op. 24 de Webern es el ejemplo clásico:

```
Si Si♭ Re | Mi♭ Sol Fa♯ | La♭ Mi Fa | Do Do♯ La
```

De tres en tres, cada grupo es **Menor 3** (3-3, [014]): Webern construyó toda la serie con una sola célula de tres notas, movida y dada vuelta. De seis en seis, las dos mitades son **6-20**, el conjunto hexatónico. Las doce en un solo grupo son el total cromático, 12-1.

Un grupo que repite el anterior se escribe una vez, donde empieza, como la hoja escribe un acorde solo donde cambia (`merge_repeats`).

Leer solo cuando algo cambia

Cortar una canción entera en grupos y nombrarlos lleva milisegundos en una canción larga: demasiado para repetirlo sesenta veces por segundo mientras suena. Así que la vista guarda lo que leyó, y de qué lo leyó:

```
struct Read {
    from: (Notation, Grouping, Option<usize>, Option<usize>),
    notes: Vec<HeardNote>,
    keys: KeyMap,
    shown: Rc<[Shown]>,
}
```

En cada fotograma reúne las notas de la canción, que es rápido, y las compara, junto con las elecciones y las tonalidades, con lo que leyó la última vez. Solo cuando algo es

distinto, una elección nueva en un menú, una nota editada en el piano roll, un cambio de tonalidad, vuelve a cortar y a nombrar los grupos.

El resultado es un `Rc<[Shown]>`, un puntero con contador de referencias a un slice. `Rc` es el primo de un solo hilo de `Arc` (ver Instrumentos: batería, sampler y Lineup Osc): clonarlo no copia los nombres, solo suma uno a un contador. La hoja de acordes lo clona en cada fotograma para dibujar los compases mientras la caché guarda el suyo, y los nombres se quedan en un solo lugar de la memoria. Toda la interfaz corre en un solo hilo, así que `Rc` basta; `Arc` también funcionaría, pero pagaría por una seguridad entre hilos que no hace falta.

Los grupos están ordenados por el paso en que empiezan. Para dibujar un compás, la hoja necesita los grupos que empiezan dentro de él, y `partition_point` los encuentra por **búsqueda binaria**:

```
let first = heard.partition_point(|shown| shown.start < bar.start);
let end = heard.partition_point(|shown| shown.start < bar.end());
let shown = heard[first..end].to_vec();
```

En un slice ordenado, «empieza antes del compás» es `true` para una primera parte y `false` para el resto, y `partition_point` devuelve dónde empieza la parte `false`. Mira unos $\log_2(n)$ elementos en lugar de todos: unos 11 para una canción con 2000 grupos.

Los nombres de los micromodos son largos, y «Acorde menor-mayor T9» no cabe en un cuarto de compás. Donde el nombre entero no cabe, un conjunto o un micromodo se escribe solo con su clase, «Acorde menor-mayor» o «3-11», y más allá como «…»; la indicación bajo el puntero da siempre el nombre entero y sus notas. Cómo hace caber la hoja los nombres en los compases está en La hoja de acordes: modelo, distribución y texto.

Las elecciones de los menús son un pequeño struct guardado en la memoria de egui, que la aplicación guarda en el disco, así que siguen ahí la próxima vez que arranca:

```
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
struct Choices {
    notation: Notation,
    grouping: Grouping,
    /// How many different notes a group of notes sounding together
    /// holds, or `None` for all of them.
    together: Option<usize>,
    /// How many notes a group of a line holds.
    line: usize,
}
```

`#[serde(default)]` sobre el struct significa que un campo que falta en lo guardado toma su valor de `Default`. Si una versión posterior añade un campo, las elecciones guardadas por esta siguen cargando.

Cómo se prueba

`set_class.rs` prueba la propia teoría: cada conjunto pertenece a una clase de su tamaño, la tabla guarda formas primas en orden con las cantidades de los libros, las clases conocidas tienen sus nombres conocidos (3-11, 4-28, 4-Z15, 7-35, 6-35, 5-35), Rahn y Forte difieren para 5-20, las clases en relación Z comparten sus intervalos, las formas normales juntan las notas al máximo, y el complemento de la escala mayor es la escala pentatónica.

`chord_sheet/notation.rs` prueba las vistas: las mismas notas en cada notación, y cada armonía renombrada con exactamente las notas de la escrita; los conjuntos que forman un acorde escritos como el acorde (3-11 T2 es Dm, Do Mi Sol La es Am7, las cuartas 4-23 T2 son A7sus4); dos acordes a la vez como dos conjuntos; los acordes de Petrushka y de los Augurios leídos en vertical; la serie de Webern leída en línea; y que cada nota de una línea cuenta salvo una repetida:

```
#[test]
fn sets_that_make_a_chord_are_written_as_the_chord() {
    assert_eq!(renamed(Notation::Chords, "3-11 T2"), "Dm");
    assert_eq!(renamed(Notation::Chords, "{C E G}"), "C");
    assert_eq!(renamed(Notation::Chords, "Acorde disminuido T11"),
"Bdim");
    // C E G A is both C6 and Am7: the simpler, more common chord.
    assert_eq!(renamed(Notation::Chords, "{C E G A}"), "Am7");
    // ...
}
```

Una prueba de la ventana (`ux_tests.rs`) toca el estilo Cuatro acordes, recorre el menú Mostrar, comprueba los nombres, y después vuelve atrás y comprueba que el proyecto está exactamente como estaba.

Los ejercicios sobre conjuntos están en Ejercicios: la teoría musical en código.

Series dodecafónicas y sus reglas

Archivos: `crates/lineup-core/src/harmony/row.rs`, `crates/lineup-core/src/serial.rs`, `crates/lineup-core/src/library/rows.rs`, `crates/lineup-core/src/harmony/voice_leading/serial.rs` y `crates/lineup-app/src/twelve_tone.rs`

Casi todo Ferrous Lineup supone música en una tonalidad: una nota que es la casa, acordes que se inclinan hacia ella, teclas de armonía numeradas de I a vii°. Una gran parte de la música del siglo XX no tiene tonalidad, y mucha se escribió con **series dodecafónicas** (*dodecafonismo*; en inglés, *twelve-tone rows*): las doce notas en un orden del que está hecha toda la obra. En la aplicación, **serie dodecafónica** es una de las clases de tonalidad; **Serie…**, en el piano roll, abre una ventana para escribir, dibujar o elegir una serie, ver su **matriz** de 48 formas y escribir cualquier forma en el patrón; la Biblioteca tiene un grupo de estilos de series, y la revisión de **Conducción de voces** conoce las reglas con las que escribían los compositores dodecafónicos. Este capítulo trata las cuatro cosas, y el Rust que hay detrás: un newtype que siempre es una serie de verdad, aritmética con doce números, backtracking, un builder que corre al compilar, conjuntos de bits y programación dinámica.

La música, en pocas palabras

En Do mayor, Do es la casa. Sol tira hacia ella, Si tira fuerte, y una obra que termina en Do suena terminada. Hacia 1921, Arnold Schoenberg buscó una manera de escribir música en la que ninguna nota fuera la casa. Su respuesta fue usar las doce notas por igual, en un orden fijo que elegía para cada obra: la **serie**. Ninguna nota vuelve hasta que hayan sonado las otras once, así que ninguna llega a ser el centro.

Esta es la serie de la Suite para piano op. 25 de Schoenberg, de 1925, una de las primeras obras construidas enteras a partir de una:

```
Mi  Fa  Sol  Do♯  Fa♯  Re♯  Sol♯  Re  Si  Do  La  La♯
```

Una serie sería aburrida si solo pudiera tocarse de una manera, así que viene en cuatro:

forma	cómo	la serie de Schoenberg sobre Mi
P, original	como está escrita	Mi Fa Sol Do♯ Fa♯ Re♯ Sol♯ Re Si Do La La♯
I, inversión	cada intervalo dado vuelta: donde P sube un semitono, I baja uno	Mi Re♯ Do♯ Sol Re Fa Do Fa♯ La Sol♯ Si La♯
R, retrogradación	P de atrás hacia adelante	La♯ La Do Si Re Sol♯ Re♯ Fa♯ Do♯ Sol Fa Mi
RI, retrogradación de la inversión	I de atrás hacia adelante	La♯ Si Sol♯ La Fa♯ Do Fa Re Sol Do♯ Re♯ Mi

Cada una de las cuatro puede empezar en cualquiera de las doce notas, lo que da **48 formas**. Se nombran con una letra y un número: el número es la nota en que empieza la forma, con Do como 0, Do♯ como 1, hasta Si como 11. Así, la serie de arriba es **P4** (empieza en Mi), e **I0** es su inversión empezando en Do. Una retrogradación toma el número de la forma que da vuelta: **R4** es P4 al revés, así que *termina* en Mi.

Algunos libros numeran las formas desde la primera nota de la propia serie, de modo que la serie tal como está escrita sea siempre P0. Entonces la serie de Schoenberg es P0, y lo que Ferrous Lineup llama I11 allí es I7. Las formas son las mismas; solo se mueven sus números. Ferrous Lineup numera una forma por la nota en que empieza, porque así una etiqueta nombra una nota que puedes encontrar en el teclado.

La matriz

Las 48 formas caben en un cuadrado de 12×12. La serie va arriba, su inversión desde la misma nota baja por el lado izquierdo, y cada fila *i* es la serie movida para empezar en la nota que está a la izquierda de esa fila:

```
      I4   I5   I7   I1   I6   I3   I8   I2   I11  I0   I9   I10
P4    Mi   Fa   Sol  Do♯  Fa♯  Re♯  Sol♯ Re   Si   Do   La   La♯    R4
P3    Re♯  Mi   Fa♯  Do   Fa   Re   Sol  Do♯  La♯  Si   Sol♯ La     R3
P1    Do♯  Re   Mi   La♯  Re♯  Do   Fa   Si   Sol♯ La   Fa♯  Sol    R1
P7    Sol  Sol♯ La♯  Mi   La   Fa♯  Si   Fa   Re   Re♯  Do   Do♯    R7
...
      RI4  RI5  RI7  RI1  ...
```

Una fila leída de izquierda a derecha es una forma P; de derecha a izquierda, una forma R. Una columna leída de arriba abajo es una forma I; de abajo arriba, una forma RI. Las etiquetas alrededor del cuadrado dicen cuál: P a la izquierda, R a la

derecha, I arriba, RI abajo. En la ventana Serie dodecafónica, haz clic en cualquier etiqueta para elegir esa forma, y su fila o su columna se ilumina.

La diagonal desde arriba a la izquierda hasta abajo a la derecha es la misma nota de punta a punta (Mi, aquí): una manera rápida de comprobar que una matriz está bien hecha.

Los grupos de notas tomados de una serie, como sus cuatro grupos de tres, son **conjuntos de clases de altura**, y la teoría que les pone nombre tiene un capítulo propio: Conjuntos de clases de altura, y una armonía con cuatro nombres.

Una serie en Rust: `ToneRow`

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
#[serde(try_from = "String", into = "String")]
pub struct ToneRow([u8; 12]);
```

Una serie son doce números, los semitonos de sus notas por encima de Do. Es un **newtype** alrededor de un array: el array es privado del módulo, así que las únicas maneras de hacer un `ToneRow` son funciones que lo comprueban:

```
pub fn new(semitones: [u8; 12]) -> Option<Self> {
    let set: PitchSet = PitchSet::from_semitones(&semitones);
    (set.len() == 12 && semitones.iter().all(|&n| n <
12)).then_some(ToneRow(semitones))
}
```

Se juntan los doce números en un conjunto de clases de altura, y si el conjunto tiene doce miembros, ninguna nota se repitió. Todo `ToneRow` del programa es, entonces, una serie de verdad: el **invariante** se comprueba una vez, aquí, y nada más tiene que preocuparse por él.

Archivos: `try_from` e `into`

Los dos atributos de serde del struct dicen: «para guardar un `ToneRow`, conviértelo en un `String`; para leer uno, lee un `String` e intenta convertirlo en un `ToneRow`». Las conversiones son implementaciones de traits comunes:

```
impl TryFrom<String> for ToneRow {
    type Error = String;

    fn try_from(text: String) -> Result<Self, String> {
        ToneRow::parse(&text)
    }
}

impl From<ToneRow> for String {
    fn from(row: ToneRow) -> Self {
        let numbers: Vec<String> =
row.0.iter().map(u8::to_string).collect();
        numbers.join(" ")
    }
}
```

Así, un archivo de proyecto guarda `"4 5 7 1 6 3 8 2 11 0 9 10"`, que una persona puede leer, en lugar de una lista de doce números entre corchetes. Guardar no puede fallar, así que es un `From` simple; leer sí puede, así que es un `TryFrom` con un error. Como la lectura pasa por `parse`, un archivo con una nota escrita dos veces se rechaza con un mensaje, en vez de dar una serie rota. `PitchSet` tiene el mismo par, así que un conjunto se guarda como `"0 3 6 9"`.

`parse` también lee el cuadro de arriba de la ventana Serie dodecafónica. Acepta nombres de notas en letras (`E F G Db Gb Eb Ab D B C A Bb`) o en solfeo (`Mi Fa Sol Reb Solb Mib Lab Re Si Do La Sib`), números (`4 5 7 1 …`) o la forma compacta de los libros de dodecafonismo, un carácter por nota con **T** y **E** para diez y once (`45716382E09T`). Cuando una nota viene dos veces, dice cuál, y qué nota falta en su lugar.

Un nombre se lee primero en letras, y en solfeo cuando eso falla. Ninguna sílaba es también un nombre en letras, porque ninguna empieza con una letra de la A a la G salvo `Do` y `Fa`, cuya segunda letra no es un sostenido ni un bemol, así que las dos nunca chocan. Antes de leer solfeo, `parse_name` escribe el nombre con una mayúscula y el resto en minúsculas, así que `SOL#` y `sib` se leen como `Sol#` y `Sib`, donde la `b` minúscula es un bemol:

```
fn parse_name(token: &str) -> Result<u8, String> {
    parse_letters(token).or_else(|error| {
        let mut chars = token.chars();
        let first = chars.next().map(|c| c.to_uppercase().collect::
<String>());
        let written = first.unwrap_or_default() +
&chars.as_str().to_lowercase();
        NoteName::read_as(&written, NoteNaming::Solfege)
            .filter(|(_, rest)| rest.is_empty())
            .map(|(name, _)| name.pitch_class().semitone())
            .ok_or(error)
    })
}
```

`or_else` corre su closure solo ante un error, y `ok_or(error)` devuelve el error de las letras cuando también falla el solfeo, así que un nombre equivocado se explica en los términos que más gente escribe.

Desde las notas lee una serie del piano roll: las doce primeras clases de altura distintas entre las notas seleccionadas (o todas), en el orden en que empiezan. `ToneRow::from_notes` se salta una nota cuya clase de altura ya está en la serie, así que puedes dibujar libremente y repetir notas, mientras aparezcan las doce. Las dos terminan en una tercera conversión, desde un `Vec<u8>`:

```
impl TryFrom<Vec<u8>> for ToneRow {
    type Error = String;

    fn try_from(semitones: Vec<u8>) -> Result<Self, String> {
        let array: [u8; 12] = semitones
            .try_into()
            .map_err(|v: Vec<u8>| trf("a row has 12 notes, not {}", &
[&v.len()]))?;
        ToneRow::new(array).ok_or_else(|| tr("a row has each of the 12
notes once").to_string())
    }
}
```

Un `Vec` puede tener cualquier cantidad de notas y un array exactamente doce, así que el `try_into` de la biblioteca estándar de uno al otro puede fallar. Cuando falla, devuelve el vector sin cambios como error, y `map_err` lo convierte en un mensaje que dice cuántas notas había. Después, `ok_or_else` convierte también el `None` de `new` en un error. Los closures solo corren cuando algo está mal, así que una serie correcta nunca arma un mensaje.

Todas las formas desde una función

```
pub fn form(self, label: RowLabel) -> ToneRow {
    let first = self.0[0];
    let n = label.number;
    let mut notes = self.0.map(|note| {
        let up = (note + 12 - first) % 12;
        if label.form.is_inverted() {
            (n + 12 - up) % 12
        } else {
            (n + up) % 12
        }
    });
    if label.form.is_backwards() {
        notes.reverse();
    }
    ToneRow(notes)
}
```

Primero se mide cada nota como `up`, su distancia por encima de la primera nota de la serie (de 0 a 11). Una forma P la pone a la misma distancia por encima del nuevo comienzo `n`; una forma I, a la misma distancia *por debajo*. Eso es todo lo que es una inversión. El `+ 12` antes de cada `%` evita que los números bajen de cero: son `u8`, y `4 - 7` no daría la vuelta a la octava, sino que detendría el programa en una compilación de depuración.

`self.0.map(...)` es el `map` de un **array**, no de un iterador: llama al closure con cada uno de los doce elementos y devuelve un `[u8; 12]` nuevo, sin `Vec` y sin `collect`. La longitud es parte del tipo, así que el resultado sigue siendo exactamente doce notas.

Con `form`, la matriz son dos líneas:

```
pub fn matrix(self) -> [[u8; 12]; 12] {
    let inversion = self.form(RowLabel::new(RowForm::Inversion,
self.0[0]));
    inversion
        .0
        .map(|start| self.starting_on(PitchClass::from_semitone(start)).0)
}
```

La inversión desde la primera nota de la serie es la columna de la izquierda; cada una de sus notas empieza una fila.

En el otro sentido, «¿qué forma de esta serie es esa?» prueba las 48 hasta que una coincide:

```
pub fn label_of(self, other: ToneRow) -> Option<RowLabel> {
    RowForm::ALL.into_iter().find_map(|form| {
        (0..12)
            .map(|n| RowLabel::new(form, n))
            .find(|&label| self.form(label) == other)
    })
}
```

`find_map` se detiene en la primera forma que da `Some`, y `find` en el primer número que coincide, así que la búsqueda termina en cuanto encuentra la forma.

Qué hace interesante a una serie

Debajo de la matriz, la ventana Serie dodecafónica dice qué tiene de especial la serie. Cada línea es una función corta.

Los pasos. `intervals()` da los once pasos de nota a nota, en semitonos hacia arriba. Para la serie de Schoenberg son `1 2 6 5 9 5 6 9 1 9 1`. Se construye con `std::array::from_fn`, que hace un array llamando a un closure con cada índice:

```
pub fn intervals(self) -> [u8; 11] {
    std::array::from_fn(|i| (self.0[i + 1] + 12 - self.0[i]) % 12)
}
```

El tipo de retorno dice once, así que `from_fn` llama al closure once veces, e `i + 1` nunca pasa de la duodécima nota.

Series de todos los intervalos. Cuando los once pasos son todos distintos, cada intervalo, de un semitono a una séptima mayor, aparece exactamente una vez. La Suite lírica de Alban Berg (1927) empieza con una serie así: Fa Mi Do La Sol Re Sol♯ Do♯ Re♯ Fa♯ La♯ Si, con el tritono justo en el medio. Son raras: de las 39.916.800 series que empiezan en Do, solo 3.856 son de todos los intervalos. Así que el botón 🎲 **Todos los intervalos** no puede limitarse a barajar notas hasta dar con una; construye la serie nota a nota:

```rust
fn all_interval_from(notes: &mut Vec<u8>, random: &mut Random) ->
Option<Vec<u8>> {
    if notes.len() == 12 {
        return Some(notes.clone());
    }
    let used: Vec<u8> = notes.windows(2).map(|w| (w[1] + 12 - w[0]) %
12).collect();
    let mut intervals: Vec<u8> = (1..12).filter(|i|
!used.contains(i)).collect();
    for i in (1..intervals.len()).rev() {
        intervals.swap(i, random.below(i + 1));
    }
    let last = *notes.last()?;
    for interval in intervals {
        let next = (last + interval) % 12;
        if notes.contains(&next) {
            continue;
        }
        notes.push(next);
        if let Some(row) = all_interval_from(notes, random) {
            return Some(row);
        }
        notes.pop();
    }
    None
}
```

Esto es **backtracking** (vuelta atrás): probar cada paso todavía no usado, en un orden al azar; si la nota en la que cae está libre, añadirla y seguir desde ahí (la función se llama a sí misma); si eso no lleva a ninguna parte, quitar la nota (`pop`) y probar el paso siguiente. Cuando ningún paso sirve, devolver `None` para que quien llamó quite *su* nota. `windows(2)` entrega cada par de vecinas, `[a, b]`, que es justo lo que es un paso. Todas las llamadas comparten un vector a través de `&mut`, así que probar una nota cuesta un `push` y un `pop`, no una copia.

Series al azar. El botón 🎲 **Al azar** baraja las doce notas con el **algoritmo de Fisher-Yates**, el mismo bucle que baraja los intervalos de arriba: recorrer desde el último lugar hasta el primero, e intercambiar cada uno con un lugar elegido al azar entre los que aún no se han hecho. Así todos los órdenes son igual de probables. Los números al azar vienen de un generador diminuto, *SplitMix64*, escrito en unas pocas líneas en vez de añadir una biblioteca:

```
pub(crate) fn next(&mut self) -> u64 {
    self.0 = self.0.wrapping_add(0x9E37_79B9_7F4A_7C15);
    let mut z = self.0;
    z = (z ^ (z >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
    z = (z ^ (z >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
    z ^ (z >> 31)
}
```

`wrapping_add` y `wrapping_mul` dicen «aquí desbordar está bien, que dé la vuelta»: un `+` normal que desborda detendría una compilación de depuración. La misma semilla da siempre la misma serie, lo que hace que las pruebas se puedan repetir. `pub(crate)` deja que otros módulos de `lineup-core` usen el generador (el entrenamiento auditivo elige sus preguntas con él) sin que forme parte de la API pública del crate.

Grupos de notas. Los compositores suelen tocar una serie en grupos: cuatro tricordios (tres notas cada uno), tres tetracordios, dos hexacordios. `segments(size)` corta la serie con `chunks_exact`, que entrega porciones de exactamente `size` notas y deja fuera una última más corta, y la ventana nombra la clase de conjunto de cada grupo (ver Conjuntos de clases de altura). Los tetracordios de la serie de Schoenberg son tres clases distintas, 4-12, 4-Z15 y 4-1; los tricordios de la serie Cuatro tríadas son todos 3-11, las tríadas mayores y menores.

Combinatoriedad. Toma las seis primeras notas de una serie. Puede que otra forma de la misma serie empiece exactamente con las *otras* seis. Toca las dos formas a la vez, y cada mitad de la música tiene las doce notas, sin que ninguna vuelva antes de tiempo. Schoenberg construyó sus obras tardías sobre esto, y se llama **combinatoriedad hexacordal**:

```
pub fn combinatorial_forms(self) -> Vec<RowLabel> {
    let missing = PitchSet::from_semitones(&self.0[..6]).complement();
    [
        RowForm::Prime,
        RowForm::Inversion,
        RowForm::RetrogradeInversion,
    ]
    .into_iter()
    .flat_map(|form| (0..12).map(move |n| RowLabel::new(form, n)))
    .filter(|&label| PitchSet::from_semitones(&self.form(label).0[..6]) ==
missing)
    .collect()
}
```

La retrogradación queda fuera porque siempre funciona (su primera mitad es la segunda mitad de la serie), así que no dice nada de la serie. La serie de la op. 25 de Schoenberg es combinatoria con I11: sus seis primeras notas, Mi Fa Sol Do♯ Fa♯ Re♯, y

las seis primeras de I11, Si La♯ Sol♯ Re La Do, forman las doce. La Serie de Lineup, escrita para Ferrous Lineup, es combinatoria con I3.

El `move` de `move |n|` hace falta porque `flat_map` saca el closure de dentro fuera del de afuera, y ese closure usa `form`: `move` hace que se lleve su propia copia de `form` en vez de tomar prestada una variable que está a punto de desaparecer.

Las series para empezar

El menú **Series famosas y de ejemplo** tiene ocho series. Cuatro vienen de obras famosas, y cuatro se escribieron para Ferrous Lineup para mostrar lo que puede hacer una serie. `ROW_PRESETS` es un array `const` de ellas, cada una con un nombre, de dónde viene y una frase sobre ella.

serie	de	qué muestra
Mi Fa Sol Do♯ Fa♯ Re♯ Sol♯ Re Si Do La La♯	Schoenberg, Suite para piano op. 25 (1925)	una de las primeras obras dodecafónicas; sus cuatro últimas notas, Si Do La La♯, deletrean BACH al revés con los nombres alemanes de las notas, donde B es Si♭ y H es Si
Fa Mi Do La Sol Re Sol♯ Do♯ Re♯ Fa♯ La♯ Si	Berg, Suite lírica (1927)	una serie de todos los intervalos, con el tritono en el medio
La Fa♯ Sol Sol♯ Mi Fa Si La♯ Re Do♯ Do Re♯	Webern, Sinfonía op. 21 (1929)	un palíndromo: al revés, es la misma serie un tritono más arriba (R9 es P3)
Si La♯ Re Re♯ Sol Fa♯ Sol♯ Mi Fa Do Do♯ La	Webern, Concierto op. 24 (1934)	cuatro grupos de tres notas, todos del mismo micromodo, Menor 3
Do Re Fa♯ Sol♯ Fa Mi La Do♯ Si Re♯ Sol La♯	Serie de Lineup	combinatoria por hexacordios con I3
Do Mi Sol Re Fa♯ La	Cuatro tríadas	cuatro tríadas que aun así usan cada nota una vez

serie	de	qué muestra
La♯ Do♯ Fa Si Re♯ Sol♯		
Do Mi Re♯ Sol Sol♯ Si Fa Re Do♯ La La♯ Fa♯	Serie espejo	un palíndromo, como la de Webern
Do Re Fa Mi Si La Re♯ Sol Sol♯ Do♯ La♯ Fa♯	Todos los intervalos	una serie de todos los intervalos

Las obras de 1925 a 1929 son de dominio público en los Estados Unidos (95 años después de su publicación), y sus compositores murieron en 1951, 1935 y 1945, así que también lo son donde los derechos de autor duran 70 años después de la muerte del autor. El Concierto de Webern se publicó en 1948, así que es de dominio público donde los derechos duran 70 años después de la muerte del autor. Solo se usan las series: doce notas cada una.

Algunas series no tienen 48 formas *distintas*. Como la R9 de Webern es su P3, cada forma R de esa serie es también una forma P, y cada forma RI una forma I: solo 24 son distintas. Con la serie de Berg y la Serie espejo pasa lo mismo: leída al revés, cada una es ella misma un tritono más arriba. `label_of` prueba primero las formas P, así que una forma así se nombra como la forma P que también es.

De una serie a notas: el serialismo

Una serie dice qué nota viene después, y nada más. Para hacer música con ella, alguien todavía tiene que decidir cuánto dura cada nota, qué tan fuerte suena, en qué octava, y si algunas notas suenan juntas. `RowWriting` guarda esas decisiones, y los controles de debajo de la matriz las fijan:

```
pub struct RowWriting {
    /// The form to start with, such as `P0`.
    pub label: RowLabel,
    pub chain: Chain,
    /// How many notes of the row sound together: 1 for a melody, 3 to
    /// play the row as four three-note chords (trichords), 6 for two
    /// hexachords.
    pub together: usize,
    /// How long each note (or chord) lasts, in steps, when the durations
    /// do not follow the row.
    pub step: usize,
    pub register: Register,
    /// The lowest note the row is written from.
    pub low: Note,
    /// Whether the durations follow the row (see the module
documentation).
    pub durations: bool,
    /// Whether the velocities follow the row's inversion.
    pub dynamics: bool,
    /// Whether the octaves follow the row's retrograde.
    pub octaves: bool,
}
```

Después (`chain`) dice qué viene tras la forma hasta llenar el patrón: nada (**Una vez**), la misma forma otra vez (**Repetir**), o las cuatro formas por turno con el mismo número (**P, R, I, RI**). **Juntas** es `together`, y **Cada** es `step`. Los tres **registros** (el menú **Octavas**) deciden las octavas cuando la serie no lo hace:

- **Cerrado** pone cada nota en la octava por encima de la nota más grave;
- **Suave** pone cada nota tan cerca como pueda de la anterior, lo que da una línea cantable (un tritono, justo a mitad de camino, sube);
- **Amplio** manda las notas grave, aguda, media, grave, aguda, media por tres octavas, las líneas a saltos de Anton Webern.

Serialismo integral

Hacia 1950, Olivier Messiaen, Pierre Boulez y Milton Babbitt llevaron la idea más lejos: si las alturas siguen una serie, ¿por qué no también las duraciones, las dinámicas y las octavas? Eso es el serialismo **integral** (o total). Las obras reales traducen las series a duraciones y dinámicas de muchas maneras; Ferrous Lineup usa tres sencillas, cada una leyendo una forma distinta de la serie para que las series no avancen a la par:

- **Duraciones**: los números de la serie (la distancia de cada nota por encima de la primera, de 0 a 11) se vuelven duraciones. La nota *k*-ésima dura tantos pasos como el número *k*-ésimo, más uno, así que de 1 a 12 semicorcheas.

- **Dinámicas**: los números de la inversión se vuelven doce velocidades, de muy suave a muy fuerte: `28 + 9 * level`, de 28 a 127.
- **Octavas**: los números de la retrogradación, contados de tres en tres (`% 3`), eligen una de tres octavas.

Para la serie de Schoenberg los números son 0 1 3 9 2 11 4 10 7 8 5 6, así que con **Duraciones** marcada sus notas duran 1, 2, 4, 10, 3, 12… pasos. Dentro de `write_row`, cada nota hace las tres preguntas por turno:

```
let lasts = if writing.durations {
    usize::from(numbers[place % 12]) + 1
} else {
    step
};
// ...
let octave = if writing.octaves {
    // The retrograde's numbers, in threes.
    Some(numbers[11 - k % 12] % 3)
} else if writing.register == Register::Wide {
    // Low, high, middle, and round again, against the row.
    Some([0, 2, 1][k % 3])
} else {
    None
};
// ...
let velocity = if writing.dynamics {
    // The inversion's numbers.
    dynamic((12 - numbers[k % 12]) % 12)
} else {
    crate::DEFAULT_VELOCITY
};
```

`place` cuenta notas a través de las formas, así que con **P, R, I, RI** la serie de duraciones sigue de una forma a la siguiente en vez de volver a empezar. `k` es `place` más el lugar de la nota en su acorde, así que cada nota de un acorde tiene su propia octava y su propia intensidad.

`write_row` es una función pura: entra una serie, las decisiones y la duración de un patrón; sale una lista de `NoteEvent`. La ventana la llama en cada fotograma para avisar cuando el patrón es demasiado corto para toda la serie («El patrón solo tiene lugar para 9 de las 12 notas»), y **Escribir P0 en el patrón actual** manda las mismas notas al piano roll, que reemplaza las notas del canal con la edición de siempre, así que **Ctrl+Z** trae de vuelta las anteriores. **Guardar en un patrón nuevo** las pone en un patrón nuevo.

Una serie como tonalidad

serie dodecafónica en los menús de tonalidad, o **Usar como tonalidad** en la ventana Serie dodecafónica para la forma que elegiste, hace de la serie la tonalidad. En el código es una clase más de tonalidad, `ScaleKind::Row(ToneRow)`, que lleva su serie consigo (Escalas, modos y tonalidades muestra cómo la trata el resto del código). La serie se guarda empezando en Do, y la fundamental de la tonalidad dice dónde empieza de verdad, así que P4 de la serie de Schoenberg se guarda así (el archivo escribe las notas siempre con letras):

```
scale: (root: E, kind: Row("0 1 3 9 2 11 4 10 7 8 5 6")),
```

Con una serie como tonalidad, el piano roll numera sus notas del **1 al 12** en el teclado, en el orden de la serie, así que puedes dibujar la serie siguiendo los números. Las teclas de armonía tocan los grupos de la serie en lugar de acordes, como arman sus acordes los compositores dodecafónicos: **3 notas** da cuatro teclas para los tricordios, **1–3**, **4–6**, **7–9** y **10–12**; **4 notas** da tres teclas, y **6 notas**, dos. Salen de cortar la serie en orden:

```
ScaleKind::Row(_) => {
    let group = row_group(size);
    Some(
        notes
            .chunks(group)
            .enumerate()
            .map(|(i, chunk)| {
                let label = format!("{}-{}", i * group + 1, i * group +
chunk.len());
                GroupKey::new(scale, Some(label), chunk.to_vec())
            })
            .collect(),
    )
}
```

Cambiar la tonalidad de Do mayor a una serie no mueve ninguna nota, porque no hay un lugar sensato al que pueda ir Mi en una serie; cambiar a la misma serie sobre otra nota las mueve todas, que es transportar. Pasar de una serie a **conjunto de clases de alturas** se queda con las tres primeras notas de la serie como conjunto, que son las que tocaban las teclas de armonía (ver Una tonalidad como conjunto: las mismas notas, otro nombre).

Estilos y demostraciones dodecafónicos

El grupo **Series dodecafónicas** de la Biblioteca tiene siete estilos:

estilo	qué hace
Melodía de la serie	la Serie de Lineup como melodía en corcheas, por P, R, I y RI
Coral de tríadas	la serie Cuatro tríadas como acordes, sobre su retrogradación en el bajo: música dodecafónica que suena casi tonal
Pads de hexacordios	media serie como acorde, las otras seis notas desde I3 como melodía: cada compás tiene las doce notas
Puntillismo	notas sueltas que saltan por tres octavas, con duraciones y dinámicas de la serie, como en Webern
Serialismo integral	alturas, duraciones, dinámicas y octavas, todas de la serie de todos los intervalos, sobre su retrogradación de la inversión en acordes de dos notas
Lo-fi dodecafónico	los tricordios de la serie sobre un ritmo lo-fi, con su forma original en el bajo: una serie también puede tener groove
Los tricordios de Webern	la serie del Concierto de Webern, con su melodía saltando por tres octavas sobre sus grupos de tres como acordes

Un estilo armónico es una progresión de acordes y una manera de tocarla. Un estilo de serie no tiene progresión, así que es una serie y una lista de **voces**, cada una diciendo cómo toca la serie un instrumento. Como el resto de la Biblioteca, son datos:

```
RowStyle {
    name: "Triad chorale",
    // ...
    row: "Four triads",
    tempo: 72.0,
    time: TimeSignature::COMMON,
    bars: 4,
    voices: &[
        voice(KEYS, RowForm::Prime)
            .chain(Chain::FourForms)
            .chords(3, 4)
            .from(Register::Nearest, 55),
        voice(BASS, RowForm::Retrograde)
            .step(8)
            .from(Register::Close, 36),
    ],
    rhythm: "",
},
```

`voice` hace una voz simple (una melodía en corcheas desde el Do central), y cada método cambia una cosa y devuelve la voz nueva:

```
impl RowVoice {
    const fn offset(self, offset: u8) -> Self {
        RowVoice { offset, ..self }
    }
    const fn chain(self, chain: Chain) -> Self {
        RowVoice { chain, ..self }
    }
    // ...
}
```

Tomar `self` por valor y devolver una copia cambiada con la **sintaxis de actualización de structs** (`..self`) deja encadenar las llamadas, como un builder. Y como son `const fn`, toda la tabla puede ser un `static`: `pub static ROW_STYLES: [RowStyle; 7]` se construye al compilar, sin código que corra al arrancar.

Aritmética de formas

Las demostraciones de **Archivo > Demostraciones > Dodecafónico** tocan cada estilo cuatro veces, como las secciones de una canción: **P: la serie**, **R: al revés**, **I: dada vuelta** y **RI: las dos**, y luego P una vez más para terminar. Con un ritmo, la batería entra después de la primera sección, con un fill antes de la última. Pero las voces de un estilo ya usan formas distintas: el bajo del Coral de tríadas toca R. ¿Qué toca el bajo en la sección R? R de R, que es la serie hacia adelante otra vez: P.

Las cuatro formas se comportan como dos interruptores, «dada vuelta» y «al revés», y combinar dos formas cambia cada interruptor una vez por cada forma que lo tiene encendido:

```
pub fn combine(section: RowForm, voice: RowForm) -> RowForm {
    let flags = |form| match form {
        RowForm::Prime => (false, false),
        RowForm::Inversion => (true, false),
        RowForm::Retrograde => (false, true),
        RowForm::RetrogradeInversion => (true, true),
    };
    let (a, b) = (flags(section), flags(voice));
    match (a.0 != b.0, a.1 != b.1) {
        (false, false) => RowForm::Prime,
        (true, false) => RowForm::Inversion,
        (false, true) => RowForm::Retrograde,
        (true, true) => RowForm::RetrogradeInversion,
    }
}
```

`a.0 != b.0` sobre dos `bool` es un *o exclusivo*: verdadero cuando exactamente uno lo es. Los matemáticos llaman a este pequeño sistema el **grupo de Klein**: cada forma se deshace a sí misma (I de I es P), y el orden no importa. Un detalle más: la melodía de Pads de hexacordios toca I3, tres semitonos *por encima* de la primera nota de la serie. Dada vuelta, «tres por encima» pasa a ser «tres por debajo», así que en las secciones I y RI ese desplazamiento pasa a ser 12 − 3 = 9:

```
let offset = if inverted {
    (12 - self.offset % 12) % 12
} else {
    self.offset % 12
};
```

Tocar una canción entera como una serie

Como con los demás estilos, un estilo de serie se puede probar de dos maneras: parado, se escribe en un patrón propio; mientras suena la canción, se adueña de toda la canción. `use_row_style` recorre cada patrón que tiene acordes, una línea de bajo o una melodía, y escribe el estilo allí, leyendo los patrones por turno como P, R, I y RI para que la canción no se repita. La tonalidad pasa a ser la serie, empezando en la tónica de la canción, y los finales quedan como están.

Una regla decide qué pasa con las partes propias de la canción: **el estilo se adueña de la armonía**. Acordes tonales o un bajo tonal bajo una serie solo chocan, así que los canales de acordes y de bajo que el estilo no usa se callan en ese patrón, y los

acordes del estilo van al canal que tocaba los acordes de la canción. La melodía se queda, salvo que el estilo tenga una propia:

```
if chords.is_some() || plays(project, bass) || melody {
    let before = project.pattern(pattern).cloned();
    let quiet: Vec<usize> = [(KEYS, chords), (BASS, bass)]
        .into_iter()
        .filter(|(name, _)| !uses(name))
        .filter_map(|(_, channel)| channel)
        .collect();
    // ... empty the `quiet` channels, write the voices ...
    if project.pattern(pattern) != before.as_ref() {
        written += 1;
        change.harmony += 1;
    }
}
```

Para saber si un patrón cambió de verdad, guarda una copia de antes (`cloned()` convierte el `Option<&Pattern>` en un `Option<Pattern>`) y compara al final. `as_ref()` vuelve a convertir la copia en un `Option<&Pattern>`, para que los dos lados de `!=` tengan el mismo tipo. Esta función, `take_over_harmony`, recibe la escritura como un closure, así que los estilos de micromodos la usan también.

Una ventana que se abre delante

egui pone delante de las demás una ventana que se abre. Pero la ventana cuyo botón **Serie…** la abrió recibió un clic en el mismo fotograma, y un clic sube la ventana donde cae, así que las dos quieren estar arriba, y la nueva puede quedar detrás. `Front` mantiene delante una ventana nueva durante sus primeros fotogramas:

```rust
#[derive(Default)]
pub(crate) struct Front {
    /// The frame the window was last drawn in.
    drawn_in: u64,
    /// The frame it opened in.
    opened_in: u64,
}

impl Front {
    /// Called each frame the window `id` is drawn, before it is.
    pub(crate) fn keep(&mut self, ctx: &egui::Context, id: egui::Id) {
        let frame = ctx.cumulative_pass_nr();
        if self.drawn_in + 1 < frame {
            self.opened_in = frame;
        }
        self.drawn_in = frame;
        if frame - self.opened_in < 3 {
            ctx.move_to_top(egui::LayerId::new(egui::Order::Middle, id));
            ctx.request_repaint();
        }
    }
}
```

`cumulative_pass_nr` cuenta los fotogramas que egui ha dibujado. Una ventana que no se dibujó en el fotograma anterior acaba de abrirse, así que recuerda ese fotograma; durante sus tres primeros fotogramas sube su propia capa arriba de todo y pide otro fotograma, de modo que los tres llegan enseguida en vez de esperar a que se mueva el ratón. Para encontrar su capa, la ventana necesita su id, así que las ventanas Serie dodecafónica, Conjuntos de clases de alturas y Micromodos ponen cada una el suyo con `.id(...)`, hecho con `egui::Id::unique` (*unique* porque nombra una sola cosa en todo el programa, como debe hacerlo el id de una ventana). La lista de reproducción hace lo mismo cuando un doble clic abre un clip en el piano roll (Doble clic en un clip para editarlo).

La ventana Serie dodecafónica es tan ancha como la matriz y se desplaza cuando la pantalla es baja, para que sus botones de abajo sigan a mano: `.resizable([false, true])` la deja hacerse más alta o más baja pero no más ancha, y `.constrain_to(area)` la mantiene dentro de la ventana principal.

Las reglas de la escritura dodecafónica

La revisión de la conducción de voces marca lo que evita la escritura a varias voces clásica: quintas paralelas, una sensible que no sube, una séptima que no baja. Cada una de esas reglas tiene que ver con una tonalidad. En la música escrita con una serie no hay tónica, así que allí no significan nada. La música dodecafónica tiene reglas

propias. Schoenberg escribía con ellas, y también sus alumnos Berg y Webern; los cursos de composición todavía las enseñan. Así que cuando la tonalidad es una **serie dodecafónica**, la casilla **Conducción de voces** revisa esas reglas en su lugar.

Lo que ves

- **Una línea Serie** debajo de los nombres de los acordes. Nombra cada forma de la serie que la revisión encuentra en las notas, como **P4** o **P10**, encima de las notas que la tocan. Pasa el puntero por encima para leer las notas de la forma: «P4: la serie que empieza en Mi: Mi Fa Sol Do# Fa# Mib Lab Re Si Do La Sib.»
- **La línea Voces**, debajo, marca lo que evita la escritura dodecafónica: **orden** para una nota fuera del orden de la serie, **rep.** para una nota repetida antes de tiempo, **hueco** para una forma sin terminar, **8va** para una nota duplicada a la octava, **tonal** para un acorde que suena a armonía tonal.
- **Contornos** alrededor de las notas en cuestión. Dos notas duplicadas se unen con una línea, y también las notas de un acorde tonal. Un contorno punteado muestra la nota que la serie pedía en su lugar.
- **La línea de resumen**, encima de las filas de acordes, lo resume todo: «Reglas dodecafónicas, P4, P10, a través de todas las pistas juntas: fuera del orden de la serie (2), duplicación a la octava (1), suena tonal (1).» Pasa el puntero por un paso con un hallazgo y la línea dice qué pasa allí; pasa el puntero por la línea para leer por qué.

Con un **conjunto de clases de alturas** como tonalidad, no se revisa nada. Un conjunto es una colección de notas sin orden y sin reglas propias, así que no hay nada contra qué revisar, y la línea de debajo de los acordes lo dice.

Las cinco reglas

La serie fija el orden. Una obra avanza por formas de la serie, cada una tocada nota tras nota, en su orden, para que todo lo que oyes venga de la serie. Notas vecinas en la serie pueden sonar juntas como acorde, en cualquier orden de abajo arriba, y una nota puede repetirse enseguida, o hacer un trino con la nota anterior: sigue siendo la misma nota de la serie. Dos errores se cuentan con cuidado. Si **intercambias** dos notas, o **dejas una fuera**, eso es una nota fuera de lugar, no todas las que vienen después: con Si tocado antes que Re en P4, la revisión marca el Si («Melodía toca Si4 donde P4 pide Re, su nota 8») y encuentra el resto de la serie en orden otra vez.

Ninguna nota vuelve antes de tiempo. Antes de que una nota vuelva, suenan las otras once. La serie se hizo para eso: ninguna nota se oye más que las demás, así que

ninguna se vuelve tónica. Una nota que vuelve después de otras notas, antes de que la forma haya hecho sonar las doce, se marca con **rep.**

Una forma se toca entera. Cuando una forma se corta y empieza la siguiente, las notas que dejó fuera faltan entre las doce. La revisión marca la última nota de una forma así con **hueco** y nombra lo que nunca sonó: «Serie sin terminar: P4 se detiene después de 8 de sus 12 notas, con Melodía en Re5; Si, Do, La y Sib nunca suenan antes de que empiece la forma siguiente.»

Sin duplicaciones a la octava. Duplicar una nota a la octava le da peso. Como explicó Schoenberg, una nota con peso puede oírse como fundamental, o incluso como tónica. La revisión marca dos partes que tocan la misma nota a la vez, y también una parte que toca una nota mientras otra parte todavía la sostiene. Una parte que duplica a otra de principio a fin, una línea en octavas, se dice una sola vez, donde empieza.

Sin acordes de armonía tonal. Una tríada mayor o menor, o una séptima de dominante, le recuerda al oído la música tonal, y entonces el oído espera que la música siga como seguiría la música tonal. Schoenberg y Webern los evitaban. La revisión marca un momento en que todo lo que suena a la vez forma uno de esos acordes, y lo nombra. **Berg** los usaba a propósito: la serie de su Concierto para violín (1935) está hecha de tríadas, Sol menor, Re mayor, La menor y Mi mayor. Así que lee **tonal** como «esto suena a una tonalidad», no como «esto está mal».

Leer la serie como lo haría un analista

Nada en las notas dice qué forma de la serie suena, ni dónde termina una forma y empieza la siguiente. Un estudiante que analiza una partitura tiene que averiguarlo, y la revisión también. Corta las notas en **grupos**, notas que empiezan juntas, prueba luego todas las maneras de repartir los grupos entre formas de la serie, y se queda con la que tiene menos problemas: cada nota fuera de orden, cada nota repetida antes de tiempo y cada nota que una forma deja fuera cuenta uno. Algunas cosas acercan la lectura a lo que hacen de verdad los compositores:

- **Las formas pueden solaparse en la unión.** Una forma puede empezar en la nota, o el acorde, en que terminó la forma anterior, de modo que las dos la comparten: una **elisión**. Si P4 termina con el acorde La Si♭ Do y P10 empieza en Si♭, P10 toma el Si♭ de ese acorde y sigue con Si.
- **Un acorde puede compartir notas con el anterior.** Esas notas cuentan para la forma nueva si las necesita, o como notas repetidas si no.
- **Una serie, o una por pista.** Todas las pistas pueden compartir una forma, como suelen hacerlo una melodía y los acordes que la acompañan, o cada pista puede

tocar formas propias, como en un canon. La revisión lee la serie de las dos maneras, se queda con la lectura con menos problemas, y la línea de resumen dice cuál eligió.
- **Los patrones se enlazan.** Cada patrón se revisa por separado, pero una serie puede seguir de un patrón al siguiente. Así, la primera forma puede haber empezado en el patrón anterior, siempre que sus notas de aquí estén en orden: la línea Serie dice entonces «…P4», y el resumen «P4 desde la nota 3». La última forma puede seguir en el patrón siguiente, así que nunca se marca como sin terminar.
- **Empates.** Cuando dos lecturas tienen la misma cantidad de problemas, la revisión prefiere formas que empiezan en su primera nota. Si dos formas siguen empatadas, nombra la primera en el orden P, I, R, RI. Los acordes pueden hacer que dos formas no se puedan distinguir: la serie del Concierto de Webern son cuatro grupos de tres notas, y sus formas R0 e I1 tienen las mismas tres notas en cada grupo, en otro orden. En el estilo Los tricordios de Webern, la melodía y los acordes tocan los dos R0 después de P0. Las notas de la melodía llegan de a una, así que solo pueden ser R0; los acordes hacen sonar cada grupo de una vez, así que encajan igual de bien con R0 que con I1, y la revisión los nombra I1, que va primero. Cualquiera de los dos nombres es correcto.

La revisión sigue las reglas clásicas del método de Schoenberg. No juzga la **música atonal libre**, escrita sin ninguna serie (para eso elige un conjunto de clases de alturas como tonalidad); el **serialismo más allá de las alturas**, si las duraciones, las dinámicas o los registros siguen también una serie; ni la **combinatoriedad**: en el estilo Pads de hexacordios lee la forma propia de cada pista, lo cual es correcto, pero no dice que las dos mitades juntas forman las doce notas.

Cómo lee la serie la revisión

Los lugares de una forma en dieciséis bits

`Walk` recorre una forma de la serie, grupo por grupo, y recuerda cuáles de los doce lugares han sonado. Son doce respuestas de sí o no, así que las guarda como los bits de un `u16`, un bit por lugar:

```rust
#[derive(Debug, Clone, Copy)]
struct Walk {
    /// One bit for each place in the form, from the first.
    used: u16,
    /// The place after the furthest one sounded.
    next: usize,
    errors: usize,
}

impl Walk {
    fn from(from: usize) -> Walk {
        Walk {
            used: (1 << from) - 1,
            next: from,
            errors: 0,
        }
    }

    fn heard(self) -> usize {
        self.used.count_ones() as usize
    }

    fn has(self, place: usize) -> bool {
        self.used & (1 << place) != 0
    }
    // ...
}
```

`Walk::from(3)` empieza un recorrido cuyos tres primeros lugares ya sonaron, en el patrón anterior o como las notas compartidas de una elisión: `(1 << 3) - 1` es `0b111`. `Walk` es `Copy`, dos palabras de máquina y un conjunto de bits, así que probar una forma no cuesta nada de preparar. `count_ones` cuenta los lugares oídos en una sola instrucción del procesador en la mayoría de las máquinas. Los conjuntos de clases de altura usan la misma idea, con doce bits para las doce notas.

¿Cuándo están en orden las notas nuevas?

Un grupo de *n* notas nuevas está en orden cuando llena los *n* lugares siguientes de la forma, en cualquier orden entre ellas. «Los lugares siguientes» puede querer decir dos cosas, y `window` prueba las dos:

```
fn open(self, places: std::ops::Range<usize>, count: usize) -> u16 {
    places
        .filter(|&place| !self.has(place))
        .take(count)
        .fold(0, |bits, place| bits | 1 << place)
}

fn window(self, taken: u16, count: usize) -> u16 {
    let first_open = self.open(0..12, count);
    let ahead = self.open(self.next..12, count);
    match (taken & ahead).count_ones() > (taken & first_open).count_ones()
{
        true => ahead,
        false => first_open,
    }
}
```

`open` arma un conjunto de lugares con un `fold` que añade con OR un bit por lugar. `first_open` son los primeros lugares todavía abiertos, lo que recupera una nota saltada justo antes. `ahead` son los lugares después de la nota más lejana oída, lo que sigue después de un salto. Gana el que tiene más notas del grupo.

Toma P4 con Re y Si intercambiadas: Mi Fa Sol Do♯ Fa♯ Mi♭ La♭ **Si Re** Do La Si♭. En Si, la forma pide Re, así que Si está fuera de lugar. En Re, el primer lugar abierto es el de Re, así que Re está en orden, y también todo lo que sigue.

Ahora toma P4 sin el Re: Mi Fa Sol Do♯ Fa♯ Mi♭ La♭ **Si** Do La Si♭. Si está otra vez fuera de lugar. En Do, el primer lugar abierto sigue siendo el de Re, que nunca llega. Pero el lugar después de la nota más lejana oída, Si, es el de Do, así que Do está en orden, y también La y Si♭. Solo se marca Si. Con `first_open` solo, cada nota después del Re que falta estaría fuera de lugar.

Un recorrido, dos trabajos

`take` recibe un grupo y encuentra lo que está fuera de lugar:

```
fn take(&mut self, form: &Form, group: &Group, mut tell: impl
FnMut(Misplaced)) {
    // ...
    for &played in &group.notes {
        let place = place_of(&played);
        let kind = if repeat(&played) {
            continue;
        } else if self.has(place) {
            SlipKind::EarlyRepeat
        } else if window & (1 << place) == 0 {
            SlipKind::RowOrder
        } else {
            continue;
        };
        self.errors += 1;
        tell(Misplaced {
            kind,
            played,
            place,
            wanted,
            heard: self.heard(),
        });
    }
    self.used |= fresh;
    if fresh != 0 {
        self.next = self.next.max(16 - fresh.leading_zeros() as usize);
    }
}
```

`let kind = if ... { continue; } else ...` funciona porque `continue` nunca da un valor: su tipo es `!`, «nunca», que encaja en lugar de cualquier tipo, así que las otras ramas deciden que `kind` es un `SlipKind`. Una nota que solo repite la nota recién tocada se salta de esta manera, y también una nota en su lugar. Al final, `16 - fresh.leading_zeros()` es el lugar que sigue al bit más alto de `fresh`, el lugar más lejano al que llegó este grupo.

El mismo `take` cuenta los problemas mientras se comparan las lecturas, y los informa para la lectura que gana. Hace las dos cosas con el closure `tell`, al que llama por cada nota fuera de lugar. Al comparar, el closure no hace nada: `walk.take(form, &groups[end - 1], |_| {})`. Al informar, convierte cada nota en un hallazgo: `walk.take(form, group, |misplaced| slips.push(misplaced.slip(form, key)))`. Ese segundo closure cambia `slips`, un vector que toma prestado, así que es un `FnMut`, no solo un `Fn`. Como `take` es genérica sobre el closure (`impl FnMut`), el compilador hace una copia de `take` para cada uno, y la vacía no cuesta nada.

La mejor lectura: programación dinámica

`read` tiene que elegir dónde empieza y termina cada forma, y qué forma es. `best[end]` es la menor cantidad de problemas en los primeros `end` grupos, y `came_from[end]` es el último tramo en el camino hasta ahí:

```
for start in 0..n {
    if best[start] == usize::MAX {
        continue;
    }
    // ...
    for from in 0..12 {
        for (index, form) in forms.iter().enumerate() {
            // ... may this form start at its place `from` here? ...
            let mut walk = Walk::from(from);
            for end in start + 1..=reach {
                walk.take(form, &groups[end - 1], |_| {});
                // ...
                let missing = if end == n { 0 } else { 12 - walk.heard()
};
                let cost = best[start] + walk.errors + missing;
                if cost < best[end] {
                    best[end] = cost;
                    came_from[end] = Stretch {
                        start,
                        end,
                        form: index,
                        from,
                    };
                }
            }
        }
    }
}
```

Cada tramo se prueba una vez por forma. El recorrido sigue de un `end` al siguiente en vez de volver a empezar, así que una forma probada sobre veinte grupos cuesta veinte pasos, no doscientos. Una forma puede empezar en un lugar posterior (`from` mayor que 0) solo al principio de todo, traída del patrón anterior, o donde las notas anteriores son las que compartió el grupo anterior (una elisión). Las formas desde su primera nota se prueban primero, y solo un costo estrictamente menor reemplaza una lectura: así se llevan los empates. Recorrer `came_from` hacia atrás desde el último grupo da las formas en orden.

`forms_of` hace la lista de formas a probar, saltando una forma que tenga las mismas notas que otra ya incluida, así que el palíndromo de Webern se lee con 24 formas, no 48. Y `read_rows` corre `read` dos veces, una con todas las notas y otra por pista, y se queda con la más barata:

```
if apart.iter().map(|(_, reading)| reading.cost).sum::<usize>() <
together.cost {
```

La misma idea, guardar la mejor manera de llegar a cada punto, reparte las notas de un acorde entre las partes en la revisión clásica (Conducción de voces y movimiento de las partes).

Notas que empiezan juntas: chunk_by

Los grupos salen de las notas ordenadas por comienzo, cortadas donde cambia el comienzo:

```
for chunk in played.chunk_by(|a, b| a.note.start_tick() ==
b.note.start_tick()) {
```

`chunk_by` entrega porciones de vecinas para las que el closure dice «mismo grupo», sin copiar nada.

Cómo se prueba

`row.rs` comprueba la teoría contra los libros: las cuatro formas de Schoenberg, nota por nota, contra las publicadas; cada una de las 48 formas de la Serie de Lineup, nombrada otra vez por `label_of`; cada fila de la matriz de Berg una forma P y cada columna una forma I, con la primera nota de la serie a lo largo de toda la diagonal; la serie de Webern un palíndromo (R9 es P3); la serie de Berg, la serie Todos los intervalos y cincuenta series al azar de todos los intervalos, de todos los intervalos; la Serie de Lineup combinatoria solo con I3; series leídas desde nombres, números y la forma compacta, y las equivocadas rechazadas con el mensaje correcto; la misma semilla dando la misma serie al azar; y una serie con una nota escrita dos veces rechazada al leerla de un archivo:

```
#[test]
fn webern_is_a_palindrome() {
    let webern = RowPreset::named("Webern's Symphony").unwrap().row;
    let r9 = webern.form(RowLabel::new(RowForm::Retrograde, 9));
    assert_eq!(webern.label_of(r9).unwrap().to_string(), "P3");
    let mirror = RowPreset::named("Mirror row").unwrap().row;
    let backwards = mirror.form(RowLabel::new(RowForm::Retrograde, 0));
    assert_eq!(mirror.label_of(backwards).unwrap().to_string(), "P6");
}
```

`serial.rs` escribe una melodía en corcheas, las cuatro formas por turno llenando el patrón, tricordios como acordes, las tres series, y comprueba que ninguna nota pasa del final del patrón. `library/rows.rs` comprueba que cada estilo usa una serie de las predefinidas y escribe sus voces en cualquier tonalidad; que Pads de hexacordios tiene las doce notas en cada compás; que una canción toma un estilo de serie, y que los acordes del estilo reemplazan a los de la canción en su propio canal en vez de sonar encima. Una prueba une las dos mitades de este capítulo: escribe cada estilo, sobre Do y sobre Fa♯, y le pasa la revisión de la conducción de voces, que tiene que leer cada uno como formas de su serie sin ninguna nota fuera de orden, así que los estilos y la revisión siguen las mismas reglas.

Las pruebas de las reglas en `voice_leading/serial.rs` escriben cada regla y sus excepciones como notas sobre la serie de Schoenberg: una serie en orden, acordes de notas vecinas, dos notas intercambiadas, una nota que falta, repeticiones y trinos, una elisión, una forma sin terminar, una melodía y unos acordes que comparten una forma, un canon, duplicaciones a la octava, acordes tonales, y las reglas clásicas apagadas en series y conjuntos. Sus nombres se leen como las reglas: `two_notes_swapped_are_one_note_out_of_order`, `a_note_may_be_repeated_or_trilled_but_not_brought_back_later`, `one_form_may_end_on_the_note_the_next_begins_with`.

Los ejercicios sobre series están en Ejercicios: la teoría musical en código.

Micromodos

Archivos: `crates/lineup-core/src/harmony/micromode.rs`, `crates/lineup-core/src/micromodal.rs`, `crates/lineup-core/src/library/micromodes.rs` y `crates/lineup-app/src/micromodes.rs`

Los conjuntos de clases de altura le dan un nombre a cada grupo de notas. Este capítulo mira de cerca los grupos más pequeños que todavía tienen un carácter propio, los grupos de **tres** notas, tal como el compositor argentino **Francisco Kröpfl** enseñaba a oírlos a sus alumnos. Los llamó **micromodos**: doce pequeños «modos» de tres notas, cada uno con un color que se aprende a reconocer.

En la aplicación, **Micromodos…**, en el piano roll, abre una ventana con los doce micromodos de Kröpfl en sus tres familias. Cada uno se dibuja en un reloj, con su forma prima en solfeo (*do - do# - mi*), sus intervalos, su conjunto de clases de altura y su hora en el Tone Clock de Peter Schat; puedes escucharlo, dibujar con él o usarlo como tonalidad. Selecciona algunas notas y la ventana lista los micromodos que contienen. Las ventanas Conjuntos de clases de alturas y Micromodos se señalan una a la otra. La Biblioteca tiene un grupo **Micromodos (Kröpfl)** con siete estilos, y **Archivo > Demostraciones > Micromodos** toca cada uno como una pieza corta. La hoja de acordes también puede mostrar cualquier armonía como micromodos (ver Una armonía, cuatro nombres).

La música, en pocas palabras

Francisco Kröpfl (1931–2021) nació en Timișoara, Rumania, y creció en la Argentina. Fue un pionero de la música electrónica en América Latina: en 1958 fundó el Estudio de Fonología Musical de la Universidad de Buenos Aires, el primer estudio de música electrónica de una universidad en el continente. También enseñó composición durante décadas, y su obra para piano *Música ’66* se escribió para poner a prueba el método que enseñaba.

Ese método parte de una observación sencilla. Elige tres notas distintas cualesquiera. Como sea que las escribas, en el orden y en las octavas que sea que las toques, las tres notas tienen tres intervalos entre sí, y esos intervalos le dan al grupo un color: Do, Re♭ y Mi tienen el mismo sonido tenso, agridulce, ya los toques como acorde, como melodía o repartidos en cuatro octavas. Sube el grupo (Re, Mi♭, Fa♯) y conserva su color. Dalo vuelta (Do, Si, La♭, los mismos intervalos hacia abajo) y el color sigue siendo la misma familia de sonido.

Así que Kröpfl contaba tres notas como **el mismo micromodo** cuando son:

- las mismas notas en otro orden u otra octava, o escritas de otra manera (Do♯ y Re♭ son aquí una sola nota),
- movidas hacia arriba o hacia abajo (transportadas), o
- dadas vuelta (invertidas).

Son exactamente las reglas de los conjuntos de clases de altura. De las 220 maneras de elegir tres notas entre doce, las reglas dejan solo **doce** micromodos distintos. Repite uno de ellos a lo largo de un pasaje y el pasaje toma su color; Pablo Cetta, compositor y teórico que estudió la música de Kröpfl, lo describe como «un 'color' o timbre global particular y reconocible».

La tabla

Cada micromodo se escribe en su **forma prima**: su forma más compacta, empezando en Do, la misma forma prima que en la ventana Conjuntos de clases de alturas. Esta es la tabla de Pablo Cetta, con los números que Allen Forte les dio a los mismos doce grupos en 1973:

	Micromodo	Forma prima	Forte	Intervalos	Formas
1	**Menor 1**	do - do# - re	3-1 [012]	dos segundas menores, una segunda mayor	12
2	**Menor 2**	do - do# - re#	3-2 [013]	segunda menor, segunda mayor, tercera menor	24
3	**Menor 3**	do - do# - mi	3-3 [014]	segunda menor, tercera menor, tercera mayor	24
4	**Menor 4**	do - do# - fa	3-4 [015]	segunda menor, tercera mayor, cuarta	24
5	**Menor 5**	do - do# - fa#	3-5 [016]	segunda menor, cuarta, tritono	24

	Micromodo	Forma prima	Forte	Intervalos	Formas
6	**Mayor 1**	do - re - mi	3-6 [024]	dos segundas mayores, una tercera mayor	12
7	**Mayor 2**	do - re - fa	3-7 [025]	segunda mayor, tercera menor, cuarta	24
8	**Mayor 3**	do - re - fa#	3-8 [026]	segunda mayor, tercera mayor, tritono	24
9	**Mayor 4**	do - re - sol	3-9 [027]	segunda mayor, dos cuartas	12
10	**Acorde disminuido**	do - mib - solb	3-10 [036]	dos terceras menores, un tritono	12
11	**Acorde menor-mayor**	do - mib - sol	3-11 [037]	tercera menor, tercera mayor, cuarta	24
12	**Acorde aumentado**	do - mi - sol#	3-12 [048]	tres terceras mayores	4

«Cuarta» quiere decir aquí una cuarta justa *o* una quinta justa: en un conjunto, un intervalo y su inversión cuentan como uno (de Do a Sol hacia arriba hay una quinta, de Sol a Do una cuarta; el par Do y Sol es el mismo de las dos maneras). «Formas» cuenta cuántos grupos distintos de tres notas son ese micromodo: la forma prima y su inversión sobre cada una de las doce notas. La mayoría tiene 24; los que son su propia inversión tienen 12; el acorde aumentado solo 4, porque subirlo una tercera mayor da otra vez las mismas tres notas. Las formas suman 220, todos los grupos de tres que hay.

Tres familias, y sus nombres

Kröpfl ordenó los doce en tres familias. Pablo Jaureguiberry, que analizó la música de Diana Rud, alumna de Kröpfl, da las reglas:

- «Los micromodos **menores** son todos aquellos que están compuestos por al menos un intervalo de segunda menor». Son cinco, de Menor 1 a Menor 5.

- «Los **mayores** tienen todos al menos una segunda mayor y a su vez carecen de semitonos». Son cuatro, de Mayor 1 a Mayor 4.
- «Los restantes contienen al menos dos terceras cada uno». Son las tres tríadas, y llevan su nombre: el acorde **disminuido**, el **menor-mayor** y el **aumentado**.

Los nombres salen de la forma prima, como explica Cetta: los nueve primeros empiezan con una segunda, menor (*do - do#*) o mayor (*do - re*), y el número sube a medida que sube la tercera nota. Así, Menor 3 es «el tercer grupo que empieza con una segunda menor», *do - do# - mi*.

Un micromodo merece una segunda mirada: el **Acorde menor-mayor**. Su forma prima, Do Mi♭ Sol, es una tríada menor. Dada vuelta desde Do es Do, La y Fa: una tríada de Fa mayor. Las tríadas menores y mayores son imágenes en espejo una de la otra, así que son un solo micromodo, y su nombre dice las dos cosas. Todos los acordes de una progresión I–IV–V–I son el mismo micromodo; lo que la música tonal hace con ellos es cuestión de *qué* formas, no de qué color.

Encontrar a mano el micromodo de tres notas

Cetta da dos ejemplos de Menor 3: Mi, Fa y La♭, y Fa, Re y Sol♭.

1. **Mi Fa La♭.** Ordena las notas dentro de una octava, tan juntas como se pueda: Mi Fa La♭ abarca cuatro semitonos, el menor ámbito. Mueve el grupo para que empiece en Do: Do Do♯ Mi. Eso es *do - do# - mi*, la forma prima de Menor 3.
2. **Fa Re Sol♭.** Ordenadas y lo más juntas posible: Re Fa Sol♭, un ámbito de cuatro semitonos. Desde Do: Do Mi♭ Mi, que es *do - mib - mi*, y no está en la tabla. Dalo vuelta: los intervalos 3 y luego 1 pasan a ser 1 y luego 3, Do Do♯ Mi. Otra vez Menor 3.

Ese es todo el algoritmo: la **forma normal** (el orden más compacto), movida a Do, comparada con su inversión, y gana la más compacta de las dos. La ventana Conjuntos de clases de alturas ya lo hace, así que Ferrous Lineup encuentra el micromodo de tres notas pidiendo su forma prima y buscándola en la tabla.

Los mismos doce, tres veces

Kröpfl no fue el único que encontró estos doce. En *The Structure of Atonal Music* (1973), el teórico estadounidense **Allen Forte** hizo la lista de todas las clases de conjuntos, y sus doce tricordios, de 3-1 a 3-12, son los micromodos en el mismo orden. El compositor neerlandés **Peter Schat** los volvió a encontrar y los ordenó como las doce

horas de su **Tone Clock** («reloj tonal», *The Tone Clock*, 1993): la hora I es Menor 1, la hora III es Menor 3, la hora XII el acorde aumentado.

Pero los tres no querían decir lo mismo con ellos. Marcela Pavia, que compara a Forte con Kröpfl, describe la clasificación de Forte como matemática, y la de Kröpfl como construida sobre cómo *percibimos* los grupos de notas, a partir de la psicología de la Gestalt; y encuentra que las dos dan resultados muy distintos cuando se usan para analizar música. Forte cuenta cada subconjunto de cada acorde; Kröpfl escucha los grupos que la música hace oír como una unidad. Ferrous Lineup muestra las dos cosas: el número de Forte junto a cada micromodo, y los micromodos junto a cada conjunto de Forte.

Los conjuntos más grandes están hechos de micromodos

Un conjunto de *n* notas tiene $n(n-1)(n-2)/6$ grupos de tres: 4 para un acorde de cuatro notas, 10 para una escala pentatónica, 35 para una escala de siete notas. Contar qué micromodos son da el **contenido de micromodos** del conjunto, una huella de sus colores. Algunos ejemplos, todos comprobados por las pruebas o calculados con las mismas funciones:

- **La séptima de dominante**, Do Mi Sol Si♭: Do Mi Sol es una tríada mayor (Acorde menor-mayor), Mi Sol Si♭ una tríada disminuida, Do Mi Si♭ es Mayor 3 (el acorde sin su quinta) y Do Sol Si♭ es Mayor 2. Uno de cada uno.
- **La escala pentatónica**, Do Re Mi Sol La (5-35): Mayor 2 cuatro veces, Mayor 4 tres veces, Acorde menor-mayor dos veces (Do Mi Sol y La Do Mi) y Mayor 1 una vez (Do Re Mi). Ningún semitono, así que ningún micromodo menor, y por eso la escala pentatónica suena tan suave.
- **La escala diatónica**, las siete notas de Do mayor (7-35): Mayor 2 ×8, Acorde menor-mayor ×6 (las tres tríadas mayores y las tres menores), Mayor 4 ×5, Menor 2 ×4, Menor 4 ×4, Mayor 1 ×3, Menor 5 ×2, Mayor 3 ×2 y una tríada disminuida (Si Re Fa). Ningún Menor 1, Menor 3 ni acorde aumentado: no se pueden tocar solo con las teclas blancas.
- **La escala de tonos enteros** (6-35): Mayor 3 ×12, Mayor 1 ×6 y el acorde aumentado dos veces. Nada más, porque todos sus intervalos son pares.
- **Do Do♯ Re♯ Mi** (4-3), del análisis de Jaureguiberry de *Tres amores* (1986) de Diana Rud: dos Menor 2 y dos Menor 3. Rud arma sus acordes de cuatro notas de modo que cada compás conserva tres notas del anterior y añade una nueva, y los micromodos que contienen cambian de a poco.

- **Do Do♯ Fa♯ Sol** (4-9), del análisis de Cetta de *Música '66*: cuatro grupos de tres, todos Menor 5. Kröpfl los pone junto a Menor 3 en la obra.

Así se enlazan las dos ventanas. Un micromodo *es* un conjunto de clases de altura, uno de los doce de tres notas, así que la ventana Conjuntos de clases de alturas marca esos doce con el nombre de su micromodo; y un conjunto más grande *está hecho* de micromodos, así que esa ventana los lista bajo sus detalles («Micromodos que contiene, en sus 35 grupos de tres»). En la ventana Micromodos, las notas que seleccionas se cuentan de la misma manera.

Una serie hecha de un solo micromodo: el Concierto de Webern

Compositores anteriores a Kröpfl ya habían oído lo mismo. La *Pieza para piano* op. 11 n.º 1 (1909) de Arnold Schoenberg empieza con Si, Sol♯ y Sol: Menor 3. Anton Webern fue más lejos. La serie de su *Concierto para nueve instrumentos* op. 24 (1934) está hecha de cuatro grupos de tres (ver Series dodecafónicas y sus reglas sobre las series):

```
Si Si♭ Re | Mi♭ Sol Fa♯ | La♭ Mi Fa | Do Do♯ La
```

Cada grupo es Menor 3, y cada uno es una forma del primero:

grupo	notas	intervalos	forma del primer grupo
1	Si Si♭ Re	baja 1, sube 4	él mismo
2	Mi♭ Sol Fa♯	sube 4, baja 1	al revés y dado vuelta (RI)
3	La♭ Mi Fa	baja 4, sube 1	al revés (R)
4	Do Do♯ La	sube 1, baja 4	dado vuelta (I)

Al comienzo del Concierto, los cuatro grupos los tocan el oboe, la flauta, la trompeta y el clarinete, uno tras otro, así que se oye el mismo color cuatro veces en cuatro instrumentos. Schat armaba armonías de la misma manera, con formas de una sola hora que juntas usan las doce notas; en sus términos, la serie de Webern está hecha de la hora III.

Webern no pensaba en micromodos: la palabra llegó décadas después. El Concierto es una obra dodecafónica, y Webern construyó su serie con cuatro formas de un solo grupo; «Menor 3» y «3-3» son nombres posteriores para ese grupo, como lo describiría un teórico. Cetta señala que Kröpfl, como los compositores del círculo de Schoenberg, prefería los grupos de tres notas, y por eso los dos se encuentran aquí.

La serie es una de las series famosas de la ventana Serie dodecafónica, **Concierto de Webern**, y el estilo **Los tricordios de Webern** la toca: la serie como melodía que salta por tres octavas, con cada grupo de tres como acorde debajo. Está en el grupo **Series dodecafónicas** de la Biblioteca y al final de **Archivo > Demostraciones > Micromodos**. El ritmo es de Ferrous Lineup, no de Webern.

Solo se copia en Ferrous Lineup música de dominio público. Webern murió en 1945, así que su música es de dominio público donde los derechos de autor duran 70 años después de la muerte del autor, y solo se usan las doce notas de la serie, nunca su partitura. Kröpfl murió en 2021 y sus alumnos viven, así que su música todavía tiene derechos de autor, y nada de ella está en la aplicación: los siete estilos de micromodos se escribieron para Ferrous Lineup.

Música con un solo micromodo

¿Cómo se escribe música con un solo micromodo? Alejandro Martínez describe el micromodo como «elemento mínimo de construcción melódico/armónica». Los acordes son formas del micromodo, y también los grupos de la melodía, así que todo el pasaje tiene un solo color. La pregunta es qué forma viene después. Ferrous Lineup ofrece dos respuestas, en `micromodal.rs`.

Mover todas las notas la misma distancia

Lo más simple es una **secuencia**: subir las tres notas el mismo intervalo cada vez. *Bruma de tonos enteros* sube Mayor 1 un tono en cada compás: Do Re Mi, Re Mi Fa♯, Mi Fa♯ Sol♯… Nunca sale de una escala de tonos enteros, el sonido de *Voiles* de Debussy. *Deriva aumentada* baja el acorde aumentado un semitono por vez; después de cuatro acordes, el quinto tiene las notas del primero, porque solo hay cuatro acordes aumentados.

Conservar dos notas, mover una

La manera más suave es conservar **dos** notas de cada grupo y mover solo la tercera, para que cada grupo crezca a partir del anterior. Para las tríadas, esto da un conjunto famoso de movimientos. Desde Do mayor (Do Mi Sol):

movimiento	conserva	mueve	da
P (paralelo)	Do y Sol	Mi baja a Mi♭	Do menor

movimiento	conserva	mueve	da
L (intercambio de sensible)	Mi y Sol	Do baja a Si	Mi menor
R (relativo)	Do y Mi	Sol sube a La	La menor

Son los tres movimientos de la **teoría neorriemanniana**, que los teóricos usan para explicar los cambios de acorde cromáticos de Wagner y Liszt. El estilo *Tríadas, dos notas fijas* camina así: Do menor, Do mayor, Mi menor, Mi mayor, Do♯ menor, Re♭ mayor, Si♭ menor, Si♭ mayor…

El mismo recorrido sirve para cada micromodo, con una sorpresa. Tres de ellos no pueden salir de un pequeño rincón de las doce notas:

- **Mayor 1** (Do Re Mi) y **Mayor 3** (Do Re Fa♯) solo tienen intervalos pares: 2, 4 y 6 semitonos. Una forma que conserva dos de sus notas tiene que añadir una tercera a una distancia par de ellas, así que se queda en la misma **escala de tonos enteros**. El recorrido llega a 6 notas, nunca a 12.
- El **acorde disminuido** solo tiene intervalos de 3 y 6 semitonos, así que se queda en un mismo **acorde de séptima disminuida**: 4 notas.
- El **acorde aumentado** solo tiene intervalos de 4 semitonos. Dos de sus notas fijan la tercera, así que ninguna otra forma comparte dos notas con él. Su recorrido lo sube un semitono en lugar de eso.

Todos los demás micromodos tienen algún intervalo impar, y su recorrido da la vuelta por las doce notas. Las pruebas comprueban cada uno de estos números.

Qué forma conservar

Varias formas comparten dos notas con la actual (para Menor 3 sobre Do Do♯ Mi hay tres: La Do Do♯, Do Re♯ Mi y Do♯ Mi Fa), así que el recorrido necesita una regla para elegir una. En orden:

1. la forma cuya **nota nueva** lleva más tiempo sin oírse, para que la música recorra todas las notas a las que puede llegar,
2. después, la forma misma usada hace más tiempo, para que un recorrido encerrado en pocas notas (Mayor 1, el acorde disminuido) igual pase por todas sus formas en vez de ir y venir entre dos,
3. después, el menor movimiento de las notas que cambian,
4. y por último los bits del conjunto, para que la elección sea siempre la misma.

Texturas

Un recorrido es una lista de formas. Una **textura** dice cómo las toca un instrumento: como **acordes** (las notas compartidas se quedan donde están y la nota que cambia se mueve al lugar más cercano: conducción de voces suave), como acordes golpeados una y otra vez (**pulso**), como una **línea** (primero la nota nueva, después las otras), como un **arpegio** (subiendo y bajando), o como un **bajo** (la fundamental de una tríada, si no, la primera nota de la forma normal). Cada instrumento de un estilo toca el mismo recorrido, cada uno en su textura, así que el bajo, los acordes y la melodía comparten siempre sus tres notas.

Una línea toca primero la nota que acaba de llegar, para que se oiga qué cambió, y después las dos que se quedaron. `write_forms` parte las notas de la forma en dos con `partition`, que manda cada elemento a una de dos colecciones según un closure de sí o no, en una sola pasada:

```
let (new, kept): (Vec<PitchClass>, Vec<PitchClass>) =
    normal.into_iter().partition(|p| !old.contains(*p));
new.into_iter().chain(kept).collect()
```

`old` es la forma anterior; las notas que no están en ella son nuevas. Después `chain` vuelve a unir las dos listas, las notas nuevas primero.

Los siete estilos:

estilo	micromodo	recorrido	sonido
Color Menor 3	Menor 3	conservar dos notas	acordes, melodía y bajo en un solo color
Puntos vieneses	Menor 5	conservar dos notas	notas sueltas que saltan por tres octavas sobre acordes suaves
Pulso de clusters	Menor 1	conservar dos notas	clusters que suben de a un semitono, sobre un ritmo techno
Bruma de tonos enteros	Mayor 1	sube un tono	arpegios en 3/4
Groove de cuartas	Mayor 4	conservar dos notas	acordes sus sobre un ritmo neo soul
Tríadas, dos notas fijas	menor-mayor	conservar dos notas	movimientos P, L y R
Deriva aumentada	aumentado	baja un semitono	cuatro acordes aumentados, dando

estilo	micromodo	recorrido	sonido
			vueltas

Una demostración toca una pasada desde el micromodo sobre Do, después dado vuelta, después desde un tritono de distancia (Fa♯), después dado vuelta desde allí, y otra vez la primera pasada. Haz clic en un estilo de la Biblioteca mientras suena la canción y toda la canción pasa a él: sus acordes y su bajo se reemplazan por el recorrido, los patrones empiezan por turno desde esos cuatro lugares, y la tonalidad pasa a ser el propio micromodo, un conjunto de clases de altura, así que las teclas de armonía tocan sus formas.

Los micromodos en Rust: una tabla en un `static`

Un micromodo son datos: un nombre, una familia, una forma prima, una línea de texto. Los doce son un array `static` de un struct simple:

```
/// One of the twelve micromodes.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct Micromode {
    /// From 1 to 12: its place in the table, which is also its Forte
    /// number (3-`number`) and its hour on Peter Schat's Tone Clock.
    pub number: u8,
    pub family: MicromodeFamily,
    /// Kröpfl's name, in Spanish, as his students use it: "Menor 3".
    pub name: &'static str,
    /// The name in English.
    pub english: &'static str,
    /// The prime form, from C.
    pub prime: PitchSet,
    /// The prime form in solfège, as in Cetta's table: "do - do# - mi".
    pub solfege: &'static str,
    /// Where you hear it: a chord or scale it belongs to, or music that
    /// is built from it.
    pub about: &'static str,
}
```

```
static MICROMODES: [Micromode; 12] = [
    Micromode {
        number: 1,
        family: MicromodeFamily::Minor,
        name: "Menor 1",
        english: "Minor 1",
        prime: prime(0b0000_0000_0111),
        solfege: "do - do# - re",
        about: "Three notes a semitone apart: the smallest cluster, a
chromatic scale in \
                miniature.",
    },
    // ...
];
```

`prime(0b0000_0000_0111)` es el conjunto {Do, Do♯, Re} escrito como los bits de un `u16`, un bit por nota, con Do como el bit más bajo (ver Un conjunto de clases de altura en doce bits). `prime` es una `const fn`, así que toda la tabla se construye al compilar y no cuesta nada al arrancar el programa. Cada campo es `Copy` (números, enums, `&'static str` y `PitchSet`, que es un `u16`), así que `Micromode` deriva `Copy` también, pero el resto del código pasa `&'static Micromode`: una referencia a la tabla, que nunca puede quedar colgando porque la tabla vive tanto como el programa. Los nombres de Kröpfl están en castellano en los dos idiomas de la aplicación; en inglés la ventana añade el nombre inglés entre paréntesis, «Menor 3 (minor 3)».

Encontrar el micromodo de tres notas reutiliza la teoría de conjuntos:

```
pub fn of(set: PitchSet) -> Option<&'static Micromode> {
    if set.len() != 3 {
        return None;
    }
    let prime = set.prime_form();
    MICROMODES.iter().find(|m| m.prime == prime)
}
```

Comprobar una tabla con pruebas

Una tabla escrita a mano puede tener errores. Las pruebas la comprueban desde tres direcciones, ninguna de las cuales confía en la tabla:

- **Contra Forte.** Las clases de conjuntos, filtradas a tres notas, tienen que ser las doce formas primas en el mismo orden, llamadas de 3-1 a 3-12.
- **Contra la cuenta.** Un bucle sobre los 4.096 conjuntos (cada `u16` de 0 a 4.095) comprueba que exactamente los conjuntos de tres notas tienen un micromodo,

que hay 220, y que las formas de cada micromodo coinciden con las veces que se lo encontró.
- **Contra las reglas.** Las familias se vuelven a deducir de los intervalos, con las reglas de Jaureguiberry, con una función que solo ven las pruebas:

```
#[cfg(test)]
fn of_intervals(vector: [u8; 6]) -> MicromodeFamily {
    if vector[0] > 0 {
        MicromodeFamily::Minor
    } else if vector[1] > 0 {
        MicromodeFamily::Major
    } else {
        MicromodeFamily::Chord
    }
}
```

`#[cfg(test)]` sobre una sola función la compila solo para `cargo test`. Sin él, `cargo clippy` avisa que la función nunca se usa, porque solo la llaman las pruebas. El solfeo también se comprueba: cada sílaba se vuelve a convertir en un número (`"do#"` es 1, `"mib"` es 3) y las notas tienen que ser la forma prima.

Los intervalos en palabras

`vector[0]` cuenta segundas menores y `vector[1]` segundas mayores: es el vector interválico de los Conjuntos de clases de altura. El mismo vector da los intervalos en palabras, para la ventana:

```
pub fn intervals(&self) -> String {
    const NAMES: [(&str, &str); 6] = [
        ("minor second", "minor seconds"),
        ("major second", "major seconds"),
        ("minor third", "minor thirds"),
        ("major third", "major thirds"),
        ("perfect fourth or fifth", "perfect fourths or fifths"),
        ("tritone", "tritones"),
    ];
    let vector = self.prime.interval_vector();
    let mut parts = Vec::new();
    for (class, &count) in vector.iter().enumerate() {
        let (one, many) = NAMES[class];
        // Whole phrases, as a Spanish number goes with its noun.
        match count {
            0 => {}
            1 => parts.push(crate::i18n::tr_string(format!("a {one}"))),
            2 => parts.push(crate::i18n::tr_string(format!("two
{many}"))),
            _ => parts.push(crate::i18n::tr_string(format!("three
{many}"))),
        }
    }
    crate::i18n::list(&parts)
}
```

Los plurales son un par de textos en vez de `format!("{name}s")`, porque en inglés el plural no siempre es añadir una «s»: «two perfect fourths or fifths», no «two perfect fourth or fifths». Y cada frase se traduce entera («a minor second», «two minor seconds»), porque en castellano el número cambia con el sustantivo: *una segunda menor*, *dos segundas menores* (ver Traducir: la clave es el texto en inglés). Después `list` une las frases:

```
pub fn list(items: &[impl AsRef<str>]) -> String {
    match items.split_last() {
        None => String::new(),
        Some((last, [])) => last.as_ref().to_string(),
        Some((last, rest)) => {
            let rest: Vec<&str> =
rest.iter().map(AsRef::as_ref).collect();
            trf("{} and {}", &[&rest.join(", "), &last.as_ref()])
        }
    }
}
```

`split_last` da el último elemento y la porción de antes, o `None` para una lista vacía. `Some((last, []))` es un **patrón de slice**: `[]` solo coincide con una porción vacía, así que esta rama es una lista de uno. Así, `["una segunda menor", "una tercera menor", "una tercera mayor"]` se vuelve «una segunda menor, una tercera menor y

una tercera mayor», con la «y» (o la inglesa «and») en su lugar y sin aritmética de índices que pueda fallar. `items: &[impl AsRef<str>]` acepta igual una porción de `String` que de `&str`.

Contar grupos de tres

`trichords` lista cada grupo de tres notas de un conjunto, con tres bucles anidados cuyos índices siempre suben, así que ningún grupo se lista dos veces:

```
pub fn trichords(set: PitchSet) -> Vec<PitchSet> {
    let notes: Vec<PitchClass> = set.iter().collect();
    let mut trios = Vec::new();
    for a in 0..notes.len() {
        for b in a + 1..notes.len() {
            for c in b + 1..notes.len() {
                trios.push([notes[a], notes[b],
notes[c]].into_iter().collect());
            }
        }
    }
    trios
}
```

`[notes[a], notes[b], notes[c]].into_iter().collect()` convierte un array de tres clases de altura en un `PitchSet`: `PitchSet` implementa `FromIterator<PitchClass>`, así que `collect` sabe construir uno, y el tipo de retorno `Vec<PitchSet>` le dice qué colección construir.

`micromode_content` los cuenta en un `[u16; 12]`, un contador por micromodo. Para mostrarlos, `micromodes_in` se queda con los encontrados y los ordena, los más frecuentes primero:

```
pub fn micromodes_in(set: PitchSet) -> Vec<(&'static Micromode, u16)> {
    let counts = micromode_content(set);
    let mut found: Vec<(&'static Micromode, u16)> = MICROMODES
        .iter()
        .zip(counts)
        .filter(|(_, count)| *count > 0)
        .collect();
    found.sort_by_key(|(m, count)| (std::cmp::Reverse(*count), m.number));
    found
}
```

`sort_by_key` ordena desde la clave más pequeña, y aquí la cuenta *más grande* va primero. `std::cmp::Reverse` envuelve un valor y lo compara al revés, así que `Reverse(8)` va antes que `Reverse(6)`. La clave es una **tupla**, y las tuplas se

comparan elemento por elemento: primero las cuentas (al revés), y solo cuando dos cuentas son iguales, el número del micromodo. Así, las cuentas iguales conservan el orden de la tabla, y la lista sale igual cada vez que se dibuja.

El recorrido: elegir con una tupla

La regla del recorrido, «la nota nueva más fresca, después la forma usada hace más tiempo, después el menor movimiento», es un solo `min_by_key` con una tupla de cuatro:

```
members
    .iter()
    .copied()
    .filter(|&set| set != current && shared(set) == most)
    .min_by_key(|&set| {
        let new = PitchSet::from_bits(set.bits() & !current.bits());
        // The new note heard most recently decides; never heard is best.
        let freshest = new
            .iter()
            .map(|p| heard[usize::from(p.semitone())].map_or(0, |i| i +
1))
            .max()
            .unwrap_or(0);
        // How far the moving notes go, each to its nearest new note.
        // ...
        // Then the form itself heard least recently, so a walk shut in
        // a few notes still goes round all its forms.
        let used = before.iter().rposition(|&f| f == set).map_or(0, |i| i
+ 1);
        (freshest, used, moves, set.bits())
    })
    .unwrap_or(current)
```

Algunas cosas para notar:

- `members` son todas las formas del micromodo (24, 12 o 4). `shared(set)` cuenta las notas que una forma tiene en común con la actual, con los bits: `set.bits() & current.bits()` se queda con las notas que están en las dos. `set.bits() & !current.bits()` se queda con las notas de `set` que *no* están en `current`: la nota nueva.
- `heard` es un array de doce `Option<usize>`, uno por clase de altura: `None` si la nota todavía no sonó, o `Some(i)` si sonó por última vez en la forma número `i`. `map_or(0, |i| i + 1)` convierte «nunca» en 0, la clave más pequeña, así que una nota nunca oída siempre gana, y una nota oída hace mucho le gana a una oída recién.

- `rposition` busca en una porción **desde el final** y da el índice del último elemento que coincide: cuándo se usó la forma por última vez. Otra vez, «nunca» se vuelve 0.
- Como la tupla se compara elemento por elemento, el orden de sus cuatro elementos *es* el orden de las reglas. Cambiar las reglas es reordenar una tupla, no reescribir una cadena de `if`.

¿Cuántas formas necesita una pasada? Un estilo dice cuántos pasos dura cada forma, y el patrón tiene una duración en pasos; la cuenta redondea hacia arriba, así que una última forma que solo cabe a medias igual se escribe:

```
let count = length.div_ceil(self.steps.max(1)).max(1);
```

`div_ceil` es la división que redondea hacia arriba: `16.div_ceil(6)` es 3, donde `16 / 6` es 2.

Los estilos se adueñan de la armonía

Un estilo de micromodo, como un estilo dodecafónico, no tiene acordes que seguir. Cuando haces clic en él mientras suena la canción, *reemplaza* la armonía de la canción: calla los acordes y el bajo que no toca, y escribe sus propias voces en los canales correspondientes. Los estilos de series hacen lo mismo, así que los dos tipos de estilo llaman a una sola función, de Series dodecafónicas y sus reglas:

```
pub(super) fn take_over_harmony(
    project: &mut Project,
    uses: &dyn Fn(&str) -> bool,
    rhythm: Option<&Rhythm>,
    mut write: impl FnMut(&mut Project, usize, usize, Option<usize>),
) -> RhythmChange {
```

Los dos closures se pasan de dos maneras distintas, a propósito:

- `uses` solo responde una pregunta («¿el estilo toca este canal?»), así que es un `&dyn Fn`: un **trait object**, una referencia a algún closure cuyo tipo se decide al ejecutar. Una sola copia compilada de `take_over_harmony` sirve para todos los closures.
- `write` se llama una vez por patrón con `&mut Project`, y hace la escritura. Es un `impl FnMut`: el compilador hace una copia de `take_over_harmony` para cada tipo de closure que se le pasa (una para los estilos de series, otra para los de

micromodos), lo que es un poco más rápido y deja que el closure cambie lo que capturó. `FnMut` es la cota más laxa que todavía se puede llamar muchas veces.

Los estilos de micromodos la llaman así:

```
pub fn use_micromode_style(project: &mut Project, style: &MicromodeStyle)
-> RhythmChange {
    let key = project.scale.root;
    let uses = |name: &str| style.voices.iter().any(|voice| voice.channel
== name);
    let change = take_over_harmony(
        project,
        &uses,
        style.rhythm(),
        |project, pattern, n, keys| {
            let start = DEMO_SECTIONS[n % DEMO_SECTIONS.len()].0;
            write_voices(project, pattern, style, key, start, keys);
        },
    );
    if change.harmony > 0 {
        project.scale = style.scale(key);
    }
    change
}
```

`n` cuenta los patrones escritos hasta ahora, así que el primer patrón de la canción empieza desde el micromodo, el siguiente dado vuelta, el siguiente a un tritono, y así: la canción no se repite. El closure recibe `project` como argumento en vez de capturarlo: `take_over_harmony` ya tiene `&mut Project`, y Rust solo permite un préstamo mutable a la vez, así que el proyecto se le presta al closure en cada llamada.

Las demostraciones se arman como las dodecafónicas: `section_patterns` hace un patrón por sección y lo llena con un closure, y `arrange_sections` los pone en la lista de reproducción, con la batería entrando después de la primera sección. `micromode_demo` y `row_demo` usan las dos, cada una pasando su propio closure.

La ventana: un reloj dibujado con el painter de egui

Las doce notas se sientan en un círculo como las horas de un reloj, Do arriba, y las tres notas del micromodo se unen en un triángulo. La forma *es* el micromodo: mover el grupo hace girar el triángulo por el reloj, y darlo vuelta lo refleja como en un espejo. Por eso Schat llamó horas a los tricordios.

```
/// The point for pitch class `n` on a clock face: C at the top, going
/// round clockwise a semitone at a time.
fn on_clock(center: Pos2, radius: f32, n: u8) -> Pos2 {
    let angle = f32::from(n) / 12.0 * TAU;
    center + Vec2::new(angle.sin(), -angle.cos()) * radius
}
```

`TAU` es 2π, una vuelta entera, así que la clase de altura `n` está a `n` doceavos de vuelta. En matemáticas, el ángulo 0 apunta a la derecha y los ángulos giran en sentido antihorario; aquí el 0 está arriba, y se gira en sentido horario. Usar `sin` para x y `-cos` para y logra las dos cosas: en 0, x es 0 e y es −1, que en una pantalla es *arriba*, porque la y de la pantalla crece hacia abajo.

`clock` pide entonces un cuadrado de la ventana con `ui.allocate_exact_size`, toma un `painter_at` en ese rectángulo (lo que se dibuje fuera se recorta), y dibuja: el círculo con `circle_stroke`, el triángulo con `Shape::convex_polygon` (relleno con el color de acento a un cuarto de fuerza, `gamma_multiply(0.25)`), un punto para cada nota, más grande para las tres del grupo, y los nombres alrededor. Los botones de la lista, cada uno un reloj diminuto y un nombre, se dibujan igual: `allocate_exact_size` con `Sense::click()` da una `Response` que sabe de pasar el puntero y de clics, como un botón de verdad.

Dos ventanas que se señalan una a la otra

Ninguna de las dos ventanas llama a la otra. Cada una devuelve una lista de **acciones**, como las demás ventanas del piano roll:

```
/// Something done in the micromode window.
pub enum MicroAction {
    /// Make this group the key.
    UseAsKey(Scale),
    /// Play these notes now.
    Hear(Vec<Note>),
    /// Show this set in the set browser.
    ShowSet(PitchSet),
}
```

La ventana de conjuntos tiene dos acciones propias para esto, `ShowMicromode` y `OpenMicromodes`. El piano roll es dueño de las dos ventanas, así que maneja las acciones: `ShowSet` llama al `pick` de la ventana de conjuntos y la abre; `ShowMicromode` llama al `pick` de la ventana Micromodos. Cada ventana solo sabe mostrar un conjunto que le dan, y el piano roll las conecta. (**Dibujar con él** no

necesita ninguna acción: las dos ventanas reciben el `ChordStamp` del piano roll como `&mut`, y lo cambian directamente.)

`pick` muestra no solo el micromodo sino la forma exacta en la que se hizo clic. Haz clic en Menor 3 bajo la serie de Webern en la ventana de conjuntos, y la ventana Micromodos se abre en Si Si♭ Re, no en Do Do♯ Mi:

```
pub fn pick(&mut self, form: PitchSet) {
    if let Some(micromode) = Micromode::of(form) {
        self.micromode = micromode;
        let (n, inverted) = micromode.prime.moved_to(form).unwrap_or((0,
false));
        self.transposition = n;
        self.inverted = inverted;
        self.raise = true;
    }
}
```

`moved_to`, de los Conjuntos de clases de altura, encuentra cómo mover la forma prima sobre la forma: subir `n` semitonos, dada vuelta o no.

Una ventana que se muestra desde otra tiene que pasar al frente, o puede quedar *detrás* de la ventana en la que hiciste clic. A una ventana que se abre la mantiene delante `Front` (ver Una ventana que se abre delante); a una que ya estaba abierta, pero detrás, la sube la bandera `raise`: `pick` la pone, y el fotograma siguiente la lee y la borra de una vez con `std::mem::take`, que devuelve el valor y deja `false` en su lugar:

```
if std::mem::take(&mut self.raise) {
    ctx.move_to_top(egui::LayerId::new(egui::Order::Middle, window_id()));
}
```

La ventana de conjuntos hace lo mismo con una bandera `reveal`, y además desplaza su larga lista hasta la clase que se le pidió mostrar: `response.scroll_to_me(Some(egui::Align::Center))` sobre la línea de esa clase, durante un solo fotograma.

Escribir las tríadas como tríadas

Do, Mi♭ y Fa♯ y Do, Mi♭ y Sol♭ son las mismas tres teclas, pero solo una es una tríada disminuida en el papel: Do Mi♭ Sol♭, dos terceras menores apiladas. Los tres micromodos de acordes se escriben como los acordes que son. El trait `NoteNames`, que escribe notas con la escritura de la tonalidad o con sostenidos o bemoles sin más, tiene un segundo método para esto:

```
pub trait NoteNames: Copy {
    fn note_name(self, pitch_class: PitchClass) -> NoteName;
    /// The notes of a chord, from its root up.
    fn chord_names(self, chord: Chord) -> Vec<NoteName>;
}
```

`Micromode::chord` encuentra la tríada de una forma (la fundamental es la nota de la forma normal desde la cual las otras dos forman un acorde conocido), y la ventana escribe sus notas con `chord_names`. El acorde aumentado no tiene una fundamental de verdad (cada una de sus notas está a una tercera mayor de la siguiente), así que se nombra desde la primera nota de su forma normal.

Fuentes

Kröpfl nunca publicó su teoría; como escribe Jaureguiberry, «esta propuesta nunca fue publicada por su autor». Se conoce por sus alumnos y por análisis de su música. Estas son las fuentes de este capítulo y de los textos de la aplicación:

- Pablo Cetta, «*La tercera es la vencida* de Francisco Kröpfl y Oscar Steimberg», *Cuadernos de Análisis y Debate sobre Músicas Latinoamericanas Contemporáneas* 1 (2018), pp. 8–36. La tabla de los doce micromodos (Ejemplo 1) con los números de Forte, las formas primas en solfeo, los ejemplos Mi Fa La♭ y Fa Re Sol♭, el análisis de *Música ’66* (Menor 3 y Menor 5, el conjunto 4-9) y de la canción de la bruja en la ópera de cámara de Kröpfl (2011), construida sobre todo con Menor 3. En línea, en el repositorio de la Universidad Católica Argentina.
- Pablo Jaureguiberry, «*Tres amores* de Diana Rud: otras particularidades de los micromodos de Francisco Kröpfl» (Universidad Nacional de Rosario, 2016), pp. 499–510. La definición, las tres familias y sus reglas, Menor 3 como uno de los preferidos de los alumnos de Kröpfl, y conjuntos de cuatro notas hechos de dos Menor 2 y dos Menor 3.
- Alejandro Martínez, «Algunos aspectos de los micromodos de Francisco Kröpfl y su contextualización histórica», en *Actas de la XXI Conferencia de la Asociación Argentina de Musicología* (2015), pp. 399–415. El micromodo como la unidad mínima de la melodía y la armonía, su uso por compositores argentinos de varias generaciones, y su lugar junto a los conjuntos de Forte, el análisis tricordal de Gilbert y el Tone Clock de Schat.
- Marcela Pavia, «La estructura de la música atonal: dos enfoques», *CodeXXI* 1 (1998), pp. 199–214. La clasificación de Kröpfl, construida sobre la percepción, comparada con la de Forte, construida sobre la matemática.
- Allen Forte, *The Structure of Atonal Music* (Yale University Press, 1973). Los tricordios de 3-1 a 3-12.

- Peter Schat, *The Tone Clock* (Harwood Academic Publishers, 1993). Las doce horas.

Los ejemplos musicales de otros compositores solo se nombran, nunca se copian, salvo las doce notas de la serie de Webern.

Cómo se prueba

`micromode.rs` comprueba la tabla y las fuentes: los doce coinciden con los tricordios de Forte en orden; los 4.096 conjuntos dan 220 grupos de tres, cada uno en exactamente un micromodo, con 12, 24 o 4 formas; las familias se deducen de los intervalos con las reglas de Jaureguiberry; el solfeo escribe cada forma prima; los nombres se encuentran en castellano y en inglés, en mayúsculas o minúsculas; Mi Fa La♭ y Fa Re Sol♭ de Cetta, Si Sol♯ Sol de Schoenberg y los cuatro grupos de la serie de Webern son Menor 3; Do Do♯ Re♯ Mi tiene dos Menor 2 y dos Menor 3; 4-9 tiene cuatro Menor 5; la séptima de dominante tiene uno de cada uno de Mayor 2, Mayor 3, el acorde disminuido y menor-mayor; y la escala de tonos enteros tiene solo Mayor 1, Mayor 3 y el acorde aumentado.

`micromodal.rs` comprueba el recorrido y la escritura: cada paso conserva dos notas; el recorrido llega a las doce notas, o a 6 para Mayor 1 y Mayor 3, o a 4 para el acorde disminuido, sin repetir una forma en sus cuatro primeros pasos; los movimientos de las tríadas son P, L o R; una secuencia de Mayor 1 se queda en una escala de tonos enteros; los acordes conservan sus notas compartidas en las mismas teclas; cada textura escribe notas de las formas correctas y nunca después del final del patrón; el bajo toca fundamentales.

```
#[test]
fn the_bass_plays_the_root_of_a_triad() {
    assert_eq!(bass_note(set(&[9, 0, 4])), PitchClass::A);
    assert_eq!(bass_note(set(&[11, 2, 5])), PitchClass::B);
    assert_eq!(bass_note(set(&[0, 1, 4])), PitchClass::C);
    assert_eq!(bass_note(set(&[11, 0, 3])), PitchClass::B);
}
```

La Do Mi es La menor, así que su bajo es La; Si Re Fa es Si disminuido; Do Do♯ Mi no es una tríada, así que su bajo es la primera nota de su forma normal, Do; y Si Do Mi♭ tampoco es una tríada, y empieza en Si.

`library/micromodes.rs` comprueba que cada estilo toca solo su micromodo, en cualquier tonalidad y desde el comienzo de cada sección, con la tonalidad convertida en el micromodo; que la primera forma empieza en la tónica; y que una canción toma

un estilo de micromodo. Las pruebas de la aplicación comprueban que los micromodos de acordes se escriben como acordes (Do Mi♭ Sol♭), que ▶ **Escucharlo** toca las tres notas desde el Do central, y que elegir una forma muestra cómo se movió.

Los ejercicios sobre micromodos están en Ejercicios: la teoría musical en código.

Conducción de voces y movimiento de las partes

Archivos: `crates/lineup-core/src/harmony/voice_leading.rs`, `crates/lineup-core/src/harmony/voice_motion.rs` y `crates/lineup-app/src/piano_roll.rs`

La línea de acordes nombra cada acorde que escribes. Esa es la mitad de la armonía. La otra mitad es la **conducción de voces**: cómo va cada parte desde su nota en un acorde hasta su nota en el siguiente. Una parte es una línea de notas, como un cantante, un instrumento o una línea dentro de los acordes de un piano. Dos obras pueden usar los mismos acordes y sonar muy distintas, fluidas o torpes, porque sus partes se mueven de otra manera.

Marca **Conducción de voces**, junto a **Notas fantasma** en el piano roll, y hace dos cosas que hace un profesor de armonía. Marca dónde las partes rompen las reglas de la escritura clásica a varias voces, la lapicera roja; y muestra cómo se mueve cada parte en cada cambio de acorde, bien o mal, que es casi todo lo que hay que aprender. Este capítulo trata la teoría, las reglas y sus excepciones, y el código: programación dinámica para partir una pista en partes, cadenas de `Option` para encontrar quintas paralelas, y una caché para que la revisión corra solo cuando algo cambió.

La música, en pocas palabras

Los cursos de armonía clásica enseñan la conducción de voces con la escritura a cuatro partes: soprano, contralto, tenor y bajo, como en los corales de Bach. Dan unas pocas reglas que mantienen a cada parte como una línea propia. La revisión sigue las tres más conocidas: **sin quintas ni octavas paralelas**, **la sensible sube**, **la séptima baja**. Se explican una por una en Las reglas, más abajo. Otras dos ideas describen cada movimiento, bueno o malo.

Grados conjuntos, saltos pequeños y saltos

Lo lejos que se mueve una parte tiene un nombre:

Movimiento	Intervalo	Semitonos	Ejemplo
Sostenida	la misma nota	0	Sol3 a Sol3

Movimiento	Intervalo	Semitonos	Ejemplo
Grado conjunto	una segunda	1 o 2	Fa4 a Mi4
Salto pequeño	una tercera	3 o 4	Si3 a Sol3
Salto	una cuarta o más	5 o más	Sol2 a Do3

Las partes internas de un coral se mueven casi siempre por grado conjunto o sostienen sus notas. El bajo salta más, porque va de fundamental a fundamental: Sol2 subiendo una cuarta a Do3 en una cadencia es uno de los movimientos de bajo más comunes que hay.

El movimiento entre dos partes

Dos partes van juntas de una de cuatro maneras:

Movimiento	Las dos partes	Ejemplo
Paralelo	van en el mismo sentido y conservan el mismo intervalo	terceras que van a terceras
Directo	van en el mismo sentido, a otro intervalo	una sexta que va a una quinta, las dos bajando
Contrario	van en sentidos opuestos	el bajo sube, la parte superior baja
Oblicuo	una se mueve, la otra se queda	una nota sostenida debajo de una que se mueve

La escritura a varias voces prefiere el movimiento **contrario** y el **oblicuo** entre las partes extremas, la línea superior y el bajo, porque son las dos que más sigue el oído. Ir en sentidos opuestos las mantiene como dos líneas. Las terceras y sextas paralelas están bien en cualquier lugar, y suenan dulces. Las quintas y octavas paralelas son lo que marca la revisión, porque hacen que dos partes suenen como una.

«El mismo intervalo» quiere decir el mismo **número** de intervalo tal como lo escribe la tonalidad, no la misma cantidad de semitonos. De Mi4 sobre Sol3 a Fa4 sobre La3, la parte de arriba sube una segunda menor y la de abajo una segunda mayor. Mi sobre Sol es una sexta mayor y Fa sobre La una sexta menor: una sexta las dos veces, así que es movimiento paralelo, sextas paralelas. Un profesor de armonía también lo cuenta así.

Lo que ves

- **Una fila Voces** debajo de las dos filas de nombres de acordes. Cada hallazgo es un bloque encima del acorde al que llegan las partes, nombrado con unas pocas letras: **5as** y **8vas** para quintas y octavas paralelas, **unís.** para unísonos, **sens.** para una sensible que no sube, **7.ª** para una séptima que no baja.
- **Líneas magenta sobre la grilla**, desde la nota que deja una parte hasta la nota a la que va, donde se rompe una regla, y **un contorno punteado** donde debería haber ido una sensible o una séptima.
- **Una línea de color para cada parte**, de cada una de sus notas a la siguiente, encima de las notas, para que el ojo pueda seguir una parte por todo el patrón, como en una partitura. Una nota sostenida no tiene línea: la parte se queda donde está. Los movimientos del cambio al que apuntas se dibujan más gruesos; apuntar a cualquier lugar de un acorde elige el cambio que llevó a él.
- **Una línea encima de la grilla.** Cuando apuntas a un cambio sin nada que marcar, nombra cada parte de arriba abajo, con adónde va, y luego el movimiento entre las partes extremas: «Piano: superior Re5 → Do5 · 2.ª Fa4 → Mi4 · 3.ª Si3 → Sol3 · inferior Sol2 → Do3; partes extremas en movimiento contrario». En una sola pista las partes se nombran por su lugar; con varias pistas, por pista. Su tooltip dice cada movimiento en palabras («baja una segunda mayor de Re5 a Do5, por grado conjunto»), y el movimiento entre cada par de partes vecinas. Cuando apuntas a un paso con un hallazgo, la línea dice en su lugar qué pasa allí, en magenta, y su tooltip dice por qué lo evita la escritura clásica a varias voces. Sin apuntar a nada, resume la revisión: «Conducción de voces, 4 partes: quintas paralelas (1), …».

El tooltip escribe cada intervalo según la tonalidad: de Si3 a Sol3 hay una tercera mayor, no «cuatro semitonos», y de Fa♯ a Sol una segunda menor. Una nota tocada otra vez aparece como sostenida en la línea, y el tooltip dice «vuelve a tocar Do5», para que la distingas de una nota que se mantiene.

Qué notas forman una parte

La revisión necesita partes, pero el piano roll tiene pistas. Una pista que toca una nota a la vez es una parte. Una pista que toca acordes, como un piano, tiene varias partes. La aplicación la parte como leerías una partitura: la nota de arriba de cada acorde va a la parte de arriba, la siguiente a la parte siguiente, y cada nota se une a la parte cuya última nota estaba más cerca. Los mensajes las llaman «Mano derecha (voz superior)» y «Mano derecha (voz inferior)», o «Piano (voz más aguda)», «Piano (voz 2 de 4)» y «Piano (voz más grave)».

Con **Notas fantasma** activado, se revisan también las otras pistas. Las pistas de batería quedan fuera.

Un silencio de una negra o más corta una línea, así que las notas a cada lado de él no se comparan entre sí. Un par de partes que toca en octavas casi todo el tiempo, como los contrabajos que doblan a los violonchelos, es una línea más fuerte, no dos partes en octavas paralelas: al menos cuatro notas juntas, y siete de cada ocho a una octava o al unísono. La revisión cuenta ese par una sola vez, y el resumen dice «una línea duplicada en octavas».

Las reglas

Sin quintas ni octavas paralelas

Toma dos partes a una quinta justa de distancia. Si las dos se mueven en el mismo sentido hacia otra quinta justa, son **quintas paralelas**. Lo mismo vale para las octavas y los unísonos. Esos intervalos se funden tan bien que, en ese momento, las dos partes suenan como una. La textura pierde una línea, así que la escritura clásica a varias voces los evita. Una quinta más una octava (una duodécima) sigue contando como quinta.

Cuando las partes van en sentidos opuestos, de una quinta a una duodécima, forman **quintas por movimiento contrario**. Cada parte se mueve distinto, pero el oído sigue oyendo dos quintas seguidas, así que la escritura estricta también las evita. Las octavas por movimiento contrario aparecen de vez en cuando entre la parte superior y el bajo en una cadencia, y el mensaje lo dice.

Do mayor a Re menor con todas las partes subiendo por grado conjunto es el caso de manual: el soprano Sol4 y el tenor Do4 están a una quinta, y van a La4 y Re4, otra vez a una quinta: «Quintas paralelas: Soprano y Tenor van de Sol4 sobre Do4 (una quinta) a La4 sobre Re4 (una quinta), las dos suben.» El tenor y el bajo, de Do4 sobre Do3 a Re4 sobre Re3, forman octavas paralelas.

Una nota sostenida o repetida no cuenta. Las dos partes tienen que moverse.

La sensible sube

La **sensible** es la nota un semitono por debajo de la tónica: Si en Do mayor, Sol♯ en La menor. Cuando un acorde de dominante (V, V7, vii°, vii°7) va a la tónica, la sensible tira

hacia arriba, a la tónica.

La regla es estricta en la parte superior y en el bajo, donde el oído sigue la línea. Una parte interna es más libre. En V7 a I, una sensible interna a menudo baja a la quinta del acorde de tónica, para que ese acorde tenga sus tres notas. Los libros la llaman sensible «frustrada», y la revisión deja tranquilas las partes internas.

La revisión conoce tres lugares donde la sensible tiene que subir:

- **V o vii° a I**, la cadencia.
- **V a vi** (V a VI en menor), la **cadencia rota**. La sensible igual sube, a la tónica, que ahora es la tercera de vi.
- **Una dominante secundaria** que va a su acorde: D7 a G en Do mayor. D7 toma prestado Fa♯ de fuera de la tonalidad, y Fa♯ es la sensible de Sol, así que sube a Sol. C7 yendo a F en Do mayor también es una dominante secundaria (V7/IV), pero su sensible, Mi, pertenece a la tonalidad, y conserva la libertad que tiene en Do.

Si los dos acordes tienen la sensible, puede quedarse donde está. Una sensible que baja por una escala (Do, Si, La) es una nota de paso, no una que tenga que subir. Y V seguido de I^6_4 no es una llegada, porque las partes siguen antes de que la sensible tenga adónde ir.

Cuando dos partes tienen la sensible, el mensaje dice también que está duplicada. La escritura clásica a varias voces nunca duplica la sensible: las dos copias no pueden subir sin formar octavas paralelas.

La séptima baja

En un acorde de séptima como G7 (Sol Si Re Fa), la séptima, Fa, es una disonancia. **Resuelve bajando por grado conjunto** en el acorde siguiente. En G7 a C, Fa baja a Mi. Si el Fa5 del soprano sube a Sol5, la revisión marca una séptima que no baja: la séptima de G7 (V7) sube a Sol5 en C (I) en lugar de bajar por grado conjunto a Mi5, y el contorno punteado marca Mi5.

La revisión lee algunos acordes como lo hacen los libros de armonía. Fa La Do Re yendo a G se llama F6 por sus notas, pero funciona como Dm7 con Fa en el bajo, ii^6_5 en Do. Su séptima es Do, que debería bajar a Si. La fundamental de un acorde de séptima disminuida sale de adónde va: Sol♯ Si Re Fa yendo a A tiene su fundamental en Sol♯, y su séptima es Fa.

Las excepciones de manual no se marcan:

- **El acorde siguiente tiene la nota.** Entonces la séptima puede quedarse donde está.
- **Un retardo.** La séptima se mantiene en el acorde siguiente y baja un poco después.
- **Notas de paso y bordaduras.** Una nota corta que llega y se va por grado conjunto forma una séptima solo de paso. Lo mismo una nota añadida fuera del pulso mientras las otras partes sostienen las suyas.
- **V^4_3 a I^6.** La séptima (Fa en G7 con Re en el bajo) puede subir a Sol mientras el bajo sube de Re a Mi una décima por debajo.
- **Un acorde de paso.** En C, luego Em7 sobre Re, luego C sobre Mi, el Re del bajo es una nota de paso entre dos formas del mismo acorde de Do.
- **Conducción cromática.** Una séptima puede subir un semitono hacia la sensible propia del acorde siguiente. En Am7 con Sol en el bajo yendo a E7 con Sol♯ en el bajo, Sol sube a Sol♯. La línea sigue subiendo, y el acorde siguiente la resuelve.
- **El bajo en otra octava.** El bajo puede tomar su nota una octava más allá: Si2 a Do2 es la sensible yendo a la tónica.

Lo que la revisión no juzga

Son las reglas de un estilo: la escritura a varias voces de Bach a Brahms, y de los ejercicios de armonía. El pop y el jazz mueven acordes enteros en paralelo a propósito, y Debussy también. Nadie espera que una séptima de jazz resuelva. Lee la revisión como un profesor del estilo clásico, no como un juez de toda la música. Donde la tonalidad es una serie dodecafónica, la revisión sigue en cambio las reglas de la escritura dodecafónica (ver Las reglas de la escritura dodecafónica), y donde es un conjunto de clases de altura, no se revisa nada:

```
pub fn of(key: Scale) -> Rules {
    match key.kind {
        ScaleKind::Row(_) => Rules::TwelveTone,
        ScaleKind::Set(_) => Rules::Neither,
        _ => Rules::Classical,
    }
}
```

Algunas reglas de los libros de armonía no se revisan: las quintas y octavas directas (ocultas) en las partes extremas, la distancia entre las partes, el cruce de partes, y qué nota de un acorde duplicar.

En el código

Repartir las notas de un acorde: programación dinámica

`voices_of` convierte las notas de una pista en partes. Recorre los acordes en el tiempo, cortando las notas donde cambia su comienzo con `chunk_by`. Para cada acorde, `share_out` decide a qué parte va cada nota. Las notas y las partes están ordenadas de la más aguda a la más grave, y una nota solo puede ir a una parte por debajo de la que tomó la nota de arriba, así que las partes nunca se cruzan. Entre todas las maneras de hacerlo, elige la de menor distancia total en semitonos.

Probar todas las maneras tardaría demasiado con un acorde grande. La **programación dinámica** lo resuelve con una tabla. `cost[i][j]` es el menor costo de ubicar las primeras `i` notas usando solo las primeras `j` partes:

```
for i in 1..=notes {
    for j in 1..=lines {
        let skip_part = cost[i][j - 1];
        let new_part = cost[i - 1][j] + NEW_PART;
        let take = cost[i - 1][j - 1] + distance(i - 1, j - 1);
        cost[i][j] = skip_part.min(new_part).min(take);
    }
}
```

Cada celda tiene tres opciones. Saltar la parte `j`, que no recibe ninguna nota en este acorde. Empezar una parte nueva para la nota `i`, con el costo grande `NEW_PART` (10.000, más que cualquier distancia), así que solo pasa cuando un acorde tiene más notas que partes. O poner la nota `i` en la parte `j`, con el costo de la distancia entre ellas. Cada celda se arma con celdas ya llenas, así que toda la tabla lleva `notas × partes` pasos. Volver hacia atrás desde la última celda recupera qué opción dio cada costo, y así adónde fue cada nota. La revisión dodecafónica usa la misma idea para encontrar qué formas de una serie se tocan.

Comparar dos partes nota contra nota

`together` lista cada momento en que alguna de las dos partes empieza una nota, con la nota en la que está cada parte en ese momento:

```
onsets
    .into_iter()
    .map(|tick| a.current(tick).zip(b.current(tick)))
    .collect()
```

`Option::zip` convierte dos `Option` en un `Option` de un par. Es `Some` solo cuando los dos son `Some`, es decir, cuando las dos partes están tocando. `consecutive` mira después cada par de momentos vecinos, y `kind_of_move` decide qué forma el movimiento:

```
fn kind_of_move(x: Hop, y: Hop) -> Option<SlipKind> {
    let moves = |(from, to): Hop| {
        let rested = to.start_tick() >= from.end_tick() + LINE_BREAK;
        (from.pitch != to.pitch && !rested).then(|| to.pitch > from.pitch)
    };
    let (up_x, up_y) = (moves(x)?, moves(y)?);
    let first = perfect(x.0.pitch.midi().abs_diff(y.0.pitch.midi()))?;
    let second = perfect(x.1.pitch.midi().abs_diff(y.1.pitch.midi()))?;
    let same_way = up_x == up_y;
    match (first, second) {
        (Perfect::Fifth, Perfect::Fifth) if same_way =>
Some(SlipKind::ParallelFifths),
        (Perfect::Fifth, Perfect::Fifth) =>
Some(SlipKind::ContraryFifths),
        (Perfect::Unison, Perfect::Unison) =>
Some(SlipKind::ParallelUnisons),
        (Perfect::Fifth, _) | (_, Perfect::Fifth) => None,
        _ if same_way => Some(SlipKind::ParallelOctaves),
        _ => Some(SlipKind::ContraryOctaves),
    }
}
```

`moves` da `Some(true)` para una parte que sube, `Some(false)` para una que baja, y `None` para una nota sostenida o un silencio de una negra o más: `bool::then` corre su closure solo cuando la condición se cumple. Cada `?` devuelve `None` de la función en cuanto algo falta, y `perfect` da `None` para un intervalo que no es unísono, quinta ni octava. Lo que queda después de los `?` es el caso interesante, dos intervalos justos seguidos, y un `match` sobre el par le pone nombre. Las ramas se prueban en orden, así que la rama de quinta a otra cosa tiene que ir antes de las dos ramas de octavas que atrapan todo lo que queda.

Nombrar una parte con un closure

Un hallazgo conoce sus partes por número. Solo la aplicación conoce los nombres de las pistas, así que `describe` recibe una función que nombra una parte:

```
pub fn describe(&self, voice_name: impl Fn(usize) -> String) -> String {
```

El piano roll le pasa un closure:

```
let name = |voice: usize| voice_name(drawing.project, check, voice);
// ...
let why: Vec<String> = here.iter().map(|slip|
slip.describe(name)).collect();
```

`name` se pasa por valor, una vez por cada hallazgo, y aun así se puede seguir usando después. Funciona porque el closure solo guarda referencias compartidas (`&Project`, `&VoiceLeading`). Un closure cuyas capturas son todas `Copy` es `Copy` también, así que cada llamada recibe su propia copia. (Clippy lo dice también: marca `describe(&name)` como un préstamo innecesario.)

Revisar solo cuando algo cambió

Revisar un patrón lleva menos de un milisegundo en una compilación release, pero el piano roll dibuja 60 fotogramas por segundo mientras suena una canción. Así que guarda la última revisión junto con aquello de lo que se hizo:

```
#[derive(Debug, Clone, PartialEq)]
struct VoiceInput {
    pattern: usize,
    /// The notes of each channel checked.
    notes: Vec<(usize, Vec<NoteEvent>)>,
    time: TimeSignature,
    /// The key on each beat.
    keys: Vec<Scale>,
}
```

Cada fotograma arma un `VoiceInput` nuevo y lo compara con el guardado. `#[derive(PartialEq)]` compara campo por campo, hasta cada nota. Comparar es mucho más barato que revisar, y cuando nada cambió se usa la revisión anterior:

```
if let Some((made_from, check)) = &self.voice_check
    && *made_from == input
{
    return Rc::clone(check);
}
```

`if let … && …` es una **let chain**, de la edición 2024. Prueba un patrón y una condición en un solo `if`.

La revisión se guarda en un `Rc<VoiceLeading>`, un puntero compartido. Dibujar necesita la revisión mientras también cambia el piano roll (`&mut self`). Tomar prestada la revisión de `self` bloquearía `self` todo el fotograma, así que el borrow checker lo rechazaría. `Rc::clone` solo suma uno a una cuenta, así que el fotograma

recibe un handle propio por el costo de incrementar un entero, sin copiar los hallazgos.

Seguir las partes en cada cambio

`voice_motion::changes` recorre los momentos en que empieza una nota nueva en alguna parte. Para cada parte toma la nota que tenía justo antes y la nota que tiene allí:

```
fn before(voice: &Voice, tick: usize) -> Option<NoteEvent> {
    let index = voice
        .notes
        .partition_point(|note| note.start_tick() < tick)
        .checked_sub(1)?;
    let note = voice.notes[index];
    (tick < note.end_tick() + LINE_BREAK).then_some(note)
}
```

Las notas de una parte están ordenadas por comienzo, así que `partition_point` encuentra con una búsqueda binaria cuántas empiezan antes de `tick`. La anterior a esa cuenta es la última nota de la parte; `checked_sub(1)?` se rinde cuando no hay ninguna. Después de un silencio de `LINE_BREAK`, una negra, la línea terminó, y la nota siguiente empieza una nueva. `bool::then_some` convierte la condición en un `Option`: `Some(note)` cuando se cumple y `None` cuando no.

Nombrar el movimiento

```
pub fn motion(upper: &Move, lower: &Move, key: Scale) -> Motion {
    let (a, b) = (upper.semitones(), lower.semitones());
    match (a.signum(), b.signum()) {
        (0, 0) => Motion::Static,
        (0, _) | (_, 0) => Motion::Oblique,
        (x, y) if x != y => Motion::Contrary,
        _ => {
            let number = |upper: Note, lower: Note| {
                let (upper, lower) = (
                    key.spell(upper.pitch_class()),
                    key.spell(lower.pitch_class()),
                );
                lower.letter.steps_up_to(upper.letter)
            };
            if number(upper.from.pitch, lower.from.pitch) ==
number(upper.to.pitch, lower.to.pitch)
            {
                Motion::Parallel
            } else {
                Motion::Similar
            }
        }
    }
}
```

`signum` es −1, 0 o 1: baja, se queda o sube. Hacer `match` sobre el par de signos resuelve primero los casos fáciles. Las dos se quedan. Una se queda (el `|` en un patrón quiere decir «o»). Signos opuestos. Solo cuando las dos van en el mismo sentido importa el intervalo. Entonces el closure `number` escribe las dos notas según la tonalidad y cuenta letras de la de abajo a la de arriba: Mi sobre Sol es `G.steps_up_to(E)`, 5 letras hacia arriba, una sexta. Contar letras, no semitonos, es lo que hace paralela una sexta mayor que va a una sexta menor (ver Los intervalos también cuentan letras).

Conservar la línea mientras la lees

La línea de encima de la grilla muestra el cambio del paso que está debajo del puntero. Pero para leer su tooltip, el puntero tiene que dejar las notas e ir a la línea, y entonces no hay ningún paso debajo. Así que el piano roll recuerda el último paso al que se apuntó, y la línea lo usa mientras el puntero está sobre la línea misma:

```
let line = Rect::from_min_size(
    ui.cursor().min,
    vec2(ui.available_width(), ui.text_style_height(&TextStyle::Body)),
);
let step = self
    .hovered_step
    .or(drawing.playhead)
    .or(self.last_step.filter(|_| ui.rect_contains_pointer(line)));
```

`ui.cursor()` es donde egui pondrá el próximo widget, así que `line` es el rectángulo que la línea está por ocupar, calculado antes de dibujarla. `Option::filter` conserva `last_step` solo cuando el closure dice que sí: cuando el puntero está dentro de ese rectángulo. Mientras suena una canción, el cursor de reproducción elige el paso cuando el puntero está en otro lugar.

Cómo se prueba

Aprendes armonía con esta revisión, así que una marca equivocada te enseñaría algo equivocado. Dos clases de pruebas la protegen:

- **Ejemplos de manual.** Las pruebas de `voice_leading.rs` escriben pequeños ejemplos a cuatro partes, unos veinte, cada parte como una línea de nombres de notas. Cada prueba escribe una regla y sus excepciones como notas y comprueba qué se encuentra: una cadencia de manual sin nada que encontrar, quintas paralelas por una duodécima, una nota sostenida, un silencio que corta una línea, una línea doblada, la sensible en una parte extrema y en una interna, la cadencia rota, una dominante secundaria, ii$^{6}_{5}$, V$^{4}_{3}$ a I^{6}, un retardo, un acorde de paso, una nota añadida fuera del pulso, la conducción cromática, el bajo en otra octava, un acorde de seis-cuatro, y una pista de acordes partida en partes.
- **Los corales de Bach.** La biblioteca music21, que usan los musicólogos, tiene su propia revisión de quintas y octavas paralelas. Las dos se corrieron sobre 350 corales de Bach, y encontraron las mismas 129 paralelas, en los mismos lugares y entre las mismas partes. Para la sensible y la séptima, cada uno de los hallazgos que quedan en esos corales es un apartamiento real de la escritura estricta, como un soprano que salta lejos de la sensible en una cadencia. Bach rompía las reglas de vez en cuando, a propósito y con buenos motivos.

```
#[test]
fn a_textbook_cadence_has_nothing_to_avoid() {
    // I IV V7 I in C major, as harmony books write it.
    let check = satb(["C5 C5 B4 C5", "G4 A4 G4 G4", "E4 F4 F4 E4", "C3 F3
G3 C3"]);
    assert_eq!(kinds(&check), vec![]);
    assert_eq!(
        check.summary(),
        "Voice leading, 4 parts: nothing classical part writing avoids."
    );
}
```

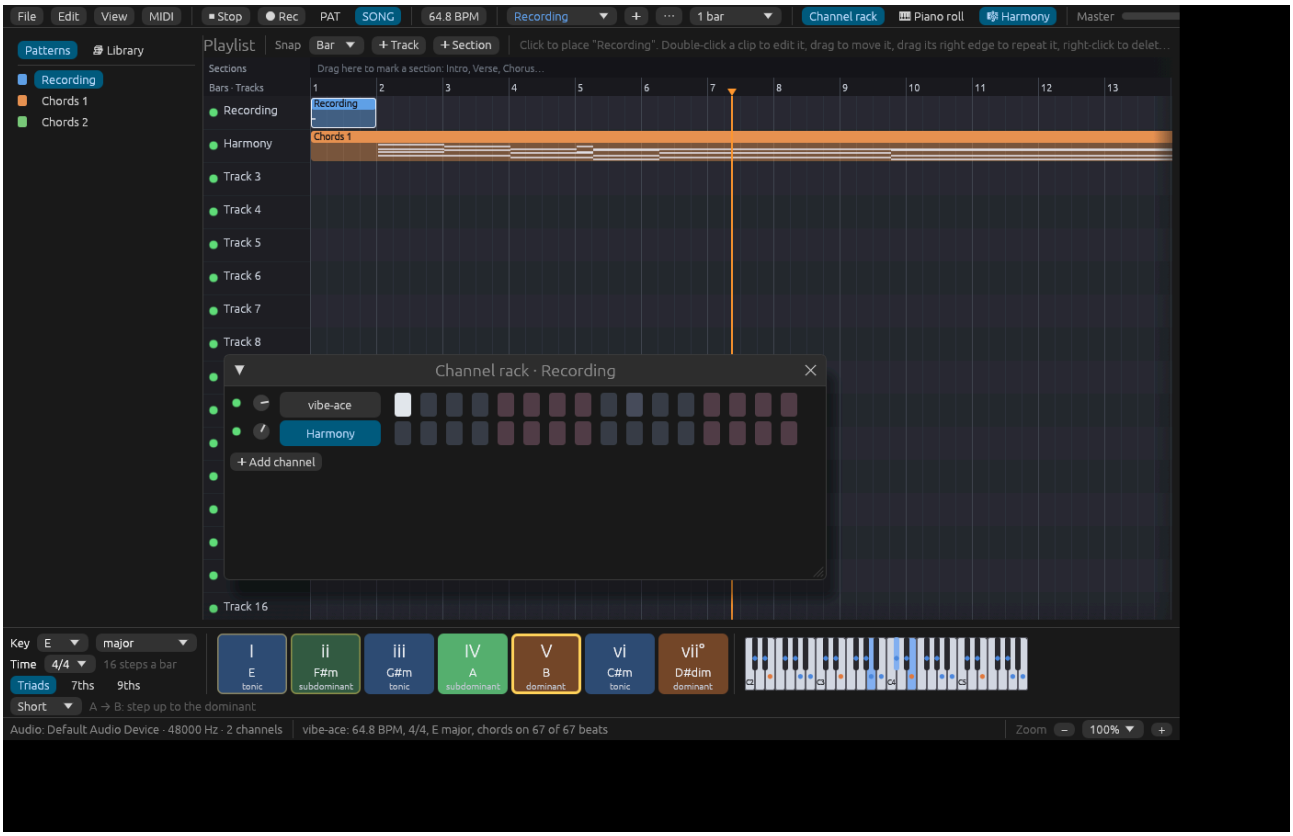
`voice_motion.rs` sigue un coral corto en una sola pista, Do mayor (Do3 Sol3 Mi4 Do5) a Fa mayor (Fa2 La3 Fa4 Do5) a G7 (Sol2 Si3 Fa4 Re5), y comprueba la línea de movimientos en el primer cambio, que con letras dice «S C5 held · A E4→F4 · T G3→A3 · B C3→F2; outer parts in oblique motion»; los tamaños sostenida, grado conjunto, grado conjunto y salto; las palabras («B: down a perfect fifth from C3 to F2, a leap»); la contralto y el tenor en movimiento paralelo, sextas paralelas, y el tenor y el bajo en movimiento contrario. En el segundo cambio, el soprano y el bajo suben los dos por grado conjunto, a una quinta y dos octavas de distancia: movimiento paralelo, y quintas paralelas. Otras pruebas comprueban una melodía sobre una nota sostenida (oblicuo), y que una quinta a una duodécima, o una sexta mayor a una sexta menor, sigue siendo paralelo.

Los ejercicios sobre la conducción de voces están en Ejercicios: la teoría musical en código.

Importar una grabación (experimental)

Archivos: `crates/lineup-listen/` y `crates/lineup-app/src/app/import.rs`

Archivo > Importar grabación…, o `ferrous-lineup song.mp3` en la línea de comandos, abre un archivo MP3, WAV, FLAC u Ogg como un proyecto nuevo, e intenta tocar junto con él. La grabación misma va a un canal sampler, y un canal **Armonía** toca el acorde oído en cada pulso. El tempo, el compás, la tonalidad y los cambios de tonalidad se fijan a partir de lo oído, así que las líneas de compás caen sobre los compases de la grabación, la hoja de acordes se llena, y las teclas de armonía se iluminan con el acorde que suena. Cómo usarlo está en Grabaciones, en el manual.



Es experimental, y el título del capítulo lo dice: el tempo, los acordes, los cambios de tonalidad y el primer tiempo fuerte son estimaciones, no una partitura leída del sonido. Un archivo de notas, MIDI, dice exactamente qué se toca; una grabación es una sola onda, con todos los instrumentos sumados, y todo hay que deducirlo de ella.

El trabajo es un crate propio, `lineup-listen`, que no sabe nada de ventanas, más un archivo en la aplicación. Este capítulo sigue al sonido desde el archivo hasta el proyecto: una transformada de Fourier para ver las notas, y **programación dinámica** tres veces, para los pulsos, los acordes y los cambios de tonalidad.

El recorrido

`analyze` es toda la escucha, como funciones simples, cada una en su propio módulo, cada una tomando los datos que hizo la anterior:

```
pub fn analyze(recording: &Recording) -> Analysis {
    let beats = beats::track(&recording.samples,
recording.sample_rate).times;
    let bpm = tempo_of(&beats);

    let whole = bpm_key::analyze(
        &recording.samples,
        1,
        recording.sample_rate,
        &bpm_key::Config::default(),
    )
    .ok()
    .and_then(|features| features.key)
    .map(|estimate| {
        let tonic = PitchClass::from_semitone(estimate.key.tonic.index());
        let kind = if estimate.key.is_minor() {
            ScaleKind::NaturalMinor
        } else {
            ScaleKind::Major
        };
        Scale::new(tonic, kind)
    });

    let frames = chroma::frames(recording);
    let beat_chromas: Vec<_> = beats
        .iter()
        .enumerate()
        .map(|(i, &start)| {
            let end = beats.get(i + 1).copied().unwrap_or(start + 60.0 /
bpm);
            chroma::average(&frames, start, end)
        })
        .collect();
    let chords = chords::chords_per_beat(&beat_chromas, whole);
    let (beats_per_bar, first_downbeat) = meter::guess(&chords);
    let keys = keys::key_spans(&beat_chromas, beats_per_bar,
first_downbeat);
    let scale = keys::main_key(&keys).or(whole);

    Analysis {
        beats,
        bpm,
        beats_per_bar,
        first_downbeat,
        scale,
        keys,
        chords,
    }
}
```

En orden:

1. **Pulsos** (`beats.rs`): cuándo cae cada pulso, en segundos, y a partir de ellos el tempo.
2. **Tonalidad** (el crate `bpm-key`): la tonalidad de toda la grabación, que ayuda a nombrar los acordes.
3. **Chroma** (`chroma.rs`): con qué fuerza suena cada una de las doce clases de altura, ventana a ventana, y luego promediado en cada pulso.
4. **Acordes** (`chords.rs`): el acorde de cada pulso.
5. **Compás** (`meter.rs`): tres o cuatro pulsos por compás, y qué pulso es el primer tiempo fuerte.
6. **Cambios de tonalidad** (`keys.rs`): la tonalidad de cada compás, encontrada por el mismo código que encuentra la tonalidad de las notas de una canción. La tonalidad del tramo más largo en una sola tonalidad, `keys::main_key` , pasa a ser la tonalidad del proyecto; `or(whole)` se queda con la tonalidad de toda la grabación cuando nada suena lo bastante como para tener una.

Luego `arrange.rs` convierte el `Analysis` en un `Project` , con los tipos que usa el resto de Ferrous Lineup: `Channel` , `Pattern` , `NoteEvent` , `Clip` . Nada de lo que viene después sabe que las notas salieron de una grabación.

La cadena después de `bpm_key::analyze` se lee tal como se ejecuta. El crate devuelve un `Result` ; `.ok()` lo convierte en un `Option` y descarta el error, ya que una grabación sin una tonalidad clara no es un fallo. `and_then` sigue solo si hay un valor, y las características pueden no tener tonalidad (`features.key` es a su vez un `Option`). `map` convierte la tonalidad del crate en una de las nuestras. Si algún paso no tiene nada, `whole` es `None` , sin ningún `if` ni ningún `match` . El último pulso no tiene un pulso siguiente donde terminar, así que `unwrap_or` le da la duración de un pulso.

Decodificar con Symphonia

Un archivo de audio es un **contenedor** (WAV, MP3, Ogg…) que guarda una o más **pistas**, cada una comprimida con un **códec**. El crate Symphonia, escrito en Rust puro, así que no hay ninguna biblioteca de C que instalar en ningún sistema, separa las dos cosas: *sondeas* el contenedor, eliges su pista de audio y le das los **paquetes** de la pista a un decodificador de su códec. Cada paquete se decodifica en un bloque de muestras.

Symphonia lee de cualquier cosa que implemente su trait `MediaSource` : un archivo, o un buffer en memoria. Lo recibe como un `Box<dyn MediaSource>` :

```
let file = File::open(path).map_err(DecodeError::Io)?;
let stream = MediaSourceStream::new(Box::new(file),
MediaSourceStreamOptions::default());
```

`dyn MediaSource` es un **trait object**: «algún tipo que implementa `MediaSource`», con el método a llamar buscado mientras el programa corre. Tipos distintos tienen tamaños distintos, así que un trait object vive siempre detrás de un puntero, aquí un `Box`. `Box::new(file)` convierte el `File` en uno sin más.

El bucle del decodificador muestra cómo los errores se *comparan con patrones*, en lugar de solo pasarse hacia arriba con `?`:

```
loop {
    let packet = match format.next_packet() {
        Ok(Some(packet)) => packet,
        Ok(None) => break,
        // A file cut short ends with an I/O error: keep what we have.
        Err(SymphoniaError::IoError(_)) => break,
        Err(err) => return Err(format_error(err)),
    };
    if packet.track_id != track_id {
        continue;
    }
    let block = match decoder.decode(&packet) {
        Ok(block) => block,
        // One damaged packet is a click, not a reason to give up.
        Err(SymphoniaError::DecodeError(_)) => continue,
        Err(err) => return Err(format_error(err)),
    };
    let channels = block.spec().channels().count().max(1);
    sample_rate = block.spec().rate();
    block.copy_to_vec_interleaved(&mut interleaved);
    samples.extend(
        interleaved
            .chunks(channels)
            .map(|frame| frame.iter().sum::<f32>() / channels as f32),
    );
    if too_long(samples.len(), sample_rate) {
        return Err(DecodeError::TooLong);
    }
}
```

Un error de entrada y salida al final quiere decir un archivo cortado, así que el bucle se queda con lo que tiene; un error de decodificación quiere decir un paquete dañado, así que lo salta; cualquier otro error termina la importación. `format_error` es un closure definido justo antes, `|err: SymphoniaError| DecodeError::Format(err.to_string())`, que convierte el error de Symphonia en el nuestro.

Las muestras vienen **entrelazadas**: izquierda, derecha, izquierda, derecha. `chunks(channels)` corta la lista en un slice por instante, y el promedio de cada slice es una muestra de una grabación mono: a los acordes no les importan la izquierda y la derecha. Una grabación más larga que `MAX_SECONDS`, diez minutos, se rechaza mientras se decodifica, ya que el análisis tiene en memoria la grabación y una copia de ella.

Los códecs de Symphonia son **features** de Cargo, activadas en el `Cargo.toml` del workspace, para que una compilación solo lleve las que usa:

```
symphonia = { version = "0.6", default-features = false, features =
["mp3", "wav", "pcm", "flac", "ogg", "vorbis"] }
```

De las muestras al chroma

Un acorde es un conjunto de clases de altura: Do mayor es Do, Mi y Sol en cualquier octava (mira Clases de altura, notas y su escritura). Para encontrar acordes en el sonido hace falta saber, momento a momento, con qué fuerza suena cada una de las doce clases de altura. Esos doce números se llaman el **chroma**, y salen de una **transformada de Fourier**, que toma un tramo de muestras y dice cuánto hay en él de cada frecuencia.

`chroma::frames` corta la grabación en **ventanas** superpuestas de 8.192 muestras, unos 0,19 s cada una a 44,1 kHz, y empieza una nueva cada 2.048 muestras (46 ms). Cada ventana se multiplica por una **ventana de Hann**, una joroba suave que la hace entrar y salir gradualmente: sin ella, el corte brusco en cada extremo le parecería a la transformada un clic, que contiene todas las frecuencias. La FFT (la transformada rápida de Fourier, del crate `realfft`) dice entonces qué tan fuerte suena cada **bin** de frecuencia en la ventana. El bin `k` es la frecuencia `k × sample_rate / 8192`, así que a 44,1 kHz los bins están a 5,4 Hz uno de otro. Las notas vecinas están más separadas que eso desde más o menos Sol2 (98 Hz) hacia arriba; la ventana es larga porque las notas graves lo necesitan.

Cada bin pertenece a una nota. La fórmula `69 + 12 × log2(f / 440)` convierte una frecuencia en un número de nota MIDI: 69 es La4, a 440 Hz, y cada semitono hacia arriba multiplica la frecuencia por la raíz duodécima de dos. El redondeo da la nota más cercana, y el resto de dividir por 12 su clase de altura:

```
let pitch_class_of = |bin: usize| {
    let frequency = bin as f32 * rate / FRAME as f32;
    let midi = 69.0 + 12.0 * (frequency / 440.0).log2();
    (midi.round() as i32).rem_euclid(12) as usize
};
```

`rem_euclid` es el resto que nunca es negativo: `-3 % 12` es `-3` en Rust, `(-3).rem_euclid(12)` es `9`. Aquí ningún bin queda por debajo de la nota MIDI 0, pero el closure no necesita saberlo.

La intensidad de cada bin se suma a su clase de altura. Tres detalles hacen que esto funcione con música real:

- La intensidad se toma como `ln(1 + 100 × magnitud)`, porque oímos la intensidad en una escala más o menos logarítmica: una nota suave una octava más arriba debe seguir contando, y la nota más fuerte no debe tapar todo lo demás.

  ```
  let level = |bin: usize| (1.0 + 100.0 * spectrum[bin].norm()).ln();
  ```

- Las notas agudas están más separadas en Hz, así que reciben más bins que las graves. Cada clase de altura se divide por su número de bins, o Si siempre parecería más fuerte que Do.

- Se guardan dos chromas por ventana. El **agudo**, de 130 a 2.100 Hz (más o menos de Do3 a Do7), es donde está el acorde. El **grave**, de 40 a 260 Hz (más o menos de Mi1 a Do4), dice sobre todo qué nota es la más grave, y la nota más grave suele ser la fundamental del acorde.

A qué clase de altura suma cada bin se calcula una sola vez, antes del bucle sobre las ventanas, como una lista de pares `(bin, clase de altura)`:

```
let bins = |(low, high): (f32, f32)| {
    let first = (low * FRAME as f32 / rate).ceil() as usize;
    let last = ((high * FRAME as f32 / rate).floor() as usize).min(FRAME /
2);
    (first.max(1)..=last)
        .map(|bin| (bin, pitch_class_of(bin)))
        .collect::<Vec<_>>()
};
let treble_bins = bins(TREBLE);
let bass_bins = bins(BASS);
```

El parámetro del closure es un **patrón**: `|(low, high): (f32, f32)|` recibe una tupla y nombra sus dos mitades, como lo haría `let (low, high) = range;`. `TREBLE` y

`BASS` son tuplas así. Con la tabla hecha, el bucle sobre miles de ventanas no hace más que sumas.

La FFT misma también se planifica una sola vez. `RealFftPlanner` elige la manera más rápida de transformar 8.192 números reales y la devuelve como un `Arc<dyn RealToComplex<f32>>`, otro trait object, que el bucle ejecuta en cada ventana con buffers creados una sola vez:

```
let mut planner = RealFftPlanner::<f32>::new();
let fft: Arc<dyn RealToComplex<f32>> = planner.plan_fft_forward(FRAME);
let mut input = fft.make_input_vec();
let mut spectrum = fft.make_output_vec();
let mut scratch = fft.make_scratch_vec();
```

`chroma::average` da luego un chroma por pulso: el promedio de las ventanas cuyo centro cae entre este pulso y el siguiente.

Del chroma a los acordes: plantillas y Viterbi

`chords::chords_per_beat` nombra el acorde de cada pulso. Hay dos ideas, la manera clásica de hacerlo sin aprendizaje automático.

Plantillas. Cada acorde que podemos reconocer es una **plantilla**: doce números, 1 para sus notas y 0 para el resto. Hay seis cualidades sobre doce fundamentales, 72 acordes, más «ningún acorde»:

```
const QUALITIES: [ChordQuality; 6] = [
    ChordQuality::Major,
    ChordQuality::Minor,
    ChordQuality::Dominant7,
    ChordQuality::Minor7,
    ChordQuality::Major7,
    ChordQuality::Diminished,
];
```

El chroma del pulso se compara con cada plantilla por **correlación**: los dos se llevan a un promedio de 0 y una longitud de 1, y su producto escalar es entonces 1 si suben y bajan juntos y 0 si no tienen relación. Es la misma comparación que hace el buscador de tonalidades con sus perfiles (mira Oír la armonía). Después, tres ajustes pequeños:

```
let Some(chord) = candidate.chord else {
    return NO_CHORD_SCORE;
};
let mut score = dot(&treble, &normalized(&candidate.template));
score += BASS_WEIGHT * bass[usize::from(chord.root.semitone())] /
bass_peak;
if scale.is_some_and(|scale|
chord.pitch_set().is_subset_of(scale.pitch_set())) {
    score += KEY_WEIGHT;
}
if chord.quality.intervals().len() > 3 {
    score -= SEVENTH_COST;
}
score
```

- Una nota grave fuerte en la fundamental del acorde suma hasta 0,15 (`BASS_WEIGHT`), y una débil resta un poco, ya que el chroma grave también está centrado en su promedio.
- Un acorde cuyas notas pertenecen todas a la tonalidad suma 0,08 (`KEY_WEIGHT`). La mayoría de los acordes de una canción cumplen eso, así que la tonalidad es un **prior** útil: lo que esperamos antes de escuchar.
- Un acorde de cuatro notas pierde 0,03 (`SEVENTH_COST`), así que gana una tríada simple salvo que la séptima esté de verdad. C7 contiene entero el acorde de Do mayor, y sin esto ganaría cada vez que un Si♭ flotara débil en el aire.

«Ningún acorde» puntúa siempre 0,25 (`NO_CHORD_SCORE`): un pulso donde ningún acorde lo supera, como el silencio, la batería sola o una voz hablada, no tiene acorde.

Suavizado. Solo, el mejor acorde de cada pulso salta de un lado a otro: una nota de paso de la melodía hace que C parezca Am durante un pulso. Los acordes reales duran. Así que cambiar de acorde cuesta 0,25 (`CHANGE_COST`), y el **algoritmo de Viterbi** encuentra la sucesión de acordes con la mejor puntuación total. Es programación dinámica: para cada pulso y cada uno de los 73 candidatos guarda la mejor puntuación de cualquier sucesión que termine ahí, hecha con los mejores del pulso anterior, y recuerda qué acorde vino antes:

```
let mut best = scores(&beats[0], &candidates, scale);
let mut from = vec![vec![0_usize; count]; beats.len()];
for (b, beat) in beats.iter().enumerate().skip(1) {
    let here = scores(beat, &candidates, scale);
    // Changing chord costs the same from anywhere, so the best way to
    // change into any chord is from the overall best one.
    let (leader, leader_score) = best
        .iter()
        .copied()
        .enumerate()
        .fold((0, f32::MIN), |a, (i, s)| if s > a.1 { (i, s) } else { a
});
    let mut next = vec![0.0; count];
    for c in 0..count {
        let (previous, total) = if best[c] >= leader_score - CHANGE_COST {
            (c, best[c])
        } else {
            (leader, leader_score - CHANGE_COST)
        };
        from[b][c] = previous;
        next[c] = total + here[c];
    }
    best = next;
}
```

A cada acorde del pulso `b` se llega o quedándose en él, o cambiando desde algún acorde y pagando la penalización. Comparar todos los pares serían 73 × 73 comparaciones por pulso. Pero cambiar cuesta lo mismo desde cualquier lado, así que el mejor cambio hacia cualquier acorde viene del mejor de todos en el pulso anterior, el `leader`: una comparación por acorde, 73 por pulso. Al final, el código toma el mejor último acorde y recorre `from` hacia atrás para leer la sucesión entera. El buscador de tonalidades usa el mismo bucle sobre compases y tonalidades (mira Dónde cambia la tonalidad).

Encontrar el líder necesita el `f32` más grande, y `f32` no tiene un `max` para un iterador: los números de coma flotante no son `Ord`, porque `NaN` no se compara con nada. `fold` lo hace a mano, llevando el mejor `(índice, puntuación)` hasta el momento. El seguidor de pulsos de más abajo hace lo mismo con `max_by` y `total_cmp`, una comparación que pone a todos los números de coma flotante, también a `NaN`, en algún orden.

Pulsos: arranques, autocorrelación y programación dinámica

`beats::track` es el seguidor de pulsos de D. P. W. Ellis, «Beat Tracking by Dynamic Programming» (2007), con los valores que la biblioteca librosa documenta como sus

valores por defecto, para que los resultados se puedan comparar con ella. Tiene tres etapas.

La fuerza de los arranques. Cuánto sonido *nuevo* empieza en cada ventana corta: 2.048 muestras, una nueva cada 512 (12 ms a 44,1 kHz). Se toma un espectrograma en la **escala mel**, 128 bandas espaciadas como las oye el oído, estrechas abajo y anchas arriba, en decibelios, y para cada ventana se suman los aumentos de cada banda desde la ventana anterior. Las disminuciones no cuentan: una nota que se apaga no es un arranque. El resultado, un número por ventana, es la **envolvente de arranques**, que tiene un pico en cada golpe de batería y en cada comienzo de nota.

El tempo. La envolvente se compara consigo misma desplazada en cada duración de pulso posible, de 30 a 320 BPM: su **autocorrelación**. Con el desplazamiento correcto, los picos caen sobre picos y la correlación es alta. Los desplazamientos cerca de 120 BPM reciben una ventaja, una campana de una octava de ancho, porque si no, los seguidores eligen la mitad o el doble del tempo con la misma facilidad que el tempo mismo:

```
let bpm = frame_rate * 60.0 / lag as f64;
let octaves = (bpm / LIKELY_BPM).log2();
let pull = (-0.5 * (octaves / LIKELY_SPREAD_OCTAVES).powi(2)).exp();
let score = correlation(lag) * pull;
```

A 60 o a 240 BPM, a una octava, una correlación cuenta 0,61 veces lo que cuenta la misma correlación a 120.

Los pulsos. De todas las maneras de colocar pulsos, el seguidor quiere aquella en que los pulsos caen sobre arranques fuertes y están separados más o menos una duración de pulso. Probar todas las colocaciones es imposible, pero la programación dinámica lo hace barato. La mejor puntuación de una cadena de pulsos que termina en la ventana `i` es la fuerza del arranque ahí, más la mejor puntuación de una cadena que termina una duración de pulso antes, más o menos, menos una penalización por ese más o menos:

```
for i in 1..envelope.len() {
    // The previous beat is between half a beat and two beats back.
    let earliest = i.saturating_sub(period * 2);
    let latest = i.saturating_sub((period / 2).max(1)).min(i - 1);
    let mut best: Option<(usize, f64)> = None;
    for (tau, &earlier) in score.iter().enumerate().take(latest +
1).skip(earliest) {
        let spacing = (i - tau) as f64 / period as f64;
        let candidate = earlier - TIGHTNESS * spacing.ln().powi(2);
        if best.is_none_or(|(_, b)| candidate > b) {
            best = Some((tau, candidate));
        }
    }
    match best {
        // A chain worth continuing: positive, so silence does not
        // chain beats that are not there.
        Some((tau, total)) if total > 0.0 => {
            score[i] = strength[i] + total;
            from[i] = Some(tau);
        }
        _ => score[i] = strength[i],
    }
}
```

`spacing` es 1 para una separación de exactamente un pulso, y su logaritmo es entonces 0: ninguna penalización. Medio pulso o dos pulsos dan `ln 2` al cuadrado, unos 0,48, por `TIGHTNESS` (100): un pulso en el lugar equivocado tiene que ser mucho más fuerte para ganar. `saturating_sub` resta pero se detiene en 0 en lugar de dar la vuelta, cosa que los números sin signo harían cerca del principio. `is_none_or` es verdadero para `None` o cuando el closure lo dice: «todavía no hay mejor, o este es mejor», en una sola llamada. Cada ventana recuerda en `from` qué pulso anterior eligió, y al final se sigue la mejor última ventana hacia atrás hasta el principio:

```
let mut at = (0..envelope.len())
    .max_by(|&a, &b| score[a].total_cmp(&score[b]))
    .unwrap_or(0);
```

La autocorrelación solo conoce ventanas enteras. A 44,1 kHz, 86 ventanas por segundo, los desplazamientos alrededor de 120 BPM son 42, 43 y 44 ventanas: 123,0, 120,2 y 117,5 BPM, nada entre medio. Así que el tempo que recibe el proyecto no es el del seguidor sino la pendiente de los pulsos mismos, la recta más cercana a «el momento del pulso n», un **ajuste por mínimos cuadrados**:

```
fn tempo_of(beats: &[f64]) -> f64 {
    let n = beats.len() as f64;
    if beats.len() < 2 {
        return 120.0;
    }
    let mean_index = (n - 1.0) / 2.0;
    let mean_time = beats.iter().sum::<f64>() / n;
    let (mut covariance, mut variance) = (0.0, 0.0);
    for (i, &time) in beats.iter().enumerate() {
        let di = i as f64 - mean_index;
        covariance += di * (time - mean_time);
        variance += di * di;
    }
    let seconds_per_beat = covariance / variance;
    if seconds_per_beat > 0.0 {
        60.0 / seconds_per_beat
    } else {
        120.0
    }
}
```

La pendiente de la mejor recta es la covarianza del índice y el tiempo dividida por la varianza del índice: segundos por pulso, promediados sobre toda la canción, con las pequeñas desviaciones de cada pulso compensándose entre sí.

El compás: dónde están los compases

Los acordes cambian sobre todo en el primer pulso de un compás. Así que `meter::guess` cuenta, para cada manera de agrupar los pulsos (de tres en tres o de cuatro en cuatro, empezando en cualquier pulso), cuántos cambios de acorde caen en el primer pulso de un grupo, y se queda con la agrupación que más supera al azar:

```rust
pub fn guess(chords: &[Option<Chord>]) -> (usize, usize) {
    let changes: Vec<usize> = (1..chords.len())
        .filter(|&i| chords[i] != chords[i - 1] && chords[i].is_some())
        .collect();
    if changes.len() < 4 {
        return (4, 0);
    }
    let mut best = (4, 0, f64::MIN);
    for beats in [4, 3] {
        for phase in 0..beats {
            let on_phase = changes.iter().filter(|&&i| i % beats ==
phase).count();
            let mut score = on_phase as f64 / changes.len() as f64 - 1.0 /
beats as f64;
            if beats == 4 {
                score += FOUR_PRIOR;
            }
            if score > best.2 {
                best = (beats, phase, score);
            }
        }
    }
    (best.0, best.1)
}
```

Si los acordes cambiaran al azar, un cuarto de los cambios caería en cada pulso de cuatro, y un tercio en cada pulso de tres. Restar ese azar, `1.0 / beats`, hace comparables los dos compases: sin eso, agrupar de tres en tres siempre parecería mejor, ya que el primer pulso de cada grupo se lleva una parte mayor. Una pequeña ventaja, `FOUR_PRIOR` (0,05), va al 4/4, ya que la mayor parte de la música popular está en cuatro. Con menos de cuatro cambios no hay de dónde sacar nada, y la respuesta es 4/4 desde el primer pulso. El resultado es `(pulsos por compás, primer tiempo fuerte)`, y el tiempo fuerte es un índice de pulso: `phase`, el primer pulso de la grabación donde empieza un compás.

Los cambios de tonalidad

`bpm-key` da una sola tonalidad para toda la grabación. Para saber dónde cambia la tonalidad, `keys.rs` le pasa el trabajo a `lineup_core::harmony::key_finding`, el código que encuentra los cambios de tonalidad en las notas de una canción (mira Dónde cambia la tonalidad). Ese código pide un chroma por compás. Una grabación no tiene notas, pero tiene el chroma de cada pulso, así que los pulsos se juntan en compases, y el chroma de cada compás hace de las notas que hay en él:

```
fn bar_chroma(beats: &[Frame]) -> key_finding::Chroma {
    let mut sum = [0.0; 12];
    for beat in beats {
        let floor = beat.treble.iter().copied().fold(f32::MAX, f32::min);
        let chroma = beat.treble.map(|v| v - floor);
        for (pc, s) in sum.iter_mut().enumerate() {
            // The note a fifth below `pc` is five semitones above it.
            *s += (chroma[pc] - FIFTH_OVERTONE * chroma[(pc + 5) %
12]).max(0.0);
        }
    }
    sum
}
```

Dos correcciones hacen que el chroma de un sonido se parezca más al de una partitura. Primero, en un espectro logarítmico toda clase de altura tiene algo de energía, por el ruido y por los armónicos de otras notas; quitar la clase de altura más débil de cada pulso, su **piso**, deja las notas que sobresalen.

Segundo, los armónicos mismos. Una nota tocada en un instrumento no es una sola frecuencia sino muchas: la nota, el doble, el triple, y así. El triple de la frecuencia está una octava y una quinta más arriba (una razón de 3 son 1.902 cents, apenas más de 19 semitonos). Así que una grabación en Do mayor está llena de Sol (de Do), Re (de Sol) y Si (de Mi) que nadie tocó, y la tonalidad sale una quinta demasiado alta, Sol por Do, más a menudo que no. Por eso cada clase de altura pierde 0,3 (`FIFTH_OVERTONE`) de la clase de altura una quinta por debajo. Una quinta por debajo son siete semitonos hacia abajo, que en un círculo de doce son cinco hacia arriba: `(pc + 5) % 12` . El comentario lo dice, porque esa línea es fácil de escribir mal.

Aquí solo cuenta el chroma agudo. Los armónicos de las notas graves llenan el chroma grave de quintas, y con él la tonalidad acertaba menos a menudo: el 80% de los pulsos frente al 84%, como deja anotado el comentario del código.

`key_spans` corta los pulsos en compases con el compás recién encontrado, y los pulsos antes del primer tiempo fuerte forman un compás propio; luego le pide a `key_finding` sus tramos, contados en compases. Cada tramo se lleva después de vuelta a pulsos:

```
let bars: Vec<_> = edges
    .windows(2)
    .map(|bar| bar_chroma(&beats[bar[0]..bar[1]]))
    .collect();
key_finding::key_spans(&bars)
    .into_iter()
    .map(|span| KeySpan {
        start: edges[span.start],
        end: edges[span.end],
        ..span
    })
    .collect()
```

`edges` guarda el pulso donde empieza cada compás, y uno más allá del último pulso. `windows(2)` lo recorre de dos en dos, solapando, `[0, 2]`, `[2, 6]`, `[6, 10]`..., que es el primer pulso de cada compás y el del compás siguiente. `KeySpan { start, end, ..span }` es la **sintaxis de actualización de structs**: un `KeySpan` nuevo con estos dos campos, y todos los demás (las tonalidades en orden, qué tan seguro) tomados de `span`.

La tonalidad de la canción, la que recibe el proyecto, es la del más largo de estos tramos:

```
pub fn main_key(spans: &[KeySpan]) -> Option<Scale> {
    spans
        .iter()
        .max_by_key(|span| span.end - span.start)
        .and_then(KeySpan::key)
}
```

`max_by_key` da el elemento con la clave más grande, aquí el de más pulsos. Acierta más a menudo que la tonalidad única de `bpm-key` (ver Qué tan bien funciona): una canción que modula, o que empieza en otra tonalidad, aleja las notas de toda la grabación de su tonalidad principal, y la tonalidad principal suele ser donde la canción pasa más tiempo.

El arreglo

`arrange.rs` construye el proyecto. La grabación pasa a ser un canal sampler con una sola nota, en el primer paso de la canción. El sampler toca su archivo desde el principio a su propia velocidad, así que el primer tiempo fuerte de la grabación tiene que caer en una línea de compás de la canción. Rara vez lo hace, así que la grabación se guarda de nuevo con silencio delante, la **entrada**: lo justo para que su primer tiempo fuerte caiga al comienzo del siguiente compás entero.

```
pub fn lead_in(analysis: &Analysis) -> LeadIn {
    let downbeat = analysis
        .beats
        .get(analysis.first_downbeat)
        .copied()
        .unwrap_or(0.0)
        .max(0.0);
    let bar = analysis.beat_seconds() * analysis.beats_per_bar as f64;
    let bars = (downbeat / bar).ceil() as usize;
    LeadIn {
        bars,
        seconds: bars as f64 * bar - downbeat,
    }
}
```

A 120 BPM en 4/4 un compás dura 2 s. Un primer tiempo fuerte a 1,3 s se mueve a 2 s, el comienzo del compás 2: un compás, y 0,7 s de silencio. `write_wav` guarda la copia, con muestras de coma flotante de 32 bits y el crate `hound`, junto al original como `song.ferrous-lineup.wav`. Los pulsos que haya antes del tiempo fuerte caen en ese primer compás, como una anacrusa en una partitura.

Los acordes pasan a ser **tramos**, un acorde sostenido a lo largo de pulsos seguidos, en pulsos de la canción. Un pulso de la grabación se desplaza según la entrada y el tiempo fuerte; uno que caería antes de que empiece la canción se deja fuera:

```
for (beat, chord) in analysis.chords.iter().enumerate() {
    let Some(song_beat) = (beat +
offset).checked_sub(analysis.first_downbeat) else {
        continue;
    };
    let Some(chord) = *chord else {
        continue;
    };
    match spans.last_mut() {
        Some(last) if last.chord == chord && last.first_beat + last.beats
== song_beat => {
            last.beats += 1;
        }
        _ => spans.push(Span {
            chord,
            first_beat: song_beat,
            beats: 1,
        }),
    }
}
```

`checked_sub` devuelve `None` donde un `-` normal sobre números sin signo se saldría por debajo de cero, y `let ... else` salta ese pulso. El `match` sobre `spans.last_mut()` alarga el último tramo cuando el mismo acorde sigue desde el

pulso inmediatamente siguiente, y empieza un tramo nuevo si no. Un pulso sin acorde termina un tramo, ya que el acorde siguiente ya no viene justo después.

`follow` hace el proyecto. Fija el tempo, el compás y la tonalidad, añade el canal sampler, con el nombre del archivo, y un canal de Lineup Osc llamado «Armonía», y pone las voces de cada tramo con el mismo `library::voicing::voice` que usa la banda, para que cada acorde se mueva lo menos posible desde el anterior (mira Cómo toca la banda una hoja de acordes). Los acordes van a patrones de 16 compases, «Acordes 1», «Acordes 2»..., colocados uno tras otro en una pista llamada «Armonía»; el patrón de la grabación está en una pista llamada «Grabación». Un tramo que cruza al patrón siguiente se corta en dos:

```
// A span that crosses into the next pattern is cut in two.
while start < end {
    let index = start / pattern_steps;
    let pattern_end = (index + 1) * pattern_steps;
    let length = end.min(pattern_end) - start;
    if notes_by_pattern.len() <= index {
        notes_by_pattern.resize(index + 1, Vec::new());
    }
    for &pitch in &voicing {
        notes_by_pattern[index].push(NoteEvent {
            velocity: CHORD_VELOCITY,
            ..NoteEvent::new(pitch, start - index * pattern_steps, length)
        });
    }
    start += length;
}
```

Los acordes se tocan con velocidad 80, más suave que la de por defecto, para quedar detrás de la grabación. Por último, `arrange::key_spans` lleva los tramos de tonalidad de pulsos a pasos de la canción de la misma manera, para que la aplicación pueda mostrarlos y marcarlos en la lista de reproducción.

La importación en la aplicación

Escuchar una canción tarda unos segundos, así que `app/import.rs` lo hace en un hilo propio, como la exportación a WAV (mira Saber cuándo ha terminado). El closure del hilo devuelve todo lo que la ventana necesita:

```
type Outcome = Result<(Project, String, Vec<KeySpan>), String>;
```

El proyecto, un resumen de lo oído para la barra de estado, y los cambios de tonalidad en pasos de la canción; o por qué falló. `type` le da un nombre corto al tipo

largo. El resumen es una frase como «canción: 98.2 BPM, 4/4, Mi menor, acordes en 310 de 312 pulsos», y se escribe en el hilo del trabajo, así que ese hilo toma primero el idioma y los nombres de las notas de la ventana, que son ajustes que viven en un hilo:

```
let language = crate::i18n::Language::current();
let naming = lineup_core::NoteNaming::current();
self.import = Some(Import {
    worker: thread::spawn(move || {
        language.set();
        naming.set();
        listen(&path)
    }),
});
```

`listen` decodifica, analiza, escribe la copia WAV (en la carpeta temporal del sistema si no se puede escribir en la carpeta del original) y construye el proyecto. En cada frame, `check_import` le pregunta al handle `is_finished()`; cuando lo está, hace `join` del hilo para obtener su resultado, cambia el proyecto con el mismo `replace_project` que usa Archivo > Abrir, le da los cambios de tonalidad a 🔍 **Buscar la tonalidad** y empieza a tocar la canción:

```
match import.worker.join() {
    Ok(Ok((project, summary, keys))) => {
        self.replace_project(project, None);
        self.key_finder.new_project(&self.project, keys);
        self.notice = Some(summary);
        self.set_mode(PlayMode::Song);
    }
```

La ventana nunca espera, así que la música sigue sonando mientras se escucha una grabación nueva. `replace_project` también llena la hoja de acordes con los acordes oídos en las notas de la pista Armonía, como hace con cualquier proyecto que llega sin acordes (mira Escribirlo).

Abrir otro proyecto mientras una importación escucha la deja caer: `replace_project` la saca de `self.import`, y la barra de estado dice **Se detuvo la importación de la grabación.** Soltar un `JoinHandle` no detiene su hilo; lo **separa** (*detach*). El hilo sigue escuchando hasta el final, y su resultado se tira, que es lo que se quiere: el proyecto nuevo se queda, y nada tiene que interrumpir el análisis a medias.

Desde el menú, la línea de comandos o soltada

Una grabación llega a la importación desde el menú, desde la línea de comandos o soltada sobre la ventana. Un archivo nombrado en la línea de comandos pasa por

`open_now`, que también abre proyectos, y que pregunta primero a `lineup_listen::is_audio_file`: una extensión de `AUDIO_EXTENSIONS`, `["mp3", "wav", "flac", "ogg"]`, en mayúsculas o minúsculas.

Un archivo soltado es más difícil de ubicar, porque un sonido soltado sobre la ventana también puede ser una muestra para el rack de canales (mira El sampler): un WAV de un golpe de batería es un sonido, un WAV de una canción entera es una grabación. `sample::is_recording` decide por lo que el sampler puede tocar y durante cuánto tiempo:

```
pub fn is_recording(path: &Path) -> bool {
    let is_wav = path
        .extension()
        .and_then(|extension| extension.to_str())
        .is_some_and(|extension| extension.eq_ignore_ascii_case("wav"));
    if !is_wav {
        return true;
    }
    hound::WavReader::open(path).is_ok_and(|reader| {
        let rate = u64::from(reader.spec().sample_rate);
        u64::from(reader.duration()) > rate *
crate::instrument::LONG_SAMPLE_SECONDS
    })
}
```

El sampler solo toca WAV, así que un archivo MP3, FLAC u Ogg solo puede ser una grabación. Un WAV lo es cuando dura más de diez segundos (`LONG_SAMPLE_SECONDS`), la misma línea a partir de la cual una voz del sampler se desvanece cuando se detiene la reproducción: pasada esa línea, un sonido se trata como un trozo de música y no como un golpe. `reader.duration()` es la cantidad de muestras por canal, leída de la cabecera del archivo sin decodificarlo. `is_project_file`, que le dice a la ventana y al rack de canales qué archivos soltados no son sonidos, cuenta una grabación entre ellos.

Por qué estos crates

Cada trabajo fue a lo que mejor lo hacía con las mismas grabaciones de prueba: demostraciones de la biblioteca renderizadas a WAV, donde se conocen el tempo, la tonalidad y el acorde de cada pulso, y unas pocas grabaciones Creative Commons con los resultados de librosa como referencia.

- **Decodificar**: Symphonia, en Rust puro, para los cuatro formatos a la vez.

- **Pulsos**: nuestro propio módulo `beats`. El crate `beat-track-rs` implementa el mismo seguidor, y el módulo da los mismos pulsos en todas las grabaciones de prueba, unas diez veces más rápido. El crate además decodifica archivos, así que metería una segunda copia de Symphonia y el crate `ndarray` en cada compilación, por 250 líneas de seguidor.
- **Tonalidad**: `bpm-key`, con los perfiles de Krumhansl y Kessler, acertaba más a menudo, y es rápido.
- **Acordes**: ningún crate lo hacía lo bastante bien, así que `chroma.rs` y `chords.rs` son nuestros, unas 300 líneas.
- **Compás**: nada de lo disponible lo adivina, de ahí `meter.rs`.

Los seguidores de pulsos neuronales, como `beat-this` y `beatnet-rs`, se dejaron de lado: necesitan modelos ONNX, así que la importación dejaría de ser Rust puro mientras corre.

Qué tan bien funciona

El README del crate tiene las tablas, medidas en 29 demostraciones de la biblioteca renderizadas a WAV:

Tempo	correcto en 20, la mitad o el doble en 8, mal en 1
Pulsos sobre la cuadrícula del tempo verdadero	27 de 29
Tonalidad de toda la canción (`bpm-key`)	24 de 29
Acordes: fundamental y mayor o menor correctos	79% de 3.887 pulsos
Compás	3/4 en las 5 demostraciones en tres, 4/4 en las otras 24
Primer tiempo fuerte sobre una línea de compás	22 de 29

La mitad o el doble del tempo es la confusión clásica de todo seguidor de tempo, y para tocar encima importa poco: los pulsos siguen cayendo en el pulso. Las canciones enteras de tríadas simples salen bien; los fallos son acordes de séptima oídos como su tríada o al revés, melodías tocadas en bloques de acordes, cuyas notas de paso parecen acordes propios, y los acordes disminuidos de los estilos más jazzísticos.

Para los cambios de tonalidad hay un informe, un test ignorado que renderiza a audio demostraciones de canciones, todas las que tienen cambios de tonalidad y una de cada cinco de las demás, y compara las tonalidades encontradas con las marcas de tonalidad que las demostraciones tienen de sus partituras:

```
cargo test --release -p lineup-listen --test key_changes -- --ignored --
nocapture
```

Medido el 30 de octubre de 2026, en 136 canciones, contando una tonalidad como correcta cuando lo son su tónica y su mayor o menor:

Tonalidades, desde una grabación	
Pulsos en la tonalidad correcta	73,7% de 36.306
La tonalidad de la canción (su tramo más largo en una tonalidad)	91 de 136
Cambios de tonalidad de las partituras encontrados a menos de dos pulsos	63 de 315
Canciones en una sola tonalidad donde igual se encuentra un cambio	3 de 52

Desde las notas de una canción, el buscador de tonalidades acierta el 87% de los compases (mira Qué tan bien escucha); desde el sonido de las mismas canciones, unos tres pulsos de cada cuatro. Los cambios rara vez se inventan, pero rara vez se encuentran justo en el pulso: el chroma de un sonido es más borroso que las notas, y una tonalidad nueva tiene que durar lo bastante para pagar su cambio. Muchos se encuentran uno o dos compases tarde, como en *Maple Leaf Rag*, donde el paso a Re♭ mayor a los 120 s se oye a los 123 s. El informe lista cada canción con sus tonalidades y dónde cambian, las verdaderas y las encontradas: el lugar donde mirar después. En las mismas canciones, la única tonalidad que `bpm-key` oye en toda la grabación acertó en 82, y por eso el proyecto toma en su lugar la tonalidad del tramo más largo.

Asperezas

- **Una copia de la grabación.** La copia WAV con su entrada existe solo porque una nota del sampler toca su sonido desde la primera muestra. El sampler puede empezar un sonido largo por la mitad (mira Empezar por la mitad), así que un desplazamiento guardado con el canal podría ocupar el lugar de la copia.
- **Detener.** Un sampler toca su sonido entero pase lo que pase, lo que está bien para un golpe de batería y mal para una canción, así que las voces de un sonido

de más de diez segundos se apagan cuando se detiene la reproducción (mira El sampler). Diez segundos es una regla práctica; un ajuste por canal sería más limpio.
- **Compases.** Solo se adivinan 3/4 y 4/4; la música en 6/8 sale como 3/4 al doble del tempo.
- **La mezcla entera.** Los acordes salen de todo lo que suena. Una melodía de notas sueltas no va acompañada de la armonía que sugiere, y una melodía fuerte puede arrastrar el acorde hacia sus propias notas.

Cómo se prueba

Cada módulo tiene tests unitarios sobre datos hechos en el test. El seguidor de pulsos recibe veinte segundos de clics a 120 BPM y tiene que encontrar el tempo con un error de menos de 3 BPM, y cada pulso a menos de 0,06 s de un clic; el silencio no tiene que dar ninguna cadena de pulsos; y los filtros mel tienen que cubrir el espectro, solapándose a la mitad. `arrange.rs` construye un `Analysis` a mano, un pulso cada medio segundo desde 0,3 s, el tiempo fuerte en el tercero, y comprueba la entrada, los tramos y el proyecto:

```
#[test]
fn chords_join_into_spans_in_song_beats() {
    let analysis = analysis();
    let spans = spans(&analysis, lead_in(&analysis));
    let firsts: Vec<_> = spans.iter().map(|s| (s.first_beat,
s.beats)).collect();
    // The two beats before the downbeat land in the lead-in bar.
    assert_eq!(firsts, [(2, 2), (4, 4), (8, 4), (13, 1)]);
}
```

Los acordes son G G C C C C G G G G, ningún acorde, C. Los dos pulsos de G antes del tiempo fuerte caen en los dos últimos pulsos del compás de entrada (los pulsos se cuentan desde 0); el C del final es un tramo propio, después del pulso sin acorde. `decode.rs` comprueba que una grabación una muestra más larga que diez minutos se rechaza, con el mensaje que ve el usuario; y la aplicación comprueba que abrir otro proyecto suelta una importación que todavía estaba escuchando.

Los ejercicios sobre la importación están en Ejercicios: la teoría musical en código.

Ejercicios: la teoría musical en código

Estos ejercicios practican esta parte sobre el código real: las notas y los acordes, los números romanos, las herramientas de edición, la escucha de acordes y tonalidades, los conjuntos, las series y los micromodos, la conducción de voces y la importación de grabaciones. Cada uno dice qué **ya existe**, para que lo amplíes en lugar de volver a construirlo. Algunos son cosas que tocar en la aplicación y escuchar; otros son piezas de Rust con un final claro; los que se llaman «Piénsalo» son preguntas, sin código que escribir. Muchos son teoría musical antes que nada: saca la respuesta en papel, y después deja que el código la compruebe.

Haz los de Rust en una rama propia. Ejecuta `cargo test` después de cada cambio, y `cargo fmt --all --check` y `cargo clippy --workspace --all-targets -- -D warnings` antes de dar uno por terminado: la CI ejecuta los tres.

Notas, escalas y acordes

1. Un acorde que todavía falta (lineup-core, fácil)

`ChordQuality::ALL` tiene treinta cualidades, desde las tríadas hasta `Dominant13` y `Major13` (mira Acordes: construirlos, nombrarlos e invertirlos). Lee `ChordQuality::intervals`, `symbol`, `name` y `numeral_suffix`, y luego añade una que no esté, como la novena menor-mayor, `MinorMajor9` (0, 3, 7, 11, 14), también a `ChordQuality::ALL`, cuya longitud, que es parte de su tipo, sube a 31. El compilador lista cada `match` que hay que actualizar. `SPELLINGS`, en `harmony/symbol.rs`, necesita una línea para la nueva cualidad, y `every_quality_is_written_and_read_back` falla hasta que su símbolo se lea de vuelta como la nueva cualidad. `no_two_qualities_have_the_same_notes` te avisa si su conjunto de alturas choca con uno que ya existe.

Como clases de altura, ¿por qué los acordes grandes se parecen tan a menudo a otros acordes sobre otra fundamental? Escribe las notas de C13 y encuentra un acorde de séptima escondido en ellas.

Practica: `match` exhaustivo, arrays cuya longitud es parte del tipo, dejar que los tests cuiden tus datos.

2. Cadd2 y el power chord (lineup-core, medio)

Los cifrados pop a veces escriben **Cadd2**: las mismas notas que Cadd9, con el Re tocado abajo, junto al Do. Añade «add2» a las escrituras de `Add9` en `SPELLINGS`, y un test que lo lea. ¿Qué se pierde al leerlo como `Add9`?

C5 es un **power chord**: solo Do y Sol, sin tercera, el favorito de la guitarra. No es una `ChordQuality`. Antes de añadirlo, haz la lista de lo que más tendría que conocerlo: `ChordQuality::family`, que todo acorde debe responder como mayor, menor o disminuido; `is_minor`; las teclas de armonía; y `find_chord`, que entonces nombraría cada quinta sola. ¿Un acorde sin tercera es mayor o menor? Decídelo, y escribe la decisión en un test.

Practica: seguir un tipo por todo el programa antes de cambiarlo.

3. El cliché de línea (lineup-core, pequeño)

El **cliché de línea** mantiene un acorde mientras una sola voz baja por semitonos: `Am | Am/G# | Am/G | Am/F#`. Léelo con `library::chart::parse_chart_in`, tócalo con `chart::accompany` y `Bass::Held`, y comprueba las notas del bajo en un test. ¿Está Sol♯ en La menor? ¿En qué menor? ¿Y qué es Am/F♯, como acorde propio, con los nombres de Acordes?

Practica: leer texto de cifrados, acordes con barra, un test sobre música que puedes cantar.

4. Una disposición drop 2 (lineup-core, medio)

`tools::inversion(pitches, times)` y `tools::invert(notes, up)` invierten acordes (mira Herramientas de edición como funciones simples). Los pianistas y arreglistas de jazz usan también otra disposición, el **drop 2**: toma un acorde en posición cerrada y baja una octava su segunda nota más aguda. Do Mi Sol Si, desde Do4 hacia arriba, queda Sol3 Do4 Mi4 Si4.

Añade `tools::drop_two(notes)` junto a `invert`, con un test como `inverting_chords_up_and_down`, y devuelve `None` cuando una nota caería por debajo de `Note::MIN`. Después ponlo en el menú **Herramientas** del piano roll. ¿El carril de acordes sigue nombrando el acorde igual, o el bajo nuevo le cambia el nombre? ¿Por qué?

Practica: manipular un `Vec` (`remove`, `insert`, `sort`), tests que comprueban tu propio resultado.

5. Ajustar a la escala, con dirección (lineup-core y lineup-app, medio)

`tools::nearest_in_key` y `tools::snap_to_scale` llevan las notas que están fuera de la tonalidad a la nota más cercana de ella, para **Ajustar a la escala** del piano roll. `nearest_in_key` prueba un semitono abajo, luego arriba, luego dos, así que en un empate gana la nota de abajo, y en Do mayor Do♯ va a Do.

Añade `nearest_in_key_towards(note, scale, up)`, que desempata en la dirección en que se mueve la melodía: un Do♯ que va subiendo hacia Re debería volverse Re. Escribe tests para notas exactamente a mitad de camino entre dos notas de la tonalidad, en los dos sentidos. `snapping_moves_only_notes_outside_the_key` fija la regla vieja; ¿cambiaría sus respuestas la función nueva?

Practica: buscar con iteradores (`find_map`), dejar la lógica fuera de la ventana.

6. Los números romanos en menor, de dos maneras (la aplicación y lineup-core, fácil)

Edición > Ajustes… > Armonía > Números romanos en tonalidades menores escribe los números de una tonalidad menor de dos maneras (mira Tonalidades menores: bIII o III).

1. En La menor armónica, ¿qué escribe cada manera para Do aumentado (Do Mi Sol♯)? ¿Y para Mi menor? Sácalo, y compruébalo con las teclas de armonía.
2. El dórico también es un modo menor, pero `Scale::is_minor_key` es falso para él, así que sus números siempre cuentan desde la escala mayor. Re dórico es Re Mi Fa Sol La Si Do. ¿Cuáles de sus números cambiarían si contaran desde su propia escala, y por qué eso podría confundir más que ayudar?
3. `MinorNumerals::with` guarda el ajuste viejo en un guardián, `let _restore = Restore(MinorNumerals::current());`, cuyo `drop` lo vuelve a poner. Renombra `_restore` a `restore` y ejecuta `cargo clippy`. Luego cámbialo por un `_` solo y ejecuta `cargo test -p lineup-core`. ¿Qué pasa cada vez, y por qué `_` se comporta tan distinto de `_restore`?

Practica: leer números romanos, guardianes que limpian en `Drop`.

7. Piénsalo: nombrar acordes

- `find_chord` prueba primero la nota del bajo como fundamental. ¿Puedes encontrar un conjunto de notas en que eso dé un nombre sorprendente? ¿Hay una regla mejor?
- Cuando nada encaja, `find_chord` añade la quinta que falta y vuelve a probar, pero solo acepta acordes de cuatro notas o más. ¿Por qué estaría mal aceptar así una tríada? Piensa en Do y Mi solos.

Los números romanos y lo que hacen los acordes

8. Después viene la napolitana (lineup-core, medio)

Después de un acorde, las teclas de armonía remarcan los acordes que suelen seguirlo, sacados de las tablas de `harmony/progression.rs` (mira Qué viene después: el conocimiento como datos). La hoja de acordes ya nombra el **acorde napolitano**, ♭II, un acorde mayor sobre el segundo grado rebajado (Re♭ en Do), pero `next_chords` no tiene un lugar para él, así que después de un acorde de Re♭ no se ilumina nada.

La napolitana suele ir a V, o a I^6_4 y luego a V. Añade un `Slot` con fundamental 1 y `ChordFamily::Major`, con un movimiento a V y su razón, a `MAJOR` y a `MINOR`. Escribe primero los tests: después de Re♭ en Do mayor, Sol; después de Re♭ en Do menor, Sol también. ¿Es Re♭ una nota de la escala de Do menor, o la napolitana está fuera de esa tonalidad también? Una `Suggestion` solo se hace para un acorde cuya fundamental es una nota de la escala: encuentra el campo de `Suggestion` que lo necesita, y explica por qué la napolitana misma nunca puede ser sugerida.

Practica: ampliar una tabla de datos, tests como especificación.

9. Las funciones de oído (la aplicación, fácil)

Escribe esto en la hoja de acordes y escucha.

1. `C Fm C`, y luego `C F C`. La única nota que cambia es La a La♭. ¿Qué le hace al ánimo? Busca una canción que conozcas que haga esto.
2. En Sol mayor, ¿qué acorde es V7/V? ¿Cuál de sus notas no está en Sol mayor, y hacia dónde quiere ir esa nota? Escríbelo y mira la función que le da la hoja.
3. Escribe una modulación de Do mayor a Fa mayor a través de un acorde pivote. ¿Qué acordes comparten las dos tonalidades? ¿Qué acorde muestra que Fa

mayor ha llegado? (¿Qué nota tiene Fa mayor que Do mayor no tiene?)

Practica: oír acordes prestados, dominantes secundarias y pivotes.

10. La escritura decide (lineup-core, pequeño)

1. `is_neapolitan` comprueba cómo se escribe el acorde con `key.tonic_name().letter.steps_up_to(symbol.root.letter) == 1`. Explica por qué Do♯ mayor en Do no pasa esa prueba mientras que Re♭ mayor sí, aunque los dos se tocan en las mismas teclas del piano. Encuentra el test que lo dice.
2. La hoja de acordes no nombra los acordes de **sexta aumentada** (italiana, francesa y alemana), que también llevan a la dominante. Busca sus notas en Do menor. ¿Cuál de ellos suena exactamente como una séptima de dominante sobre La♭, y qué en su escritura distingue a los dos? ¿Qué tendría que comprobar un `Role::AugmentedSixth`?

Practica: por qué la música escribe las notas, con letras además de semitonos.

11. Una fila de acordes prestados (lineup-app, medio)

Las canciones en tonalidades mayores suelen tomar prestados acordes de la tonalidad menor sobre la misma tónica: iv (Fa menor en Do), ♭VI (La♭), ♭VII (Si♭) y ♭III (Mi♭). Es el **intercambio modal**, detrás de muchos momentos agridulces del pop, y la hoja de acordes ya nombra esos acordes como prestados. Las teclas de armonía, en cambio, solo tocan los acordes propios de la tonalidad.

Añade una segunda fila de teclas, más pequeña, debajo de las teclas de armonía, con estos cuatro acordes, construidos a partir de números romanos con `library::numeral::parse` y `RomanChord::in_key`. Dales su propio color, y una ayuda emergente que diga de qué acorde de la tonalidad menor viene cada uno.

Practica: reutilizar los números romanos de la biblioteca, la distribución en egui.

12. Piénsalo: tablas o reglas

`next_chords` guarda lo que sabe en tablas de fundamental, familia y movimientos. Casi toda la tabla se podría calcular a partir de unas pocas reglas: fundamentales que bajan una quinta, pasos hacia arriba hasta la dominante. ¿Qué entradas se perderían las reglas? ¿Es mejor una tabla que dice claramente lo que significa que unas reglas más cortas?

Las herramientas de edición

13. Humanizar las velocidades (lineup-core y lineup-app, fácil)

Las notas tocadas todas exactamente igual de fuerte suenan mecánicas. Las demostraciones ya varían sus velocidades (`library/interpret.rs`), pero el piano roll no tiene ninguna herramienta para tus propias notas. Añade `tools::humanize(notes, amount, seed) -> Vec<NoteEvent>`, que sube o baja cada velocidad como mucho `amount`, dentro de 1 a 127, y un elemento «Humanizar velocidad» en el menú **Herramientas** del piano roll.

Ferrous Lineup no tiene un crate de números aleatorios, y no lo necesita: `ToneRow::random` y el solista usan pequeños generadores propios, y `library/interpret.rs` usa el mezclador «SplitMix64», unas pocas líneas de `wrapping_mul` y desplazamientos. ¿Por qué la función recibe una `seed` en lugar de inventarse una? ¿Cómo la probarías, y qué debería dar la misma semilla dos veces seguidas? ¿De dónde sacaría el piano roll una semilla distinta cada vez?

Practica: funciones puras con una semilla explícita, `clamp`.

14. Retrogradar una frase (lineup-core y lineup-app, medio)

Tocar una frase al revés es una manera clásica de hacer una variación. Las series tienen su retrógrado (mira Series dodecafónicas y sus reglas), pero el piano roll no tiene una herramienta para una frase. Añade `tools::retrograde(notes: &[NoteEvent], span: usize) -> Vec<NoteEvent>`: dentro de `span` pasos, cada nota conserva su altura y su duración, y su lugar se refleja.

Primero decide qué significa «reflejar» para una nota de duración 2 en un tramo de 16: ¿se refleja su comienzo, su final, o los dos? ¿Qué elección mantiene la frase dentro del tramo y suena más como la original tocada al revés? ¿Qué pasa con `nudge` y `trim`? Añade un elemento «Retrogradar» al menú **Herramientas**, con un test sobre una frase de cuatro notas que puedas cantar.

Practica: decidir una regla musical antes de programarla, los campos de `NoteEvent`.

15. Herramientas de duración (lineup-core y lineup-app, medio)

Duración, en el piano roll, fija cuánto duran las notas *nuevas*, pero ninguna herramienta cambia las notas ya escritas. Añade dos:

- `tools::set_note_length(notes, steps)` hace que cada nota dure `steps`, conservando su comienzo y su altura;
- `tools::legato(notes, pattern_length)` hace que cada nota dure hasta que empiece la siguiente, para que una línea quede ligada.

`NoteEvent` guarda su duración en pasos enteros, más `trim`, cuántos ticks antes del final de su último paso se detiene. Una nota alargada tiene que borrar un `trim` que ya no significa nada, y `tidy_notes` mantiene las notas de un patrón en orden y dentro de él. Comprueba qué pasa cuando dos notas de un canal se solaparían. ¿Por qué `set_note_length` no es simplemente `note.length = steps`?

Practica: duraciones de notas y ticks, respetar las reglas de un patrón.

16. Una herramienta de rasgueo, o un rasgueo por canal (los tres crates, más difícil)

Un guitarrista toca un acorde una cuerda tras otra. La guitarra de la banda ya lo hace: `strum`, en `band/guitar.rs`, empieza cada cuerda unos ticks después de la anterior. Tus propias notas todavía no pueden. Piensa dos diseños antes de escribir código.

- **En las notas.** Una herramienta Rasguear da a cada nota de un acorde un `nudge` creciente, desde la nota más grave hacia arriba, con unos ticks de separación. Nada más cambia en `NoteEvent`: el archivo ya guarda `nudge`, y Cuantizar devuelve las notas a la cuadrícula. Un nudge es como mucho 23 ticks, un paso menos uno.
- **En el instrumento.** Cada canal recibe un ajuste de «rasgueo» en milisegundos. Cuando varias notas de un canal empiezan en el mismo frame, el hilo de audio las empieza con esa separación, la más grave primero. Nada cambia en el patrón, pero `Channel`, el comando `InsertChannel`, el `project_commands` de la aplicación y el formato del archivo necesitan el campo nuevo.

¿Cuál te deja rasguear un acorde y el siguiente no? ¿Cuál cambia menos archivos? Los estudios de verdad tienen los dos.

Practica: diseñar a través de varios crates.

17. Piénsalo: las herramientas

- La selección guarda las notas por valor, lo que funciona porque un canal nunca tiene dos notas en el mismo lugar: `same_place` compara altura, comienzo y `nudge`. Imagina dos notas exactamente en el mismo lugar. ¿Qué fallaría en la

selección y en `replace_notes`? ¿Lo arreglaría un identificador permanente para cada nota, y qué costarían los identificadores en el archivo?
- `tools::transpose` se niega a mover un acorde si una nota saldría del rango MIDI. Otro diseño movería las notas que pueden moverse y dejaría el resto. ¿Cuándo sería mejor? ¿Cómo podría el piano roll decirte por qué no pasó nada?

Oír la armonía y encontrar la tonalidad

18. Un bajo caminante, a mano (lápiz y papel, fácil)

Oír la armonía calcula cómo puntúan C y C7 sobre un bajo caminante, con el Si♭ de paso pesando 0,15 por paso, y dice que con 0,35 C seguiría ganando, por 0,941 a 0,925. Calcula tú esos dos números: el Si♭ pesa entonces 1,4 y el perfil 29,4. Luego cambia `PASSING_BASS` en `harmony/hearing.rs` y escribe el caso como un test. ¿Con qué peso gana C7?

Practica: seguir una regla de puntuación a mano antes de confiar en ella.

19. Cuándo cortar un compás (lápiz y papel, fácil)

Cuando la hoja de acordes se llena a partir de las notas, un corte cuesta 0,08 a mitad del compás (`CUT_HALF_BAR`) y 0,12 en cualquier otro pulso (`CUT_BEAT`) (mira Cortar un compás en acordes). Un compás de 4/4 toca C, y luego G7. Cada uno, oído solo, puntúa 1,02 por pulso; el compás entero, oído como un solo tramo, 1,0 por pulso.

¿Corta la hoja el compás cuando G7 llega en el pulso 4, durante un pulso? ¿Y cuando llega en el pulso 3, durante dos? Mira el `if total > here.0` del bucle: ¿qué pasa en un empate? ¿Qué tan alta tiene que ser la puntuación de cada acorde solo para que haya corte en cada caso?

Practica: programación dinámica sobre un caso pequeño.

20. El acorde de los Augurios (la aplicación y lápiz, fácil)

Toca Mi, Sol♯ y Si con la mano izquierda y Mi♭, Sol, Si♭ y Re♭ con la derecha: el acorde que abre los Augurios primaverales de *La consagración de la primavera*. Ahora toca la escala de La♭ menor armónica, La♭ Si♭ Do♭ Re♭ Mi♭ Fa♭ Sol, y comprueba que todas las

notas del acorde están en ella. ¿Por qué `in_one_key`, en `harmony/hearing.rs`, deja fuera la menor armónica?

Luego una pregunta sobre la caché de `write_heard_chords`: su clave guarda también las notas de los compases de alrededor, no solo las del compás. ¿Por qué?

Practica: la escritura enarmónica, qué tiene que incluir la clave de una caché.

21. Medir el oído de la biblioteca (lineup-core, medio)

La biblioteca tiene un oído propio más sencillo, `library::listen::hear_chords`, para los patrones de la banda, y `every_style_is_heard_as_its_chart` lo comprueba sobre cada estilo tal como está escrito. Escribe un informe propio, un test ignorado: recorre los estilos, toca cada cifrado con cada `Comping` y algunas líneas de `Bass` a través de `chart::accompany`, escúchalo, y cuenta cuántas veces vuelven los acordes como estaban escritos. Imprime las combinaciones que más fallan. Luego intenta subir el número cambiando una constante al principio de `listen.rs`, sin romper el test que ya existe. Ejecútalo con `cargo test --release`: el mismo test corre muchas veces más rápido.

Practica: medir antes de cambiar, bucles anidados sobre enums.

22. Dónde cambia de tonalidad la Habanera (la aplicación, fácil)

1. Abre la demostración Habanera (la de Bizet, de *Carmen*), activa 🔍 **Buscar la tonalidad**, y haz clic en los compases 9, 10 y 11 de la regla. ¿Dónde exactamente pasa Re menor a Re mayor? ¿Es el estribillo en Re mayor solo «la misma melodía en otra tonalidad»? Compara las primeras notas de los dos.
2. Mantén La, Do y Mi en el teclado de escribir durante unos compases, y luego añade Sol♯, la séptima elevada. ¿Se vuelve el oyente más seguro de La menor? ¿Por qué importa tanto una sola nota?

Practica: oír una modulación, qué hace que una tonalidad sea una tonalidad.

23. El precio de un cambio de tonalidad (lineup-core, fácil)

En `harmony/key_finding.rs`, pon `CHANGE_COST` en 0,8 y ejecuta el informe:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored
--nocapture
```

¿Cuántos cambios se encuentran ahora, y cuántos de ellos no están en las partituras? Prueba 3,0. ¿Qué te dice eso de la penalización?

Practica: ajustar midiendo.

Conjuntos, series y micromodos

24. Un recorrido por la ventana Serie (la aplicación, fácil)

1. Abre **Archivo > Demostraciones > Dodecafónico > Coral de tríadas** y pulsa Espacio. Mira los nombres de las secciones en la lista de reproducción: la misma música, luego al revés, invertida, y las dos cosas.
2. Abre el piano roll (**F7**) y haz clic en **Serie…**. Elige **Series famosas y de ejemplo > Suite de Schoenberg**. ¿Qué dice la ventana de sus grupos de tres, y con qué forma es combinatoria?
3. Haz clic en la etiqueta **I11** de arriba de la matriz, pon **Juntas** en **6: hexacordios**, y haz clic en **Escribir I11 en el patrón actual**: dos hexacordios como acordes. Pulsa **Ctrl+Z** para deshacerlo.
4. Marca las tres casillas de **Serialismo**, pon **Octavas** en **Amplio**, y vuelve a escribir P4: puntos de sonido al estilo de Webern.
5. Haz clic en **Conjuntos…**, escribe `octatonic`, y haz clic en **Usar como tonalidad**. Las teclas de armonía tocan ahora acordes de la escala octatónica.

25. Formas distintas (lineup-core, pequeño)

Una serie tiene 48 formas, pero no siempre son 48 series distintas. Escribe un test que las junte en un `HashSet<ToneRow>` y las cuente. Comprueba que la serie de la Sinfonía de Webern tiene 24 y la de la Suite de Schoenberg 48. ¿Puede alguna serie tener menos de 24? (¿Podría una serie ser una de sus propias formas I? Escribe qué significa eso para cada nota, y verás que a lo sumo dos notas podrían cumplirlo.)

Practica: `HashSet`, una demostración contando.

26. Series derivadas (lineup-core, medio)

El Concierto, Op. 24, de Webern usa una serie cuyos cuatro grupos de tres pertenecen todos a la misma clase, 3-3 [014] (mira Micromodos). Una serie así se llama **derivada**. Añade `ToneRow::is_derived(size)`, verdadero cuando cada segmento de ese tamaño tiene la misma clase de conjunto, y un `random_derived` que construya una con vuelta atrás, como hace `random_all_interval`.

Practica: vuelta atrás, `segments` y clases de conjuntos.

27. A la manera de Forte (lineup-core, pequeño)

En seis clases, la forma prima de Forte difiere de la de Rahn, que es la que usa Ferrous Lineup (mira Conjuntos de clases de altura). Escribe `forte_prime()`, que desempata desde el principio del conjunto en lugar de desde el final, y muéstrala en los detalles de la ventana Conjuntos cuando sea distinta. ¿Por qué el truco de «el número más pequeño» no puede hacer esta directamente? (Piensa en qué bits mira primero una comparación de números.)

Practica: bits como conjuntos, leer una definición con cuidado.

28. Multiplicar por 5 (lineup-core, medio)

Multiplicar cada clase de altura por 5 (módulo 12) convierte la escala cromática en el círculo de cuartas: 0 5 10 3 8 1… Esta operación, **M5**, es otra manera de transformar una serie o un conjunto, que usaron algunos compositores después de Webern. Añade `PitchSet::multiply` y `ToneRow::multiply`, y encuentra qué clases de conjuntos convierte M5 en una clase distinta. (Vale la pena mirar los pares relacionados por Z.)

Practica: aritmética modular, probar sobre todas las clases de conjuntos.

29. Una serie propia (lineup-core y lineup-app, medio)

Con **Duraciones** marcada, la serie escribe sus propios números como duraciones. Deja que el usuario escriba una segunda serie, usada solo para las duraciones, y muestra las dos en la ventana. `RowWriting` deriva `Copy`: ¿por qué sigue compilando cuando guarda una segunda `ToneRow`, y por qué no compilaría si la segunda serie fuera un `Vec<u8>`?

Practica: `Copy` y lo que le pide a cada campo.

30. Las reglas dodecafónicas de oído (la aplicación, fácil)

1. Elige **serie dodecafónica** como tonalidad, elige **Suite de Schoenberg** en **Serie…** con Mi como nota de la tonalidad, y escribe P4 (Mi Fa Sol Do♯ Fa♯ Mi♭ La♭ Re Si Do La Si♭) como melodía. Marca **Conducción de voces**. Luego intercambia dos notas, y luego deja una fuera. ¿Cuántas notas marca la revisión cada vez, y adónde va el contorno punteado?
2. Escribe las tres primeras notas de P4 como un acorde, y el resto como melodía. Luego escribe las notas 1, 2 y 4 como el acorde. ¿Qué cambia?
3. Escribe P4 en una pista y P4 una octava más abajo en otra, con el mismo ritmo. ¿Qué dice la revisión, y por qué lo dice una sola vez?
4. Escribe un acorde de Do, Mi y Sol con una serie como tonalidad. Luego Sol, Si, Re y Fa. ¿Qué regla rompen, y por qué Berg la rompía a propósito?

31. Una nota repetida (lineup-core, pequeño)

En `Walk::take`, en `harmony/voice_leading/serial.rs`, una nota que solo repite la nota recién tocada nunca está fuera de lugar. Encuentra las líneas que lo hacen así. Luego escribe un test en que un acorde comparte una nota con el acorde anterior, y comprueba que no se marca nada.

Practica: leer máscaras de bits, un test para una excepción a una regla.

32. Un recorrido por los micromodos (la aplicación, fácil)

1. Abre el piano roll (**F7**) en el patrón **Acordes** de la canción inicial, pulsa **Ctrl+A**, y haz clic en **Micromodos…**. Lee los colores de Do mayor: ocho Mayor 2, seis tríadas, una disminuida.
2. Haz clic en **Menor 3**, y luego en **▶ Escucharlo**. Prueba **Invertido (I)** y **Transportarlo**, y mira cómo el triángulo gira y se da vuelta en el reloj.
3. Haz clic en **3-3 en Conjuntos →**. La ventana Conjuntos se abre en 3-3. Elige **7-35** y haz clic en uno de los micromodos de **Micromodos que contiene** para volver.
4. Abre **Archivo > Demostraciones > Micromodos > Tríadas, dos notas fijas** y pulsa Espacio. Escucha la única nota que se mueve en cada acorde.
5. Haz clic en **Los tricordios de Webern**, en el grupo **Series dodecafónicas** de la Biblioteca. Con **3 notas** elegido bajo la tonalidad, las teclas de armonía tocan sus cuatro grupos de tres, cada uno con la etiqueta 3-3. Ahora cambia el menú de la tonalidad de **serie dodecafónica** a **conjunto de clases de alturas**: la tonalidad

pasa a ser el primer grupo de tres de la serie, y las teclas de armonía tocan sus formas.
6. De vuelta en la canción inicial, mientras suena, haz clic en **Groove de cuartas** en la Biblioteca: la canción sigue en acordes sus.

33. Los micromodos de una melodía (lineup-core, pequeño)

Escribe `melody_micromodes(notes: &[PitchClass]) -> Vec<Option<&'static Micromode>>`, que nombra el micromodo de cada tres notas seguidas: las notas 1 a 3, 2 a 4, 3 a 5 y así (`windows(3)` hace el recorrido). ¿Por qué la respuesta es `None` en algunas ventanas? Pruébalo con la serie del Concierto de Webern: sus cuatro grupos son Menor 3, pero ¿qué son las ventanas que quedan a caballo entre dos grupos?

Practica: `windows`, `Option` en una lista.

34. Todos los colores a la vez (lineup-core, pequeño)

Recorre `SetClass::all()` y encuentra los conjuntos más pequeños cuyo contenido de micromodos no tiene ningún cero: conjuntos que contienen los doce micromodos. ¿Cuántas notas necesitan, y cuáles son? (Para el tamaño: la tríada aumentada y Menor 1 tienen que caber los dos.)

Practica: buscar en una tabla, `micromodes_in`.

35. Las tonalidades de Schat (lineup-core, medio)

La serie del Concierto de Webern reparte las doce notas en cuatro formas de Menor 3. Escribe una búsqueda con vuelta atrás que encuentre todas las maneras de repartir las doce notas en cuatro formas de un micromodo, y cuéntalas para cada uno de los doce. Un micromodo no puede hacerlo de ninguna manera: ¿cuál, y por qué? (Piensa en qué notas pueden tomar las formas de cada micromodo.)

Practica: vuelta atrás, contar con cuidado.

36. Un recorrido que recuerda más (lineup-core, medio)

El recorrido de un estilo de micromodos que conserva dos notas elige primero la nota nueva más fresca, y solo después la forma usada hace más tiempo (mira El recorrido: elegir con una tupla). Haz un recorrido que use con seguridad todas las formas a las

que puede llegar antes de que vuelva ninguna, y comprueba en un test que Menor 3 visita sus 24 formas en 24 pasos. ¿Es siempre posible? ¿Para qué micromodos puede un recorrido de «conservar dos notas» visitar cada forma exactamente una vez? (Es una pregunta sobre caminos en un grafo.)

Practica: grafos, vuelta atrás.

37. Escuchar la familia (lineup-app, pequeño)

Añade un ▶ junto al nombre de cada familia en la ventana Micromodos que toque sus micromodos uno tras otro, desde el Do central. `MicroAction::Hear` toca las notas juntas; ¿qué añadirías para tocarlas por turnos? Mira cómo se detiene la vista previa de un acorde en el piano roll.

Practica: acciones como datos, el tiempo en la ventana.

La conducción de voces

38. Octavas paralelas (la aplicación, fácil)

Escribe I–IV–V–I en Do mayor a cuatro voces, con el bajo Do3 Fa3 Sol3 Do3 y la soprano Mi5 Fa5 Re5 Mi5. Marca **Conducción de voces**. Luego mueve el Re5 de la soprano a Sol5 y lee lo que dice la revisión. ¿Por qué se evitan las octavas entre la soprano y el bajo, mientras que las terceras y las sextas entre ellos son bienvenidas?

39. La sensible y la séptima (la aplicación, fácil)

1. Escribe G7 a C con la séptima, Fa, en la contralto bajando a Mi. Luego escribe G7 a Am, la cadencia rota. ¿Adónde tiene que ir ahora la sensible? ¿Y la séptima?
2. La revisión deja en paz la sensible en las voces interiores. Escribe V7 a I con el tenor yendo de Si hacia abajo a Sol. ¿Qué gana el acorde de Do, y qué le faltaría si el tenor subiera a Do?
3. Escribe C, F, G7, C a cuatro voces con el bajo Do3 Fa2 Sol2 Do3, y cada voz interior moviéndose por grado conjunto o manteniéndose. Lee la línea de movimientos en cada cambio: ¿están las voces extremas alguna vez en movimiento paralelo o directo?

40. De un unísono a una octava (lineup-core, pequeño)

En `kind_of_move`, un unísono que va a una octava cae hasta los dos últimos brazos del `match`. Saca qué tipo recibe cuando las dos voces suben, y luego escribe un test en `voice_leading.rs` que lo compruebe, con el ayudante `satb` que usan los otros tests.

Practica: leer un `match` brazo por brazo.

41. Un paso de tres semitonos (lineup-core, pequeño)

`Move::size` llama paso a uno o dos semitonos. Una segunda aumentada, La♭ a Si, son tres semitonos, y su nombre dice «segunda». ¿Qué tamaño le da el código, y cuál le daría un profesor de armonía? Cambia `size` para que cuente letras, como hace `motion`, y escribe un test. ¿Qué otros movimientos cambian de tamaño?

Practica: letras y semitonos, otra vez.

Importar una grabación

42. Una plantilla sus4 (lineup-listen, medio)

`chords.rs` conoce seis cualidades. Añade `ChordQuality::Sus4` a `QUALITIES` y mira qué hace en una grabación con un acorde sus: renderiza a WAV una demostración que tenga uno (el test `write_wav` de `tests/key_changes.rs` lo hace). Más candidatos son más maneras de equivocarse. ¿Necesita `SEVENTH_COST` un primo para los acordes raros?

Practica: medir un cambio sobre datos conocidos.

43. Seis por ocho (lineup-listen, medio)

`meter::guess` conoce 3 y 4 pulsos por compás. Añade el 6/8: seis pulsos en dos grupos de tres, donde los acordes cambian en los pulsos 1 y 4. ¿Cómo distingues 6/8 de 3/4 al doble del tempo, cuando los dos ponen cambios cada tres pulsos? (Quizás no se pueda solo con los acordes: ¿qué otra cosa podría distinguirlos?)

Practica: los límites de un método.

44. Una grabación sin copia (lineup-audio, lineup-core y lineup-listen, más difícil)

La importación guarda una copia de la grabación con una entrada de silencio (mira Asperezas), solo porque una nota del sampler toca su sonido desde la primera muestra. El sampler ya puede empezar una voz por la mitad (`start_at`, que usa `resume`, mira Empezar por la mitad). Dale a `SampleSource::File` un desplazamiento de comienzo en segundos, negativo para un silencio antes del sonido, y deja que el `note_on` del sampler empiece ahí. Luego borra `write_wav`, y usa la entrada como desplazamiento. ¿Qué tiene que sumarle `resume` a `elapsed`? Y `SampleSource` se guarda en los archivos de proyecto: ¿qué número de formato necesita el cambio, y cómo se lee un archivo más viejo?

Practica: cambiar un tipo que se guarda en archivos, el hilo de audio.

45. Una tonalidad clara, o la de toda la grabación (lineup-listen, fácil)

`keys::main_key` toma la tonalidad del tramo más largo, aunque la tonalidad de ese tramo no sea clara (`KeySpan::is_clear`, mira Los cambios de tonalidad). Prueba otras dos reglas: la tonalidad del tramo *claro* más largo, y la tonalidad de toda la grabación que da `bpm-key` (`whole` en `analyze`) cuando el tramo más largo no es claro. Ejecuta el informe de `tests/key_changes.rs` para cada una. ¿Qué regla acierta más canciones, y son las mismas las canciones en que falla?

Practica: dejar que decida una medición.

46. Piénsalo: escuchar

- Los acordes se nombran por pulso, y el chroma de un pulso es el promedio de sus ventanas. ¿Qué le pasa a un acorde que cambia en el «y» de un pulso, como en mucho jazz y mucha música latina?
- ¿Por qué el compás se adivina a partir de los acordes y no de la intensidad de los pulsos, como siente el tiempo fuerte quien escucha?
- La importación corre en su propio hilo y se suelta, no se detiene, cuando se abre otro proyecto. ¿Cuándo valdría la pena detenerla, y cómo le dirías a un hilo que se detenga? (Mira cómo se detiene una exportación.)

La biblioteca: estilos y ritmos como datos

Archivos: `crates/lineup-core/src/library/`, `crates/lineup-core/src/sources.rs`, `crates/lineup-app/src/library_browser.rs` y `crates/lineup-app/src/app/library.rs`

Un músico que se sienta a tocar una bossa nova sabe cuatro cosas: los acordes, cómo los toca la guitarra, qué hace el bajo y el patrón de la batería. Nada de eso es código. Es *conocimiento*, y la mejor manera de meter conocimiento en un programa es escribirlo como datos, en una forma que una persona pueda leer y comprobar, y que el código que lo usa sea pequeño.

La biblioteca hace eso: 215 estilos armónicos, 139 ritmos y 6 rellenos, cada uno unas pocas líneas en una tabla, más los estilos dodecafónicos y de micromodos de los capítulos de teoría y las canciones de Cómo se escribe una demostración. Este capítulo recorre las notaciones y después sigue un estilo desde la tabla hasta una canción: qué canal toca qué, cómo cambia de estilo una canción entera, cómo cada parte de la canción recuerda de dónde vino, y la pestaña Biblioteca que lo lista todo. Cómo se usa la Biblioteca está en el capítulo del manual Los estilos y la Biblioteca.

Cifrados: acordes compás por compás

Una progresión escrita con nombres de acordes, «C G Am F», solo sirve en Do. Escrita con números romanos, «I V vi IV», sirve en cualquier tonalidad (ver Leer un número romano). Un **cifrado** escribe los números compás por compás, como una hoja guía, con `|` entre compases:

```
I | IV V | vi | ii7 V7
```

Los acordes de un compás se lo reparten por igual, así que «IV V» es medio compás cada uno. Un acorde que repite el anterior se sostiene, no se vuelve a tocar: «I | I» son dos compases de un mismo acorde. `chart::parse_chart(text, bar)` convierte un cifrado en `ChordSpan`s, en pasos de un compás de `bar` pasos:

```
pub struct ChordSpan {
    pub chord: RomanChord,
    pub start: usize,
    pub length: usize,
    /// Whether the band plays it early (see [`Push`]).
    pub push: Option<Push>,
    /// Whether the band stops playing its style here (see [`Stop`]).
    pub stop: Option<Stop>,
}
```

Devuelve `None` cuando no se puede leer un acorde, un compás está vacío o un compás no se puede repartir por igual entre sus acordes: una entrada de la tabla que está mal hace fallar una prueba en lugar de tocar algo sin sentido. `push` y `stop` son las marcas de Golpes, sostenidos, silencios y anticipaciones.

El comping: cómo se tocan los acordes

Los mismos cuatro acordes pueden ser una balada, un reggae o un tema de house. Lo que cambia es el ritmo con que se golpean los acordes, lo que los músicos de jazz llaman **comping** (de *accompanying*, acompañar), y lo que hace el bajo. Cada manera de hacer comping es una tabla corta de **golpes**: cuándo empieza una nota, cuánto dura, qué notas del acorde toca y con qué fuerza.

```
struct Hit<T> {
    /// When it starts, in steps from the start of the pattern's period.
    at: usize,
    /// How long it lasts, in steps.
    len: usize,
    what: T,
    velocity: u8,
}
```

Para los acordes, `what` es un `Voices`: `All`, todas las notas; `Upper`, todas menos la más grave, el golpe ascendente más suave de un rasgueo; o `Nth(n)`, una nota contada desde abajo, para los arpegios. Para el bajo es un `Tone`: la fundamental, la quinta, la octava, la tercera, la séptima, la fundamental del acorde siguiente tocada antes (una **anticipación**) o la nota a un semitono de ella, que conduce a la siguiente. Un único struct genérico sirve para los dos, con `T` puesto dos veces.

```
Comping::Pulse => (4, &[hit!(0, 2, All, 96), hit!(2, 2, All, 78)]),
Comping::Quarters => (4, &[hit!(0, 2, All, 100)]),
Comping::Offbeat => (4, &[hit!(2, 1, All, 92)]),
```

El primer número es el **período**: los golpes se repiten cada 4 pasos, un pulso. `Pulse` son corcheas parejas, la segunda más suave; `Offbeat` es el *skank* del reggae, un acorde corto en la segunda corchea de cada pulso. El largo, corto, largo, largo de la habanera ocupa 8 pasos:

```
Comping::Habanera => (
    8,
    &[
        hit!(0, 3, All, 96),
        hit!(3, 1, Upper, 74),
        hit!(4, 2, All, 86),
        hit!(6, 2, All, 84),
    ],
),
```

`macro_rules!`, y por qué no una función

`hit!` es una pequeña **macro declarativa**:

```
macro_rules! hit {
    ($at:expr, $len:expr, $what:expr, $velocity:expr) => {
        Hit {
            at: $at,
            len: $len,
            what: $what,
            velocity: $velocity,
        }
    };
}
```

Una macro es código que escribe código. `macro_rules!` compara lo que recibe con un patrón, aquí cuatro expresiones (`$at:expr` y las demás), y las pega en la plantilla de la derecha. `hit!(0, 2, All, 96)` se convierte en `Hit { at: 0, len: 2, what: All, velocity: 96 }` antes de que el compilador compruebe nada.

¿Por qué no una `const fn hit(...) -> Hit`? Hay que mirar lo que devuelve el `match`: `&'static [Hit<Voices>]`, un slice prestado que vive todo el programa. Escribir `&[ ... ]` con valores simples da exactamente eso: el compilador ve una expresión hecha solo de constantes, la pone en los datos de solo lectura del programa y deja que el préstamo viva para siempre. Eso se llama **promoción de constantes**. Solo ocurre con valores simples: una *llamada a función* dentro de los corchetes, aunque sea una `const fn`, no se promueve, y el compilador dice que el temporal prestado se descarta al final de la sentencia. La macro hace que la tabla sea

corta de escribir y siga hecha de valores simples. El comentario sobre la macro lo dice, para que nadie la «simplifique» convirtiéndola en una función.

Estilos: una tabla en un `static`

Un estilo lo junta todo: un cifrado en un modo, una manera de hacer comping, una línea de bajo, un tempo habitual y el ritmo que lo acompaña.

```
pub struct Style {
    pub name: &'static str,
    pub genre: Genre,
    /// The mode the chords are written in: the project's scale becomes
    /// this mode when the style is used.
    pub mode: ScaleKind,
    pub time: TimeSignature,
    /// The progression, as a chord chart (see [`chart::parse_chart`]):
    /// numerals counted from the tonic, bars separated by `|`.
    pub chart: &'static str,
    pub comping: Comping,
    pub bass: Bass,
    /// The tempo the style is usually played at, in BPM.
    pub tempo: f32,
    /// The name of the rhythm in [`super::RHYTHMS`] that goes with it, or
    /// "" for none.
    pub rhythm: &'static str,
    /// One or two sentences on what to listen for.
    pub about: &'static str,
}
```

Cada campo es un `&'static str`, algo `Copy` o un enum pequeño, así que un `Style` es `Copy` y la tabla no necesita reservar memoria. Una entrada se lee como una frase:

```
style(
    "Four chords",
    Pop,
    Major,
    "I | V | vi | IV",
    Comping::Pulse,
    Bass::Pulse,
    110.0,
    "Pop beat",
    "The four chords behind hundreds of hits: home, the dominant, the sad
relative minor, \
     and the subdominant.",
),
```

`style` es una `const fn` que pone 4/4, con `#[allow(clippy::too_many_arguments)]` porque nueve argumentos en una fila de tabla son más claros que nueve nombres de campo. En la aplicación, este estilo se llama «Cuatro acordes» y su ritmo «Ritmo pop»: las tablas guardan los nombres en inglés y la interfaz los traduce al mostrarlos. Los estilos en otro compás añaden `.in_time(THREE_FOUR)`, otra `const fn`, que usa la **sintaxis de actualización de structs**:

```
const fn in_time(self, time: TimeSignature) -> Style {
    Style { time, ..self }
}
```

`..self` quiere decir «todos los demás campos como en `self`». La tabla misma es un `static`:

```
pub static STYLES: &[Style] = &[
```

Un `const` se pega en cada lugar que lo usa, como una macro; un `static` es un único valor en una única dirección durante todo el programa. Para una tabla grande que se usa en muchos lugares, `static` guarda una sola copia, y un `&'static Style` que apunta a ella es tan barato de pasar como un número y nunca puede quedar colgando.

`Style::render(tonic)` toca un estilo en cualquier tonalidad: el cifrado se convierte en tramos, cada número romano en un acorde de la tonalidad, los acordes se disponen uno tras otro moviéndose lo menos posible, y el comping y el bajo los tocan. Ese es el trabajo propio de la banda, en Cómo toca la banda una hoja de acordes.

Ritmos, una línea por tambor

Los bateristas y las cajas de ritmos piensan un ritmo como una cuadrícula: una fila por tambor, una columna por paso. La biblioteca escribe exactamente eso, como texto:

```
rhythm(
    "Pop beat",
    Pop,
    110.0,
    &[
        (Kick, "X... .... X.x. ...."),
        (Snare, ".... X... .... X..."),
        (HiHat, "x.x. x.x. x.x. x.x."),
    ],
    "Snare roll",
    "The backbeat: kick on 1 and 3, snare on 2 and 4, hi-hat on every
eighth note.",
),
```

Cada carácter es un paso. `X` es un golpe fuerte, `x` uno normal, `o` una **nota fantasma** (*ghost note*) suave y `.` silencio. `h` y `l` tocan el tambor cinco semitonos más agudo o más grave, para los tonos abierto y apagado de una conga o para dos toms. Los espacios solo sirven para ver bien los pulsos, y `|` separa compases, en los ritmos de más de uno. «Snare roll» (el «Redoble de caja» de la aplicación) nombra el **relleno**, un corte de un compás que cierra una sección.

Un carácter se convierte en un golpe:

```
fn hit(character: char) -> Option<Option<Hit>> {
    match character {
        '.' => Some(None),
        'X' => Some(Some((120, 0))),
        'x' => Some(Some((96, 0))),
        'o' => Some(Some((56, 0))),
        'h' => Some(Some((96, 5))),
        'l' => Some(Some((96, -5))),
        _ => None,
    }
}
```

Un `Option` dentro de un `Option` dice dos cosas distintas: el `None` de fuera es «este no es un carácter que un ritmo pueda tener», así que una línea con una errata se rechaza; el `None` de dentro es «un paso de silencio», una respuesta perfectamente válida. Aquí `Hit` es un alias de `type`, `(u8, i32)`: una intensidad y un desplazamiento en semitonos.

Las mismas dos letras escriben los usul turcos de los ritmos balcánicos. En la darbuka, `l` es el golpe grave, **düm**, y `h` el agudo, **tek**, y los comentarios citan de dónde viene cada patrón:

```
rhythm(
    "Aksak",
    Balkan,
    90.0,
    &[
        (Kick, "X... .... X... ......"),
        (Conga, "l... h.h. l... h...h."),
        (Shaker, "x.o. x.o. x.o. x.o.o."),
    ],
    "",
    "The Turkish aksak, 9/8 counted 2 + 2 + 2 + 3, on the darbuka: düm,
te-ke, \
     düm, tek-tek, the low düm on the first and third beats, the long last
beat \
     split two and one.",
)
.in_time(TimeSignature::of_groups(&[2, 2, 2, 3], 8)),
```

Cuando un ritmo entra en un patrón, se reparte compás por compás en el compás del patrón:

```
for start in (0..length).step_by(bar) {
    let steps = &bars[(start / bar) % bars.len()];
    for (step, hit) in steps.iter().enumerate().take(bar) {
```

`step_by(bar)` da el comienzo de cada compás, y `take(bar)` corta un compás largo en la barra de compás, así que un ritmo de 4/4 en un patrón de 3/4 pierde su último pulso en cada compás en lugar de desplazarse por encima de las barras. Unos pocos ritmos, los marcados `follows_the_meter`, se escriben de nuevo para el compás propio de cada compás, para la música cuyo compás cambia todo el tiempo (ver Una orquesta para música que cambia de compás).

Un ritmo para cualquier compás

Cuando cambias a mano el compás de una canción, hace falta un ritmo en el nuevo. `Rhythm::for_time` pregunta, por orden, hasta que algo responde:

```
RHYTHMS
    .iter()
    .filter(in_time)
    .find(same_kind)
    .or_else(|| {
        USUAL
            .iter()
            .filter_map(|name| Rhythm::named(name))
            .find(in_time)
    })
    .or_else(|| RHYTHMS.iter().find(in_time))
    .or_else(|| Rhythm::grouped_for(time, same_kind))
    .unwrap_or_else(|| Rhythm::basic(time))
```

Un ritmo del mismo tipo en el compás nuevo (un tango en 3/4 es un vals de tango); si no, el habitual (un vals para 3/4); si no, cualquiera en ese compás; si no, uno con el mismo número de las mismas notas por compás, agrupadas a su manera (7/8 se toca como 2+2+3/8, una rachenitsa); y si no, un ritmo sencillo hecho para ese compás. Cada `or_else` solo se ejecuta si todo lo anterior no encontró nada. Cómo pasan las notas de la canción al nuevo compás está en Pasar una canción a otro compás.

Ese último ritmo sencillo tiene un problema: un `Rhythm` guarda líneas `&'static str`, y 5/4 no está en ninguna tabla. `Rhythm::basic` hace el texto la primera vez que se lo piden, y lo guarda mientras dure el programa:

```
pub fn basic(time: TimeSignature) -> &'static Rhythm {
    static MADE: Mutex<Vec<&'static Rhythm>> = Mutex::new(Vec::new());
    let mut made = MADE.lock().unwrap_or_else(PoisonError::into_inner);
    if let Some(beat) = made.iter().find(|beat| beat.time == time) {
        return beat;
    }
    let lines: Vec<(Drum, &'static str)> = basic_lines(time)
        .into_iter()
        .map(|(drum, line)| (drum, &*String::leak(line)))
        .collect();
```

`String::leak`, `Vec::leak` y `Box::leak` renuncian a propósito a su memoria: nunca se libera, y a cambio se obtiene una referencia `&'static` a ella. Filtrar memoria cada vez crecería sin fin, así que los ritmos hechos se guardan en un `static Mutex<Vec<...>>` y cada compás se hace una sola vez: un puñado de cadenas pequeñas durante la vida del programa. `Mutex::new` es una `const fn`, y eso es lo que deja que un `static` guarde uno. Un candado envenenado (un hilo entró en pánico mientras lo tenía) sigue guardando datos válidos aquí, así que `unwrap_or_else(PoisonError::into_inner)` los toma igualmente.

Adónde van las notas

Las tablas no saben nada de canales. `library/apply.rs` decide adónde van sus notas: los acordes a un sintetizador llamado «Keys» (o al canal que tengas seleccionado), la línea de bajo a un sintetizador llamado «Bass», la melodía de una canción a «Lead» y cada tambor al canal que lo toca. Los canales que faltan se añaden, con un piano para Keys y un bajo eléctrico tocado con los dedos para Bass como sonidos General MIDI.

```
pub fn chords_channel(project: &mut Project, selected: Option<usize>) ->
Option<usize> {
    let free = |(index, channel): (usize, &Channel)| {
        plays_notes(channel)
            && channel.name != BASS
            && channel.name != LEAD
            && !project.is_locked(index)
    };
    if let Some(index) = selected
        && project
            .channels()
            .get(index)
            .is_some_and(|channel| free((index, channel)))
    {
        return Some(index);
    }
    let other = project
        .channels()
        .iter()
        .enumerate()
        .position(|(index, channel)| channel.kind == ChannelKind::Synth &&
free((index, channel)));
    find_synth(project, KEYS)
        .or(other)
        .or_else(|| synth_named(project, KEYS))
}
```

`free` es un closure guardado en una variable, para usarlo dos veces. Un canal **bloqueado** nunca se elige para los acordes (ver Forma, secciones, variaciones y bloqueos). La cadena del final muestra la diferencia entre `or` y `or_else`: `other` ya está calculado, así que `or` lo toma como valor; `synth_named` *añade un canal*, así que solo debe ejecutarse si no se encontró nada más, y va en un closure con `or_else`.

Los tambores se encuentran por lo que son, no por su nombre: el canal cuyo tipo es `Sampler(SampleSource::Builtin(drum))`. Un bombo renombrado «Bombo» sigue siendo el bombo. Un sintetizador llamado «Bass» que se envía por MIDI a un sintetizador externo sigue siendo el bajo, porque `plays_notes` acepta también los canales de salida MIDI (ver MIDI).

Tocar una canción en otro estilo

Probar un estilo en un patrón propio es fácil. Cambiar el estilo de una canción entera es más difícil: «hagamos esta como un reggae» quiere decir los mismos acordes, tocados de otra manera por todos. Pero un patrón no guarda sus acordes; guarda notas. Así que la biblioteca primero los **oye**, con el segundo oído de Oír la armonía, y después los vuelve a tocar.

Un ritmo dice cómo le gusta que se toquen los acordes a través del estilo del que es la batería:

```
pub fn feel(rhythm: &Rhythm) -> (Comping, Bass) {
    let same_time = |style: &&Style| style.time == rhythm.time;
    STYLES
        .iter()
        .find(|style| style.rhythm == rhythm.name)
        .or_else(|| {
            STYLES
                .iter()
                .filter(same_time)
                .find(|style| style.genre == rhythm.genre)
        })
        .or_else(|| STYLES.iter().find(same_time))
        .map_or((Comping::Pulse, Bass::Pulse), |style| {
            (style.comping, style.bass)
        })
}
```

El ritmo «Salsa» es la batería del estilo «Salsa montuno» (el «Montuno de salsa» de la aplicación), así que toca sus acordes como un montuno sobre un bajo de tumbao. Un ritmo que ningún estilo usa toma prestado un estilo de su género en su compás, así que un ritmo en 3/4 nunca recibe una manera de tocar pensada para 4/4. ¿Por qué `&&Style`? `STYLES.iter()` da `&Style`, y `filter` le pasa a su closure una referencia a cada elemento: una referencia a una referencia.

`recomp` vuelve a tocar un patrón: oye los acordes de su canal de acordes sobre su bajo, los dispone y deja que el nuevo comping y el nuevo bajo toquen cada tramo. Los tramos donde no se oyó ningún acorde conservan sus notas, así que una introducción sin acordes queda como estaba. `play_song` lo hace con cada patrón, y deja tres cosas en paz a propósito: la melodía, los patrones de la banda (que siguen la hoja de acordes, ver Cómo toca la banda una hoja de acordes) y los finales, un patrón cuyas notas empiezan todas en su primer paso, como un acorde final tenido con un platillo. Donde la batería anterior tenía un relleno en su último compás, va el relleno del ritmo nuevo. Un relleno es un último compás que no suena como ningún compás anterior:

```
// A fill is a last bar that plays like no bar before it.
let fill = length >= 2 * bar && {
    let last = drums_in(project, pattern, length - bar, bar);
    (0..length - bar)
        .step_by(bar)
        .all(|start| drums_in(project, pattern, start, bar) != last)
};
```

En Rust un bloque `{ ... }` es una expresión, cuyo valor es su última línea, así que puede ir después de `&&`; y como `&&` se detiene en cuanto su lado izquierdo es `false`, el bloque solo se ejecuta para patrones de dos compases o más. Un ritmo en otro compás primero cambia el compás de toda la canción, compás por compás, así que una canción en 4/4 tocada como vals pasa a ser una canción en 3/4.

Qué partes tomar

La ventana que abre un estilo en la aplicación pregunta qué partes tomar, en tres grupos: bajo **Acompañamiento**, **Batería**, **Partes de acordes y bajo** y **Voces / instrumentos**; bajo **Armonía**, **Progresión de acordes**; bajo **Tiempo**, **Compás**, **Tempo** y **Swing**. Son un struct pequeño:

```
pub struct StyleParts {
    pub rhythm: bool,
    pub instruments: bool,
    pub time_signature: bool,
    pub progression: bool,
    /// Take the preset's usual tempo where the meter stays the same. Off
    /// by default: the song keeps its tempo, so its own notes last as
    /// long as before.
    pub tempo: bool,
    pub swing: bool,
    /// Copy the preset's chord, bass and other accompaniment parts (a
    /// rhythm's are those of its matching style). Off, the song's own
    /// stay, as its drums do with `rhythm` off.
    pub matching_style: bool,
}
```

`rhythm` es la casilla **Batería** y `matching_style` la casilla **Partes de acordes y bajo**. Cada casilla va por su cuenta: lo que queda sin marcar, la canción lo conserva. En un patrón, cada casilla es un paso de `write_style_parts`, y cada paso toca solo sus propias notas:

```
if parts.time_signature {
    meter(project, index, style.time);
}
if parts.matching_style {
    recomp(project, index, style.comping, style.bass);
}
if parts.progression {
    progression(project, index, style);
}
// ...
if parts.rhythm
    && let Some(rhythm) = Rhythm::named(style.rhythm)
{
    write_rhythm(project, index, rhythm);
}
if parts.instruments {
    style_instruments(project, style, selected);
}
```

`recomp` vuelve a tocar los canales de acordes y de bajo y deja la batería; `write_rhythm` escribe los carriles de batería y deja el resto. Así una canción puede tomar la batería de un estilo y las partes de acordes y bajo de otro, en cualquier orden. Una banda guarda esa misma separación en un campo propio, `drums` (ver Cómo toca la banda un cifrado). La línea `if parts.rhythm && let Some(rhythm) = ...` es una *cadena de let* (*let chain*): el bloque se ejecuta solo cuando la casilla está marcada *y* se encuentra el ritmo del estilo, con `rhythm` ligado dentro de él.

Voces / instrumentos da los sonidos del estilo a los canales que tocan sus partes: Teclados y Bajo, o el canal elegido en el rack. Una canción importada de un archivo MIDI no tiene ninguno de los dos; sus canales llevan los nombres que les dio el archivo, como «PIANO MELODIA», «BANDONEON» o «CONTRABAJO». Entonces `own_players_instruments` da los sonidos del estilo a los músicos propios de la canción, a cada uno según la familia del instrumento que tenía, y deja la melodía en paz:

```
let program = match channel.program? {
    general_midi::CONTRABASS => band.bass_program(),
    // Violins to tremolo strings, and the string ensembles.
    40..=44 | 48..=51 => extras.strings.as_ref()?.program,
    program => match general_midi::FAMILIES[general_midi::family(program)]
{
        "Piano" => piano,
        "Organ" => extras.organ.as_ref().map_or(piano, |organ|
organ.program),
        "Guitar" => extras
            .guitar
            .as_ref()
            .map_or(piano, |guitar| guitar.program),
        "Bass" => band.bass_program(),
        _ => return None,
    },
};
```

El código está dentro de un cierre que se pasa a `filter_map`, que devuelve un `Option`, así que cada `?` y el `return None` significan «este canal conserva su sonido»: un canal sin programa General MIDI, las cuerdas cuando el estilo no tiene, un coro o un bandoneón, que ningún estilo tiene. Un contrabajo está en la familia de las cuerdas de General MIDI, así que se reconoce primero, por su número, y toma el bajo. La melodía es cualquier canal cuyo nombre lo diga (Lead, Melody, Melodía, Voice, Voz, Canto), o si no el que mostraría la partitura de melodía.

Dos métodos pequeños corrigen la elección según el tipo de preajuste, cada uno tomando `self` por valor y devolviendo una copia cambiada: `for_rhythm` apaga la progresión, que un ritmo no tiene; y `following` mantiene los compases y el tempo de la canción para un estilo que toca cada compás en su propio compás. Se encadenan: `parts.for_rhythm().following(rhythm.follows_the_meter)`.

La elección se guarda entre ejecuciones, en la memoria de egui, con el nombre `apply-options`. Hasta que la cambies, es todo menos la progresión de acordes, así que la canción conserva sus propios acordes:

```
const FIRST_OPTIONS: StyleParts = StyleParts {
    rhythm: true,
    instruments: true,
    time_signature: true,
    progression: false,
    tempo: true,
    swing: true,
    matching_style: true,
};
```

`#[serde(default)]` en el struct hace que un campo añadido más tarde se lea con su valor por defecto desde una copia guardada más antigua. Una `const` de un struct se escribe entera allí donde se usa, así que `FIRST_OPTIONS` no cuesta nada al ejecutarse.

El mismo estilo, la misma canción

Prueba Bluegrass, después Celtic, después Bluegrass otra vez: la tercera vez debería sonar exactamente como la primera. Un estilo no debe aplicarse encima del anterior, con el compás y el tempo del anterior, o cada clic alejaría un poco más la canción de sí misma. Así que la aplicación guarda, junto a cada versión de la canción que hizo la Biblioteca, la canción tal como vino, en su propio compás:

```
pub(super) struct SongSource {
    original: Project,
    library: Tried,
    /// Latest preset components, applied without changing the source
meter.
    reference: Project,
}
```

Cada clic vuelve a empezar desde `reference`, y pregunta si el compás del estilo es distinto del de *esa* canción, no del que le dejó el último estilo:

```
let changes_meter = !lineup_core::remeter::is_in(&source.reference, time);
```

El tempo sigue la misma idea. **Tempo** pone el tempo del estilo donde suena la mayor parte de la canción, con `take_tempo` (ver La franja de tempo): una canción que empieza con un compás lento y luego toca a 121 BPM, con un estilo hecho para 180, toca a 180 donde tocaba a 121, y su compás lento sigue siendo igual de más lento que antes. Dar el tempo del estilo al primer compás de la canción haría que un archivo MIDI con una entrada lenta se disparara.

De dónde vino cada parte

Tomar solo algunas partes significa que una canción puede tener su batería de un estilo, sus partes de acordes y bajo de otro, su tempo de un tercero y sus acordes de la demostración con la que empezó. La Biblioteca lo muestra, una línea por parte: Tempo «Tango marcato (estilo)», Acordes «El Choclo (demostración)», y **editado, era …** en cuanto cambias tú esa parte. `sources.rs` lleva ese registro.

Las nueve partes son un enum, en el orden en que las lista la Biblioteca: la batería, las partes de acordes y bajo, la melodía, los instrumentos, el compás, la tonalidad, los acordes, el tempo y el swing. Un conjunto de ellas son nueve bits de un `u16`, un bit por parte:

```
pub struct PartSet(u16);
```

```
pub fn contains(self, part: SongPart) -> bool {
    self.0 & part.bit() != 0
}
```

Todo cambio que trae partes pasa por una única función, que compara la canción antes y después:

```
pub fn bring_in<R>(
    &mut self,
    source: &Source,
    taken: PartSet,
    edited: PartSet,
    change: impl FnOnce(&mut Project) -> R,
) -> R {
    let before = SongPart::ALL.map(|part| self.part_fingerprint(part));
    let result = change(self);
    for (part, before) in SongPart::ALL.into_iter().zip(before) {
        let now = self.part_fingerprint(part);
        // A key that changed with what was taken came with it: a
        // minor progression puts the song in minor, a row style puts
        // it in its row.
        let came = taken.contains(part)
            || (part == SongPart::Key && now != before && taken !=
PartSet::NONE);
        if came && !edited.contains(part) {
            self.sources.set(
                part,
                Some(PartSource {
                    source: source.clone(),
                    print: Some(now),
                }),
            );
        } else if now != before && !edited.contains(part) {
            self.follow_part(part, before, now);
        }
    }
    result
}
```

Las partes tomadas ahora vienen de `source`, aunque no hayan cambiado: un estilo en 4/4 aplicado a una canción que ya está en 4/4 le da igual su compás. Una parte que cambió solo junto con ellas, como unos acordes movidos a un compás nuevo, *las*

sigue: sigue viniendo de donde venía, y está como vino si lo estaba antes. La tonalidad es la única parte que ninguna casilla nombra: viene con lo que la trajo, así que la tonalidad de una progresión menor viene de su estilo.

El `print` es una **huella**: un hash `u64` de la parte tal como vino, hecho con `std::hash::DefaultHasher` a partir de sus notas, su tempo, sus acordes. Si una parte está editada no se guarda en ningún sitio; se pregunta cada vez, comparando la huella con la parte tal como está ahora:

```
edited: came.print != Some(self.part_fingerprint(part)),
```

Así, editar el tempo lo deja «editado, era …», y volverlo a poner como estaba, o deshacer, lo deja otra vez como vino, sin código que mantenga al día un indicador. El archivo del proyecto guarda solo la fuente y un indicador `edited`, nunca el hash: `DefaultHasher` puede cambiar de algoritmo en otra versión de Rust, y un hash guardado llamaría entonces editada a cada parte de cada archivo antiguo. Al cargar, la huella se vuelve a calcular a partir de la parte tal como está.

Tres de las partes son notas de los mismos patrones: la batería, la melodía y las partes de acordes y bajo, que son todo lo demás. Se distinguen por canal, con un único ayudante que calcula el hash de las notas de los canales que elige un cierre:

```
SongPart::Melody => self.hash_notes(&mut hasher, is_lead),
```

y las partes de acordes y bajo pasan `|channel| !is_drum(channel) && !is_lead(channel)`. Con una banda, la batería y las partes de acordes y bajo son lo que toca la banda, así que sus huellas calculan en cambio el hash de los nombres de los estilos de los que vienen.

Las versiones anteriores conocían solo seis de las partes, y rechazarían un archivo que nombrara una que no conocen. Así que la batería, la melodía y la tonalidad se guardan en un campo propio, `more_sources`, que esas versiones se saltan. Se escribe siempre que alguna parte vino de algún sitio, aunque esté vacío, así que un archivo sin él es de antes; para un archivo así, `sources_of_an_older_file` deduce de dónde vinieron esas tres: la batería con los patrones, la melodía de una demostración con la demostración, y la tonalidad con los acordes.

Un detalle más. El registro de fuentes no es la música, y dos canciones que suenan igual deberían ser iguales: la aplicación compara proyectos para saber si hay cambios sin guardar. Así que su `PartialEq` dice que dos registros cualesquiera son iguales:

```
impl PartialEq for Sources {
    fn eq(&self, _: &Self) -> bool {
        true
    }
}
```

Un `impl` escrito a mano es la manera en que un tipo participa en un `PartialEq` derivado (el del `Project`) sin cambiar la respuesta.

La pestaña Biblioteca

La Biblioteca lista los estilos armónicos, los ritmos y tus propios estilos en un único catálogo, ordenado por los nombres tal como se muestran en el idioma elegido:

```
pub enum Preset<'a> {
    Style(&'static Style),
    Rhythm(&'static Rhythm),
    User(usize, &'a UserStyle),
}
```

Dos tipos de entrada toman prestado de tablas que viven para siempre; tus propios estilos viven en la aplicación, así que `Preset` tiene un tiempo de vida, `'a`, que dice que no puede vivir más que la lista de la que toma prestado. Cada pregunta que hace la lista (el nombre, el compás, el tempo, si trae una progresión) es un método con un único `match`, así que el resto del navegador nunca pregunta de qué tipo es. Un ritmo que es justo la batería de un estilo con el mismo nombre, tempo y aire se lista una sola vez, como el estilo.

Las casillas bajo la búsqueda dejan solo los preajustes parecidos a la canción, «En 4/4» (en su compás) o «Cerca de 120 BPM» (cerca de su tempo), o los que tienen lo que buscas: «Con acordes» (una progresión), «Swing» (una batería con swing) o un instrumento elegido en un menú, «Cualquier instrumento». Lo que miran en un preajuste es un struct `Copy` pequeño,

```
struct Traits {
    time: TimeSignature,
    /// The speed it was made for, in beats a minute.
    tempo: f32,
    /// Whether it plays every bar in its own meter, so it fits any time
    /// signature.
    any_meter: bool,
    /// Whether it brings a chord progression.
    chords: bool,
    /// Whether its drums swing.
    swing: bool,
}
```

y la decisión es un único método:

```
fn keeps(self, traits: Traits, instruments: impl FnOnce() ->
Vec<Instrument>) -> bool {
    let filters = self.filters;
    // 7/8 as 2+2+3 is still 7/8.
    let same_time = traits.any_meter || traits.time.ungrouped() ==
self.time.ungrouped();
    let ratio = traits.tempo / self.tempo;
    let near = ratio.max(1.0 / ratio) <= NEAR_SPEED;
    (!filters.time || same_time)
        && (!filters.speed || near)
        && (!filters.chords || traits.chords)
        && (!filters.swing || traits.swing)
        && filters
            .instrument
            .is_none_or(|instrument| instruments().contains(&instrument))
}
```

Las líneas se leen como reglas unidas por «y»: `!filters.time || same_time` es «o la casilla no está marcada, o el compás es el mismo». `ratio.max(1.0 / ratio)` convierte «la mitad de rápido» (0,5) y «el doble de rápido» (2,0) en el mismo número, 2, así que más rápido y más lento se juzgan igual; con `NEAR_SPEED` en 1,15, una canción a 130 BPM conserva los preajustes hechos para 113 a 149,5. Comparar proporciones en lugar de una diferencia en BPM trata igual las canciones lentas y las rápidas: ±15 BPM sería mucho a 60 y poco a 180. `keeps` toma `self` por valor: el struct es `Copy` y ocupa unos pocos bytes, así que no cuesta más que una referencia.

Los instrumentos son la única pregunta que cuesta algo: los de un preajuste son los de la banda que lo tocaría, calculados a partir de su estilo. Así que llegan como un cierre, `instruments`, al que se llama solo al final, cuando hay un instrumento elegido. `Option::is_none_or` es `true` para `None`, ningún instrumento elegido, y si no pregunta al cierre por el que hay dentro. `&&` se detiene en el primer `false`, así que

un preajuste que una casilla anterior ya dejó fuera nunca llega a calcular sus instrumentos.

La lista se agrupa por género, un `egui::CollapsingHeader` por género con su cuenta y tus propios estilos primero bajo **Estilos de usuario**, salvo que **Ajustes > Biblioteca de estilos** diga **Lista alfabética**. Una búsqueda abre cada grupo que tiene una coincidencia. Cuando la canción cambia de estilo, el grupo del nuevo se abre durante medio segundo, lo justo para que egui desplace su fila iluminada hasta la vista, y luego deja los grupos como los tenías.

Probar, cambiar y deshacer

Lo que hace un clic depende de la canción. Con una banda, como en una canción de **Archivo > Nuevo** o en una demostración, el estilo pasa a ser la manera en que la banda toca la hoja de acordes (ver Cómo toca la banda una hoja de acordes). Mientras la canción suena sin banda, el clic cambia la canción entera, así que la música sigue. Si no, escribe el estilo en un **patrón de prueba** propio y lo toca ahí.

Hacer clic en otro estilo debería *reemplazar* la prueba, no ir apilando patrones, para poder hojear la biblioteca. Pero en cuanto empiezas a editar ese patrón, es tuyo. La aplicación distingue los dos casos guardando una copia de lo que escribió:

```
pub(super) struct Audition {
    pattern: usize,
    /// The pattern as the library left it. If it still looks like this,
    /// the next click may rewrite it.
    written: Pattern,
    style: Option<&'static Style>,
    rhythm: Option<&'static Rhythm>,
    row_style: Option<&'static RowStyle>,
    micromode_style: Option<&'static MicromodeStyle>,
}
```

```
fn unchanged_audition(&self) -> Option<&Audition> {
    let pattern = self.project.pattern(self.pattern)?;
    self.auditions
        .iter()
        .rev()
        .find(|audition| audition.pattern == self.pattern &&
&audition.written == pattern)
}
```

`==` entre dos `Pattern`s compara cada nota, gracias al `PartialEq` derivado: en lugar de seguir cada edición, se guarda una copia y se compara, la misma idea que con los cambios sin guardar. La aplicación guarda las últimas 64 pruebas, no solo la más

nueva, y busca desde la más nueva con `rev()`. Eso es por el deshacer: después de que Ctrl+Z traiga de vuelta el estilo anterior, el patrón coincide con una copia más antigua, y el siguiente clic lo sigue cambiando en el mismo lugar.

Los estilos armónicos y los ritmos toman las partes que marcaste (ver Qué partes tomar). Los estilos de series y los de micromodos no tienen esa elección: se prueban de la misma manera, y solo se diferencian en las notas que escriben. Así que sus pasos comunes son un único método, y el paso distinto es un closure que se le pasa:

```
fn write_pattern_style(
    &mut self,
    name: &'static str,
    tempo: f32,
    rhythm: Option<&'static Rhythm>,
    write: impl FnOnce(&mut Project, usize),
) -> Option<(usize, Pattern)> {
```

`impl FnOnce(&mut Project, usize)` acepta cualquier closure que se pueda llamar una vez con esos dos argumentos. El compilador hace una copia del método para cada closure que se le pasa, así que la llamada no cuesta más que el código escrito a mano. `FnOnce` es el que menos pide de los tres traits de closure (`Fn`, `FnMut`, `FnOnce`): todo closure es al menos `FnOnce`, así que pedirlo los acepta todos.

Cada escritura pasa por `edit`, así que una prueba es un solo paso de deshacer, el hilo de audio se entera por el diff de siempre, y guardar funciona sin más. Dentro, `bring_in` registra de dónde vinieron las partes.

Cómo se prueba

Las tablas son datos, y los datos pueden estar mal tan fácilmente como el código, sin que nada se rompa. Así que las pruebas leen la biblioteca entera: `every_style_parses_and_fits_in_a_pattern`, `names_are_unique_and_rhythms_exist`, `every_genre_has_styles`, `styles_play_in_any_key`, y para los ritmos `every_line_is_whole_bars_of_known_characters`, `names_are_unique_and_fills_exist` y `congas_play_high_and_low`. `a_new_time_signature_keeps_the_kind_of_rhythm`, `an_odd_meter_gets_a_rhythm_grouped_its_way` y `a_time_signature_with_no_rhythm_gets_a_basic_beat` siguen a `for_time` por su cadena.

`apply.rs` comprueba adónde van las notas (`styles_go_to_keys_and_bass` , `chords_go_around_a_locked_channel`), que `recomping_keeps_the_chords` , que una canción entera toma un ritmo o un estilo nuevos, que un vals pone la canción en 3/4 y, la más amplia, `every_demo_takes_every_kind_of_library_click` . `sources.rs` comprueba una parte que llega, se edita y se deshace:

```
#[test]
fn a_part_brought_in_is_edited_once_changed_and_not_after_undoing_it() {
    let mut project = Project::demo();
    project.bring_in_all(&demo());
    assert_eq!(from(&project, SongPart::Tempo), Some(("El Choclo",
false)));
    let tempo = project.tempo;
    project.scale_tempo(Tempo::new(tempo.bpm() + 10.0));
    assert_eq!(from(&project, SongPart::Tempo), Some(("El Choclo",
true)));
    assert_eq!(from(&project, SongPart::Swing), Some(("El Choclo",
false)));
    project.scale_tempo(tempo);
    assert_eq!(from(&project, SongPart::Tempo), Some(("El Choclo",
false)));
}
```

Para las casillas, `a_styles_sounds_go_to_an_imported_songs_players_by_family_but_the_tune` arma una canción como un tango importado, con un «PIANO MELODIA», un bandoneón, una guitarra, un contrabajo, cuerdas y un coro, y comprueba qué canales toman los sonidos de Tango y de Bossa nova y cuáles conservan los suyos; `a_songs_keys_and_bass_still_take_a_styles_sounds_alone` comprueba que una canción con Teclados y Bajo cambia solo ahí. En `band.rs` , `drums_and_chord_parts_come_on_their_own_from_builtin_and_saved_styles` da a una banda de Rock ride primero la batería de Bossa nova, del estilo incorporado y de una copia guardada de él, y comprueba que el piano y el bajo quedan como estaban, que el archivo se vuelve a leer igual y que, cuando llegan también las partes de acordes y bajo, la banda vuelve a ser simplemente Bossa nova, en el formato que necesitaba antes; `a_styles_chord_parts_can_come_without_its_drums` toma las partes de acordes y bajo de Bossa nova sobre la batería de Rock ride, lo que necesita el formato 29. `a_style_tempo_goes_where_most_of_the_song_is` comprueba `take_tempo` en una canción con un primer compás lento.

En la aplicación, `the_library_tick_boxes_keep_only_presets_like_the_song` cuenta los preajustes listados a medida que se marcan las casillas; `the_same_style_brings_the_same_tempo_whatever_was_tried_before` pulsa Bluegrass, Celtic, Bluegrass y Celtic en una canción en 2/4 y en 4/4, y comprueba que cada estilo trae el mismo tempo cada vez; y

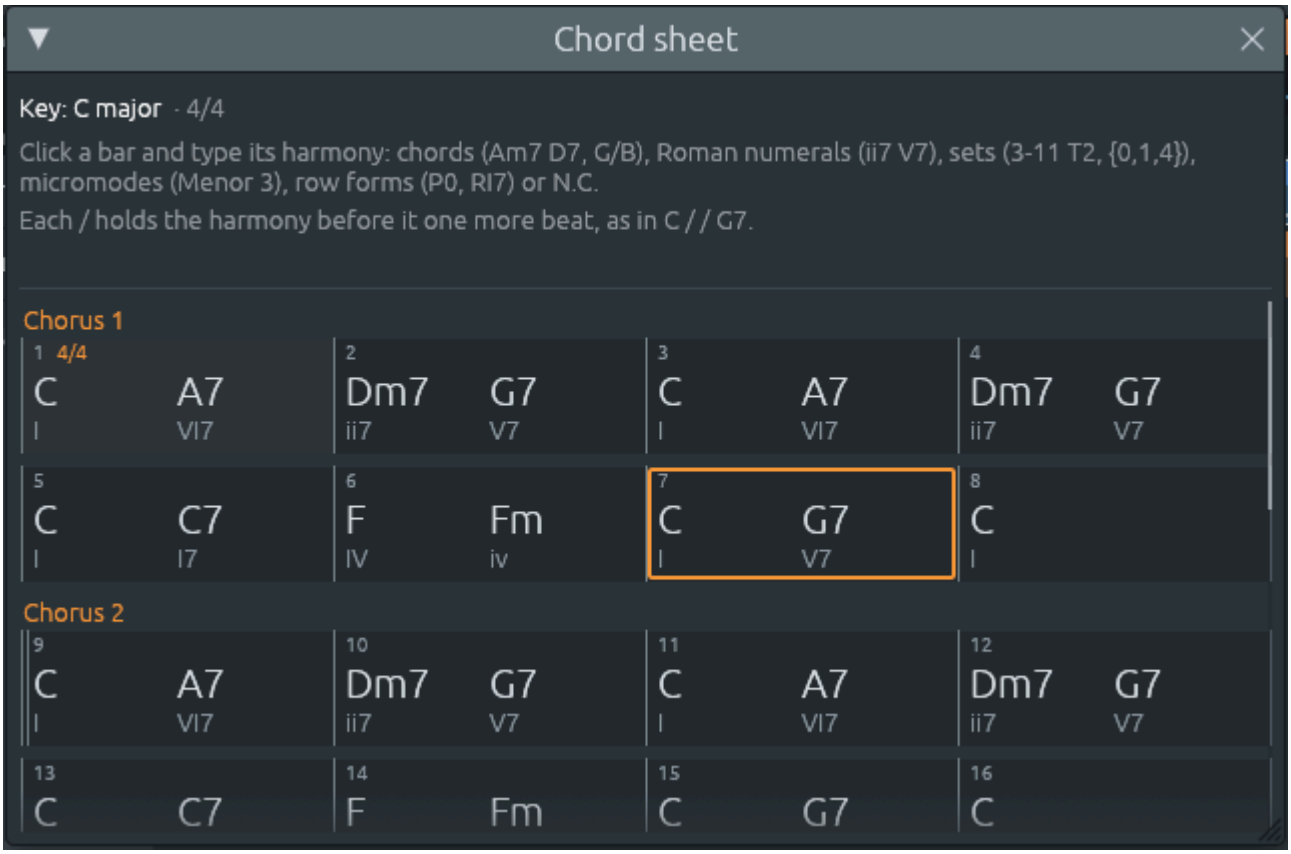
`trying_row_and_micromode_styles_rewrites_one_pattern_while_it_is_unchanged` comprueba que las pruebas caen en un único patrón y que cada una es un solo paso de deshacer.

Los ejercicios sobre la biblioteca están en Ejercicios: la banda.

La hoja de acordes: modelo, distribución y texto

Archivos: `crates/lineup-core/src/chord_sheet.rs`, `crates/lineup-core/src/chord_sheet/text.rs`, `crates/lineup-core/src/bars.rs`, `crates/lineup-app/src/chord_sheet.rs` y `crates/lineup-app/src/chord_sheet/layout.rs`

Una banda de acompañamiento necesita conocer la canción, y los músicos anotan una canción como una cuadrícula de compases con nombres de acordes dentro: «C», Tab, «Am7 D7», Tab. Esa cuadrícula es la **hoja de acordes**. Se abre con **F5**, desde la barra de herramientas o desde **Ver > Hoja de acordes**, como una ventana sobre la lista de reproducción, o en el lugar de la lista de reproducción, en la ventana principal. La banda de los capítulos siguientes toca a partir de ella, y las vistas de armonía de la Parte III nombran lo que hay en ella.



Este capítulo sigue la hoja desde lo que escribes en un compás hasta cómo se guarda, de dónde vienen sus acordes, cómo sigue a la canción cuando se mueven, insertan o eliminan compases, cómo la ventana distribuye los compases, y cómo una hoja se convierte en texto plano y vuelve. Cómo se usa está en el capítulo del manual La hoja de acordes.

Qué cabe en un compás

Haz clic en un compás y escribe. Un compás admite cualquiera de estas cosas, separadas por espacios:

Se escribe	Qué es	Ejemplo
Un símbolo de acorde	cualquier acorde que lea Acordes	`Am7`, `G/B`, `F#m7b5`, `C6/9`
Un poliacorde	dos acordes a la vez, el de arriba primero, unidos por `|`	`Eb7|Fb`, `C|F#`
Un número romano	un acorde de la tonalidad, en una tonalidad de siete notas	`ii7 V7` en Do mayor se escribe Dm7 G7
Un conjunto de clases de altura	por su nombre de Forte, movido con `Tn` o `TnI`	`3-11 T2` es Re Fa La
	o por sus notas, entre llaves	`{0,1,4}`, `{D F A}`, `{t e 0}`
Un micromodo	uno de los doce de Kröpfl, movido igual	`Menor 3`, `Menor 3 T4I`
Una forma de serie	una forma de la serie dodecafónica de la tonalidad	`P0`, `I5`, `R3`, `RI7`
Sin acorde	nada toca la armonía	`N.C.` o `NC`
Una prolongación	la armonía anterior sigue un pulso más	`/` o `.`
Nada	el compás sigue con la armonía anterior	
`%`	el compás anterior, otra vez	`

Una barra después de un acorde indica una nota en el bajo, así que las dos capas de un poliacorde se unen con una barra vertical: `Eb7|Fb` es Mi♭7 sobre Fa♭ mayor, el acorde con que empiezan los Augurios primaverales. Los acordes también se pueden escribir en solfeo, «Lam7», nombre como nombre la aplicación las notas (ver Escribir un acorde de las dos maneras), y una armonía puede llevar las marcas de un golpe o de una anticipación, «C.» o «^C» (ver Golpes, sostenidos, silencios y anticipaciones).

Lo que escribes se anota de una sola manera. Un número romano se convierte en el acorde que es en la tonalidad, así que «V7» escrito en Si♭ mayor se anota F7; un acorde se deletrea como lo deletrean las teclas de armonía, «CM7» pasa a ser Cmaj7; un conjunto escrito por sus notas pasa a su nombre de Forte, `{D F A}` es 3-11 T2. En

una tonalidad menor, un número romano que escribes se lee como la hoja escribe los números, así que «VII» en La menor es Sol mayor, como lo escribe el ajuste de Tonalidades menores: bIII o III. Los cifrados de la biblioteca se leen de la otra manera, desde la escala mayor, porque así están escritos; `read_bar` y `read_typed_bar` solo se diferencian en eso.

Dónde caen las armonías

Las partituras guía tienen una convención para varios acordes en un compás, y la hoja la sigue:

Escrito en 4/4	Pulsos
`Am7`	Am7 en el 1, todo el compás
`Am7 D7`	Am7 en el 1, D7 en el 3: dos pulsos cada uno
`C F G`	C en el 1, F en el 3, G en el 4: dos, uno y uno
`C Am Dm G7`	uno en cada pulso
`C / / G7`	C en el 1, G7 en el 4
`/ G7`	el acorde anterior sigue un pulso, G7 en el 2

En 3/4, dos acordes ocupan dos pulsos y uno: «C G7» pone G7 en el pulso 3. La regla del reparto parejo es una línea: la armonía número `k` de `n` en un compás de `beats` pulsos va en el pulso `(k × beats) / n`, redondeado hacia arriba.

```
let beat = if holds {
    place
} else {
    (harmony_count * beats.len()).div_ceil(harmonies)
};
```

`div_ceil` es la división entera redondeada hacia arriba, así que para tres armonías en cuatro pulsos los pulsos son 0, ⌈4/3⌉ = 2 y ⌈8/3⌉ = 3. Cuando el compás tiene algún `/`, cada palabra es un pulso (`place` es la posición de la palabra), así que puedes poner un acorde en el pulso que quieras. Un compás con más armonías que pulsos se rechaza: «este compás tiene 3 pulsos, y tiene 4 armonías escritas.»

Nombres de varias palabras

«Menor 3 T2» es una armonía escrita en tres palabras, y «C Am» son dos armonías en dos palabras, así que un compás no se puede partir por los espacios y leer cada palabra por separado. En cambio, en cada palabra el lector prueba primero el tramo

de palabras más largo, hasta cuatro, y se queda con el primero que se lee como una armonía:

```
let holds_at = words[at..]
    .iter()
    .position(|word| matches!(*word, "/" | "."))
    .map_or(words.len(), |offset| at + offset);
let longest = (at + 1..=holds_at.min(at + MOST_WORDS))
    .rev()
    .find_map(|end| {
        read_marked(&words[at..end].join(" "), key, numerals)
            .ok()
            .map(|marked| (marked, end))
    });
```

`(a..=b).rev()` cuenta hacia atrás de `b` a `a`, y `find_map` se detiene en el primer `Some`. Así, «Menor 3 T2 C» lee «Menor 3 T2 C» (no es una armonía), después «Menor 3 T2» (un micromodo), y sigue desde «C». Un tramo nunca pasa de un `/` suelto (`holds_at` es donde está el siguiente), porque esa barra es un pulso: «A E / B» es La, Mi sostenido dos pulsos y después Si, no La y el acorde con bajo E/B.

Si una palabra no es ninguna armonía, el compás queda abierto con el cursor al final, y el motivo se escribe sobre la hoja en rojo:

> Compás 3: «Hm7» no es un acorde (Am7, G/B), dos acordes a la vez (Eb7|Fb), un conjunto (3-11 T2, {0,4,7}), un micromodo (Menor 3), una forma de serie (P0, RI7) ni N.C.

No se escribe nada en la canción hasta que se lee el compás entero.

Volver a escribir un compás

Al hacer clic en un compás, su campo de texto se llena con lo que tiene, así que `write_bar(markers, bar)` tiene que dar un texto que `read_bar` vuelva a leer como los mismos marcadores. Escribe solo los nombres cuando caen donde los pondría la regla del reparto, «Am7 D7», y si no, una palabra por pulso, con `/` en los pulsos que siguen, «C / / G7». Un `/` al final no dice nada y se quita, salvo cuando es el último que queda:

```
while words.last().is_some_and(|word| word == "/")
    && words[..words.len() - 1].iter().any(|word| word == "/")
{
    words.pop();
}
```

«C G7 C /» pone tres acordes en los tres primeros pulsos. Sin su barra, «C G7 C» serían tres acordes repartidos por el compás, en los pulsos 1, 3 y 4.

Cómo se guarda la hoja

La hoja no es una cuadrícula de compases en memoria. Es una lista de **marcadores de acorde**, cada uno diciendo «desde este paso, la armonía es esta», como los marcadores de compás, de tempo y de tonalidad:

```
pub struct ChordMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub harmony: HarmonyName,
    /// Whether the band plays it early.
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub push: Option<Push>,
    /// Whether the band stops playing its style here.
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub stop: Option<Stop>,
}
```

La armonía es una de las seis clases que admite un compás:

```
pub enum HarmonyName {
    /// A chord symbol: "Am7", "G/B".
    Chord(ChordSymbol),
    /// Two chords at once, one over the other, the upper one first:
    /// "Eb7|Fb" is E♭7 over F♭ major, the chord that opens the Augurs of
    /// Spring. Lead sheets draw the two chords as a fraction, with a
    /// line between them; a slash would mean a bass note.
    Polychord(ChordSymbol, ChordSymbol),
    /// No chord, written "N.C.": nothing plays the harmony here.
    NoChord,
    /// A pitch-class set on these very notes, named by its Forte number
    /// and how it is moved from the prime form: "3-11 T2" is D F A.
    Set(PitchSet),
    /// One of Kröpfl's micromodes on these three notes: "Menor 3 T2".
    Micromode(PitchSet),
    /// A form of the twelve-tone row of the key here: "RI7".
    Row(RowLabel),
}
```

Una armonía dura hasta el siguiente marcador, como un acorde en una partitura guía dura hasta que se escribe el siguiente. Por eso un compás sin nada no necesita guardar nada, y por eso «la armonía en el paso 200» es una búsqueda del último marcador en el 200 o antes:

```
pub fn chord_at(&self, step: usize) -> Option<ChordMarker> {
    let markers = &self.playlist.chord_markers;
    let after = markers.partition_point(|marker| marker.start <= step);
    after.checked_sub(1).map(|index| markers[index])
}
```

`partition_point` hace una búsqueda binaria en la lista ordenada y devuelve cuántos marcadores empiezan en `step` o antes; el anterior a ese índice es la respuesta. `checked_sub(1)` es `None` cuando no hay ninguno, antes del primer acorde.

Todo cambio de la hoja pasa por un único método, que escribe las armonías de un tramo de la canción, normalmente un compás:

```
pub fn set_chords(&mut self, start: usize, end: usize, markers: &
[ChordMarker]) -> bool {
```

Los marcadores que ya había entre `start` y `end` se van, y `markers` ocupan su lugar. Escribir un compás entero de una vez, en lugar de añadir o mover marcadores sueltos, coincide con lo que haces en la hoja: escribes un compás, y lo que escribiste es lo que tiene el compás. Después se ordena la lista:

```
pub(crate) fn tidy(markers: &mut Vec<ChordMarker>, end: usize) {
    markers.retain(|marker| marker.start < end);
    markers.reverse();
    markers.sort_by_key(|marker| marker.start);
    markers.dedup_by_key(|marker| marker.start);
}
```

Dos marcadores en un mismo paso: debe ganar el último escrito. `dedup_by_key` se queda con el *primero* de cada serie de claves iguales, así que la lista se invierte antes de ordenarla. `sort_by_key` es un orden **estable**, que mantiene los elementos iguales en el orden en que estaban, así que después de invertir, el más nuevo de cada paso queda primero y sobrevive. Deshacer no necesita nada nuevo: el `edit` de la aplicación guarda una copia del proyecto anterior a cada cambio (ver Deshacer y rehacer).

Guardar marcadores en la línea de tiempo, y no una lista de compases, permite que la hoja comparta el código que mueve los otros marcadores: cuando una herramienta desplaza todo lo que hay después de un paso, los marcadores de acorde se desplazan en el mismo bucle que los de compás, tempo y tonalidad. En qué compás cae un marcador se calcula al dibujarlo.

En el archivo

La armonía se guarda como el texto que escribirías, no como números:

```
chord_markers: [
    (start: 0, harmony: "C"),
    (start: 8, harmony: "A7"),
    (start: 16, harmony: "Dm7"),
    (start: 64, harmony: "Menor 3 T2I"),
],
```

Eso requiere dos atributos en el enum:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
#[serde(into = "String", try_from = "String")]
```

`into = "String"` le dice a serde que guarde un `HarmonyName` convirtiéndolo primero en un `String`, con `impl From<HarmonyName> for String`, y `try_from = "String"` que cargue uno leyendo un `String` y convirtiéndolo de vuelta con `impl TryFrom<String> for HarmonyName`. `try_from` y no `from`, porque leer puede fallar: un archivo con una armonía que no se puede leer se rechaza con el motivo, como cualquier archivo roto. Las dos conversiones trabajan en letras, nombre como

nombre la aplicación las notas, para que un archivo sea el mismo para todos (ver Siempre en letras y en inglés):

```
impl From<HarmonyName> for String {
    fn from(harmony: HarmonyName) -> Self {
        NoteNaming::Letters.with(|| harmony.to_string())
    }
}
```

Un proyecto con acordes en la hoja se guarda con el formato 12 o posterior, y uno con un poliacorde con el formato 25 (ver El número de versión).

De dónde vienen los acordes

Una canción se abre con sus acordes ya en la hoja, así que la hoja nunca está en blanco para una canción que toca acordes.

Una **demostración hecha a partir de un cifrado** escribe su propio cifrado, parte por parte, cada parte en su tonalidad. La tercera parte de El Choclo, en La mayor, dice A E7 | E7 A | F#7 Bm | B7 E7: el VI7 ii | II7 V7 del cifrado nombrado en La mayor.

```
fn write_chart(project: &mut Project, at: usize, spans: &[ChordSpan], key:
Scale) {
    let Some(end) = spans.last().map(|span| at + span.start + span.length)
else {
        return;
    };
    let markers: Vec<ChordMarker> = spans
        .iter()
        .map(|span| {
            ChordMarker::new(
                at + span.start,
                HarmonyName::Chord(span.chord.symbol_in(key)),
            )
            .pushed(span.push)
            .stopped(span.stop)
        })
        .collect();
    project.set_chords(at, end, &markers);
}
```

`let ... else` toma el final del último tramo, o vuelve enseguida cuando el cifrado está vacío (`spans.last()` es `None`). `symbol_in(key)` es la misma función que usa el menú Progresión para nombrar «V7» como A7 en Re menor, así que la hoja deletrea los acordes de una demostración exactamente como los lee la banda. `pushed` y

`stopped` toman `self` por valor y devuelven una copia cambiada, así que se encadenan.

Cualquier otra canción, un archivo guardado sin acordes, un archivo MIDI o una grabación importados, recibe los acordes que la aplicación oye en ella, al abrirse y mientras su hoja esté vacía. El botón **Escribir los acordes que toca la canción** lo vuelve a hacer cuando se lo pides, en lugar de lo que dice la hoja; con una banda tocando a partir de una hoja que tiene acordes, el botón se oculta, ya que las notas de la banda salen de la hoja. Cómo se oye la canción, un compás cortado en tramos con programación dinámica y cada tramo nombrado entero, está en Cortar un compás en acordes. Dos reglas hacen que el resultado se lea como un cifrado y no como un análisis. La nota después de la barra es la nota más grave de toda la banda, la línea de bajo incluida, así que unos acordes de piano dispuestos Do Mi La sobre un bajo que toca La son Am, no Am/C. Y una sección empieza con su acorde escrito de nuevo, como un cifrado vuelve a escribir el acorde al comienzo de cada parte:

```
let section_starts = self.playlist.sections.iter().any(|s| s.start ==
start);
if section_starts || markers.last().is_none_or(|last| last.harmony !=
harmony) {
    markers.push(ChordMarker::new(start, harmony));
}
```

`is_none_or` es verdadero para una lista vacía, así que la primera armonía siempre se escribe.

Seguir a la canción

Como los marcadores están en la línea de tiempo de la lista de reproducción, las herramientas que mueven cosas en ella también los mueven a ellos:

- **Copiar una sección**, **escribir sus repeticiones** o **poner una introducción delante** hacen sitio con `Project::make_room`, que desplaza cada clip, sección y marcador desde un paso en adelante, los marcadores de acorde con el resto. Cada copia de una sección recibe sus acordes, empezando por aquel con el que abre la sección, aunque ese acorde se haya escrito en un compás anterior.
- **Pasar la canción a otro compás** mueve cada marcador con su compás y lo deja en el pulso más cercano del compás nuevo, ya que un acorde cambia en un pulso (ver Pasar una canción a otro compás).
- **Cambiar la tonalidad** mueve la hoja como se mueven las notas, así que sigue nombrando lo que suena.

Un cambio de tonalidad es un método de `HarmonyName`:

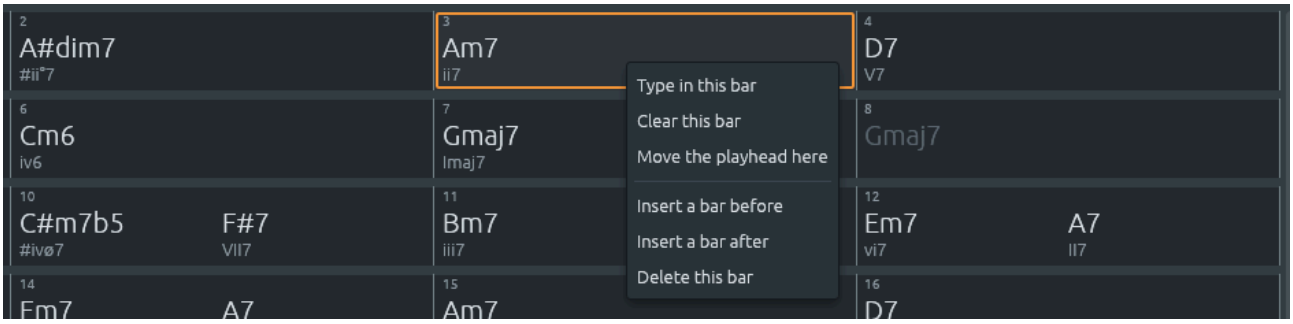
```
match self {
    HarmonyName::Chord(symbol) if from.kind == to.kind => {
        HarmonyName::Chord(symbol.transposed(from.spell(from.root),
to.spell(to.root)))
    }
    HarmonyName::Chord(symbol) => HarmonyName::Chord(chord_moved(symbol,
from, to)),
    // ...
    HarmonyName::NoChord => HarmonyName::NoChord,
    HarmonyName::Set(set) => HarmonyName::Set(set.transpose(up)),
    HarmonyName::Micromode(set) =>
HarmonyName::Micromode(set.transpose(up)),
    HarmonyName::Row(label) => {
        HarmonyName::Row(RowLabel::new(label.form, label.number + up))
    }
}
```

El primer brazo tiene una **guarda de match**, `if from.kind == to.kind`: acepta un acorde solo cuando el modo no cambia. Entonces el acorde se mueve por letras, como en Llevar una hoja de acordes a otra tonalidad: G7/B en Do mayor es A7/C# en Re mayor. Cuando cambia el modo, el segundo brazo lleva cada nota del acorde al mismo grado de la nueva escala, como se mueven las notas de los patrones: el acorde de Do mayor en Do mayor pasa a ser Do menor en Do menor, y G7 pasa a ser Gm7, porque su Si baja a Si♭. Los conjuntos y los micromodos no tienen grados, así que conservan su forma y solo se mueven con la fundamental; una forma de serie se mueve con la serie.

Insertar y eliminar compases

Un cifrado que estás escribiendo rara vez sale bien a la primera. Al puente le falta un compás antes de la vuelta, o una transcripción tiene un compás de más. Haz clic derecho en un compás de la hoja, o en un número de compás de la regla de la lista de reproducción, para **Insertar un compás antes**, **Insertar un compás después** y **Eliminar este compás** (o **Eliminar los compases 5–8** cuando hay varios seleccionados). Cada uno es un solo paso de deshacer, y toda la canción después de ese lugar se mueve:

```
before:  | C    | F    | G7   | C    |
insert a bar after bar 2:
after:   | C    | F    | %    | G7   | C    |
```



Qué es un compás insertado

Un compás nuevo está **vacío**, y vacío quiere decir «aquí no cambia nada»: la armonía que venía antes sigue a través de él, y también la tonalidad, el tempo y la variación de la banda. Eso es lo que significa también un compás vacío en un cifrado, y por eso el menú no pregunta qué poner en él.

Está en el **compás** (la métrica) del compás junto al que se inserta, y en la **sección** de ese compás. Un compás insertado antes del primer compás de una sección de vals es un compás de vals, aunque el anterior esté en 4/4: estabas mirando el compás de vals cuando pediste otro como él. Todo lo que hay después se mueve un compás más tarde: clips, secciones, el bucle, los marcadores de compás, tempo, tonalidad y acordes, y las variaciones y los estilos por sección de la banda.

Qué se lleva un compás eliminado

Un compás eliminado se lleva sus notas y sus acordes, y todo lo que hay después se mueve un compás antes. La pregunta es qué pasa con **lo que sonaba al final de él**. Toma C, F, G7, C con la tonalidad cambiando a Sol mayor en el compás 3 y el tempo a 90 en el compás 4, y elimina el compás 3. Los marcadores del compás 3 se van con él, pero la tonalidad de Sol mayor debía seguir después, y lo sigue debiendo. Así que, antes de mover nada, `close_bars` lee lo que sonaba al final de los compases eliminados:

```
// What goes on at the end of the bars goes on from their start.
let time = meter.time_at(end);
let tempo = self.tempo_map().tempo_at(end);
let key = self.key_at(end);
```

y después vuelve a poner cada cosa donde estaban los compases. Los marcadores que dicen lo mismo que el anterior se quitan después, así que eliminar un compás en medio de un tramo largo en una sola tonalidad no deja una marca de tonalidad de más.

La armonía es la delicada. Si un G7 empieza en un compás eliminado y se sostiene en el siguiente, el G7 debe seguir allí. Pero si el compás anterior al eliminado ya es un G7, escribir otro G7 sería un segundo marcador que dice lo mismo. El código comprueba las dos cosas:

```
if let Some(marker) = going_on
    && self.playlist.chord_markers.iter().all(|m| m.start != start)
    && start
        .checked_sub(1)
        .and_then(|step| self.chord_at(step))
        .is_none_or(|before| before.harmony != marker.harmony)
{
    self.playlist
        .chord_markers
        .push(ChordMarker::new(start, marker.harmony));
}
```

Esto es una **cadena de let** (*let chain*): `if let` y condiciones simples unidas con `&&`, estables desde Rust 1.88 con la edición 2024. Cada parte puede usar lo que ligó un `let` anterior: `marker` se usa en la última línea de la condición. Sin cadenas de let esto necesita un `if let` con un `if` anidado dentro. `start.checked_sub(1)` es `None` en el primer compás, donde no hay «antes»; `and_then` deja pasar el `None` y si no busca el acorde un paso antes; e `is_none_or` dice que sí cuando no hay acorde antes, o cuando el que hay es distinto.

Mover cada paso: un closure

Insertar usa `make_room`, el mismo método que copiar una sección. Eliminar necesita lo contrario, y un caso más: lo que estaba **dentro** de los pasos eliminados. Un closure dice adónde va cualquier paso de la canción:

```
// Where a step of the song goes: the deleted bars close up.
let place = |step: usize| match step {
    step if step <= start => step,
    step if step >= end => step - steps,
    _ => start,
};
```

Los pasos anteriores a los compases eliminados se quedan; los posteriores se mueven `steps` antes; los de dentro caen donde estaban los compases. Después se aplica a todo lo que tiene un lugar en la canción. El final de una sección también pasa por `place`, así que una sección que contenía los compases se acorta, y una hecha solo de ellos queda con largo 0 y se quita. El bucle se conserva solo si queda algo de él:

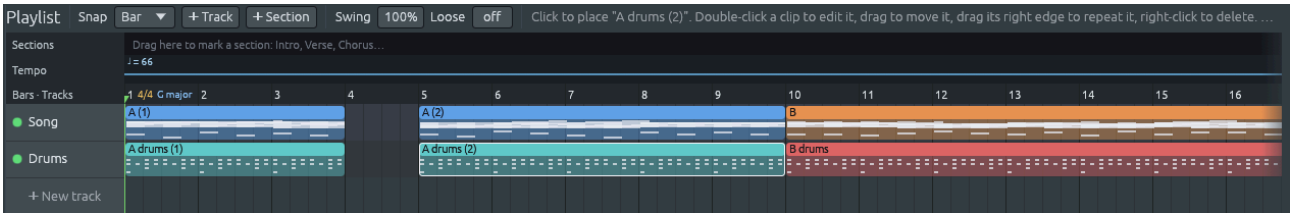
```
playlist.loop_region = playlist.loop_region.and_then(|region| {
    let (from, to) = (place(region.start), place(region.end));
    (from < to).then_some(crate::LoopRegion::new(from, to))
});
```

`bool::then_some` convierte `true` en `Some(value)` y `false` en `None`: aquí, «el bucle si todavía tiene largo».

Las notas: abrir el patrón o cortar el clip

Un clip toca un **patrón**, y las notas están en el patrón, no en el clip. Insertar un compás en medio de un clip tiene que hacer sitio en las notas que toca. Hay dos casos:

- **El patrón lo toca solo este clip, una vez.** Entonces el propio patrón puede cambiar: recibe los pasos vacíos o pierde los eliminados, y el clip toca el patrón más largo o más corto.
- **El patrón se toca también en otro lugar**, en otro clip o en este mismo repetido. Entonces el patrón no debe cambiar, o los otros lugares cambiarían con él. El clip se **corta en dos** donde entra o sale el compás, y cada mitad toca un patrón propio: «Riff (1)» y «Riff (2)».




```
/// Whether clip `index` plays its pattern once and no other clip
/// plays it, so the pattern can change with it.
fn only_clip(&self, index: usize) -> bool {
    let clip = self.playlist.clips[index];
    clip.repeats == 1
        && self
            .playlist
            .clips
            .iter()
            .enumerate()
            .all(|(other, playing)| other == index || playing.pattern !=
clip.pattern)
}
```

`enumerate` empareja cada clip con su índice, así que `all` puede saltarse el propio clip (`other == index`) y preguntar por cada uno de los demás. Un corte que cae entre dos repeticiones de un clip no necesita ningún patrón nuevo: un clip que toca cuatro

veces un ritmo de un compás, al eliminarse su segundo compás, pasa a ser un clip que lo toca una vez y otro que lo toca dos.

Las notas que se sostienen a través del lugar se **cortan** allí, y las de después se mueven:

```
/// `notes` with `steps` empty steps put in at step `at`: the notes from
/// there on move later, and a note held across it ends there.
fn opened(notes: &[NoteEvent], at: usize, steps: usize) -> Vec<NoteEvent>
{
    let at_tick = at * TICKS_PER_STEP;
    notes
        .iter()
        .map(|note| match note.start_tick() < at_tick {
            true => cut_short(note, at_tick),
            false => NoteEvent {
                start: note.start + steps,
                ..*note
            },
        })
        .collect()
}
```

Una nota sostenida sobre el compás nuevo se corta donde empieza el compás, porque el compás nuevo debe estar vacío: una nota de antes sonando debajo de lo que escribas allí sería una sorpresa. `NoteEvent { start: …, ..*note }` es la sintaxis de actualización de structs, una copia de `*note` con `start` cambiado; funciona con `*note` porque `NoteEvent` es `Copy`. Las mitades de un patrón cortado conservan su color, y una mitad cortada solo para eliminarla se quita otra vez al final, así que eliminar no deja patrones vacíos.

Todo o nada

Dos cosas pueden hacer imposible insertar o eliminar: la canción pasaría de su largo máximo, o no hay sitio para los patrones que necesita un corte (una canción tiene como mucho 999). Cualquiera de las dos se descubre a mitad de camino, cuando algunos clips ya cambiaron. Así que los dos empiezan guardando una copia del proyecto:

```
let before = self.clone();
if !self.open_clips(at, steps) || !self.make_room(at, steps) {
    *self = before;
    return false;
}
```

y la vuelven a poner si algo falla. `*self = before` reemplaza el proyecto entero, moviendo `before` a su lugar. La aplicación dice entonces «No hay lugar para otro compás en la canción.» o «No hay lugar para los patrones que necesita el corte.», y nada cambió. `||` se detiene en el primer `true`, así que `make_room` solo se ejecuta si `open_clips` salió bien.

La banda sigue

Las partes de la banda se escriben a partir de la hoja de acordes. Mover todo un compás mueve también sus notas, y la aplicación no debe confundir eso con una edición tuya: una parte que editas queda **congelada**, como la dejaste, y deja de seguir la hoja. Así que los dos métodos se ejecutan dentro de `moving_the_band`, que también usan las herramientas que transponen la canción entera o le cambian el compás:

```
pub fn insert_bars(&mut self, bar: usize, after: bool, count: usize) ->
bool {
    self.moving_the_band(|project| project.open_bars(bar, after, count))
}
```

`moving_the_band` recibe el closure como `impl FnOnce(&mut Project) -> R`, lo ejecuta sobre el proyecto y después toma las partes movidas como propias de la banda (ver Cómo toca la banda una hoja de acordes).

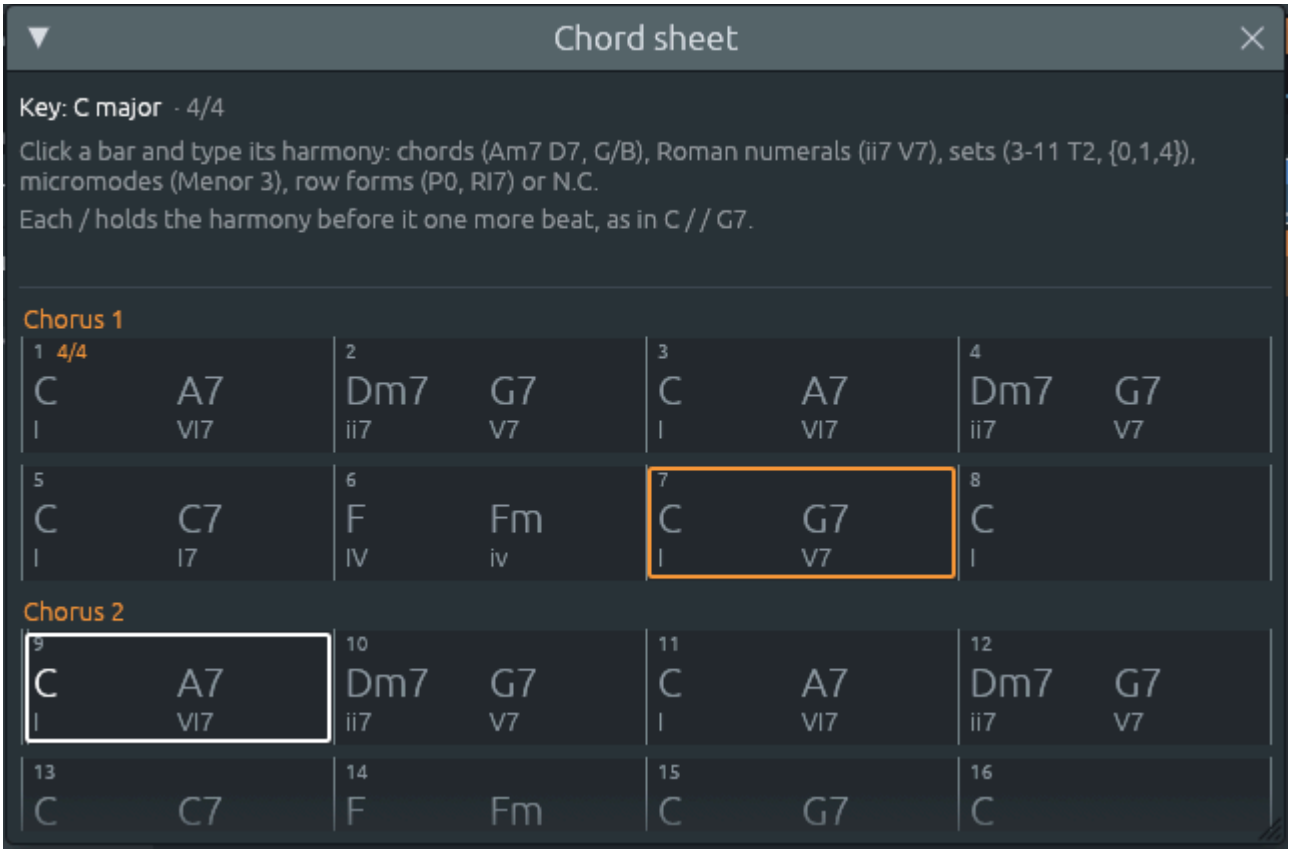
Los dos menús, el de la hoja y el de la regla, son una única función, `bar_edits`, así que siempre ofrecen las mismas opciones con las mismas pistas. Devuelve lo que se pulsó como un valor, un `BarEdit`; la hoja lo envuelve en su propia acción y la lista de reproducción en la suya, y la aplicación aplica cualquiera de las dos con `edit_bars`, que además selecciona el compás nuevo en la hoja para que puedas escribir en él enseguida.

La ventana

La ventana está en `lineup-app`, y casi todo es dibujo. Sus filas de arriba tienen los controles de la banda, la tonalidad, el menú **Mostrar** que nombra la armonía como acordes, conjuntos o micromodos (ver Una armonía, cuatro nombres), la forma de la canción y las herramientas de práctica; los capítulos de cada una las explican. Debajo vienen los compases.

Cada sección empieza una línea nueva bajo su nombre, «Estrofa ×3» para una sección que se toca tres veces. Una doble barra marca dónde empieza una sección, y la barra final (fina y después gruesa) dónde termina la canción. Cada armonía está donde cae su pulso, con su número romano debajo en la tonalidad, sus notas debajo si es un conjunto o un micromodo y, para un acorde de fuera de la tonalidad, lo que hace allí («V7/ii», «iv · prestado»; ver Qué hace un acorde). Una inversión lleva las cifras del bajo cifrado: G7/B es V^6_5. Un compás sin nada escrito muestra, tenue, la armonía que sigue, para que no parezca vacío. Un cambio de tonalidad se escribe arriba a la derecha de su compás, «→ Sol mayor». Al señalar una armonía se dice qué es: «La7: La séptima de dominante, V7/ii en Do mayor … Sus notas: La Do# Mi Sol.» Mantener pulsado el botón del ratón sobre ella la toca, con su nota del bajo debajo.

Mientras suena la canción, el compás que suena lleva un marco blanco y el acorde que suena es el nombre más brillante de la hoja; los demás se atenúan. La hoja se desplaza junto con la canción.



Un rectángulo por compás

La hoja no está hecha de botones. La ventana toma el espacio que necesitan todas las líneas, después calcula un `Rect` para cada compás (una `Cell`, con su línea y su columna) y le pregunta a egui por cada uno:

```
let id = ui.scope_id().with(("chord-sheet-bar", bar.number));
let response = ui.interact(rect, id, Sense::click_and_drag());
```

`ui.interact` convierte cualquier rectángulo en algo sobre lo que se puede pasar el ratón, hacer clic o arrastrar, y devuelve el mismo `Response` que devolvería un botón. El id tiene que ser el mismo de un frame al siguiente, porque egui recuerda por id sobre qué widget se pulsó el ratón; el número de compás lo asegura. Los nombres, los textos de debajo y las barras se pintan después con `painter.text`, `painter.vline` y `painter.rect_stroke`.

Un campo de texto sobre un compás

Se escribe en un `TextEdit` corriente, puesto sobre el compás con `ui.put(rect, field)`. Intro toma el texto y selecciona el compás siguiente, Tab lo toma y sigue escribiendo en el compás siguiente, Mayús+Tab vuelve uno atrás, y Escape deja el compás como estaba. Tres cosas requirieron cuidado.

La primera letra. Seleccionas un compás y escribes «A»: el campo se abre en ese frame, así que la hoja ya leyó la «A». El campo se abre con la «A» dentro y el cursor detrás, escribiendo el cursor en el estado guardado del campo antes de dibujarlo. Intro o F2 abren un compás con lo que tiene, todo seleccionado, para que escribir lo reemplace:

```
let mut state = egui::text_edit::TextEditState::load(ui.ctx(),
id).unwrap_or_default();
let range = match fresh {
    Fresh::SelectAll =>
egui::text::CCursorRange::select_all_str(&editing.text),
    Fresh::CursorAtEnd =>
egui::text::CCursorRange::one(egui::text::CCursor::new(
        editing.text.chars().count(),
    )),
};
state.cursor.set_char_range(Some(range));
state.store(ui.ctx(), id);
ui.memory_mut(|memory| memory.request_focus(id));
```

El cursor cuenta caracteres, no bytes: un «♯» escrito en un compás ocupa tres bytes en UTF-8, y es un solo lugar para el cursor.

Tab y las flechas. egui usa Tab para pasar de un widget a otro, y un campo de texto suelta el teclado con él. En la hoja, Tab quiere decir «compás siguiente», así que el campo saca el Tab de la entrada antes de que egui lo vea:

```
let (tab, back) = if had_focus {
    ui.input_mut(|input| {
        let back = input.consume_key(Modifiers::SHIFT, Key::Tab);
        (input.consume_key(Modifiers::NONE, Key::Tab), back)
    })
} else {
    (false, false)
};
```

y le dice a egui qué teclas se queda la hoja mientras tiene el teclado:

```
const KEEP_KEYS: egui::EventFilter = egui::EventFilter {
    tab: true,
    horizontal_arrows: true,
    vertical_arrows: true,
    escape: false,
};
```

El Intro que cierra el campo sigue «pulsado» en ese frame, y las teclas propias de la hoja leen Intro como «abrir este compás». Sin una protección, Intro cerraría un compás y abriría el siguiente a la vez. La hoja recuerda si se estaba escribiendo en un compás cuando empezó el frame, y solo lee sus teclas si no:

```
let was_editing = self.editing.is_some();
self.edit_field(ui, project, &cells, &keys, sheet.id, actions);
// ...
if has_keyboard && self.editing.is_none() && !was_editing {
    ui.memory_mut(|memory| memory.set_focus_lock_filter(sheet.id,
KEEP_KEYS));
    self.keys(ui, project, &cells, typing_keys, actions);
}
```

Mientras la hoja tiene el teclado, las cifras son parte de lo que escribes, «3-11 T2» o «C6/9», así que las teclas de armonía no las tocan. Con el teclado del equipo encendido (las letras tocan notas), una letra no puede empezar a escribir en un compás; Intro o un doble clic lo abren primero.

Acciones, no cambios

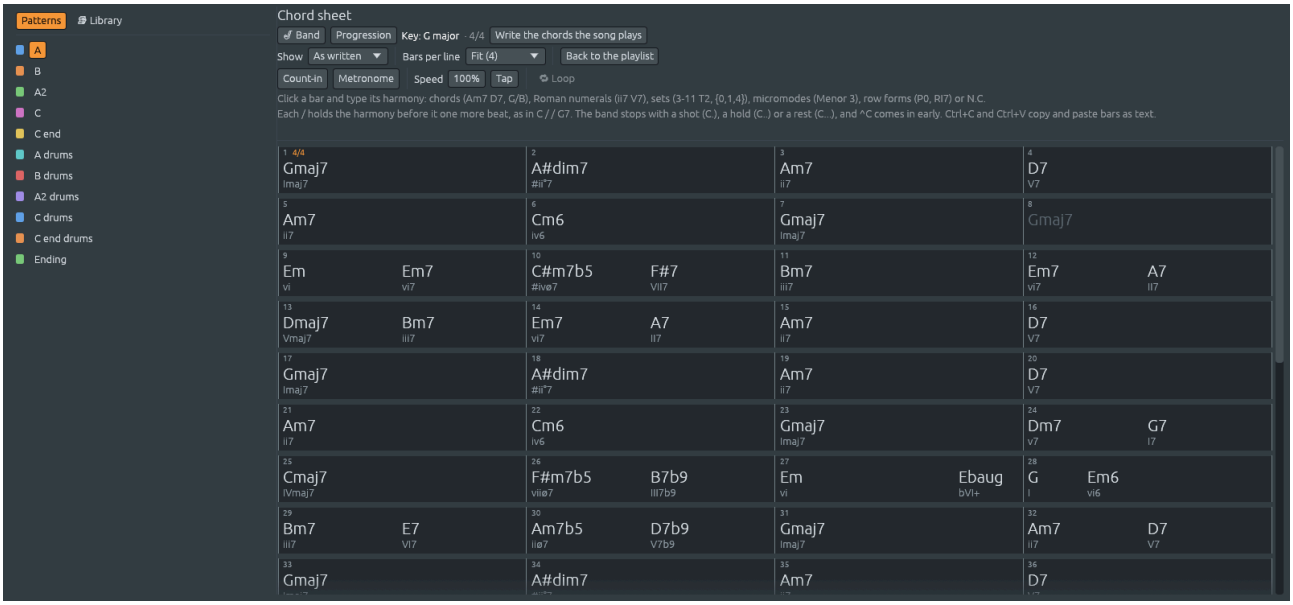
Como los editores de la Parte II, la hoja nunca cambia el proyecto ella misma. Dibuja a partir de `&Project` (un préstamo compartido, de solo lectura) y devuelve una lista de lo que se hizo:

```
pub enum SheetAction {
    // ...
    /// Write the harmonies of the steps from `start` to `end`: one bar.
    SetChords {
        start: usize,
        end: usize,
        markers: Vec<ChordMarker>,
    },
    /// Write the chords the song plays on the empty sheet.
    WriteHeardChords,
    // ...
    /// Insert or delete bars.
    Bars(BarEdit),
    /// Move the playhead to this step of the playlist's timeline.
    Seek(usize),
    /// Play these notes now, as a harmony key does.
    NoteOn(Vec<Note>),
    NoteOff(Vec<Note>),
    // ...
}
```

`LineupApp::apply_sheet` hace después cada una: `SetChords` a través de `edit`, para que se pueda deshacer, `Seek` moviendo el cursor de reproducción, y las notas por el mismo camino que las teclas de armonía. La ventana no puede olvidarse de registrar un paso de deshacer, porque no tiene ninguna manera de cambiar el proyecto; el borrow checker lo rechazaría.

Líneas que se ajustan a los nombres

Un cifrado impreso suele tener cuatro compases por línea. La hoja escribe nombres mucho más largos que «C» y «G7»: poliacordes como «Eb7|Fb», conjuntos como «3-11 T7I», micromodos como «Acorde menor-mayor T7I», cada uno con un número romano o sus notas debajo. Así que el número de compases por línea es una elección, **Compases por línea**, junto a **Mostrar**. Por defecto es **Ajustar a los nombres**: tantos compases como quepan con cada nombre, y el número o las notas de debajo, a tamaño completo. El menú dice cuántos son ahora, como en **Ajustar (4)**. O elige un número del 1 al 16; entonces la línea tiene esos, y los nombres que no caben se escriben más pequeños. ⛶ **Llenar la ventana** lleva la hoja a la ventana principal, al lugar de la lista de reproducción, donde una línea tiene más sitio.



Por qué 1, 2, 4, 8, 12 o 16

Ajustar podría poner cualquier número de compases en una línea: cinco, siete, once. Un cifrado impreso nunca lo hace, y por una razón. La mayoría de las canciones están hechas de **frases de cuatro compases**, y un cifrado pone el primer compás de una frase al comienzo de una línea, así que el ojo encuentra los compases 1, 5, 9 y 13 en la misma columna. Con cinco compases por línea, la segunda frase empezaría en la segunda columna de la segunda línea, la tercera en la tercera: se perdería la forma de la canción. Así que los números que ajusta son los que mantienen las frases juntas: 1 y 2 entran justos en una frase, y 4, 8, 12 y 16 están hechos de frases enteras. Doce es un blues de doce compases en una línea.

```
/// The counts of bars a line holds when they fit, from the most: each
/// divides a four-bar phrase or is made of them.
const FITS: [usize; 6] = [16, 12, 8, 4, 2, 1];

/// No bar is made narrower than this to fit more on a line, however
/// short its names.
const NARROWEST_BAR: f32 = 110.0;
```

`NARROWEST_BAR` está para las canciones de nombres cortos. Una hoja de «C» y «G» pondría si no dieciséis compases en una ventana que muestra cuatro con holgura, cada uno demasiado estrecho para hacer clic o escribir en él.

La elección es un tipo:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default, Serialize,
Deserialize)]
pub enum Columns {
    /// As many as fit with every name at its full size.
    #[default]
    Fit,
    /// This many, from 1 to [`MOST_BARS`]: names that don't fit shrink.
    Bars(usize),
}
```

`#[default]` sobre una variante le dice a `#[derive(Default)]` qué variante es `Columns::default()`. El número sale de un único método:

```
pub fn count(self, width: f32, needed: f32) -> usize {
    match self {
        Columns::Bars(bars) => bars.clamp(1, MOST_BARS),
        Columns::Fit => {
            let bar = needed.max(NARROWEST_BAR);
            FITS.into_iter()
                .find(|&bars| width / bars as f32 >= bar)
                .unwrap_or(1)
        }
    }
}
```

`FITS` va de más compases a menos, así que `find` se detiene en el número más grande cuyos compases siguen siendo lo bastante anchos. Si ni siquiera uno cabe (un nombre muy largo en una ventana muy estrecha), sigue siendo un compás por línea, y sus nombres se escriben más pequeños.

`FITS.into_iter()` recorre el array **por valor**, así que sus elementos son `usize`. `find` le pasa cada elemento a su closure por referencia, como un `&usize`, para que pueda probarlo sin tomarlo y devolverlo si es el que busca. El patrón `&bars` en el parámetro del closure saca el `usize` de la referencia.

`count` es una función pura de dos números, así que su prueba no necesita ninguna ventana:

```
// Short names: the narrowest bar decides.
assert_eq!(Columns::Fit.count(760.0, 60.0), 4);
assert_eq!(Columns::Fit.count(1300.0, 60.0), 8);
// Long names take more room, and whole phrases still start lines:
// room for three bars makes lines of two.
assert_eq!(Columns::Fit.count(760.0, 240.0), 2);
```

A 1300 píxeles, doce compases serían de 108 píxeles cada uno, por debajo del compás más estrecho; ocho son de 162. A 760 píxeles con nombres que necesitan 240,

cabrían tres compases (253 cada uno), pero tres no es un número que mantenga juntas las frases, así que son dos.

Qué tan ancho debe ser un compás

Antes de dibujar, la hoja le pregunta a cada compás qué tan ancho debe ser, y toma el más ancho. Lo que tiene que caber es cada **nombre** y la **primera palabra de debajo**: el número romano de un acorde, o las notas de un conjunto («Do Mi Sol»), que suele ser la más ancha de las dos. Las palabras después del número, como «prestado» en «bVII7 · prestado», se escriben solo donde hay sitio. El color del número ya dice «prestado», y la pista lo dice todo; contar esas palabras haría que un acorde prestado en el último pulso de un compás dejara en la mitad los compases de todas las líneas.

```
fn first_word(caption: &str) -> &str {
    caption.split(" · ").next().unwrap_or(caption)
}
```

`split` da los trozos entre los separadores, uno por vez, y `next` toma el primero. Devuelve un `Option`, porque un iterador puede estar vacío; `split` nunca lo está, pero el tipo no lo sabe, y `unwrap_or` dice qué usar si lo estuviera.

Cada nombre se coloca **en su pulso, o justo después del nombre anterior** cuando ese necesita el sitio. En un compás de 4/4 con «C A7», C empieza en el pulso 1 y A7 en el 3. Pero en un compás donde Dm7/C dura una semicorchea y G7 el resto, G7 se escribe justo después de Dm7/C, un poco tarde, como hace un cifrado impreso cuando los acordes se apiñan. Un nombre nunca empieza antes de su pulso, así que el orden de los nombres es siempre el de los acordes. Un `Row` guarda, para cada nombre de un compás, su pulso (desde 0 al comienzo del compás hacia 1 al final) y su ancho, medido con las fuentes del painter (`layout_no_wrap`) como se dibujarán los nombres:

```
struct Row {
    /// For each name, its beat, from 0 at the start of the bar towards 1
    /// at its end, and its width.
    names: Vec<(f32, f32)>,
}
```

¿Qué tan estrecho puede ser el interior del compás? Como ningún nombre empieza antes de su pulso, **cada nombre, con todos los que vienen después, tiene que caber entre su pulso y el final del compás**:

```
fn packed_width(&self) -> f32 {
    (0..self.names.len())
        .map(|first| {
            let (beat, _) = self.names[first];
            let from = &self.names[first..];
            let widths: f32 = from.iter().map(|(_, width)| width).sum();
            let gaps = NAME_GAP * (from.len() - 1) as f32;
            (widths + gaps) / (1.0 - beat)
        })
        .fold(0.0, f32::max)
}
```

En «C / / A7», A7 empieza en el pulso 4 y tiene la última negra del compás: un A7 de 30 píxeles de ancho necesita un compás de 120. C empieza en el pulso 1 y necesita los dos nombres, con el espacio entre ellos, en el compás entero. El compás necesita el mayor de los dos. Para la semicorchea de Dm7/C, son los dos nombres uno junto al otro, no un compás dieciséis veces más ancho que «Dm7/C». `&self.names[first..]` es un **slice**: una vista del vector desde `first` hasta el final, sin copiarlo. A `sum` hay que decirle qué suma, aquí `f32`, porque muchos tipos se pueden sumar a partir de los mismos elementos.

El mismo `Row` coloca los nombres al dibujar el compás:

```
fn lefts(&self, left: f32, right: f32) -> Vec<f32> {
    let on_beat = |beat: f32| left + (right - left) * beat;
    let mut free = left;
    let packed: Vec<f32> = self
        .names
        .iter()
        .map(|&(beat, width)| {
            let start = on_beat(beat).max(free);
            free = start + width + NAME_GAP;
            start
        })
        .collect();
    if free - NAME_GAP <= right + 0.5 {
        packed
    } else {
        self.names.iter().map(|&(beat, _)| on_beat(beat)).collect()
    }
}
```

El closure que recibe `map` cambia `free`, una variable de fuera, cada vez que se lo llama: es un `FnMut`, que `map` acepta porque lo llama una vez por elemento, en orden. Cuando los nombres no caben todos así, en una línea con un número de compases elegido por ti, cada uno empieza en su pulso y se dibujan más pequeños. Medir y dibujar comparten un `Row`, así que la hoja nunca hace sitio para algo que no dibuja.

El compás más ancho de la canción es un **fold**:

```
.map(|bar| {
    self.needed_width(ui.painter(), project, &meter, &keys, &bar, marks,
&heard)
})
.fold(0.0, f32::max),
```

¿Por qué no `.max()`? `Iterator::max` necesita `Ord`, un orden total, y `f32` no lo tiene: `NaN` no es ni menor ni mayor que ningún número. `f32::max` es una función corriente que toma dos `f32` y devuelve el mayor (ignorando un `NaN`), así que hacer fold con ella desde `0.0` da el compás más ancho, o 0 para una canción sin compases. Pasar `f32::max` por su nombre funciona porque una función es un valor que se puede llamar, como lo sería un closure `|a, b| a.max(b)`. Medir cada compás cuesta un poco en cada frame, pero egui guarda en caché el texto que distribuyó hace poco, así que volver a medir el mismo nombre es una consulta.

Con un número elegido, un compás puede ser demasiado estrecho para sus nombres. Entonces se escriben más pequeños, con tres reglas: todos los nombres de un compás a un mismo tamaño, el que permite el más apiñado, porque un compás que mezcla nombres grandes y pequeños se lee como si los pequeños importaran menos; un nombre que sigue de un compás anterior solo se achica a sí mismo; y un conjunto o un micromodo cuyo nombre no cabe ni al tamaño más pequeño se escribe solo por su clase («Menor 3» por «Menor 3 T8I»), o, donde ni eso cabe, como «…»: un nombre cortado se leería como otro nombre.

Ocupar el lugar de la lista de reproducción

La ventana principal tiene un panel central, donde se dibuja la lista de reproducción. Cuando la hoja llena la ventana, ese panel dibuja la hoja en su lugar, junto a los patrones y la Biblioteca:

```
// The chord sheet can take the playlist's place, beside the
// pattern picker and the library.
self.chord_sheet.remember(ctx);
let sheet_fills = self.chord_sheet.fills_window();
// ...
let playlist = egui::CentralPanel::default_margins().show(ui, |ui| {
    if sheet_fills {
        let actions = self
            .playlist
            .browser(ui, &self.project, self.pattern, tried);
        let sheet = self.chord_sheet.main(
            ui,
            &self.project,
            playing,
            typing,
            (practice, self.playlist.library_browser_mut()),
        );
        (actions, sheet)
    } else {
        let actions = self.playlist.show(
            ui,
            &self.project,
            self.pattern,
            song_position,
            tried,
            playlist_zoom,
        );
        (actions, Vec::new())
    }
});
```

Las dos ramas devuelven un par de listas de acciones: lo que pide la lista de reproducción (o el navegador junto a la hoja), y lo que pide la hoja. El `if` es una expresión, así que las dos ramas tienen que tener el mismo tipo; la rama de la lista de reproducción devuelve un `Vec` vacío para las acciones de la hoja. `ChordSheet::main` y la ventana propia de la hoja dibujan el mismo contenido con el mismo método, `contents`; solo cambia el marco que lo rodea, así que todo lo que hace la hoja en su ventana lo hace en la ventana principal, sin una segunda copia del código.

`remember` se llama **antes** que `fills_window`. Lee las elecciones guardadas por la ejecución anterior, la primera vez que se lo llama. Llamado después, el primer frame dibujaría la lista de reproducción y el segundo la hoja: un destello de la vista equivocada cada vez que arranca la aplicación.

Las teclas de zoom amplían lo que está bajo el puntero (ver ¿Qué hay bajo el puntero?). Con la hoja en el lugar de la lista de reproducción, el área de la lista tiene que contar como nada, o las teclas ampliarían una lista que nadie ve:

```
let playlist = match self.chord_sheet.fills_window() {
    true => egui::Rect::NOTHING,
    false => self.playlist.area(),
};
```

`Rect::NOTHING` no contiene ningún punto, así que «¿está el puntero sobre la lista de reproducción?» es siempre falso, y las teclas cambian el tamaño de la interfaz. La hoja en la ventana principal conserva el sitio que tiene, así que con nombres más pequeños caben más compases por línea. En su propia ventana no: la ventana se achica con todo lo demás, y tiene los mismos compases.

Recordar las elecciones

Las dos elecciones son un struct pequeño, guardado con el estado de la aplicación:

```
#[derive(Debug, Clone, Copy, PartialEq, Default, Serialize, Deserialize)]
#[serde(default)]
struct Choices {
    columns: Columns,
    /// Whether the sheet takes the playlist's place in the main window.
    fills: bool,
}
```

`#[serde(default)]` sobre el struct dice: un campo que falta en lo guardado toma su valor de `Choices::default()`, así que una tercera elección añadida más tarde sigue cargando las dos primeras. `remember` guarda una copia de lo último que guardó, y solo escribe cuando las elecciones difieren de ella:

```
match self.saved {
    None => {
        self.choices = ctx
            .data_mut(|data| data.get_persisted(choices_id()))
            .unwrap_or_default();
        self.saved = Some(self.choices);
    }
    Some(saved) if saved != self.choices => {
        let choices = self.choices;
        ctx.data_mut(|data| data.insert_persisted(choices_id(), choices));
        self.saved = Some(choices);
    }
    Some(_) => {}
}
```

El brazo `None` se ejecuta una vez, en el primer frame. El segundo brazo tiene una guarda de match: acepta solo cuando el patrón coincide y la condición se cumple.

Todo lo demás cae en `Some(_) => {}`, nada que hacer, que es casi todos los frames.

Las flechas caen en la columna correcta

Tenga una línea los compases que tenga, las flechas arriba y abajo van al compás de arriba o de abajo. Nunca cuentan compases: la hoja guarda cada compás que dibujó como una celda con su **línea** y su **columna**, y la flecha abajo busca la celda de la línea siguiente cuya columna está más cerca:

```
let in_line = |line: usize, column: usize| {
    cells
        .iter()
        .filter(|cell| cell.line == line)
        .min_by_key(|cell| cell.column.abs_diff(column))
        .map(|cell| cell.bar.number)
};
```

Las celdas salen de lo que se dibujó, así que están bien con cuatro compases por línea, con dieciséis y en una línea corta al final de una sección. `min_by_key` sirve aquí, a diferencia de `max` más arriba, porque la clave es un `usize`, que es `Ord`.

Hojas de acordes como texto

Los músicos se mandan cifrados como texto plano: en un correo antes de un ensayo, en un foro, en una aplicación de notas. Nadie acordó un formato, pero la mayoría de los cifrados se parecen, porque copian la manera en que se imprime un cifrado:

```
Title: Autumn Song
Key: C major
Time: 4/4
Tempo: 120
Style: Bossa nova

[Verse ×2]
| C      | Am7 | Dm7 G7 | C   |
| F      | G7  | G7     | C   |

[Chorus]
| F      | G   | Em7 A7 | Dm |
```

Archivo > Importar hoja de acordes… abre un archivo así (o uno soltado sobre la ventana) como una canción nueva con banda: la tonalidad, el compás, el tempo y el estilo del cifrado, sus secciones en la lista de reproducción, sus acordes en la hoja. Si

algo no se puede leer, no se importa nada, y la barra de estado dice dónde: «línea 4: compás 3: «Qx7» no es un acorde …». **Archivo > Exportar hoja de acordes…** escribe la hoja de cualquier canción en el mismo formato, con los compases alineados en columnas. En la hoja, **Ctrl+C** copia como texto los compases seleccionados (o la hoja entera, con sus ajustes y secciones, cuando están todos o ninguno seleccionados), **Ctrl+X** los corta, y **Ctrl+V** escribe el texto del portapapeles desde el compás seleccionado. Una línea sin ningún `|` se lee como un compás, así que «Dm7 G7» copiado de un mensaje va directo a un compás. Pegar toma solo los acordes: la canción conserva su tonalidad, su compás, su tempo, su banda y sus secciones.



El formato

Todo es opcional salvo los compases. Las palabras del formato son en inglés, como en el texto que exporta la aplicación.

Línea	Significa
`Title: Autumn Leaves`	El nombre de la canción. Una primera línea que no es otra cosa también es el título, como arriba en un cifrado impreso.
`Key: G major`	La tonalidad: `G`, `Em`, `E minor`, `A Dorian`, `C harmonic minor` y las demás escalas tonales (no las series ni los conjuntos, que se eligen en la aplicación). Una tonalidad

Línea	Significa
	conserva el nombre escrito: `Gb major` sigue siendo Sol♭ mayor.
`Time: 3/4`	El compás, cualquiera que acepte la aplicación: `6/8`, `5/4`, `2+2+3/8`.
`Tempo: 132`	Negras por minuto; `132 BPM` también se lee.
`Style: Bossa nova`	El estilo de la banda, por su nombre.
`[Verse]`	Una sección empieza en el compás siguiente. `[Chorus ×2]`, `[Chorus x2]` o `[Chorus] x2` la toca dos veces.
`| C | Am7 | Dm7 G7 |`	Compases. Una línea puede tener cualquier número de ellos.
`# Preguntarle a Ana por el puente`	Una nota para las personas, que se salta.

Dentro de un compás va exactamente lo que escribes en un compás de la hoja, marcas incluidas. `||`, `|:` y `:|` se leen como barras simples; las repeticiones se escriben con `×2` en una sección, que es como las guarda la lista de reproducción. Una línea `Key:` o `Time:` entre compases cambia la tonalidad o el compás desde el compás siguiente. Los números romanos se leen en la tonalidad del cifrado, así que

```
Key: F major
| I | vi | ii7 V7 | I |
```

da F, Dm, Gm7 C7, F, y el mismo texto con `Key: D major` da D, Bm, Em7 A7, D: una progresión en cualquier tonalidad. Sin línea `Style:`, la banda toca un estilo en el compás del cifrado, y sin compás tampoco, el estilo con que empieza una canción nueva.

Un compás vacío se lee como en la hoja: la armonía anterior sigue a través de él. `%`, el signo de repetición, vuelve a tocar el compás anterior, como lo lee un músico: después de un compás de dos acordes, «| C G | % |» es C G dos veces, y después de un compás de un solo acorde es lo mismo que un compás vacío.

`read_bar` solo ve su propio compás, así que el signo lo lee `read_typed_bar_after`, que recibe además el compás anterior y sus marcadores. `repeat_bar` copia esos marcadores en el compás nuevo, cada uno a los mismos pasos del comienzo, con su golpe, sostenido, silencio o anticipación. Un primer marcador que solo seguiría con la armonía que ya suena, como en «| C | % |», se deja fuera, para que ese compás guarde exactamente lo mismo que uno vacío:

```
pub fn repeat_bar(before: &[ChordMarker], before_bar: &Bar, bar: &Bar) ->
Vec<ChordMarker> {
    let in_bar: Vec<&ChordMarker> = before
        .iter()
        .filter(|marker|
(before_bar.start..before_bar.end()).contains(&marker.start))
        .collect();
    let sounding = in_bar.last().map(|marker| marker.harmony);
    let mut markers: Vec<ChordMarker> = in_bar
        .iter()
        .map(|marker| ChordMarker {
            start: bar.start + (marker.start - before_bar.start),
            ..**marker
        })
        .filter(|marker| marker.start < bar.end())
        .collect();
    if let Some(first) = markers.first()
        && first.start == bar.start
        && Some(first.harmony) == sounding
        && first.push.is_none()
        && first.stop.is_none()
    {
        markers.remove(0);
    }
    markers
}
```

La canción guarda los acordes, no el texto: una vez leído, el compás tiene un C y un G propios, y al hacer clic en él muestra «C G».

Leer el texto

`text::read` convierte el texto en un `TextSheet`, y solo comprueba la disposición. Todavía no puede leer los compases: cuántos pulsos tiene un compás depende del compás que rige en ese punto, que solo la canción conoce cuando las líneas `Time:` anteriores están en su lugar.

```
pub struct TextSheet {
    pub title: Option<String>,
    /// The key the song starts in.
    pub key: Option<Scale>,
    /// The time signature the song starts in.
    pub time: Option<TimeSignature>,
    /// In quarter notes per minute.
    pub tempo: Option<f32>,
    /// The name of the band style to play it in, as written, and its
    /// line.
    pub style: Option<(String, usize)>,
    /// The bars, sections and changes, in order.
    pub items: Vec<Item>,
}

pub enum Item {
    /// A bar, as it is typed in the chord sheet: "Dm7 G7".
    Bar { text: String, line: usize },
    /// A section starts with the next bar.
    Section {
        name: String,
        repeats: usize,
        line: usize,
    },
    /// The key changes from the next bar on.
    Key(Scale),
    /// The time signature changes from the next bar on.
    Time(TimeSignature),
}
```

`Item` es un enum cuyas variantes guardan cosas distintas: dos tienen campos con nombre, como un struct, y dos guardan un valor, como una tupla. Un `match` sobre un `Item` tiene que decir qué hace con cada una de las cuatro, así que cuando se añada una quinta clase de línea, el compilador listará cada lugar que tiene que aprenderla. Los compases y las secciones guardan el número de su línea, para que un problema encontrado después, al leer el compás, todavía pueda nombrarla. Las líneas se numeran con `enumerate`, que cuenta desde 0:

```
for (index, line) in text.lines().enumerate() {
    let number = index + 1;
    let line = line.trim();
    if line.is_empty() || line.starts_with('#') {
        continue;
    }
```

El error es un struct pequeño con la línea y el mensaje, e `impl fmt::Display` dice cómo se imprime, «línea 4: …». Toda función que puede fallar devuelve `Result<_, TextError>`, y `?` pasa el error hacia arriba enseguida. `map_err` convierte otro error en este por el camino:

```
let key = read_key(value).map_err(|message| error(number, message))?;
```

Una línea es un ajuste cuando tiene dos puntos y la palabra de antes es uno de los cinco ajustes, con mayúsculas o minúsculas. `split_once(':')` parte «Key: G major» en los primeros dos puntos, en `("Key", " G major")`, y devuelve `None` cuando no los hay:

```
fn setting(line: &str) -> Option<(String, &str)> {
    let (name, value) = line.split_once(':')?;
    let name = name.trim().to_lowercase();
    ["title", "key", "time", "tempo", "style"]
        .contains(&name.as_str())
        .then_some((name, value))
}
```

Un encabezado de sección puede poner su cuenta dentro de los corchetes o después. `read_times` lee «×2», «x2» o «X 2»:

```
fn read_times(text: &str) -> Option<Option<usize>> {
    let number = text
        .strip_prefix(['×', 'x', 'X'])?
        .trim()
        .parse::<usize>()
        .ok()?;
    Some((1..=MAX_REPEATS).contains(&number).then_some(number))
}
```

`strip_prefix` recibe un array de caracteres y quita el que esté al comienzo del texto. La respuesta es una opción dentro de una opción, porque hay tres casos que distinguir: `None` es «esto no es una cuenta» (el encabezado sigue con otras palabras), `Some(None)` es «una cuenta, pero demasiadas veces» (un error), y `Some(Some(2))` es dos veces. Donde se lee el encabezado, una cadena de let busca la cuenta al final del nombre:

```
if repeats.is_none()
    && let Some((first, last)) = name.rsplit_once(char::is_whitespace)
    && let Some(times) = read_times(last)
{
    repeats = Some(times);
    name = first.trim().to_string();
}
```

Hacer la canción

`text::import` empieza desde `Project::with_band()`, la canción que hace **Archivo > Nuevo**, y pone lo que dice el cifrado, en el orden que necesita la banda: primero el estilo (que puede pasar la canción vacía a su propio compás), después el compás, la tonalidad y el tempo del cifrado por encima. Luego recorre los elementos una vez, llevando el paso donde empieza el compás siguiente:

```
Item::Bar { text, line } => {
    if step + time.steps_per_bar() > MAX_SONG_STEPS {
        return Err(error(*line, tr("the song cannot be this long.")));
    }
    bars.push((step, text, *line));
    step += time.steps_per_bar();
}
Item::Time(new) => {
    time = *new;
    project.set_time_marker(TimeMarker::new(step, time));
}
Item::Key(key) => {
    project.set_key_marker(KeyMarker::new(step, *key));
}
Item::Section { name, repeats, .. } => sections.push((name.clone(),
*repeats, step)),
```

Una sección termina donde empieza la siguiente, o al final de la canción. Con cada marcador en su lugar, cada compás se lee con `read_typed_bar`, que recibe el compás (para saber sus pulsos) y la tonalidad al comienzo (para leer los números romanos). Todos los acordes entran en la canción en un solo `set_chords`, y se le dice a la banda que toque la hoja. `text::paste` hace la misma lectura en una canción que ya existe, desde un compás, y devuelve los marcadores en lugar de escribirlos; la ventana los convierte en un único `SheetAction::SetChords`, la misma acción que escribir en un compás, así que deshacer y la banda no necesitan nada nuevo.

Escribir el texto

`text::write` recorre los compases con `write_bar`, la función que llena el campo de texto de un compás al hacer clic en él, y junta líneas: palabras («[Verse ×2]», «Key: D minor») o una fila de hasta cuatro compases. Una sección, un cambio de tonalidad o de compás empieza una fila nueva, y un compás vacío se escribe `%` cuando volver a tocar el compás anterior suena igual, como después de «| C |». Después de «| C G |» un compás sostenido se escribe `/ / / /`, una `/` por pulso, porque `%` se volvería a leer como C G. Escribe en letras y en inglés, muestre lo que muestre la aplicación, para que el texto se lea igual dondequiera que se pegue:

```
pub fn write(project: &Project, title: Option<&str>) -> String {
    Language::English.with(|| NoteNaming::Letters.with(||
write_sheet(project, title)))
}
```

Después `layout` alinea los compases. Busca el compás más ancho de cada una de las cuatro columnas, y luego rellena cada compás hasta el ancho de su columna:

```
for (width, bar) in widths.iter_mut().zip(bars) {
    *width = (*width).max(bar.chars().count());
}
```

```
.map(|(bar, width)| format!("{bar:<width$}"))
```

`{bar:<width$}` escribe `bar` alineado a la izquierda (`<`) en un campo tan ancho como la variable `width` (el `$` dice que el ancho sale de una variable). Los anchos se cuentan con `chars().count()`, no con `len()`: `len()` cuenta los bytes del texto UTF-8, y una letra fuera del ASCII simple (ø, ×, ♭) ocupa dos o tres bytes pero un solo lugar en la pantalla. Una tonalidad que el texto no puede decir, una serie dodecafónica o un conjunto, se escribe como una nota para las personas, «# Key: … (chosen in the app)», para que el cifrado se siga leyendo y nadie se confunda.

Copiar, cortar y pegar

egui convierte Ctrl+C, Ctrl+X y Ctrl+V en eventos propios, con el texto pegado ya tomado del portapapeles del sistema. Es igual en Linux, en macOS (donde las teclas son Cmd+C, Cmd+X y Cmd+V) y en Windows. La hoja los busca entre los eventos del frame mientras tiene el teclado y no se está escribiendo en ningún compás; mientras se escribe en un compás, van al campo de texto como siempre.

```
for event in &input.events {
    match event {
        egui::Event::Copy => copy = true,
        egui::Event::Cut => cut = true,
        egui::Event::Paste(text) => paste = Some(text.clone()),
        _ => {}
    }
}
```

Para copiar, `ctx.copy_text(text)` pone el texto en el portapapeles al final del frame. Cortar es copiar seguido de un `SetChords` sin marcadores sobre los compases seleccionados. Sin ningún compás seleccionado, copiar toma la hoja entera, pero cortar no vaciaría nada, así que deja el portapapeles como estaba.

Cómo se prueba

La mejor prueba de un par de funciones como `write` y `read` es un **viaje de ida y vuelta**: escribir algo, volver a leerlo y comprobar que se obtiene lo mismo. `every_song_demo_comes_back_from_its_text` hace eso con cada demostración de canción: la hoja de acordes de cada una se escribe como texto, se importa, y sus acordes, tonalidades y compases se comparan con los de la demostración. Esa prueba es la razón de que «C G7 C /» conserve su barra y de que un nombre nunca pase por encima de un `/`. Unas pocas demostraciones tienen un acorde entre dos pulsos, que una hoja de acordes no puede escribir, así que la prueba compara los acordes que caen en los pulsos, y comprueba que menos de una demostración de cada diez tiene un acorde así. Junto a ella, `a_bar_is_written_back_as_it_reads` y `every_name_is_written_back_the_same_way` hacen lo mismo con un compás y con un nombre.

En `chord_sheet.rs`, `harmonies_spread_over_the_beats_as_on_a_lead_sheet` comprueba la tabla de más arriba, `a_bar_cannot_hold_more_than_its_beats` el rechazo, `every_way_of_naming_harmony_is_read` y `nonsense_is_refused_with_a_reason` los nombres, `numerals_are_chords_in_the_key` y `typed_numerals_in_minor_follow_the_way_numerals_are_written` los números romanos, y `harmonies_move_with_the_key` un cambio de tonalidad. En `text.rs`, `a_chart_is_read_line_by_line`, `bar_lines_are_read_however_they_are_drawn`, `a_mistake_says_its_line`, `numerals_are_read_in_the_charts_key`, `pasted_bars_go_from_the_bar_chosen_on` y `the_sheet_is_written_in_columns` cubren el texto.

Las pruebas de `bars.rs` trabajan sobre una canción pequeña: un patrón de cuatro compases que toca C, F, G7, C, una estrofa sobre todo él, un bucle sobre los compases 2 y 3, la tonalidad de Sol desde el compás 3 y 90 BPM desde el compás 4. `an_inserted_bar_moves_everything_after_it` comprueba cada nota, acorde, tonalidad, tempo, la estrofa y el bucle; `a_bar_inserted_before_a_change_of_time_takes_the_new_time`, `a_clip_of_a_pattern_played_elsewhere_too_is_cut_in_two`, `a_deleted_bar_closes_up`, `a_deleted_bar_leaves_no_pattern_behind`, `a_harmony_from_a_deleted_bar_goes_on_where_it_was` y `a_repeated_clip_loses_the_repeats_in_the_deleted_bars` siguen los casos de más arriba, y `the_band_follows_the_bars` comprueba que ninguna parte de la banda quedó congelada. Para oír que nada más se movió, la traza de reproducción de una demostración antes y después de insertar muestra cada nota anterior al compás en el mismo momento, y cada nota posterior exactamente un compás más tarde.

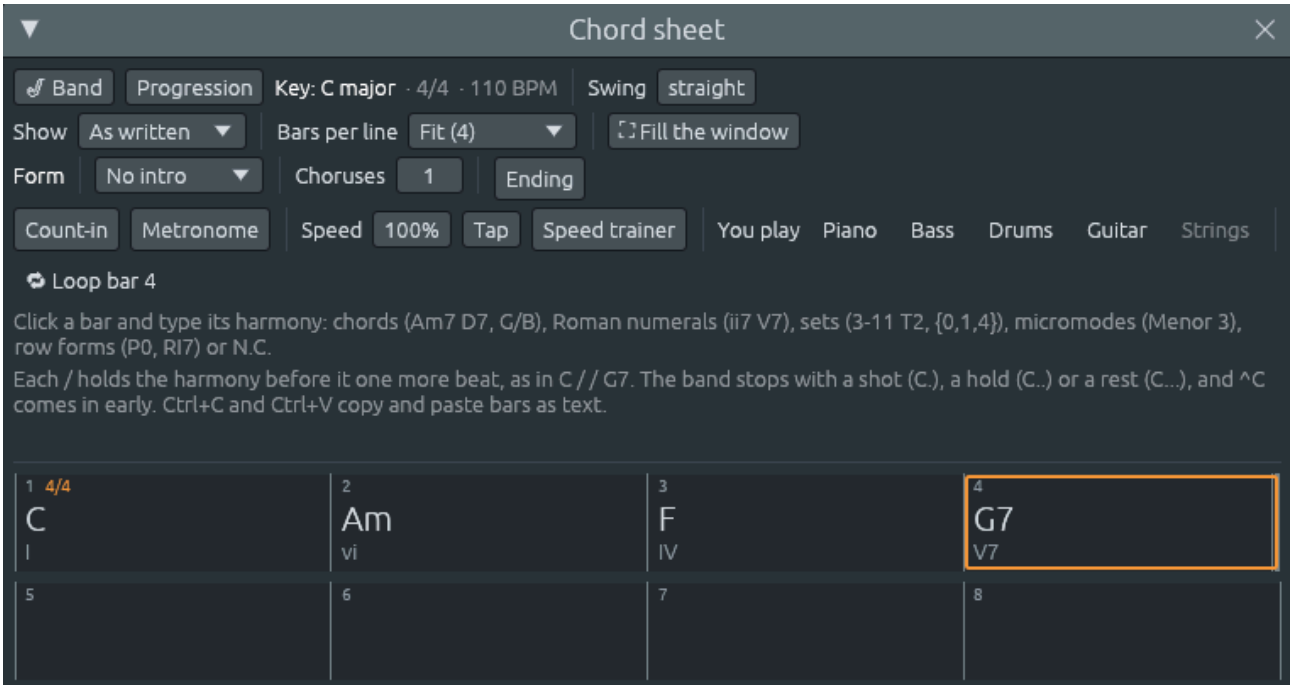
En la aplicación, `lines_fit_as_many_whole_phrases_as_there_is_room_for` comprueba `count`. `lines_hold_as_many_bars_as_show_every_name_in_full` empieza una canción cuyo primer compás es «Dm7/C F/C G7/D C/E» y comprueba que cada nombre se dibuja a tamaño completo, 20 puntos; después elige 8 compases y encuentra «Dm7/C» más pequeño.
`the_words_after_a_numeral_are_written_where_there_is_room` y `a_short_chord_keeps_its_name_whole_with_the_next_just_after_it` comprueban la colocación, `the_zoom_keys_over_the_sheet_in_the_main_window_zoom_the_sheet` y `the_sheet_fills_the_main_window_and_goes_back` la ventana principal, y `a_bars_menu_inserts_a_bar_and_deletes_it` y `the_bar_numbers_insert_and_delete_bars_too` los dos menús.

Los ejercicios sobre la hoja de acordes están en Ejercicios: la banda.

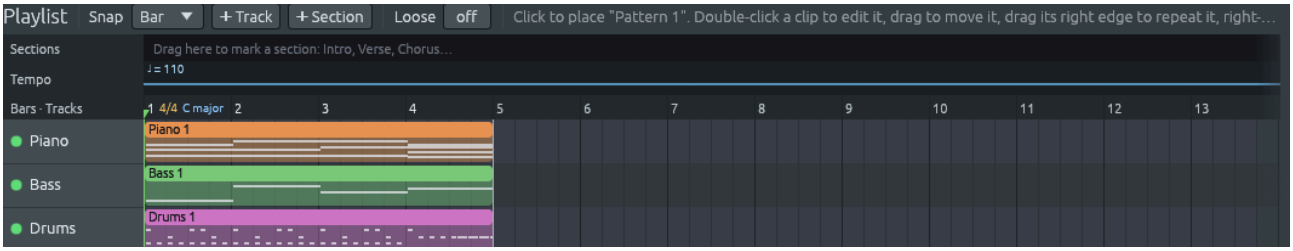
Cómo toca la banda una hoja de acordes

Archivos: `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/band/`, `crates/lineup-core/src/library/comping.rs`, `crates/lineup-core/src/library/follow.rs`, `crates/lineup-app/src/chord_sheet/band.rs` y `crates/lineup-app/src/app/follow.rs`

La hoja de acordes guarda la armonía de una canción. La banda la toca. Escribe `C`, Tab, `Am`, Tab, `Dm7 G7`, Tab, `C`, pulsa Espacio, y un piano, un bajo y una batería tocan esos acordes, con una guitarra, cuerdas, un órgano o una sección de vientos donde el estilo los tiene. Cambia el estilo, la tonalidad o un acorde, y la banda lo vuelve a tocar de la manera nueva.



Las partes de la banda son patrones corrientes en pistas propias, llamados «Piano 1», «Bajo 1», «Batería 1». Haz doble clic en uno y el piano roll muestra cada nota que toca la banda; nada está escondido, lo que importa en un programa con el que se aprende: puedes ver cómo un pianista de bossa nova reparte un Dm7 por el compás, o dónde pone un bajo caminante su nota de entrada.



Este capítulo cuenta cómo se separa un estilo de los acordes que toca, qué toca cada músico y cómo, cómo se vuelven a escribir las partes cuando cambia la hoja sin perder lo que editaste, la banda más grande (guitarra, cuerdas, órgano, vientos), y el modo de teclado arreglador en que la banda sigue el acorde que mantienes pulsado. La forma de la canción, las variaciones y los estilos por sección están en Forma, secciones, variaciones y bloqueos; los cortes en lo que toca la banda, en Golpes, sostenidos, silencios y anticipaciones. Cómo se usa la banda está en los capítulos del manual Tu primera canción y Una banda más grande y los vientos.

La música, en pocas palabras

Unas cuantas palabras aparecen en todo el código de la banda:

- **Comp** (de *accompany*): un acorde tocado debajo de la melodía; como verbo, tocar esos acordes. El **comping** es el ritmo y las disposiciones que usa quien toca los acordes, como los acordes sincopados de un pianista de bossa.
- **Disposición** (*voicing*): qué notas de un acorde suenan y en qué octavas. La **conducción de voces** lleva cada voz de un acorde a la voz más cercana del siguiente, para que los acordes se unan con suavidad.
- **Cambios** (*changes*): la progresión de acordes de una canción, compás por compás; lo que guarda la hoja de acordes.
- **Bajo caminante** (*walking bass*): una nota por pulso, primero la fundamental, notas del acorde, y medio tono hacia la fundamental siguiente.
- **Colchón** (*pad*): acordes sostenidos suavemente debajo de la banda, normalmente por cuerdas o un órgano.
- **Vuelta** (*turnaround*): los acordes al final de una sección que llevan de nuevo a su comienzo, como I vi ii V.

Qué tocar y cómo tocarlo

Un estilo de la biblioteca es un paquete: «Bossa nova» quiere decir Dm7 G7 Cmaj7 *y* los acordes sincopados del piano *y* el bajo en el 1 y en el «y» del 2 *y* los golpes de aro.

Una banda de acompañamiento separa las dos mitades. Los acordes son tuyos, en la hoja; el estilo solo dice cómo tocarlos.

Mitad	Qué es	Dónde se elige
Qué tocar	una progresión de números romanos	el menú **Progresión** de la hoja, del cifrado de cada `Style`
Cómo tocarlo	comping, bajo, batería, los músicos que añade	la **Biblioteca**, como un `BandStyle`

Un `BandStyle` es un estilo sin los acordes:

```
pub struct BandStyle {
    pub name: &'static str,
    pub genre: Genre,
    pub time: TimeSignature,
    /// The tempo it is usually played at, in BPM.
    pub tempo: f32,
    pub comping: Comping,
    pub bass: Bass,
    /// The drums, or `None` for a band without them.
    pub rhythm: Option<&'static Rhythm>,
    pub about: &'static str,
}
```

Los estilos de banda no se escriben a mano. Cada ritmo de la biblioteca ya sabe cómo van con él el piano y el bajo (`apply::feel`, ver Tocar una canción en otro estilo), así que la lista se hace a partir de los ritmos la primera vez que se pide, con cuatro estilos sin batería añadidos: **Coral** (cada acorde sostenido, como un órgano toca un himno), **Bajo de Alberti** (el acorde quebrado grave, agudo, medio, agudo, como en las sonatas de Mozart), **Arpegios** y **Zarabanda** en 3/4.

```
pub fn band_styles() -> &'static [BandStyle] {
    static STYLES: OnceLock<Vec<BandStyle>> = OnceLock::new();
    STYLES.get_or_init(|| {
        let mut styles: Vec<BandStyle> = RHYTHMS
            .iter()
            .map(|rhythm| {
                let (comping, bass) = apply::feel(rhythm);
                BandStyle {
                    name: rhythm.name,
                    genre: rhythm.genre,
                    time: rhythm.time,
                    tempo: rhythm.tempo,
                    comping,
                    bass,
                    rhythm: Some(rhythm),
                    about: rhythm.about,
                }
            })
            .chain(WITHOUT_DRUMS)
            .collect();
        // A stable sort: within a genre, the library's order stays.
        styles.sort_by_key(|style| style.genre);
        styles
    })
}
```

Un `static` tiene que construirse al compilar el programa, y un `Vec` no se puede, porque necesita memoria del heap. `std::sync::OnceLock` es una caja que empieza vacía y se llena una vez, la primera vez que se ejecuta `get_or_init`; cada llamada posterior recibe el mismo valor. Varios hilos pueden competir por llamarla y solo se ejecuta un closure. El tipo de retorno `&'static [BandStyle]` dice «esta lista vive tanto como el programa», y eso permite que un `&'static BandStyle` esté en el proyecto como estilo de la banda sin ningún tiempo de vida que seguir. `sort_by_key` es estable, así que los estilos de un mismo género conservan el orden de la biblioteca.

Una banda en la canción

Archivo > Nuevo hace una canción con una hoja vacía y una banda lista para tocarla en **Ritmo pop**:

```
pub fn with_band() -> Self {
    let mut project = Project::empty();
    project.set_band_style(BandStyle::named(DEFAULT_STYLE));
    project.remove_unused_tracks();
    project
}
```

A partir de ahí, tres lugares dan forma a la banda:

- **La Biblioteca.** Un clic en un estilo o un ritmo lo convierte en la manera de tocar de la banda, con las partes marcadas en su ventana: la batería, las partes de acordes y bajo, las voces, la progresión de acordes, el compás, el tempo y el swing (ver Qué partes tomar). Una canción con música propia pero sin banda recibe una cuando se toman las partes de acordes y bajo con las voces.
- **El menú 🎷 Banda de la hoja** elige quién toca y con qué sonido: una **Formación de instrumentos** («Piano, guitarra acústica + cuerdas», «Cornos, timbales + cuerdas» y seis más), una casilla y un instrumento para cada músico, la **Sección de vientos**, el **Solista** (ver El solista) y **Sin banda**.
- **El menú Progresión de la hoja** es la Biblioteca otra vez, mostrando solo los estilos con una progresión de acordes; un clic escribe esa progresión desde el compás seleccionado, en la tonalidad de la canción: I vi ii V en Do es Cmaj7 Am7 Dm7 G7, la vuelta con que terminan tantos standards. Un acorde que ya está sonando no se vuelve a escribir, así que «I | I | IV» escribe C, nada, y después F, como un cifrado. En una hoja vacía, la canción toma el modo de la progresión: «i bVII bVI» es Am G F en La menor.

Como las partes pueden venir de lugares distintos, la banda las guarda por separado:

```rust
pub struct Band {
    pub custom_rhythm: Option<Box<crate::library::UserStyle>>,
    pub custom_instruments: Option<Box<crate::library::UserStyle>>,
    pub style: &'static BandStyle,
    /// The instrumentation can come from a different style than the
rhythm.
    pub instrument_style: &'static BandStyle,
    /// The exact library style supplying the comping, when one was
picked.
    pub harmony_style: Option<&'static Style>,
    /// The chords and the bass played as plain accents on the beats: how
    /// songs saved so play. The Library no longer chooses it.
    pub accents_only: bool,
    /// Whose drums play, when not those of `style` or `custom_rhythm`.
    pub drums: BandDrums,
    // ...
}

pub enum BandDrums {
    /// Those of the style the band plays the chords and the bass in, or
    /// of the saved style it plays them from.
    Style,
    /// Those of another style.
    Of(&'static BandStyle),
    /// Those of a saved style.
    Saved(Box<crate::library::UserStyle>),
    /// None: the song's own drums play.
    Song,
}
```

`style` dice cómo se tocan los acordes y el bajo; `drums` dice de quién es la batería que toca con ellos; `instrument_style` dice quién toca y con qué; y `harmony_style` es el estilo de la biblioteca elegido, cuando su comping difiere del de su ritmo. Los dos campos `custom_` guardan estilos tuyos (ver Tus propios estilos). `accents_only` toca los acordes y el bajo en los pulsos en que se siente el compás, sea cual sea el estilo; solo las canciones guardadas así siguen tocando de ese modo.

Tomar las partes de acordes y bajo de un estilo pero no su batería conserva la batería que tocaba la banda: `drums` pasa de `Style` a `Of` el estilo anterior, así que deja de seguir a `style`. Tomar solo la batería pone `Of` el nuevo. Cuando las dos vuelven a ser la misma, `tidy_drums` convierte `Of` otra vez en `Style`, así que una banda tiene una sola manera de decir «la batería de este estilo», y su archivo sigue siendo uno que una versión anterior puede leer. Cada músico le pregunta a la banda, no al estilo, cómo tocar:

```
fn comping_in(&self, variation: Variation) -> Comping {
    if self.accents_only {
        Comping::Accents
    } else {
        self.playing_style().comping_in(variation)
    }
}
```

Elegir un estilo puede cambiar tres cosas más. Un estilo en otro compás pasa la canción a ese compás, compás por compás, como lo hace un ritmo de la biblioteca (ver Pasar una canción a otro compás): cada compás conserva su duración en segundos, así que una canción en 4/4 a 110 BPM tocada como Vals de jazz pasa a 3/4 a 82,5 BPM (4 × 60 / 110 = 3 × 60 / 82,5 = 2,18 segundos por compás). El tempo es el del estilo mientras la hoja está vacía, y el tuyo una vez que escribiste acordes. Y la canción toma el swing del estilo, porque el swing es parte de cómo se toca un estilo. El piano y el bajo reciben el instrumento General MIDI del género (un piano eléctrico para los estilos soul y electrónicos, un contrabajo para el jazz y el tango, un bajo sintetizado para el hip hop), para que suenen bien con un SoundFont o por la salida MIDI.

Qué toca cada músico

Un estilo de banda sabe *cómo*; la hoja sabe *qué*. Juntarlos lleva tres pasos.

Las armonías, como notas

`played_harmonies` recorre la hoja y convierte cada armonía en un `Played`: cuándo empieza y termina, las notas con que se tocan los acordes (su disposición) y lo que toca el bajo debajo.

```
pub struct Played {
    pub start: usize,
    pub end: usize,
    /// The notes the chords are played with, from low to high. Empty for
    /// no chord: nothing plays.
    pub voicing: Vec<Note>,
    /// What the bass plays.
    pub bass: BassPart,
}

/// What the bass plays under a harmony.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum BassPart {
    /// A chord: its line goes over the chord's notes from `note` (its
    /// root, or a slash chord's bass), and may lead into `next`, the
    /// bass note of the harmony after it.
    Chord {
        chord: Chord,
        note: Note,
        next: Note,
    },
    /// A group of notes with no root, such as a pitch-class set: the
    /// bass keeps the rhythm of its line on this one note and its octave.
    Pedal(Note),
    /// Nothing.
    Rest,
}
```

Los acordes se disponen como la biblioteca dispone un cifrado, cada uno a partir del anterior, moviendo cada voz lo menos posible, para que C Am F G7 no salte por todo el teclado (ver Conducción de voces). Sus notas del bajo forman una línea (`bass_line`), así que `G7/B` tiene Si en el bajo. Las otras maneras en que Ferrous Lineup nombra la armonía se tocan por sus notas:

- **Un poliacorde** se toca como está escrito: el acorde de abajo, el de arriba apilado justo encima, sobre el bajo del acorde de abajo.
- **Un conjunto de clases de altura o un micromodo** no tiene fundamental, así que no hay una nota del bajo «correcta». La banda toma la primera nota de su forma normal (su orden más compacto): 3-11 T2, Re Fa La, se toca desde el Do central hacia arriba, sobre un Re. La elección es arbitraria pero fija, así que el mismo conjunto siempre suena igual.
- **Una forma de serie** es una línea de doce notas, no un acorde. La banda la toca como cuatro acordes de tres notas, en el orden de la serie, uno tras otro, cada uno sobre su primera nota. Cortar una serie en grupos y hacer sonar cada grupo como un acorde es una manera habitual de armonizarla; Webern construyó la serie de su Concierto, Op. 24, con cuatro grupos así.
- **N.C.** no toca nada; la batería sigue.

El ritmo, ajustado a los compases

Los patrones de comping de la biblioteca empiezan de nuevo en cada acorde. Eso está bien para un cifrado donde los acordes cambian en la barra de compás, pero una hoja puede decir `Dm7 G7` (un cambio en el pulso 3) o `C / / G7` (en el 4). Empezar el patrón de nuevo en el pulso 3 movería el groove cada vez que cambia el acorde, como si el baterista contara «uno» en medio del compás. Una banda mantiene su groove y deja que los acordes caigan dentro. Así que `Comping::play_in_bars` pone los golpes del ritmo en cada compás, desde su primer pulso, y cada golpe toca la armonía que esté sonando cuando cae:

```
for bar in bars {
    let (period, hits) = self.hits(bar.time);
    let from = if self.follows_the_bar() {
        bar.start
    } else {
        period_start(bar, period)
    };
    for offset in (0..bar.end() - from).step_by(period) {
        for hit in hits.iter() {
            let at = from + offset + hit.at;
            if at < bar.start || at >= bar.end() {
                continue;
            }
            let index = harmonies.partition_point(|harmony| harmony.start
<= at);
            let Some(index) = index.checked_sub(1) else {
                continue;
            };
            // ...
        }
    }
}
```

Las armonías están ordenadas por su comienzo, así que `partition_point` cuenta, con una búsqueda binaria, las armonías que ya empezaron en `at`; la que suena es la anterior. Un acorde tan corto que ningún golpe cae en él (un G7 en el «y» del 4, en un patrón que solo golpea en el 1 y el 3) no se oiría, y un acorde saltado es una armonía equivocada, así que un acorde así se golpea una vez donde empieza. Un patrón más largo que un compás, como la clave de dos compases de la bossa nova, cuenta desde el compás 1: `period_start` lo empieza en los compases impares y lo sigue en los pares, como un músico cuenta «uno, dos» a lo largo del par.

Pulsos para cualquier compás

La mayoría de los patrones de comping son una tabla fija de golpes. Dos no: los **acentos simples** (`Comping::Accents`), un acorde en cada pulso en que se siente el compás, y el **aksak** de los estilos balcánicos, los contratiempos dentro de cada pulso. En 7/8 sentido como 2+2+3 caen en un lugar, en 5/4 sentido como 2+3 en otro, así que sus golpes se construyen a partir de los `felt_groups` del compás (ver Pulsos de distinta longitud):

```
fn hits(self, time: TimeSignature) -> (usize, Cow<'static, [Hit<Voices>]>)
{
    match self {
        // ...
        Comping::Aksak => {
            let unit = time.unit_steps();
            // A chord a little shorter than its note, so the next
            // one is heard to start, but never under a step.
            let length = (unit - unit / 4).max(1);
            let mut hits = Vec::new();
            let mut at = 0;
            for group in time.felt_groups() {
                for note in 1..usize::from(group) {
                    let velocity = if note == 1 { 84 } else { 74 };
                    hits.push(hit!(at + note * unit, length, Voices::All,
velocity));
                }
                at += usize::from(group) * unit;
            }
            (time.steps_per_bar(), Cow::Owned(hits))
        }
        _ => {
            let (period, hits) = self.pattern();
            (period, Cow::Borrowed(hits))
        }
    }
}
```

La función devuelve un `Cow`, «clonar al escribir» (*clone on write*; ver El orden de todo): los patrones fijos entregan sin coste un préstamo de su tabla `static`, `Cow::Borrowed`, y los dos construidos un `Vec` propio, `Cow::Owned`. Quien llama lee los dos de la misma manera. Estos dos patrones cuentan desde la barra de compás, empiece donde empiece el acorde (`follows_the_bar`), ya que los pulsos de un compás irregular están donde los pone el compás.

El bajo

`Bass::play_in_bars` es la misma idea para la línea del bajo: empieza de nuevo donde empieza cada armonía, para que la fundamental caiga en el cambio, y de nuevo en cada barra de compás, para que un acorde sostenido ocho compases no se convierta en una sola caminata larga. Solo el último trozo de una armonía conduce a la siguiente. Un bajo caminante toca una nota por pulso:

```
fn walk(span: usize, down: bool) -> Vec<(usize, Tone)> {
    const UP: [Tone; 6] = [
        Tone::Third,
        Tone::Fifth,
        Tone::Seventh,
        Tone::Octave,
        Tone::Fifth,
        Tone::Third,
    ];
    const DOWN: [Tone; 6] = [
        Tone::LowSeventh,
        Tone::LowFifth,
        Tone::LowThird,
        Tone::LowFifth,
        Tone::LowSeventh,
        Tone::Root,
    ];
    let beats = span.div_ceil(4);
    (0..beats)
        .map(|beat| {
            let tone = if beat == 0 {
                Tone::Root
            } else if beat == beats - 1 {
                Tone::Approach
            } else if beats == 3 {
                if down { Tone::LowFifth } else { Tone::Fifth }
            } else if down {
                DOWN[(beat - 1) % DOWN.len()]
            } else {
                UP[(beat - 1) % UP.len()]
            };
            (beat * 4, tone)
        })
        .collect()
}
```

La fundamental en el 1, notas del acorde en el medio, y en el último pulso la **aproximación**, una nota medio tono por encima o por debajo de la nota del bajo del acorde siguiente. La línea recibe esa nota misma, no solo su nombre: una fundamental se toca entre Do2 y Si2 (ver Las armonías, como notas), así que de G7 a C la nota siguiente es Do2, y la aproximación es el medio tono del lado más cercano a la

nota anterior, Re♭2 bajando desde Re2. Sabiendo solo «Do», la línea podría elegir Si2 y después caer una séptima al Do2 que sigue.

Las notas del medio suben por el acorde (`UP`) o bajan por él (`DOWN` , las notas del acorde por debajo de la fundamental). `Bass::play` arma las dos líneas y se queda con la que llega al acorde siguiente con pasos más chicos, la que baja solo mientras se mantenga por encima de Mi1 (`BASS_FLOOR`), la cuerda más grave de un bajo eléctrico. Así, sobre C yendo a Am la caminata sube, Do2 Mi2 Sol2 Sol♯2, hacia La2, y sobre G7 yendo a C baja, Sol2 Fa2 Re2 Re♭2, hacia Do2. En un compás de tres la caminata va fundamental, quinta, aproximación: sobre C7 yendo a F, Do2 Sol2 Fa♯2, hacia Fa2. La tercera queda fuera porque la tercera de Do *es* el medio tono bajo Fa: la línea llegaría a su aproximación un pulso antes.

La batería

La batería toca el ritmo del estilo en cada compás, con su relleno en el último compás de cada ocho (`PHRASE_BARS` , como se construyen la mayoría de las canciones), en el último compás antes de una sección o de un cambio de variación, y al final de la introducción y de la vuelta. Un cambio de variación o de estilo empieza con un platillo crash. Las variaciones, la introducción y el final están en Forma, secciones, variaciones y bloqueos.

Cada parte, como patrones

Una canción entera no cabe en un patrón (un patrón tiene como mucho 256 pasos, 16 compases de 4/4), y un patrón gigante sería difícil de leer. `chunks` corta la canción en tramos: compases enteros de un mismo compás, dentro de una sección, tan largos como puede ser un patrón. `write_part` corta las notas en esas líneas (una nota que dura más allá del final de un tramo sigue al comienzo del siguiente) y compara las notas de cada tramo con las de los anteriores:

```
let index = match contents.iter().position(|known| *known == content) {
    Some(index) => index,
    None => {
        contents.push(content);
        contents.len() - 1
    }
};
```

Los tramos que tocan exactamente las mismas notas comparten **un** patrón, con dos clips de él, como un productor copia un clip en lugar de tocar dos veces una parte. Una canción de dos vueltas iguales tiene «Piano 1» dos veces en la lista de

reproducción. Los patrones se escriben encima de los que ya tenía el músico, así que «Piano 1» conserva su número mientras editas la hoja, y un piano roll abierto sobre él muestra las notas nuevas en lugar de cerrarse. Cada patrón que escribe la banda lleva una etiqueta:

```
/// The band's player whose part it holds, if the band wrote it (see
/// [`crate::band`]). The band writes these patterns again when the
/// chord sheet changes.
pub player: Option<Player>,
```

Esa etiqueta es la manera en que la banda vuelve a encontrar su propio trabajo, y en que la biblioteca sabe que debe dejarlo en paz cuando toca una canción entera en otro ritmo.

Volver a tocar cuando cambia la hoja

La banda toca lo que la hoja dice *ahora*. Escribe un acorde, cambia la tonalidad, pasa la canción a otro compás, arrastra una sección: las partes se vuelven a escribir, en el mismo paso de deshacer que tu cambio, así que un Ctrl+Z deshace los dos. Todo cambio del proyecto pasa por `LineupApp::edit` (ver Deshacer y rehacer), así que ahí es donde se avisa a la banda:

```
let before = self.project.clone();
let result = change(&mut self.project);
// The band plays what the chord sheet says now, in the same step.
self.project.follow_chord_sheet();
```

Volver a escribir cada parte después de cada clic sería un desperdicio: mover un clip de la melodía no cambia nada de lo que toca la banda. Así que `follow_chord_sheet` primero toma una **huella** de aquello a partir de lo cual se tocan las partes (el estilo, los marcadores de acorde, las tonalidades, los compases, las secciones, la forma) y la compara con la que tomó la última vez:

```rust
pub fn follow_chord_sheet(&mut self) -> bool {
    let Some(band) = &self.band else {
        return false;
    };
    if band.heard == Some(self.band_inputs()) {
        return false;
    }
    self.band_channels(false);
    self.write_band();
    true
}
```

Una huella, o hash, es un número calculado a partir de unos datos, de manera que datos distintos dan casi seguro números distintos. Comparar dos números de 64 bits es mucho más barato que comparar dos hojas de acordes, y la banda solo tiene que guardar el número, no una copia de la hoja.

Una parte que editas sigue siendo tuya

Una banda que escribiera sus partes de cero cada vez perdería una edición de la línea del bajo la próxima vez que cambiara algo. Aquí, una parte que cambias a mano queda **congelada**: la banda la deja como la dejaste, y la hoja lo dice, con un botón **Volver a tocarla** que descarta tu versión y hace que la banda escriba la parte de nuevo a partir de la hoja. Los demás músicos siguen siguiendo la hoja.



¿Cómo sabe la banda que editaste una parte, con la herramienta que sea: el piano roll, el channel rack, un clip arrastrado, la biblioteca? No vigila las herramientas. Guarda, para cada músico, una huella de la parte que escribió por última vez, y la compara con una huella de la parte tal como está ahora:

```
pub fn is_frozen(&self, player: Player) -> bool {
    self.band.as_ref().is_some_and(|band| {
        band.written[player.index()].is_some_and(|written| written !=
self.part_print(player))
    })
}
```

Dos `is_some_and` anidados se leen como una sola frase: «hay una banda, y escribió la parte de este músico, y eso no es lo que hay ahora». `part_print` mezcla dónde empieza cada clip del músico, cuántas veces se repite y cada nota de su patrón. En qué pista está un clip y qué número tiene su patrón quedan fuera, así que bajar la pista Piano, o eliminar un patrón tuyo (lo que renumera los de después), no congela nada.

FNV-1a

Las huellas se guardan en el archivo del proyecto, así que una parte que congelaste sigue congelada cuando abres la canción mañana. El `DefaultHasher` de la biblioteca estándar no promete dar el mismo número en la próxima versión de Rust, así que no se puede guardar. La banda usa **FNV-1a**, un hash clásico de unas pocas líneas:

```
struct Fingerprint(u64);

impl Fingerprint {
    fn new() -> Self {
        Fingerprint(0xCBF2_9CE4_8422_2325)
    }

    fn bytes(&mut self, bytes: &[u8]) {
        for &byte in bytes {
            self.0 ^= u64::from(byte);
            self.0 = self.0.wrapping_mul(0x0100_0000_01B3);
        }
    }

    fn number(&mut self, number: usize) {
        self.bytes(&(number as u64).to_le_bytes());
    }
}
```

Para cada byte: mezclarlo con el número con XOR, y después multiplicar por un primo grande, `0x100000001B3`. El número inicial y el primo son las constantes publicadas del algoritmo, así que todo programa que implementa FNV-1a obtiene el mismo resultado. Multiplicar dos `u64` grandes desborda; en una compilación de depuración un `*` simple detendría el programa con «attempt to multiply with overflow», y

`wrapping_mul` dice que quedarse solo con los 64 bits bajos es lo que queremos, como hace un hash a propósito. `to_le_bytes` escribe los bytes de un número empezando por el más bajo (*little-endian*), sea cual sea el ordenador, así que la misma canción da la misma huella en todas partes. Una parte sin ninguna nota es el número inicial mismo, 14695981039346656037: lo verás en un archivo guardado, para unas cuerdas que esperan a la variación b.

Mover la canción entera no es editar

Cambiar la tonalidad de la canción transpone cada nota, también las de la banda. Las partes de la banda ya no coinciden con sus huellas, pero tú no las editaste: la herramienta movió todo. Pasar la canción a otro compás, copiar una sección e insertar un compás son lo mismo. Estas herramientas se ejecutan dentro de `moving_the_band`:

```
pub(crate) fn moving_the_band<R>(&mut self, change: impl FnOnce(&mut
Project) -> R) -> R {
    self.keeping_parts(|_, _| true, change)
}
```

```
fn keeping_parts<R>(
    &mut self,
    keep: impl Fn(&Project, Player) -> bool,
    change: impl FnOnce(&mut Project) -> R,
) -> R {
    let following: Vec<Player> = Player::ALL
        .into_iter()
        .filter(|&player| {
            self.band
                .as_ref()
                .is_some_and(|band|
band.written[player.index()].is_some())
                && !self.is_frozen(player)
                && keep(self, player)
        })
        .collect();
    let result = change(self);
    let prints: Vec<(Player, u64)> = following
        .into_iter()
        .map(|player| (player, self.part_print(player)))
        .collect();
    if let Some(band) = &mut self.band {
        for (player, print) in prints {
            band.written[player.index()] = Some(print);
        }
    }
    result
}
```

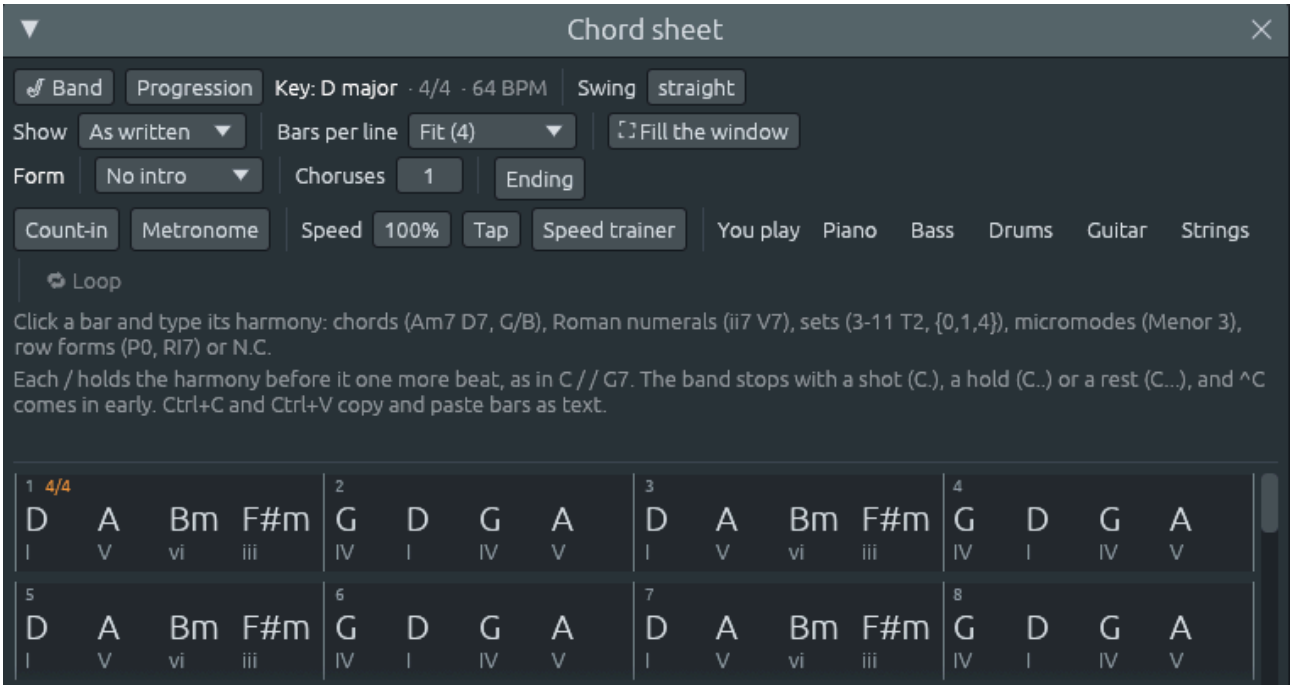
Anota qué partes seguían la hoja antes, ejecuta la herramienta, y después toma sus huellas nuevas como propias de la banda. Una parte congelada se mueve con la herramienta como cualquier otra nota, y sigue congelada: tu versión del piano, ahora en Re. `keep` es un `impl Fn`, que se llama muchas veces, mientras que `change` es un `impl FnOnce`, que se llama una vez. Las huellas se juntan en un `Vec` antes de tomar `self.band` como préstamo mutable, porque `part_print` también necesita `self`, y el borrow checker permite uno u otro a la vez. Eliminar un canal usa el mismo método con otro `keep`: a una parte que no tocaba nada en el canal eliminado solo se le corrieron los números de canal, y eso no es una edición; una parte que perdió notas queda congelada.

Una banda sobre una canción con música propia

Abre una demostración, como el **Canon en Re**, y toma las partes de acordes y bajo y las voces de un estilo de la Biblioteca. Una demostración ya toca su propio acompañamiento, así que primero la banda:

1. escribe en la hoja los acordes que toca la canción, si está vacía (ver De dónde vienen los acordes), y
2. **silencia** los canales de acordes y de bajo de la propia canción (la biblioteca los llama «Keys» y «Bass», que la interfaz en español muestra como Teclados y Bajo) y, cuando también llega la batería del estilo, sus pistas que solo tienen batería,

para que las dos no toquen una encima de la otra. La melodía sigue sonando, ahora con la banda detrás. Sin la batería del estilo, la banda entra con `drums: BandDrums::Song`, y la batería propia de la canción sigue sonando. Un estilo que sigue a la canción, como la orquesta, entra junto a las partes de la canción en lugar de silenciarlas (ver Una orquesta para música que cambia de compás).



Sin banda quita los patrones, las pistas y los canales de la banda (solo los que añadió, y solo si nada más toca en ellos), y vuelve a hacer sonar lo que silenció. Recuerda lo que silenció por su *nombre*, no por su número, así que vuelven los canales correctos aunque hayas añadido o quitado otros entretanto. Los acordes se quedan en la hoja.

Una banda más grande: guitarra, cuerdas y órgano

Un piano, un bajo y una batería son un trío de jazz, y la sección rítmica de la mayoría de las bandas, pero una banda de rock sin su guitarra o un tango sin sus violines no suenan como ellos mismos. Así que un estilo puede traer hasta tres músicos más, cada uno tocando como lo usa esa música:

Estilo	Quién entra	Qué toca
Ritmo pop	guitarra de acero, cuerdas en b	acordes rasgueados, cuerdas sostenidas
Ritmo de rock	guitarra saturada, órgano de rock en b	power chords en corcheas
Shuffle, Rock and roll	guitarra eléctrica limpia	el shuffle de boogie
Swing	guitarra de jazz	cuatro por compás
Funk, Disco	guitarra apagada, y cuerdas en el disco	semicorcheas, rascadas
One drop, Steppers, Rocksteady	guitarra limpia, órgano	el chop en el 2 y el 4, el burbujeo del órgano
Bossa nova	guitarra de nailon	el ritmo de bossa
Tangos	cuerdas	acordes sostenidos
Coral	órgano de iglesia	acordes sostenidos
Hip hop, la mayoría de los estilos electrónicos	nadie	

«Cuerdas en b» quiere decir que esperan a la variación b: haz clic derecho en un compás, elige **Variación b**, y desde ahí entran las cuerdas, sosteniendo los acordes una octava por encima del piano, como las cuerdas levantan un estribillo pop.

La música: cómo toca un acorde un guitarrista

Un pianista puede tocar cualquier nota de un acorde: diez dedos, cada tecla al alcance. Un guitarrista tiene seis cuerdas, afinadas desde la más grave

Cuerda	6	5	4	3	2	1
Nota al aire	Mi2	La2	Re3	Sol3	Si3	Mi4
MIDI	40	45	50	55	59	64

y una mano izquierda con cuatro dedos para pisarlas contra los **trastes**, un semitono cada uno. Cada cuerda toca una nota a la vez, o ninguna. Una cuerda sin pisar suena **al aire**, gratis; por eso las tonalidades de Mi, La, Re y Sol, cuyos acordes usan muchas cuerdas al aire, son las tonalidades de la guitarra. Así que un acorde de guitarra es una **forma**: un traste (o nada) para cada cuerda. Los libros de acordes la escriben con seis caracteres desde la cuerda grave de Mi, con `x` para una cuerda que no se toca y `0` para una al aire:

```
C    x32010     la cuerda La en el traste 3 (Do), Re en el 2 (Mi), Sol al
aire (Sol),
                Si en el 1 (Do), Mi aguda al aire (Mi): Do Mi Sol Do Mi
G    320003
Am   x02210
F    133211     el primer dedo plano sobre el traste 1: una "cejilla"
```

La cuerda más grave que suena tiene que ser el **bajo** del acorde, la nota que toca el bajista (la fundamental, o en un acorde con barra la nota después de la barra). Por eso la forma de Do deja fuera la cuerda grave de Mi: un Mi allí la convertiría en C/E, que los libros de acordes escriben `032010`.

Ferrous Lineup no guarda un libro de acordes. Encuentra las formas como las aprende un guitarrista, probando los trastes al alcance y quedándose con la más fácil. Para cada **posición** en el mástil (el traste donde se apoya el primer dedo), prueba todas las maneras de poner una nota del acorde en cada cuerda dentro de cuatro trastes, se queda con las que una mano puede tocar y las puntúa:

- toda nota **necesaria** tiene que estar: la fundamental, la tercera (o la segunda o la cuarta de un acorde sus), la sexta o la séptima, una quinta que no sea justa, y la extensión más alta, que da nombre al acorde (la 13 de un G13);
- una quinta justa y las extensiones más bajas son **bienvenidas**, y las formas con más de ellas son mejores;
- las formas bajas en el mástil, con **cuerdas al aire** y en más cuerdas son mejores;
- hasta cuatro cuerdas pisadas está bien. Más necesitan una **cejilla**, un dedo plano sobre el traste más bajo, que pisa todas las cuerdas por encima de él, así que ninguna de ellas puede ir al aire, y deja tres dedos para el resto.

Una regla más mantiene claro el sonido. Dos notas juntas suenan turbias en el grave: una tercera entre Mi2 y Sol♯2 retumba, mientras que una quinta allí está bien. Los arreglistas tienen una tabla de la nota más grave desde la que cada intervalo suena claro, los **límites de intervalo en el grave**, y una forma que baja de ellos cuesta un poco más. Un poco, no mucho: el Sol al aire, `320003`, empieza con Sol2 y Si2, una tercera por debajo de su límite (Si♭2), y sigue siendo el Sol que toca todo guitarrista. Su sonido es parte del sonido de la guitarra.

Con esas reglas, la búsqueda encuentra los acordes al aire de los libros de acordes (C `x32010`, G `320003`, D `xx0232`, Am `x02210`, G7 `320001`, Cmaj7 `x32000`), acordes con cejilla en otras tonalidades (F `133211`, la forma de Mi al aire un traste más arriba, y B `x24442`, la forma de La) y una forma para cada tipo de acorde en cada tonalidad.

Un guitarrista no golpea las cuerdas todas juntas. La púa las barre: en un **golpe hacia abajo** de las cuerdas graves a las agudas, en un **golpe hacia arriba** de vuelta desde las agudas, y la mano que sube rara vez llega a las graves. Los ritmos de la biblioteca

ya marcan los golpes ligeros «hacia arriba», entre los pulsos. La guitarra toca cada golpe hacia abajo en todas las cuerdas, de grave a agudo, con un tick entre una y otra (unos 5 ms a 120 BPM, así que un rasgueo de seis cuerdas dura unos 25 ms, como uno de verdad), y cada golpe hacia arriba en las cuatro agudas, de agudo a grave. Cómo el swing mueve un rasgueo entero está en Swing.

Lo que toca el guitarrista de cada estilo:

- **Acordes rasgueados** (pop, folk, country): la forma completa en el ritmo del estilo.
- **Power chords** (rock, punk, metal): solo la fundamental, su quinta y la octava, `A5` = La2 Mi3 La3. La distorsión añade notas nuevas en las sumas y las diferencias de las frecuencias tocadas. Para una quinta, cuyas frecuencias están cerca de 3:2, caen alineadas con la fundamental, una o dos octavas por debajo; para una tercera caen en notas desafinadas respecto del acorde, que zumba y se agria. Así que los guitarristas de rock dejan la tercera al bajo y a quien canta, y «A5» quiere decir «La sin su tercera».
- **Cuatro por compás** (swing): acordes de tres notas, la fundamental, la tercera y la séptima, golpeados en cada pulso, el segundo y el cuarto un poco más fuerte, como Freddie Green marcó el tiempo en la banda de Count Basie durante casi cincuenta años. Las formas son los **shells** del jazz: desde una fundamental en la cuerda grave de Mi, la séptima viene después y la tercera una octava más arriba (G7 = Sol2 Fa3 Si3); desde la cuerda La, la tercera y después la séptima (C7 = Do3 Mi3 Si♭3).
- **El shuffle de boogie** (blues, rock and roll): solo dos cuerdas, la fundamental con su quinta en un pulso y con su sexta en el siguiente, las dos corcheas de cada pulso: La y Mi, La y Mi, La y Fa♯, La y Fa♯. Sobre un acorde menor se queda en la quinta.
- **El chop** (reggae, soul, Motown, bluegrass): un acorde corto en las cuatro cuerdas agudas en los pulsos 2 y 4, con la caja, dejando las graves al bajo. El country lo llama el «chick» del «boom-chick». En el **ska** la guitarra toca entre todos los pulsos.
- **Rascar** (funk, disco): semicorcheas en una guitarra apagada, algunas acentuadas, la mayoría solo un «chk» percusivo.

Las **cuerdas** sostienen cada acorde mientras dura, una octava por encima del piano, suavemente: un colchón. Las baladas usan las cuerdas de ataque lento, los estilos sintetizados las de sintetizador. El **órgano** también sostiene acordes, como un órgano de iglesia en un coral o un Hammond en un blues lento. En el reggae toca el **burbujeo**: acordes cortos entre todos los pulsos en el grave, y en los pulsos 2 y 4 una octava más arriba, con el chop de la guitarra.

Cada músico toca un instrumento General MIDI elegido por el estilo. La tabla General MIDI los numera desde 1, y los mensajes MIDI desde 0, así que el código dice 24 para la guitarra de nailon:

Programa (desde 0)	Instrumento	Estilos
24	Guitarra acústica (nailon)	bossa nova, samba, bolero, folclore, rumba flamenca
25	Guitarra acústica (acero)	pop, folk, country, bluegrass, vals
26	Guitarra eléctrica (jazz)	swing
27	Guitarra eléctrica (limpia)	shuffle, rock and roll, surf, Motown, reggae, ska
28	Guitarra eléctrica (apagada)	funk, disco
29, 30	Guitarra saturada, con distorsión	rock, punk, metal
48, 49, 50	Cuerdas, cuerdas lentas, cuerdas sintetizadas	pop, tango, baladas, R&B, trance
16, 17, 18, 19	Órgano de tiradores, percusivo, de rock, de iglesia	blues, reggae, rock, coral

Músicos en arrays

La banda guarda un canal y una huella por músico, en arrays indexados por el músico:

```
pub const COUNT: usize = 8;

pub const ALL: [Player; Player::COUNT] = [
    Player::Piano,
    Player::Bass,
    Player::Drums,
    Player::Guitar,
    Player::Strings,
    Player::Organ,
    Player::Soloist,
    Player::Horns,
];
```

```
pub(crate) channels: [Option<usize>; Player::COUNT],
```

`band.channels[player.index()]` responde «¿qué canal?» para cada músico, e `index` es `self as usize`: un enum sin campos se convierte en su posición. El largo del array es parte de su tipo, `[Option<usize>; 8]`, así que el compilador comprueba que `ALL` liste de verdad `COUNT` músicos. `write_band` recorre `Player::ALL`, y todo lo que viene después, los patrones, las pistas y las huellas que mantienen congelada una parte editada, funciona igual para cada músico.

Quién toca en cada estilo: una tabla de constantes

Los músicos que añade cada estilo son un valor de este tipo:

```
pub struct Extra<Part> {
    pub part: Part,
    pub program: u8,
    pub only_in_b: bool,
}
```

```
pub struct Extras {
    pub guitar: Option<Extra<GuitarPart>>,
    pub strings: Option<Extra<PadPart>>,
    pub organ: Option<Extra<PadPart>>,
}
```

`Extra` es **genérico**: `Part` es un parámetro de tipo, que se completa como `GuitarPart` para la guitarra y como `PadPart` para las cuerdas y el órgano. Los tres campos «qué toca, en qué instrumento, espera a la b» se escriben una sola vez para los dos. Los estilos se listan en una tabla que construye el compilador:

```
const NONE: Extras = Extras {
    guitar: None,
    strings: None,
    organ: None,
};

const fn with_guitar(part: GuitarPart, program: u8) -> Extras {
    Extras {
        guitar: guitar(part, program),
        ..NONE
    }
}
```

```
const BY_STYLE: &[(&str, Extras)] = &[
    // Pop: a strummed acoustic guitar, strings for the chorus.
    (
        "Pop beat",
        Extras {
            guitar: guitar(STRUMMED, STEEL_GUITAR),
            strings: pad_in_b(STRINGS),
            ..NONE
        },
    ),
    // ...
];
```

Una `const fn` se puede llamar donde Rust necesita un valor en tiempo de compilación, como en un `const`. `..NONE` es la sintaxis de actualización de structs: los campos no escritos toman sus valores de `NONE`, así que cada estilo nombra solo los músicos que tiene. La prueba `every_style_listed_is_a_band_style` comprueba que cada nombre de la tabla es un estilo de banda de verdad, para que una errata no deje a un estilo sin su guitarra sin que nadie lo note.

Enums que llevan datos, y una función como parámetro

Lo que toca la guitarra es un enum cuyas variantes pueden llevar un valor:

```
pub enum GuitarPart {
    Strum(Comping),
    StylesRhythm,
    Chop,
    Power(Comping),
    FourToTheBar,
    Boogie,
}
```

`Strum(Comping::Bossa)` y `Strum(Comping::Waltz)` son una variante con distintos ritmos dentro. Un `match` puede mirar dentro, con patrones `|` anidados, como en la variación b, donde los acordes sostenidos y punteados pasan a ser rasgueados:

```
pub fn in_variation(self, variation: Variation) -> GuitarPart {
    match (variation, self) {
        (Variation::B, GuitarPart::Power(Comping::Held)) =>
GuitarPart::Power(Comping::Pulse),
        (
            Variation::B,
            GuitarPart::Strum(Comping::Held | Comping::ArpUp |
Comping::ArpUpDown),
        ) => GuitarPart::Strum(Comping::Strum),
        (_, part) => part,
    }
}
```

Los acordes rasgueados, los power chords y los shells solo se diferencian en qué notas recibe cada armonía; tocar el ritmo y rasguearlo es lo mismo. Así que `strummed` recibe la disposición como parámetro:

```
fn strummed(
    rhythm: Comping,
    bars: &[Bar],
    harmonies: &[Played],
    voice: fn(&Played) -> Vec<Note>,
) -> Vec<NoteEvent> {
```

`fn(&Played) -> Vec<Note>` es un tipo de **puntero a función**: cualquier función corriente con esa firma encaja, y quien llama pasa `shape`, `power_chord`, `shell` o `top_strings` por su nombre. Encontrar una forma prueba miles de combinaciones de trastes, y una canción repite sus acordes, así que `strummed` recuerda las formas que encontró en un `HashMap`:

```
let voicing = shapes.entry(key).or_insert_with(|| voice(harmony)).clone();
```

`entry` encuentra el lugar para `key` en el mapa, y `or_insert_with` llama al closure solo si ese lugar está vacío: cada acorde distinto se busca una sola vez.

Probar cada traste: recursión con un callback

Cuántas cuerdas usa una forma no se sabe de antemano, así que la búsqueda no puede ser un número fijo de bucles `for` anidados. Es una función que elige un traste para una cuerda y se llama a sí misma para la siguiente:

```
fn try_frets(
    choices: &[Vec<u8>],
    low: usize,
    top: usize,
    frets: &mut Vec<u8>,
    found: &mut impl FnMut(&[u8]),
) {
    let string = low + frets.len();
    if string > top {
        found(frets);
        return;
    }
    for &fret in &choices[string] {
        frets.push(fret);
        try_frets(choices, low, top, frets, found);
        frets.pop();
    }
}
```

`frets` es un único `Vec` compartido por todos los niveles: un nivel añade el traste de su cuerda, deja que terminen los niveles más profundos, y lo vuelve a sacar antes de probar el siguiente. Cuando todas las cuerdas tienen un traste, `found` recibe la forma entera. `found` es un closure que la puntúa y se queda con la mejor, así que cambia algo de fuera de sí mismo: eso lo hace un `FnMut`, y se pasa como `&mut` para que cada nivel de la recursión llame al mismo closure, en lugar de que cada uno tome el suyo.

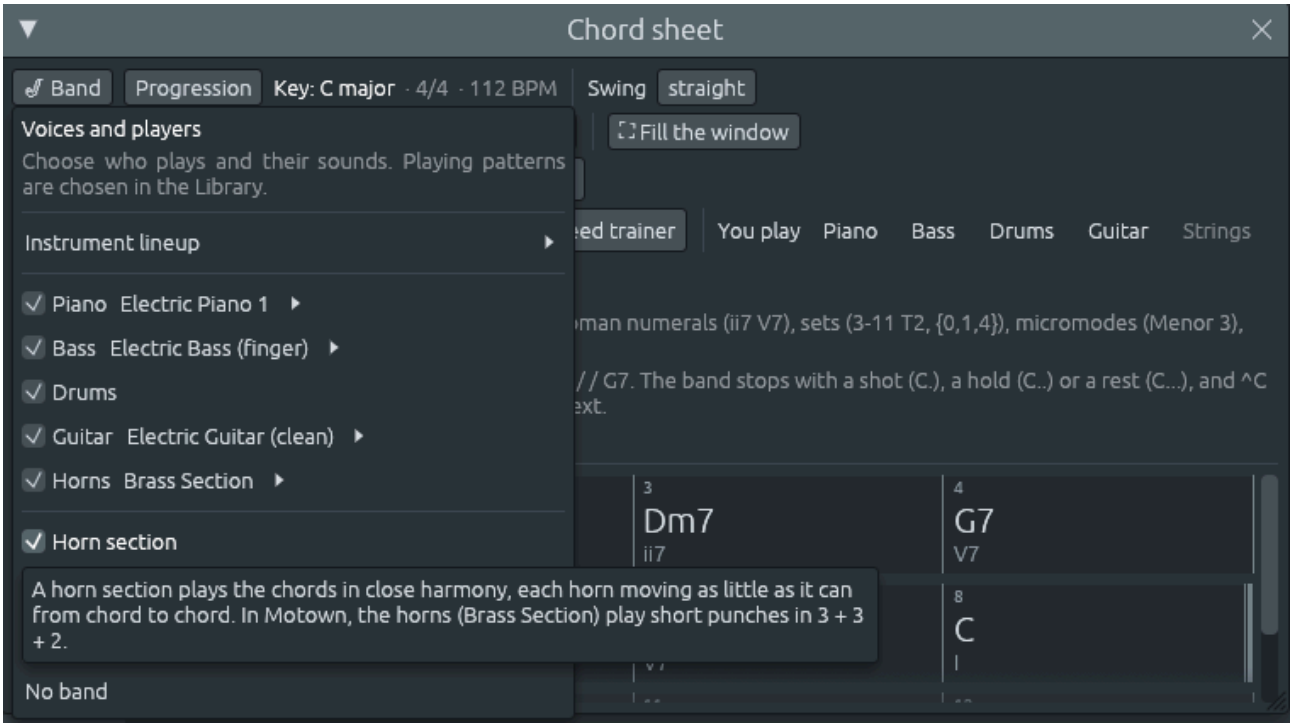
Una sección de vientos

Los discos de soul y de funk tienen una. También las big bands, las orquestas de salsa, las bandas de ska y las bandas de metales de los Balcanes: trompetas, saxofones y trombones tocando los acordes juntos. Marca **Sección de vientos** en el menú Banda, y entra con la banda una pista **Vientos**, que toca la hoja de acordes con una parte que le va al estilo:

Estilo	Qué tocan los vientos
Baladas (Balada pop, Balada de jazz, Blues lento, Neo soul, R&B, Bossa nova, Bolero…)	notas largas sostenidas debajo de la banda
Swing, shuffle, blues, rock and roll	golpes de big band en el pulso 1 y justo antes del 3 (el ritmo charlestón)
Pop, rock, soul, hip hop, latino, second line	golpes cortos en 3 + 3 + 2 (el tresillo)

Estilo	Qué tocan los vientos
Funk	ataques secos de funk en semicorcheas
House y otros estilos electrónicos	ataques sincopados, como en la música house
Reggaeton, moombahton	acordes cortos en el 3 + 3 + 2 del dembow
Ska	acordes cortos entre los pulsos
Marcha, vals, polca, danzas balcánicas	contratiempos, el «pa» del «um-pa» de una banda de metales
Tango, folk, clásica, reggae, estilos sin batería	notas largas
Otros estilos en 3/4 o 6/8	notas largas
Otros estilos en 5/4, 7/8 y otros compases irregulares	un golpe en cada pulso en que se siente el compás

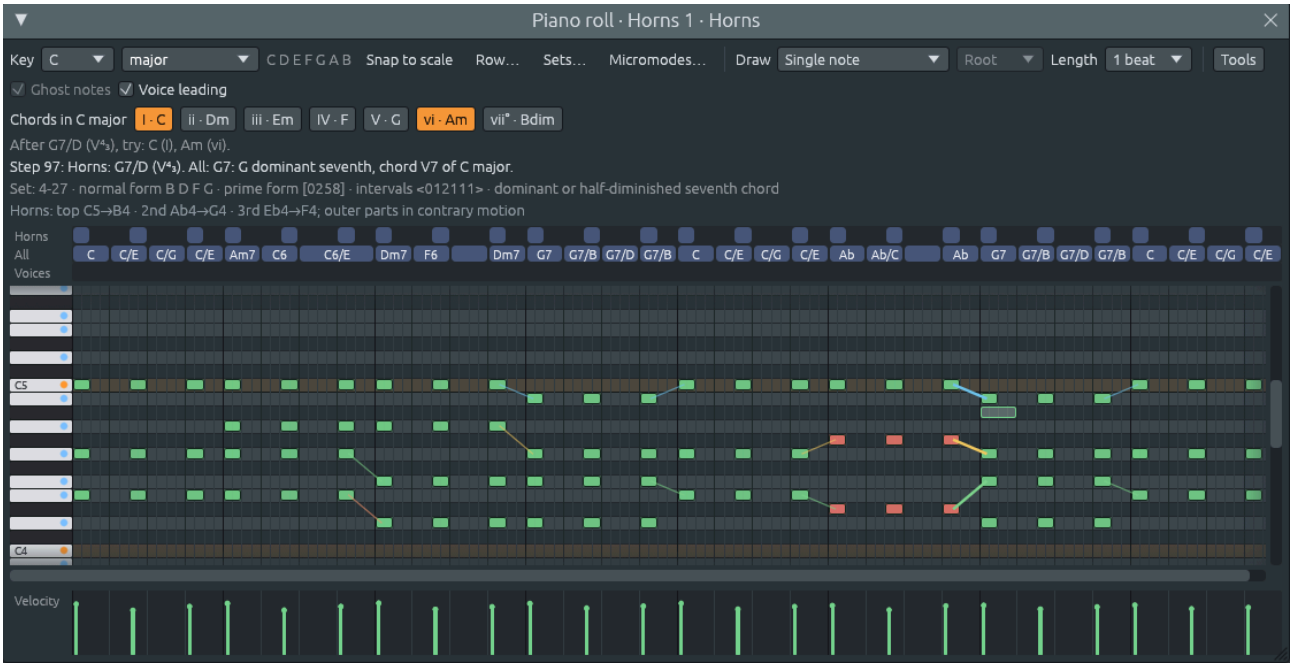
El ritmo **3 + 3 + 2** divide las ocho corcheas de un compás de 4/4 en grupos de tres, tres y dos, con un acento al comienzo de cada uno: en el pulso 1, en el «y» del 2 y en el pulso 4. Tocado así, el compás empuja hacia adelante. Es el tresillo de la música cubana, y está en todas partes en las líneas de vientos del pop y del soul. Los **contratiempos** son los de una banda de metales: en una marcha en 2/4 la tuba toca «um» en el pulso y los vientos «pa» entre medio; en un vals el bajo tiene el pulso 1 y los vientos los pulsos 2 y 3. El instrumento es **Sección de metales** (General MIDI 61) en la mayoría de los estilos, **Corno francés** en los clásicos y **Metales sintetizados** en los electrónicos.



Armonía cerrada, moviéndose lo menos posible

Los vientos que tocan acordes juntos los disponen en **posición cerrada**: las notas del acorde tan juntas como se pueda, dentro de una octava, debajo del viento más agudo, la **primera voz** (*lead*). Los arreglistas escriben así las partes de vientos porque hace que tres o cuatro instrumentos suenen como uno grande. Los vientos tocan las notas que toca el piano en el acorde, en la octava que sea: tantos vientos como notas distintas tiene la disposición del piano, hasta cuatro. Un acorde de más de cuatro notas, como C13 (Do Mi Sol Si♭ Re La), deja fuera primero su quinta, y después su fundamental, que el bajo toca de todos modos: los vientos tocan Mi, Si♭, Re, La, lo que hace que el acorde suene como suena. La primera voz se queda entre Fa4 y Fa5, donde una sección de trompetas suena brillante sin forzar.

De un acorde al siguiente, los vientos toman la disposición cerrada que menos se mueve: cada uno va a la nota más cercana que puede. Do a La♭ es Mi4 Sol4 Do5 a Mi♭4 La♭4 Do5: dos vientos se mueven un semitono, abajo y arriba, y la primera voz sostiene el Do. La♭ a G7: la primera voz baja de Do5 a Si4, la segunda de La♭4 a Sol4, la tercera sube de Mi♭4 a Fa4, y una cuarta entra en Re4 para el acorde de séptima. Después, de G7 a Do, Si4 sube a Do5 y Fa4 baja a Mi4, como hacen la sensible y la séptima de una dominante.



Los vientos también evitan las **quintas paralelas** cuando pueden. Dos vientos a una quinta justa que pasan a otra quinta justa suenan como un solo instrumento, y la textura pierde una línea (ver Conducción de voces). Así que cada par de vientos que se movería en quintas paralelas cuenta como 24 semitonos de movimiento, más que cualquier otro camino al acorde siguiente, y los vientos eligen otra disposición.

En el código

Lo que tocan los vientos son notas largas, o acordes cortos en el ritmo de una de las maneras de hacer comping del piano:

```
pub enum HornPart {
    /// Long notes: each chord held for as long as it lasts.
    Pads,
    /// Short chords in the rhythm of a way of comping.
    Hits(Comping),
}
```

`Comping` ya tenía todos los ritmos que necesitan los vientos (Charleston, Tresillo, Funk, Stabs, Dembow, Offbeat, Aksak para los contratiempos y Accents), así que los vientos los reutilizan. `HornPart::of` lee el estilo: primero dos listas de nombres de estilos, después el compás, después el género:

```
if style.rhythm.is_none() || HELD.contains(&style.name) {
    return HornPart::Pads;
}
if let Some(&(_, comping)) = BY_STYLE.iter().find(|(name, _)| *name ==
style.name) {
    return HornPart::Hits(comping);
}
```

`iter().find` devuelve un `Option<&(&str, Comping)>`, y el patrón `Some(&(_, comping))` desarma el par de una vez: `&` para la referencia, `_` para el nombre que no necesita. `close` construye una disposición de arriba abajo, empezando en la primera voz y tomando la nota más cercana por debajo que sea una nota del acorde que ningún viento toque todavía:

```
fn close(lead: u8, classes: &[u8]) -> Vec<u8> {
    let mut notes = vec![lead];
    let mut above = lead;
    while notes.len() < classes.len() {
        let Some(below) = (1..12)
            .filter_map(|down| above.checked_sub(down))
            .find(|note| {
                classes.contains(&(note % 12)) && notes.iter().all(|other|
other % 12 != note % 12)
            })
        else {
            break;
        };
        notes.push(below);
        above = below;
    }
    notes.reverse();
    notes
}
```

`checked_sub` da `None` en lugar de dar la vuelta por debajo de la nota 0, y `filter_map` los descarta. `choose` prueba cada primera voz de Fa4 a Fa5 que sea nota del acorde, construye la disposición cerrada debajo y se queda con la más barata:

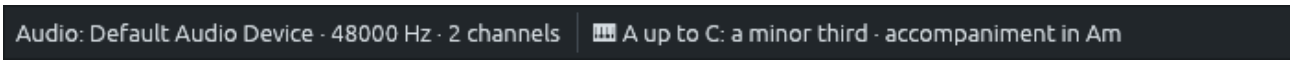
```
fn choose(classes: &[u8], last: Option<&[u8]>) -> Vec<u8> {
    (LEAD_LOW..=LEAD_HIGH)
        .filter(|lead| classes.contains(&(lead % 12)))
        .map(|lead| close(lead, classes))
        .min_by_key(|voicing| cost(voicing, last))
        .unwrap_or_default()
}
```

`cost` suma cuánto se mueve cada viento, emparejándolos de arriba abajo, más 24 por cada par en quintas paralelas. `min_by_key` devuelve la primera de las más

baratas, y `unwrap_or_default` da una disposición vacía, silencio, para una armonía sin notas. Se prueban como mucho trece primeras voces, cada una con cuatro vientos como mucho, así que probarlas todas es rápido; no hace falta nada más ingenioso.

Seguir los acordes que mantienes

Un **teclado arreglador**, el teclado doméstico con un botón de «estilo», toca batería, acordes y una línea de bajo, y la mano izquierda solo le dice *qué acorde*. Mantén Do y la banda toca en Do; pasa a La menor y el mismo groove sigue en La menor. **MIDI > El acompañamiento sigue los acordes** hace eso con cualquier canción: tócala, mantén un acorde en un teclado MIDI (o en las teclas de letras, con **Teclado del equipo como piano**), y desde la siguiente nota que toquen, los acordes y la línea de bajo pasan a él. La barra de estado dice en qué acorde está el acompañamiento.



Como los teclados arregladores, no necesita el acorde entero:

Teclas pulsadas	Acorde
una tecla, como Sol	su acorde mayor: Sol
dos teclas a una tercera menor, como Re y Fa	el acorde menor sobre la de abajo: Re menor
dos teclas a una tercera mayor o a una quinta	el acorde mayor sobre la de abajo
dos teclas a una séptima menor, como Sol y Fa	el acorde de séptima: G7
tres teclas o más	el acorde que forman

El último acorde se queda cuando sueltas, como hace el botón de «memoria» de un teclado arreglador. Solo una tecla que *baja* pide un acorde nuevo, así que soltar las teclas una a una no cuenta como tocar las que quedan.

Antes de poder mover una nota, la aplicación necesita saber para qué acorde se escribió. Oye cada patrón una vez, con el segundo oído de Oír la armonía, y guarda lo que oyó hasta que cambia el proyecto. Después cada nota de un tramo oído como, por ejemplo, Do mayor se mueve al acorde mantenido, por ejemplo La menor. Los teclados arregladores tienen una tabla para esto, escrita por quien hizo el estilo; Ferrous Lineup lo calcula a partir de lo que *hace* cada nota en su acorde, su función:

```
enum Role {
    Root,
    /// The third, or what stands for it in a "sus" chord.
    Third,
    Fifth,
    /// The seventh, or the sixth of a sixth chord.
    Seventh,
    /// A ninth, eleventh or thirteenth.
    Tension,
}
```

Los acordes en bloque, tres notas o más que empiezan juntas, se disponen de nuevo para el acorde nuevo tan cerca como se pueda de donde estaban, así que la mano casi no se mueve. Las notas sueltas, como en los arpegios y las líneas de bajo, conservan su función: la fundamental de Do va a la fundamental de La menor, la tercera (Mi) a la tercera (Do), la quinta (Sol) a la quinta (Mi); una séptima va a la séptima nueva, o a la fundamental si el acorde nuevo no tiene. Las notas fuera del acorde, como las notas de paso de un bajo caminante, se mueven tanto como la fundamental. Cada nota movida cae en la octava más cercana a donde estaba, lo que mantiene grave una línea de bajo y con la misma forma un arpegio:

```
fn nearest(pitch_class: u8, near: Note) -> Note {
    let near = i32::from(near.midi());
    let up = (i32::from(pitch_class) - near).rem_euclid(12);
    // Up to a tritone up, or else down.
    let midi = if up <= 6 { near + up } else { near + up - 12 };
    Note::new(midi.clamp(0, 127) as u8)
}
```

`rem_euclid` es el resto que nunca es negativo: `-3 % 12` es `-3` en Rust, pero `(-3i32).rem_euclid(12)` es `9`, que es lo que quiere decir «cuánto hay que subir hasta el próximo La».

La canción misma nunca cambia: deshacer, guardar y exportar a WAV ven las notas como están escritas. Las notas movidas van solo al hilo de audio, con el comando que ya usa cada edición para reemplazar las notas de un canal en un patrón (ver Enviar comandos con un ring buffer):

```
self.send(Command::SetNotes {
    pattern: moved.pattern,
    channel: moved.channel,
    notes: swing::swung(&moved.notes, swing, hits),
});
```

El hilo de audio cambia la lista entre dos bloques de sonido, así que la siguiente nota que empieza ya está en el acorde nuevo; las notas que ya suenan siguen como

estaban. Cuando se detiene la reproducción o se desactiva la opción, se vuelven a enviar las notas propias del proyecto. «Solo una tecla que baja» es una bandera, que se pone en cada Note On y se lee con `std::mem::take`, que la devuelve y deja `false` en su lugar:

```
if !std::mem::take(&mut self.live.follow.key_went_down) {
    return;
}
```

En el archivo

Un proyecto con banda guarda su estilo por nombre, dónde termina, sus canales, las huellas y lo que silenció. Una canción nueva de **Archivo > Nuevo** con C, Am, F y G7 en su hoja guarda esto:

```
band: Some((
        style: "Pop beat",
        end: 64,
        piano: Some(0),
        bass: Some(1),
        guitar: Some(2),
        strings: Some(3),
        written: (Some(14171519900600248065), Some(11668178586522433030),
Some(3477506964485452990)),
        written_too: (Some(13661597678858777105),
Some(14695981039346656037), None),
        heard: Some(6239919094066095261),
        muted: (),
        since: 0,
    )),
```

y cada patrón de la banda lleva `player: Some(Piano)` y así. Las huellas están repartidas de manera rara: `written` tiene tres, `written_too` tres más. La banda en memoria tiene un único array de ocho, pero el archivo tiene que seguir pudiendo leerse con versiones anteriores donde se pueda. Una banda de piano, bajo y batería se guardaba como `written: (…, …, …)` antes de que hubiera otros músicos, así que esas tres siguen donde estaban, y las demás van en campos propios que se omiten cuando están vacíos:

```
/// The fingerprints of the piano, the bass and the drums.
pub written: [Option<u64>; 3],
/// The fingerprints of the guitar, the strings and the organ, left
/// out when there are none, so a band of three reads as before.
#[serde(default, skip_serializing_if = "none_written")]
pub written_too: [Option<u64>; 3],
/// The fingerprint of the soloist's part.
#[serde(default, skip_serializing_if = "Option::is_none")]
pub written_solo: Option<u64>,
```

`skip_serializing_if` deja un campo fuera del archivo cuando la función nombrada lo dice, y `default` lo completa cuando un archivo no lo tiene. Convertir el array en los campos del archivo es un solo patrón:

```
let [piano, bass, drums, guitar, strings, organ, soloist, horns] =
band.written;
```

Un patrón de array desarma un array en nombres, uno por elemento, y el compilador comprueba que sean exactamente ocho. Así una banda de tres se escribe exactamente como antes, y su canción conserva el formato que necesitaba; una banda necesita el formato 13, una banda más grande el 17, un solista el 19, una sección de vientos el 28 y la batería de otro estilo el 29, guardada como `drums: Some(Of("Bluegrass"))` o `drums: Some(Song)` (ver El número de versión), y una versión anterior los rechaza con un mensaje en lugar de perder un músico sin decir nada. Un archivo cuya banda toca un estilo que esta versión no conoce, o en un canal que no existe, se rechaza con el motivo en lugar de abrirse mal.

Cómo se prueba

`band.rs` tiene las pruebas más amplias de la banda: `a_new_song_is_an_empty_sheet_with_a_band`, `the_band_plays_what_is_typed`, `the_band_follows_every_change_of_the_sheet`, `an_edited_part_stays_until_it_is_played_again`, `a_frozen_part_stays_frozen_in_the_file`, `sets_micromodes_and_rows_are_played_by_their_notes`, `no_chord_silences_every_style`, `a_style_in_three_puts_the_song_in_three`, `a_walking_bass_leads_into_each_chord_by_a_semitone` (en Swing y en un vals de jazz), `a_band_in_a_demo_mutes_the_songs_own_accompaniment`, `deleting_a_players_channel_freezes_only_that_part`, `long_songs_are_cut_into_patterns_and_repeats_share_one`, y para las partes del estilo `instruments_can_change_without_changing_the_performance`,

`the_progression_is_independent_of_the_rhythm_and_sounds` y `separate_style_components_survive_saving_and_undo`. `every_rhythm_is_a_band_style_and_names_are_unique` lee la lista entera. En `comping.rs`, `a_walking_line_leads_into_the_next_chord` comprueba Do2, Mi2, Sol2, Fa♯2 sobre C7 yendo a F, `a_walking_line_goes_the_way_of_the_next_chord` las cuatro líneas de C, Am, F, G7 y que, de cada acorde a cada fundamental, el último paso sea de medio tono y ningún paso más ancho que una sexta, y `plain_accents_follow_the_grouping_without_a_styled_bass_line` los pulsos de un compás irregular.

Las pruebas de la guitarra comprueban la música: `open_chords_are_the_ones_in_chord_books`, `other_keys_take_barre_chords`, `a_slash_chord_has_its_bass_lowest`, `every_chord_in_every_key_has_a_shape_with_its_needed_notes`, `power_chords_and_shells`, `a_down_stroke_goes_up_the_strings_and_an_up_stroke_comes_down`, `a_chop_strikes_the_top_four_strings_on_two_and_four` y `the_boogie_alternates_fifth_and_sixth`. Las de los vientos: `the_horns_move_as_little_as_they_can` (Do, La♭, G7, Do como más arriba), `the_horns_never_move_in_parallel_fifths_when_they_can_help_it`, `a_big_chord_leaves_out_its_fifth_and_root` y `every_style_has_horns_that_fit_it`; y en `band.rs`, `horns_come_to_any_style_when_asked` los lleva a seis estilos y comprueba que cada nota de los vientos es una nota de su acorde, que cada disposición está en posición cerrada y que la primera voz está entre Fa4 y Fa5.

Seguir un acorde mantenido se comprueba con `keys_ask_for_chords`, `notes_keep_their_role`, `a_seventh_goes_to_the_root_of_a_triad`, `every_style_follows_any_chord_in_its_chords` (para cada estilo, el canal de acordes toca solo notas del acorde mantenido) y `every_demo_song_follows_a_chord`. Y la traza de reproducción de una canción con banda lista cada nota que toca cada músico, compás por compás, que es como se oye un cambio en la banda sin escuchar.

Los ejercicios sobre la banda están en Ejercicios: la banda.

Una orquesta para música que cambia de compás

Archivos: `crates/lineup-core/src/library/rhythm.rs`, `crates/lineup-core/src/library/style.rs`, `crates/lineup-core/src/library/apply.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/band/following.rs`, `crates/lineup-app/src/app/library.rs` y `crates/lineup-app/src/library_browser.rs`

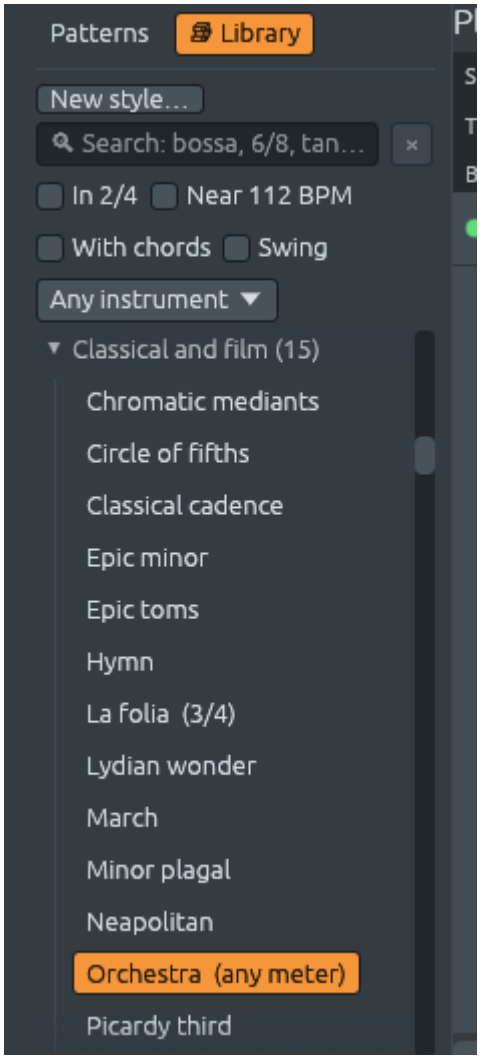
Cada estilo de la Biblioteca está escrito en un solo compás, y aplicarlo pone la canción en ese compás (ver Pasar una canción a otro compás). Eso está bien para una canción pop, y está mal para La consagración de la primavera. El **Juego del rapto** cambia de compás 32 veces en 67 compases: un estilo en 4/4 lo convertiría todo en 4/4, y una banda que silencia los acordes y el bajo propios de la canción silenciaría a Stravinsky.

El estilo **Orquesta** está hecho para esa música. Toca cada compás en la métrica de ese compás, conserva los compases y el tempo de la canción, y toca junto a la partitura en lugar de silenciarla. Es el estilo que encaja con once demostraciones hechas a partir de partituras de orquesta (ver De una interpretación MIDI o una partitura a una demostración): Promenade, el vals en cinco de la *Sinfonía Patética*, Boléro, La Mer, los Augurios primaverales, el Juego del rapto y cinco escenas de Petrushka.

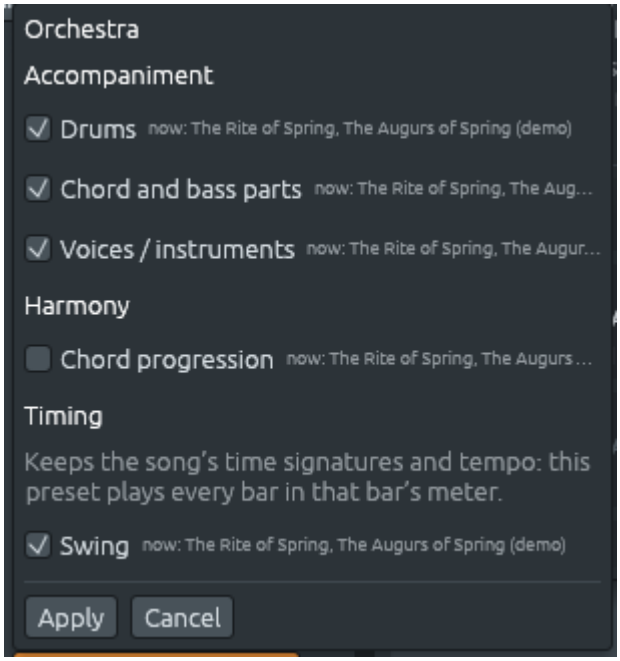
Probarlo

1. Abre **Archivo > Demostraciones**, escribe `augurios` en el cuadro de búsqueda y pulsa Intro: **La consagración de la primavera, Augurios primaverales** se abre y suena. Detenla.

2. Mira la pestaña **Biblioteca**. **Orquesta (cualquier métrica)** está iluminado en el grupo **Clásica y de cine**, como lo está siempre un estilo que encaja. «Cualquier métrica» dice que el estilo no está en ningún compás propio.



3. Hazle clic. Bajo **Acompañamiento**, **Batería**, **Partes de acordes y bajo** y **Voces / instrumentos** están marcadas; bajo **Armonía**, **Progresión de acordes** no lo está. Bajo **Tiempo** no hay casillas Compás ni Tempo, solo una línea que dice que el preajuste conserva los compases y el tempo de la canción.



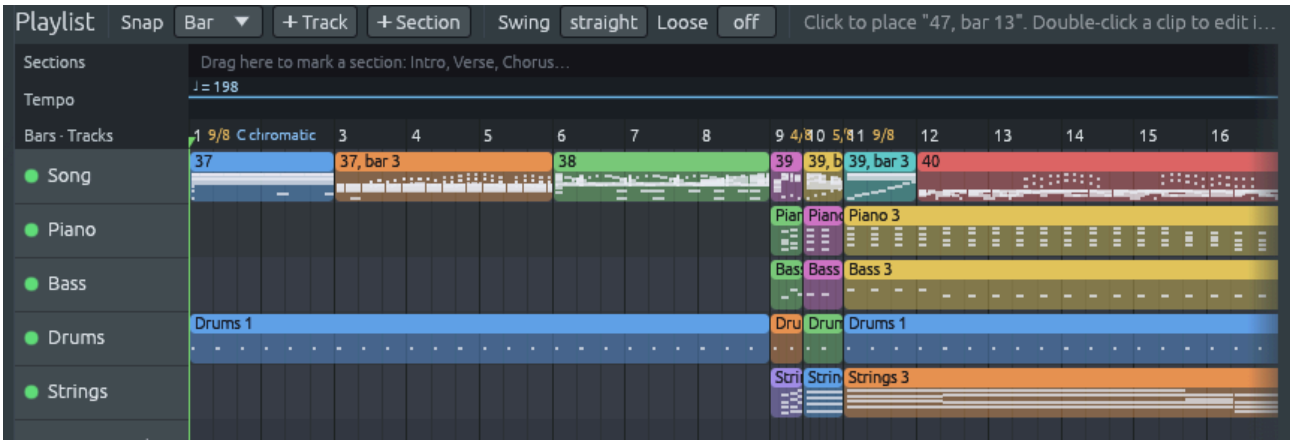
4. Haz clic en **Aplicar**. Aparecen cuatro pistas bajo la canción: **Piano**, **Bajo**, **Batería** y **Cuerdas**. La pista Canción sigue encendida, el cuadro Compás sigue diciendo 2/4 y el tempo sigue diciendo 112.

5. Pulsa Reproducir. Las cuerdas mantienen cada acorde bajo las cuerdas que martillean de Stravinsky. Los cornos, los timbales y el bombo se quedan callados hasta el compás 3, y luego golpean *con* los acentos de la partitura: fuera del pulso dos veces en el compás 3, fuera del pulso en el compás 5, en el pulso en los compases 6 y 7, fuera de nuevo en el 8. En los compases tranquilos 9 a 12 descansan, y las cuerdas tocan suave.



Las pistas conservan los nombres de los músicos de la banda, pero en este estilo la pista Piano toca cornos franceses y la pista Bajo timbales; el **Mezclador** muestra los instrumentos.

Ahora haz lo mismo con **La consagración de la primavera, Juego del rapto**, que empieza en 9/8 a la negra con puntillo = 132 de Stravinsky (el cuadro de tempo muestra 198, la misma velocidad contada en negras) y pasa por 4/8, 5/8, 12/8, 6/8, 7/8, 3/4, 2/8 y más. Las pistas se cortan en clips cortos, uno por cada tramo de compases en una misma métrica, y cada clip toca sus compases como se siente esa métrica.



La orquesta toca los acordes que la aplicación reconoció en la partitura, los de la hoja de acordes (**F5**). Donde no se reconoció ningún acorde, solo toca el bombo. Si marcas **Progresión de acordes** en el popup, toca sus propios acordes en su lugar, I y ♭VII del modo mixolidio, como la progresión de un estilo siempre reemplaza los acordes de la canción.

Solo las voces

Desmarca **Batería** y **Partes de acordes y bajo** antes de **Aplicar**, de modo que solo **Voces / instrumentos** quede marcado, y entran las mismas cuatro pistas. Marca solo la batería, o solo las partes de acordes y bajo, y también entran: una orquesta que sigue a la canción siempre trae sus propios músicos y sus propios patrones, en pistas propias, y los canales de la partitura conservan sus instrumentos y sus notas.

Los demás estilos tratan una sola casilla de otra manera (ver Qué partes tomar). Elegido con sus partes de acordes y bajo y sus voces pero sin su batería, un estilo trae su banda en pistas propias, y la batería propia de la canción sigue sonando. Elegido solo con sus voces, no trae banda: sus sonidos van a los músicos propios de la canción. Elegido solo con su batería o solo con sus partes de acordes y bajo, se abre en un patrón de prueba para que lo escuches primero.

La música

Pulsos sentidos

Un compás de 9/8 tiene nueve corcheas, pero nadie cuenta nueve. Las corcheas vienen en tres grupos de tres, y se sienten tres pulsos, cada uno de una negra con puntillo. Un compás de 5/8 tiene cinco corcheas en dos pulsos, uno corto de dos y uno largo de tres. Un compás de 2/4 a tempo rápido pasa como un solo pulso. Estos son los pulsos *sentidos*, y `TimeSignature::felt_groups` los da para cualquier compás (ver Pulsos de distinta longitud).

Un director marca los pulsos sentidos, no cada corchea: en el Juego del rapto, tres pulsos para un compás de 9/8, dos para 5/8, uno para 2/8. En música cuyo compás cambia a cada rato, eso es lo que mantiene unida a una orquesta, y es lo que toca el estilo Orquesta:

Compás	Se siente como	Los cornos y los timbales golpean en la corchea (o negra)
2/8, 3/8, 2/4, 3/4	un pulso	1
4/8	2 + 2	1 y 3
5/8	2 + 3	1 y 3
6/8	3 + 3	1 y 4
7/8	2 + 2 + 3	1, 3 y 5
9/8	3 + 3 + 3	1, 4 y 7
12/8	3 + 3 + 3 + 3	1, 4, 7 y 10
4/4	2 + 2	negras 1 y 3

Los cornos golpean el acorde en cada uno de esos pulsos y los timbales tocan su fundamental; cuando un acorde cambia entre dos pulsos, tocan el nuevo donde empieza. La línea de batería propia del estilo golpea fuerte el primer pulso del compás y suave los demás, pero junto a una partitura el bombo conserva solo el primer pulso, y solo donde la música suena fuerte: un bombo en cada pulso de un compás tranquilo taparía la partitura.

Tocar junto a la partitura

Una banda toca el mismo groove compás tras compás. Una orquesta que se une a una sinfonía no puede: la música crece y se apaga, se detiene, y pone sus acentos

donde el compositor los quiere. Los Augurios primaverales son el caso famoso. Las cuerdas martillean el mismo acorde en cada corchea, todas igual de fuerte, y ocho cornos tiran acentos sobre algunas de esas corcheas en lugares que nadie puede predecir: la segunda y la cuarta corchea de un compás, la segunda de otro, la primera de los dos siguientes. El impacto de la pieza es que los acentos nunca se asientan en un patrón. Un bombo en cada primer pulso los convertiría en uno.

Así que la Orquesta escucha la partitura antes de tocar, y cambia lo que tocarían sus músicos:

- **Dinámica.** Toca tan fuerte como la partitura, compás a compás: más suave en un compás *piano*, más fuerte en uno *forte*. Donde la partitura calla, calla.
- **Acentos.** Donde la partitura acentúa un acorde, los cornos lo golpean, los timbales golpean su bajo y el bombo golpea con ellos, tan fuerte como la partitura.
- **Pasajes de acentos.** Donde los acentos vienen juntos, en al menos dos compases de nueve (un compás y los cuatro a cada lado), los cornos, los timbales y el bombo tocan *solo* los acentos, para que el patrón irregular siga siendo el del compositor.
- En todo lo demás se atienen a los pulsos sentidos, y las cuerdas mantienen los acordes.

¿Qué cuenta como acento? Un acorde que pasa las dos primeras pruebas de abajo, o una nota que pasa la tercera, tal como tú los oirías:

1. Toda la orquesta lo toca mucho más fuerte que el resto de su compás, *y* que el mismo lugar en los compases de alrededor: casi el doble de fuerte. Un acorde tutti después de una textura fina.
2. Un instrumento en él toca una nota corta, entre silencios: nada una corchea antes, nada una corchea después, y silencio cuando acaba la nota. Los cornos de los Augurios hacen eso. Una melodía de negras no (sus notas se tocan), y tampoco un acorde mantenido cuatro compases, que solo entra. Por eso el Villancico de las campanas, cuyo coro mantiene un acorde bajo la melodía, no tiene acentos, y su orquesta toca un golpe por compás desde el principio.
3. La partitura lo marca ella misma, un sforzando o un acento fuerte: muy fuerte (velocidad 120 o más) y una quinta parte más fuerte que el término medio de su parte en ese compás.

Un compás donde la mayoría de las corcheas contarían como acentos no tiene ninguno: la música simplemente subió de volumen.

Por qué cornos y timbales

El estilo necesita una parte que marque la armonía en el pulso y una parte que marque el bajo, los trabajos que hacen un piano y un bajo en una banda. En una orquesta esos trabajos son desde hace mucho de los cornos y los timbales: en una sinfonía de Haydn o de Beethoven, los acordes fuertes que cierran una frase son los cornos y las trompetas tocando la armonía mientras los timbales golpean la tónica o la dominante debajo. Las cuerdas mantienen cada acorde mientras dura, como hace el colchón de cuerdas de una banda (ver Una banda más grande).

La orquesta toca *junto* con la partitura. En una demostración pop, la banda silencia los acordes y el bajo propios de la canción, porque dos pianos tocando los mismos acordes con dos disposiciones distintas suenan a error. En La consagración, las partes de la canción *son* la pieza: silenciarlas dejaría solo a la banda. Así que la Orquesta entra por encima, y cada canal de la partitura conserva su propio silencio.

Cómo funciona

Un ritmo que sigue el compás

Un ritmo de la biblioteca está escrito en un solo compás: sus líneas de batería son compases de pasos, `X... .... o... ....` para un compás de 4/4, donde `X` es un golpe fuerte, `o` uno suave y `.` silencio. La banda extiende una línea compás tras compás, y en una canción con marcadores de compás escribe cada tramo de compases en una misma métrica como un patrón propio.

El ritmo de la Orquesta tiene una línea, ese compás de 4/4 para el bombo, y una bandera en `Rhythm`:

```
pub struct Rhythm {
    // ...
    pub follows_the_meter: bool,
}
```

Los ritmos se construyen con un `const fn rhythm(...)`, y un método constructor pone la bandera, igual que `swung` pone el swing:

```
const fn following_the_meter(self) -> Rhythm {
    Rhythm {
        follows_the_meter: true,
        ..self
    }
}
```

`..self` es la *struct update syntax*: cada campo no nombrado toma su valor de `self`. Un `const fn` puede hacerlo porque solo mueve valores, así que toda la tabla de ritmos se sigue construyendo en tiempo de compilación.

Las notas de batería de un compás salen de `Rhythm::notes_in(drum, time, length)`. Para un ritmo corriente extiende la línea compás a compás, como hace `Rhythm::notes`. Para uno que sigue el compás, primero escribe la línea de nuevo para `time`:

```
fn in_meter(line: &str, own: TimeSignature, time: TimeSignature) -> String
{
    let steps: Vec<char> = line
        .split('|')
        .next()
        .unwrap_or_default()
        .chars()
        .filter(|character| !character.is_whitespace())
        .collect();
    let first = steps.first().copied().unwrap_or('.');
    let other = felt_starts(own)
        .get(1)
        .and_then(|&start| steps.get(start).copied())
        .unwrap_or('.');
    let mut bar = vec!['.'; time.steps_per_bar()];
    for (index, start) in felt_starts(time).into_iter().enumerate() {
        if let Some(step) = bar.get_mut(start) {
            *step = if index == 0 { first } else { other };
        }
    }
    bar.into_iter().collect()
}
```

Lee el primer compás de la línea (una línea puede tener varios, separados por `|`), se queda con el golpe de su primer paso, `first`, y con el del comienzo de su segundo pulso sentido, `other`, y los vuelve a poner en los pulsos sentidos de `time`. `felt_starts(time)` convierte los grupos sentidos en pasos: un grupo de tres corcheas son tres veces `time.unit_steps()`, los pasos que dura una corchea. Para 9/8 eso da los pasos 0, 6 y 12, y la `X` y la `o` del compás de 4/4 caen ahí: `X.....o.....o.....`.

Un ritmo que sigue el compás no tiene compás propio, así que `Rhythm::for_time`, que elige un ritmo para el compás nuevo de una canción, nunca lo elige.

Un estilo que sigue a la canción

Un estilo sabe si sigue a la canción preguntándole a su ritmo:

```
pub fn follows_the_song(&self) -> bool {
    super::Rhythm::named(self.rhythm).is_some_and(|rhythm|
rhythm.follows_the_meter)
}
```

`Option::is_some_and` es `true` solo cuando hay un valor y el closure dice que sí para él, así que un estilo cuyo ritmo falta no sigue nada.

Aplicar un estilo puede poner la canción en el compás y el tempo del estilo, cuando esas casillas están marcadas. Las partes que se aplican, `StyleParts`, tienen un paso que quita las dos cuando el estilo sigue a la canción:

```
pub fn following(mut self, follows: bool) -> Self {
    if follows {
        self.time_signature = false;
        self.tempo = false;
    }
    self
}
```

Aplicar en la Biblioteca pasa por ahí, y `Project::set_band_style` pregunta `follows_the_song` antes de poner la canción en el compás del estilo, así que ninguno de los dos cambia el compás de La consagración. El menú **Estilo** de una sección en la hoja de acordes, que solo ofrece estilos en el compás propio del compás elegido, ofrece la Orquesta en cualquier métrica. En la aplicación, la misma bandera cambia las casillas de Tiempo por la línea que viste en el popup, y la Biblioteca añade «(cualquier métrica)» al nombre del estilo.

Unirse a la canción

Aplicar en la Biblioteca termina en `band_plays_parts`, en `crates/lineup-app/src/app/library.rs`. Primero decide si entra una banda:

```
let follows = match action {
    LibraryAction::Style(style) | LibraryAction::StyleInSong(style) => {
        style.follows_the_song()
    }
    LibraryAction::Rhythm(rhythm) | LibraryAction::RhythmInSong(rhythm) =>
{
        rhythm.follows_the_meter
    }
    _ => false,
};
let has_music = !self.project.playlist().chord_markers().is_empty()
    || self.project.playlist().clips().iter().any(|clip| {
        self.project
            .pattern(clip.pattern)
            .is_some_and(|pattern| !pattern.is_empty())
    });
// A beat made up in the library has no chord and bass parts of its
// own: it brings no band, only its drums.
let made_up = match action {
    LibraryAction::Rhythm(rhythm) | LibraryAction::RhythmInSong(rhythm) =>
{
        BandStyle::named(rhythm.name).is_none()
    }
    _ => false,
};
let joins = if follows {
    parts.rhythm || parts.matching_style || parts.instruments
} else {
    parts.instruments && parts.matching_style && has_music && !made_up
};
if self.project.band().is_none() && !joins {
    return false;
}
```

`match` con `|` entre dos patrones ejecuta el mismo brazo para los dos, y `_ => false` cubre cualquier otro tipo de elemento de la Biblioteca. `has_music` es verdadero cuando la canción tiene acordes en la hoja de acordes, o notas cuyos acordes la banda puede oír; `Iterator::any` se detiene en el primer clip cuyo patrón tiene una nota. Una banda toca las partes de acordes y bajo de un estilo, así que solo entra cuando se toman, con las voces en que las toca; la batería sola es cosa de los canales de batería de la propia canción. Cuando `band_plays_parts` devuelve `false`, quien la llamó sigue por el camino de siempre: abre el estilo en un patrón de prueba o, mientras la canción suena, lo aplica a los patrones propios de la canción.

La misma regla está en `Project::apply_band_parts`, en `band.rs`, que construye la banda, con una línea más: una banda que entró sin la batería del estilo se la deja a la canción.

```
if !parts.rhythm && !follows {
    band.drums = BandDrums::Song;
}
```

Escuchar la partitura

`crates/lineup-core/src/band/following.rs` es un módulo de la banda. `band.rs` lo declara con `mod following;`, y Rust lo encuentra en `band/following.rs`, una carpeta con el nombre del archivo padre. Su tipo es `pub(super)`: visible para `band.rs` y para nada más.

`Following::hear(project, bars)` recoge lo que toca la canción. Recorre los clips que se oyen, salta los patrones propios de la banda y anota el comienzo de cada nota en un `BTreeMap<usize, Vec<(usize, u8, usize)>>`: de paso a una lista de (canal, velocidad, duración). Un `BTreeMap` mantiene sus claves en orden, así que `struck.range(bar.start..bar.end())` da las notas de un compás, en orden de tiempo, sin mirar las demás.

Devuelve un `Option<Following>`: `None` cuando la canción no toca ninguna nota, una hoja de acordes sin nada más. Entonces no hay partitura que seguir, y la orquesta toca como lo escribió su estilo. Lo mismo vale para los compases antes de la primera nota de la partitura y después de que acabe la última:

```
let first = *struck.keys().next()?;
let last = struck
    .iter()
    .flat_map(|(&at, notes)| notes.iter().map(move |&(_, _, length)| at +
length.max(1)))
    .max()?;
```

El `?` después de `next()` y de `max()` devuelve `None` desde `hear` cuando no hay nada, igual que pasa un error hacia arriba desde una función que devuelve un `Result`. `flat_map` convierte la lista de notas de cada paso en los pasos donde terminan, un solo iterador largo, y `max` toma el último. El `move` delante del closure interior copia `at` dentro de él: el iterador interior se usa después de que el closure exterior haya vuelto, cuando su propio `at` ya no existe.

Qué tan fuerte suena un compás: la nota más fuerte en cada paso que toca, y la del medio de esas, dividida entre 100 (mezzoforte en las demostraciones).

```
fn middle(mut values: Vec<u32>) -> u32 {
    if values.is_empty() {
        return 0;
    }
    values.sort_unstable();
    values[(values.len() - 1) / 2]
}
```

El valor del medio (la *mediana*) en lugar del promedio, porque un solo acento fuerte no debe volver fuerte todo el compás.

Los umbrales se escriben como fracciones, para que toda la aritmética quede en números enteros:

```
const TUTTI: (u32, u32) = (9, 5);
// ...
let (times, by) = TUTTI;
let stands_out =
    strength(at) * by >= in_bar * times && strength(at) * by >= elsewhere
* times;
```

`strength(at) * 5 >= in_bar * 9` es `strength(at) >= 1,8 * in_bar` sin un número de coma flotante a la vista. `elsewhere` es el mismo lugar en los dos compases anteriores y en los dos siguientes, el más suave de los dos lados; el primer compás no tiene compases antes, así que compara solo con los de después:

```
let elsewhere = match (same_place(before.clone()),
same_place(after.clone())) {
    (Some(before), Some(after)) => before.min(after),
    (one, other) => one.or(other).unwrap_or(0),
};
```

Hacer `match` sobre un par de `Option` cubre los cuatro casos en dos brazos: el primero toma los dos valores, el segundo el que exista, o 0.

Para saber si un canal golpea «solo», `hear` guarda en orden los comienzos de cada canal y busca el lugar de la nota entre ellos con `partition_point`, una búsqueda binaria: el número de comienzos antes de `at`.

```
let place = steps.partition_point(|&step| step < at);
let before = place
    .checked_sub(1)
    .is_none_or(|before| steps[before] + ALONE < at);
let after = steps
    .get(place + 1)
    .is_none_or(|&after| after > at + ALONE.max(length));
```

`checked_sub(1)` es `None` para la primera nota en lugar de bajar de cero, y `Option::is_none_or` es `true` para `None`: no haber nota antes de la primera vale tanto como un silencio.

`Following::play_along(player, notes, ...)` toma luego lo que escribió un músico del estilo y lo cambia: se quitan las notas de un compás en silencio, se quitan las notas de los cornos y los timbales en un pasaje de acentos, el bombo conserva solo los primeros pulsos de los compases fuertes, y cada velocidad se escala según lo fuerte que suena el compás. Después añade los golpes: en cada acento, los cornos tocan las notas del acorde que suena ahí, los timbales su bajo y el bombo un golpe, tan fuerte como el acento de la partitura.

`write_band`, que escribe la parte de cada músico, lo llama solo para un estilo que sigue a la canción:

```
let mut part = self.band_notes(&playing, player, &score);
if let Some(following) = &following
    && playing.style.follows_the_song()
{
    part = self.played_along(following, &playing, player, part, &score);
}
```

`if let ... && ...` es una *let chain*: el `if` se ejecuta solo cuando el patrón coincide y la condición después de `&&` se cumple.

Muchos patrones cortos

La banda escribe un patrón para cada músico y cada tramo de compases en una misma métrica. Dos tramos pueden compartir un patrón cuando tocan las mismas notas, como la batería de dos tramos de 9/8, pero los acordes cambian de un tramo a otro. El Juego del rapto tiene 33 tramos, y su orquesta escribe 92 patrones además de los 34 de la partitura. Un proyecto admite hasta 999 patrones (`MAX_PATTERNS` en `project.rs`), y la ventana dodecafónica lee la misma constante cuando dice que una serie necesitaría más patrones de los que caben en un proyecto.

Cómo se prueba

Los tests de `band.rs`:

- `an_orchestra_plays_every_bar_of_the_rite_of_spring_in_its_own_meter` comprueba que el Juego del rapto conserva cada cambio de compás, sus

cambios de tempo y el silencio de cada canal, que los músicos son un corno francés, timbales y un conjunto de cuerdas, que el bombo toca solo con los acentos de la partitura o en el primer pulso de un compás, y no en todos los compases, y que los cornos y los timbales tocan en pulsos sentidos, cambios de acorde o acentos.

- `the_orchestra_strikes_with_the_accents_of_the_augurs_of_spring` copia solo las voces y comprueba que los canales y las notas de la canción quedan como estaban, que los cornos, los timbales y el bombo golpean en los pasos 18, 22, 34, 40, 48 y 58 de los primeros ocho compases y en ningún otro, y que en los compases 9 a 12 descansan y las cuerdas tocan por debajo de la velocidad 60.
- `a_tune_or_a_held_chord_is_not_an_accent` no oye ningún acento en el Villancico de las campanas, y comprueba que los cornos tocan el primer pulso de cada uno de sus primeros ocho compases.
- `with_no_score_to_follow_the_orchestra_plays_as_written` la aplica a una hoja de acordes sin notas, y comprueba que todos los músicos tocan.
- `a_band_brought_without_drums_leaves_them_to_the_song` copia las partes de acordes y bajo y las voces de un estilo que no sigue a la canción, Pop emotivo, sobre el tango El Choclo, sin su batería, y comprueba que su banda entra sin batería propia mientras la canción conserva sus sonidos, sus notas y su batería; tomar la batería después la trae en lugar de la de la canción.
- `copying_only_the_voices_brings_no_band` pasa solo las voces de Pop emotivo a `apply_band_parts` sobre El Choclo, y comprueba que deja la canción como estaba: las voces solas van a los canales propios de la canción, por el camino de la Biblioteca, no a través de una banda.
- `picking_the_orchestra_keeps_the_meters_of_the_song` hace de la Orquesta el estilo de la banda en los Augurios, con `set_band_style`, y después de **Ritmo pop**: la primera conserva el 2/4, el segundo pone la canción en 4/4.

En `rhythm.rs`, `a_rhythm_that_follows_the_meter_plays_each_bar_in_its_own` comprueba los golpes que escribe `notes_in` para 2/4, 3/4, 9/8, 5/8, 7/8 y 3+2+2/8, y que `for_time` nunca elige la Orquesta.

En la aplicación, `copying_only_the_voices_or_the_patterns_brings_the_band_and_leaves_the_song_alone` en `ux_tests.rs` recorre a clics el popup de la Biblioteca para los Augurios, el Juego del rapto, La habitación de Petrushka, el Villancico de las campanas y un tango, con una sola casilla marcada, y comprueba que la banda entró y que los canales y las notas de la canción no cambiaron.

Para oírlo sin escuchar, guarda los Augurios después de **Aplicar** y lee su traza de reproducción:

```
ferrous-lineup --trace augurs.lineup --out augurs.tsv
awk -F'\t' '$11 == "Kick" && $3 <= 16 { print $3, $4, $5, $16 }'
augurs.tsv
```

La columna 3 es el compás, la 4 el pulso, la 5 cuánto se ha avanzado dentro del pulso en pasos (una negra tiene cuatro, así que 2 es la corchea después del pulso) y la 16 la velocidad:

```
3 1 2.000 127
3 2 2.000 127
5 1 2.000 127
6 1 0.000 127
7 1 0.000 127
8 1 2.000 127
14 1 2.000 127
14 2 2.000 127
16 1 2.000 127
```

Fuera del pulso dos veces en el compás 3, fuera de él en el compás 5, en él en los compases 6 y 7, fuera en el compás 8: los acentos de los cornos tal como los escribió Stravinsky, sin nada en los compases tranquilos entre el 8 y el 14.

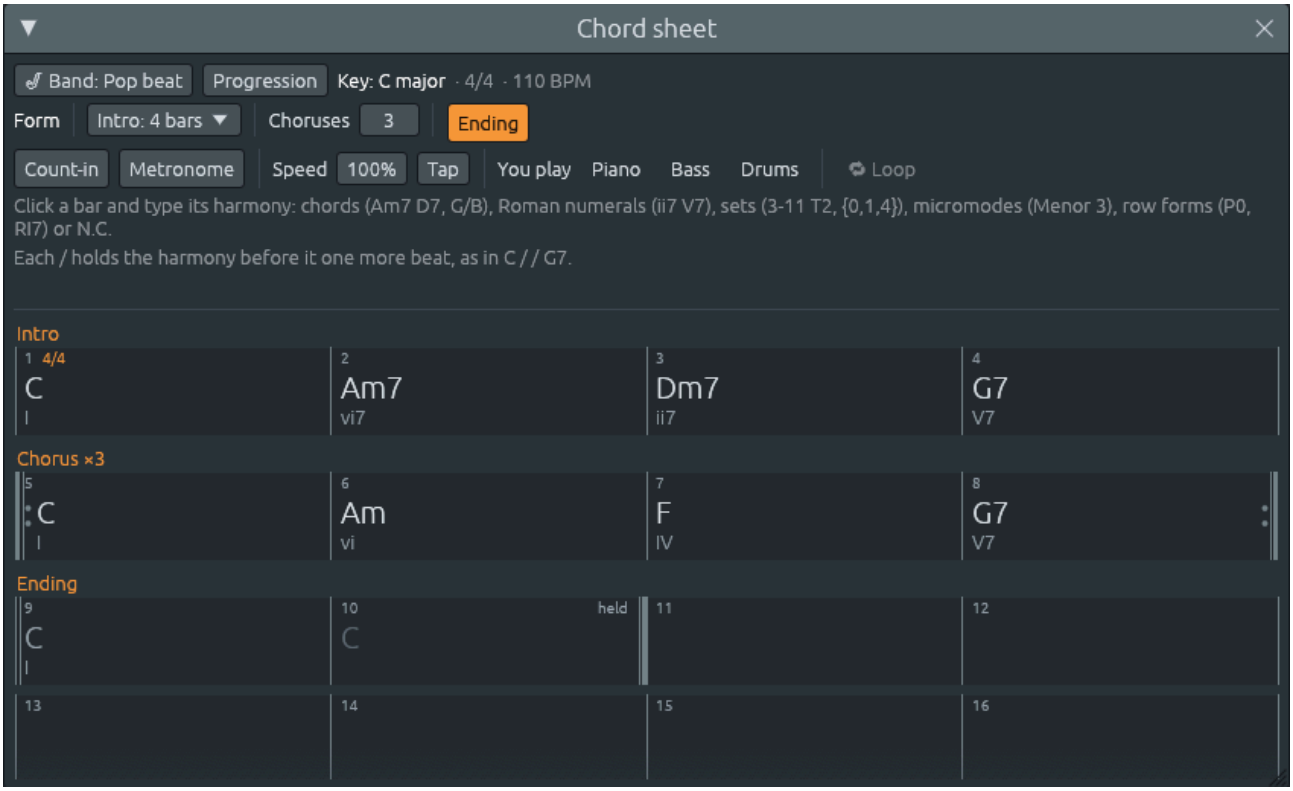
Forma, secciones, variaciones y bloqueos

Archivos: `crates/lineup-core/src/band/form.rs`, `crates/lineup-core/src/band/variation.rs`, `crates/lineup-core/src/band/sections.rs`, `crates/lineup-core/src/band/locks.rs`, `crates/lineup-core/src/lock.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/arrangement.rs`, `crates/lineup-core/src/swing.rs`, `crates/lineup-app/src/chord_sheet.rs`, `crates/lineup-app/src/chord_sheet/form.rs`, `crates/lineup-app/src/playlist/sections.rs` y `crates/lineup-app/src/channel_rack.rs`

Una banda no toca una melodía una vez, igual del primer compás al último, y se detiene en seco. Toca unos compases para asentar la tonalidad y el groove, luego la melodía, quizá varias veces, y termina en el acorde de tónica. Toca la estrofa con suavidad y aprieta en el estribillo. Un tema de jazz latino puede tocar su estrofa como bossa nova y su puente en swing. Y si tú tocas la parte de piano, la banda toca a tu alrededor.

Este capítulo trata de esas cuatro herramientas: la **forma** (una introducción, vueltas y un final), las dos **variaciones** de todo estilo, un **estilo para cada sección** y los **bloqueos**, que apartan una pista del camino de los estilos. Las tres primeras viven en la hoja de acordes; los bloqueos, en el rack de canales.

La forma: una introducción, vueltas y un final

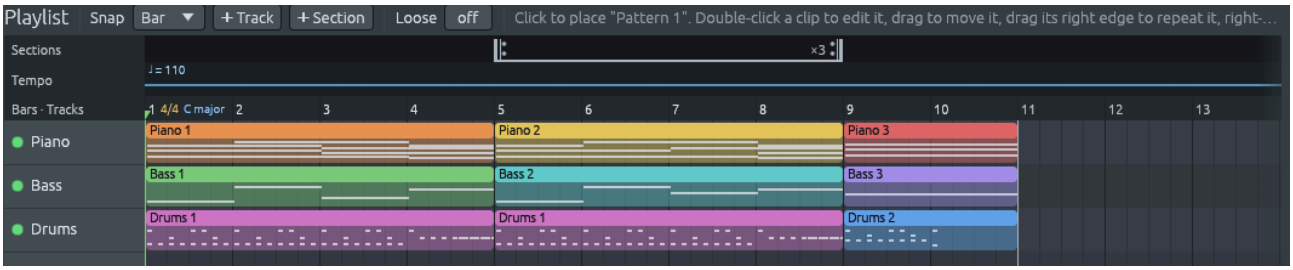


Probarla

Escribe unos compases de acordes en la hoja (**F5**), o elige una progresión. Bajo los botones de la hoja hay una fila llamada **Forma**:

- **Introducción**: Sin introducción, 2, 4 u 8 compases. La banda pone esa cantidad de compases delante de la canción, con acordes que llevan a su primer acorde. Se escriben en la hoja bajo el encabezado **Introducción**, así que puedes leerlos, oírlos (mantén uno pulsado con el ratón) y cambiarlos. Elegir una longitud otra vez los escribe de nuevo; **Sin introducción** los quita. El resto de la canción, melodía incluida, se corre para hacer sitio.
- **Vueltas**: cuántas veces toca la banda la melodía. La hoja la muestra una vez, bajo **Vuelta ×3**, entre los signos de repetición de la música impresa: 𝄆: al principio, :𝄇 al final. Mientras suena, la fila Forma muestra por qué vuelta va, «▶ 2 de 3».
- **Final**: dos compases después de la última vuelta. La banda va al acorde de tónica de la tonalidad, toca un compás de él en el estilo, y luego lo golpea una vez y lo sostiene todo el último compás, con un golpe de platillo crash. Los dos compases se muestran en gris bajo **Final**, el último marcado «sostenido», con la doble barra final después.

La lista de reproducción también muestra la repetición: a lo largo del borde inferior de la tira de secciones, los mismos signos de repetición en los dos extremos de la vuelta, y «×3».



Una vuelta es una pasada completa

En una canción pop el estribillo es la parte que vuelve después de cada estrofa (en «Let It Be», «When I find myself in times of trouble» empieza una estrofa y «Let it be, let it be» es el estribillo). Los músicos de jazz hablan de otra cosa: un **chorus**, o una **vuelta**, es una pasada por toda la melodía, desde el principio de la partitura hasta su último compás. Un solista «toma dos vueltas»; la banda toca «el tema» (la melodía) durante una vuelta, luego solos durante varias, y luego el tema otra vez. La fila Forma usa la palabra en ese sentido: una vuelta es cada compás de la hoja después de la introducción. Las estrofas y los estribillos de una canción pop están todos dentro de una vuelta. Si tu canción tiene secciones propias (Estrofa ×2, Estribillo) en la lista de reproducción, se repiten como siempre, dentro de cada vuelta.

La introducción: un turnaround

Un **turnaround** es el puñado de acordes al final de una melodía que le «dan la vuelta» y la llevan de nuevo al principio. Es la introducción más natural que existe: lleva al primer acorde, así que cuando la melodía empieza suena como una llegada, no como un comienzo salido de la nada.

El motor de un turnaround es la **séptima de dominante**. En Do mayor, G7 (Sol Si Re Fa) guarda dos notas que quieren moverse: Si, un semitono por debajo de Do, tira hacia arriba a Do, y Fa, un semitono por encima de Mi, tira hacia abajo a Mi. Tocado antes de C, G7 hace que Do suene a casa. Y el acorde una quinta por encima de Sol, Re menor, lleva a Sol de la misma manera que Sol lleva a Do: las fundamentales que caen por quintas (La → Re → Sol → Do) son la manera más fuerte en que pueden moverse los acordes.

Introducción	Hacia Do mayor	Hacia La menor
2 compases	Dm7, G7 (ii7 V7)	Bm7b5, E7 (iiø7 V7)

Introducción	Hacia Do mayor	Hacia La menor
4 compases	C, Am7, Dm7, G7 (I vi7 ii7 V7)	Am, Fmaj7, Bm7b5, E7 (i bVImaj7 iiø7 V7)
8 compases	los cuatro acordes, dos compases cada uno	los mismos, dos compases cada uno

Hacia un acorde menor el ii es **semidisminuido** (Si Re Fa La sobre Am): ese es el acorde que está de verdad sobre la segunda nota de La menor. El V sigue siendo una séptima de dominante, E7, con Sol sostenido: la **sensible**, un semitono por debajo de La, tomada de la escala menor armónica, porque un Mi menor simple no tiraría hacia La ni de lejos con tanta fuerza.

Los números se cuentan desde **el acorde al que lleva la introducción**, como si fuera la tónica, no desde la tonalidad de la canción. La mayoría de las canciones empiezan en su tónica, y entonces las dos cosas coinciden. Una canción en Do que empieza en F (IV) recibe Gm7, C7 como introducción de dos compases: el ii y el V *de Fa*, así que la introducción sigue llevando directamente al primer acorde. Un blues que empieza en C7 recibe C7, Am7, Dm7, G7. Los acordes se escriben desde la letra con que está escrito el primer acorde: hacia Ebm la introducción de cuatro compases es Ebm, Cbmaj7, Fm7b5, Bb7, no los mismos sonidos escritos desde Re sostenido.

Un conjunto de clases de alturas, un micromodo o una forma de serie no tienen dominante que lleve a ellos, así que la introducción los toca desde el primer compás: un **vamp**, la banda asentando el sonido antes de que entre la melodía. Lo mismo un acorde disminuido, al que ningún turnaround resuelve.

El final: a casa, y sostenido

El final va a la **tónica**, el acorde hacia el que se inclina toda la tonalidad. ¿Qué acorde de tónica? Si la melodía empieza en la tónica, el final toca ese mismo acorde: un blues en Do que empieza en C7 termina en C7, una balada que empieza en Cmaj7 termina en Cmaj7. Si no, toca la tríada propia de la tonalidad, C en Do mayor y Am en La menor. La tonalidad es aquella en la que termina la última vuelta, así que una canción que modula termina en su tonalidad nueva. Una canción en un conjunto o una serie no tiene tónica; termina donde empezó, en su primera armonía.

El último compás se toca distinto de todos los demás: el piano golpea el acorde una vez y lo sostiene, el bajo sostiene la fundamental, y la batería toca un golpe de bombo con el platillo crash, el «Final con crash» de la biblioteca. El último compás de la vuelta ya lleva un fill de batería (lleva a la vuelta siguiente o al final), así que el primer compás del final es groove simple, y la canción no termina con dos fills seguidos.

El final no está escrito en la hoja: sus acordes son de la banda, calculados cada vez desde la tonalidad. Escribe un acorde en un compás del final y se convierte en un compás de la melodía: la vuelta crece para incluirlo, y el final se corre detrás de él.

`Form`: tres números que viajan con la banda

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
#[serde(default)]
pub struct Form {
    /// How many steps of intro come before the chorus: 0 for none.
    pub intro: usize,
    /// How many times the chorus plays: 1 to [`MAX_REPEATS`].
    pub choruses: usize,
    /// Whether the band plays an ending after the last chorus.
    pub ending: bool,
}
```

`Form` vive dentro de la `Band`, porque solo una banda toca una forma. Es `Copy`: tres números simples no cuestan nada de copiar, así que las funciones toman un `Form`, no un `&Form`. `#[serde(default)]` en el struct significa que cualquier campo que falte en un archivo toma su valor de `Form::default()`.

¿Dónde están la introducción y la vuelta? Solo se guarda la longitud de la introducción, en pasos. La vuelta se calcula cuando hace falta, desde el final de la introducción hasta el final del último compás escrito:

```
pub fn chorus(&self) -> Option<(usize, usize)> {
    let band = self.band.as_ref()?;
    let (start, end) = (band.form.intro, band.end);
    (start < end).then_some((start, end))
}
```

así que escribir un acorde después del último compás alarga la vuelta sin nada que mantener al día. El final se calcula de la misma manera (`Project::ending`): los dos compases después de la vuelta, y la armonía que elige `ending_harmony`.

La introducción mueve la canción

Poner compases delante de una canción significa correr todo lo demás más tarde: clips, secciones, marcadores de compás, de tempo, de tonalidad y de acordes, la región de bucle. Eso lo hace `Project::make_room`, la misma función que hace sitio al copiar una sección. Quitar la introducción necesita lo contrario, `Project::cut_start`, que además tiene que conservar cómo suena la canción donde termina el corte: la

canción *empieza* entonces en el compás, el tempo, la tonalidad y la armonía que sonaban allí.

```
let made = self.moving_the_band(|project| {
    project.cut_start(old);
    steps == 0 || project.make_room(0, steps)
});
```

`moving_the_band` ejecuta el cambio y luego le dice a la banda que sus partes se movieron con todo lo demás (ver Mover la canción entera no es editar). Sin él, la banda compararía sus clips con las huellas que guardó, los encontraría en lugares nuevos y pensaría que los editaste: cada parte quedaría congelada.

Tocar la vuelta otra vez: el arreglo

La lista de reproducción es la canción *como se dibuja*; el arreglo (ver Secciones, repeticiones y bucles) es la canción *como se toca*, con cada repetición desplegada. Toca dos veces una sección marcada ×2. La vuelta es una capa más alrededor de eso:

```
let mut played: Vec<_> = once(from, start).collect();
for chorus in 1..=choruses {
    played.extend(
        pieces(project, start, end)
            .into_iter()
            .map(|piece| (piece, Some((chorus, choruses)))),
    );
}
played.extend(once(end, to));
```

Las piezas de la introducción una vez, las de la vuelta `choruses` veces, y luego el final una vez. Cada pasada recuerda por qué vuelta va en `Pass::chorus`, que la fila Forma y la lista de reproducción leen para mostrar «▶ 2 de 3».

Como todo lo que suena lee el arreglo (el hilo de audio, las exportaciones WAV y MIDI, los clics del metrónomo, el tempo de cada paso), ninguno necesita saber nada de vueltas. Una región de bucle dentro de la vuelta no la abarca entera, así que ahí la vuelta no se repite: un bucle de los compases 3 a 4 toca los compases 3 a 4.

El último compás: un closure, dos maneras de tocar

El piano toca cada compás con el acompañamiento del estilo, y el último compás de un final con `Comping::Held`; el bajo, igual con `Bass::Held`. Así que la banda toca sus compases en **tramos** que dejan fuera el compás sostenido (más sobre los tramos en

Tocar por tramos), y luego toca el compás sostenido aparte. La manera de tocar se le pasa a la función como un **closure**:

```
fn held_bar(
    held: Option<Bar>,
    harmonies: &[Played],
    play: impl Fn(&[Bar], &[Played]) -> Vec<NoteEvent>,
) -> Vec<NoteEvent> {
    held.map_or_else(Vec::new, |bar| {
        play(&[bar], &cut(harmonies, bar.start, bar.end()))
    })
}
```

`impl Fn(&[Bar], &[Played]) -> Vec<NoteEvent>` quiere decir «cualquier cosa que se pueda llamar con compases y armonías y devuelva notas». El piano pasa `|bars, played| Comping::Held.play_in_bars(bars, played)`, el bajo el de `Bass::Held`. `cut` recorta las armonías al compás, y además impide que una armonía cortada antes de tiempo «lleve» a alguna parte: un bajo caminante caminaría si no hacia el acorde siguiente antes de que llegue.

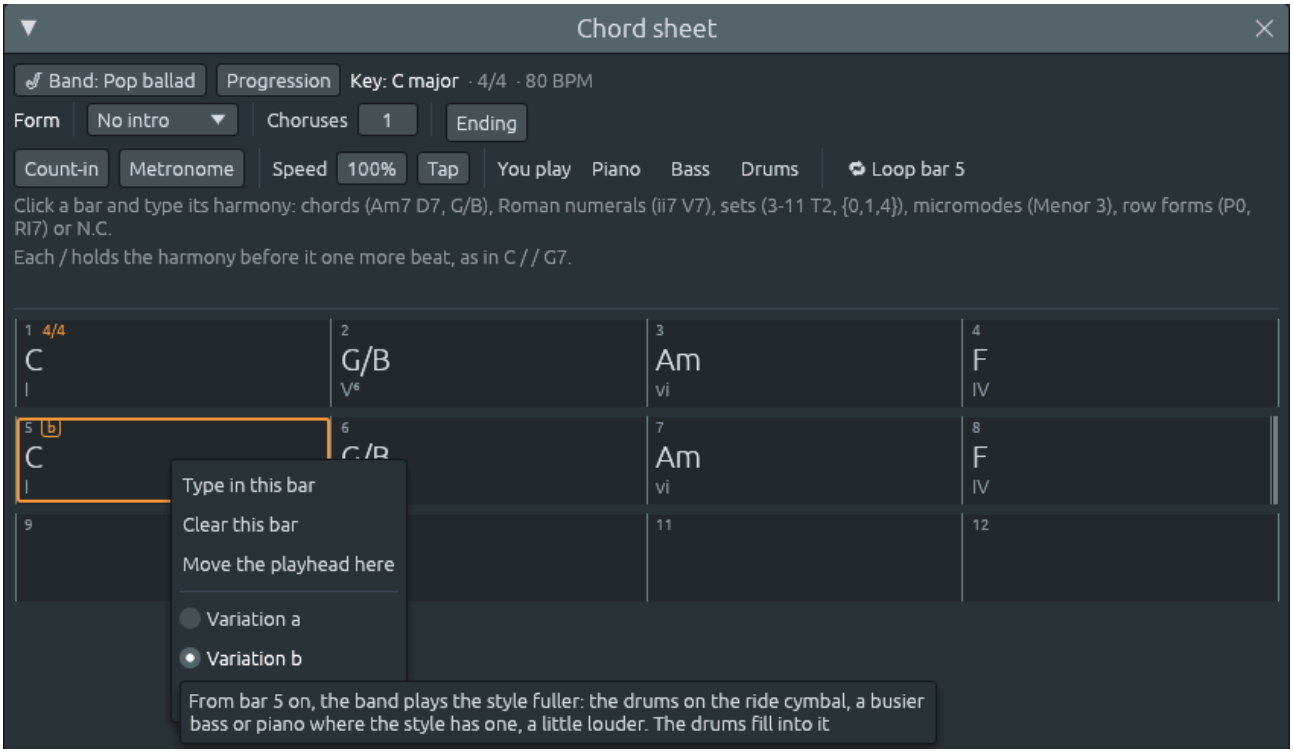
La armonía del final no está en la hoja, así que la banda la añade a la lista de marcadores desde la que toca, como si estuviera escrita allí:

```
if let Some(ending) = ending {
    markers.push(ChordMarker::new(ending.start, ending.harmony));
}
```

Luego se dispone a partir del acorde anterior, como cualquier otro, así que el piano llega al último acorde con suavidad.

Variaciones a y b: la banda aprieta

Una banda no toca todas las partes de una canción igual. La estrofa es más ligera, para que se entienda la letra; el estribillo es más lleno, y el baterista, el bajista y el pianista levantan a la vez. Un grupo de jazz toca la primera vuelta de una balada con suavidad y luego empieza a swinguear más fuerte. Por eso todo estilo tiene dos variaciones: **a** es el estilo tal como es, y **b** es el mismo estilo tocado más lleno.

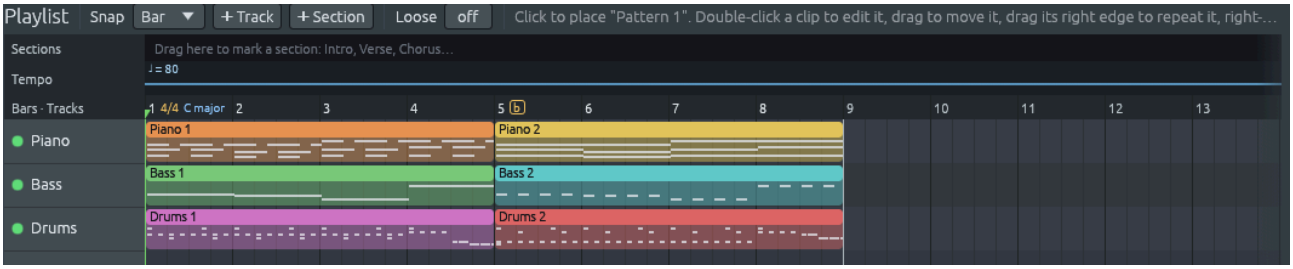


Probarlas

Haz clic derecho en un compás de la hoja de acordes (**F5**) y elige **Variación a** o **Variación b**. La banda toca esa variación desde ese compás hasta el cambio siguiente, o hasta el final de la canción. Para cambiar solo algunos compases, selecciónalos primero: clic en el primero, Mayús+clic en el último, y luego clic derecho en uno de ellos. Los compases después de la selección conservan lo que tocaban. Así se toca más lleno solo el puente de una canción, o solo la última vuelta. Un clic derecho en el nombre de una sección de la hoja selecciona sus compases y abre el mismo menú.

Una letra enmarcada marca cada cambio, en la hoja después del número del compás y en la regla de compases de la lista de reproducción, como las letras de ensayo impresas sobre el pentagrama en las partes de una banda. Elegir la variación que ya suena no pone ninguna marca; volver un compás atrás quita la marca.

La batería anuncia cada cambio: el compás anterior es un **fill**, y el primer pulso de la nueva variación lleva un platillo **crash**. Es la misma señal que dan los bateristas de verdad, y la razón por la que un oyente sabe que «aquí empieza algo nuevo» aun sin oír la letra.



Qué quiere decir «más lleno»

La variación b no inventa un estilo nuevo. Abre el mismo, como lo hacen los músicos:

Quién	Variación a	Variación b
Batería	el tiempo en el charles cerrado	el tiempo en el platillo **ride**, salvo que el estilo ya lleve el tiempo en el ride
Batería, en una balada pop o rock	**cross-stick** en el aro de la caja	la caja entera
Bajo, un estilo de jazz tocado a dos, en 4/4	**a dos**: fundamental y quinta, una cada dos pulsos	**caminando**: una nota en cada pulso
Bajo, blues shuffle	negras	la línea **boogie**, arriba y abajo por el acorde
Bajo, pop, rock, soul o folk en 4/4	la fundamental sostenida	la fundamental en cada pulso
Piano, balada pop o rock	el acorde, y luego sus notas una a una	corcheas parejas
Guitarra	acordes que resuenan, o punteados; power chords sostenidos	rasgueados; power chords machacados en corcheas
Cuerdas u órgano, en algunos estilos	callados	entran
Todos		un poco más fuerte: la velocidad de cada nota, una décima más alta

Algunas palabras de esa tabla:

- El **charles** (hi-hat) son dos platillos que el baterista cierra con un pedal. Cerrado da un «tic» corto y seco. El **ride** es un solo platillo grande que resuena; las corcheas sobre él hacen un brillo continuo que llena más el sonido. Pasar de uno al otro es la manera más común en que un baterista levanta un estribillo.
- Un **cross-stick** es la baqueta apoyada sobre la caja, con su punta golpeando el aro: un sonido suave, de madera. Golpear el parche de la caja suena mucho más fuerte. Las baladas están llenas de estrofas con cross-stick y estribillos con la caja entera.
- Un bajista de jazz **a dos** toca en los pulsos 1 y 3, normalmente la fundamental y la quinta, lo que deja aire alrededor de una melodía tranquila. **Caminando** llena cada pulso con una línea que lleva al acorde siguiente: sobre Dm7 camino de G7, el bajo de la Balada de jazz toca Re, Fa, La y Fa♯, el Fa♯ un semitono por debajo del Sol al que se dirige. Pasar de «a dos» a «a cuatro» es como una sección rítmica de jazz sube la temperatura sin tocar más fuerte.

Donde un estilo no tiene una manera más ocupada bien conocida de tocar (un tango, una bossa nova, un coral), la variación b conserva sus patrones y solo cambia la batería y el volumen. Las reglas son pocas a propósito, para que b siempre suene a la misma banda, no a otra canción. Las partes de la guitarra y de los colchones en b se describen con la banda más grande (ver Enums que llevan datos, y una función como parámetro).

Marcadores que solo marcan cambios

```
pub struct VariationMarker {
    pub start: usize,
    pub variation: Variation,
}
```

Los marcadores viven en la `Band`, junto a su `Form`, porque solo una banda toca variaciones. Un marcador que no cambia nada (la variación a al principio de la canción, o una b justo después de otra b) solo ensuciaría la hoja, así que después de cada cambio `tidy` se queda solo con los de verdad:

```
pub(crate) fn tidy(markers: &mut Vec<VariationMarker>) {
    markers.sort_by_key(|marker| marker.start);
    markers.dedup_by(|later, earlier| {
        later.start == earlier.start && {
            *earlier = *later;
            true
        }
    });
    let mut playing = Variation::A;
    markers.retain(|marker| {
        let changes = marker.variation != playing;
        playing = marker.variation;
        changes
    });
}
```

Primero en orden de comienzo; luego, de dos marcadores en el mismo paso, gana el último (`dedup_by` le pasa al closure el elemento posterior y el que se quedó antes que él, y el closure copia el posterior encima); luego solo quedan los cambios. El closure que se le pasa a `retain` cambia `playing`, una variable de fuera. Un closure que cambia lo que captura es un `FnMut`: se puede llamar muchas veces, pero no desde dos lugares a la vez, porque cada llamada necesita la variable para ella sola. `retain` pide exactamente eso.

Qué variación suena en un paso es el último marcador en ese paso o antes:

```
markers
    .iter()
    .rev()
    .find(|marker| marker.start <= step)
    .map_or(Variation::A, |marker| marker.variation)
```

`rev()` recorre la lista hacia atrás, así que `find` se detiene en el primer marcador que no está después del paso.

Tocar por tramos

Los compases se agrupan en **tramos**, trechos en una misma variación, y cada tramo se toca con la manera de tocar de esa variación:

```
fn played_in_runs(
    runs: &[Run],
    harmonies: &[Played],
    play: impl Fn(Variation, &[Bar], &[Played]) -> Vec<NoteEvent>,
) -> Vec<NoteEvent> {
```

La parte del piano son las dos llamadas juntas, los tramos y luego el compás sostenido de un final:

```
Player::Piano => {
    let mut notes =
        played_in_runs(&score.runs, &score.harmonies, |variation, bars,
played| {
            band.comping_in(variation).play_in_bars(bars, played)
        });
    notes.extend(held_bar(score.held, &score.harmonies, |bars, played| {
        Comping::Held.play_in_bars(bars, played)
    }));
```

Cada tramo recibe solo sus propias armonías, recortadas en sus bordes con `cut`. Una canción sin marcadores es un solo tramo.

`comping_in` y `bass_in` son pequeños `match` sobre la variación, el género del estilo y su manera de tocar:

```
match (variation, self.genre, self.bass) {
    (Variation::B, Genre::Jazz, Bass::RootFifth) if common =>
Bass::Walking,
    (Variation::B, Genre::Blues, Bass::Quarters) if common =>
Bass::Boogie,
    // ...
    (_, _, bass) => bass,
}
```

Hacer `match` sobre una tupla deja que una sola línea nombre toda una situación, y el último brazo deja todo lo demás como está.

La batería en otros tambores

La batería conserva los patrones del ritmo y cambia qué tambor los toca: `drum_in(Drum::HiHat, Variation::B)` es `Drum::Ride` cuando el ritmo no tiene ride propio. La banda escribe cada patrón en el canal del tambor que debe sonar, así que las corcheas del charles caen en el canal del ride. Un canal que la canción todavía no tiene (un ride, un crash) se añade la primera vez que una variación lo necesita, y nunca si no: una canción sin marcadores conserva exactamente los canales que tiene.

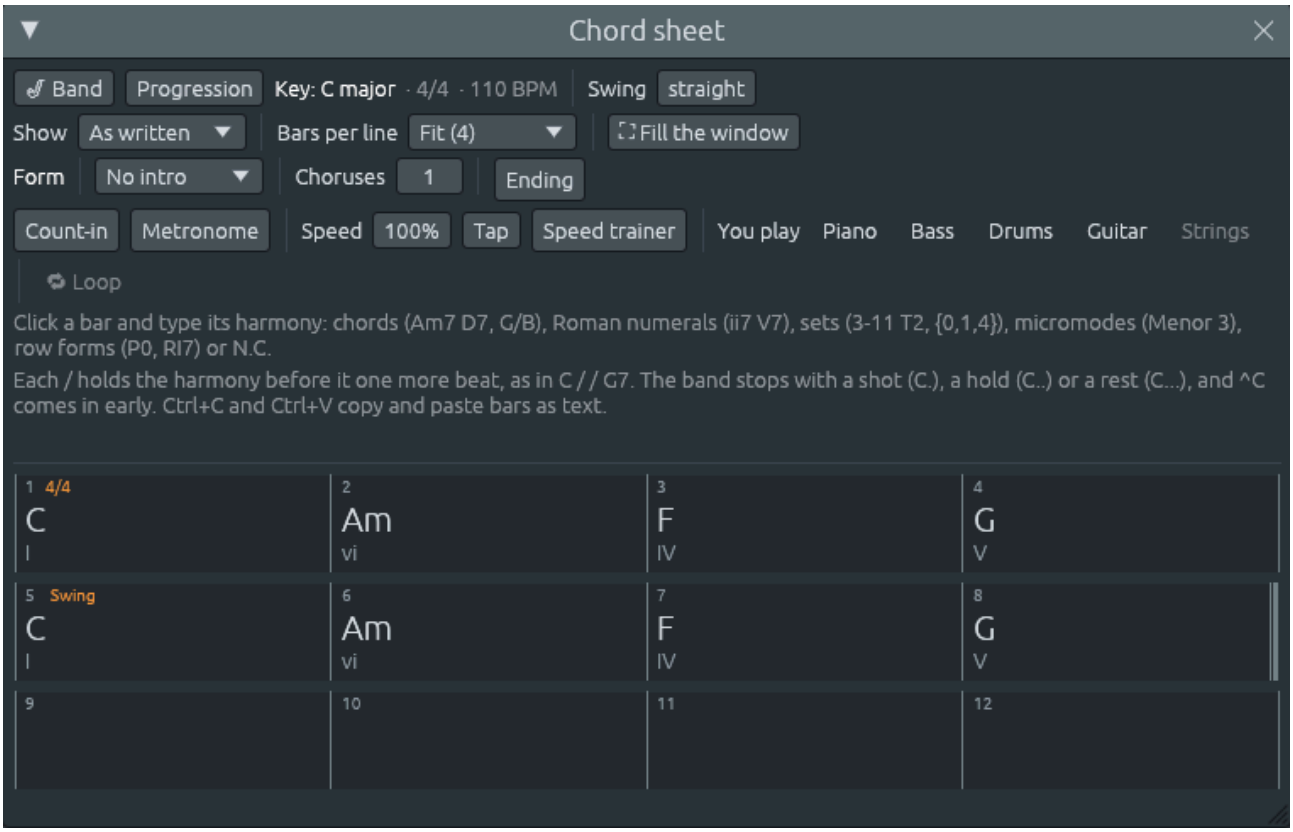
Cada cambio de variación también empieza un **trozo** nuevo, el trecho que cubre un patrón, así que la lista de reproducción muestra las partes a y b como clips separados que puedes ver, silenciar o editar.

Los marcadores se mueven con la canción

Todo lo que mueve la canción mueve también los marcadores: una introducción puesta delante (`make_room`), una introducción quitada (`cut_start`), una sección copiada (`copy_section` , que copia las variaciones que tiene dentro), un compás nuevo (`remeter`) y **Terminar la canción aquí**, que quita los marcadores después del final.

Un estilo para cada sección

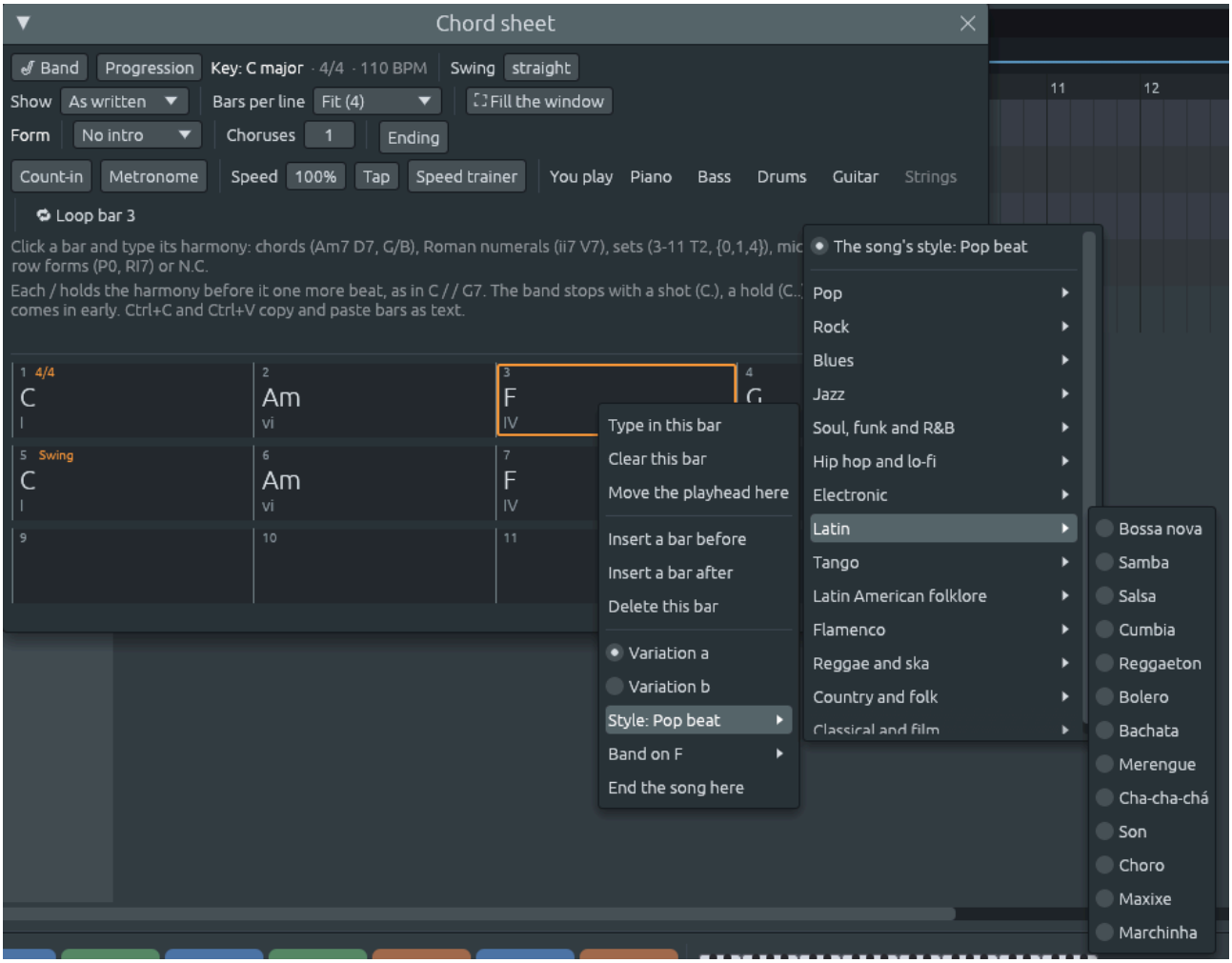
Las variaciones hacen más llena una parte de una canción, pero en el mismo estilo. Muchas canciones cambian de carácter a mitad de camino: un tema de jazz latino toca su estrofa como bossa nova y su puente en swing; una canción pop pasa a medio tiempo en el puente; un arreglo empieza como balada y termina como canción de rock. Por eso cada parte de una canción puede tener un estilo propio.



Probarlo

1. Abre la hoja de acordes con **F5**, con una banda tocando (cualquier demostración, o un estilo de la Biblioteca).

2. Haz clic derecho en el nombre de una sección de la hoja. Se seleccionan los compases de la sección, y el menú tiene **Variación a**, **Variación b** y **Estilo: …**, que nombra el estilo que suena ahí ahora.
3. Abre **Estilo**, luego un género, y elige un estilo, por ejemplo **Latina** › **Bossa nova**. Los compases de esa sección se tocan ahora como bossa nova. La hoja escribe «Bossa nova» después del número del primer compás, y el aviso dice qué compases cambiaron.
4. El mismo menú **Estilo** está en el menú de clic derecho de cada compás. Sin selección, cambia el estilo desde ese compás en adelante; con compases seleccionados (clic en el primero, Mayús+clic en el último), cambia solo esos, y los compases de después conservan lo que tocaban.
5. **El estilo de la canción**, arriba del todo en el menú, vuelve al estilo propio de la banda.



El menú solo lista estilos en el compás de esos compases, y la Orquesta, que toca en cualquiera. Un vals en medio de una canción en 4/4 necesitaría compases de tres pulsos, lo que es cambiar los compases de la canción, no solo cómo los toca la banda. Cambiar el compás es un paso aparte, así que aquí sigue aparte.

Qué cambia y qué se queda

El estilo de una sección cambia **cómo** toca la banda:

- el **acompañamiento** del piano, el ritmo con que golpea los acordes,
- la **línea de bajo**,
- la **batería**,
- los patrones de la guitarra, las cuerdas y el órgano, donde la banda los tiene,
- el **swing**: una sección en swing dentro de una canción pop recta hace swing con sus corcheas, y una sección de rock dentro de una canción de jazz las toca parejas.

No cambia **quién** toca ni los sonidos con que toca. La banda conserva sus instrumentos, así que el mismo piano y el mismo bajo pasan de la estrofa en bossa nova al puente en swing, como lo haría una banda de verdad. Para cambiar los instrumentos, usa la **Formación de instrumentos** de la banda.

La batería avisa al oyente de que viene un cambio, como hace con una variación: el compás antes del cambio toca el fill del estilo que termina, y el primer pulso del estilo nuevo lleva un platillo crash. El patrón de acompañamiento de cada músico vuelve a empezar desde el principio en el primer compás del estilo nuevo, para que un patrón de bossa nova de dos compases no empiece por la mitad.

Marcadores, igual que las variaciones

```
pub struct StyleMarker {
    pub start: usize,
    pub style: Option<&'static BandStyle>,
}
```

Un marcador con `style: None` vuelve al estilo propio de la banda. Los marcadores viven en la `Band`, junto a los de variación, y siguen las mismas reglas: un `tidy` propio se queda solo con los marcadores que cambian algo, y todo lo que mueve la canción los mueve también.

`&'static BandStyle` es una referencia que vive todo el programa: los estilos de banda se construyen una vez (`band_styles()` los guarda en un `OnceLock`) y nunca se liberan. Por eso un marcador puede guardar una referencia en lugar de una copia o un nombre. Un archivo no puede guardar una referencia, así que `StyleMarkerFile` guarda el nombre del estilo y lo vuelve a buscar al leer la canción; un nombre que esta versión no conoce es un error que lo dice.

Cada estilo toca la canción entera, y cada uno se queda con sus compases

La manera más sencilla de escribir una parte con varios estilos no es enseñarle a cada pieza de la banda qué son las secciones. En su lugar, `write_band` le pide a cada estilo que suena en algún sitio la canción entera, y conserva cada nota solo donde suena ese estilo:

```
for style in sections::styles_played(&band.styles) {
    let playing = band.playing(style);
    let mut part = self.band_notes(&playing, player, &score);
    if let Some(following) = &following
        && playing.style.follows_the_song()
    {
        part = self.played_along(following, &playing, player, part,
&score);
    }
    notes.extend(sections::where_it_plays(part, &band.styles, style));
}
```

`band.playing(style)` es una copia de la banda con los patrones del otro estilo. `band_notes` escribe la parte de un músico para un estilo (el `if let` del medio es la orquesta tocando junto a una partitura). `where_it_plays` conserva las notas que empiezan donde suena `style`, y corta una nota que sonaría dentro del estilo siguiente:

```
notes
    .into_iter()
    .filter(move |(_, note)| style_in(markers, note.start) == style)
    .map(move |(channel, mut note)| {
        if let Some(change) = next_change(markers, note.start)
            && note.start + note.length > change
        {
            note.length = change - note.start;
            note.trim = 0;
        }
        (channel, note)
    })
```

`move` hace que cada closure sea dueño de lo que usa (`markers` y `style`, los dos baratos de copiar), para que el iterador se pueda devolver desde la función y usar después de que termine. Una canción sin estilos de sección tiene un solo estilo, `None`, y conserva todas las notas.

Los fills salen de la misma idea. La batería toca un fill en el compás antes de un cambio de variación, y un cambio de estilo también es un cambio. Cuando la banda toca el estilo que se va durante toda la canción, ese estilo tiene un fill antes del

cambio, y el compás antes del cambio le pertenece, así que su fill se conserva. El estilo que llega tiene un crash en el cambio, y el cambio es su primer compás, así que su crash se conserva.

Swing por sección

El swing se toca, no se escribe: las notas se quedan en la cuadrícula, y la reproducción retrasa la segunda corchea de cada pulso (ver Swing). Un patrón de la banda en una sección con estilo propio hace swing como ese estilo:

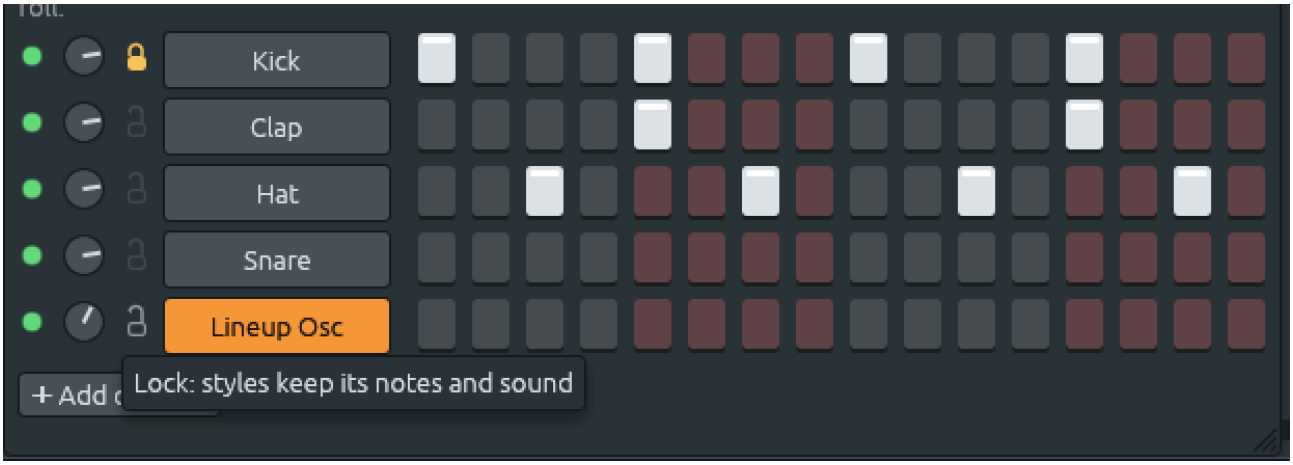
```
pub fn pattern_swing(&self, pattern: usize) -> u8 {
    let swing = self.section_swing(pattern).unwrap_or(self.swing);
    if swing > 0 && swings_in(self.pattern_time(pattern)) {
        swing
    } else {
        0
    }
}
```

`section_swing` mira dónde está el primer clip del patrón y devuelve el swing de esa sección, o `None` (el swing de la canción) para tus propios patrones y para las secciones en el estilo propio de la banda. La banda empieza un patrón nuevo donde empieza el estilo de una sección, y dos trechos con las mismas notas pero distinto swing nunca comparten patrón, así que cada patrón hace swing de una sola manera.

La traza de reproducción lo muestra: en una canción recta en Ritmo pop cuyos compases 5 a 8 están en Swing, los contratiempos del ride y del piano en el compás 5 empiezan 2,667 semicorcheas dentro del pulso, el último tercio de un tresillo, mientras que en los compases 1 a 4 caen en la cuadrícula.

Bloquear una pista: los estilos tocan a su alrededor

Supón que tocaste tú una parte de piano, o elegiste una guitarra de nailon para los acordes, y ahora quieres oír la canción como bossa nova. Elegir el estilo reescribiría cada pista que toca: tu parte de piano se cambiaría por el acompañamiento del estilo, y tu guitarra volvería al instrumento del estilo. Por eso puedes **bloquear** una pista antes. Una pista bloqueada conserva sus notas y su instrumento, y el estilo toca el resto de la canción a su alrededor.



Probarlo

1. Abre una demostración, por ejemplo **Oda a la alegría**, y muestra el rack con **F6**.
2. Haz clic en el candado pequeño entre la perilla de **Teclados** y su nombre. Se cierra y se vuelve naranja. El clic derecho en el nombre también tiene **Bloquear** y **Desbloquear**, y **Ctrl+Z** lo deshace.
3. En la **Biblioteca**, aplica otro estilo, por ejemplo **Bossa nova**. La banda entra en sus propias pistas, pero le deja el piano a tus Teclados: el aviso termina con 🔒 **Se conservan: Teclados.**, la lista de la Biblioteca de dónde viene cada parte tiene una fila **Bloqueado**, y la banda de la hoja de acordes dice «🔒 El canal Teclados está bloqueado: la banda le deja el piano.»
4. Prueba estilos uno tras otro en modo patrón. Cada prueba copia las filas bloqueadas en el patrón de prueba, así que el bajo nuevo sigue a los acordes bloqueados.
5. Desbloquea Teclados otra vez. La banda vuelve a tocar la parte del piano, y el aviso dice que tus Teclados quedan en silencio para que los dos no toquen uno encima del otro.

Lo que mueve la canción *entera* sigue moviendo una pista bloqueada con ella: un compás nuevo pone sus notas en los compases nuevos, y un cambio de tonalidad las lleva a la tonalidad nueva. Un bloqueo protege una parte de los estilos, no de que la canción cambie de forma. Una pista bloqueada siempre se puede editar a mano.

Un bit por canal

El rack tiene como mucho 32 canales, así que el conjunto de los bloqueados cabe en un `u32`, un bit por canal:

```rust
pub struct Locked(u32);

// One bit for each channel there can be.
const _: () = assert!(MAX_CHANNELS <= u32::BITS as usize);

impl Locked {
    pub fn contains(self, channel: usize) -> bool {
        channel < MAX_CHANNELS && self.0 & (1 << channel) != 0
    }
}
```

`Locked(u32)` es un **struct tupla**: un tipo nuevo envuelto alrededor de un `u32`, al que se llega con `.0`. Envolverlo quiere decir que nadie puede sumar dos conjuntos por error ni pasar un número de canales donde se espera un conjunto. La línea `const _` es una comprobación que hace el compilador: si alguien subiera `MAX_CHANNELS` por encima de 32, la compilación fallaría justo ahí en lugar de que un bloqueo cayera en silencio en el canal equivocado.

Quitar un canal renumera los que vienen después, así que el conjunto también tiene que correrse. `without` conserva los bits por debajo del canal quitado y baja una posición los de encima:

```rust
let below = self.0 & ((1 << index) - 1);
let above = self.0.checked_shr(index as u32 + 1).unwrap_or(0) << index;
Locked(below | above)
```

`checked_shr` devuelve `None` en lugar de desbordarse cuando el desplazamiento es de 32 o más, lo que pasa cuando se quita el último canal.

Una sola puerta para todos los estilos

Los estilos escriben en los patrones desde muchos sitios: un estilo entero, un ritmo, un estilo guardado, un estilo de serie o de micromodo, un fill, el acorde final. Si cada uno tuviera que acordarse de saltar los canales bloqueados, alguno se olvidaría. Así que la biblioteca nunca recibe un `&mut Pattern`. Recibe un `StylePattern`, que solo escribe las filas que no están bloqueadas:

```
pub(crate) struct StylePattern<'a> {
    pattern: &'a mut Pattern,
    locked: Locked,
}

impl StylePattern<'_> {
    pub(crate) fn set_notes(
        &mut self,
        channel: usize,
        notes: impl IntoIterator<Item = NoteEvent>,
    ) -> bool {
        !self.locked.contains(channel) && self.pattern.set_notes(channel,
notes)
    }
}

impl Deref for StylePattern<'_> {
    type Target = Pattern;

    fn deref(&self) -> &Pattern {
        self.pattern
    }
}
```

`Deref` deja que un `StylePattern` se lea exactamente como un `Pattern` (`pattern.length()`, `pattern.notes(channel)`), mientras que escribir solo pasa por sus propios métodos. Es una forma corriente en Rust: un **envoltorio** que reparte la lectura gratis y vigila la escritura. El instrumento tiene una puerta del mismo tipo, `Project::set_sound`, que rechaza un canal bloqueado. Un test aplica cada estilo, ritmo, estilo guardado, estilo de serie y de micromodo con cada canal bloqueado por turno, y comprueba que el bloqueado nunca cambia.

La banda toca alrededor de un bloqueo

La banda escribe sus propios patrones, una parte por músico, así que tiene sus propias reglas (en `band/locks.rs`):

- un músico cuyo canal está bloqueado se salta: su parte se queda como está;
- los canales de batería bloqueados conservan sus notas mientras los demás tambores cambian;
- si tus propios Teclados o Bajo están bloqueados y tocan, el piano o el bajo de la banda se quedan callados, para no doblar la parte de la canción;
- cuando la banda se va (**Sin banda**), una parte de la banda bloqueada se queda en la canción como tuya.

La banda guarda una huella de aquello desde lo que toca, para saber cuándo escribir otra vez (ver Una parte que editas sigue siendo tuya). El bloqueo cuenta en esa huella solo cuando hay algo bloqueado, así que una canción sin nada bloqueado conserva la huella que tenía.

En el archivo

Las cuatro cosas se guardan con la banda, y cada una se deja fuera del archivo cuando no se usa, así que una canción que no las usa se escribe como se escribiría sin ellas:

```
#[serde(default, skip_serializing_if = "Form::is_plain")]
pub form: Form,
#[serde(default, skip_serializing_if = "Vec::is_empty")]
pub variations: Vec<VariationMarker>,
```

`Form::is_plain` es `true` para la forma que tiene una canción sin la fila Forma: sin introducción, una vuelta, sin final. Una canción que usa la forma necesita al menos el formato 14, las variaciones el formato 15, un canal bloqueado el formato 23, y los estilos de sección el formato 24 (ver Formato del archivo de proyecto).

Cómo se prueba

Cada módulo prueba su parte:

- `form.rs`: `choruses_play_the_sheet_again_and_again` comprueba que la lista de reproducción muestra la vuelta una vez y que el arreglo la toca tres veces (192 pasos para una vuelta de 64), con «2 de 3» en el paso 70; `an_intro_leads_into_the_chorus` y `the_intro_moves_the_melody_along` ponen y quitan introducciones, y comprueban que la melodía se corre y que las partes de la banda no quedan congeladas; `an_intro_is_a_turnaround_into_the_first_chord` comprueba turnarounds hacia acordes mayores y menores, las grafías hacia Ebm y Gb, y un blues; `the_ending_holds_the_tonic` comprueba que el último compás es un acorde sostenido 16 pasos, con un solo crash; `the_form_is_saved` comprueba que el archivo se lee igual.
- `variation.rs`: `markers_keep_only_changes`, `the_drummer_moves_to_the_ride`, `the_drums_fill_into_a_change`, `variation_b_is_louder`, `players_switch_to_their_busier_way`, `a_jazz_ballad_walks_in_variation_b`,

`markers_move_with_the_song` y `a_copied_section_plays_the_same_variations`.

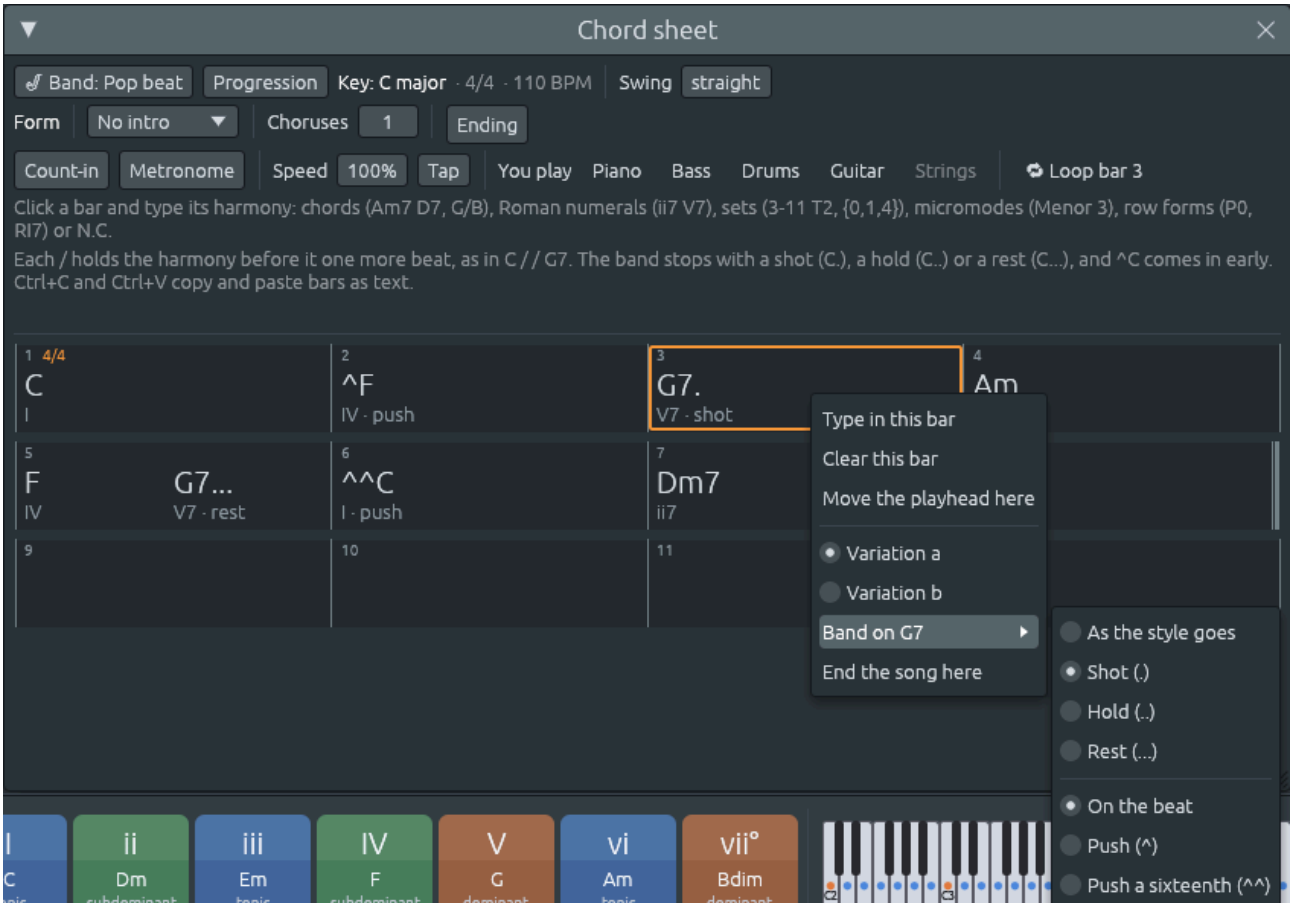
- `sections.rs`: `a_section_plays_as_its_style_does`, `a_section_plays_its_drums_and_starts_its_pattern_on_its_first_bar`, `a_section_without_drums_rests_them`, `a_swing_section_swings_in_a_straight_song` y `markers_move_with_the_song_and_are_saved`.
- `lock.rs`: `no_library_choice_writes_a_locked_channel` es el test que prueba cada estilo con cada canal bloqueado; `removing_a_channel_renumbers_the_ones_after_it` comprueba `without`. `band/locks.rs` comprueba cada regla de la banda, y que `no_band_leaves_the_locked_parts_to_the_song`.

En la aplicación, `ux_tests.rs` hace clic en el candado del rack y lo deshace (`the_rack_locks_a_channel_and_undo_unlocks_it`), prueba Doo-wop y luego Bossa nova en un patrón alrededor de unos Teclados bloqueados, y comprueba que los avisos y la hoja de acordes dicen qué se conserva (`the_library_and_the_band_say_what_is_kept`).

Golpes, sostenidos, silencios y anticipaciones

Archivos: `crates/lineup-core/src/chord_sheet.rs`, `crates/lineup-core/src/band/stops.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/file.rs` y `crates/lineup-app/src/chord_sheet.rs`

Un estilo toca el mismo groove compás tras compás, y eso es lo que lo hace un estilo. Pero escucha a una banda de verdad: al final de una frase todos golpean un acorde juntos y se detienen en seco, o sostienen un acorde bajo quien canta, o dejan de tocar un compás para que el baterista o el solista quede solo, o entran con el acorde siguiente un poco antes del pulso. Los arreglistas escriben esos momentos en la partitura, y son lo que hace que una banda suene arreglada y no en bucle. La hoja de acordes tiene marcas para ellos, y las partituras de las canciones de demostración usan las mismas marcas.



Probarlo

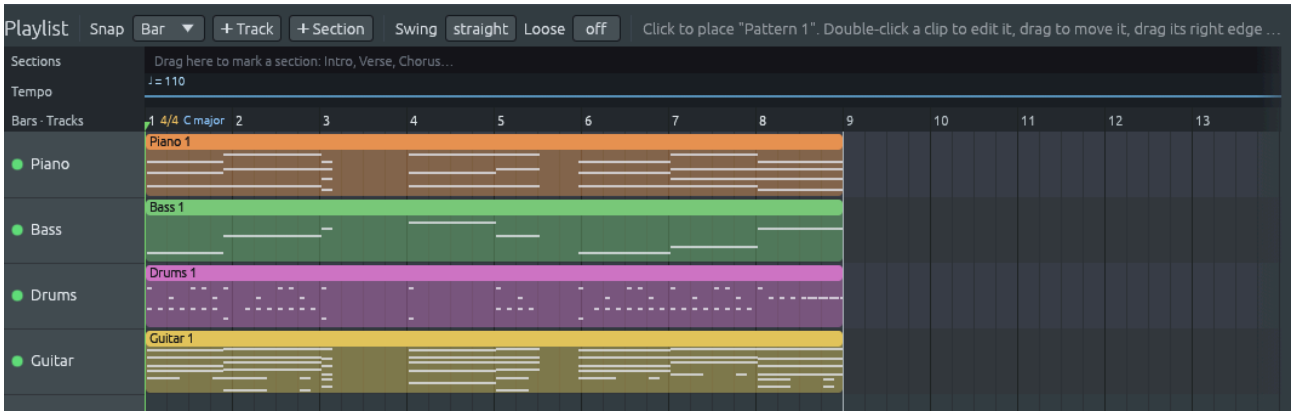
Pulsa **Ctrl+N** para una canción nueva con banda, y escribe estos compases en la hoja de acordes, uno en cada compás:

```
| C | ^F | G7. | Am.. | F G7... | ^^C | Dm7 | G7 |
```

Bajo cada armonía marcada, una palabra dice lo que hace la banda ahí: **anticipación**, **golpe**, **sostenido** o **silencio**. Pasa el ratón por encima de una para leer la frase entera, como «Sostenido: la banda lo toca y lo sostiene hasta la siguiente armonía». Pulsa Espacio y escucha:

- compás 2: la banda toca F en el «y» del pulso 4 del compás 1, una corchea antes, y lo deja sonar sobre la barra de compás, donde está escrito;
- compás 3: toda la banda golpea G7 una vez, corto, con un bombo y un crash, y luego nada hasta el compás 4;
- compás 4: Am golpeado y sostenido todo el compás;
- compás 5: F en el estilo durante dos pulsos, luego silencio para G7;
- compás 6: C una semicorchea antes;
- compases 7 y 8: el estilo otra vez.

La lista de reproducción muestra lo mismo: las partes de la banda tienen huecos y notas largas donde están las marcas.



Las marcas también se ponen sin escribir: haz clic derecho en un compás y elige **Banda en G7** (una entrada por cada armonía del compás), y luego una parada o una anticipación. **Según el estilo** y **En el pulso** quitan las marcas. Como todo cambio de la hoja, la banda vuelve a tocar, deshacer lo devuelve, y las marcas se copian con los compases, también como texto (ver Hojas de acordes como texto):

```
| C. | ^F | G7.. Am... | ^^C |
```

La música: romper el groove

Golpes

Un **golpe** (en inglés *shot*, *hit*, *stab* o *kick*) es un acorde que toda la banda toca a la vez, corto y acentuado, seguido de silencio. Las partituras de big band están llenas de ellos: los vientos golpean un acorde, y el baterista lo toca con la banda en el platillo crash y el bombo, a menudo con un fill corto antes para «prepararlo». Una tanda de golpes, uno al principio de cada compás o de cada dos, se llama **stop-time**: la banda toca solo los golpes y deja el resto del compás a quien canta, al solista o a un bailarín de claqué. En las estrofas de «Hoochie Coochie Man» de Muddy Waters la banda toca solo un riff corto al principio de cada compás y la voz llena el silencio; es uno de los grooves de stop-time más conocidos del blues.

Aquí un golpe es una corchea. Cada músico golpea el acorde a su manera (el piano su disposición, el bajo la nota grave, la guitarra un rasgueo), un poco más fuerte que el estilo, y la batería toca un bombo y un crash. Luego la banda calla hasta la siguiente armonía. Así funciona el compás 3 de arriba: el silencio es parte de la música, y la armonía sigue siendo G7 para quien toque encima.

Sostenidos

Un **sostenido** es un acorde golpeado y sostenido. Las bandas lo usan al final de una frase, para un acorde grande bajo la última palabra de un verso, o para dejar que un solista toque libre sobre una armonía. No es un calderón: un calderón detiene el pulso, y la banda espera al director, mientras que un sostenido dura exactamente hasta la siguiente armonía, y el pulso sigue por debajo. Cada músico golpea el acorde con un bombo y un crash, como en un golpe, y lo deja sonar. Una forma de serie, que la banda toca como tricordios uno tras otro, se sostiene de a un tricordio: cada uno se golpea al llegar y se sostiene hasta el siguiente.

Silencios

Un **silencio** es la banda dejando de tocar. Los arreglistas lo llaman un **corte** (*break*): en el jazz, la banda se detiene un compás o dos y el solista toca solo (el famoso «break» al final de la primera vuelta de un tema), y en el funk y el hip hop los compases donde solo toca el baterista le dieron nombre al «breakbeat». Aquí descansa toda la banda, también la batería; escribe tú el corte de la batería en el piano roll, o tócalo con **Tocas tú** (ver Herramientas de práctica).

Anticipaciones

Una **anticipación** (en inglés *push*) es un acorde tocado antes del pulso al que pertenece y sostenido sobre él. En C | ^F, el acorde F está escrito en el pulso 1 del compás 2 pero se toca en el «y» del pulso 4 del compás 1, y la banda no vuelve a golpear en el pulso 1. Una partitura lo escribe como una corchea ligada sobre la barra de compás:

```
compás 1            compás 2
1   2   3   4 y   | 1   2   3   4
C   C   C   C F---|-F   F   F   F
              ^ aquí se golpea el F, y se sostiene sobre el pulso 1
```

Sin la anticipación, el primer tiempo es el momento más fuerte del compás, y todo el mundo lo espera. Con ella, el acorde llega antes y el pulso 1 queda en silencio: la música se inclina hacia delante, como si no pudiera esperar. Eso es la **síncopa**, un acento fuera del pulso, y las anticipaciones están por todas partes en el pop, el rock, el funk y la música latina. Los bajistas de salsa anticipan casi todos los acordes: en el **tumbao**, el bajo toca la fundamental del acorde siguiente en el pulso 4, sostenida sobre la barra de compás. Una anticipación de semicorchea (`^^C`) es más sutil, el tirón del funk y de la música latina rápida.

Una anticipación se puede combinar con una parada: `^G7.` es un golpe una corchea antes.

Por qué puntos y circunflejos

Los puntos después de una armonía y los circunflejos antes son una manera corta de escribir las marcas sin soltar el teclado: un punto para la más corta (un golpe), tres para el silencio más largo (un silencio). El circunflejo `^` se parece al **marcato**, el signo de acento que los músicos escriben sobre una nota para tocarla con fuerza, y una anticipación se toca con acento. `N.C.` (sin acorde) conserva sus propios puntos: son parte de su nombre.

Cómo funciona

Dos marcas opcionales en un marcador

Una armonía de la hoja es un `ChordMarker`, con dos campos opcionales, `push` y `stop`, `None` cuando no está marcada (ver Cómo se guarda la hoja). Sus tipos son dos enums pequeños:

```
pub enum Stop {
    /// "C.": the whole band strikes the chord once, short, with a kick
    /// and a crash cymbal, and then is silent.
    Shot,
    /// "C..": the band strikes the chord and holds it.
    Hold,
    /// "C...": the band is silent. The harmony goes on, for whoever
    /// plays over it.
    Rest,
}
```

```
pub enum Push {
    /// "^C": an eighth note early.
    Eighth,
    /// "^^C": a sixteenth note early.
    Sixteenth,
}
```

Cada uno sabe escribirse (`marks`: `"."`, `".."`, `"..."`, `"^"`, `"^^"`), cuántos pasos antes llega una anticipación (`steps`: 2 o 1), y una frase para las pistas (`describe`). Los dos campos quedan fuera del archivo cuando son `None`, así que una canción sin marcas se escribe como se escribiría sin ellas. El formato del archivo solo pasa a ser 18 cuando algún marcador tiene una marca, cosa que se comprueba con `any`:

```
let has_breaks = playlist
    .chord_markers()
    .iter()
    .any(|marker| marker.push.is_some() || marker.stop.is_some());
```

`ChordMarker::pushed` y `ChordMarker::stopped` toman un marcador por valor y lo devuelven cambiado, así que un marcador marcado se lee como una frase: `ChordMarker::new(16, f).pushed(Some(Push::Eighth))`.

Leer las marcas

Escribir `^^Am7/G..` tiene que dar una anticipación, la armonía `Am7/G` y una parada. `read_marks` quita las marcas de los dos extremos y las cuenta:

```
let name = text.trim_start_matches('^');
let push = match text.len() - name.len() {
    0 => None,
    1 => Some(Push::Eighth),
    2 => Some(Push::Sixteenth),
    _ => {
        return Err(
            tr("a push is written ^ (an eighth early) or ^^ (a
sixteenth).").to_string(),
        );
    }
};
if is_no_chord(name) {
    return Ok((push, name, None));
}
let bare = name.trim_end_matches('.');
let stop = match name.len() - bare.len() {
    0 => None,
    1 => Some(Stop::Shot),
    2 => Some(Stop::Hold),
    3 => Some(Stop::Rest),
    _ => {
        return Err(tr(
            "after a harmony, . is a shot, .. a hold and ... a rest: no
more dots.",
        )
        .to_string());
    }
};
Ok((push, bare, stop))
```

`trim_start_matches` devuelve un **slice** de la misma cadena sin los circunflejos, no una copia, así que la diferencia de las dos longitudes es cuántos quitó. Un circunflejo y un punto ocupan un byte cada uno en UTF-8, así que contar bytes es contar marcas. Un sin acorde (`N.C.`, `NC`) se deja entero, o su último punto se volvería un golpe. El resto, `bare`, pasa al analizador que lee acordes, números romanos, conjuntos y series.

Escribir un marcador de vuelta es lo mismo al revés, `format!` con las marcas a los dos lados:

```
pub fn text(&self) -> String {
    let carets = self.push.map_or("", Push::marks);
    let dots = self.stop.map_or("", Stop::marks);
    format!("{carets}{}{dots}", self.harmony)
}
```

`map_or("", Push::marks)` da `""` para `None` y, si no, llama a `Push::marks` con el valor de dentro. Un método se puede pasar por su nombre así, como una función que recibe el valor como primer argumento.

Después del estilo: reescribir cada parte

Los músicos no necesitan saber nada de marcas. Cada estilo escribe su parte como siempre, y luego un último paso, `Breaks::apply` en `band/stops.rs`, la reescribe. Primero `Breaks::of` convierte los marcadores en trechos de la canción:

- una **parada** dura desde su armonía (antes, si además está anticipada) hasta donde entra la armonía siguiente (antes, si esa está anticipada);
- una **anticipación** dura desde donde se toca hasta la armonía siguiente. Una anticipación nunca llega hasta la armonía anterior: si F está escrito un paso después de C, no puede entrar antes que C.

Luego, para cada músico, las paradas:

```
for stop in &self.stops {
    clear(&mut notes, stop.start, stop.end);
    let Some(harmony) = harmony_at(stop.written) else {
        continue;
    };
    match stop.stop {
        Stop::Shot => notes.extend(accented(strike(&Played {
            start: stop.start,
            end: (stop.start + SHOT_STEPS).min(stop.end),
            ..harmony.clone()
        }))),
        // A row form is played as trichords one after another:
        // each is struck as it comes, and held.
        Stop::Hold => {
            // ...
        }
        Stop::Rest => {}
    }
}
```

`clear` quita las notas que empiezan en el trecho, y acorta las que empezaron antes y seguirían sonando, para que a un golpe le siga un silencio de verdad. `accented` sube

cada nota al menos a la velocidad 100, un poco más fuerte que el estilo. `SHOT_STEPS` vale 2, una corchea. Un silencio no golpea nada: su brazo está vacío.

Una anticipación vacía el pulso en que está escrita y golpea la armonía antes. Cuánto se sostiene depende del músico: hasta donde sigue la parte de ese músico. En la canción de arriba, el piano del Ritmo pop toca corcheas, así que el F anticipado se sostiene desde el «y» del pulso 4 hasta el «y» del pulso 1, una negra; un bajo que tocara blancas lo sostendría hasta el pulso 3.

Cada músico golpea a su manera

¿Qué golpea el piano, y qué el bajo? `apply` no lo sabe; recibe un closure que sí lo sabe:

```
pub(crate) fn apply(
    &self,
    mut notes: Vec<NoteEvent>,
    harmonies: &[Played],
    strike: impl Fn(&Played) -> Vec<NoteEvent>,
) -> Vec<NoteEvent> {
```

El piano pasa sus acordes sostenidos, el bajo una función que toca una nota, y la guitarra y las cuerdas comprueban primero si tocan en esa variación (unas cuerdas que solo entran en la variación b no deberían dar de pronto un golpe en la variación a):

```
score.breaks.apply(notes, &score.harmonies, |played| {
    match extras.guitar_in(variation_at(played.start)) {
        Some(_) => guitar::play(
            GuitarPart::Strum(Comping::Held),
            Comping::Held,
            &score.bars,
            std::slice::from_ref(played),
        ),
        None => Vec::new(),
    }
})
```

Este closure usa `extras`, `variation_at` y `score`, variables de la función que lo rodea. Un closure que **captura** lo que lo rodea lo lleva consigo, así que no es un simple puntero a función como el `fn(&Played) -> Vec<Note>` que recibe `strummed` (ver Enums que llevan datos, y una función como parámetro), y su tipo no tiene un nombre que puedas escribir. `impl Fn(&Played) -> Vec<NoteEvent>` dice «cualquier tipo que se pueda llamar así», y el compilador escribe una versión de `apply` para

cada closure que se le pasa. Una función corriente también sirve: el bajo pasa `bass_struck` por su nombre.

`std::slice::from_ref(played)` convierte un `&Played` en un slice de uno, `&[Played]`, sin copiarlo, porque `guitar::play` recibe un slice.

¿Por qué el bajo no usa su propio estilo sostenido, `Bass::Held`? Porque ese vuelve a golpear en cada barra de compás, y un sostenido de dos compases se golpearía dos veces. `bass_struck` toca una sola nota para todo el trecho.

La batería

La batería es distinta: sus notas vienen en pares, `(channel, note)`, un canal por tambor. `apply_to_drums` quita el groove de cada trecho y añade los golpes, en los canales del bombo y del crash:

```
notes.retain(|(_, note)| !(stop.start..stop.end).contains(&note.start));
```

`(a..b).contains(&x)` se lee mejor que `a <= x && x < b`, y para una anticipación, `(push.start..=push.written)` con `..=` incluye el paso en que está escrita. El bombo y el crash necesitan canales antes de poder tocar: un estilo sin crash recibe uno, solo cuando la hoja tiene una marca, a través de

```
.chain(
    (!breaks.is_empty())
        .then_some([Drum::Kick, Drum::Crash])
        .into_iter()
        .flatten(),
)
```

`then_some` convierte un `bool` en un `Option`: `Some([...])` cuando es verdadero, `None` cuando es falso. `into_iter().flatten()` da entonces los dos tambores o nada, así que la lista de tambores que necesita el estilo crece solo cuando hace falta.

La banda vuelve a tocar cuando cambia una marca

Para eso no hace falta código. La banda recuerda una huella de todo aquello desde lo que toca, incluidos los marcadores de acorde, hecha con su texto `Debug` (ver Una parte que editas sigue siendo tuya). El texto `Debug` de un marcador incluye sus marcas, así que cambiar un golpe por un sostenido cambia la huella, y la banda vuelve

a tocar. Las partes que editaste a mano siguen congeladas hasta **Volver a tocarla**, como siempre.

Cómo se prueba

Los tests de `band/stops.rs` trabajan solo con `Breaks`, con unas pocas armonías `Played` hechas a mano: `marks_become_stretches_up_to_the_next_harmony`, `a_push_never_reaches_back_past_the_harmony_before`, `a_shot_strikes_once_and_the_band_stops`, `a_hold_lasts_until_the_next_harmony_and_a_rest_is_silent`, `a_hold_over_a_row_form_holds_each_trichord_as_it_comes`, `a_push_comes_in_early_and_is_held_over_the_beat` y `the_drums_stop_and_hit_with_the_band`. En `chord_sheet.rs`, `marks_tell_the_band_where_to_stop_and_push` lee cada marca, sobre acordes, números romanos y micromodos, `N.C.` con y sin anticipación, y los errores (`C....`, `^^^C`, un `^` solo); `marks_are_written_back_as_typed` las vuelve a escribir. `the_band_stops_and_pushes_where_the_sheet_says` en `band.rs` toca una canción con marcas y comprueba cada músico, y `shots_holds_rests_and_pushes_need_format_eighteen` en `file.rs` comprueba que solo las armonías marcadas llevan `push:` y `stop:` en el archivo.

Para oírlo sin escuchar, guarda la canción de arriba y lee su traza de reproducción: el F del piano empieza en el compás 1 en el pulso 4, dos pasos dentro del pulso; en el compás 3 el piano golpea G7 una vez, una corchea; en el compás 4 su Am dura todo el compás; y el C anticipado empieza en el compás 5 en el pulso 4, tres pasos dentro.

El solista

Archivos: `crates/lineup-core/src/harmony/chord_scale.rs`, `crates/lineup-core/src/band/solo.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/file.rs`, `crates/lineup-app/src/chord_sheet.rs` y `crates/lineup-app/src/piano_roll.rs`

La banda acompaña: toca los acordes para que otro toque la melodía o improvise sobre ellos. El **solista** es ese otro, un músico que improvisa una línea sobre la hoja de acordes. Es experimental y está apagado hasta que lo enciendes, porque un solo hecho con reglas es un estudio, no una interpretación. Pero es un estudio útil: cada nota que toca tiene una razón, y el piano roll muestra la razón, así que puedes ver cómo se construye una línea improvisada y probar las mismas reglas en tu propio instrumento.

Probarlo

Pulsa **Ctrl+N** para una canción nueva con banda, y escribe estos compases en la hoja de acordes, un acorde por compás:

```
| Dm7 | G7 | Cmaj7 | A7 | Dm7 | G7 | C6 | C6 |
```

En la **Biblioteca**, aplica **Swing** (en Jazz). Luego abre el menú 🎷 **Banda** de la hoja y marca **Solista (experimental)**. Pasa antes el ratón por encima: la pista dice quién es el solista en este estilo, un saxo tenor. Aparece una pista **Solista** en la lista de reproducción, y el rack de canales recibe un canal Solista con el saxo tenor. Pulsa Espacio: sobre la banda, el saxo toca frases de dos compases en corcheas con swing, con una respiración entre ellas.

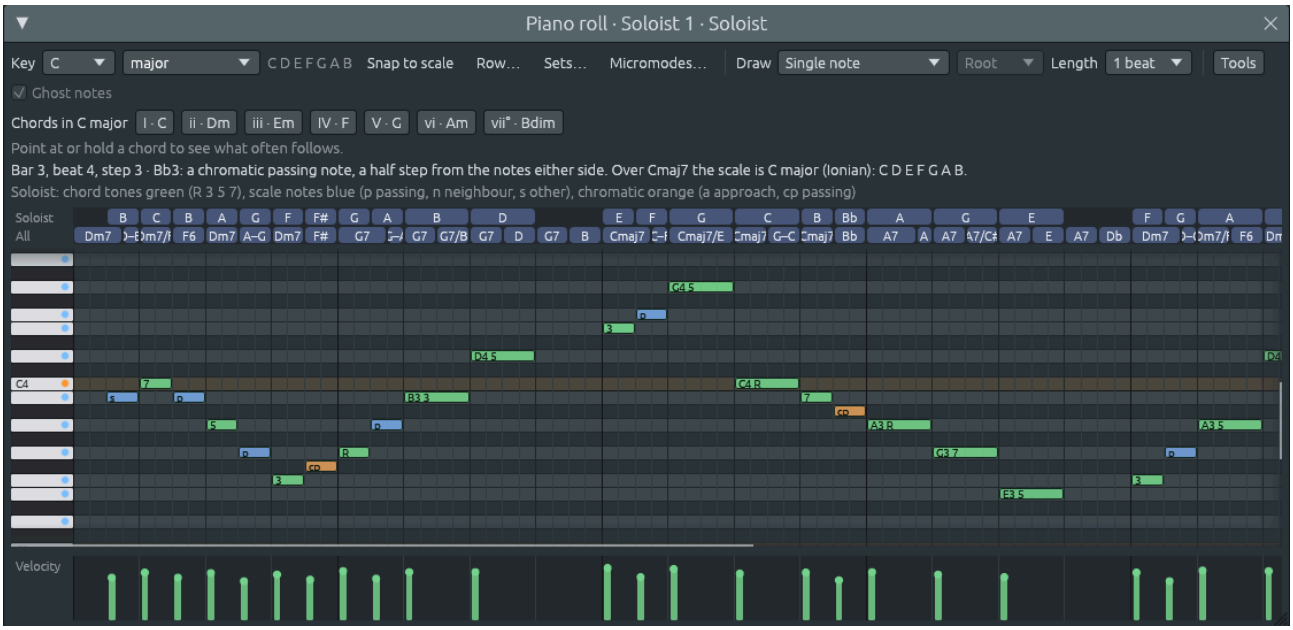
Otro solo, en el mismo menú, toca uno distinto; deshacer trae de vuelta el anterior. Desmarca el solista y la pista y el canal se van. Como toda parte de la banda, el solo se vuelve a tocar cuando cambia la hoja, y si editas sus notas se queda como lo dejaste hasta que pulsas **Volver a tocarla** (u **Otro solo**).

Ahora haz doble clic en el clip del solista para abrir su parte en el piano roll. Sus notas tienen el color de lo que son:

- **verde** para las notas del acorde, marcadas con cuál es: `R` para la fundamental, `3`, `5`, `7` (y `6`, `9` cuando el acorde las tiene), o `•` para una nota de un conjunto,

que no tiene fundamental desde la que contar;
- **azul** para las demás notas de la escala del acorde: `p` una nota de paso, `n` una bordadura, `s` cualquier otra;
- **naranja** para las notas cromáticas, de fuera de la escala: `a` una aproximación, `cp` una nota de paso cromática.

Señala una nota, o toca la canción, y la línea sobre la cuadrícula dice qué es, en una frase:



En este solo, el compás 3 termina con Si3 (la 7ª de Cmaj7) en el pulso 4, luego Si♭3 en el «y», y luego La3 en el pulso 1 del compás 4, la fundamental de A7: tres notas a un semitono una de otra, bajando hacia el acorde siguiente. El Si♭ no está en ninguno de los dos acordes, y no pasa nada: está en la mitad débil del pulso y lleva a algún sitio.

La música: cómo se construye una línea improvisada

Las notas del acorde en el pulso

Pregúntale a un profesor de jazz cómo empezar a improvisar sobre una progresión, y la primera regla es casi siempre la misma: **toca notas del acorde en los pulsos fuertes, y lo que las una en medio**. En el pulso, el oído compara la melodía con la armonía; entre los pulsos la melodía está en movimiento y el oído perdona más. Una línea que cae en una nota del acorde en cada pulso «dibuja los acordes»: se oyen en ella aun sin la banda.

Las notas del acorde que más dicen son la **3ª** y la **7ª**, las **notas guía**. Dicen si un acorde es mayor o menor y qué tipo de séptima tiene, y de un acorde al siguiente de un ii–V–I se mueven por semitonos: la 7ª de Dm7 (Do) cae a la 3ª de G7 (Si), y la 7ª de G7 (Fa) cae a la 3ª de Cmaj7 (Mi). Muchos solos son notas guía unidas.

El solista elige, para cada pulso, la nota del acorde **más cercana** a donde va: la línea sube o baja, y mantiene su dirección hasta girar en el borde del registro del instrumento, o de vez en cuando por su cuenta. Una de cada cinco veces se salta una nota del acorde y salta a la siguiente. Eso da líneas que se mueven sobre todo por pasos pequeños, como las melodías, con un salto aquí y allá.

Escalas de acorde

Entre las notas del acorde van otras notas, y cuáles depende del acorde: sobre Dm7, ¿la nota entre La y Do es un Si o un Si♭? La enseñanza del jazz responde con las **escalas de acorde**: cada acorde recibe una escala de siete notas que lo contiene, y las notas de en medio salen de esa escala.

La escala no se elige solo a partir del acorde, sino del acorde **en su tonalidad**. Sobre Dm7 en Do mayor, el Si es natural, porque la música está en Do: la escala es Do mayor desde Re, Re Mi Fa Sol La Si Do, llamada **Re dórico**. Los modos de Do mayor son lo que tocan los acordes de Do mayor:

Acorde en Do mayor	Su escala	Notas
Cmaj7 (I)	Do jónico	Do Re Mi Fa Sol La Si
Dm7 (ii)	Re dórico	Re Mi Fa Sol La Si Do
Em7 (iii)	Mi frigio	Mi Fa Sol La Si Do Re
Fmaj7 (IV)	Fa lidio	Fa Sol La Si Do Re Mi
G7 (V)	Sol mixolidio	Sol La Si Do Re Mi Fa
Am7 (vi)	La eólico	La Si Do Re Mi Fa Sol
Bm7♭5 (vii)	Si locrio	Si Do Re Mi Fa Sol La

Siete nombres para las mismas siete notas. Los nombres importan porque dicen cuál es la fundamental y dónde caen los semitonos desde ella: el dórico es una escala menor con 6ª mayor (Si sobre Re), el lidio una escala mayor con 4ª aumentada (Si sobre Fa), el mixolidio una escala mayor con 7ª menor (Fa sobre Sol). Esas son las notas que le dan a cada acorde su color en la tonalidad.

Los demás acordes reciben su escala como la elegiría un músico que piensa en la tonalidad, en este orden:

1. **Un acorde de la tonalidad** toca la escala de la tonalidad desde su fundamental, como en la tabla. En una tonalidad menor, un acorde que usa la 7ª elevada (E7 en La menor, con Sol♯) toca la **menor armónica** de la tonalidad desde su fundamental: La menor armónica desde Mi, Mi Fa Sol♯ La Si Do Re, llamada **Mi frigio dominante**.
2. **Un acorde prestado de la tonalidad homónima** toca la escala de esa tonalidad: Fm en Do mayor viene de Do menor, así que toca Do menor desde Fa, que es Fa dórico. B♭7 en Do mayor (también de Do menor) toca Si♭ mixolidio.
3. **Una dominante secundaria** toca la escala del acorde al que lleva. D7 en Do mayor es el V7 de Sol, así que toca Sol mayor desde Re, Re mixolidio. A7 en Do mayor es el V7 de Dm, un acorde menor, así que toca Re menor armónica desde La: La Si♭ Do♯ Re Mi Fa Sol, La frigio dominante. Sobre A7, un Si♭ es una nota de la escala, que apunta al acorde de Re menor que sigue.
4. **Cualquier otro** toca la escala que suele recibir su tipo: **lidio** para un acorde mayor de fuera de la tonalidad (el Re♭ de la sexta napolitana en Do), **lidio dominante** para una séptima de dominante que viene de fuera y no lleva a ninguna parte, **disminuida semitono-tono** para 7♭9 y 7♯9, **disminuida tono-semitono** para una séptima disminuida, **de tonos enteros** para un acorde aumentado, **locrio** para uno semidisminuido, **dórico** para un acorde menor. En una tonalidad que no es una escala de siete notas (una escala de blues, una pentatónica), cada acorde recibe la escala habitual de su tipo, con los acordes mayores sobre la escala mayor (jónica) y las dominantes sobre la mixolidia.

Una regla más: **las notas del propio acorde ganan**. E9 en Do mayor lleva a Am, así que parte de La menor armónica (Mi Fa Sol♯ La Si Do Re), pero su 9ª es Fa♯, y Fa y Fa♯ no pueden estar los dos en la escala con el mismo nombre. El Fa♯ del acorde reemplaza al Fa: Mi Fa♯ Sol♯ La Si Do Re, **mixolidia ♭6**.

Hay una segunda escuela, que da a cada acorde su propia escala sin preguntar por la tonalidad (dórico para todo m7, mixolidio para todo 7). Es más fácil de enseñar y a menudo suena igual. Pensar en la tonalidad está más cerca de cómo se escribieron los temas, y mantiene iguales de un acorde a otro las notas que los acordes de la tonalidad tienen en común.

Las notas entre los pulsos

Cada tipo de nota entre las notas del acorde tiene un nombre, y la mayoría son más viejos que el jazz: los libros de armonía las llaman **notas extrañas al acorde**.

- Una **nota de paso** llena el hueco entre dos notas del acorde a una tercera, por grado conjunto, en la misma dirección: Do **Re** Mi sobre C.

- Una **bordadura** (o nota auxiliar) se aleja un paso de una nota del acorde y vuelve: Mi **Fa** Mi.
- Una **nota de paso cromática** llena un tono entero con el semitono de en medio: Re **Re♯** Mi.
- Una **aproximación cromática** entra en una nota del acorde desde un semitono por debajo (o por encima): **Fa♯** Sol. Cuando el solista empieza una frase con una nota justo antes del pulso, una **anacrusa**, suele ser una de estas.
- Cualquier otra nota de la escala, alcanzada o dejada con un salto, se marca `s`, y una nota de fuera de la escala que no hace nada de esto se marca `x`. El solista no toca notas `x`, pero una parte que edites sí puede tenerlas.

Aquí un paso es un semitono o un tono, o, en la menor armónica, la segunda aumentada entre su 6ª y su 7ª (Fa y Sol♯ en La), porque esas dos están una al lado de la otra en la escala.

Las notas de paso cromáticas en el «y» del pulso son parte del sonido del **bebop**, el jazz de Charlie Parker y Dizzy Gillespie: un semitono añadido entre dos notas de la escala mantiene las notas del acorde en los pulsos cuando una línea corre en corcheas. Los profesores de jazz lo escribieron más tarde como las «escalas bebop».

Frases y respiración

Quien canta tiene que respirar, y quien toca un instrumento de viento también, así que las líneas vienen en **frases**. El solista toca frases de dos compases. Cada una puede empezar un pulso tarde o con una anacrusa, y deja uno o dos pulsos vacíos al final. Una línea que nunca se detiene suena a ejercicio; los silencios son lo que la hace sonar a alguien que habla. Luego las corcheas llevan swing como la banda (ver Swing), y el fraseo de melodía de la biblioteca da forma a su volumen, como hace con las melodías de las demostraciones.

El solista toca sobre las vueltas, no sobre la introducción ni el final (ver La forma), y sigue tocando a través de los golpes y los silencios (ver Golpes, sostenidos, silencios y anticipaciones): la banda se detiene precisamente para que un solista se oiga solo. Sobre «sin acorde» se queda callado. Sobre un conjunto de clases de alturas, un micromodo o una forma de serie no hay escala que elegir, y el solista toca las notas del propio conjunto, en los pulsos y entre ellos.

Quién toca

Cada familia de estilos tiene su solista, como instrumento General MIDI y con el registro en el que toca. La familia es la de los instrumentos de la banda (su

Formación de instrumentos):

Estilos	Solista	Registro
Pop	saxo alto	Sol3 a Sol5
Rock	guitarra saturada	La3 a Do6
Blues	armónica	Do4 a Do6
Jazz	saxo tenor	Re3 a Fa5
Soul, funk y R&B	saxo alto	Sol3 a Sol5
Hip hop y lo-fi	vibráfono	Do4 a Do6
Electrónica	solista de sierra	Do4 a Do6
Latina	trompeta	Si♭3 a La5
Tango	acordeón de tango	Sol3 a Do6
Folclore latinoamericano	flauta de pan	Mi4 a Mi6
Flamenco	guitarra de nailon	Mi3 a La5
Balcánica y turca	clarinete	Re3 a Si♭5
Reggae y ska	trompeta	Si♭3 a La5
Country y folk	violín folk	Sol3 a Mi6
Clásica y de cine	flauta	Fa4 a Fa6

El acordeón de tango es el sustituto que General MIDI tiene para el bandoneón. Los registros son el centro cómodo de cada instrumento, no todo él: los solos se quedan donde un instrumento canta.

Cómo funciona

Elegir una escala de acorde

`chord_scale`, en `harmony/chord_scale.rs`, sigue las cuatro reglas de arriba, un `if` detrás de otro:

```
pub fn chord_scale(chord: Chord, key: Scale) -> PitchSet {
    if !key.kind.is_tonal() || !key.is_heptatonic() {
        return usual_scale(chord, false);
    }
    let harmonic = |root: PitchClass| Scale::new(root,
ScaleKind::HarmonicMinor);
    let source = if key.is_chord_of_key(chord) {
        match key.uses_raised_seventh(chord) {
            true => harmonic(key.root),
            false => key,
        }
    } else if let Some(parallel) = key.borrowed_from(chord) {
        match parallel.is_diatonic(chord) {
            true => parallel,
            false => harmonic(key.root),
        }
    } else if let Some(target) = key.secondary_dominant_target(chord) {
        match target.quality.is_minor() {
            true => harmonic(target.root),
            false => Scale::new(target.root, ScaleKind::Major),
        }
    } else {
        return usual_scale(chord, true);
    };
    with_chord_notes(source, chord)
}
```

Casi toda la teoría musical está en otra parte: `is_chord_of_key`, `borrowed_from` y `secondary_dominant_target` son lo que usa la hoja de acordes para nombrar acordes prestados y dominantes secundarias (ver Los métodos de la tonalidad). Esta función solo los pone en el orden en que un músico haría las preguntas.

Tres trozos de Rust que merecen atención:

- Un `if` es una **expresión** en Rust: `let source = if ... { a } else if ... { b } else { ... };` le da a `source` el valor de la rama que se ejecutó. La última rama no da un valor; hace `return` desde toda la función, lo que está permitido porque `return` tiene el tipo `!` («never», nunca), que encaja en cualquier sitio.
- `else if let Some(parallel) = key.borrowed_from(chord)` comprueba y desempaqueta a la vez: la rama se ejecuta solo cuando hay una tonalidad homónima, y dentro de ella `parallel` es esa tonalidad.
- `harmonic` es un **closure**, una función pequeña escrita en su sitio, usada tres veces. `|root: PitchClass| Scale::new(...)` recibe una fundamental y da una escala.

`usual_scale` es un solo `match` sobre la especie del acorde, con una condición para los acordes de fuera de la tonalidad:

```
Major | Add9 | Major6 | Six9 | Major7 | Major9 | Major13 if outside =>
LYDIAN,
Major | Add9 | Major6 | Six9 | Major7 | Major9 | Major13 | Sus2 => IONIAN,
```

y termina uniendo la escala y el acorde. Un `PitchSet` son doce bits en un número, uno por cada clase de alturas, así que unir dos conjuntos es un o bit a bit de los números:

```
PitchSet::from_bits(scale.bits() | chord.pitch_set().bits())
```

Las notas del acorde ganan, letra a letra

```
fn with_chord_notes(source: Scale, chord: Chord) -> PitchSet {
    let names = source.chord_notes(chord);
    let mut set = chord.pitch_set();
    for pitch_class in source.pitch_classes() {
        let letter = source.spell(pitch_class).letter;
        let replaced = names.iter().any(|name| {
            name.letter == letter
                && name.pitch_class() != pitch_class
                && !source.contains(name.pitch_class())
        });
        if !replaced {
            set.insert(pitch_class);
        }
    }
    set
}
```

El conjunto empieza con las notas del acorde, y cada nota de la escala de origen se le une salvo que una nota del acorde tenga la **misma letra** pero otra altura, de fuera de la escala. Las letras, no las alturas, deciden qué nota se reemplaza: el Fa♯ de E9 reemplaza a Fa, no a Sol. Por eso la escala se escribe con nombres de nota (`NoteName`, una letra y una alteración) antes de compararla, y no solo con números.

La última condición, `!source.contains(...)`, evita que las grafías enarmónicas hagan daño. Fdim7 en Do mayor se escribe Fa La♭ Do♭ Mi𝄫, y Do♭ y Mi𝄫 tienen las letras de Do y de Mi: sin la condición sacarían Do y Mi de la escala, aunque Do♭ es Si y Mi𝄫 es Re, las dos ya en Do mayor. El test `every_chord_scale_holds_its_chord` toca cada especie de acorde sobre cada fundamental en varias tonalidades y comprueba que cada escala contiene su acorde y tiene de seis a nueve notas.

Nombrar la escala

Los nombres son una tabla de slices, los semitonos sobre la fundamental y el nombre:

```
pub(crate) const NAMES: &[(&[u8], &str)] = &[
    (IONIAN, "major (Ionian)"),
    (DORIAN, "Dorian"),
    (&[0, 1, 3, 5, 7, 8, 10], "Phrygian"),
    // ...
    (&[0, 1, 4, 5, 7, 8, 10], "Phrygian dominant"),
    // ...
];

pub fn chord_scale_name(scale: PitchSet, root: PitchClass) ->
Option<&'static str> {
    let relative = scale.relative_to(root);
    NAMES
        .iter()
        .find(|(steps, _)| PitchSet::from_semitones(steps) == relative)
        .map(|(_, name)| *name)
}
```

`relative_to(root)` gira la escala para que la fundamental del acorde sea 0. Entonces es el mismo conjunto que una de las filas de la tabla, o ninguno. `&'static str` dice que el nombre vive tanto como el programa: es texto escrito en el código, así que la función puede darlo sin copiarlo. La interfaz en español lo traduce al mostrarlo: «dórico», «frigia dominante».

Qué es cada nota

Los tipos de nota son un `enum`, y uno de ellos lleva datos:

```
pub enum NoteRole {
    ChordTone(Option<u8>),
    Passing,
    Neighbour,
    ChromaticPassing,
    Approach,
    ScaleTone,
    Outside,
}
```

`ChordTone(Some(3))` es la 3ª; `ChordTone(None)` es una nota de un conjunto, que no tiene fundamental desde la que contar. `note_roles` lee una línea y le da un papel a cada nota. Para una nota de la escala mira a sus vecinas, `before` y `after`, que son `None` al principio de una frase o después de un silencio:

```
match (before, after) {
    (Some(before), Some(after))
        if before == after && is_step(before, pitch, over.scale) =>
    {
        NoteRole::Neighbour
    }
    (Some(before), Some(after))
        if (before < pitch) == (pitch < after)
            && before != pitch
            && pitch != after
            && is_step(before, pitch, over.scale)
            && is_step(pitch, after, over.scale) =>
    {
        NoteRole::Passing
    }
    _ => NoteRole::ScaleTone,
}
```

Hacer `match` sobre una **tupla** de dos `Option` comprueba los dos a la vez, y el `if` después de un patrón, una **guarda**, añade una condición. Los brazos se prueban en orden, así que la bordadura (la misma nota antes y después) se encuentra antes que la nota de paso. `(before < pitch) == (pitch < after)` compara dos `bool`: es verdadero cuando la línea sube las dos veces o baja las dos veces, que es lo que hace pasar a una nota de paso.

`note_roles` lee cualquier línea de la misma manera, la del solista o una que hayas editado, así que el piano roll puede pintar cada nota del solista a partir de ella. Su color es un `match` sobre un `Option<NoteRole>`, también con guardas:

```
let mut fill = match role {
    Some(role) if role.is_chord_tone() => NOTE,
    Some(role) if role.is_chromatic() => SOLO_CHROMATIC,
    Some(_) => SOLO_SCALE,
    None if scale.contains(note.pitch.pitch_class()) => NOTE,
    None => NOTE_OUTSIDE,
};
```

Los brazos `None` son lo que recibe cualquier otra parte: verde en la tonalidad, rojo fuera de ella.

Dados que siempre sacan lo mismo

Un solo improvisado necesita azar: qué nota del acorde, si saltar, si empezar tarde. Pero la banda tiene que tocar las mismas notas cada vez que se abre la canción, o las partes que congelaste ya no coincidirían con la hoja. Así que el solista tira dados **pseudoaleatorios**: una función pequeña que convierte un número, la **semilla**, en una

sucesión de números que parecen al azar pero son siempre los mismos a partir de la misma semilla.

```
struct Dice(u64);

impl Dice {
    fn next(&mut self) -> u64 {
        self.0 = self.0.wrapping_add(0x9E37_79B9_7F4A_7C15);
        let mut x = self.0;
        x = (x ^ (x >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
        x = (x ^ (x >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
        x ^ (x >> 31)
    }

    fn chance(&mut self, percent: u64) -> bool {
        self.next() % 100 < percent
    }
}
```

Es **SplitMix64**, un generador conocido: suma una constante impar grande, y luego revuelve los bits con desplazamientos, o exclusivos (`^`) y multiplicaciones. En Rust, un `+` o un `*` corriente que se desborda detiene una compilación de depuración con un panic, porque un desbordamiento suele ser un error. Aquí es lo que se busca, así que `wrapping_add` y `wrapping_mul` lo dicen: el resultado da la vuelta pasado el `u64` más grande, como espera el algoritmo. Los guiones bajos de `0x9E37_79B9_7F4A_7C15` son solo para leerlo.

`struct Dice(u64)` es un **struct tupla**: un struct con un solo campo sin nombre, al que se llega con `self.0`. Envolver el número en un tipo propio quiere decir que lo único que se puede hacer con él es `next`, `chance` y `below`.

La semilla se guarda en el proyecto (`Band.solo: Option<u64>`, `None` sin solista, 1 cuando lo marcas), forma parte de la huella de la banda, y **Otro solo** le suma uno. Una huella nueva quiere decir que la banda vuelve a tocar, así que una semilla nueva toca un solo nuevo, y deshacer, que devuelve la semilla anterior, devuelve el solo anterior.

Improvisar

`improvise` corta la vuelta en frases de dos compases con `chunks(2)`, y cada frase en **pulsaciones**: una pulsación es un pulso, o una negra de un pulso más largo (una blanca en 2/2 tiene dos), así que el solista se mueve en corcheas sea cual sea el compás. La línea es un struct con lo que un músico lleva en la cabeza: hacia dónde va y la última nota que tocó.

```
struct Line {
    soloist: Soloist,
    up: bool,
    last: Option<u8>,
    notes: Vec<NoteEvent>,
}
```

`phrase` primero elige una nota del acorde para cada pulsación, y solo después rellena entre ellas, porque lo que va entre un pulso y el siguiente depende de dónde cae el siguiente. Empieza decidiendo cuánto de la frase tocar:

```
let late = usize::from(count >= 4 && dice.chance(35));
let rest = match count {
    6.. => 1 + dice.below(2),
    3.. => 1,
    _ => 0,
};
let Some(window) = pulses.get(late..count - rest) else {
    return;
};
```

`usize::from(bool)` convierte `true` en 1 y `false` en 0. `6..` y `3..` son **patrones de rango**: 6 o más, después 3 o más (los brazos se prueban en orden). `let ... else` desempaqueta un `Option` o sale: `get` con un rango da un slice de las pulsaciones, o `None` si el rango no cabe, y entonces la frase no se toca.

La anacrusa usa una **let chain**: varias condiciones y un `let` unidos con `&&` en un solo `if`.

```
if index == 0
    && pickup
    && pulse.start >= 2
    && let Some(over) = over_at(overs, pulse.start - 2)
{
    let approach = approach(target, over.scale, dice);
    self.play(approach, pulse.start - 2, 2, BETWEEN);
}
```

Se lee como una frase: si es el primer pulso, y salió una anacrusa, y hay sitio antes, y suena una armonía ahí, toca una nota de aproximación una corchea antes del pulso. Antes de la edición 2024 de Rust, el `let` necesitaba un `if` propio dentro del primero.

`fill` elige qué va entre dos notas del acorde: notas de paso cuando las notas de la escala de en medio caben en el tiempo y cada una está a un paso de la siguiente, una bordadura cuando las dos notas del acorde son la misma, el semitono de en medio

cuando están a un tono, y, para un salto demasiado grande para recorrerlo, las notas de la escala justo antes del destino o un semitono por debajo de él.

Unirse a la banda

El solista es un `Player` más, después del piano, el bajo, la batería, la guitarra, las cuerdas y el órgano. Un estilo nunca lo cuenta entre sus músicos (`Player::Soloist | Player::Horns => false` en `BandStyle::players`); la banda lo añade cuando hay semilla:

```
if self.has_soloist() {
    players.push(Player::Soloist);
}
```

Todo lo que la banda más grande le da a cada músico viene con él: su propio canal con su programa General MIDI, su pista y su patrón, quedar congelado cuando se edita, **Tocas tú** para silenciarlo y tocar el solo tú. Dos cosas son distintas. Las notas del solista pasan por el fraseo de **melodía** de la biblioteca en lugar del de acompañamiento, y los golpes, los sostenidos y los silencios no lo tocan.

Una canción con solista necesita al menos el formato 19. Sin solista, los campos nuevos quedan fuera y el archivo queda como sería sin ellos.

Cómo se prueba

En `harmony/chord_scale.rs` , un test por regla: `chords_of_the_key_play_its_modes` , `the_dominant_of_a_minor_key_plays_harmonic_minor` , `borrowed_chords_play_the_parallel_key` , `secondary_dominants_play_the_scale_of_their_target` y `other_chords_play_the_scale_of_their_kind` , cada uno comprobando las notas escritas de una escala, como `D E F G A B C` ; y `every_chord_scale_holds_its_chord` , descrito arriba.

En `band/solo.rs` , `roles_name_what_each_note_does` lee líneas cortas escritas a mano y comprueba cada papel, y `a_role_is_described_with_its_chord_and_scale` la frase que muestra el piano roll. Tres tests tocan muchos solos, sobre varias progresiones y semillas: `chord_tones_on_the_beats_and_nothing_unexplained_between` comprueba la regla de las notas del acorde en los pulsos y que ninguna nota quede sin explicar, `a_solo_is_one_line_in_range_that_breathes` que las notas nunca se pisan, se

quedan a menos de un tono del registro del solista, callan donde no hay acorde y dejan un silencio de una corchea o más cada dos compases, y `the_same_seed_plays_the_same_solo` los dados.

En `band.rs`, `a_soloist_improvises_when_asked`, `the_soloist_plays_the_chorus_only`, `over_a_set_the_soloist_plays_its_notes_on_the_beats` y `a_soloist_is_saved_in_format_nineteen` comprueban el solista como miembro de la banda.

Tus propios estilos: el editor de estilos

Archivos: `crates/lineup-app/src/style_editor.rs`, `crates/lineup-app/src/app/styles.rs`, `crates/lineup-core/src/library/user_style.rs`, `crates/lineup-core/src/library/apply.rs` y `crates/lineup-core/src/band.rs`

La biblioteca es conocimiento escrito como datos: un estilo es un cifrado, un patrón de comping, una línea de bajo y un ritmo, cada uno unas pocas líneas en una tabla. El **editor de estilos** te deja escribir un groove que esas tablas no tienen, con el rack de canales y el piano roll que ya conoces, y guardarlo como un estilo propio. La banda lo toca luego sobre cualquier hoja de acordes, llevando cada nota al acorde del momento.

Probarlo

1. Abre **Archivo > Nuevo**. Pulsa **F6** para el rack de canales y haz clic en **Editar estilo…**. El editor se abre con una copia de la banda que toca la canción, **Ritmo pop (copia)**, en una ventana titulada **Editor de estilos · rack de canales**.
2. Abre **Empezar desde…**, escribe `bossa` en **Buscar preajustes** y elige un estilo. Haz clic en **Escuchar ×4**.
3. Haz clic en algunos pasos del bombo y de la caja. Haz clic en la vista previa de una nota del piano para abrir el piano roll del editor; dibuja una figura de acordes y pulsa **Espacio** para oírla.
4. Haz clic en **Guardar como…**, llámalo `Mi groove` y guárdalo en la carpeta que abre el diálogo. Cierra el editor, abre la **Biblioteca** y busca **Mi groove** entre los preajustes (bajo **Estilos de usuario** cuando se agrupan por género).
5. Aplícalo con **Batería** y **Partes de acordes y bajo** marcadas y **Progresión de acordes** sin marcar. La canción conserva sus acordes y la banda toca tu patrón sobre ellos. Guarda el proyecto, cierra la app, mueve el archivo `.lineup-style` a otro sitio y abre el proyecto otra vez: la banda lo sigue tocando.

Dónde está

- En el **rack de canales**, **Editar estilo…** abre el estilo que toca la canción («Editar una copia de las partes repetidas del estilo actual», dice su ayuda); **Nuevo estilo…** empieza uno vacío.
- **Ver > Editor de estilos…** abre el editor, o lo trae al frente.

- La **Biblioteca** tiene un botón **Nuevo estilo…** sobre su lista. Haz clic derecho en un estilo o un ritmo: uno integrado ofrece **Editar una copia en el rack de canales…**; uno tuyo, **Editar en el rack de canales…**.

Editar estilo… empieza desde lo que toca la canción. Un estilo tuyo se abre como sí mismo, así que **Guardar estilo** escribe de vuelta en su archivo. Una banda se abre como una copia de todos sus músicos, dos compases de lo que tocan sobre un acorde de C, llamada «… (copia)». Un estilo o un ritmo que se está probando en un patrón se abre como una copia suya, y sin nada que copiar el editor empieza vacío: un bombo, una caja, un charles y un piano, sobre la progresión `I | V | vi | IV`.

Qué edita

Un estilo de usuario es **un patrón que se repite**, de uno a dieciséis compases y de 256 pasos como mucho (así que en 12/8, con 24 pasos por compás, diez compases como mucho). Contiene carriles de batería, uno por cada sonido de batería, y como mucho un carril por cada músico con altura de la banda: piano, bajo, guitarra, cuerdas, órgano, solista y vientos.

La fila de arriba tiene **Empezar desde…**, **Guardar estilo**, **Guardar como…**, **Deshacer**, **Rehacer**, **Escuchar ×4** y **Detener la escucha**. Debajo están **Nombre**, **BPM**, el compás, **Compases** y **Swing**; luego **Progresión**, los acordes que trae el estilo, en números romanos o nombres de acordes con los compases separados por `|`; **Descripción**; y **Asignación de partes**, que dice qué músico es cada fila, de modo que una fila llamada «Rhodes» puede ser el piano.

El rack es el mismo widget que el de la canción, con las filas del borrador: haz clic en los pasos para escribir la batería. Haz clic en la vista previa de una nota para abrir el piano roll, donde dibujar, mover, estirar y el menú Herramientas funcionan como en un patrón. El piano roll del editor tiene su propia ventana y sus propias teclas, y nunca edita el de la canción.

Escuchar ×4 toca el patrón cuatro veces, tal como está escrito sobre Do, y se detiene; **Espacio** lo arranca o lo para mientras el editor tiene el teclado, y **Detener la escucha** lo termina antes. La canción se pausa mientras suena. **Ctrl+S** guarda y **Ctrl+Mayús+S** guarda como.

El editor tiene un historial propio. **Ctrl+Z**, **Ctrl+Mayús+Z** y **Ctrl+Y** deshacen y rehacen el borrador sin tocar el historial de la canción. Nada llega a la canción hasta que aplicas el estilo guardado, y entonces aplicarlo es un solo paso que se puede deshacer.

La música: una parte sobre todos los acordes

La ayuda bajo los campos lo dice: «Escribe las partes con altura sobre Do mayor. Siguen los acordes de la canción al aplicarlas». Un músico que aprende una figura de acompañamiento la aprende como una forma, fundamental, tercera, quinta, séptima, y toca la misma forma sobre todos los acordes. Escribir la forma sobre Do mayor le da nombres fijos: Do es la fundamental, Mi la tercera, Sol la quinta, Si la séptima.

Qué toca cada nota escrita

Escrita sobre Do	Toca sobre el acorde
Do	su fundamental
Mi	su tercera, o la nota que tiene en su lugar un acorde suspendido (Fa en Csus4)
Sol	su quinta, disminuida o aumentada si la del acorde lo es
Si	su propia séptima (la sexta de un acorde de sexta); si no la tiene, la séptima de la tonalidad
Re, Fa, La	la nota de la tonalidad uno, tres o cinco grados por encima de la fundamental: la 9ª, la 11ª o la 13ª
Do♯, Mi♭, Fa♯, La♭, Si♭	la misma distancia desde la fundamental: un color que se conserva

El arpegio escrito Do–Mi–Sol–Si, en una canción en Do mayor:

Acorde	Toca
Dm7	Re Fa La Do
G7	Sol Si Re Fa
Cmaj7	Do Mi Sol Si
Am	La Do Mi Sol
F	Fa La Do Mi
Csus4	Do Fa Sol Si
Am6	La Do Mi Fa♯

Sobre Am y F, que no tienen séptima, el Si toma la séptima propia de la tonalidad sobre la fundamental: Am suena como Am7 y F como Fmaj7, como los colorearía un pianista que acompaña en Do mayor. Si quieres tríadas simples, no escribas el Si.

Una escala se vuelve el modo de cada acorde

Escribe la escala entera, Do Re Mi Fa Sol La Si, y la banda la toca desde la fundamental de cada acorde con las notas de la tonalidad:

Acorde	Toca	Que es
Dm7	Re Mi Fa Sol La Si Do	Re dórico
G7	Sol La Si Do Re Mi Fa	Sol mixolidio
Am	La Si Do Re Mi Fa Sol	La eólico
Bdim	Si Do Re Mi Fa Sol La	Si locrio
F	Fa Sol La Si Do Re Mi	Fa lidio

Son los modos de Do mayor (ver Escalas, modos y tonalidades): las notas de la tonalidad, empezadas desde cada acorde. Así, el Re que escribes sobre Do, una novena, es el Mi de la tonalidad sobre Dm7 pero Fa sobre Em, y el La, una sexta, es Do sobre Em, no Do♯: la línea nunca sale de la tonalidad mientras los acordes estén en ella.

Acordes de fuera de la tonalidad

Un acorde con una nota de fuera de la tonalidad, como un Fm prestado o el E7 que lleva a Am, no puede tomar las notas de la tonalidad para su novena o su sexta. Conserva las distancias escritas sobre Do, las notas de su propia escala mayor, salvo que un acorde menor toma la séptima menor:

Acorde	Do Re Mi Fa Sol La Si toca	Do–Mi–Sol–Si toca
Fm	Fa Sol La♭ Si♭ Do Re Mi♭	Fa La♭ Do Mi♭
E7	Mi Fa♯ Sol♯ La Si Do♯ Re	Mi Sol♯ Si Re
B♭	Si♭ Do Re Mi♭ Fa Sol La	Si♭ Re Fa La

Es más simple que las escalas de acorde del solista, que al E7 que lleva a Am le darían el Fa y el Do de La menor. Un estilo es un patrón que debe sonar igual sobre todos los acordes; el solista elige las notas una a una.

Las notas cromáticas conservan su distancia desde la fundamental. Un Mi♭ escrito, la tercera menor sobre Do, toca Si♭ sobre G7, la tercera «blue» contra el Si del acorde; un Si♭ escrito toca Do sobre Dm7 y Mi♭ sobre F.

El bajo, el registro y las notas largas

El bajo sigue las mismas reglas, con una más: sus Do tocan la nota del bajo de un acorde con barra. Una línea escrita Do–Sol–Do–Mi (fundamental, quinta, octava, tercera) toca Re–La–Re–Fa sobre Dm7, y Mi–Sol–Mi–Mi sobre C/E: los Do se vuelven Mi, el Sol sigue siendo la quinta del acorde y el Mi su tercera.

Una parte se mueve como mucho un tritono desde donde la escribiste. Sobre un acorde de Do a Fa♯ sube; sobre Sol a Si baja, así que una figura escrita alrededor del Do central se queda alrededor de él: sobre G7 el arpegio es Sol3 Si3 Re4 Fa4, debajo del Do central, no una quinta por encima.

Una nota que se sostiene a través de un cambio de acorde se corta ahí y sigue en el acorde nuevo. La batería se copia tal como está escrita. Un estilo de dos compases o más cuenta sus compases desde el principio de la canción: uno de dos compases toca su primer compás en los compases 1, 3, 5 y así sucesivamente, y en un compás más corto que los del patrón se dejan fuera las notas que no caben.

Quién toca qué

Un estilo que solo tiene batería, como la copia de un ritmo, cambia solo la batería: el piano, el bajo y los demás de la banda siguen como estaban. En cuanto un estilo tiene un carril con altura, se encarga de la armonía de la banda: el piano, el bajo, la guitarra, las cuerdas y el órgano tocan solo los carriles que escribiste, y el que no tiene carril se calla. El solista y los vientos tocan sus propias partes salvo que el estilo tenga un carril para ellos. Tus partes también se detienen en los golpes, sostenidos, silencios y anticipaciones de la hoja, y suenan más fuerte en la variación b.

Tus estilos son archivos

Guardar estilo escribe en el archivo del que vino el estilo. **Guardar como…** pide uno nuevo. Los dos escriben archivos `.lineup-style`: si escribes otra extensión, se sustituye, así que `mi groove.txt` se convierte en `mi groove.lineup-style`.

El archivo es el estilo mismo, escrito en RON (Rusty Object Notation, el formato de los archivos de proyecto) por `serde`:

```
pub struct StyleLane {
    pub player: Player,
    pub channel: Channel,
    pub notes: Vec<NoteEvent>,
}
```

```
pub struct UserStyle {
    pub format: u32,
    pub name: String,
    pub about: String,
    pub time: TimeSignature,
    pub tempo: Tempo,
    pub swing: u8,
    pub mode: ScaleKind,
    pub chart: String,
    pub bars: usize,
    pub lanes: Vec<StyleLane>,
}
```

Cada carril lleva su `Channel` entero: el instrumento, su programa General MIDI, su volumen y sus demás ajustes. Los nombres se escriben con letras aunque la app muestre solfeo, así que un estilo guardado en un idioma se abre en el otro.

Los estilos viven en una **carpeta de estilos**, por defecto una carpeta `styles` dentro de la carpeta de datos de la app. **Edición > Ajustes… > Biblioteca > Tus estilos > Carpeta de estilos…** elige otra. La Biblioteca lista solo esa carpeta: un estilo guardado en otro sitio no aparece, y el aviso tras guardar lo dice: «Estilo guardado: … La Biblioteca muestra los estilos de …; Ajustes > Carpeta de estilos la cambia».

La Biblioteca lee cada archivo `.lineup-style` de la carpeta al arrancar, y otra vez después de guardar o de cambiar de carpeta, los ordena por nombre y avisa de los que no puede leer: «Algunos estilos no se pudieron cargar: …». Se suman a los preajustes, junto a los estilos y ritmos integrados. No tienen género, así que **Agrupar por género** los pone bajo **Estilos de usuario (N)**. El filtro **Con acordes** deja los que tienen progresión, y el menú de instrumentos lista los sonidos de sus carriles. **Empezar desde…** en el editor muestra la misma lista y la misma búsqueda, con **Estilo vacío** primero.

Aplicar uno de tus estilos funciona como con cualquier preajuste (ver Qué partes tomar): **Batería** trae sus carriles de batería, **Partes de acordes y bajo** sus otros carriles, **Voces / instrumentos** sus sonidos, y **Compás**, **Tempo**, **Swing** y **Progresión de acordes** lo suyo. Deja la progresión sin marcar para conservar los acordes de la canción. Solo las partes de acordes y bajo necesitan una banda que las toque, así que en una canción sin banda, un estilo tuyo necesita esas partes marcadas para que suenen sus carriles; su tempo, su compás, su swing o sus acordes solos cambian la canción tal como está.

El estilo viaja con la canción

Cuando se aplican las **Partes de acordes y bajo**, el `UserStyle` entero se guarda en la banda, y cuando se aplica la **Batería**, en su `drums`, como `BandDrums::Saved`:

```
pub custom_rhythm: Option<Box<crate::library::UserStyle>>,
pub custom_instruments: Option<Box<crate::library::UserStyle>>,
```

El primero guarda los patrones; el segundo, los instrumentos y sus ajustes. El proyecto no necesita el archivo `.lineup-style`: bórralo, muévelo o abre la canción en otro ordenador, y la banda sigue tocando. Un proyecto que lleva un estilo tuyo necesita al menos el formato 22 (ver el formato del archivo de proyecto). Por eso un estilo puede desaparecer de la Biblioteca sin romper una canción que lo usa: la canción guarda su propia copia.

Cómo funciona

Un proyecto de un patrón, propio

El editor no tiene un segundo rack ni un segundo piano roll. Guarda un `Project` desechable de un solo patrón, construido por `UserStyle::pattern_project`, junto al estilo y al músico de cada fila:

```
struct Draft {
    style: UserStyle,
    project: Project,
    roles: Vec<Player>,
}
```

Luego llama a los mismos widgets que las ventanas de la canción (ver Dibujar devuelve acciones). El rack devuelve `RackAction`s, el piano roll devuelve `RollAction`s, y el editor los aplica a `draft.project` en vez de a la canción. Añadir un tambor, mover una nota o abrir el piano roll pasan por el código que ya lo hace. La única acción que rechaza es un patrón nuevo: «Los estilos contienen un solo patrón. Guarda los patrones nuevos desde el piano roll de la canción».

Su deshacer es el propio `History<T>` de la app (ver `History<T>`), con el borrador entero como `T`:

```
history: History<Draft>,
```

De las filas de vuelta al estilo

Después de cada fotograma, `sync` copia el borrador de vuelta al estilo: métrica, tempo y swing, y luego un `StyleLane` por canal con el músico y las notas de ese canal, cortadas al final del patrón:

```
draft.style.time = draft.project.time_signature;
draft.style.tempo = draft.project.tempo;
draft.style.swing = draft.project.swing;
```

`UserStyle::validate` comprueba el resultado antes de guardarlo o aplicarlo, una regla cada vez, cada una con un mensaje para quien usa la app («El patrón debe entrar en 1–16 compases y 256 pasos», «Asigna un solo carril a {}»):

```
if !(1..=16).contains(&self.bars) || self.length() > MAX_STEPS {
    return Err(tr("The pattern must fit in 1–16 bars and 256
steps").to_string());
}
```

```
if lane.player != Player::Drums && !players.insert(lane.player) {
    return Err(trf(
        "Assign only one lane to {}",
        &[&tr(lane.player.name())],
    ));
}
```

Las demás comprueban el formato, el nombre, la progresión (cada compás debe leerse como acordes), el número de carriles, el swing, los ajustes de cada carril y que cada nota quede dentro del patrón que se repite. Leer un archivo también llama a `validate`, así que un estilo dañado o editado a mano se rechaza con el motivo en vez de tocar cualquier cosa.

Aplicar: primero sobre una copia

`Project::apply_user_style` cambia la canción en varios pasos: la banda, la métrica, el tempo, los acordes. Leer la progresión en los compases de la canción puede fallar a medio camino, en una métrica en la que el cifrado no cabe. Así que trabaja sobre un clon, y solo un resultado terminado reemplaza la canción:

```
style.validate()?;
// The chart is read into the song's bars, which can fail halfway
// in another meter, after the band and the tempo changed: so it
// is applied to a copy, and an error changes nothing.
let mut project = self.clone();
project.apply_valid_user_style(style, parts)?;
*self = project;
Ok(())
```

`?` devuelve el error en el acto, y la canción queda como estaba. Si la canción no tiene banda, se crea una para que toquen los carriles del estilo; los músicos para los que el estilo no tiene carril tocan como la banda suele hacerlo en ese compás.

Seguir los acordes

Cuando la banda escribe sus partes, `UserStyle::play_lane` repite los compases del estilo sobre los de la canción y le pasa cada nota con altura a `adapt_pitch`, con la armonía y la tonalidad de ese punto. Su comentario es la tabla de arriba en palabras:

```
/// The note a template note written over C major plays over `harmony`,
/// in `key`. C, E and G are the chord's root, third and fifth, and B is
/// its seventh, or sixth, if it has one. Over a chord of the key, B, D, F
/// and A take the note of the key as many steps above the root: an A over
/// Em in C major is C, not C#, and a B over G is F. Over a chord from
/// outside the key they keep their distance from the root, as in its own
/// major, but a minor chord takes the flat seventh. The other notes are
/// chromatic colours and keep their distance from the root.
```

Las notas del acorde salen de un `match` sobre la clase de altura de la nota escrita (0 es Do, 4 Mi, 7 Sol, 11 Si) y la séptima propia del acorde, si la tiene:

```
let own_seventh = [10, 11, 9].into_iter().find(|&seventh| has(seventh));
let pc = source.midi() % 12;
let target = match (pc, own_seventh) {
    (0, _) => 0,
    (4, _) => intervals.get(1).copied().unwrap_or(4),
    (7, _) => intervals.get(2).copied().unwrap_or(7),
    (11, Some(seventh)) => seventh,
    (11, None) if has(3) => 10,
    _ => pc,
};
```

`intervals` son las notas del acorde en semitonos sobre su fundamental, así que `intervals.get(1)` es su segunda nota, la tercera o una nota suspendida, e `intervals.get(2)` su quinta. `own_seventh` busca una séptima menor (10), una séptima mayor (11) o una sexta (9), la nota que tiene en ese lugar un acorde de sexta

o una séptima disminuida. Después, Re, Fa, La y un Si sin séptima que tomar se cuentan en grados:

```
let steps = match (pc, own_seventh) {
    (2, _) => 1,
    (5, _) => 3,
    (9, _) => 5,
    (11, None) => 6,
    _ => return Some(note),
};
```

Si la tonalidad tiene siete notas y contiene todas las del acorde, la nota va a la nota de la tonalidad que está esos grados por encima de la fundamental. Si no, se queda donde la puso el primer `match`.

Guardar sin perder el archivo anterior

Guardar escribe un archivo temporal junto al de verdad y lo renombra:

```
let temporary = path.with_extension(format!("{EXTENSION}.tmp"));
```

Un renombrado reemplaza un archivo de un solo paso, así que un cierre inesperado o un disco lleno mientras se escribe deja el estilo anterior entero, nunca medio archivo.

Cómo se prueba

En `library/user_style.rs`, `template_notes_stay_in_the_key` comprueba la tabla de arriba nota por nota: un Si sobre Am es Sol y sobre D7 Do, un La sobre Em es Do, un Si sobre Fm es Mi♭ y sobre B♭ La.
`every_builtin_can_be_copied_edited_and_saved` y `every_band_style_can_be_copied_with_all_its_players` copian cada estilo y cada banda integrados, validan la copia y la vuelven a leer de su texto.
`a_copy_is_named_and_described_in_the_apps_language` comprueba «Bossa nova copy» y «Bossa nova (copia)», y `user_parts_follow_chords_and_survive_a_project_round_trip` aplica una parte de piano de dos notas sobre Dm y G7, comprueba las terceras que toca, guarda el proyecto (formato 22), lo vuelve a abrir y cambia los acordes.

En la app, `style_window_previews_and_saves_the_editable_pattern`, `discarding_a_draft_leaves_nothing_of_it_to_undo` y `style_confirmations_block_draft_shortcuts_and_escape_keeps_edits` manejan la

ventana del editor;
`saved_styles_reload_alphabetically_with_their_instruments_and_patterns`
vuelve a leer una carpeta de estilos;
`edit_style_copies_the_whole_band_of_a_new_song` comprueba que **Editar estilo…**
en una canción nueva trae piano, bajo, batería y guitarra; y
`saving_a_style_elsewhere_keeps_the_styles_folder`, el aviso.

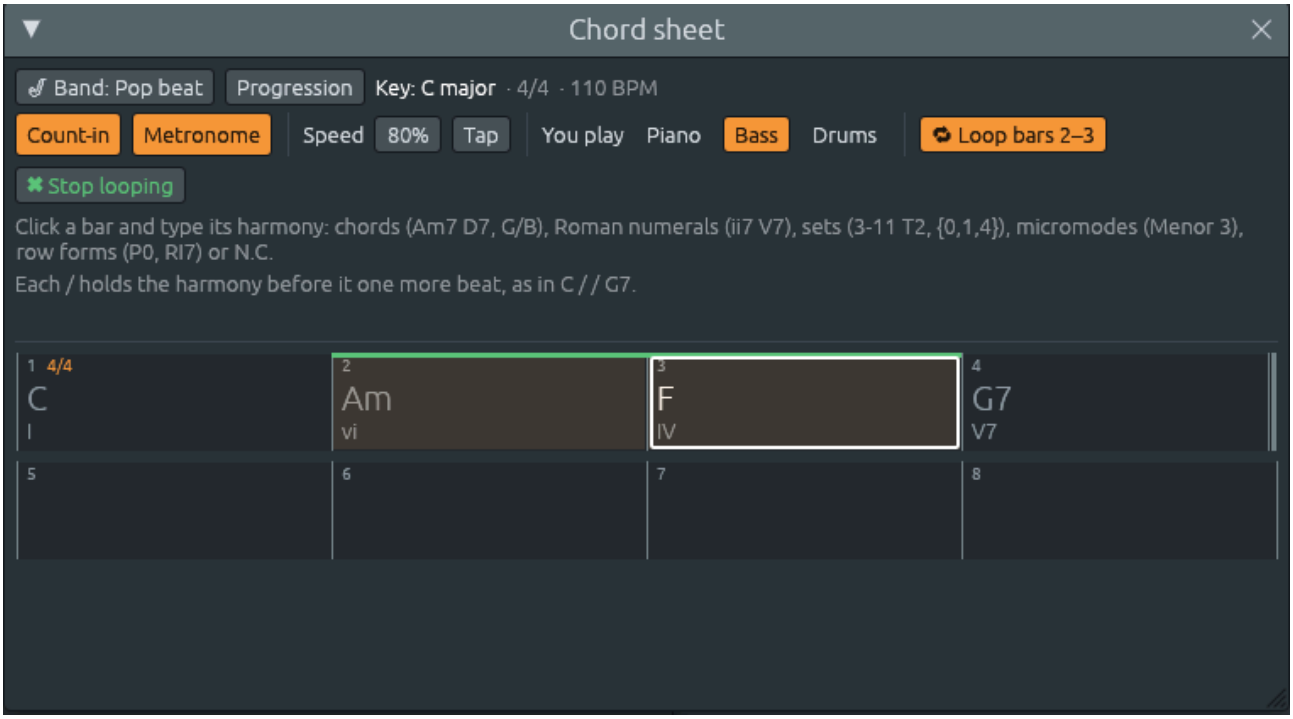
Herramientas de práctica: metrónomo, entrenador de velocidad, entrenamiento auditivo

Archivos: `crates/lineup-audio/src/metronome.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-audio/src/phrase.rs`, `crates/lineup-core/src/meter.rs`, `crates/lineup-core/src/sequencer.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/history.rs`, `crates/lineup-core/src/ear_training.rs`, `crates/lineup-app/src/app/practice.rs`, `crates/lineup-app/src/chord_sheet/practice.rs`, `crates/lineup-app/src/ear_training.rs` y `crates/lineup-app/src/app/ear.rs`

Una banda que toca tu hoja de acordes es más útil cuando tocas con ella. Este capítulo trata de las herramientas que busca un músico que practica con una grabación: una **cuenta de entrada**, un **metrónomo**, una **velocidad** más lenta y un **entrenador de velocidad** que la vuelve a subir, **marcar el tempo** pulsando, quitarle una parte a la banda para tocarla **tú**, y **repetir** los compases en los que estás trabajando. Luego, la otra mitad de practicar: el **entrenamiento auditivo**, aprender a nombrar lo que oyes sin mirar.

Nada de esto forma parte de la canción. Una canción guardada y una exportación WAV conservan el tempo propio de la canción y no llevan clics.

Practicar con la banda



Probarlo

Abre la hoja de acordes (**F5**). Debajo de los botones Banda y Progresión está la fila de práctica:

- **Cuenta de entrada**: cuando pulsas Reproducir, primero suena un compás de clics, y la música empieza en el siguiente «uno». Empieza en medio de un compás y la cuenta sigue hasta donde está el cursor de reproducción: desde el pulso 3 de un compás de 4/4 oyes «1 2 3 4 1 2», y la música entra en el 3, así que el pulso nunca salta.
- **Metrónomo**: un clic en cada pulso mientras suena la música. El primer pulso de cada compás es más agudo (La6, 1760 Hz) que los demás (Mi6, una cuarta más abajo, 1318,5 Hz), para que siempre sepas dónde está el «uno».
- **Velocidad**: a qué velocidad toca la banda, como porcentaje del tempo de la canción, de 25% a 200%. La canción conserva su propio tempo: la caja TEMPO sigue diciendo 110, con «TEMPO 80%» encima. El **Entrenador de velocidad** de al lado sube la velocidad un poco cada vez que el bucle da la vuelta (ver El entrenador de velocidad más abajo).
- **Pulsar**: haz clic en él al pulso unas cuantas veces y el tempo de la canción pasa a ser el tuyo. Las pulsaciones seguidas son un solo paso de deshacer.
- **Tocas tú**: un botón por cada músico del estilo de la banda. La banda deja fuera esa parte y la tocas tú. Para todos los músicos salvo la batería, tu teclado (MIDI, o el teclado del ordenador) cambia al sonido de esa parte.

- **Repetir los compases**: haz clic en un compás, Mayús+clic en otro (o mantén Mayús y pulsa una flecha), y 🔁 **Repetir los compases 2–3** los toca una y otra vez. ✖ **Dejar de repetir** toca toda la canción otra vez desde donde estás.

Cuenta de entrada y Metrónomo también son dos botones con icono al lado de Reproducir en la barra de herramientas, así que funcionan sin la hoja de acordes, sobre cualquier canción o patrón, y **Edición > Ajustes… > Reproducción > Práctica** los tiene junto con la velocidad de práctica y el entrenador de velocidad. La cuenta de entrada, el metrónomo y los números del entrenador se recuerdan para la próxima vez; la velocidad empieza en 100% cada vez que arranca la app, con el entrenador apagado, para que una canción nunca esté lenta por sorpresa.

De dónde salen los clics

Los clics no son un canal de la canción. Si lo fueran, se guardarían con ella, se exportarían a cada archivo WAV, los silenciaría el mezclador y la banda los movería cuando reescribe sus partes. Los hace el propio hilo de audio, que ya sabe cuándo empieza cada paso.

Lo que necesita saber es *qué clic tocar en cada paso*. Eso depende del compás, que puede cambiar en medio de la canción, así que el hilo de la interfaz lo calcula y envía la respuesta como una lista, una entrada por paso:

```
pub enum Click {
    #[default]
    None,
    Beat,
    Bar,
}
```

```
pub fn click(self, step: usize) -> Click {
    if self.is_bar_start(step) {
        Click::Bar
    } else if self.is_beat_start(step) {
        Click::Beat
    } else {
        Click::None
    }
}
```

`Beat` quiere decir el pulso *que se siente*, el que marcas con el pie (ver Pulsos de distinta longitud):

Compás	Clics en un compás	Por qué
4/4	1 2 3 4	cuatro negras
3/4	1 2 3	un vals
6/8	1 2	dos negras con puntillo: seis corcheas sentidas como dos grupos de tres
2/2	1 2	dos blancas, el «compás partido»
2+2+3/8	1 2 3	tres pulsos, el último más largo

Así que un metrónomo en 6/8 no hace seis clics por compás. Una giga, o Noche de paz, se siente en dos, y marcar cada corchea lo escondería.

La lista de la canción sale de `Project::played_clicks`, que cuenta los compases tal como *suena* la canción: una sección en 3/4 que se repite está en 3/4 las dos veces (ver La canción tal como suena). La del patrón seleccionado sale de `Project::pattern_clicks`. Van al hilo de audio como `Command::SetSongClicks` y `Command::SetPatternClicks`. Cuando llega una lista nueva, el hilo de audio la pone en su sitio con `std::mem::replace` y devuelve el `Vec` viejo al hilo de la interfaz como `Garbage::Clicks`, para liberarlo allí (ver Liberar memoria en otro sitio).

El clic en sí

`ClickVoice`, en `metronome.rs`, toca una onda senoidal corta que se apaga:

```
const RING: f32 = 0.03;
```

```
decay: (-6.9 / (RING * sample_rate)).exp(),
```

En cada frame el nivel se multiplica por `decay`. Después de `RING` segundos (30 ms, 1440 frames a 48 kHz) se ha multiplicado por `decay` 1440 veces, que es $e^{-6,9} \approx 0,001$, una milésima de donde empezó: −60 dB, demasiado bajo para oírse. Lo bastante largo para oír una altura, lo bastante corto para terminar mucho antes del pulso siguiente incluso a 300 BPM.

La cuenta de entrada: un segundo reloj

El hilo de audio tiene un `Sequencer`, el reloj que dice cuándo empieza cada paso (ver El secuenciador). También mueve el cursor de reproducción, que la ventana lee para dibujar por dónde va la canción. Durante una cuenta de entrada la canción no ha

empezado, así que el cursor no debe moverse. La cuenta tiene un reloj propio, `counter`, y un pequeño registro de cuánto ha avanzado:

```
/// A count-in under way: the bar it counts (where it starts in the
/// loop's clicks, and how many steps it has), how many steps it lasts in
/// all, and how many have gone by.
#[derive(Debug, Clone, Copy)]
struct CountIn {
    bar: usize,
    length: usize,
    steps: usize,
    counted: usize,
}
```

Reproducir la pone en marcha:

```
let (bar, length) = count_in_bar(self.loop_clicks(), self.start);
let length = length.max(1);
let into_the_bar = self.start.saturating_sub(bar) % length;
self.counter.set_tempo(self.tempo_of_step(self.start));
self.counter.reset();
self.counting = Some(CountIn {
    bar,
    length,
    steps: length + into_the_bar,
    counted: 0,
});
```

`count_in_bar` busca hacia atrás desde el cursor el último `Click::Bar`, y hacia delante el siguiente, para encontrar el compás en el que está el cursor. La cuenta dura ese compás entero *más* los pasos desde su comienzo hasta el cursor: desde el paso 6 de un compás de 16 pasos, 16 + 6 = 22 pasos, «1 2 3 4 1 2», y la música empieza en el paso 6, medio pulso dentro del pulso 2. Cuenta al tempo en que empezará la música, así que una canción que empieza lenta se cuenta lenta.

Después, en cada frame:

```
if let Some(count) = self.counting {
    self.count_in_frame(count);
}
```

```
fn count_in_frame(&mut self, mut count: CountIn) {
    if self.counter.tick().is_none() {
        return;
    }
    if count.counted == count.steps {
        self.counting = None;
        self.sequencer.start_at(self.start);
        self.playing = true;
        self.trace_play(self.start);
        return;
    }
    let step = count.bar + count.counted % count.length;
    self.click.strike(click_at(self.loop_clicks(), step));
    count.counted += 1;
    self.counting = Some(count);
}
```

Cuando la cuenta termina, el secuenciador de la canción arranca *en ese mismo frame*: el bucle de frames atiende la cuenta antes de pedirle un paso al secuenciador de la canción, así que la primera nota cae justo un compás después del primer clic. La prueba `the_count_in_comes_before_the_music` lo comprueba al frame: a 120 BPM un paso son 6.000 frames a 48 kHz, los clics llegan en los frames 0, 24.000, 48.000 y 72.000 (cada uno se oye un frame después, porque una senoide empieza en cero), el cursor espera, y la primera nota suena en el 96.000.

`CountIn` deriva `Copy`, así que `if let Some(count) = self.counting` hace una copia, y `count_in_frame` recibe la copia. Si tomara una referencia dentro de `self.counting`, ese préstamo de `self` seguiría vivo mientras la función llama a `self.counter.tick()` y a `self.click.strike(…)`, que necesitan `&mut self`: el borrow checker lo rechazaría. Trabajar sobre una copia y escribirla de vuelta al final (`self.counting = Some(count)`) es la salida habitual para un estado pequeño como este.

Detener, durante la cuenta, la termina. Mover el cursor o cambiar entre modo patrón y modo canción durante la cuenta la vuelve a empezar, contando el compás en el que empezará ahora la reproducción. Las teclas que tocas durante la cuenta no se graban: la canción no tiene posición hasta que empieza la música, y la grabación solo guarda las notas que la tienen.

La velocidad

La velocidad de práctica nunca toca el proyecto. `Project::tempo` y las marcas de tempo se quedan como están; solo cambia lo que se le dice al hilo de audio:

```
pub(super) fn at_speed(tempo: Tempo, speed: f32) -> Tempo {
    if speed == 1.0 {
        tempo
    } else {
        Tempo::new(tempo.bpm() * speed)
    }
}
```

Todo comando que lleva un tempo pasa por ahí: `SetTempo` después de una edición, y el tempo de cada paso de la canción, así que una canción que se frena al final se sigue frenando, desde el 80% de su tempo. Una exportación WAV construye sus comandos a velocidad `1.0`, y por eso nunca sale más lenta.

Marcar el tempo

Marcar el tempo es aritmética con los momentos de las pulsaciones. Pulsaciones a los 10,0, 10,5, 11,0 y 11,5 segundos son tres pulsos en 1,5 segundos:

> BPM = 60 × pulsos ÷ segundos = 60 × 3 ÷ 1,5 = 120

Usar la primera y la última pulsación de la serie, en vez de solo las dos últimas, compensa la irregularidad de una mano humana. Solo cuentan las últimas ocho, así que se sigue un tempo que se asienta mientras pulsas, y una pausa de más de dos segundos empieza una serie nueva. El resultado se redondea a un BPM entero.

Pulsas los pulsos que marca el metrónomo, y no siempre son negras, mientras que un tempo en BPM cuenta negras. Así que la app calcula primero cuántas negras dura un pulso:

```
let time = self.project.time_signature;
let beats = time.beat_starts().len().max(1);
let quarters = time.steps_per_bar() as f64 / beats as f64 /
f64::from(STEPS_PER_BEAT);
```

```
let bpm = (60.0 * quarters * beats as f64 / (last - first)).round() as
f32;
```

En 6/8 un compás tiene 12 pasos y dos pulsos, así que un pulso son 6 pasos, una negra con puntillo, 1,5 negras. Pulsar 80 pulsos por minuto fija 120 BPM. En 2/2 un pulso es una blanca, dos negras. En un 7/8 simple cada corchea es un pulso; en 2+2+3/8 los tres pulsos son desiguales, así que un pulso marcado cuenta como su promedio, 7/6 de negra.

Cada pulsación es un clic, y cada clic empieza normalmente un paso de deshacer nuevo (ver Varios clics, un paso). Deshacer un tempo pulsación a pulsación sería pesado, así que `History` tiene un método para eso:

```
pub fn reopen(&mut self, label: &'static str) -> bool {
    if self.redo.is_empty() && self.undo_label() == Some(label) {
        self.open = true;
    }
    self.open
}
```

A partir de la tercera pulsación de una serie, la app vuelve a abrir el paso «marcar el tempo» antes de cambiar el tempo, así que toda la serie es un paso. Solo lo hace cuando el último paso *es* «marcar el tempo» y no se deshizo nada desde entonces: nunca une una pulsación a otro cambio.

Tocas tú

Quitarle una parte a la banda silencia la pista de ese músico en la lista de reproducción:

```
pub fn set_you_play(&mut self, player: Player, you: bool) -> bool {
    let Some(track) = self.player_track(player) else {
        return false;
    };
    self.playlist.tracks[track].muted = you;
    true
}
```

Silenciar, y no quitar, importa. La banda sigue escribiendo la parte cuando cambias un acorde, así que cuando vuelves a hacer clic en el botón toca lo que la hoja dice *ahora*. Y como la huella de una parte deja fuera la pista en la que están sus clips (ver Una parte que editas sigue siendo tuya), silenciarla no cuenta como edición: la parte no se congela.

Para todos los músicos salvo la batería, la app también selecciona el canal de ese músico, el que toca tu teclado: te encargas del bajo *con* el sonido del bajo. La batería toca en varios canales (bombo, caja, charles…), así que no hay un solo canal que seleccionar.

Repetir los compases que elijas

La hoja recuerda, además del compás seleccionado, un `anchor`: el otro extremo de una selección hecha con Mayús.

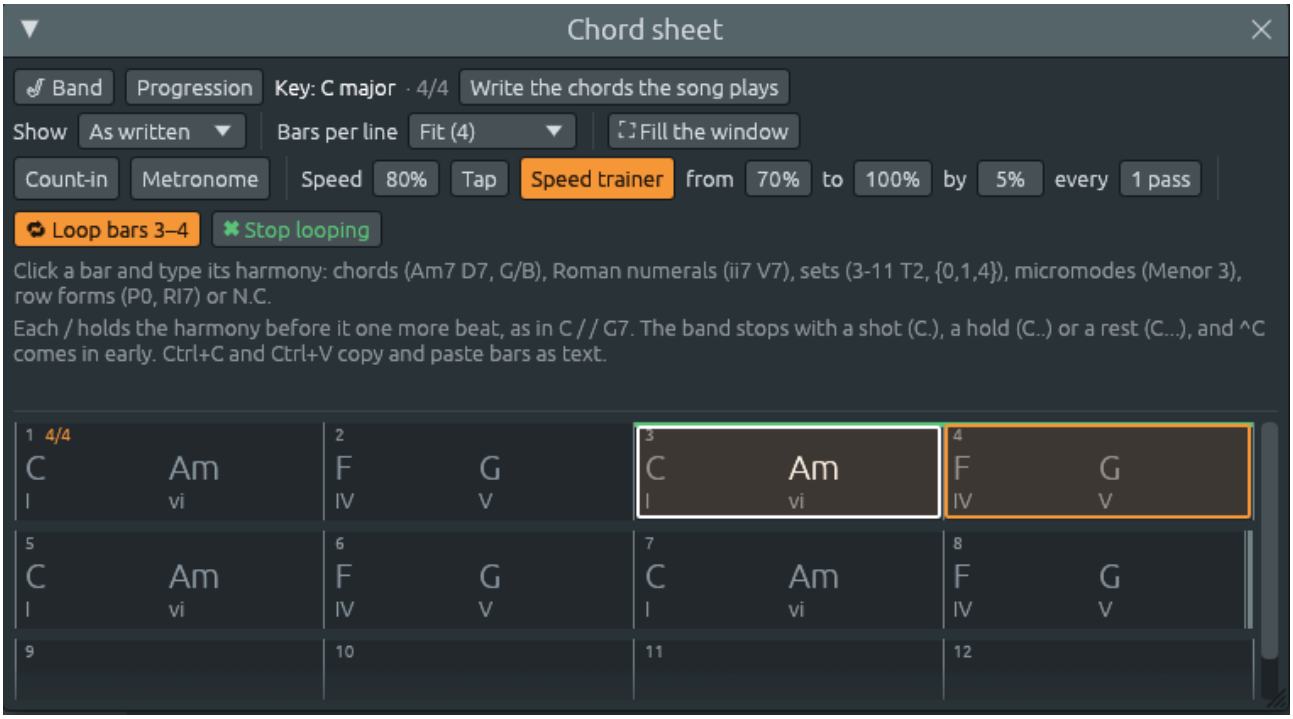
```
fn select_to(&mut self, bar: usize, shift: bool) {
    self.anchor = match (shift, self.anchor, self.selected) {
        (false, ..) => None,
        (true, Some(anchor), _) => Some(anchor),
        (true, None, selected) => selected,
    };
    self.selected = Some(bar);
    self.clipboard = None;
}
```

El primer Mayús+clic guarda como ancla el compás que estaba seleccionado; los siguientes mueven solo el otro extremo, como en un editor de texto. Suprimir borra todos los compases seleccionados, en un paso de deshacer.

Repetir los compases convierte la selección en la región de bucle de la lista de reproducción, que el modo canción ya sabe tocar (ver Secciones, repeticiones y bucles), pasa a modo canción y va al comienzo del bucle. Los compases después del final de la canción no se pueden repetir: no hay nada que tocar ahí, y la ayuda del botón lo dice. En la hoja, los compases que se repiten llevan una línea verde arriba, el color del bucle en la lista de reproducción.

El entrenador de velocidad

Un pasaje que no puedes tocar a velocidad completa se aprende igual en todos los instrumentos: despacio, hasta que sale bien, luego un poco más rápido, y otra vez, hasta llegar al tempo. Con un metrónomo eso obliga a parar cada pocas vueltas para girar una perilla. El **entrenador de velocidad** la gira por ti: cada vez que el bucle da la vuelta, la banda toca unos puntos más rápido, hasta llegar a la velocidad que buscas.



Probarlo

1. Pulsa **Ctrl+N** para una canción nueva con banda, a 110 BPM, y escribe cuatro compases en la hoja de acordes: `| C | Am | F | G |`. Haz clic en el compás 3, Mayús+clic en el compás 4 y haz clic en 🔁 **Repetir los compases 3–4**.
2. Haz clic en **Entrenador de velocidad**. La velocidad baja a **70%**, y aparecen cuatro casillas junto al botón: **desde 70% hasta 100% por 5% cada 1 pasada**.
3. Pulsa Reproducir. La banda empieza al 70% de 110 BPM, 77 BPM. Cada vez que termina el compás 4 y vuelve el compás 3, la velocidad sube un 5% del tempo de la canción: 75%, 80%, 85%… En la séptima vuelta toca al 100%, el tempo propio de la canción, y se queda ahí. La barra de estado dice por dónde va: «Entrenador de velocidad: 80%, camino al 100%».
4. Pulsa **Detener** (■). La velocidad vuelve al 70%, lista para empezar lenta la próxima tanda. **Pausa** (❙❙) conserva la velocidad, así que Reproducir sigue desde donde estabas, a la velocidad a la que habías llegado.

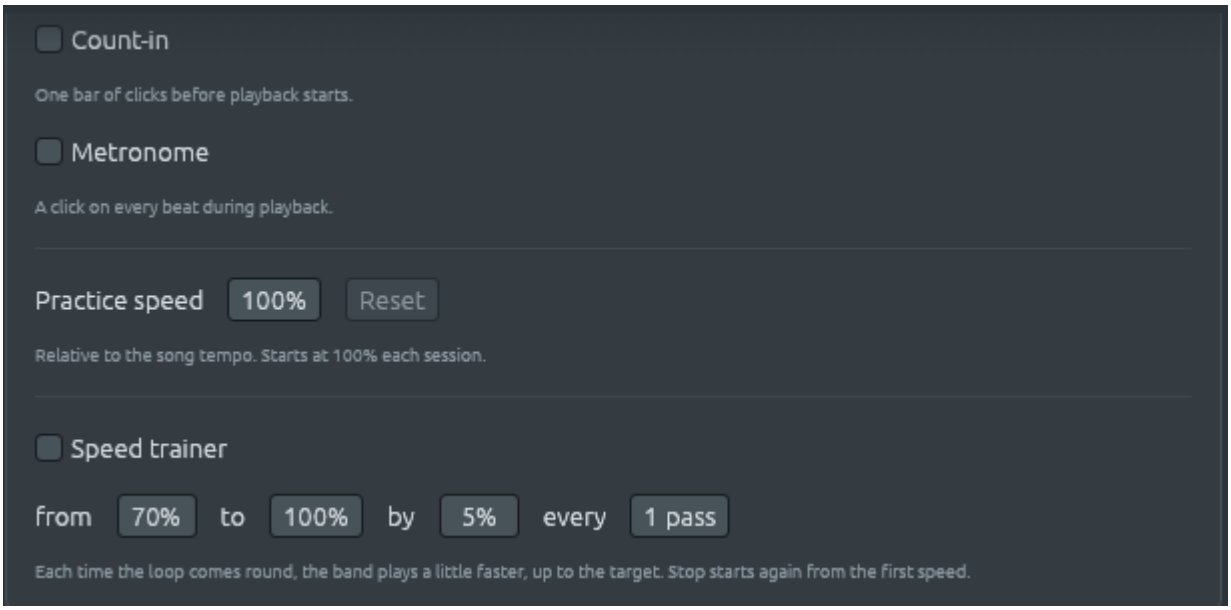
Las cuatro casillas se arrastran, o se escriben con doble clic:

- **desde**: la velocidad a la que empieza Reproducir.
- **hasta**: la velocidad a la que llegar. 100% es el tempo de la canción; 110% lo pasa, para tener margen cuando lo toques de verdad. Un objetivo por debajo del comienzo frena en vez de acelerar.
- **por**: cuánto más rápida es cada velocidad nueva, del 1% al 25% del tempo de la canción. El último paso se detiene en el objetivo, así que 96% por 8% da 100%.

- **cada**: cuántas veces suena el bucle a cada velocidad antes de la siguiente, de 1 a 16. Con **cada 2 pasadas** tocas cada velocidad dos veces: una para conseguirla, otra para comprobar que la tienes.

Una velocidad que pones a mano en la casilla **Velocidad**, durante una tanda, es desde donde sigue el entrenador. Hacer clic otra vez en **Entrenador de velocidad** lo apaga y deja la velocidad donde llegó.

«El bucle» es lo que la reproducción repite: los compases que repites, la canción entera cuando no se repite nada (vuelve a empezar desde el compás 1 al terminar), o el patrón, en modo patrón. El entrenador funciona con los tres. Los mismos controles están en **Edición > Ajustes… > Reproducción**, debajo de la velocidad de práctica:



Contar las vueltas

Para ir más rápido «cada vez que el bucle da la vuelta», la app tiene que saber cuándo pasa. La ventana podría mirar el cursor y contar una vuelta cada vez que el número de paso baja, del último paso al primero. Pero el número de paso también baja cuando haces clic en un compás anterior mientras suena la música, y eso no es una vuelta: el entrenador aceleraría cada vez que saltas atrás para volver a probar un compás.

Quien lo sabe es el `Sequencer` del hilo de audio, que cuenta sus **vueltas** (`laps`): las veces que siguió tocando más allá del último paso y volvió al paso 0 (ver El tick). Dentro de `tick`:

```
if self.coming_round {
    self.laps += 1;
}
self.next_step = (self.next_step + 1) % self.length;
self.coming_round = self.next_step == 0;
```

`coming_round` se activa cuando suena el último paso, y la vuelta se cuenta cuando suena el paso 0 después. Un salto (`start_at`, que envía un clic en un compás), un reinicio (`reset`) y un bucle acortado por debajo de donde iba la reproducción (`set_length`) desactivan `coming_round`, así que ninguno cuenta.

El hilo de audio guarda la cuenta después de cada paso en un entero atómico, como hace con el cursor, y `AudioEngine::laps()` la lee. La ventana la mira una vez por fotograma:

```
fn came_round(&mut self, laps: usize) -> Option<u16> {
    let seen = self.laps_seen.replace(laps);
    if !self.training {
        return None;
    }
    // A new audio thread, after a restart, counts from 0 again.
    let new = laps.checked_sub(seen?)?;
    if new == 0 {
        return None;
    }
    // ...
}
```

Comparar con la cuenta que vio la última vez, en vez de buscar «hubo una vuelta», hace que nunca se pierda una: si la ventana estuvo oculta unos segundos y no se dibujó, encuentra varias vueltas nuevas a la vez y avanza la velocidad una vez por cada una.

En una función que devuelve un `Option`, `valor?` quiere decir «el valor de dentro, o devuelve `None` desde aquí». `seen?` devuelve `None` la primera vez, cuando todavía no se vio nada. `checked_sub` es una resta que da `None` en vez de bajar de cero: después de **Reiniciar audio** un hilo de audio nuevo vuelve a contar desde 0, `laps` es menor que `seen`, y el entrenador espera a la siguiente vuelta en vez de calcular un número enorme de ellas. Un simple `laps - seen` entraría en pánico ahí en una compilación de depuración, y daría la vuelta hasta un número cercano a 2^{64} en una de release.

En el pulso, no un paso tarde

La ventana se dibuja unas 60 veces por segundo, así que se entera de una vuelta hasta un fotograma después, y el hilo de audio necesita luego un bloque más para recibir un comando. Para entonces el paso 0 de la vuelta nueva ya empezó, al tempo viejo, y la velocidad nueva empezaría en el paso 1: la primera semicorchea de cada pasada un poco lenta. Nadie notaría una semicorchea un 5% más larga, pero el clic del primer pulso llegaría unos milisegundos tarde, en cada vuelta.

Así que la ventana no espera a la vuelta. En cuanto sabe la velocidad siguiente (justo después de la última vuelta, o cuando enciendes el entrenador), le envía al hilo de audio los tempos de esa velocidad, para usarlos **a partir de la vuelta siguiente**:

```
pub struct LapTempos {
    /// In place of the [`Command::SetTempo`] tempo.
    pub tempo: Tempo,
    /// In place of the [`Command::SetSongTempos`] list.
    pub song: Vec<Tempo>,
}
```

```
SetLapTempos(Option<LapTempos>),
```

El hilo de audio los guarda hasta que el bucle da la vuelta. Justo antes de que suene un paso ya pregunta a qué tempo va ese paso; antes, pregunta si es el paso 0 que vuelve, y si lo es, toma los tempos que esperan:

```
if let Some(step) = self.sequencer.due() {
    if self.sequencer.comes_round()
        && let Some(tempos) = self.lap_tempos.take()
    {
        self.tempo = tempos.tempo;
        let old = std::mem::replace(&mut self.song_tempos, tempos.song);
        self.throw_away(Garbage::Tempos(old));
    }
    let tempo = self.tempo_of_step(step);
    if tempo != self.tempo_now {
        self.set_tempo(tempo);
    }
}
```

`take()` vacía la ranura, así que los tempos se usan una sola vez. La lista vieja vuelve al hilo de la interfaz para liberarse, como toda lista que suelta el hilo de audio.

Cuando la ventana ve después la vuelta, envía otra vez los mismos tempos con los habituales `SetTempo` y `SetSongTempos`. Si llegaron a tiempo, eso no cambia nada: son exactamente los mismos números. Importa cuando no llegaron: si el pedido alcanzó

al hilo de audio después de que el bucle ya diera la vuelta, el reenvío pone la velocidad nueva un paso tarde, y el pedido siguiente reemplaza al viejo antes de que pueda sonar una vuelta tarde.

Mantener el pedido al día

Un pedido espera una vuelta entera, y la canción puede cambiar mientras tanto: escribes un acorde, y la parte nueva de la banda va al hilo de audio con la lista de tempos de la canción. Una lista de tempos calculada antes podría tener la longitud equivocada, o faltarle un cambio de tempo que acabas de hacer. Así que cada vez que la ventana envía `SetTempo` o `SetSongTempos`, lo cuenta (`tempos_sent`), y guarda, junto al cambio de velocidad que pidió, la cuenta de ese momento. Una vez por fotograma los compara con lo que pediría ahora:

```
let change = self.practice.next_change();
let sent = change.map_or(0, |_| self.practice.tempos_sent);
if self.practice.lap_change != Some((change, sent)) {
    let tempos = change.map(|(_, next)| {
        let (tempo, song) = self.tempos_at(next);
        LapTempos { tempo, song }
    });
    self.send(Command::SetLapTempos(tempos));
    self.practice.lap_change = Some((change, sent));
}
```

`next_change` es `None` mientras el entrenador está apagado, en su objetivo, o antes de la última pasada a una velocidad cuando toca cada velocidad más de una vez; entonces la ventana envía `SetLapTempos(None)`, que retira los tempos que aún esperan. Detener, una velocidad escrita a mano o números nuevos en el entrenador cambian `next_change`, así que el pedido los sigue también.

Cuánto dura una tanda

Dos compases de 4/4 son 8 pulsos. Con los valores por defecto sobre los compases 3–4 de la canción de arriba, a 110 BPM:

Vuelta	Velocidad	BPM	Empieza a los
1	70%	77	0 s
2	75%	82,5	6,23 s
3	80%	88	12,05 s
4	85%	93,5	17,51 s

Vuelta	Velocidad	BPM	Empieza a los
5	90%	99	22,64 s
6	95%	104,5	27,49 s
7	100%	110	32,08 s

Ocho pulsos a 77 BPM duran 8 × 60 ÷ 77 = 6,23 segundos; a 82,5 BPM los ocho siguientes duran 5,82, y 6,23 + 5,82 = 12,05. Cada cambio llega en el primer paso del bucle, que el registro de reproducción muestra como una línea `# tempo:` en el paso 0.

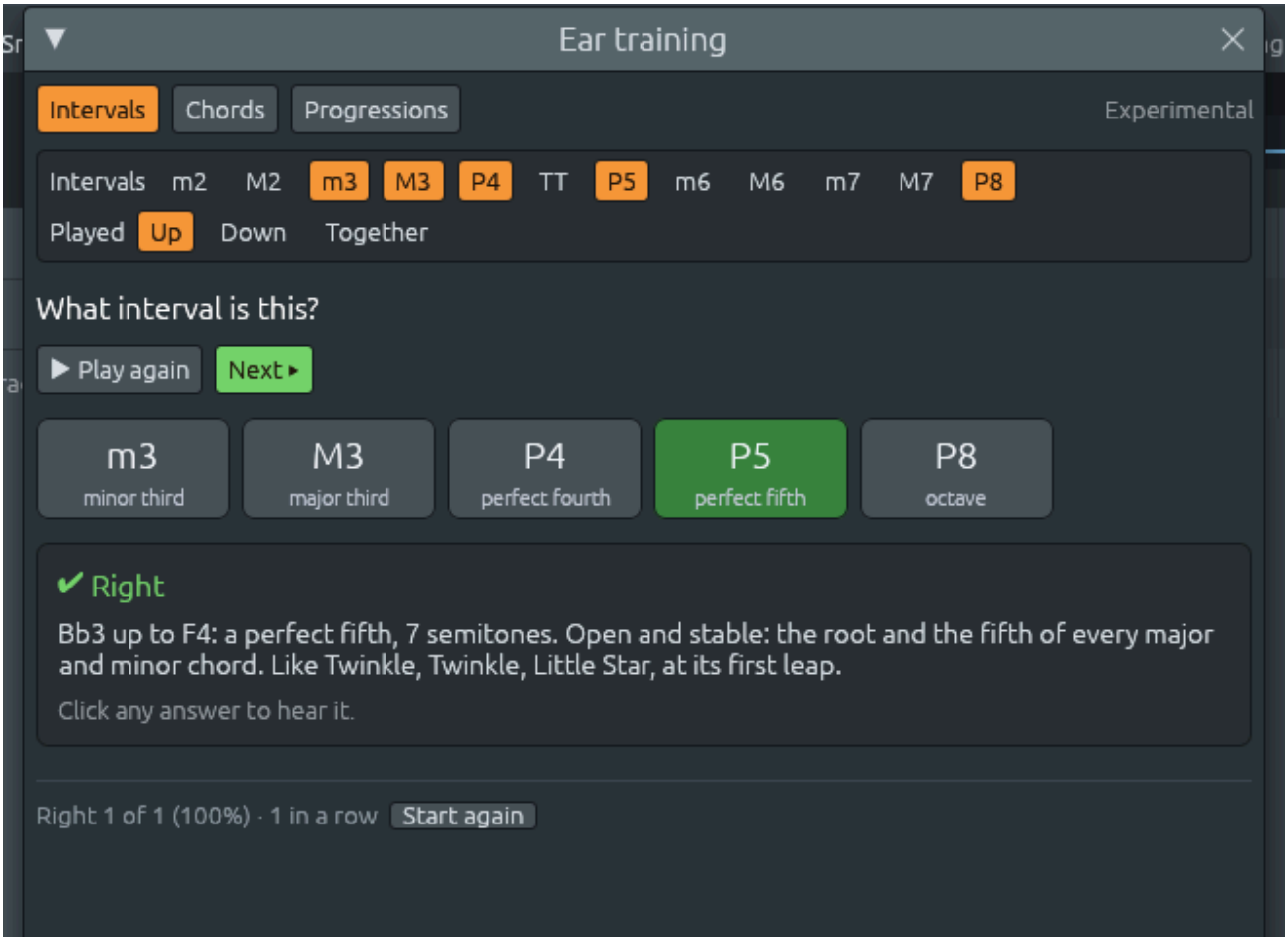
Entrenamiento auditivo

Todo lo demás en la app te ayuda a *ver* la armonía: nombres de acordes en la hoja, números romanos y colores en las teclas de armonía, nombres de conjuntos en la ventana dodecafónica. Los músicos también necesitan *oírla*: distinguir una tercera menor de una mayor, un acorde disminuido de uno menor, y el V de una canción de su IV, sin mirar. Es una habilidad propia, y se aprende como la enseñan los juegos de entrenamiento auditivo: algo suena, lo nombras, y te dicen al momento si acertaste.

Es **experimental** y no estorba: vive en su propia ventana, que está cerrada hasta que la abres, y no cambia nada de la canción ni del archivo.

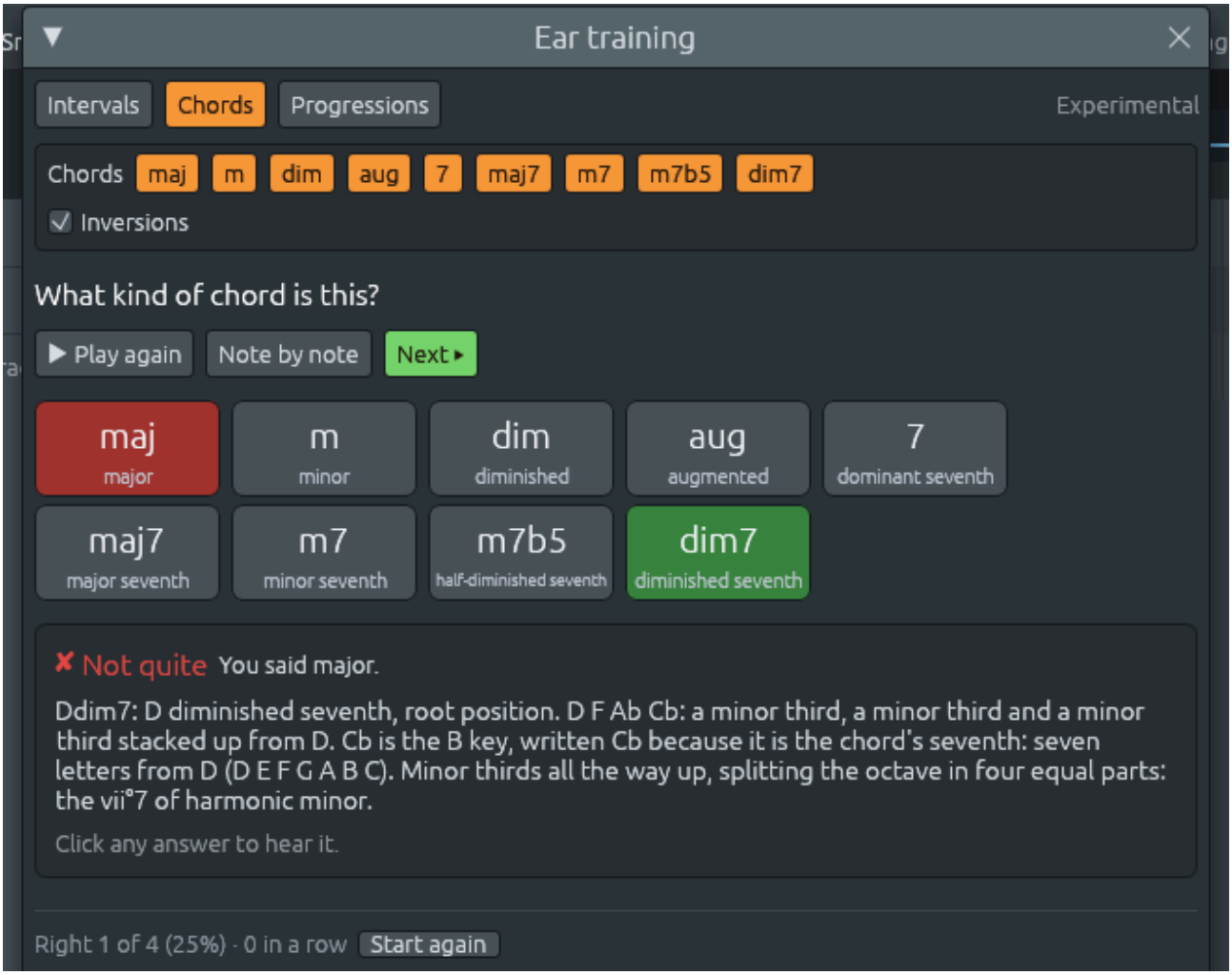
Probarlo

Ver > Entrenamiento auditivo abre la ventana. Arriba hay tres pestañas: **Intervalos**, **Acordes** y **Progresiones**. Debajo, un recuadro dice qué preguntar; luego **▶ Empezar** hace la primera pregunta y la toca. Si la canción estaba sonando, se pausa, para que la pregunta se oiga sola.

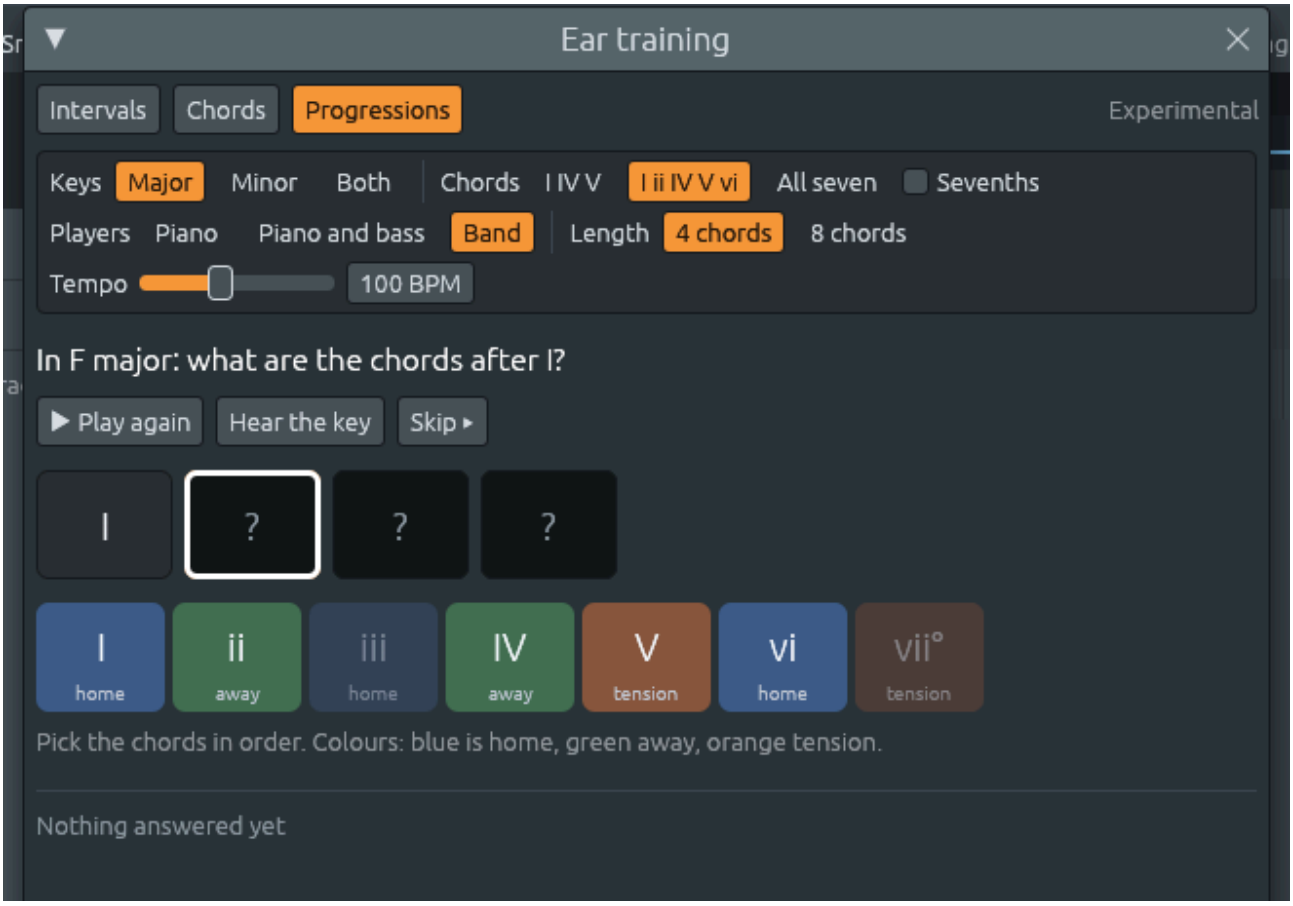


Intervalos. Suenan dos notas, y los botones grandes son las respuestas. El recuadro de arriba elige qué intervalos se pueden preguntar (para empezar, una tercera menor, una tercera mayor, una cuarta, una quinta y la octava) y cómo se tocan: **Arriba** (primero la nota grave), **Abajo** (primero la aguda) o **Juntas**. Menos opciones lo hacen más fácil; empieza con unas pocas que suenen muy distintas, y añade las demás de una en una. En la interfaz en español los botones se llaman 3m, 3M, 4J, 5J, 8J y así: el número y la clase, M mayor, m menor, J justa.

Después de responder, tu botón se pone verde o rojo, el correcto se pone verde, y un recuadro explica la respuesta: las dos notas con sus octavas, el nombre completo del intervalo, cuántos semitonos abarca, qué le da su sonido y, hacia arriba, una melodía que empieza con él. Luego cada botón de respuesta toca *su propio* intervalo desde la misma nota, para que puedas oír el que dijiste al lado del correcto. **▶ Volver a tocar** repite la pregunta todas las veces que quieras; **Siguiente ▸** hace otra (antes de responder es **Saltar ▸**, y una pregunta saltada no cuenta).



Acordes. Suena un acorde, y nombras su clase: mayor, menor, disminuido, aumentado, o uno de cinco acordes de séptima. La fundamental cambia de una pregunta a otra; solo cuenta la clase. **Nota por nota** toca el acorde una nota cada vez desde el bajo hacia arriba, cada una sostenida, así que termina como el acorde entero. Con **Inversiones** marcado, la nota más grave no siempre es la fundamental.



Progresiones. La banda toca unos acordes en una tonalidad, un compás cada uno, y los nombras por sus números romanos. La pregunta dice la tonalidad («En Fa mayor: ¿cuáles son los acordes después de I?») y el primer acorde es siempre I, la casa, para que sepas desde dónde empiezas. Las respuestas son las teclas de armonía, con sus colores y su función (casa, fuera, tensión) pero con los nombres de los acordes ocultos. Las teclas de los acordes que no se pueden preguntar aparecen apagadas. Mientras toca la banda, un marco blanco recorre las casillas, como en las teclas de armonía, para que sepas qué compás estás oyendo.

Haz clic en las teclas en orden, una por cada acorde después de I. Luego cada casilla se pone verde o roja, una roja dice cuál era el acorde («era IV»), y el recuadro escribe los acordes, sus números y por qué cada uno sigue al anterior, con las palabras de las ayudas de las teclas de armonía:



Después de responder, cada casilla y cada tecla tocan su acorde en el piano, y **Escuchar la tonalidad** toca I, IV, V y I: una cadencia que dice dónde está la casa, lo que ayuda cuando te pierdes.

El recuadro de arriba elige las **Tonalidades** (Mayor, Menor o Ambas), los **Acordes** que pueden salir (I IV V; I ii IV V vi; o Los siete), si son **Séptimas** (V7, ii7, Imaj7…), los **Músicos** (Piano, Piano y bajo, o la Banda con batería), la **Duración** (4 o 8 acordes) y el **Tempo** (de 60 a 160 BPM).

La puntuación. La línea de abajo cuenta los aciertos, el porcentaje y cuántos seguidos. Cada pestaña lleva la suya, y en una progresión cada acorde cuenta como una respuesta. **Empezar de nuevo** la pone a cero. La puntuación se olvida al cerrar la app; las opciones de los recuadros se recuerdan.

La teoría en las respuestas

La app enseña con sus respuestas, así que tienen que ser exactamente correctas. Esto es lo que dicen, y por qué.

Intervalos

El nombre de un intervalo tiene dos partes. El **número** cuenta letras (nombres de nota), incluidos los dos extremos: de Do a Mi son Do, Re, Mi, tres notas, una *tercera*. De Do a Mi♭ también es una tercera: los mismos nombres. La **clase** sale luego de los semitonos: una tercera de 4 semitonos es *mayor*, de 3 *menor*.

Segundas, terceras, sextas y séptimas son mayores o menores. Unísonos, cuartas, quintas y octavas se llaman **justas**: invertidas (la nota grave subida una octava), un intervalo justo da otro justo, mientras que uno mayor se vuelve menor y uno menor mayor. Los números de un intervalo y de su inversión suman nueve, sus semitonos doce: una tercera mayor (4) invertida es una sexta menor (8), una cuarta justa (5) una quinta justa (7). Un semitono más que mayor o justo es *aumentado*, uno menos que menor o justo *disminuido*.

Botón	Nombre	Semitonos	Desde Do	Una melodía que empieza con él
2m	segunda menor	1	Do–Re♭	Tiburón, las dos notas del tiburón
2M	segunda mayor	2	Do–Re	Cumpleaños feliz, en su primer paso hacia arriba
3m	tercera menor	3	Do–Mi♭	Greensleeves
3M	tercera mayor	4	Do–Mi	When the Saints Go Marching In
4J	cuarta justa	5	Do–Fa	la Marcha nupcial de Wagner
TT	tritono (cuarta aumentada)	6	Do–Fa♯	Maria, de West Side Story
5J	quinta justa	7	Do–Sol	Estrellita, ¿dónde estás?, en su primer salto
6m	sexta menor	8	Do–La♭	The Entertainer, en su primer salto hacia arriba

Botón	Nombre	Semitonos	Desde Do	Una melodía que empieza con él
6M	sexta mayor	9	Do–La	My Bonnie Lies over the Ocean
7m	séptima menor	10	Do–Si♭	Somewhere, de West Side Story
7M	séptima mayor	11	Do–Si	
8J	octava	12	Do–Do	Over the Rainbow

Las melodías empiezan con el intervalo hacia arriba, así que la app nombra una solo cuando la pregunta fue hacia arriba. La séptima mayor no tiene ninguna: pocas melodías conocidas empiezan con ella.

El **tritono** es el único intervalo con dos números. Seis semitonos por encima de Do está la tecla de Fa♯ o la de Sol♭: de Do a Fa♯ son cuatro nombres (Do Re Mi Fa), una *cuarta aumentada*; de Do a Sol♭ son cinco (Do Re Mi Fa Sol), una *quinta disminuida*. Las mismas teclas, el mismo sonido, dos escrituras. Su nombre dice lo que es: tres tonos enteros, exactamente media octava.

Escritura. La explicación escribe las notas como lo haría un músico. La nota de abajo es una tecla blanca, o una negra escrita con sostenido o con bemol, la que escriba el *par* con menos alteraciones: de Do♯ a Mi es una tercera menor (escribirla de Re♭ a Fa♭ costaría dos bemoles), pero de Re♭ a Fa es una tercera mayor (de Do♯ a Mi♯ costaría dos sostenidos). Cuando las dos son igual de simples, una tecla negra conserva su nombre habitual: de Si♭ a Re♭, no de La♯ a Do♯. El tritono es una cuarta aumentada o una quinta disminuida por la misma regla: de Fa a Si (sin alteración) en vez de Fa a Do♭, y una cuarta aumentada cuando las dos son igual de simples: de Do a Fa♯ en vez de Do a Sol♭, de Si♭ a Mi en vez de Si♭ a Fa♭ o La♯ a Mi.

Acordes

Todo acorde que toca el ejercicio está hecho de terceras, apiladas desde la fundamental (ver Acordes). Las nueve clases, sobre Do:

Botón	Nombre	Símbolo	Notas	Terceras, desde abajo
maj	mayor	C	Do Mi Sol	mayor, menor
m	menor	Cm	Do Mi♭ Sol	menor, mayor

Botón	Nombre	Símbolo	Notas	Terceras, desde abajo
dim	disminuido	Cdim	Do Mi♭ Sol♭	menor, menor
aug	aumentado	Caug	Do Mi Sol♯	mayor, mayor
7	séptima de dominante	C7	Do Mi Sol Si♭	mayor, menor, menor
maj7	séptima mayor	Cmaj7	Do Mi Sol Si	mayor, menor, mayor
m7	séptima menor	Cm7	Do Mi♭ Sol Si♭	menor, mayor, menor
m7b5	séptima semidisminuida	Cm7b5 (Cø7)	Do Mi♭ Sol♭ Si♭	menor, menor, mayor
dim7	séptima disminuida	Cdim7 (C°7)	Do Mi♭ Sol♭ Si♭♭	menor, menor, menor

Dos acordes que a menudo se enseñan pronto quedan **fuera a propósito**: los acordes de sexta y los suspendidos. C6 es Do Mi Sol La, exactamente las mismas teclas que Am7 (La Do Mi Sol); Csus2 (Do Re Sol) tiene las teclas de Gsus4 (Sol Do Re). Tocados solos, nada los distingue: lo que hace que uno sea C6 y el otro Am7 es el bajo y la música de alrededor. Un examen que los preguntara marcaría mal respuestas correctas. Una prueba comprueba que ningún par de las nueve clases comparte notas sobre dos fundamentales cualesquiera.

Algunas escrituras sorprenden al principio: **Si♭♭** en Cdim7, **Do♭** en Fdim. No son errores. La séptima de un acorde de C es alguna clase de Si (Do Re Mi Fa Sol La Si, siete nombres), y una séptima disminuida está un semitono por debajo de la séptima menor Si♭, así que se escribe Si doble bemol, aunque sea la tecla de La. Del mismo modo, la quinta de F es alguna clase de Do, y una quinta disminuida está un semitono por debajo de Do: Do♭, la tecla de Si. La explicación lo dice cada vez que escribe una nota con un nombre que no es el habitual de su tecla, porque el nombre dice qué función cumple la nota en el acorde.

Inversiones. Un acorde está *invertido* cuando la nota más grave no es su fundamental: la tercera (primera inversión), la quinta (segunda inversión) o la séptima (tercera inversión). El símbolo nombra el bajo después de una barra: C/E es Do mayor

con Mi en el bajo (ver Inversiones). Un acorde invertido conserva su clase, así que la respuesta no cambia, pero suena menos asentado y cuesta más nombrarlo; para eso está la casilla **Inversiones**.

Dos clases son especiales. Las notas del acorde aumentado están a la misma distancia (una tercera mayor entre cada una, y una tercera mayor desde la de arriba de vuelta a la fundamental), y lo mismo las de la séptima disminuida (una tercera menor). Pon cualquiera de sus notas en el bajo y obtienes exactamente las mismas teclas que el acorde en estado fundamental sobre *esa* nota: Caug/E es también Eaug (Mi Sol♯ Si♯, donde Si♯ es la tecla de Do). El oído no puede saber qué nota es la fundamental, así que la explicación de un acorde aumentado o de séptima disminuida invertido dice qué otro acorde es también. La clase sigue siendo segura, así que la pregunta sigue siendo justa.

Progresiones

Los números y sus colores vienen de las teclas de armonía (ver Las teclas de armonía): mayúsculas para acordes mayores, minúsculas para menores, ° para disminuidos, y tres funciones, casa (azul), fuera (verde) y tensión (naranja). En Do mayor:

```
I    ii    iii   IV    V     vi    vii°
C    Dm    Em    F     G     Am    Bdim
```

Una tonalidad menor se lee `i ii° bIII iv V bVI bVII`. Los números cuentan a lo largo de la escala *mayor* desde la misma tónica, así que los acordes sobre la tercera, la sexta y la séptima rebajadas llevan un bemol: en La menor, C es bIII. Y V es mayor, como en casi toda la música en tonalidad menor: E mayor en La menor, con el Sol♯ que tira hacia el La. Con **Séptimas** las teclas pasan a ser `Imaj7 ii7 iii7 IVmaj7 V7 vi7 viiø7` en mayor e `i7 iiø7 bIIImaj7 iv7 V7 bVImaj7 bVII7` en menor.

Cómo se hace una progresión. Empieza en I. Cada acorde siguiente suele ser uno de los movimientos que las teclas de armonía sugieren después del acorde anterior (la misma lista detrás de sus tenues pistas amarillas, ver Qué viene después), el movimiento más habitual más a menudo. Una vez de cada cuatro es en cambio cualquier acorde elegido, para que también se oigan los acordes a los que rara vez llevan los movimientos habituales, como iii. Nunca toca el mismo acorde dos veces seguidas, vuelve a I solo al final, y termina en I o en V: un punto final, o una semicadencia, una pregunta que espera su respuesta.

Qué toca la banda. Cada acorde dura un compás de cuatro pulsos. El piano lo toca en los pulsos 1 y 3, cada acorde dispuesto cerca del anterior, como se mueven las manos de un pianista (ver Tocar una tecla). El bajo toca la fundamental en los mismos pulsos, en su octava grave. La batería lleva el tiempo: bombo en 1 y 3, caja en 2 y 4,

charles en cada corchea. El último acorde se sostiene todo el compás sobre un platillo. Con el **Piano** solo, el piano también toca la fundamental debajo del acorde: sin bajo, una disposición cerrada podría oírse como una inversión.

Pistas para escuchar.

- **Sigue el bajo.** Toca las fundamentales, así que su línea *es* la progresión. Una fundamental que baja una quinta (de V a I, de ii a V) suena a llegada.
- **I y V son la pareja más fácil.** V es tensión, I es reposo; V casi pide I.
- **IV y ii llevan los dos fuera y hacia V.** Comparten dos notas (Fa y La en Do mayor: Fa La Do y Re Fa La), así que suenan parecidos; ii es el más oscuro.
- **vi es una casa triste.** Comparte dos notas con I (Do Mi Sol y La Do Mi), y lo mismo iii (Mi Sol Si). Si un acorde suena a casa pero menor, probablemente es vi.
- **Escuchar la tonalidad** cada vez que pierdas de vista dónde está la casa.

Cómo está construido

El trabajo se reparte según las líneas que la app ya tiene. `lineup-core` hace las preguntas y sus explicaciones y no sabe nada de ventanas ni de sonido. `lineup-audio` toca unas notas aparte de la canción. `lineup-app` dibuja la ventana y une las dos cosas.

Una pregunta a partir de un número

Cada pregunta sale de una **semilla** de 64 bits:

```
pub fn new(settings: &IntervalSettings, seed: u64) -> Self {
    let mut random = Random(seed);
    let intervals = settings.intervals.places(Interval::ALL.len());
    let directions = settings.directions.places(Direction::ALL.len());
    let interval =
Interval::ALL[intervals[random.below(intervals.len())]];
    let direction =
Direction::ALL[directions[random.below(directions.len())]];
    let low = Note::new(INTERVAL_LOWEST + random.below(INTERVAL_SPAN) as
u8);
    IntervalQuestion {
        low,
        interval,
        direction,
    }
}
```

`Random` es el pequeño generador SplitMix64 con el que barajan las series dodecafónicas (ver Qué hace interesante a una serie): la misma semilla da siempre los

mismos números, así que la misma pregunta. Eso simplifica las pruebas, porque pueden hacer miles de preguntas y comprobar cada una, y una semilla que falla se puede repetir exactamente.

La ventana toma su primera semilla del reloj, y cada una de las siguientes de `next_seed`:

```
pub fn next_seed(seed: u64) -> u64 {
    Random(seed).next()
}
```

¿Por qué no sumar simplemente la constante del propio generador, `0x9E37_79B9_7F4A_7C15`? Porque eso es exactamente lo que el generador se hace a sí mismo para cada número: la pregunta siguiente empezaría un número más adelante en la serie de la anterior, así que su primer número (la clase de un acorde) sería el segundo de la pregunta anterior (su fundamental). Después de un Do, un Mi o un La♭, el acorde siguiente habría sido siempre mayor. Una semilla tomada de la *salida* del generador empieza números nuevos; la prueba `one_question_does_not_give_the_next_away` hace 2.000 acordes seguidos y comprueba que el acorde después de un Do puede ser de cualquiera de las cuatro clases. Si la pregunta es igual a la anterior, la ventana lo intenta otra vez, hasta ocho veces, así que rara vez sale la misma pregunta dos veces seguidas.

Un conjunto de respuestas en dieciséis bits

Qué intervalos o acordes se preguntan es un conjunto de posiciones en una lista. La lista es corta (doce intervalos como mucho), así que el conjunto es un `u16`, un bit por posición, como un conjunto de clases de altura son doce bits (ver La teoría de conjuntos en doce bits):

```
pub struct Choices(u16);
```

```
pub fn has(self, place: usize) -> bool {
    self.0 & (1 << place) != 0
}

/// Adds the answer at `place`, or takes it away. The last one left
/// stays: there is always something to ask.
pub fn toggle(&mut self, place: usize) {
    let toggled = self.0 ^ (1 << place);
    if toggled != 0 {
        self.0 = toggled;
    }
}
```

`^` (o exclusivo) invierte un bit. Negarse a apagar el último hace que siempre se pueda hacer una pregunta, sin un caso especial en ningún otro sitio.

Cues: qué tocar, en segundos

Una pregunta no se toca a sí misma: devuelve **cues**, valores simples que dicen quién toca qué, cuándo y con qué fuerza.

```
/// One note of a question: who plays it, when, for how long and how loud.
#[derive(Debug, Clone, Copy, PartialEq)]
pub struct Cue {
    pub part: Part,
    /// When it starts, in seconds from the start of the question.
    pub start: f64,
    /// How long it is held, in seconds.
    pub length: f64,
    pub note: Note,
    pub velocity: u8,
}
```

Dejar el sonido fuera de `lineup-core` permite a las pruebas comprobar *qué* se tocaría sin tocarlo: que cada compás de una progresión toca su acorde sobre su fundamental, que el piano solo añade la fundamental, que la nota más grave de una pregunta de acorde es el bajo que nombra su símbolo.

Las respuestas con las que comparar usan la misma idea. `cues_for(interval)` toca otro intervalo desde la misma nota de abajo (o, hacia abajo, desde la misma nota de arriba), y `cues_as(quality)` otra clase de acorde sobre la misma fundamental con la misma nota del acorde en el bajo. Así, cuando haces clic en una respuesta equivocada después de responder, oyes exactamente lo que dijiste al lado de lo que sonó.

Una frase en el hilo de audio

La canción está hecha de patrones y clips sobre una línea de tiempo. Una pregunta no: son unas pocas notas que empiezan *ahora*, en instrumentos propios, mientras los canales de la canción se quedan como están. Así que `lineup-audio` tiene una `Phrase`: sus instrumentos y sus notas, con sus tiempos en frames.

El hilo de audio no debe reservar ni liberar memoria (ver La regla de oro del hilo de audio). Así que el hilo de la interfaz construye la frase entera, instrumentos incluidos, y la envía en un solo comando:

```
PlayPhrase(Box<Phrase>),
```

El `Box` importa. Un `Command` es tan grande como su variante más grande, y la prueba `commands_stay_small` los mantiene en 80 bytes o menos para que la cola siga siendo pequeña. Una `Phrase` tiene dos `Vec` y unos contadores, unos 80 bytes propios; dentro de un `Box`, el comando lleva solo un puntero.

En el hilo de audio, la frase solo empieza cada nota cuando le llega su momento y suma sus instrumentos a la mezcla:

```
pub fn next_sample(&mut self) -> f32 {
    while let Some(note) = self.notes.get(self.next)
        && note.start <= self.clock
    {
        if let Some(instrument) =
self.instruments.get_mut(note.instrument) {
            instrument.note_on(note.pitch, note.velocity, note.gate,
self.sample_rate);
        }
        self.next += 1;
    }
    self.clock += 1;
    let mix: f32 = self
        .instruments
        .iter_mut()
        .map(Instrument::next_sample)
        .sum();
    mix * VOLUME
}
```

Las notas están ordenadas por comienzo, así que `next` solo avanza: una comparación por muestra cuando no empieza nada. `while let ... && ...` es una cadena de `let` en un bucle: sigue mientras *haya* una nota siguiente *y* le toque. Cada nota recibe su duración (`gate`) al empezar, y el instrumento la suelta solo, así que la frase no necesita una lista de notas que parar.

Cuando llega una frase nueva, la vieja vuelve al hilo de la interfaz por la cola de basura para liberarse. Una frase que ya tocó su última nota espera dos segundos, para que el sonido se apague, y luego se va por el mismo camino. `StopPhrase` (que se envía al cerrar la ventana o cambiar de pestaña) suelta todas las notas y descarta las que aún no empezaron.

Los instrumentos salen de `build_instrument`, la misma función que usan los canales: el piano es un canal con el programa General MIDI 0 (Piano de cola acústico), el bajo el programa 33 (Bajo eléctrico (dedos)), la batería el kit integrado. Así que con **Instrumentos del SoundFont** encendido (**Edición > Ajustes… > Audio**), las preguntas suenan con el piano, el bajo y la batería del SoundFont, como la banda (ver SoundFonts y General MIDI).

La ventana: acciones hacia fuera

La ventana no tiene el motor de audio. Devuelve lo que quiere oír como una lista de acciones, y la app las toca:

```
pub enum EarAction {
    /// Play these notes now, in place of anything it played before.
    Play(Vec<Cue>),
    /// Stop what it is playing.
    Stop,
}
```

Dentro, la pregunta y las respuestas dadas hasta ahora son un solo enum, `Asked`. Dibujar las respuestas necesita cambiarlas *y* llamar a métodos de la ventana (para sumar a la puntuación, para tocar un sonido), algo que Rust no permite mientras las respuestas están prestadas desde `self`. La ventana saca la pregunta de `self` mientras la dibuja, y la vuelve a poner:

```
match self.asked.take() {
    Some(Asked::Interval {
        question,
        mut answer,
    }) => {
        self.interval_answers(ui, &question, &mut answer, actions);
        self.asked = Some(Asked::Interval { question, answer });
    }
    // ... chords and progressions the same way
}
```

`Option::take` deja `None` en su lugar, así que durante esas pocas líneas `self` y la pregunta son dos valores separados, y los dos se pueden tomar prestados libremente.

Las opciones de la ventana se recuerdan como los demás ajustes: un struct con `#[serde(default)]` guardado en la memoria de egui, que eframe guarda con el tamaño de la ventana.

Cómo se prueba

El metrónomo y la cuenta de entrada se prueban en el motor de audio, frame a frame: `the_count_in_comes_before_the_music`, `the_count_in_runs_on_into_a_bar_started_in_the_middle` y `stopping_ends_the_count_in`. Las vueltas se prueban dos veces:

`laps_count_the_loops_played_to_the_end`, en el secuenciador, toca un bucle de cuatro pasos con saltos, reinicios y un bucle acortado, y `laps_count_the_times_the_loop_came_round` y `lap_tempos_start_on_the_loops_first_step`, en el motor, comprueban la cuenta y que una velocidad nueva empieza justo con el bucle.

En la app, `the_trainer_moves_on_as_the_loop_comes_round`, `the_next_speed_is_asked_for_ahead_of_the_lap_it_starts_on` y `the_trainer_can_play_each_speed_more_than_once` manejan el entrenador con cuentas de vueltas inventadas, incluida una ventana que se perdió unos fotogramas y un hilo de audio que se reinició; `taps_set_the_tempo_of_their_beats` y `only_the_last_taps_count` comprueban el marcado del tempo, en 4/4 y en 6/8; y `the_trainer_moves_towards_its_target_and_stays_there` las velocidades 70, 75, … 100, un paso que no divide el camino y un objetivo por debajo del comienzo.

El núcleo del entrenamiento auditivo tiene 22 pruebas. Las que vigilan la teoría:

- **Cada intervalo desde cada nota** (`every_interval_is_spelled_as_its_name_says_from_every_note`): para las 12 notas de abajo y los 12 intervalos, las dos notas escritas son las teclas correctas, ninguna necesita doble sostenido ni doble bemol, y el nombre calculado a partir de las dos notas escritas es el del intervalo.
- **Escrituras a mano** (`intervals_are_written_the_simple_way`): Do♯ Mi, Re♭ Fa, Fa Si, Mi La♯, Do Fa♯, Mi Do, Mi♭ Re, Fa♯ Fa♯, Si♭ Re♭, Si♭ Mi, La♭ Mi♭.
- **Acordes contra el analizador** (`the_app_names_a_chord_in_root_position_as_the_question_does`): 500 preguntas de acordes en estado fundamental pasan por el analizador que usa la app para nombrar lo que tocas en un teclado MIDI, que debe nombrar la misma clase. No invertidos: el analizador oye con razón Am7/C como C6.
- **Cada pregunta de acorde toca su acorde con el bajo correcto**, en 2.000 semillas con las inversiones encendidas.
- **Explicaciones escritas** (`chords_are_spelled_from_their_root`): Bdim7, E♭maj7/G, Fdim/C♭, el Si♭♭ de Cdim7, Caug/E (también Eaug), Ddim7/C♭ (también Bdim7).
- **Los números** (`numerals_are_those_of_the_harmony_keys`) coinciden con las teclas de armonía en mayor y en menor, tríadas y séptimas.
- **Las progresiones** empiezan en I, nunca repiten un acorde, vuelven a I solo al final, terminan en I o V y usan todos los acordes elegidos; cada compás toca su propio acorde sobre su fundamental.
- **Salen todas las tonalidades, fundamentales y notas de abajo**: 24 tonalidades, 12 fundamentales, 16 notas de abajo.

La frase tiene cuatro pruebas propias en `phrase.rs` (las notas empiezan a tiempo en cualquier orden en que lleguen, una frase termina cuando se ha apagado, detener descarta las notas que faltan, las notas de un instrumento que no existe se dejan fuera), y tres en el motor: una frase suena sin canción cargada y se descarta después de apagarse, una frase nueva ocupa el lugar de la anterior, y una frase detenida no toca nada más.

La partitura guía: de las notas a la notación

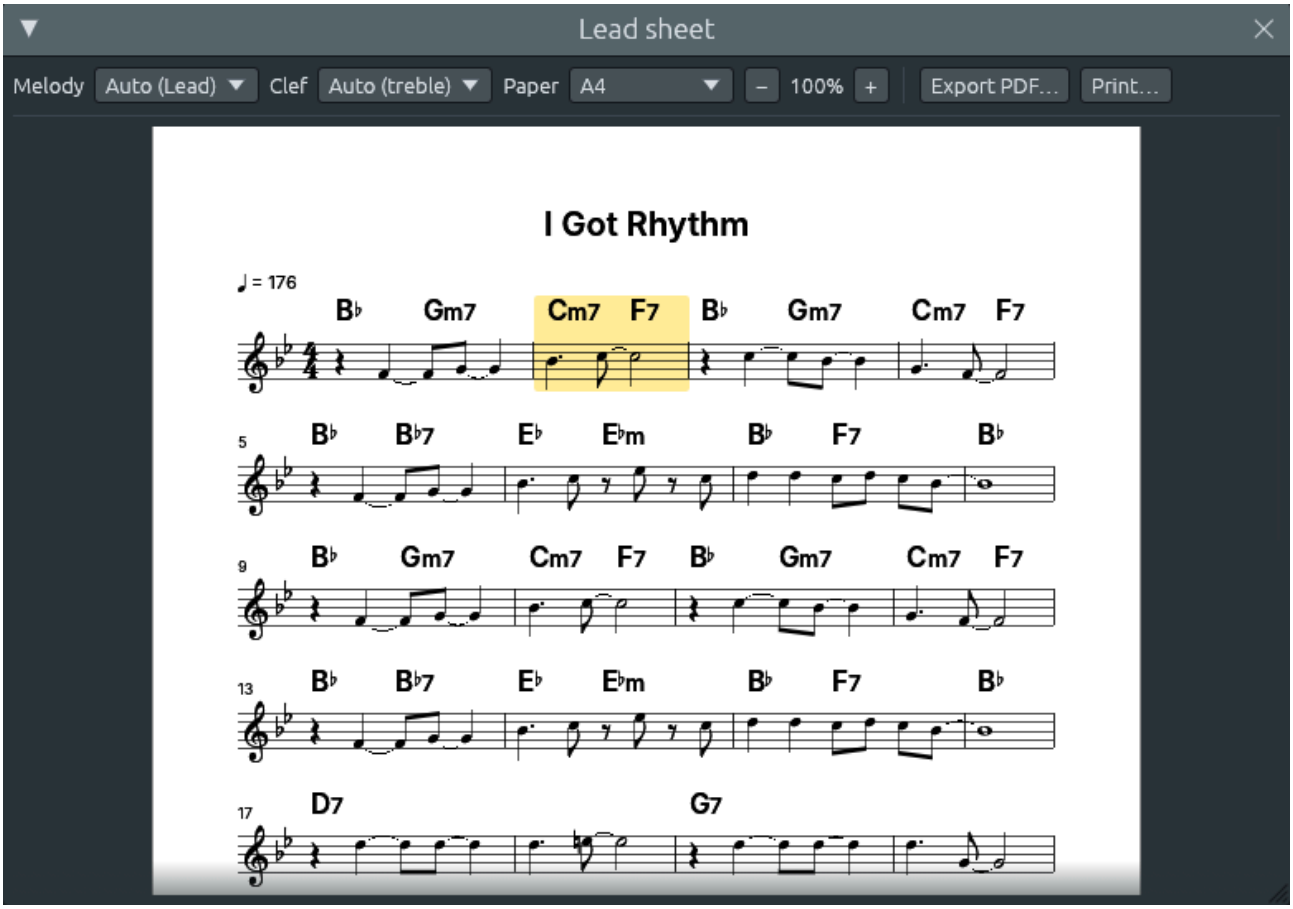
Archivos: `crates/lineup-core/src/lead_sheet.rs` y los módulos de `crates/lineup-core/src/lead_sheet/` (`melody.rs`, `spelling.rs`, `rhythm.rs`, `score.rs`, `layout.rs`, `glyphs.rs`, `transpose.rs`), y en la app `crates/lineup-app/src/lead_sheet.rs` con `fonts.rs` y `pdf.rs`

Una **partitura guía** (en inglés *lead sheet*) es lo que lee una banda. Tiene un pentagrama con la melodía, la armonía escrita encima como símbolos de acorde, y la tonalidad, el compás, las secciones y las repeticiones de la canción. Deja fuera todo lo demás: la línea de bajo, la batería y cómo dispone el piano sus acordes quedan para que los músicos lo inventen, al estilo de la noche. Los músicos de jazz, pop y folclore llevan libros enteros de ellas.

Ferrous Lineup tiene todo lo que necesita una partitura guía: la melodía son las notas de un canal, y los acordes son los de la hoja de acordes (ver La hoja de acordes). La ventana **Partitura guía** los escribe en páginas que puedes imprimir, tal como suenan o para músicos de instrumentos en Si♭, Mi♭ y Fa. Solo lee la canción: no cambia nada de ella ni del archivo.

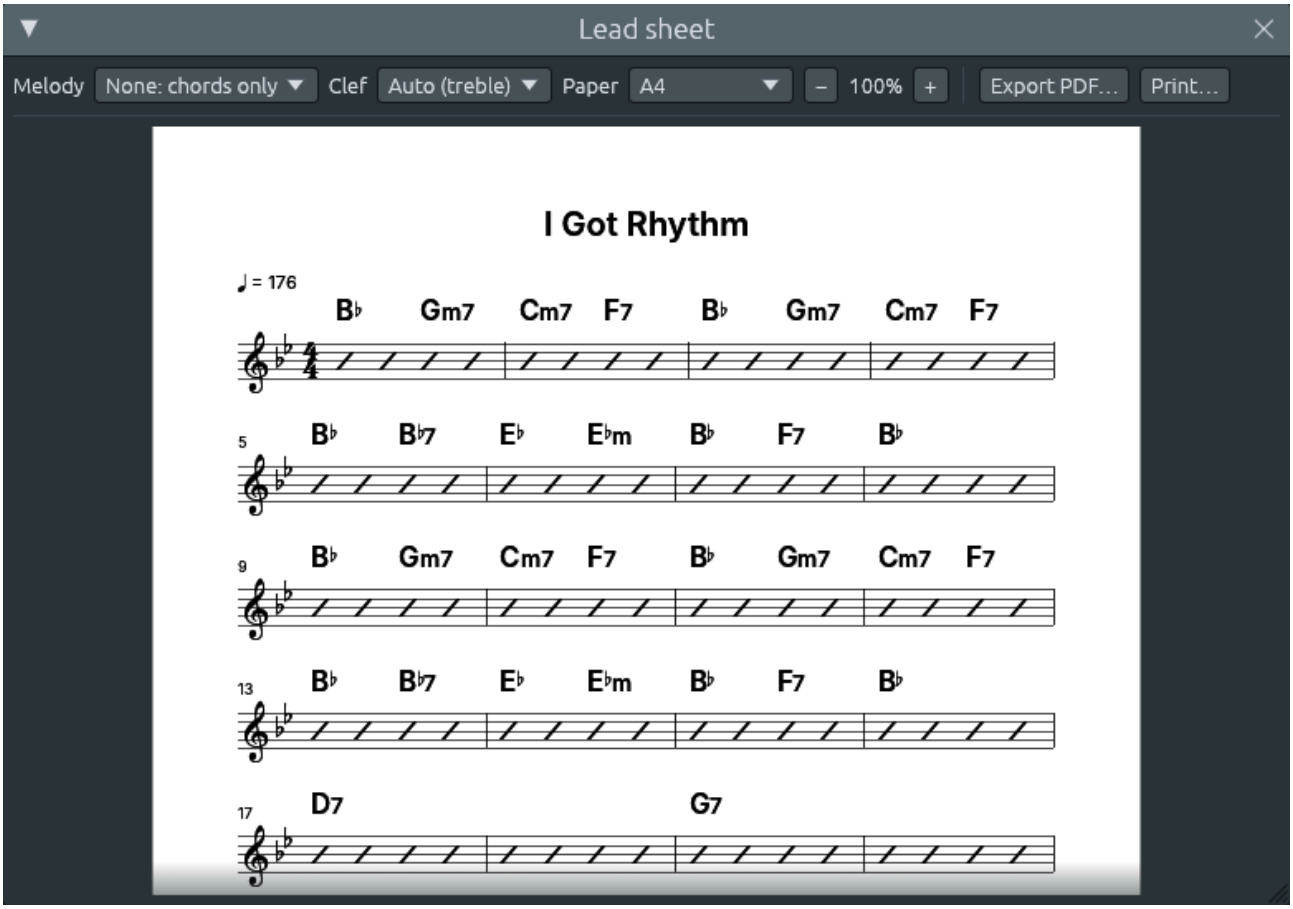
Probarlo

Ver > Partitura guía abre la ventana. La partitura se distribuye en páginas, como se imprimirá, y el compás que suena se ilumina mientras suena la canción. Cuando el compás que suena queda fuera de la vista, la ventana se desplaza hasta él.



Arriba:
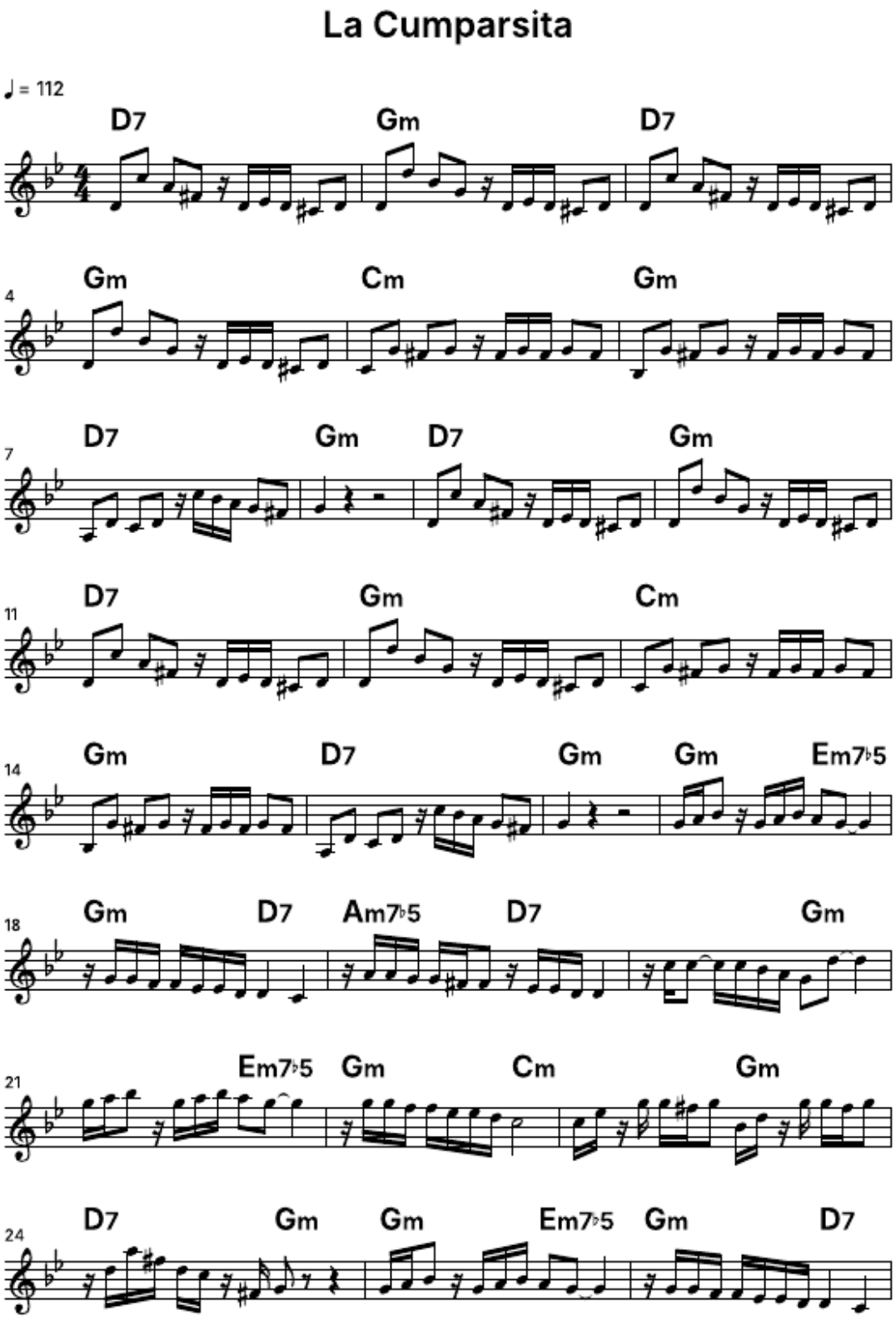
- **Melodía** elige el canal cuyas notas son la melodía. **Automático** toma el canal llamado Melodía, que tienen todas las demostraciones de canciones, o si no el canal cuyas notas son más agudas en promedio, que es donde suele estar una melodía; el menú dice cuál, como «Automático (Melodía)». Se puede elegir cualquier canal con notas, el bajo para la parte de un bajista. **Ninguna: solo acordes** no escribe melodía: cada compás lleva una barra por pulso, bajo los símbolos de acorde, como lee su parte una sección rítmica.
- **Clave**: **Automático** escribe una melodía grave en clave de fa y las demás en clave de sol; **Sol** y **Fa** eligen una.
- **Para** escribe la partitura para músicos de instrumentos en Si♭, Mi♭ o Fa (trompeta, saxofones, corno), en la tonalidad en que leen (ver Partes para instrumentos en Si♭, Mi♭ y Fa más abajo). **Altura real** la escribe tal como suena.
- **Papel**: A4 (el que se usa casi en todas partes) o Carta (en Estados Unidos y Canadá).
- **−** y **+** acercan o alejan las páginas, del 50% al 200%.
- **Exportar PDF...** guarda la partitura como archivo PDF.
- **Imprimir...** escribe el PDF en la carpeta temporal del ordenador y lo abre en tu visor de PDF, para imprimir desde allí. Imprimir así funciona igual en Linux, macOS y Windows, y el visor muestra los ajustes propios de la impresora.



El papel, el zoom y los instrumentos de **Para** se recuerdan para la próxima vez.

Leer una partitura guía

Aquí está la primera página de La Cumparsita tal como la imprime el PDF:



1 / 2

De izquierda a derecha en la primera línea:

- La **clave**. La clave de sol se enrosca alrededor de la línea de Sol (la segunda línea desde abajo); los dos puntos de la clave de fa rodean la línea de Fa.
- La **armadura**: los sostenidos o bemoles de la tonalidad, para no tener que escribirlos en cada nota. Dos bemoles (Si♭ y Mi♭) son Si♭ mayor o Sol menor. La Cumparsita está en Sol menor, y su Fa♯ (la sensible de la menor armónica) lleva un sostenido cada vez, porque la armadura no lo tiene.
- El **compás**, solo en la primera línea y donde cambia.
- El **tempo**: «♩ = 112», 112 negras por minuto. En 6/8 es una negra con puntillo, el pulso que marcas.

Encima del pentagrama están los **símbolos de acorde**, cada uno sobre el lugar donde empieza su acorde. Un acorde se escribe una sola vez, cuando cambia; sigue sonando hasta el siguiente. El número pequeño al principio de cada línea es el **número de compás** de su primer compás, para que una banda pueda decir «desde el compás 21».

Más adelante, una partitura guía puede contener:

- **marcas de ensayo**: el nombre de una sección («A», «Estrofa», «Estribillo») en un recuadro, donde empieza la sección. Cada sección empieza una línea nueva, así que la forma de la canción se ve de un vistazo;
- **signos de repetición**: una barra de compás gruesa y una fina con dos puntos, alrededor de una sección que se toca dos veces; «×3» sobre el final cuando se toca tres veces o más;
- una **doble barra** (dos líneas finas) antes de una sección nueva y antes de un cambio de tonalidad, donde se escribe la armadura nueva, con becuadros para los sostenidos o bemoles que tenía la tonalidad vieja y la nueva no;
- la **barra final** (fina y gruesa) al final.

De las notas a la notación

La notación no es una imagen del piano roll. El piano roll dice «esta tecla, desde este paso, durante tantos pasos»; la notación tiene que elegir un nombre para cada nota (¿La♯ o Si♭?), escribir cada duración como figuras que se puedan contar, y mostrar dónde están los pulsos. Los músicos leen con soltura porque estas elecciones siguen reglas, y de esas reglas trata este capítulo. Cada paso es un módulo de `lineup-core`, y ninguno sabe nada de ventanas ni de archivos PDF.

La melodía, nota a nota (`melody.rs`)

Las notas del canal de la melodía se leen a lo largo de la lista de reproducción, clip a clip, como suenan. Una partitura guía tiene una nota cada vez, así que donde empiezan varias notas juntas se queda la más aguda (la regla del «perfil»: una melodía armonizada en terceras se lee como su voz de arriba), y una nota que todavía suena cuando empieza la siguiente se corta ahí.

Las notas tocadas un poco antes o después (grabadas desde un teclado, o importadas de un archivo MIDI) se llevan a la semicorchea más cercana. Una partitura guía muestra lo que se quiere decir, no cada matiz de una interpretación.

Qué nombre de nota (`spelling.rs`)

La misma tecla del piano se puede escribir de dos maneras: la tecla negra entre Sol y La es Sol♯ o La♭. Cuál es la correcta depende de lo que *hace* la nota, y las reglas se prueban en este orden:

```
pub fn spell(key: Scale, pitch: Note, chord: Option<ChordSymbol>, next:
Option<Note>) -> NoteName {
    let pitch_class = pitch.pitch_class();
    if key.has_letter_for(pitch_class) {
        return key.spell(pitch_class);
    }
    if let Some(symbol) = chord {
        let names = symbol.chord().spell(symbol.root);
        let bass = symbol.bass.into_iter();
        if let Some(name) = names
            .into_iter()
            .chain(bass)
            .find(|name| name.pitch_class() == pitch_class &&
name.accidental.abs() <= 1)
        {
            return name;
        }
    }
    if key.kind.is_tonal()
        && let Some(next) = next
    {
        let step = i32::from(next.midi()) - i32::from(pitch.midi());
        let neighbour = match step {
            1 => Some(key.spell(next.pitch_class()).letter.up(-1)),
            -1 => Some(key.spell(next.pitch_class()).letter.up(1)),
            _ => None,
        };
        if let Some(letter) = neighbour {
            let name = NoteName::with_letter(pitch_class, letter);
            if name.accidental.abs() <= 1 && !is_odd(name) {
                return name;
            }
        }
    }
    let name = key.spell(pitch_class);
    if is_odd(name) {
        NoteName::chromatic(pitch_class)
    } else {
        name
    }
}
```

1. **Una nota de la tonalidad** se escribe como la escribe la tonalidad: en La mayor, la tecla negra encima de Do es Do♯, nunca Re♭.
2. **Una nota del acorde** que suena con ella se escribe como la escribe el acorde. Sobre E7 en Do mayor, la tecla negra encima de Sol es Sol♯, la tercera mayor de Mi (Mi Sol♯ Si Re). Sobre Fm en Do mayor la misma tecla es La♭, la tercera menor de Fa (Fa La♭ Do). Los nombres siguen las terceras del acorde.
3. **Una vecina cromática**, una nota que se mueve un semitono hasta la siguiente, toma su nombre de hacia dónde va la línea: subiendo, es el nombre de abajo

alterado hacia arriba (Fa♯ yendo a Sol); bajando, el nombre de arriba alterado hacia abajo (Si♭ yendo a La). La nota «se inclina» hacia donde va.
4. **Cualquier otra** recibe el nombre que la tonalidad da a las notas que están fuera de ella.

`is_odd` mantiene sencillas las notas de fuera: fuera de la tonalidad, una tecla blanca del piano se escribe como tal. En una tonalidad con muchos bemoles, un Si entre Do y Si♭ se escribe Do – Si – Si♭, no Do – Do♭ – Si♭, que se lee peor aunque Do♭ sea la «misma» nota.

Una vez que cada nota tiene su nombre, el compás decide qué **alteraciones** imprimir. Una alteración dura hasta la barra de compás: en Sol menor, el primer Fa♯ de un compás lleva sostenido, el siguiente Fa♯ de ese compás no, y un Fa después lleva becuadro. El compás siguiente vuelve a empezar desde la armadura.

Las series y los conjuntos de clases de altura no tienen armadura, así que una canción en una serie o en un conjunto se escribe sin ella, cada tecla negra con su propio sostenido o bemol.

Cuánto dura (`rhythm.rs`)

Una nota de cuatro semicorcheas es una negra, esté donde esté. Pero quien lee cuenta pulsos, y la notación tiene que mostrarlos. Así que cada lugar de un compás tiene una **fuerza**:
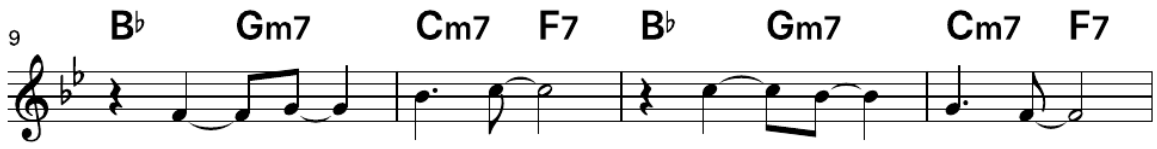
```
/// How strong a place in a bar is, counted in sixteenths from its start:
///
/// - 4 for the bar line;
/// - 3 for the middle of a bar of four even beats (beat 3 of 4/4, the
///   third dotted quarter of 12/8);
/// - 2 for any other beat;
/// - 1 for the halves of a beat (the "and" of 4/4), or the thirds of a
///   dotted beat (each eighth of 6/8);
/// - 0 for anything smaller.
pub fn strength(time: TimeSignature, offset: usize) -> u8 {
```

Las reglas, tal como las usan los grabadores de música (están escritas en *Behind Bars*, de Elaine Gould, el libro de referencia del grabado musical), dicen qué notas pueden cruzar qué lugares:

- Una nota puede sostenerse por encima de un pulso solo si empieza en un pulso. Una negra que empieza en el «y» del pulso 2 en 4/4 esconde el pulso 3, así que se escribe como dos corcheas **ligadas**: la segunda nota no se vuelve a tocar, la ligadura (una curva fina) sostiene la primera.

- Nada esconde el centro de un compás de cuatro pulsos, salvo que empiece en la barra de compás. Una nota del pulso 2 al final del compás es una negra ligada a una blanca.
- Se deja pasar un patrón porque todo lector lo conoce: la síncopa «♪ ♩ ♪», una negra desde la mitad de un pulso hasta la mitad del siguiente, dentro de una mitad del compás.
- En 6/8 y en métricas agrupadas como 2+2+3/8 las notas se sostienen solo de pulso a pulso, para que se vean los grupos.
- Los silencios son más estrictos todavía: un silencio cubre varios pulsos solo desde la barra de compás o desde el centro del compás, como un silencio de blanca en el pulso 1 o 3 de 4/4.

Cuando una nota no cabe en una figura, se corta primero en el lugar más fuerte que hay dentro de ella, y los trozos se ligan. Aquí hay cuatro compases de I Got Rhythm: las síncopas de la melodía salen como negras y corcheas ligadas por encima del pulso 3, el centro del compás, así que quien lee siempre ve dónde está.



Las **barras** unen corcheas y semicorcheas pulso a pulso (por el compás entero cuando los pulsos son corcheas, como en 3/8), así que aquí también se ven los pulsos. Una barra se inclina con las notas, pero nunca más de un espacio del pentagrama, y una semicorchea junto a una corchea lleva una segunda barra corta propia, un «muñón».

Qué contiene un compás (`score.rs`)

`score.rs` junta todo esto compás a compás. Un compás contiene sus notas y silencios (cada uno con su figura, su nombre, su alteración y su ligadura), los acordes que empiezan en él, y lo que se escribe en sus bordes: una tonalidad o un compás nuevos, el comienzo de una sección, signos de repetición. Antes de la primera nota de la melodía y después de la última, un compás donde la banda toca un acorde lleva barras en vez de silencios, así que una introducción o un final que la banda toca sola se lee como «toca estos acordes».

Dónde va todo (`layout.rs`)

Después todo se coloca en la página, en puntos (1/72 de pulgada). El pentagrama mide 21,2 puntos de alto (7,5 mm), cuatro **espacios** de 5,3 puntos, el tamaño habitual de una partitura guía. Todas las medidas de la distribución se cuentan en espacios.

Las notas reciben espacio según lo que duran, pero no en proporción: una blanca no recibe el doble que una negra, o un compás de semicorcheas quedaría aplastado junto a un compás de redondas. Los grabadores espacian según algo parecido a la raíz cuadrada de la duración:

```
/// How much room a note or rest gets for lasting `steps` sixteenths,
/// before the line is stretched: about 2.6 spaces for a sixteenth, 4.2
/// for a quarter, 7.4 for a whole note.
fn spring(steps: usize) -> f32 {
    sp(1.0 + 1.6 * (steps as f32).sqrt())
}
```

Un símbolo de acorde es más ancho que una nota, así que un cambio de acorde rápido («Gm G7» sobre dos negras) separa sus notas hasta que los símbolos no se tocan.

Una línea tiene hasta cuatro compases, menos si no caben, y una sección nueva empieza una línea nueva. Cada línea completa se **estira** luego hasta llegar exactamente al margen derecho. Cuánto estirarla se encuentra probando: se dibuja la línea con un estiramiento de 0,25 y de 6, y luego se parte el intervalo por la mitad 24 veces, quedándose con la mitad donde cae el ancho. Es una **búsqueda binaria**, la misma idea que buscar una palabra en un diccionario abriéndolo por la mitad:

```
fn fit(&self, bars: Range<usize>, width: f32) -> f32 {
    let wide = |stretch: f32| self.draw_system(bars.clone(),
stretch).width;
    let (mut low, mut high) = (0.25_f32, 6.0_f32);
    // ...
    for _ in 0..24 {
        let middle = (low + high) / 2.0;
        if wide(middle) > width {
            high = middle;
        } else {
            low = middle;
        }
    }
    low
}
```

`wide` es un **closure**, una pequeña función escrita en línea que toma prestados `self` y `bars` de su alrededor. La línea se dibuja hasta 26 veces para colocarla una; es

rápido, porque dibujar una línea solo calcula posiciones.

La última línea de una sección o de la canción suele ser corta. Estirada hasta el margen, sus pocas notas quedarían muy separadas, así que se espacia como las líneas completas y se deja corta, como hacen los grabadores. Luego las líneas van a las páginas, repartidas para que cada página se llene de arriba abajo, salvo la última.

Una lista de marcas

El resultado de la distribución es una lista de **marcas** por página: valores simples que dicen qué dibujar y dónde.

```
pub enum Mark {
    /// A symbol of the music font with its origin (the SMuFL baseline)
    /// at `at`; `size` is the font size, the height of a staff.
    Music {
        glyph: glyphs::Glyph,
        at: Point,
        size: f32,
    },
    /// Text with the left end of its baseline at `at`.
    Text {
        text: String,
        font: TextFont,
        at: Point,
        size: f32,
    },
    /// A straight line `width` thick, with square ends.
    Line { from: Point, to: Point, width: f32 },
    // ... Beam and Tie, each with two points and a thickness
}
```

La ventana pinta estas marcas en la pantalla y `pdf.rs` escribe las mismas marcas en el PDF, así que lo que se imprime es exactamente lo que se vio. Es la misma separación que usa la app en todas partes: `lineup-core` decide, la app solo dibuja. También hace que la distribución se pueda probar sin ventana: una prueba puede distribuir una canción y buscar una cabeza de nota por nota, un recuadro por compás, y nada fuera de la página.

Las fuentes

Los símbolos musicales salen de una **fuente musical**. Las fuentes que siguen **SMuFL**, el Standard Music Font Layout, dan a cada símbolo un punto de código fijo (la clave de sol es U+E050 en todas) y están dibujadas para que un em, el tamaño de la fuente,

sea la altura del pentagrama. La app usa **Bravura**, la fuente con la que se diseñó SMuFL, de Steinberg. Las palabras y los símbolos de acorde van en **Inter**, una fuente hecha para pantallas que también se imprime limpia. Las dos son libres, bajo la SIL Open Font License, y van dentro del programa con `include_bytes!`, así que la partitura se ve igual en todos los ordenadores:

```
const BRAVURA: &[u8] = include_bytes!("../../assets/fonts/Bravura.otf");
```

`include_bytes!` es una macro que lee el archivo al compilar el programa y pone sus bytes dentro de él, como un `&'static [u8]`: un slice que vive tanto como el programa.

La distribución necesita saber qué ancho tiene cada símbolo y cuánto sube y baja. Esos números salen de los metadatos de la propia Bravura, copiados en `glyphs.rs`, y una prueba lee el archivo de la fuente para comprobar cada uno.

El texto se mide con la misma fuente con la que se dibuja. `lineup-core` no tiene fuentes, así que pregunta a través de un **trait**:

```
pub trait TextWidth {
    /// The width of `text` in `font` at `size` points, in points.
    fn text_width(&self, text: &str, font: TextFont, size: f32) -> f32;
}
```

La app lo implementa sumando los anchos de las letras leídos del archivo de la fuente con el crate `ttf-parser`; las pruebas lo implementan con una regla simple (cada letra la mitad de ancha que el tamaño de la fuente). La distribución recibe un `&dyn TextWidth` y funciona con cualquiera de los dos.

Los símbolos de acorde siguen la costumbre de las partituras guía: la letra de la fundamental grande, el resto un poco más pequeño («Am7», «G7»), y signos ♯ y ♭ de verdad, de la fuente musical, elevados, en vez de «#» y «b». Las fundamentales van **en letras sea cual sea el nombre de las notas en la app**: con solfeo elegido, la hoja de acordes dice Rem7, pero la partitura guía escribe Dm7, como lo hacen las partituras guía impresas en todos los idiomas, para que una banda la pueda leer en cualquier parte.

El PDF

Una página de PDF es una lista de operaciones de dibujo, muy parecida a las marcas: ir a un punto, trazar una línea hasta otro, rellenar una forma. El crate `pdf-writer` las

escribe en el formato del PDF; funciona en todas las plataformas y no necesita nada del sistema.

Las letras y los símbolos no se incrustan como fuentes. Cada uno se copia como su **contorno**, las curvas que lo dibujan, leído del archivo de la fuente con `ttf-parser`, y se rellena. Un símbolo que se usa muchas veces, como una cabeza de nota negra, se escribe una vez como «form XObject» y luego se coloca dondequiera que aparezca, escalado a su tamaño:

```
content
    .save_state()
    .transform([scale, 0.0, 0.0, scale, x, height - at.y])
    .x_object(Name(name.as_bytes()))
    .restore_state();
```

El `transform` es una matriz que escala la forma de las unidades de la fuente a puntos y la mueve a su sitio. Fíjate en `height - at.y`: el PDF cuenta la y hacia arriba desde el pie de la página, la distribución la cuenta hacia abajo desde arriba, así que cada y se da la vuelta. El PDF se ve igual en cualquier visor, sin instalar nada.

Partes para instrumentos en Si♭, Mi♭ y Fa

Pon una partitura guía de una canción en Do delante de un trompetista y de un pianista, pídeles que toquen juntos la primera nota, y no van a coincidir. El pianista toca Do; la trompeta, leyendo el mismo Do, suena el Si♭ un tono más abajo. La trompeta es un **instrumento transpositor**: las notas que lee no son las que suenan. Para tocar con la banda, el trompetista necesita una partitura escrita un tono más arriba, en Re.

Por qué algunos instrumentos leen en otras tonalidades

Los instrumentos de viento vienen en familias de distintos tamaños: los saxofones del soprano al barítono, los clarinetes, los cornos. Cada tamaño suena más grave o más agudo, pero los dedos son los mismos: en todos los saxofones, los mismos dedos tocan la nota escrita Do. Si la música se escribiera como suena, alguien que pasa del alto al tenor tendría que aprender digitaciones nuevas para cada nota. Así que la música se escribe para los dedos, y cada instrumento se nombra por la nota que suena cuando toca un Do escrito: una trompeta **en Si♭**, un saxofón alto **en Mi♭**, un corno **en Fa**.

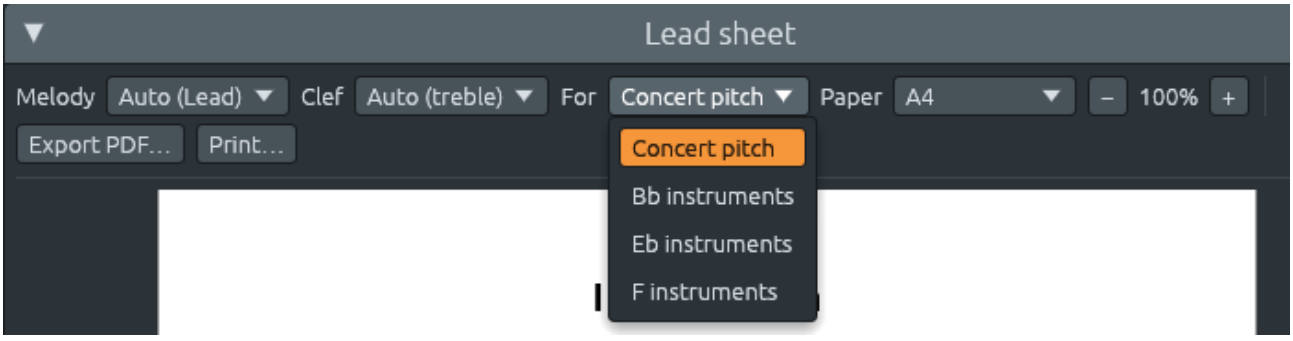
La música escrita como suena está en **altura real**, o «en Do». El piano, la guitarra, el bajo, la voz, la flauta, el violín y el trombón la leen así.

Instrumentos	Un Do escrito suena	La parte se escribe
Piano, guitarra, bajo, voz, flauta, violín, trombón	Do	como suena
Trompeta, corneta, fliscorno, clarinete, saxofón soprano	el Si♭ de abajo	una segunda mayor más arriba
Saxofón tenor	el Si♭ una octava y una segunda más abajo	una novena mayor más arriba
Saxofón alto	el Mi♭ de abajo	una sexta mayor más arriba
Saxofón barítono	el Mi♭ una octava y una sexta más abajo	una octava y una sexta más arriba
Corno, corno inglés	el Fa de abajo	una quinta justa más arriba

La parte siempre se escribe más aguda de lo que suena, y dentro de una familia por el mismo intervalo, octava más octava menos, así que una canción en Do se escribe en Re para todos los instrumentos en Si♭, en La para todos los instrumentos en Mi♭ y en Sol para todos los instrumentos en Fa.

Cómo usarlo

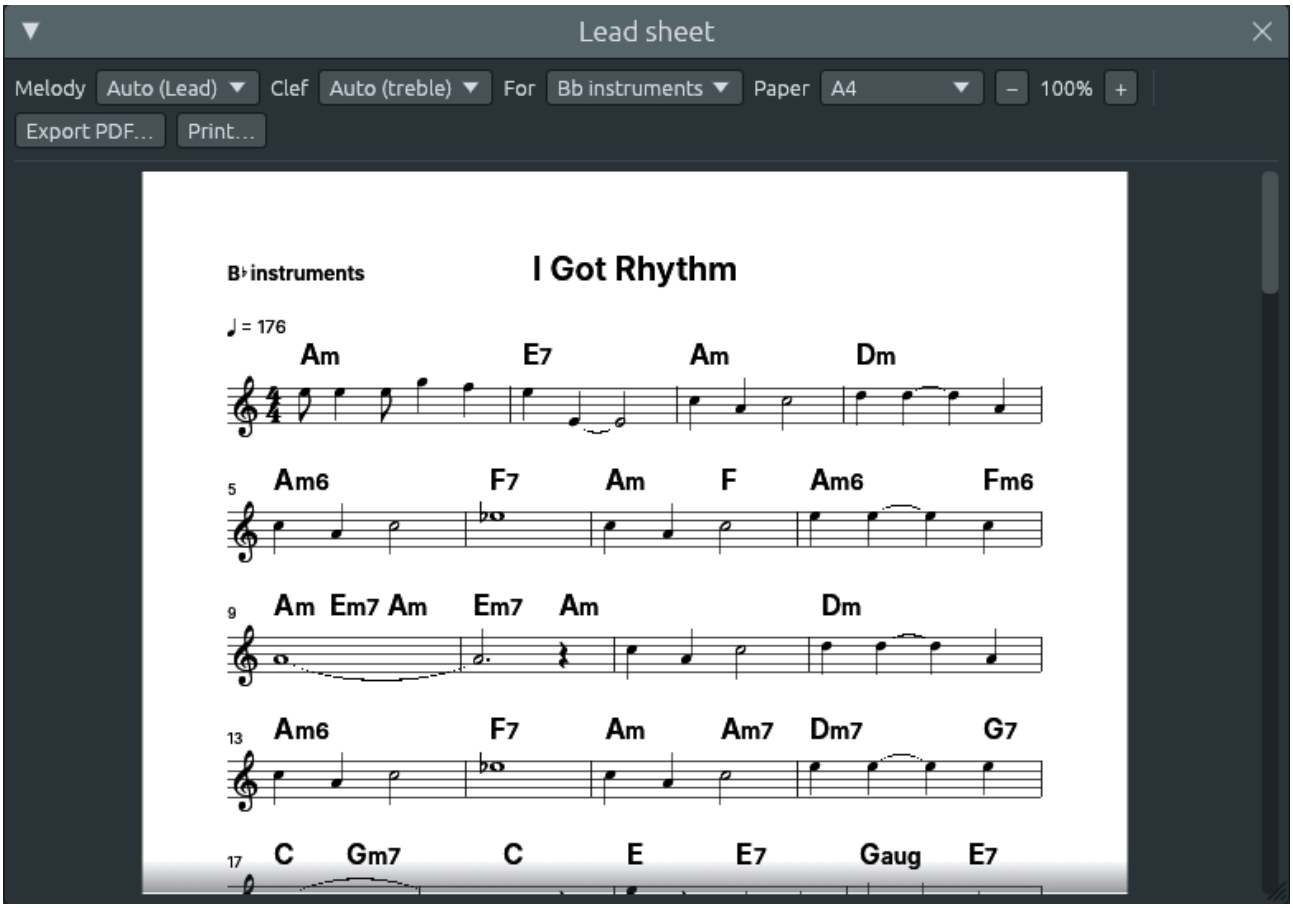
En la ventana Partitura guía, **Para** ofrece:



- **Altura real** escribe la canción tal como suena.
- **Instrumentos en Si♭**: trompeta, clarinete, saxofón soprano y tenor.
- **Instrumentos en Mi♭**: saxofón alto y barítono.
- **Instrumentos en Fa**: corno y corno inglés.

Pasa el ratón sobre una opción para ver para qué instrumentos es y cuánto más arriba se escribe. La partitura cambia al momento: la armadura, la melodía y los símbolos de acorde se mueven juntos, y la esquina superior izquierda de la primera página dice para quién es la parte, «Instrumentos en Si♭», donde las partes impresas nombran su instrumento. **Exportar PDF…** propone un nombre de archivo con los instrumentos, para que las partes de una canción no se pisen unas a otras. La elección se recuerda: un saxofonista que abre otra canción la encuentra escrita para él de nuevo.

Aquí está la demostración I Got Rhythm para instrumentos en Si♭. La canción está en Si♭ mayor, dos bemoles; la parte está en Do mayor, sin sostenidos ni bemoles. Esta es una de las razones por las que tantos temas de jazz están en tonalidades con bemoles: el Si♭ real es Do para la trompeta y el saxofón tenor, y el Mi♭ real es Do para el saxofón alto.



La demostración empieza con una introducción sobre acordes de Sol menor (Gm, D7, Cm). En la parte para Si♭ son Am, E7 y Dm; en la parte para Mi♭, en Sol mayor, Em, B7 y Am; en la parte para Fa, en Fa mayor, Dm, A7 y Gm. La canción, su archivo y lo que toca la banda se quedan como están. Solo la partitura se escribe distinta.

La tonalidad: nunca más de seis sostenidos o bemoles

Mover la tonalidad es sumar semitonos: dos para los instrumentos en Si♭ (de Si♭ a Do), nueve para los instrumentos en Mi♭ (de Mi♭ a Do) y siete para los instrumentos en Fa (de Fa a Do).

```
pub fn semitones(self) -> u8 {
    self.note().pitch_class().semitones_up_to(PitchClass::C)
}
```

```
pub fn key(self, key: Scale) -> Scale {
    Scale::new(key.root.transpose(i32::from(self.semitones())), key.kind)
}
```

`note()` es la nota que suena con un Do escrito, Si♭ para `Transposition::BFlat`, así que los nombres de los instrumentos y sus distancias salen de un solo lugar. La clase de tonalidad se conserva: Fa menor es Sol menor para una trompeta, Re dórico es Si dórico para un saxofón alto.

La tonalidad nueva es un `Scale` como cualquier otro, así que se nombra como la app nombra cualquier tonalidad (ver Sostenidos o bemoles: cómo se escribe una tonalidad): cada una de las doce tonalidades tiene un nombre con como mucho seis sostenidos o bemoles, y es el que se usa. Esto importa para las partes, porque contar hacia arriba por nombres daría a veces tonalidades en las que nadie escribe:

Canción	Parte en Si♭	Parte en Mi♭	Parte en Fa
Do	Re	La	Sol
Fa	Sol	Re	Do
Si♭	Do	Sol	Fa
Mi♭	Fa	Do	Si♭
La♭	Si♭	Fa	Mi♭
Re♭	Mi♭	Si♭	La♭
Sol	La	Mi	Re
Re	Mi	Si	La
La	Si	Fa♯	Mi
Mi	Fa♯	Re♭	Si
Si	Re♭	La♭	Fa♯
Fa♯	La♭	Mi♭	Re♭

Una canción en Mi para un saxofón alto, una sexta mayor más arriba, estaría estrictamente en Do♯ mayor, siete sostenidos. Re♭ mayor tiene las mismas notas con cinco bemoles, y es lo que escribe un copista. Una canción en Fa♯ para una trompeta estaría en Sol♯ mayor, que necesita un Fa doble sostenido en su armadura; La♭ mayor, cuatro bemoles, tiene las mismas notas. Una prueba recorre las doce tonalidades mayores y menores para cada instrumento y comprueba que ninguna parte tiene más de seis.

Los acordes: movidos con nombre y todo

Un acorde tiene que conservar su lugar en la tonalidad: el V7 de la canción es el V7 de la parte. Un cifrado se lleva a otra tonalidad contando nombres además de semitonos (ver Llevar una hoja de acordes a otra tonalidad), y la parte usa la misma función, de la tonalidad de la canción a la de la parte:

```
pub fn harmony(self, harmony: HarmonyName, key: Scale) -> HarmonyName {
    harmony.moved(key, self.key(key))
}
```

Como va de tónica a tónica, sigue también la escritura de la tonalidad. De Do a Re cada nombre sube uno, así que Dm7 G7 Cmaj7 pasa a Em7 A7 Dmaj7. De Fa♯ a La♭ cada nombre sube dos, así que el V7 de Fa♯, C♯7, pasa a E♭7, el V7 de La♭, y nunca D♯7. La demostración de bossa nova Garota de Ipanema está en Re♭; para un saxofón alto, en Si♭, sus D♭maj7 E♭7 E♭m7 D7 son B♭maj7 C7 Cm7 B7.

Las notas: cada intervalo conserva su nombre

La melodía se mueve los mismos semitonos, pero sus notas también necesitan nombres, y aquí un atajo saldría mal. La manera obvia es subir cada nota y volver a escribirla en la tonalidad nueva con las reglas de Qué nombre de nota. La Marcha turca de Mozart, en La menor, muestra el problema. Su melodía se apoya en Mi desde el semitono de abajo, Re♯–Mi. En Si menor, la tonalidad de la trompeta, los mismos sonidos se volverían a escribir como Fa–Fa♯: las reglas mantienen sencilla una tecla blanca fuera de la tonalidad, así que escriben Fa en vez de Mi♯. Pero Re♯–Mi es una segunda menor, una nota que se apoya en la siguiente, y Fa–Fa♯ es un unísono aumentado, un mismo nombre alterado hacia arriba. Quien toca lee otra línea.

Así que las notas se escriben primero en altura real, donde las reglas conocen la tonalidad y los acordes en que se escribió la canción, y luego cada nombre se mueve los mismos nombres y semitonos que la tonalidad:

```
pub fn name(self, name: NoteName, key: Scale) -> NoteName {
    let written = self.key(key);
    if written == key {
        return name;
    }
    if !key.as_key().kind.is_tonal() {
        return
written.spell(name.pitch_class().transpose(i32::from(self.semitones())));
    }
    let (from, to) = (key.spell(key.root), written.spell(written.root));
    let moved = name.above(
        from.pitch_class().semitones_up_to(to.pitch_class()),
        from.letter.steps_up_to(to.letter),
    );
    if moved.accidental.abs() >= 2 && written.spell(moved.pitch_class())
!= moved {
        moved.simplified()
    } else {
        moved
    }
}
```

`NoteName::above` es la misma función que mueve las fundamentales de los acordes, así que las notas y los acordes se mueven exactamente los mismos nombres. El Re♯–Mi de la Marcha turca es Mi♯–Fa♯ para una trompeta, Si♯–Do♯ para un saxofón alto y La♯–Si para un corno: una segunda menor cada vez, como lo escribió Mozart. Una serie o un conjunto no tienen nombres propios (el segundo `if`), así que sus notas toman sus nombres habituales en la tonalidad nueva.

Las últimas líneas se ocupan de las tonalidades que se renombraron. Una canción en Mi tiene una sexta rebajada, Do. Para un saxofón alto, en Re♭ en vez de Do♯, los nombres se mueven de Mi a Re, y Do pasa a ser Si doble bemol (Si♭♭), la sexta rebajada de Re♭ escrita en sentido estricto. Nadie quiere leer eso, y es la misma nota que La, así que el nombre se simplifica, como en los símbolos de acorde. Un doble sostenido o doble bemol que tiene la propia tonalidad nueva se queda: la sensible de Do♯ menor armónica es Si♯, y en la parte de trompeta, en Re♯ menor armónica, es Do doble sostenido (Do♯♯), la sensible de esa tonalidad.

La octava: dónde se sienta la melodía en el pentagrama

El intervalo dice qué notas escribir, no en qué octava. Un saxofón tenor suena una novena por debajo de lo escrito, una octava más grave que una trompeta, y aun así en los libros de partituras guía para instrumentos en Si♭ los dos leen la misma página: el tenor simplemente la suena una octava más abajo. Lo que importa en la página es que la melodía se siente en el pentagrama, no en una escalera de líneas adicionales por encima.

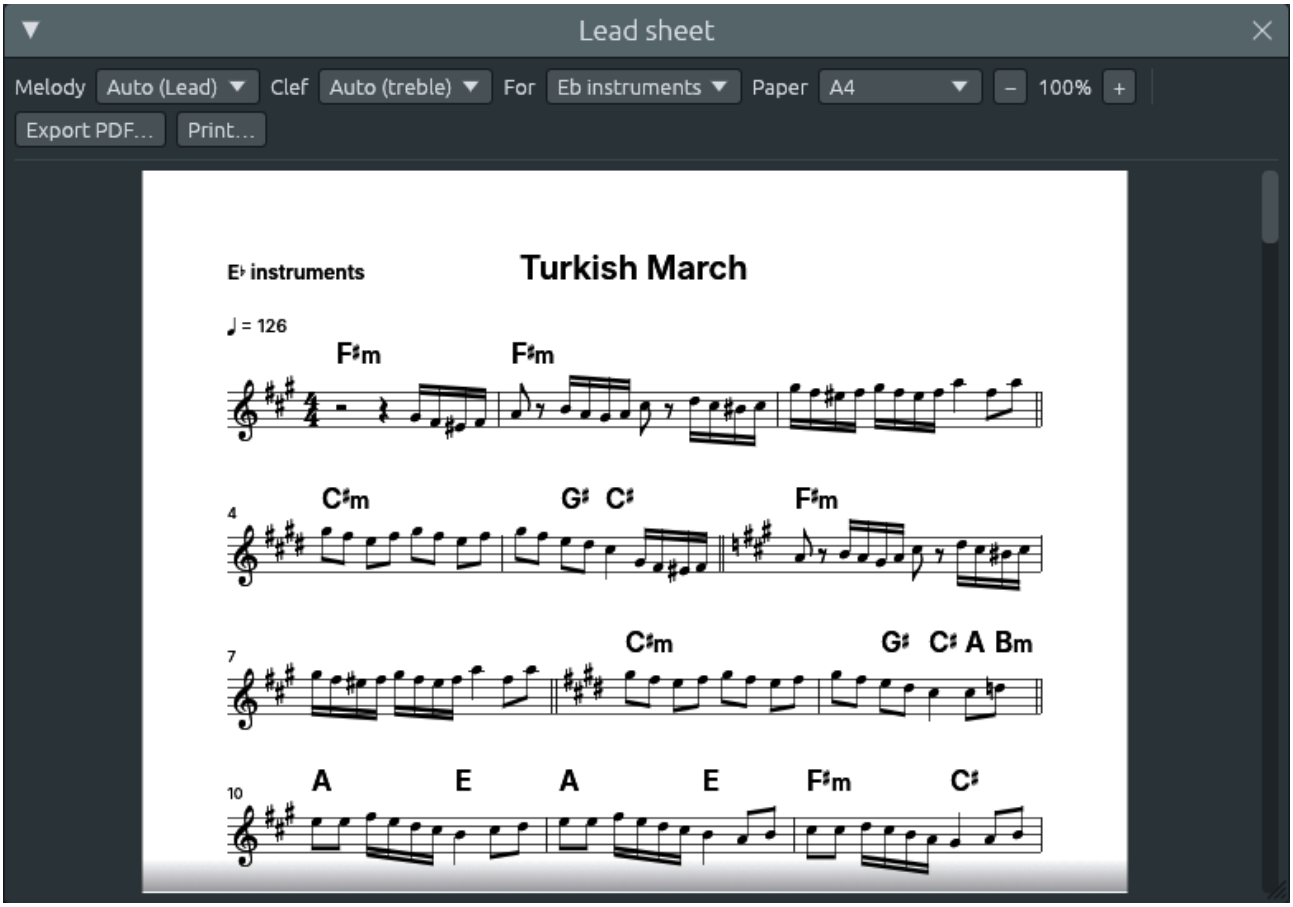
Así que la parte se escribe por el intervalo del instrumento, movida por octavas enteras hasta que la nota central de la melodía quede lo más cerca posible de la línea central del pentagrama (Si4 en clave de sol):

```
pub fn shift(self, pitches: &[Note], clef: Clef) -> i32 {
    let interval = i32::from(self.semitones());
    if self == Transposition::Concert {
        return 0;
    }
    let mut sorted = pitches.to_vec();
    sorted.sort();
    let Some(median) = sorted.get(sorted.len() / 2) else {
        return interval;
    };
    let middle = i32::from(clef.middle_line().midi());
    let median = i32::from(median.midi());
    (-4..=4)
        .map(|octaves| interval + 12 * octaves)
        .min_by_key(|shift| ((median + shift - middle).abs(), (shift -
interval).abs()))
        .unwrap_or(interval)
}
```

La nota central es la **mediana**, la que queda en medio una vez ordenadas las notas, como al elegir la clave: unas pocas notas muy agudas o muy graves no la arrastran. Para una melodía en el registro de una voz, el resultado es el propio intervalo del instrumento: una segunda mayor arriba para los instrumentos en Si♭, una sexta mayor para los de Mi♭, una quinta para los de Fa, así que trompeta, saxofón alto y corno suenan la melodía donde está. Una melodía alrededor de Do6 sale una séptima menor *más abajo* para los instrumentos en Si♭, los mismos nombres una octava más abajo, con su nota central en Re5, donde una trompeta la puede tocar.

`min_by_key` se queda con el candidato de clave más pequeña, y aquí la clave es un par. Los pares se comparan como palabras en un diccionario: por el primer elemento, y por el segundo solo cuando los primeros son iguales. Así que decide la distancia a la línea central, y cuando dos octavas están igual de cerca, gana la más cercana al intervalo propio del instrumento.

Las partes para instrumentos transpositores usan la clave de sol, que leen todos ellos; **Clave** todavía puede elegir la clave de fa, y entonces la melodía se centra en la línea central de ese pentagrama, Re3.



En el código

La partitura guía está dividida en pasos, y solo algunos saben de instrumentos:

- `transpose.rs` tiene el tipo `Transposition` con `key`, `harmony`, `name` y `shift`.
- `score.rs` le pide la tonalidad de cada compás, los acordes, los nombres de las notas y la octava. Dos closures lo mantienen corto:

  ```
  let key_at = |step: usize| transposition.key(keys.key_at(step));
  let harmony_of =
      |marker: ChordMarker| transposition.harmony(marker.harmony,
  keys.key_at(marker.start));
  ```

  Toman prestados `keys` y `transposition` de la función que los rodea, así que cada llamada solo dice qué paso o qué marca.

- `layout.rs` escribe «Instrumentos en Si♭» arriba a la izquierda de la primera página, con el bemol de la fuente musical como en los símbolos de acorde. Cuando el título es demasiado ancho para dejar sitio al lado, las palabras van en una línea debajo.

- `melody.rs` y `rhythm.rs` no saben nada de instrumentos: la melodía se lee, se escribe en altura real, se redondea y se corta en figuras de la misma manera para todos los instrumentos.

En la app, la transposición se guarda con el papel y el zoom en los ajustes de la ventana:

```
#[derive(Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(default)]
struct Settings {
    paper: Paper,
    zoom: f32,
    /// Who reads the sheet: a player keeps their instrument from song to
    /// song.
    transposition: Transposition,
}
```

`#[serde(default)]` deja que llegue un campo sin romper nada: los ajustes guardados sin `transposition` lo toman de `Settings::default()`, altura real. `Transposition` toma su valor por defecto de un atributo en una variante:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default, Serialize,
Deserialize)]
pub enum Transposition {
    /// Written as it sounds: piano, guitar, bass, voice, flute, violin,
    /// trombone and every other instrument "in C".
    #[default]
    Concert,
    // ... BFlat, EFlat and F
}
```

El archivo del proyecto no cambia: quién lee la partitura es una elección de quien la lee, no parte de la canción.

Cómo se prueba

`lineup-core` prueba cada paso de la partitura guía por separado:

- **escritura** (`spelling.rs`): los nombres de la tonalidad; los nombres de un acorde (Sol♯ sobre E7 y La♭ sobre Fm en Do mayor); los pasos cromáticos según la dirección; las teclas blancas fuera de la tonalidad que se quedan sencillas; las tonalidades menores que suben su sensible; las alteraciones que duran hasta la barra de compás; las armaduras en las dos claves; los becuadros de un cambio de tonalidad;

- **ritmo** (`rhythm.rs`): la fuerza de cada lugar; las notas en el pulso que se sostienen por encima de pulsos; el centro de un compás de cuatro; una nota fuera del pulso que no esconde el pulso siguiente; la síncopa «♪ ♩ ♪» sin ligar; los pulsos compuestos enteros; los compases desiguales partidos en sus pulsos; los silencios que muestran cada pulso; las barras por pulso; cada trozo una figura escrita;
- **melodía y partitura** (`melody.rs`, `score.rs`): la nota de arriba de un acorde como melodía, las notas fuera de la cuadrícula en el paso más cercano, un clip repetido que vuelve a sonar, las notas ligadas por encima de la barra de compás, la clave de fa para melodías graves, barras o silencios en compases sin melodía, marcas de ensayo, repeticiones y cambios de tonalidad;
- **distribución** (`layout.rs`): una cabeza de nota por nota y un recuadro por compás, todo dentro de la página, una sección que empieza una línea nueva, acordes solos sobre barras, símbolos de acorde que nunca se pisan y usan alteraciones de verdad con cualquier nombre de notas, barras y ligaduras, repeticiones, y una parte que dice quién la lee, al lado o debajo del título;
- **partes** (`transpose.rs` y `score.rs`): el intervalo de cada instrumento; una canción en Do escrita en Re, La y Sol, las tonalidades menores y los modos que conservan su clase; ninguna tonalidad de parte con más de seis sostenidos o bemoles, para las doce tonalidades mayores y menores y cada instrumento; los acordes que conservan su lugar (ii–V–I en Do es Em7 A7 Dmaj7, C♯7 en Fa♯ es E♭7 en La♭, también un poliacorde y N.C.); las notas que conservan sus intervalos (Re♯–Mi en La menor pasa a Mi♯–Fa♯, Si♯–Do♯ y La♯–Si; la sexta rebajada de Mi es La en Re♭; Do♯♯ se queda en Re♯ menor armónica); la octava de una melodía alrededor de Sol4, de una melodía aguda y de una línea de bajo, en las dos claves; una parte de trompeta en Re mayor; un cambio de tonalidad en las tonalidades de las partes; la clave de sol para las partes.

La app tiene cuatro: cada símbolo de la fuente musical está en Bravura al tamaño que cree la distribución, el texto se mide con la fuente, una canción se vuelve un PDF con un objeto de página por página y sus formas escritas una sola vez, y una parte se guarda con un nombre propio, `Op. 24 (Bb instruments).pdf`.

Las pruebas no pueden decir si una partitura se lee bien, así que las partituras también se leen como las lee un músico: Ode to Joy, Für Elise, Minuet in G, Careless Love, Korobeiniki, Funiculì, funiculà, The Entertainer, la Sonata Claro de luna, El Choclo, Greensleeves, La Cumparsita, When the Saints Go Marching In e I Got Rhythm en altura real, e I Got Rhythm, la Marcha turca, con sus muchos cambios de tonalidad, y Garota de Ipanema como partes.

Ejercicios: la banda

Estos ejercicios practican esta parte sobre el código real: la biblioteca, la hoja de acordes, la banda y sus músicos, la orquesta, la forma de la canción, las paradas, el solista, tus propios estilos, las herramientas de práctica y la partitura guía. Cada uno dice qué **ya existe**, para que lo amplíes en lugar de volver a construirlo. Algunos son cosas que probar en la aplicación y escuchar; otros son piezas de Rust con un final claro; los que se llaman «Piénsalo» son preguntas, sin código que escribir.

Haz los de Rust en una rama propia. Ejecuta `cargo test` después de cada cambio, y `cargo fmt --all --check` y `cargo clippy --workspace --all-targets -- -D warnings` antes de dar uno por terminado: la CI ejecuta los tres. Un ejercicio que añade algo al archivo de proyecto necesita el siguiente número de formato libre (ver El número de versión).

La biblioteca

1. Un estilo propio (lineup-core, fácil)

`STYLES`, en `library/style.rs`, es una tabla de filas `style(...)` (ver Estilos: una tabla en un `static`). Añade una: la progresión de una canción que te guste, escrita en números romanos. Por ejemplo el Canon de Pachelbel entero, `I | V | vi | iii | IV | I | IV | V` (el estilo **Balada de piano** tiene solo sus seis primeros acordes), con `Comping::ArpUpDown` y `Bass::Held`. Ejecuta `cargo test -p lineup-core` antes de abrir la aplicación: si un número romano está mal escrito, si un compás no puede contener sus acordes o si el nombre ya existe, `every_style_parses_and_fits_in_a_pattern` o `names_are_unique_and_rhythms_exist` nombran el estilo.

Después pruébalo en la aplicación sobre una canción en otra tonalidad. ¿Qué nombres de acordes cambian, y qué números romanos siguen iguales?

Practica: tablas `const fn`, literales de struct, leer los fallos de los tests.

2. Un ritmo con cencerro y un fill (lineup-core, fácil)

Añade un patrón de batería a `RHYTHMS` en `library/rhythm.rs`, en el bloque de su género, con una línea para `Kick`, otra para `Snare` y otra para `Cowbell`, y un fill que exista, como `"Tom fill"`. En 4/4 cada línea tiene dieciséis caracteres, de entre `X`, `x`, `o`, `h`, `l` y `.`; los espacios son solo para leer. Pon el cencerro en un contratiempo o en una clave, no en cada paso, para que se oiga como un color propio.

Está terminado cuando `every_line_is_whole_bars_of_known_characters` y `names_are_unique_and_fills_exist` pasan, la Biblioteca muestra el ritmo con su frase y, con la banda tocándolo, el fill llega en el último compás de cada ocho.

Practica: datos como texto, tests que leen la tabla entera.

3. Una tercera chacarera (lineup-core, fácil)

La biblioteca tiene un ritmo `Chacarera` en 6/8 y un `Chacarera trunca` en 3/4. Lee los dos primero: un compás de cada uno tiene doce pasos. ¿En qué pasos empiezan los pulsos de un compás de 6/8, y en cuáles los de uno de 3/4? Ahora escribe un tercer patrón de bombo para la misma danza, con el parche (`LowTom`) en los pulsos del 6/8 y el aro (`Rim`) en los del 3/4, el dos contra tres del género, y un estilo que lo use. Comprueba en el rack que los colores de los pulsos vienen en dos grupos de seis.

Practica: leer los datos que ya existen antes de añadir, compases con el mismo número de pasos.

4. Cerca del tempo de la canción (la aplicación, fácil)

En la Biblioteca, en una canción a 60 BPM, marca **Cerca de 60 BPM**. ¿Qué tempos conserva? Después en una canción a 180. `NEAR_SPEED` vale 1,15: calcula los márgenes antes de mirar, y di por qué un margen fijo de ±15 BPM habría sido peor (ver La pestaña Biblioteca).

Practica: proporciones.

5. El mismo género (lineup-app, medio)

Añade una cuarta casilla, **Mismo género que el estilo de la canción**, que conserve los preajustes del género que toca la banda. ¿Qué tiene que saber `Narrowing`, de dónde lo sacas, y qué debería hacer la casilla en una canción sin banda?

Practica: añadir un filtro sin tocar los demás.

6. Piénsalo: closures que se ejecutan una vez

`write_pattern_style` recibe `write: impl FnOnce(&mut Project, usize)`. Los dos closures que se le pasan hoy solo leen lo que capturan. ¿Funcionaría `impl Fn` con ellos? ¿Qué podría hacer un closure futuro que `Fn` rechazaría, y por qué pedir `FnOnce` es lo más amable para quien escriba el siguiente?

La hoja de acordes

7. Cuántos compases caben (lápiz, fácil)

`Columns::count` (ver Por qué 1, 2, 4, 8, 12 o 16) recibe el ancho de la hoja y el ancho que necesita el compás más ancho. Calcula a mano cuántos compases caben en una línea de una hoja de 900 píxeles cuando el compás más ancho necesita 100, 120 y 250 píxeles. Después añade esos tres casos al test que está junto a `count`, y ejecútalo.

Practica: leer de cerca una función corta.

8. Insertar varios compases (lineup-app, fácil)

Insertar un compás antes e **Insertar un compás después** llaman a `project.insert_bars(bar, after, 1)`. Añade **Insertar compases…**, que pregunte cuántos. ¿Qué línea de `open_bars` ya hace sitio para más de uno, y qué impide que una canción crezca más allá de su máximo?

Practica: usar lo que el núcleo ya ofrece.

9. Un compás dentro de una repetición (lineup-core, medio)

Escribe un test en `bars.rs`: un clip repetido tres veces, y un compás insertado en medio de su segunda vuelta. Escribe primero qué clips, y en qué pasos, esperas después; luego ejecútalo. Los tests `a_clip_of_a_pattern_played_elsewhere_too_is_cut_in_two` y `a_repeated_clip_loses_the_repeats_in_the_deleted_bars` muestran cómo preparar uno.

Practica: escribir lo esperado antes de que el código te lo diga.

10. Piénsalo: antes o después

`close_bars` lee el compás, el tempo y la tonalidad al final de los compases eliminados *antes* de mover nada (ver Qué se lleva un compás eliminado). ¿Qué leería si preguntara después, y cómo sonaría la canción?

11. Dos compases otra vez (lineup-core, medio)

`%` vuelve a tocar el compás anterior (`repeat_bar`). Los cifrados tienen también un signo para los dos compases anteriores, un `%` con dos barras, dibujado sobre la barra de compás de los dos compases que ocupa. Haz que `%%` escrito en un compás repita los dos compases anteriores, en ese compás y en el siguiente, que tiene que estar vacío. `read_typed_bar_after` lee un compás y recibe un compás anterior: ¿qué hay que darle ahora, y cómo se entera la ventana de la hoja de que escribir en un compás escribió dos? ¿Qué debería escribir `text::write` para dos compases que repiten los dos anteriores?

Practica: una notación que abarca más que el lugar donde se escribe.

12. Un título en la canción (lineup-core y lineup-app, medio)

`text::read` lee una línea `Title:` en `TextSheet::title` , pero la canción no la guarda: **Exportar hoja de acordes…** escribe como título el nombre del archivo de la canción. Guarda el título en el proyecto y escríbelo de vuelta. Es un campo nuevo del archivo: ¿qué lee un archivo antiguo?

Practica: un campo nuevo desde el formato de texto hasta el archivo de proyecto.

13. Pegar en la tonalidad de la canción (lineup-app, medio)

Una partitura pegada desde un mensaje conserva los nombres de sus acordes, aunque su línea `Key:` diga otra tonalidad que la de la canción. Añade **Pegar en la tonalidad de la canción**, que mueva los acordes pegados de la tonalidad de la partitura a la de la canción. `HarmonyName::moved(from, to)` ya lo hace para una armonía, con los nombres correctos. ¿Qué necesita saber el pegado que hoy no guarda?

Practica: reutilizar una función de teoría en una acción de edición.

14. Compases por línea en el texto (lineup-core, fácil)

`text::write` pone como mucho cuatro compases por línea, `BARS_PER_LINE`. Conviértelo en un parámetro. ¿Qué parte de la alineación en columnas supone cuatro, y cómo tiene que cambiar `widths`?

Practica: convertir una constante en un parámetro, arrays y vectores.

15. Piénsalo: otros formatos de partitura

- ChordPro, un formato habitual en cancioneros, escribe los acordes dentro de la letra: `[C]Twinkle twinkle [F]little [C]star`. No tiene barras de compás. ¿Qué tendría que adivinar un lector para hacer un `TextSheet` con él?
- Algunos músicos quieren el número romano grande y el nombre del acorde pequeño debajo. `HarmonyName::caption` da el número romano de un acorde. ¿Qué debería decir la línea grande para un conjunto, que no tiene número romano?

Cómo toca la banda

16. Líneas de bajo a la vista (la aplicación, fácil)

Escribe `C | Am | F | G7 | C` y elige **Vals de jazz**. Mira el bajo en el piano roll: ¿qué nota toca en el tiempo 3 de cada compás, y a qué distancia está, por su nombre, de la fundamental del compás siguiente? Después elige **Swing**, en 4/4: ¿qué pone ahora el walking en los tiempos 2 y 3? Comprueba tus respuestas con `walk` en El bajo.

Practica: leer una línea de bajo.

17. Dónde cambia el acorde (la aplicación, fácil)

Escribe un compás `Dm7 G7` y un compás `C / / G7` en **Ritmo pop**. ¿En qué tiempo empieza G7 en cada uno? Mira la parte del piano: ¿cambia su ritmo en esos compases, o solo sus notas?

18. Bajo de Alberti (la aplicación, fácil)

Elige **Bajo de Alberti** sobre `C | Am | F | G7`. El patrón es «grave, aguda, media, aguda». ¿Qué nota es la grave en el compás de Am, y por qué no es La? (Pista: mira los cuatro compases, y Las armonías, como notas.)

19. Una parte que cambiaste (la aplicación, fácil)

En una canción nueva, cambia una nota del bajo y después escribe un acorde nuevo en el compás 2. ¿Qué partes siguen al acorde nuevo, y cuál no? ¿Qué dice el aviso? Pulsa **Volver a tocarla**: ¿qué pasa con tu nota?

20. Una banda sobre una demostración (la aplicación, fácil)

Abre el **Canon en Re** y pulsa **Balada pop** en la Biblioteca. Mira el rack de canales: ¿qué canales quedan silenciados, y cuáles siguen tocando? Después elige **Sin banda** en el menú de la banda. ¿Vuelve la demostración a estar exactamente como estaba?

21. Piénsalo: lo que deja fuera una huella

- `part_print` deja fuera la pista en la que está un clip y el número de su patrón. ¿Qué vería el usuario si no lo hiciera?
- `band_styles()` devuelve `&'static [BandStyle]`. ¿Por qué `Band` puede guardar un `&'static BandStyle` sin parámetro de lifetime, y no podría guardar un `&BandStyle` prestado de un `Vec` local?

Una banda más grande, vientos, y seguir tus acordes

22. Cuatro guitarristas (la aplicación, fácil)

Haz una hoja de acordes con `C Am F G` y prueba **Ritmo pop**, **Folk**, **Ritmo de rock** y **Swing**. Pulsa **Piano**, **Bajo** y **Batería** en la fila **Tocas tú**, para que solo quede la guitarra. ¿Cómo toca cada guitarrista los mismos cuatro acordes?

23. Posiciones de acordes (lápiz y piano roll, fácil)

En **Ritmo pop**, lee en el piano roll las notas de la guitarra para G y para F, y escribe sus posiciones como lo hacen los libros de acordes (G es `320003`; ver La música: cómo toca un acorde un guitarrista). ¿Por qué la posición de F no tiene cuerdas al aire? ¿Qué posición tocarías para E7, y qué notas del acorde duplica? Comprueba tus respuestas con `open_chords_are_the_ones_in_chord_books` y `other_keys_take_barre_chords`.

24. Acordes completos con distorsión (lineup-core, fácil)

En `band/extras.rs`, cambia la guitarra de Ritmo de rock de `GuitarPart::Power(Comping::Pulse)` a `GuitarPart::Strum(Comping::Pulse)`: posiciones completas por la guitarra saturada. Escúchalo con una SoundFont. ¿Por qué lo evitan los guitarristas de rock? Vuelve a dejarlo como estaba.

25. Una guitarra para Lo-fi (lineup-core, fácil)

Lo-fi no tiene músicos extra. Dale una guitarra de cuerdas de nailon que arpegie los acordes, `GuitarPart::Strum(Comping::ArpUp)`, en `BY_STYLE` de `extras.rs`. Después escribe mal el nombre del estilo: ¿qué test falla, y qué imprime?

Practica: tablas de constantes, tests que vigilan nombres.

26. Piénsalo: la guitarra

- Sobre un acorde mayor, el boogie shuffle se mece entre la quinta y la sexta; sobre un acorde menor se queda en la quinta (ver `the_boogie_alternates_fifth_and_sixth`). ¿Qué acorde forman La, Do, Mi y Fa, y por qué sería arriesgada una sexta menor sobre La menor? ¿En qué convertiría el acorde una sexta mayor, Fa♯?
- `try_frets` recibe `found: &mut impl FnMut(&[u8])`. Prueba con `found: impl FnMut(&[u8])`. ¿Qué dice el compilador sobre la llamada recursiva, y por qué?

27. Un segundo teclado (lineup-core, más difícil)

Añade un piano eléctrico (programa 4) que toque el acompañamiento del estilo una octava más arriba en los estilos **R&B** y **Neo soul**. ¿Qué partes de `band.rs` y de

`file.rs` tienen que cambiar (mira `written` y `written_too` en En el archivo)? Comprueba que una canción cuya banda tiene solo tres músicos se guarda con el mismo número de formato que antes.

28. Vientos sostenidos y golpeados (la aplicación, fácil)

Marca **Sección de vientos** en **Swing** y en **Balada pop**, sobre los mismos acordes. Compara las notas de los vientos en el piano roll: ¿qué es igual, y qué cambia? ¿Por qué la conducción suave de las voces importa todavía más en notas largas que en notas cortas?

29. Notas comunes (la aplicación, fácil)

Escribe `C Am F G` en **Ritmo pop** con los vientos, y anota la disposición de los vientos para cada acorde, desde el piano roll. ¿Qué viento mantiene su nota de un acorde al siguiente, y durante cuántos acordes? Una nota que comparten dos acordes es una **nota común**: ¿por qué no se mueve el viento que la toca?

30. Quintas baratas (lineup-core, fácil)

`cost` cobra `PARALLEL_FIFTHS`, 24 semitonos, por cada par de vientos que se mueve en quintas paralelas. Ponlo en 2 y ejecuta `cargo test -p lineup-core horns`. ¿Qué test falla, entre qué dos acordes, y qué vientos se mueven en quintas? Vuelve a dejarlo como estaba.

31. Piénsalo: nombres en dos tablas

`HELD` y `BY_STYLE`, en `horns.rs`, nombran estilos por su nombre. ¿Qué pasa si se cambia el nombre de un estilo en la biblioteca? Encuentra el test que lo detectaría.

32. Un punto de división (lineup-app, medio)

Los teclados con acompañamiento suelen tener un **punto de división**: solo las teclas por debajo de él eligen el acorde, para que la mano derecha pueda tocar una melodía. Añade uno, por ejemplo por debajo de Do4, como segunda opción junto a **El acompañamiento sigue los acordes**. ¿Qué teclas llegan ahora a `chord_held`, y qué sigue pasando con las de arriba?

33. Dos teclas a una segunda mayor (lineup-core, fácil)

`chord_held` lee dos teclas por el intervalo que hay entre ellas (ver Seguir los acordes que mantienes). Haz que dos teclas a una segunda mayor sean un acorde sus2 sobre la más grave. ¿Qué test necesita un caso nuevo? Antes de tu cambio, mantener Fa debajo de Sol no pedía ningún acorde; ¿qué pide ahora, y es eso lo que querría decir un músico que las mantiene?

34. Piénsalo: acordes al instante

Las notas que ya suenan conservan su altura hasta que terminan, así que un acorde mantenido todo un compás solo cambia en el compás siguiente. Los teclados con acompañamiento tocan el acorde nuevo al instante. ¿Qué tendría que saber el hilo de audio para hacerlo, y cuánto costaría?

La orquesta

35. Palmas con el bombo (la aplicación, fácil)

Aplica la Orquesta a **La consagración de la primavera, Augurios primaverales** y silencia la pista **Canción**. Da palmas con el bombo. ¿Puedes predecir dónde llega el golpe siguiente? Vuelve a activar la canción y escucha los acentos de los vientos cayendo en las mismas corcheas.

36. Pulsos sentidos a mano (lápiz, fácil)

`TimeSignature::felt_groups` da 13/16 como 2 + 2 + 2 + 2 + 2 + 3. ¿En qué semicorcheas golpean los vientos en un compás de 13/16 en el que se atienen a los pulsos sentidos? Abre **Petrushka, La habitación del moro**, aplica la Orquesta, busca un compás de 13/16 en la regla y compruébalo. Si ahí tocan otra cosa, ¿qué regla de Tocar junto a la partitura lo explica?

37. `in_meter` para el compás del tresillo (lineup-core, fácil)

Calcula a mano la línea que escribe `in_meter` para un compás de 3+3+2/8 a partir del `X... .... o... ....` de la Orquesta. Después añádela a

`a_rhythm_that_follows_the_meter_plays_each_bar_in_its_own` en `rhythm.rs`, y ejecútalo.

38. Un estilo en 4/4 sobre compases que cambian (la aplicación, fácil)

Con la Orquesta tocando el **Juego del rapto**, pulsa **Ritmo pop** en la Biblioteca. ¿Qué le pasa a la regla, y por qué un estilo en 4/4 tiene que hacerlo, cuando la Orquesta no lo hizo?

39. Solo las voces (la aplicación, fácil)

Abre **Shchedryk (Villancico de las campanas)**, aplica la Orquesta con solo **Voces / instrumentos** marcado, y abre el **Mezclador**. ¿Qué instrumentos tocan los canales de la pista Canción? ¿Cuáles tocan las pistas nuevas Piano y Bajo?

40. Acentos que nadie escribió (lineup-core, fácil)

En `band/following.rs`, cambia `TUTTI` de `(9, 5)` a `(3, 2)` y ejecuta `cargo test -p lineup-core accents_of_the_augurs`. ¿Aparecen acentos que Stravinsky no escribió? Vuelve a dejarlo como estaba.

Forma, variaciones, secciones y bloqueos

41. La introducción lleva a algún sitio (la aplicación, fácil)

Escribe `| Dm7 | G7 | Cmaj7 | Cmaj7 |`, un ii–V–I en Do, y dale una introducción de 2 compases. ¿Qué acordes escribe la introducción? No llevan a C: ¿a dónde llevan, y por qué?

42. El último acorde (la aplicación, fácil)

Pon la tonalidad en La menor (el menú **Tonalidad** junto a las teclas de armonía), escribe una canción que empiece en Am y activa el final. ¿En qué acorde termina? Ahora cambia el primer acorde a F: ¿en qué acorde termina ahora, y por qué sigue siendo correcto?

43. Cuánto dura la canción (la aplicación, fácil)

Exporta como WAV una canción con una introducción de 4 compases, tres vueltas y un final. ¿Cuántos compases dura? Comprueba tu respuesta con la duración del archivo y el tempo.

44. La forma se mueve (la aplicación, fácil)

- Dale a una canción una introducción de 8 compases, cambia un acorde de ella, y después cambia la introducción a 2 compases. ¿Qué pasa con tu acorde? Deshazlo.
- Escribe un acorde en el primer compás del final. ¿Dónde está ahora el final, y dónde el signo de fin de repetición?

45. Piénsalo: la forma en Rust

- `Form` es `Copy`; `Band` no. Mira sus campos: ¿por qué no podría serlo `Band`?
- En `played_pieces`, la vuelta solo se repite `if from <= start && end <= to`. ¿Qué tocaría sin esa comprobación una región de bucle que va del medio de la introducción al medio de la vuelta?

46. A dos, y después caminando (la aplicación, fácil)

Elige el estilo **Balada de jazz**, escribe `| Cmaj7 | Am7 | Dm7 | G7 |` dos veces, y pon los compases 5 a 8 en la variación b. Abre los patrones del bajo en el piano roll. ¿Cuántas notas tiene un compás en a, y en b? ¿Qué notas toca el bajo en el compás de Dm7 de cada una?

47. Fills y crashes (la aplicación, fácil)

- En un **Ritmo pop** de ocho compases, pon el compás 5 en la variación b y el 6 otra vez en a. ¿Qué compases tienen un fill de batería, y dónde caen los crashes? ¿Por qué podría sonar inquieto en una canción de verdad?
- Elige **Swing** y toca unos compases en cada variación. ¿Qué cambia y qué se queda? ¿Por qué el tiempo sigue en el mismo platillo?

48. Un estilo para algunos compases (la aplicación, fácil)

- Escribe `| C | Am | F | G |` dos veces en **Ritmo pop**, y haz que los compases 5 a 8 sean **Bossa nova**. ¿Dónde empieza la línea de bajo de la bossa nova, y en qué se diferencia de la del pop?
- Toca una sección en **Coral**, que no tiene batería. ¿Qué toca la batería ahí, y en el compás de antes?

49. Bloqueos (la aplicación, fácil)

- Abre **Oda a la alegría**, bloquea **Teclados** en el rack y aplica **Doo-wop**. ¿Qué pistas cambiaron, y qué conservó Teclados?
- En una canción nueva, bloquea el bombo y aplica **Bossa nova**. ¿Qué toca ahora el bombo?

50. Piénsalo: marcas y préstamos

- `tidy`, para las marcas de variación, ordena, después llama a `dedup_by` y después a `retain`. ¿Qué saldría mal si llamara a `retain` antes de ordenar?
- `played_in_runs` recibe `play: impl Fn(...)`. ¿Podría recibir un `FnMut`? ¿Qué cambiaría donde se la llama?
- `where_it_plays` devuelve `impl Iterator`. ¿Qué cambiaría si devolviera un `Vec`, y por qué `notes.extend(...)` acepta las dos cosas?
- `StylePattern` guarda un `&'a mut Pattern`. ¿Qué dice el borrow checker si llamas a otro método de `Project` mientras uno está vivo, y por qué eso es bueno?

Las paradas y el solista

51. Stop-time (la aplicación, fácil)

Escribe un blues de doce compases en Fa con golpes en los cuatro primeros compases: `| F7. | F7. | F7. | F7. | Bb7 | Bb7 | F7 | F7 | C7 | Bb7 | F7 | C7 |`, y elige **Shuffle**. Toca en los huecos con la escala de blues de Fa (Fa La♭ Si♭ Si Do Mi♭). ¿Qué cambia en la sensación cuando la banda vuelve en el compás 5?

52. Anticipaciones (la aplicación, fácil)

Escribe `| C | Am | F | G |` en **Ritmo pop** y escúchalo. Después anticipa todos los acordes menos el primero: `| C | ^Am | ^F | ^G |`. Da palmas con el ritmo del piano de las dos maneras. ¿Qué tiempo ya no se golpea? Después compara el piano y el bajo en el piano roll: ¿cuánto mantiene cada uno un acorde anticipado?

53. Un sostenido de dos compases (lineup-core, medio)

Escribe `C..` en el compás 1, nada en el compás 2 y `G7` en el compás 3, para que el sostenido dure dos compases, y mira el bajo: una sola nota. Cada músico golpea a su manera explica por qué el bajo usa `bass_struck` y no `Bass::Held`. Escribe un test en `band.rs` que fallaría si usara `Bass::Held`.

54. Un golpe de una negra (lineup-core, fácil)

Un golpe dura una corchea: `SHOT_STEPS`, en `band/stops.rs`, vale 2. Haz que dure una negra y ejecuta `cargo test -p lineup-core`. Fallan tests no solo en `stops.rs` (`a_shot_strikes_once_and_the_band_stops`) sino también en las partituras y las demostraciones de la biblioteca: ¿por qué les importa a esos cuánto dura un golpe? Corrige lo que esperan, o decide que el cambio estaba mal.

55. Una anticipación de una negra (lineup-core, medio)

Añade `^^^C`, una anticipación de una negra: una variante de `Push`. Deja que el compilador te muestre cada `match` que tiene que aprenderla (`steps`, `marks`, `describe`). ¿Qué otros sitios la necesitan? (Pista: `read_marks` rechaza hoy tres acentos circunflejos, y `breaks_menu`, en la aplicación, enumera las anticipaciones.)

56. Piénsalo: un fill antes de una parada

La batería toca un fill en el compás anterior a un cambio. ¿Debería conservarlo un golpe o un silencio en ese compás? Busca dónde añade el fill `drum_part` y dónde quita el groove `apply_to_drums`, decide, y escribe un test para lo que elegiste.

57. Notas guía y escalas de acorde (la aplicación, fácil)

- Con el solista activado, abre su parte en el piano roll y busca las notas marcadas `3` y `7`. ¿Dónde pasa una 7.ª a una 3.ª medio tono más abajo en un cambio de acorde? Canta solo las 3.ª y las 7.ª, una por compás, sobre la banda.
- Escribe `| Am7 | D7 | Gmaj7 | E7 | Am7 | D7 | G6 | G6 |` en Sol mayor. Calcula la escala de cada acorde con las reglas de Escalas de acorde antes de mirar. Un acorde toca un Fa natural, aunque la tonalidad tiene Fa♯: ¿cuál, y qué otra nota de su escala no está en Sol mayor?
- En La menor, escribe `| Bm7b5 | E7 | Am | Am |`. ¿Qué nota de la escala de E7 no está en La menor natural? Búscala con el oído en el solo.

58. Saltos y rodeos (lineup-core, medio)

- El solista salta a la segunda nota del acorde una vez de cada cinco (`dice.chance(20)` en `target`). Ponlo en 60 y toca unos cuantos solos con **Otro solo**. ¿Cómo cambia la línea? ¿Qué tests siguen pasando, y por qué?
- Añade un **rodeo** (*enclosure*), una figura favorita del bebop: antes de una nota objetivo, una nota de la escala por encima y luego la nota medio tono por debajo, como Re Si Do hacia Do. Dale un `NoteRole` nuevo, deja que el compilador te muestre cada `match` que tiene que aprenderlo, y escribe un test al estilo de `roles_name_what_each_note_does`.

59. Un solo nuevo en cada vuelta (lineup-core, más difícil)

Con **Vueltas** en 3, el solista toca el mismo solo tres veces. Haz que cada vuelta sea un solo nuevo. ¿Dónde se repite la vuelta (busca `repeated_chorus`), y cómo podría cada vuelta tener su propia semilla mientras la canción sigue sonando igual cada vez que se abre?

Tus propios estilos y las herramientas de práctica

60. Lee los archivos (la aplicación y un editor de texto, fácil)

Guarda un estilo desde el editor de estilos (ver Probarlo) y abre el archivo `.lineup-style` en un editor de texto. Busca su `format`, `bars`, `chart` y los `lanes` con cada canal y sus notas. Después aplícalo a una canción, guarda la canción y busca

`custom_rhythm` en su archivo: el estilo entero está dentro. ¿Por qué tiene que estarlo, y qué número de formato tiene como mínimo la canción?

61. Dominantes fuera de la tonalidad (lineup-core, medio)

Sobre un acorde fuera de la tonalidad, `adapt_pitch`, en `user_style.rs`, conserva las distancias escritas sobre Do, así que el E7 que lleva a Am toca Mi Fa♯ Sol♯ La Si Do♯ Re (ver Acordes de fuera de la tonalidad). Las escalas de acorde del solista, en `harmony/chord_scale.rs`, le darían el Fa y el Do de La menor. Úsalas para los acordes de dominante fuera de la tonalidad. ¿Qué notas escritas de un estilo sobre E7 en Do mayor suenan ahora distinto? Escribe primero el test, al estilo de `template_notes_stay_in_the_key`, y escucha: ¿vale más un patrón que suena igual sobre cada acorde, o las notas correctas?

62. Cuántos compases (lápiz, fácil)

Un estilo de usuario tiene como mucho 256 pasos y como mucho dieciséis compases. ¿Cuántos compases puede tener un estilo en 7/8, y uno en 6/4?

63. Lo que cuenta el metrónomo (la aplicación, fácil)

- Activa el metrónomo sobre **Noche de paz**, en 6/8. ¿Cuántos clics hay en un compás, y dónde cae el agudo? Ahora **Greensleeves**, en 3/4: las mismas seis corcheas por compás, contadas de otra manera.
- Con la cuenta de entrada activada, pausa una canción en medio de un compás y vuelve a pulsar Reproducir. ¿Qué oyes antes de la música, y por qué es mejor que contar un compás entero y empezar en medio del siguiente?
- Pon la velocidad al 50% y exporta un archivo WAV. ¿Cuánto dura, comparado con la canción a velocidad normal? ¿Por qué?

64. Marcar el tempo (la aplicación, fácil)

- Marca 4 veces a un ritmo regular, y después 4 veces más deprisa sin parar. ¿Qué tempo obtienes, y por qué no el promedio de los ocho intervalos?
- En una canción en 6/8, marca 80 veces por minuto. ¿Qué tempo recibe la canción, y por qué?
- Encárgate del bajo con **Tocas tú**, cambia un acorde y devuelve el bajo: ¿está el acorde nuevo en su parte?

65. Piénsalo: el clic y la cuenta de entrada

- `ClickVoice::strike(Click::None)` vuelve enseguida. ¿Qué oirías si pusiera `level` a 0?
- En `begin_play`, ¿qué sale mal si no se llama a `self.stop()` cuando llega la orden de reproducir mientras la canción ya suena?

66. El entrenador de velocidad a mano (la aplicación, fácil)

- Pon **desde 50% hasta 100% por 10% cada 2 pasadas**. Anota la velocidad de cada una de las ocho primeras vueltas, y después tócalo con la traza activada y compruébalo.
- Pon **desde 100% hasta 60%**. ¿Qué hace ahora el entrenador? Piensa en un uso (pista: un pasaje que ya puedes tocar rápido, pero todavía no parejo).
- Empieza una tanda, pulsa **Pausa** en medio y luego Reproducir. Después Detener y Reproducir. ¿Qué velocidad hay cada vez, y por qué es útil que los dos botones sean distintos?
- Con el entrenador activado y nada en bucle, toca una canción entera. ¿Cuándo sube la velocidad?

67. Piénsalo: los mensajes del entrenador

- En `came_round`, `self.laps_seen.replace(laps)` se ejecuta antes de comprobar `self.training`. ¿Qué saldría mal al activar el entrenador en medio de una canción si se ejecutara después?
- El hilo de audio tira con `throw_away` una lista vieja de tempos por vuelta cuando una petición nueva la sustituye. ¿Qué pasaría si simplemente asignara la lista nueva, y por qué importa eso justamente en el hilo de audio?

68. Más para el oído (lineup-core, medio)

- **Una melodía para la séptima mayor.** `Interval::tune` no tiene ninguna hacia arriba. Busca una canción cuya melodía empiece con una, compruébala en un piano y añádela. ¿Tiene que cambiar `going_up_names_a_tune`?
- **Melodías hacia abajo.** Añade un `tune_down` (el comienzo de la Quinta sinfonía de Beethoven es una tercera mayor hacia abajo) y úsalo cuando la pregunta fue hacia abajo.
- **Lo que fallas.** Lleva la cuenta, para cada intervalo, de cuántas veces se preguntó y cuántas acertaste, y pregunta más a menudo los débiles. ¿Cómo usaría

`IntervalQuestion::new` esas cuentas sin perder la regla «misma semilla, misma pregunta»?

69. Piénsalo: el entrenamiento auditivo

- Los acordes de sexta quedan fuera porque C6 y Am7 tienen las mismas notas. ¿Podrían preguntarse con justicia si el acorde sonara siempre sobre su fundamental una octava más abajo? ¿Seguirían teniendo sentido las inversiones?
- Haz que el ejercicio de progresiones toque la propia hoja de acordes de la canción, con los nombres ocultos. ¿Qué necesitaría de `chord_sheet.rs`?

La partitura guía y las partes

70. Decisiones de notación (lineup-core, medio)

- **Alteraciones de cortesía.** Cuando un compás tiene un Fa♯ y el siguiente un Fa natural por la armadura, muchas partituras guía imprimen un becuadro entre paréntesis como recordatorio. ¿Dónde tendría que saber `BarAccidentals` del compás anterior?
- **Semidisminuido.** Algunas partituras guía escriben Em7♭5 como Eø7. Añade una opción para ello. ¿Es una decisión de `layout.rs`, o del nombre del acorde?

71. Piénsalo: la página

- **Casillas de repetición.** Cuando el último compás de una sección repetida cambia la segunda vez, las partituras guía usan primera y segunda casilla: un corchete «1.» sobre un compás, «2.» sobre el otro. ¿Qué tendría que contarle la lista de reproducción a `score.rs`?
- **Compases de silencio.** Una melodía que calla ocho compases podría mostrar un solo compás con una línea gruesa y un «8» encima. ¿Dónde iría en `systems`, y cómo cambia los números de compás de la línea siguiente?

72. Más instrumentos (lineup-core, medio)

- **La octava exacta.** Añade una opción que escriba cada parte exactamente como la toca el instrumento: una novena arriba para el saxo tenor, una octava y una

sexta para el barítono. ¿Qué función cambia? ¿Cuántas líneas adicionales necesita una melodía alrededor de Do5 para el tenor, y por qué los libros de partituras guía no lo hacen?

- **Instrumentos en Re y en Sol.** Una trompeta en Re suena un tono más aguda de lo escrito, una flauta contralto en Sol una cuarta más grave. Añádelos. ¿Qué devuelven `note()` y `semitones()`, y qué más de la ventana y de la etiqueta tiene que saber de ellos?
- **El clarinete en Mi♭** suena una tercera menor *más aguda* de lo que lee. ¿Es una `Transposition` nueva, o la parte del saxo alto en otra octava? ¿Qué tendría que saber `shift` para distinguirlas?

73. Partes como texto, y la tonalidad real (lineup-core, fácil)

- Dale a **Exportar hoja de acordes…** la misma elección de instrumentos. ¿Qué función, ella sola, hace el trabajo para cada acorde? (Ver Los acordes: movidos con nombre y todo.)
- Algunas partes imprimen la tonalidad propia de la canción debajo del instrumento, «Tonalidad real: Si♭». ¿Dónde iría en `layout.rs`, y qué necesita saber `lay_out` que ahora no sabe?

Cómo se escribe una demostración

Archivos: `crates/lineup-core/src/library/song.rs`, `crates/lineup-core/src/library/song/`, `crates/lineup-core/src/library/chart.rs`, `crates/lineup-core/src/library/demo.rs`, `crates/lineup-core/src/library/interpret.rs` y `crates/lineup-app/src/app/demo_menu.rs`

Archivo > Demostraciones abre 343 canciones, de canciones infantiles a La consagración de la primavera, y una demostración corta de cada estilo, de cada estilo dodecafónico y de cada estilo de micromodos. Haces clic en una y se abre, con sus acordes en la hoja de acordes, y suena. Qué escuchar en ellas está en la guía de escucha del manual. Este capítulo trata de qué *es* una demostración.

Una demostración de canción no es un archivo. Es una constante de Rust, compilada dentro del programa: la melodía y los acordes escritos como texto, como los escribe una partitura guía, y unos pocos campos que dicen cómo se toca la canción. `song_demo` convierte la constante en un `Project` corriente cada vez que la abres. Eso tiene tres ventajas:

- **Los tests leen todas las canciones.** Un compás con una semicorchea de más, un acorde que no se puede leer, una forma que nombra una parte que no existe: cada cosa hace fallar `cargo test` antes de que nadie la oiga.
- **Una demostración es igual en todas partes.** No puede faltar, estar dañada ni quedar vieja; cada versión nueva de la aplicación trae su propia copia.
- **Lo que necesitan las canciones, lo aprende la biblioteca.** Una pieza que cambia de tonalidad, de compás o de tempo, una ronda, una orquesta que pasa su melodía de la flauta al clarinete: cada una se convirtió en una función de `Part`, y cualquier canción puede usarla.

Por lo mismo, una demostración es de solo lectura: no hay archivo donde escribir tus cambios, así que **Guardar** pide uno nuevo (ver La barra de título).

Una canción es una constante

Esta es una canción entera, la ronda Frère Jacques:

```
pub(super) const FRERE_JACQUES: Song = Song {
    name: "Frère Jacques",
    origin: "French round, 18th century",
    feel: "Round",
    genre: Genre::Folk,
    tonic: C,
    mode: Major,
    time: COMMON,
    tempo: 104.0,
    comping: Comping::ArpUpDown,
    bass: Bass::RootFifth,
    rhythm: "Folk",
    band: SONG_AND_GUITAR,
    style: "Folk",
    parts: &[
        Part::new("Alone", FRERE_JACQUES_TUNE, FRERE_JACQUES_CHORDS),
        Part::new("Round", FRERE_JACQUES_TUNE,
FRERE_JACQUES_CHORDS).with_voices(&[
            (1, 0),
            (2, 2),
            (3, 4),
            (4, 6),
        ]),
        // ...
    ],
    form: &["Alone", "Round", "Round again", "Last voices"],
    drums_from: 0,
    about: "A round: four voices sing the same tune, each two bars after
the last. It works \
            because every two bars of the tune fit the same chord.",
};
```

La melodía y los acordes son constantes aparte, encima, porque las usan cuatro partes:

```
const FRERE_JACQUES_TUNE: &str = "C5:4 D5 E5 C5 | C5 D5 E5 C5 | E5 F5 G5:8
| E5:4 F5 G5:8 | \
     G5:2 A5 G5 F5 E5:4 C5 | G5:2 A5 G5 F5 E5:4 C5 | C5 G4 C5:8 | C5:4 G4
C5:8";
const FRERE_JACQUES_CHORDS: &str = "I | I | I | I | I | I | I | I";
```

Los campos se leen de arriba abajo como la cabecera de una partitura guía: qué es y de dónde viene, su tonalidad, su compás y su tempo; luego cómo la viste la banda, una manera de acompañar y una línea de bajo de la biblioteca (ver El comping: cómo se tocan los acordes), un ritmo por su nombre, los instrumentos (`band`) y el estilo al que más se parece, que la Biblioteca muestra seleccionado mientras la demostración está abierta. Después, la música: las **partes** (`parts`), cada una una sección como una estrofa o un estribillo, y la **forma** (`form`), el orden en que se tocan, por nombre. Una parte que se toca tres veces se escribe una vez y se nombra tres. `drums_from` es la

entrada de la forma donde entra la batería: 0 para el principio, 1 cuando la canción abre con una anacrusa o una introducción que la batería deja tranquila.

Un archivo por género

343 canciones harían un archivo de decenas de miles de líneas. Así que `song.rs` guarda los tipos, las constantes compartidas, la lista y los tests, y las canciones viven al lado, un archivo por género:

```
library/
  song.rs          Song, Part, Band, SONGS, los tests
  song/
    balkan.rs
    blues.rs
    classical.rs
    ...
    tango_played.rs
```

`mod tango;` dentro de `song.rs` hace que Rust busque `song/tango.rs`: cada archivo de género es un **módulo hijo** de `song`. Una canción es `pub(super) const`, visible para su padre y nada más: fuera de la biblioteca nadie necesita el nombre `HABANERA`, solo la canción, que encuentra en la lista.

Cada archivo de género empieza con una línea:

```
use super::*;
```

`super` es el módulo padre y `*` trae todos los nombres que este ve, también los privados: en Rust un elemento privado de un módulo es visible para los hijos de ese módulo. Así una canción puede usar `COMMON`, `SIX_EIGHT`, las bandas y cualquier otra constante que `song.rs` se guarda para sí. Dos de los `use` del propio padre son los que más importan:

```
use PitchClass::*;
use ScaleKind::*;
```

Permiten que una canción escriba `tonic: A` y `mode: Dorian` en lugar de `PitchClass::A` y `ScaleKind::Dorian`. `Genre` no se importa así, a propósito: `Genre::Blues` y `ScaleKind::Blues` serían los dos `Blues`, y dos importaciones con `*` que traen el mismo nombre son un error en cuanto se usa ese nombre. Así que un género se escribe siempre completo, `genre: Genre::Blues`, y `mode: Blues` es la escala.

La lista

```
pub static SONGS: &[Song] = &[
    pop::AULD_LANG_SYNE,
    pop::LONDON_BRIDGE,
    pop::OLD_MACDONALD,
```

Cada canción aparece en la lista por su nombre. Rust no puede unir slices mientras compila, así que los archivos de género no pueden traer cada uno su propia lista; nombrar cada `const` es la manera sencilla. Una `const` se copia allí donde se usa, así que la lista guarda las canciones mismas, y como es una `static`, tiene un único lugar en memoria durante toda la ejecución: el menú y la Biblioteca pueden repartir referencias `&'static Song` a ella. La lista va agrupada por género, en el orden de `Genre::ALL`, y un test lo comprueba con `windows(2)`, que da cada par de vecinos:

```
SONGS.windows(2).all(|pair| pair[0].genre <= pair[1].genre),
```

`<=` funciona con géneros porque `Genre` deriva `PartialOrd`, que ordena las variantes de un enum tal como están escritas. El menú ordena las canciones por título por su cuenta (ver El menú Demostraciones), así que una canción nueva va al final del bloque de su género, y dos cambios que agregan canciones a la vez no se pelean por las mismas líneas.

De dónde salen las notas

Cada canción lleva `origin`, quién la escribió y cuándo, y encima un comentario `// Source:` que nombra la partitura, el cifrado, el grabado o la grabación de donde se tomaron sus notas, y qué se simplificó. El comentario de la Promenade, por ejemplo, nombra el grabado de LilyPond del que se leyó y explica cómo se eligió un acorde para cada pulso. Casi todas las canciones son tan viejas que son de todos; las pocas que todavía tienen derechos de autor en algún lugar lo dicen en su comentario. Veinticinco son melodías nuevas para géneros con pocas libres, escritas como esos géneros escriben las suyas, y su `origin` dice «Written for Ferrous Lineup». Cómo una interpretación o una partitura se convierte en una constante es el capítulo siguiente, De una interpretación MIDI o una partitura a una demostración.

La melodía, como texto

Una melodía se escribe en una notación pequeña, que lee `chart::parse_melody`:

Texto	Significa
`C4`, `F#5`, `Bb3`	una nota: su letra, `#` o `b`, y su octava (`C4` es el do central)
`:4`	su duración en pasos (4 es una negra); sin ella, tan larga como la nota anterior
`:1/2`, `:1/4`	más corta que un paso: una fusa, una semifusa, para adornos y escalas rápidas
`-:8`	un silencio, aquí medio compás de 4/4
`_:8`	una ligadura: la nota anterior sigue, también a través de una barra de compás
`|`	una barra de compás: no cambia nada, pero cada compás se comprueba
`A3+C#4+E4:4`	un acorde, sus notas a la vez
`&`	otra voz, desde el principio, al mismo tiempo: `E5:4 D5 C5 B4 & C3:16`
`(3`	las tres notas o silencios siguientes son un tresillo, tres en el tiempo de dos
`!p`, `!f`, `!sfz`	una dinámica, de `!pp` a `!ff`, o un sforzando en la nota siguiente
`!<`, `!>`	un crescendo o un diminuendo, hasta la dinámica siguiente
`>A4:2`, `D5:2.`	un acento delante de una nota; un punto de staccato después

Las marcas no ocupan tiempo. Las notas de un tresillo empiezan entre los pasos de la cuadrícula, así que se colocan al tick (ver Notas entre pasos). La estrofa de Greensleeves se lee así:

```
C5:8 D5:4 | E5:6 F#5:2 E5:4 | D5:8 B4:4 | G4:6 A4:2 B4:4 | ...
```

Una melodía que empieza antes de su primer compás, como el «In a» de «Clementine», tiene una **anacrusa**. Se escribe como una parte propia, un compás que es casi todo silencio:

```
Part::new("Pickup", "-:8 C5:3 C5:1", "I"),
```

Eso es un compás de 3/4: dos pulsos de silencio y luego «In a», una corchea con puntillo y una semicorchea. El primer tiempo fuerte de la melodía cae entonces en el primer tiempo de la estrofa, donde lo esperan la batería y los acordes.

Los acordes, como cifrado

Debajo de la melodía va un cifrado, compás por compás (ver Cifrados: acordes compás por compás): números romanos como `i | bIII | bVII | v`, o nombres como los escribe una partitura guía, `C#m | G#7/B#`, leídos en la tonalidad de la parte. Las marcas de la hoja de acordes también funcionan: un cifrado puede anticipar un acorde o detener la banda (ver Golpes, sostenidos, silencios y anticipaciones). El cifrado es lo que toca la banda, con el acompañamiento y la línea de bajo de la canción, y lo que muestra la hoja de acordes.

En algunas músicas el acompañamiento es parte de la pieza: la mano izquierda de un rag, un coral de Bach, un coro. Entonces la parte lo escribe, y sus acordes no se acompañan:

- `with_left(...)` escribe notas para el canal **Teclados**, en la notación de las melodías, `&` incluido;
- `with_lines(&[...])` escribe **otros instrumentos**, cada uno una `Line` con el nombre de su canal, un programa de General MIDI y sus notas. Las líneas con el mismo nombre en cualquier parte suenan en el mismo canal, así que el coro de la Misa Criolla es una sola línea, «Choir» (Coro), con su contralto, su tenor y su bajo unidos por `&`.

Una parte escrita entera puede tener cifrado igual: entonces nombra lo que toca la música, para la hoja de acordes, y la banda no lo toca.

Partes que cambian algo

`Part::new(name, melody, chords)` crea una parte sencilla. Las demás son la misma parte con un campo cambiado, por una `const fn` que usa la **sintaxis de actualización de structs**:

```
const fn in_key(self, tonic: PitchClass, mode: ScaleKind) -> Part {
    Part {
        key: Some((tonic, mode)),
        ..self
    }
}
```

`..self` quiere decir «todos los demás campos como en `self`». Como solo mueve valores de un lado a otro, puede ser `const`, y una canción sigue siendo una constante

con `.in_key(FSharp, Major)` al final de una de sus partes. Las demás funcionan igual:

Método	La parte…	Por ejemplo
`in_key(tonic, mode)`	está en otra tonalidad: una modulación	el rey de la montaña sube a fa♯ mayor
`in_time(time)`	está en otro compás	un estribillo en 3/4 en una canción en 2/4
`in_meters(&[...])`	tiene un compás para cada compás	los 5/4 y 6/4 de la Promenade
`with_tempo(&[...])`	tiene las indicaciones de tempo de la partitura	«Più vivo», un ritardando, un calderón
`played(comping, bass, rhythm)`	se acompaña a su manera	el estribillo en 3/4 necesita el acompañamiento de un vals
`with_voices(&[...])`	la cantan varias voces	una ronda
`played_by(&[...])`	pasa su melodía a otros instrumentos	Boléro, de la flauta al clarinete
`marked(dynamic)`	se toca así de fuerte	el comienzo pianissimo de Boléro
`with_drums(&[...])`	tiene la batería escrita	la caja de Boléro
`with_left`, `with_lines`	escribe su acompañamiento	la mano izquierda de un rag, un coro

En la gruta del rey de la montaña usa dos:

```
Part::new("Higher", MOUNTAIN_HIGHER,
MOUNTAIN_HIGHER_CHORDS).in_key(FSharp, Major),
// "Più vivo": faster at once, at Grieg's own metronome mark.
Part::new("Più vivo", MOUNTAIN_THEME, MOUNTAIN_CHORDS).with_tempo(&
[tempo(0, 154)]),
```

Una tonalidad propia

Una parte en otra tonalidad lee sus números romanos en esa tonalidad, así que `I` en la parte en fa♯ mayor es un acorde de fa♯, y la demostración pone una marca de

tonalidad donde empieza y otra donde la música vuelve. La canción responde en qué tonalidad está una parte:

```
pub fn part_scale(&self, part: &Part) -> Scale {
    match part.key {
        Some((tonic, mode)) => written_as_tune(Scale::new(tonic, mode),
[part]),
        None => self.scale(),
    }
}
```

`written_as_tune` decide el *nombre* de la tonalidad. Un `PitchClass` tiene una variante para cada una de las doce notas, con sostenidos (`ASharp`, nunca un si bemol), pero la Promenade está en si♭ mayor, no en la♯ mayor, y Take Five en mi♭ menor, no en re♯ menor. En lugar de hacer que cada canción deletree su tonalidad, la biblioteca cuenta las alteraciones de la melodía y nombra la tonalidad como está escrita la melodía:

```
fn written_as_tune<'a>(key: Scale, parts: impl IntoIterator<Item = &'a
Part>) -> Scale {
    let (sharps, flats) = parts
        .into_iter()
        .map(|part| accidentals(part.melody))
        .fold((0, 0), |(sharps, flats), (s, f)| (sharps + s, flats + f));
    match sharps.cmp(&flats) {
        Ordering::Greater => key.with_spelling(Spelling::Sharps),
        Ordering::Less => key.with_spelling(Spelling::Flats),
        Ordering::Equal => key,
    }
}
```

`fold` lleva un par a través de las partes y suma las dos cuentas a la vez, y `cmp` devuelve un `Ordering`, que el `match` lee como «más sostenidos», «más bemoles» o «igual». Una melodía con tantos de unos como de otros, o sin ninguno, deja a la tonalidad su nombre habitual. Cómo se escribe una tonalidad está en Sostenidos o bemoles: cómo se escribe una tonalidad.

La música que no está en ninguna tonalidad, donde las doce notas están en casa, lo dice con la escala cromática: el Juego del rapto de principio a fin, con `mode: Chromatic`, y partes de Petrushka con `.in_key(C, Chromatic)`. Una parte así no tiene armadura, y cada nota que toca cuenta como dentro de su tonalidad.

Un compás para cada compás

La mayoría de las partes está en el compás de la canción o en uno propio. Algunas músicas cambian de compás casi en cada compás: la Promenade alterna 5/4 y 6/4, La consagración de la primavera pasa por 2/8, 3/8, 3/4, 5/8 y más. Una parte con `in_meters(&[...])` tiene una indicación de compás por compás. Un patrón tiene una sola indicación de compás, así que `Song::part_runs` corta una parte así en **tramos** de compases con la misma indicación, cada uno una parte propia con su porción de la melodía y del cifrado:

```
let mut runs: Vec<(usize, usize, TimeSignature)> = Vec::new();
for (bar, &time) in part.meters.iter().enumerate() {
    match runs.last_mut() {
        Some((_, bars, last)) if *last == time => *bars += 1,
        _ => runs.push((bar, 1, time)),
    }
}
```

`last_mut` da el último tramo, si lo hay, para cambiarlo en su sitio: un compás con la misma indicación lo alarga un compás, y cualquier otro empieza un tramo nuevo. Los trozos del texto salen de `bars_of`, que devuelve un slice del propio `&'static str` de la parte entre dos barras de compás, así que un tramo es una `Part` como cualquier otra y no cuesta ninguna copia. La demostración nombra el patrón de cada tramo siguiente por el compás donde empieza, «Llamada, compás 2», y pone una marca de compás donde cambia la indicación. La consagración de la primavera necesitaría más patrones de los que admite un proyecto, así que un tramo con las mismas notas que un tramo anterior, en el mismo compás, vuelve a tocar ese patrón anterior, como un productor reusa un loop.

El tempo de la partitura

Un `TempoMark` es una de las tres cosas que escribe una partitura:

```
pub enum TempoMark {
    /// From `at` on, `bpm` beats a minute, at once: a new section with
    /// a new tempo, such as "Più mosso" (more movement), "Adagio" or
    /// "Tempo I" (the first tempo again).
    At { at: usize, bpm: u16 },
    /// From `at`, slowing down or speeding up bit by bit until `bpm`,
    /// reached `over` steps later: "rit." or "rall." (slowing down),
    /// "accel." or "stringendo" (speeding up).
    Glide { at: usize, bpm: u16, over: usize },
    /// A fermata (𝄐), the "bird's eye" over a note or a rest: from `at`,
    /// for `steps` steps, the music waits, lasting twice as long; then
    /// it carries on at the tempo before.
    Fermata { at: usize, steps: usize },
}
```

`tempo(at, bpm)`, `glide(at, bpm, over)` y `fermata(at, steps)` son `const fn` que los crean, contando pasos desde el principio de la parte. La demostración convierte cada uno en marcas de tempo en la lista de reproducción, cada vez que se toca la parte. Un calderón se convierte en dos: la mitad del tempo mientras dura la nota sostenida, y después el tempo de antes. El proyecto no tiene calderón propio; la mitad de rápido es exactamente «el doble de largo» (ver Cambios de tempo).

Rondas y cánones

`with_voices` dice quién canta la melodía, y desde qué compás de la parte: un `(voice, bar)` para cada copia. La ronda de Frère Jacques son cuatro voces, cada una dos compases después de la anterior:

```
Part::new("Round", FRERE_JACQUES_TUNE, FRERE_JACQUES_CHORDS).with_voices(&
[
    (1, 0),
    (2, 2),
    (3, 4),
    (4, 6),
]),
```

Un compás negativo es una voz que empezó en la parte anterior, así que solo se oye su final: «Round again» deja que las voces 2 a 4 terminen sus últimos compases mientras empiezan de nuevo. La voz 1 es el canal Melodía, y cada otra voz recibe un canal propio, «Voz 2», «Voz 3» y así.

Otros instrumentos, más fuerte y más suave

`played_by` toma una lista de `(program, semitones)`: cada instrumento toca la melodía, esa cantidad de semitonos más arriba, en un canal con su nombre. Boléro pasa su melodía de la flauta al clarinete, al fagot y así; su clarinete en mi♭ la toca una octava más arriba, `(gm::CLARINET, 12)`. `marked` fija lo fuerte que se toca la parte, de pianissimo a fortissimo:

```
pub fn scale(self) -> f32 {
    match self {
        Dynamic::Pianissimo => 0.4,
        Dynamic::Piano => 0.55,
        Dynamic::MezzoPiano => 0.75,
        Dynamic::MezzoForte => 1.0,
        Dynamic::Forte => 1.15,
        Dynamic::Fortissimo => 1.3,
    }
}
```

`with_drums` escribe una parte de batería, en la notación de las melodías, con `x` para un golpe y `X` para uno acentuado. En lugar de un segundo analizador, `drum_line` convierte cada `x` en un `C4` y cada `X` en un `D4`, lee la línea como una melodía, y después hace de cada nota un golpe de un paso, acentuado si era un `D4`, tan fuerte como dicen las dinámicas de la línea: tresillos, silencios y dinámicas salen gratis.

La banda de una canción

`band` nombra los programas de General MIDI de los canales Melodía, Teclados y Bajo, y quizá una sección de vientos:

```
pub struct Band {
    pub lead: u8,
    pub keys: u8,
    pub bass: u8,
    pub horns: Option<Horns>,
}
```

La mayoría de las bandas son constantes que comparten las canciones de un género:

```
const BOSSA: Band = Band::new(gm::TENOR_SAX, gm::NYLON_GUITAR,
gm::ACOUSTIC_BASS);
```

`Band::new` y `with_horns` son `const fn`, así que una banda se puede construir dentro de una constante. Una sección de vientos es un programa y una manera de acompañar: los saxofones de una big band sostienen los acordes (`Comping::Held`) y sus metales los golpean en el pulso y justo antes del pulso 3 (`Comping::Charleston`). Los violines de una orquesta típica de tango también son una «sección de vientos», con un programa de cuerdas: la demostración llama al canal «Cuerdas» para los programas 40 a 51 y «Vientos» para cualquier otro. Una canción con una banda con vientos es algo distinto de la sección de vientos del estilo de banda, que cualquier canción puede encender.

De una canción a un proyecto

`song_demo(song)` construye el proyecto en cuatro pasos.

Canales. La Melodía, un canal para cada voz de más, los Teclados solo si alguna parte toca acordes o una mano izquierda, el Bajo salvo que la canción no tenga línea de bajo, los vientos, un canal para cada instrumento al que se pasa una melodía, y uno para cada nombre de línea. Cada uno recibe su programa.

Un patrón para cada parte, o para cada tramo de una parte cuyo compás cambia. Ahí va la melodía en los canales de sus voces, el cifrado acompañado con el acompañamiento y el bajo de la parte (`accompany_in_bars`, que mantiene el acompañamiento al paso de los compases, así un acorde de un pulso igual se toca), la parte de los vientos, la mano izquierda y las líneas escritas, y la batería escrita. Después dos pasadas hacen que suenen tocados (ver la sección siguiente) y la dinámica de la parte los escala.

Un patrón de batería para cada parte, con el ritmo de la canción o el de la parte, en el compás de la parte, con un fill al final de las partes de cuatro compases o más, salvo que la parte escriba su batería.

El arreglo. La forma se recorre una vez, y cada entrada pone los patrones de su parte uno tras otro en la pista Canción y su batería en la pista Batería desde `drums_from`, con marcas de tonalidad, de tempo y de compás donde cambian, y el cifrado escrito en la hoja de acordes. Un último compás en el acorde de tónica, con un platillo, termina la canción, en la tonalidad de la última parte, así una canción que modula termina en la tonalidad nueva. Una pieza cuya última parte está escrita entera termina donde termina su partitura, y lo mismo un cifrado que termina en un corte, como el «chan-chan» de un tango: la partitura ya dice cómo se detiene la música.

Dos detalles la terminan. Las canciones, los himnos, los tangos y las piezas de concierto se frenan un poco en su último compás, como hace quien toca donde no hay caja de ritmos a la que seguir:

```
fn ends_broadening(song: &Song) -> bool {
    use super::Genre::*;
    matches!(song.genre, Classical | Folk | Soul | Tango | Folklore)
}
```

Eso es un *poco ritardando* hasta nueve décimos del tempo, salvo que la partitura ya marque cómo va el tempo de la última parte. Y el swing es el del ritmo de la canción, salvo en el folk y la música clásica, cuyas corcheas quedan parejas sobre cualquier ritmo:

```
pub fn swing(&self) -> u8 {
    match self.genre {
        Genre::Folk | Genre::Classical => 0,
        _ => Rhythm::named(self.rhythm).map_or(0, |rhythm| rhythm.swing),
    }
}
```

El resultado es un proyecto corriente, para desarmar y cambiar: nada en él está marcado como demostración.

Tocar como una persona

Una partitura escrita nota a nota, cada nota tan fuerte como la siguiente, suena a máquina. `interpret.rs` cambia las velocidades unos pocos puntos sobre 127, como lo haría quien toca; las notas se quedan donde están. Para una melodía, `perform` le da forma a cada frase:

- el primer tiempo del compás un poco más fuerte, los otros tiempos menos, las notas a contratiempo más ligeras;
- cuanto más aguda una nota en su frase, más se canta;
- una nota larga, apoyada; la última nota de una frase, relajada;
- y nunca dos notas del todo iguales.

Una frase termina en un silencio de una corchea o más, la respiración más corta de quien canta, o después de cuatro compases. Los acordes y el bajo, cuyos acentos ya escribió el acompañamiento, solo pierden su regularidad: las notas de un acorde un poco distintas entre sí, los acordes un poco más. Después `follow` hace que la banda toque tan fuerte como está marcada la melodía donde empieza cada una de sus

notas, así un crescendo en la melodía arrastra a la banda, y un acento en la melodía es medio acento en la banda.

«Nunca dos iguales» no puede querer decir «distinto cada vez»: una demostración tiene que sonar igual cada vez que se abre, y un test que comprueba sus notas tiene que ver las mismas notas. Así que las pequeñas diferencias salen de una función que parece aleatoria pero no lo es:

```
fn wobble(seed: u64, value: u64) -> f32 {
    let mut x = seed ^ value.wrapping_mul(0x9E37_79B9_7F4A_7C15);
    x = (x ^ (x >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
    x = (x ^ (x >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
    x ^= x >> 31;
    (x >> 40) as f32 / (1u64 << 23) as f32 - 1.0
}
```

Es el paso de mezcla de **SplitMix64**: multiplicaciones y desplazamientos que esparcen los bits de la entrada tan a fondo que entradas vecinas dan salidas sin relación. La entrada es el paso y la altura de la nota; la semilla se hace con los nombres de la canción, la parte y el canal, con FNV-1a (ver FNV-1a). Los 24 bits más altos del resultado, divididos por 2^{23}, dan un número de 0 a 2, y el menos uno lo lleva de −1 a 1. `wrapping_mul` multiplica y descarta lo que no cabe en 64 bits, lo que un `*` normal trataría como desbordamiento. La forma que se le da queda pequeña al lado de la diferencia entre piano y forte: una velocidad cambiada por `clamp` queda entre 24 y 124, o donde estaba escrita si eso queda más afuera, así un pianissimo sigue suave.

El menú Demostraciones

El menú muestra las demostraciones por tipo (**Canciones**, **Estilos**, **Dodecafónico**, **Micromodos**), las canciones y los estilos por género, y cada lista en orden alfabético. Las listas de la biblioteca guardan el orden en que se escribieron; el menú ordena una copia, que construye la primera vez que se muestra, en un `LazyLock`:

```
static DEMOS: LazyLock<Demos> = LazyLock::new(||
Language::English.with(all_demos));
static DEMOS_ES: LazyLock<Demos> = LazyLock::new(||
Language::Spanish.with(all_demos));
```

Una `static` tiene que conocerse cuando se compila el programa; un `LazyLock` es una `static` cuyo valor calcula la clausura la primera vez que alguien la lee, y lo guarda. Hay dos, porque los títulos se ordenan tal como se muestran, y en español muchos se muestran en español.

Alfabético, para personas

Comparar texto letra por letra, por número de Unicode, pone todas las mayúsculas antes de todas las minúsculas y «Ábalos» después de «Zamba». Así que el menú ordena por una copia **plegada** de cada título: en minúsculas, sin tildes, sin puntuación, con el guion como un espacio. «Für Elise» se ordena como «fur elise», justo después de «Funeral March»:

```
fn by_title(a: &str, b: &str) -> Ordering {
    fold(a).cmp(&fold(b)).then_with(|| a.cmp(b))
}
```

`then_with` desempata, dos títulos que se pliegan igual, por los títulos mismos, así el orden nunca depende del azar. `fold` es un solo `match` sobre los caracteres: cada letra con tilde se convierte en la simple, `♯` y `♭` en `#` y `b`, las letras, los dígitos, `#`, `/` y `+` se quedan (así se pueden buscar «F#», «3/4» y «2+2+3/8»), y cualquier grupo de espacios y signos entre palabras se convierte en un espacio.

Buscar

Un cuadro de búsqueda está encima de los submenús y tiene el teclado en cuanto se abre el menú. Cada demostración tiene tres textos plegados: su título (en los dos idiomas, así «Noche de paz» encuentra Silent Night en cualquiera de los dos), sus detalles (quién la escribió, su estilo, su género, su tonalidad y su compás) y las notas sobre ella. Cada palabra escrita tiene que empezar una palabra en uno de ellos: `bach` encuentra a Bach, no a Offenbach:

```
fn starts_a_word(text: &str, words: &str) -> bool {
    text.match_indices(words)
        .any(|(at, _)| at == 0 || text[..at].ends_with(' '))
}
```

Las coincidencias se ordenan, la mejor primero:

```
enum Rank {
    TitleStart,
    TitleWords,
    Details,
    Words,
    About,
}
```

Primero va el título que empieza con lo escrito, luego cada palabra encontrada en el título, luego la frase encontrada en los detalles, como «re menor» en una canción en re menor, luego las palabras encontradas por separado, y al final una palabra encontrada solo en las notas. `Rank` deriva `Ord`, así que las variantes se comparan en el orden en que están escritas, y un orden **estable** por rango mantiene el orden alfabético entre iguales. Los dos rangos más débiles se muestran más tenues, así resaltan las coincidencias que más probablemente buscabas.

Cómo se prueba

Las canciones son datos, y los tests los leen todos. En `song.rs`: `every_bar_of_every_part_adds_up` (cada compás de cada melodía y mano izquierda dura lo que su indicación de compás, cada cifrado se puede leer, una parte con un compás por compás tiene tantas indicaciones como compases), `forms_name_real_parts_and_rhythms_exist`, `every_song_names_a_real_style`, `songs_in_three_play_in_three`, `songs_that_change_meter_have_a_style_that_follows_it`, `every_genre_has_songs_and_they_are_in_order`, `keys_are_written_as_their_tunes_are` (Take Five en mi♭ menor, el trío del Maple Leaf Rag en re♭ mayor), `melodies_are_mostly_in_their_key` (al menos el 80% de las notas de una canción en la tonalidad de su parte, lo que atrapa un fa escrito por un fa♯) y `the_first_notes_are_right` (Oda a la alegría empieza fa♯ fa♯ sol la).

En `demo.rs`, `every_song_makes_a_demo` construye las 343, y otros comprueban una cosa cada uno: `a_round_has_its_voices_in_the_right_places`, `a_part_in_its_own_time_gets_time_markers`, `runs_of_the_same_bars_share_a_pattern`, `tempo_markings_go_on_the_playlist`, `songs_and_hymns_slow_down_a_little_at_the_end`, `a_song_that_ends_after_a_modulation_ends_in_the_new_key`, `a_piece_written_out_in_full_ends_where_its_score_ends`, `a_big_band_has_its_horns`, `the_band_follows_the_dynamics_of_the_tune_and_the_breaks_of_the_chart` y `every_demo_keeps_a_sensible_tempo` (cada marca de tempo entre tres décimos y dos veces y media el tempo de la canción). `interpret.rs` comprueba que `piano_stays_soft_and_fortissimo_loud` y que `a_tune_gets_its_accents_but_stays_close_to_how_it_was_written`. `demo_menu.rs` comprueba el plegado, el orden y los rangos, como `matches_only_in_the_notes_come_last`.

Además de los tests unitarios, cualquier demostración se puede tocar sin ventana con la traza de reproducción, `ferrous-lineup --trace "Frère Jacques" --out round.tsv`, y `ferrous-lineup --demos` lista los nombres que acepta.

De una interpretación MIDI o una partitura a una demostración

Archivos: `crates/lineup-core/src/library/song/`, `crates/lineup-core/src/midi_file/import.rs` y `crates/lineup-core/tests/key_finding.rs`

Una demostración es una constante de texto (ver Cómo se escribe una demostración). Las melodías cortas se escriben a mano. Las piezas más largas se leen de una versión que una computadora puede leer, un archivo MIDI, un grabado o una partitura guía, con pequeños scripts escritos para cada tipo de fuente. Los scripts no son parte de la aplicación: lo que escriben sí, como constantes de Rust corrientes, y el comentario `// Source:` encima de cada canción dice de dónde se leyó y qué se dejó afuera. Este capítulo trata de los métodos: cómo se leen de las notas un pulso, una melodía, una tonalidad y unos acordes, cómo reciben las notas su nombre, y qué conserva una demostración de una partitura. Casi todo es teoría musical puesta a trabajar. Qué escuchar en las piezas está en la guía de escucha del manual.

Tipos de fuente

Fuente	Qué contiene	Para
ABC (las colecciones de John Chambers y Henrik Norbeck, The Traditional Tune Archive)	una melodía escrita con letras, a menudo con nombres de acordes	canciones y danzas folk, los Balcanes
Partituras guía en MusicXML (OpenEWLD, la parte de dominio público del archivo Wikifonia)	una melodía con los nombres de los acordes encima	canciones de casi todos los géneros: estándares de swing, blues, latinas, soul, pop
Grabados de LilyPond (el Mutopia Project, OpenScore)	la partitura entera, cada voz, como texto	piezas clásicas
Secuencias MIDI de una partitura	cada voz, tocada con clic	la Misa Criolla, La consagración de la primavera, Petrushka, arreglos de tango y de folclore

Fuente	Qué contiene	Para
Interpretaciones MIDI	un pianista tocando libre, sin clic	tangos tocados al piano
Una transcripción en grados de la escala (TheoryTab de Hooktheory)	la melodía y los acordes como números en una tonalidad	el tema de The Big Bang Theory
SymbTr	música turca de makam, nota a nota	las canciones turcas
Imágenes de las páginas de una partitura impresa	lo que lee el ojo, compás por compás	algunos tangos y canciones

Los formatos de texto ya se parecen a la notación de la aplicación, así que sus scripts sobre todo traducen. Los tresillos de una partitura guía pasan a las semicorcheas más cercanas, como los redondea la propia partitura guía de la aplicación, y donde un acorde de una partitura guía choca con su propia melodía se usa el acorde que suelen tocar los músicos de jazz ahí, con una nota en el comentario. Donde una fuente no trae acordes, el comentario dice que los acordes son una armonización sencilla. Muchas danzas de los Balcanes se escriben en semicorcheas (7/16, 11/16); aquí se escriben en corcheas, cada nota el doble de larga, lo que suena igual al tempo indicado.

Leer una partitura

Un grabado, a través de MIDI

LilyPond es un lenguaje para imprimir música a partir de texto. Estas son las primeras notas de los primeros violines de «Porque un niño nos ha nacido» de Handel:

```
re'''4 sol''8 sol'' do''' do''' do''' do''' |
```

Los nombres de las notas son los italianos, los mismos que en español (`re`, `sol`, `do`), cada `'` sube la nota una octava, y el número es la duración (`4` una negra, `8` una corchea). LilyPond escribe una partitura como archivo MIDI además de como página, y el script lee ese archivo compás por compás:

- **Dinámicas.** LilyPond toca cada indicación de dinámica a una velocidad fija (*p* 69, *mf* 86, *f* 95), así que esas velocidades se leen de vuelta como las indicaciones, `!p`, `!mf`, `!f`.
- **Staccato.** LilyPond toca una nota en staccato la mitad de larga de lo escrito. Para distinguir una negra en staccato de una corchea, la partitura se compila dos veces, una sin sus puntos de staccato: una nota que es más corta en el primer archivo está en staccato, y lleva un `.` en la demostración.
- **Fusas.** Los trinos escritos de Bach son más rápidos que un paso. La notación puede escribir medio paso, `D5:1/2`.

Algunas partituras dejan cosas afuera, y el comentario dice qué se agregó: la edición del Mesías imprime sus fortes solo en los primeros violines, así que un forte vale para todas las cuerdas. Las ediciones también tienen errores. El grabado de Die Forelle escribe re natural en los compases 43 y 44, contra un re bemol en la mano izquierda. Los compases 19 y 20 tienen la misma música con re bemol, y también una segunda edición, la de OpenScore: la demostración toca re bemol.

Una anacrusa de su propia duración

Alfonsina y el mar empieza una corchea antes de su primer compás completo. En lugar de un compás de 6/8 que es casi todo silencio, como escribe su anacrusa «Clementine», la anacrusa puede ser un compás de su propia duración, porque una parte puede tener su propio compás:

```
const ONE_EIGHT: TimeSignature = TimeSignature::new(1, 8);
```

```
Part::new("Pickup", "!mf F#4:2", "")
    .in_time(ONE_EIGHT)
```

`TimeSignature::new` es una `const fn`, así que una indicación de compás también puede ser una constante.

Un coro son tres canales, no uno

Dos voces de un coro a menudo cantan la misma nota. Supón que las contraltos empiezan un mi en semicorcheas mientras los tenores sostienen un mi durante un pulso entero: en un mismo canal son dos notas con la misma altura en el mismo paso, y un patrón nunca guarda dos notas así (ver Nunca dos notas en el mismo lugar): gana la primera, y el mi sostenido de los tenores se perdería. Por eso el coro tiene sus contraltos, tenores y bajos en tres líneas, tres canales, como están en tres

pentagramas de la partitura. El coro de la Misa Criolla es una sola línea, «Choir» (Coro), con sus voces unidas por `&`: donde dos de ellas coinciden en una nota, esa nota suena una vez.

Bajo cifrado: acordes escritos como números

Los compositores barrocos no escribían los acordes del teclado. Debajo de la línea del bajo escribían números pequeños, las **cifras**, cada una un intervalo sobre la nota del bajo, y quien tocaba completaba los acordes. La edición de Chrysander del Mesías imprime cifras sobre parte del bajo, y el cifrado de la demostración de Handel las sigue donde aparecen:

Cifra	Significa	En este coro
ninguna	una tríada en estado fundamental	casi todos los compases
6	una tríada en primera inversión: la tercera en el bajo	compás 1: si con 6 es `G/B`
6/5	un acorde de séptima en primera inversión	compás 1: fa♯ con 6/5 es `D7/F#`
4/2	un acorde de séptima en tercera inversión: la séptima en el bajo	compás 28: sol con 4/2 es `A7/G`; compás 43: do con 4/2 es `D7/C`
6 con una barra	primera inversión, con la sexta alzada un semitono	compás 26: mi con un 6 alzado es `C#dim/E`
+ solo	la tercera alzada	compás 22: la con do♯, `A`
un becuadro	la tercera natural	compás 58: re con fa natural, `Dm`

Una séptima quiere bajar un grado, así que un acorde de 4/2, con la séptima en el bajo, va casi siempre seguido de un 6 sobre la nota de abajo: `A7/G` va a `D/F#` en el compás 29, `D7/C` a `G/B` en el 44. Cómo escribe las inversiones la hoja de acordes está en Acordes con barra e inversiones del manual.

Leer una secuencia

Una **secuencia** es un archivo MIDI hecho en un secuenciador, cada voz en su pista, con clic. La Misa Criolla, La consagración de la primavera y Petrushka salen de

secuencias de sus partituras. Las barras de compás están en el archivo; lo que tiene que hacer el script es volver a leer la interpretación como partitura:

- cada nota va a la semicorchea más cercana, o al tercio de una más cercano donde la secuencia toca un tresillo, que entonces cae justo en el tick (ver Notas entre pasos);
- el mapa de tempo se lee como las indicaciones de la partitura: una caída profunda que vuelve es un **calderón**, una serie de cambios pequeños un **deslizamiento** (*rit.* o *accel.*), y cualquier otra cosa un tempo nuevo;
- las velocidades se leen como dinámicas, de *pp* a *ff*;
- un **trémolo**, el mismo acorde tocado más rápido que una semicorchea, se escribe como un solo acorde sostenido.

Una partitura de orquesta tiene más líneas que canales una banda. El script mantiene la melodía delante y la sigue de instrumento en instrumento con `played_by`, da canales propios a las líneas que importan, y escribe el resto en el canal Teclados. Las líneas dobladas al unísono o a la octava se escriben una vez. Las notas de adorno, los trinos y los glissandos se simplifican, y el comentario dice cómo.

Leer una interpretación

Un pianista que toca libre, sin clic, deja un archivo MIDI sin barras de compás: la cuadrícula del archivo no significa nada. Los tangos de `tango_played.rs` salen de archivos así. Todo lo que diría una partitura hay que oírlo.

Encontrar el pulso

El pulso se encuentra como lo encuentra quien escucha, con el método de «Beat tracking by dynamic programming» de Dan Ellis (2007), el mismo que la aplicación usa con una grabación (ver Pulsos: arranques, autocorrelación y programación dinámica): cuánto sonido nuevo empieza en cada momento, el período más estable en eso, y luego la cadena de pulsos que cae en más arranques sin alejarse de ese período. Cada compás recibe entonces el tempo al que se tocó, y se escribe una marca de tempo donde el tempo se mueve medio pulso por minuto o más. Uno tiene 61.

Un pulso estable fuera de la cuadrícula

Algunos archivos se secuencian con un clic estable pero se guardan a otro tempo, así que los pulsos del clic duran una fracción de tick y se corren a través de los compases

del propio archivo. Las notas de Balderrama no caen cerca de la cuadrícula del archivo, pero su bombo golpea cada 293 o 294 ticks del primer compás al último: un pulso estable. Para encontrarlo exactamente, numera los golpes por el pulso en que caen, `k = 0, 1, 2, ...`, y traza una recta por ellos, tick contra `k`. La recta de **mínimos cuadrados** es la que tiene la menor suma de errores al cuadrado, y su pendiente, la duración del pulso, tiene una fórmula:

```
period = Σ (k − k̄)(t − t̄) / Σ (k − k̄)²
```

donde `k̄` y `t̄` son los promedios. En Balderrama la corchea dura 293,966 ticks, y los golpes se apartan de la recta unos pocos ticks, milésimas de segundo: una máquina, no una persona.

La recta da el pulso pero no el compás: ¿cuál de las seis corcheas es la primera? La armonía cambia en el primer tiempo, y de las notas largas del coro la mayoría empieza en una misma corchea de las seis. Esa corchea es la primera del compás, y resulta que la melodía empieza dos corcheas antes, con una anacrusa, como suele hacer una zamba.

La melodía y la tonalidad

La **melodía** es la línea de arriba: de las notas que suenan en cada momento en la pista de la melodía, la más aguda (el **perfil**, como lee la melodía la partitura guía también). Las notas debajo de ella que la doblan, como una segunda voz en terceras, van en una línea propia.

La **tonalidad** sale de cuánto suena cada una de las doce clases de altura, comparado con perfiles de tonalidad, como lo hace el propio detector de tonalidad de la aplicación (ver Comparar: el método de Krumhansl-Schmuckler). Donde elige mal, la tonalidad se pone a mano, y una parte que cambia de tonalidad recibe su propio `in_key`. El test de que al menos cuatro notas de cada cinco estén en la tonalidad de su parte (`melodies_are_mostly_in_their_key`) atrapa también una elección equivocada de menor: la melodía de Balderrama canta re natural mucho más que re sostenido, así que en mi menor armónica solo 228 de sus 288 notas están en la tonalidad y el test falla, mientras que en mi menor natural están 257.

Los acordes

Para cada compás, medio compás o pulso, el script puntúa cada acorde común sobre las doce fundamentales: las notas dentro del acorde suman a su favor, las de afuera en contra, y una nota del bajo en la fundamental del acorde suma de más. Los

acordes de la tonalidad reciben un pequeño premio, así una nota de paso no convierte un la menor en algo exótico, y un compás se divide solo donde la división puntúa claramente mejor. Es la misma idea que la escucha de acordes de la propia aplicación, y tiene la misma debilidad: las notas solas nombran cada color como un acorde propio. Una sexta que agrega el pianista convierte Bm en Bm6, y una nota de la melodía de paso le agrega una novena bemol a F♯7.

Una **partitura guía** de la canción arregla eso. Sus compases no coinciden con una interpretación libre (el pianista agrega una introducción, repite una sección, estira un compás), así que sus acordes no se ponen sobre los compases uno por uno. En cambio, la partitura le da a la lectura su **vocabulario**: solo se pueden elegir los acordes que usa la partitura. Una partitura en otra tonalidad primero se **transporta**, cada acorde movido el mismo intervalo, lo que mantiene la función de cada acorde en la tonalidad: la partitura de Uno está en sol y el pianista toca en do, así que G pasa a C, D7 a G7 y Em a Am.

Los compases 38 a 40 de Yira, yira, leídos solo de las notas, son `Bm6 F#7b9 | Bm F#7 | Bm A7`. Limitados a los acordes de la partitura son

```
Bm F#7 | Bm F#7 | Bm A7
```

La mano izquierda colorea el si menor con su sexta, sol♯, y la melodía toca sol, una novena bemol sobre F♯7. Debajo de los colores están los acordes sobre los que se construye la canción, i y V7 de si menor.

Sin partitura guía, los acordes se comprueban de oído y por lo que hacen. Tiempos viejos termina en re, fa, la y si. Esas cuatro notas son también si, re, fa, la, **Bm7♭5**, pero el bajo toca re, y el acorde viene después de A7, la dominante de re menor: es **Dm6**, re menor con sexta agregada. El nombre de un acorde sigue a su bajo y a su función, no solo a sus notas.

El nombre de las notas

Un archivo MIDI tiene números, no nombres. La nota 70 es la sostenido o si bemol, y el archivo no dice cuál. La tonalidad nombra las siete notas de su escala; las otras cinco siguen al acorde que las rodea:

- **Una nota del acorde se escribe como la tiene el acorde.** En F♯7 (fa♯, la♯, do♯, mi) la nota 70 es la sostenido, la tercera del acorde; en Gm (sol, si♭, re) es si bemol.
- **Un acorde sobre una nota de fuera de la tonalidad se nombra por adónde va.** Una séptima de dominante baja una quinta, así que el acorde antes de C♯m

es G♯7. Un acorde de séptima que baja un semitono es un **sustituto tritonal**: el acorde antes de G es A♭7, no G♯7. Una séptima disminuida sube un semitono: G♯dim7 va a A.

- **Una nota de fuera del acorde se escribe por adónde va.** Una nota de paso cromática entre dos notas a un tono de distancia es sostenido al subir y bemol al bajar: la, la♯, si y sol, sol♭, fa.
- **Nada de mi sostenido, si sostenido, do bemol ni fa bemol para una nota de paso**: en el papel parecen errores por fa, do, si y mi. Las notas propias de la tonalidad sí pueden serlo: en si menor la cuarta alzada que lleva a la dominante, fa sostenido, es mi sostenido.

Cómo nombra la propia aplicación las notas de fuera de la tonalidad, sin mirar ningún acorde, está en Notas de fuera de la tonalidad; cómo las nombra la partitura guía, que mira el acorde como estas reglas, en Qué nombre de nota.

Una transcripción en grados de la escala

El tema de The Big Bang Theory sale de una **transcripción** del sitio TheoryTab de Hooktheory, donde las canciones se escriben como las lee una clase de teoría: cada nota un **grado** de la escala en la tonalidad (1 es la tónica, 5 la quinta), cada acorde un número romano. El conjunto de datos Sheet Sage (Chris Donahue y otros, 2022) reúne unas 26.000 en un archivo. Una nota ahí se ve así:

```
{"sd": "6", "octave": 0, "beat": 1, "duration": 1}
```

Grado 6, en la octava que empieza en la tónica, en el pulso 1, de un pulso de duración. El tema está en sol mayor, y con la tónica en G4 la melodía va de D4 a G5, el registro en que la escribe una partitura guía:

Escrito ahí	En sol	Aquí
sd 6, octava 0, 1 pulso	mi, encima de la tónica	`E5:4`
sd 5, octava 0, ½ pulso	re	`D5:2`
sd 1, octava 0, ¼ de pulso	sol, la tónica	`G4:1`
sd 5, octava −1, ½ pulso	re, debajo de la tónica	`D4:2`

La transcripción también está **alineada** con la grabación: el pulso 1 de la estrofa está a los 7,6 segundos y su pulso 33 a los 19,66 segundos. 32 pulsos en 12,06 segundos son 32 × 60 / 12,06 = 159,2 pulsos por minuto, así que la estrofa lleva `.with_tempo(&[tempo(0, 159)])` después de una introducción a 130.

Lo que una demostración tiene que conservar

Dos tests sobre todas las canciones deciden cómo se corta una pieza en partes y cómo se escribe su tempo.

Una exportación MIDI tiene que volver nota por nota.
`every_demo_comes_back_note_for_note` exporta cada demostración como archivo MIDI, lo importa, lo vuelve a exportar y compara los dos archivos. El importador corta una canción larga en patrones en barras de compás que ninguna nota cruza (ver Cortar la canción en patrones); donde una nota sostenida cruza cada barra de compás durante muchos compases, tiene que cortar una, y la nota cortada se vuelve a tocar. Una demostración vuelve a tocar las notas sostenidas en los límites de sus propias partes de todos modos, así que un límite en la barra de compás correcta da un corte limpio: el Credo de la Misa Criolla, cuyo bajo se liga sobre casi todas las barras de compás, tiene uno en el compás 30, donde entran los solistas.

Una demostración mantiene un tempo razonable.
`every_demo_keeps_a_sensible_tempo` mantiene cada marca de tempo entre tres décimos y dos veces y media el tempo de la canción. El último acorde de una secuencia a menudo se sostiene frenando el clic casi del todo: la secuencia del Credo sostiene su último acorde a 20 pulsos por minuto, y la demostración a 46.

Otro test lee las canciones de una manera que pide cuidado. El test del detector de tonalidad mide cuántos compases de las demostraciones oye en su tonalidad escrita, y una pieza en dos tonalidades a la vez haría que esa cifra no significara nada: diga la tonalidad que diga, la mitad de la música no está de acuerdo. Esas canciones van en una lista propia:

```
const POLYTONAL: &[&str] = &["The Rite of Spring, The Augurs of Spring"];
```

El test principal las deja afuera, y un segundo test comprueba que la tonalidad escrita esté entre las que se oyen, pero que no sea la única:

```
assert!(
    result.right_bars < result.bars,
    "{name} is heard in one key: drop it from POLYTONAL"
);
```

El segundo assert mantiene honesta la lista: si el detector llegara a oír los Augurios en una sola tonalidad, la canción dejaría de pertenecer a ella.

Comparar una demostración con su fuente

El último paso es tocar la demostración con la traza de reproducción:

```
ferrous-lineup --trace "Balderrama" --out balderrama.tsv
```

y buscar cada nota de la fuente en la traza: para una partitura en el mismo paso y la misma altura, en el canal correcto y con su duración; para una interpretación en la misma altura con un margen de un quinto de segundo. Lo que falta tiene que ser lo que una demostración deja afuera a propósito:

- los unísonos en un mismo canal, que suenan una vez (dos manos en el canal Teclados, dos guitarras que tocan la misma nota, las voces de un coro en una sola línea);
- los trémolos y los rasgueos más rápidos que una semicorchea, escritos como acordes sostenidos;
- los golpes de batería que caen en el mismo paso de un carril;
- las notas que nombra el comentario, como un error de una edición corregido.

Ejercicios: las canciones como datos

Estos ejercicios practican esta parte con las canciones de verdad: las constantes de `library/song/`, la demostración en que se convierten, y la música de la que se leyeron. Cada uno dice qué **ya existe**, para que lo extiendas en lugar de construirlo otra vez. La mayoría son cosas para abrir en la aplicación y escuchar, o para resolver con lápiz; algunos son trozos de Rust con un final claro; los que se llaman «Piénsalo» son preguntas, sin código que escribir.

Haz los de Rust en una rama propia. Ejecuta `cargo test` después de cada cambio, y `cargo fmt --all --check` y `cargo clippy --workspace --all-targets -- -D warnings` antes de dar uno por terminado: el CI ejecuta los tres.

Las canciones como constantes

1. Una canción nueva en el archivo de su género (lineup-core, media)

Agrega una melodía de dominio público a las demostraciones. `Song`, `Part`, `SONGS` y los archivos de género ya existen; agregas una entrada, no una función (ver Una canción es una constante).

1. Lee dos canciones del archivo de género que elijas, como `folk.rs` o `blues.rs`, y copia la forma de una, cada campo de `name` a `about`.
2. Escribe la melodía con `|` entre compases y un cifrado debajo, y dale a la parte un nombre como «Verse» o «Tune».
3. Hazla `pub(super) const`, y agrégala a `SONGS` en `song.rs`, al final del bloque de su género.
4. Deja que los tests encuentren los errores: un compás que no suma, un acorde que no se puede leer, un estilo en otro compás, una nota que no está en la tonalidad.

```
cargo test -p lineup-core library
cargo run --release -- --demos
```

Está terminado cuando los tests pasan y la canción está en **Archivo > Demostraciones > Canciones**, en su género, se abre en la hoja de acordes y suena.

Practica: literales de structs, datos `const`, leer los fallos de los tests.

2. Extender una canción que ya está (lineup-core, media)

Shenandoah toca su canción dos veces después de una anacrusa, y **Danny Boy** una estrofa y un estribillo. Elige una y agrégale una parte que la demostración todavía no toca: una introducción solo para la banda, una última estrofa o una coda. Escríbela compás por compás, dale un cifrado y agrégala a la `form` de la canción. Después abre la demostración y mira la hoja de acordes: ¿muestra los acordes que escribiste, en los compases que querías?

Practica: la notación de las melodías, `form` como una lista de nombres.

3. Un último estribillo un semitono más arriba (lineup-core, fácil)

Muchas canciones pop suben su último estribillo un semitono. Agrega una copia del estribillo de una canción con `.in_key(...)` un semitono por encima de la tonalidad de la canción, y ponla al final de la `form`. ¿Qué números romanos del cifrado hay que cambiar? (Ninguno: se leen en la tonalidad de la parte.) Escucha el final: el último compás de la demostración está en el acorde de tónica de la tonalidad nueva, como comprueba `a_song_that_ends_after_a_modulation_ends_in_the_new_key`.

Practica: métodos constructores sobre un valor `const`.

4. Un canon a la quinta (lineup-core, media)

`with_voices` toma pares `(voice, bar)`, así que cada voz de una ronda canta la melodía a la misma altura. Muchos cánones tienen la segunda voz una quinta más arriba o más abajo. Convierte cada par en un trío, `(voice, bar, semitones)`, y mueve las notas de cada voz sus semitonos donde la demostración construye las voces. Cada ronda ya escrita recibe un `0`. Después prueba Frère Jacques con su segunda voz `7` semitonos más arriba: ¿qué notas chocan, y por qué una ronda necesita sus copias a la misma altura para encajar en el mismo acorde?

Practica: cambiar un tipo y seguir los errores del compilador.

5. Las primeras notas de tu canción (lineup-core, fácil)

`the_first_notes_are_right` comprueba que la Oda a la alegría empieza fa♯ fa♯ sol la y Estrellita, ¿dónde estás? do do sol sol. Agrega una línea para una canción que conozcas bien, con sus cuatro primeras notas como números MIDI y sus nombres en el mensaje. Equivoca una a propósito primero, y lee lo que dice el fallo.

Practica: `assert_eq!` con un mensaje, los números de nota MIDI.

6. Apoyar la nota más aguda (lineup-core, media)

`perform`, en `library/interpret.rs`, le da a una nota de una melodía hasta seis puntos de velocidad según lo aguda que sea en su frase. Quien canta suele apoyarse en la *única* nota más aguda de una frase más que en las notas que la rodean. Dale a esa nota dos puntos más, y escribe el test primero: en Estrellita, ¿dónde estás?, ¿cuál es la nota más aguda de la primera frase? Mantén pasando el test `a_tune_gets_its_accents_but_stays_close_to_how_it_was_written`.

Practica: `Iterator::max_by_key`, escribir el test antes que el código.

7. Buscar por compás y tonalidad (la aplicación, fácil)

Abre **Archivo > Demostraciones** y escribe `7/8` en el cuadro de búsqueda, luego `5/4`, luego `2+2+3/8`. Después prueba `re menor` y `D menor`. ¿Qué coincidencias van primero, y cuáles más tenues? ¿Por qué `fold` conserva `/` y `+` cuando quita todos los demás signos (ver Buscar)?

8. Piénsalo: métodos que toman `self`

`in_key` y `with_tempo` son `const fn` que toman `self` por valor y devuelven una `Part` nueva. ¿Por qué eso permite encadenarlas dentro de una constante, `Part::new(...).in_key(...).with_tempo(...)`? ¿Por qué no funcionaría ahí un método que tome `&mut self`?

Leer la música

9. ¿Qué tecla negra? (lápiz, fácil)

En do mayor, nombra la tecla negra entre fa y sol, con las reglas de El nombre de las notas: sobre un acorde D7; en la línea sol, ?, fa; y sobre un A♭7 que va a G. Después nombra la tecla negra entre la y si sobre Gm, y sobre F♯7.

10. ¿Una nota nueva, o un modo nuevo? (lápiz y la aplicación, fácil)

Escribe las escalas de mi menor natural, mi menor armónica y mi dórico. ¿Qué nota cambia en cada una? Después toca los compases 21 a 23 de **Balderrama**. La melodía canta do♯: ¿sobre qué acorde? ¿Está la música en mi dórico ahí, o el do♯ es una nota de camino a otra? ¿Y por qué la canción está escrita en mi menor natural y no en mi menor armónica (ver La melodía y la tonalidad)?

11. Dos maneras de contar seis (la aplicación, fácil)

Da seis palmadas parejas, después agrúpalas 3+3 (UNO dos tres CUATRO cinco seis) y 2+2+2 (UNO dos TRES cuatro CINCO seis). Toca **Acuarela del río**, una polca en 6/8, y acompaña con palmas al bajo, luego a la batería. ¿Qué agrupación toca cada uno?

12. El tempo de un pianista (la traza de reproducción, fácil)

Traza **Uno**:

```
ferrous-lineup --trace "Uno" --out uno.tsv
```

y lee sus líneas `# tempo:` de los primeros ocho compases. ¿Dónde frena el pianista, y en qué lugar de la frase? ¿Cuántas veces cambia el tempo en toda la canción?

13. Fundamentales a una quinta (la aplicación, fácil)

En **Porque un niño nos ha nacido**, toca los compases 41 y 42 con solo el canal Teclados sin silenciar. Escribe los acordes desde el segundo pulso del compás 41 y sus fundamentales. ¿A qué distancia está cada fundamental de la siguiente? La nota que muestra el menú Demostraciones para la canción nombra la misma cadena.

14. Otro acorde de 4/2 (la aplicación, fácil)

El compás 28 del coro de Handel tiene `A7/G`, un acorde de séptima con la séptima en el bajo, cifrado 4/2 (ver Bajo cifrado). El compás 45 tiene otro. ¿Qué acorde es, qué acorde lo sigue, y la nota siguiente del bajo está un grado más abajo?

15. Las dominantes de la trucha (la aplicación, fácil)

Die Forelle (La trucha) está en re♭ mayor, y su tercera estrofa es la que cambia. Busca los cuatro compases de E♭7 en la hoja de acordes. ¿Qué acorde viene después, y de qué es dominante? ¿Y de qué es dominante E♭7?

16. Por qué do menor necesita si natural (la aplicación, fácil)

En **Las ovejas pueden pastar tranquilas**, busca en la hoja de acordes los compases de la sección central en do menor. ¿Qué acorde trae el si natural, y por qué lo necesita una tonalidad menor? (Compara la escala de la tonalidad, la menor armónica, con do menor natural.)

17. Quién toca el Boléro (la aplicación, fácil)

Toca **Boléro** y anota, para cada exposición de la melodía, el instrumento que la toca: te lo dicen los nombres de los patrones en la pista Canción. ¿Cuántas exposiciones hay antes de que la música deje do mayor, y a qué tonalidad va? Después lee las partes del Boléro en `classical.rs`: en «Horn, celesta and piccolos» (Corno, celesta y flautines), `played_by` les da a los flautines 19 y 28 semitonos. ¿Qué intervalos son esos por encima de la melodía del corno, y qué notas de la serie armónica copian?

18. El acorde de Petrushka (lápiz, fácil)

Toca do, mi, sol y fa♯, la♯, do♯ juntos: dos tríadas mayores a un tritono, el acorde de Petrushka. Ahora toca la escala octatónica do, re♭, mi♭, mi, fa♯, sol, la, si♭, que alterna semitonos y tonos. ¿Qué notas de la escala no están en el acorde? ¿Por qué el acorde no puede estar en ninguna tonalidad mayor o menor?

19. Un modo sobre un bordón (la aplicación, fácil)

Abre **Petrushka, La feria de carnaval al atardecer**, activa 🔍 **Buscar la tonalidad** y toca la Danza de los cocheros, escrita en mi mixolidio. ¿Qué tonalidad muestra el detector? Mi mixolidio tiene las notas de la mayor: ¿por qué un detector que compara cuánto suena cada nota oye la tonalidad con las mismas notas? Después imprime el informe de tonalidades:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored
--nocapture
```

y busca en él el último cuadro de Petrushka. ¿Qué tonalidades da la partitura, y cuáles oye el detector en su lugar?

20. Un acorde que la tonalidad no tiene (la aplicación, fácil)

Tea for Two está en la♭ mayor. Busca la marca de tonalidad donde empieza su segunda parte. ¿Qué acorde de la♭ es do mayor? (Cuenta desde la♭: do es la tercera nota, así que do mayor es III, un acorde que la tonalidad de la♭ no contiene.)

21. De menor a mayor (la aplicación, fácil)

La estrofa de **Volver** está en sol menor y su estribillo en sol mayor. Busca la marca de tonalidad donde empieza el estribillo. ¿Qué dos notas de sol menor armónica cambian en sol mayor? Búscalas en los primeros compases del estribillo.

22. Dos tempos y cuatro acordes (lápiz y la traza de reproducción, fácil)

Traza el **Tema de The Big Bang Theory** y busca la primera nota del compás 5 y del compás 6. ¿Cuánto dura un compás de 4/4 a 159 pulsos por minuto? (4 × 60 / 159 = 1,509 segundos.) ¿Y a 130? Después:

- Nombra los cuatro acordes de la estrofa, IVadd9 V I vi7, en sol y luego en la.
- En el compás 10 la melodía alterna sol y fa♯ sobre G, y luego sobre Em7. Sobre G, fa♯ es la séptima mayor del acorde; ¿qué es sobre mi menor?

23. Un tango contra su partitura guía (investigación, media)

Elige uno de los tangos de `tango_played.rs`, como **Patotero sentimental**, y busca una partitura guía. Transporta sus acordes a la tonalidad del pianista, y compara una parte compás por compás con el cifrado de la demostración. ¿Qué acordes nombraron las notas solas por un color (una sexta, una novena) en lugar del acorde sobre el que se construye la canción?

24. Piénsalo: un script dentro de la aplicación

Los scripts que leen archivos MIDI y grabados no son parte de la aplicación. ¿Qué haría falta para que **Archivo > Importar archivo MIDI…** escribiera una constante de demostración en lugar de un proyecto? Haz la lista de lo que hoy decide una persona y tendría que decidir el importador: la tonalidad, dónde empiezan las partes, qué pista es la melodía, qué acordes permite una partitura guía. ¿Cuáles de esas cosas ya sabe adivinar la aplicación (ver Oír la armonía)?

Cómo se prueba el código: tests unitarios, tests de ventana, trazas

Archivos: el `mod tests` al final de la mayoría de los archivos de `crates/*/src`, `crates/lineup-core/tests/`, `crates/lineup-listen/tests/` y `crates/lineup-app/src/app/ux_tests.rs`

Ferrous Lineup tiene más de 1300 tests: unos 900 en `lineup-core`, 150 en `lineup-audio`, 260 en `lineup-app` y unas pocas decenas en los otros dos crates. Cada capítulo de este libro termina con los tests de lo que describe. Este trata de los tipos de test, por qué existe cada uno, y cómo ejecutarlos.

Ejecutarlos

```
cargo test                          # every crate
cargo test -p lineup-core           # one crate
cargo test frequency                # only tests whose name contains
"frequency"
cargo test -- --nocapture           # show what the tests print
```

Un test es una función corriente marcada con `#[test]`. Pasa si termina, y falla si entra en pánico, que es lo que hacen `assert!`, `assert_eq!` y `unwrap` sobre un error. `cargo test` compila los crates en un modo de prueba, ejecuta cada test en hilos en paralelo, y lista los que fallaron con sus mensajes.

Los tests se ejecutan normalmente en una compilación de **depuración** (debug), y la integración continua los ejecuta así también. Eso importa: en una compilación de depuración un `+` o un `*` que desborda entra en pánico, y se comprueba cada `debug_assert!`, mientras que una compilación release da la vuelta al número y se salta los `debug_assert!`. Un error de ese tipo pasa en release y falla en depuración, así que los tests deben ejecutarse en depuración. Los pocos crates que harían una compilación de depuración demasiado lenta para tocar música, como el sintetizador de SoundFonts, se compilan optimizados incluso entonces (ver Compilaciones de depuración). Los informes largos, más abajo, son la excepción: se ejecutan con `--release`.

Tests unitarios, junto al código

La mayoría de los tests están al final del archivo que prueban:

```
#[cfg(test)]
mod tests {
    use super::*;
```

`#[cfg(test)]` deja el módulo afuera de toda compilación salvo `cargo test`, así que los tests no le cuestan nada al programa. `use super::*` trae todo lo del archivo, también lo privado, porque un módulo hijo ve los elementos privados de su padre. Así un test unitario puede comprobar una función auxiliar que el resto del programa nunca ve.

El sonido se puede probar sin escucharlo. Una onda senoidal cruza el cero dos veces en cada ciclo, así que contar cuántas veces cambia de signo la señal mide su altura:

```
fn sign_changes(samples: &[f32]) -> usize {
    samples
        .windows(2)
        .filter(|pair| (pair[0] < 0.0) != (pair[1] < 0.0))
        .count()
}
```

`windows(2)` da cada par de muestras vecinas, `filter` se queda con los pares a cada lado del cero, y `count` los cuenta. Un segundo de A4 (el la de 440 Hz) tiene que cambiar de signo unas 880 veces, dos veces 440:

```
// A4 is 440 Hz: about 880 sign changes in one second.
assert!((860..=900).contains(&changes), "got {changes}");
```

El margen deja lugar al ataque al comienzo de la nota. Los iteradores son perezosos: nada se ejecuta hasta que `count` pide valores, y el compilador convierte la cadena en un bucle tan rápido como uno escrito a mano.

Tests sobre datos

Buena parte de Ferrous Lineup son datos: 343 canciones, 215 estilos armónicos, 139 ritmos, los estilos dodecafónicos y de micromodos. Los tests los leen todos, así que un error en una tabla falla antes de que nadie lo oiga: un compás que no suma, un acorde que no se puede leer, un nombre repetido, un estilo cuyo ritmo no existe (ver

Cómo se prueba para la biblioteca y Cómo se prueba para las canciones). Sus mensajes nombran la entrada, así que un fallo dice qué canción abrir.

Tests en `tests/`, desde afuera

Un archivo en la carpeta `tests/` de un crate es un **test de integración**: un programa propio que usa el crate como lo haría cualquier otro crate, así que solo ve lo que el crate hace público.

Archivo	Qué comprueba
`lineup-core/tests/chord_reading.rs`	qué tan bien nombra la hoja de acordes lo que tocan las demostraciones más difíciles: el acorde de los Augurios, el acorde de Petrushka, una pila de cuartas
`lineup-core/tests/key_finding.rs`	cuántos compases de las demostraciones oye el detector de tonalidad en su tonalidad escrita, y que encuentre las modulaciones claras en su compás
`lineup-core/tests/damaged_files.rs`	que un proyecto, un archivo MIDI o una hoja de acordes dañados terminen en un error o en un proyecto, nunca en un cierre inesperado
`lineup-listen/tests/key_changes.rs`	qué tan bien se encuentran los cambios de tonalidad en demostraciones convertidas en audio

Archivos dañados

`damaged_files.rs` lee cientos de copias dañadas de proyectos, archivos MIDI y hojas de acordes buenos, como hace un **fuzzer**, y comprueba que cada una termine en un proyecto o en un error, nunca en un cierre inesperado. Cómo los daña, y el desbordamiento que encontró y que solo ve una compilación de depuración, están en Los archivos dañados nunca hacen caer la aplicación.

Informes

Algunos tests miden en lugar de comprobar. `report`, en `key_finding.rs` y en `chord_reading.rs`, imprime una tabla, canción por canción, de qué tan bien se oye

cada demostración. Están marcados con `#[ignore]`, así que `cargo test` se los salta, y se ejecutan aparte:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored
--nocapture
```

Los tests corrientes de al lado impiden que la cifra baje: `most_bars_of_the_demos_are_heard_in_their_key` falla si menos del 87% de los compases se oyen en su tonalidad. El informe dice por qué un cambio movió esa cifra; el test se asegura de que nunca se mueva para el lado equivocado sin que nadie lo note (ver Qué tan bien escucha).

Tests de ventana

egui dibuja la ventana entera de nuevo en cada fotograma, a partir del estado de la aplicación (ver Modo inmediato). Eso hace que la ventana misma sea fácil de probar: un test puede ejecutar fotogramas sin pantalla, darles clics y teclas, y leer lo que se dibujó. `ux_tests.rs` tiene un `Screen` para eso:

```
fn new() -> Self {
    let ctx = egui::Context::default();
    theme::apply(&ctx);
    ctx.all_styles_mut(|style| style.animation_time = 0.0);
    let mut app = LineupApp::with_audio(&ctx, None,
Audio::Unavailable("test".into()));
```

La aplicación arranca como lo hace sin tarjeta de sonido (`Audio::Unavailable`), con las animaciones apagadas para que un fotograma muestre el final de cada animación, y nunca lee ni escribe una copia de recuperación de verdad ni descarga un SoundFont. `draw` ejecuta un fotograma con los eventos dados; egui devuelve las formas que pintaría, y `text_rect` busca una etiqueta entre ellas:

```
fn text_rect(output: &FullOutput, label: &str) -> Option<Rect> {
    output
        .shapes
        .iter()
        .rev()
        .find_map(|shape| match &shape.shape {
            Shape::Text(text) if text.galley.text() == label => {
```

`click` dibuja un fotograma, busca la etiqueta, y envía una pulsación y una liberación del puntero en su centro. Así un test se lee como una persona usando la aplicación:

```
fn f1_and_the_help_menu_show_the_keyboard_shortcuts() {
    let mut screen = Screen::new();
    let output = screen.draw(vec![]);
    assert!(text_rect(&output, "Keyboard shortcuts").is_none());

    screen.key(Key::F1);
    let output = screen.draw(vec![]);
    assert!(screen.app.shortcuts_open);
```

126 tests de ese archivo abren demostraciones, aplican estilos, hacen clic en el mezclador y el rack, cambian el tamaño de las ventanas y reinician la aplicación. Otros más pequeños están junto a un solo widget, como el selector de compás o la lista de reproducción, con una función auxiliar `frame` que dibuja solo ese widget en una ventana propia.

Trazas: lo que se tocó

Un test del secuenciador puede contar notas, pero el sonido de una canción es lo que el reproductor envía a los instrumentos, paso a paso, al tempo de cada momento. La traza de reproducción escribe exactamente eso, una línea por nota, tocando sin ventana como lo hace **Archivo > Exportar WAV…**. Es la herramienta para una pregunta que un test unitario no puede hacer con facilidad, como «¿esta demostración sigue sonando igual que antes?»: se traza antes y después de un cambio y se comparan las dos tablas. El MIDI se prueba igual, sin sintetizador: una canción se exporta a bytes, se vuelve a leer y se compara (ver Probar sin sintetizador).

Formato y lints

Dos herramientas más mantienen el código en orden, y la integración continua ejecuta las dos:

- `cargo fmt` reescribe el código en el estilo estándar de Rust, así nadie discute dónde van las llaves. `cargo fmt --all --check` solo dice si algo cambiaría.
- `cargo clippy` es el **linter** de Rust. Conoce cientos de patrones que compilan pero podrían ser más simples, más rápidos o más correctos, y explica cada uno. `-D warnings` convierte cada aviso en un error.

```
cargo fmt --all --check
cargo clippy --workspace --all-targets -- -D warnings
```

`--all-targets` revisa también los tests y los scripts de compilación. La integración continua instala el Rust estable más nuevo en cada ejecución, y un Rust nuevo a menudo trae lints nuevos, así que ejecuta `rustup update` antes de revisar: un lint que tu Clippy más viejo no conoce hará fallar la ejecución igual.

Tres sistemas e integración continua

Archivos: `.github/workflows/ci.yml`, `crates/lineup-app/src/main.rs`, `crates/lineup-app/build.rs` y `Cargo.toml`

Ferrous Lineup funciona en Linux, macOS y Windows con un solo código. Casi nada de ese código sabe en qué sistema está, y tres flujos de trabajo en GitHub comprueban cada cambio en los tres. Este capítulo trata de cómo un programa en Rust se mantiene portable, y de cómo se ejecutan esas comprobaciones.

Quién habla con el sistema operativo

Ferrous Lineup nunca llama por su cuenta al sistema operativo para el sonido, las ventanas o los diálogos. Lo hacen cuatro crates:

Tarea	Crate	Linux	macOS	Windows
Sonido	`cpal`	ALSA (también a través de PipeWire o PulseAudio)	Core Audio	WASAPI
Ventana, teclas y ratón	`winit`, dentro de `eframe`	X11 o Wayland	AppKit	Win32
Dibujo	`wgpu`, dentro de `eframe`	Vulkan u OpenGL	Metal	DirectX 12 o Vulkan
Diálogos de archivo	`rfd`	el portal del escritorio	los paneles de abrir y guardar del Finder	el diálogo del Explorador

Cada uno de estos crates tiene el código de todos los sistemas y se queda solo con el que corresponde al compilar. La aplicación llama a `cpal::default_host()`, dibuja con egui y abre un `rfd::FileDialog`; nada de eso nombra un sistema. Cada biblioteca se eligió por eso: algo que funcionara en un solo sistema habría que reemplazarlo para llegar a los demás.

El código propio del workspace nombra un sistema en cuatro lugares:

- `main.rs` esconde la ventana de consola en Windows (más abajo);
- `build.rs` pone el icono y la versión dentro del `.exe` de Windows;
- la partitura guía elige `cmd`, `open` o `xdg-open` para mostrar un PDF en el visor del sistema;
- la ventana de sonidos muestra una carpeta dentro de la carpeta personal como `~/...` solo en Linux y macOS, donde la gente escribe así las rutas.

Cómo un crate guarda el código de un solo sistema: `cfg`

Dentro de `cpal`, el host por defecto se elige así (abreviado):

```
#[cfg(any(target_os = "linux", target_os = "freebsd"))]
mod platform_impl {
    pub fn default_host() -> Host {
        AlsaHost::new().expect("the default host should always be
available").into()
    }
}

#[cfg(target_vendor = "apple")]
mod platform_impl {
    pub fn default_host() -> Host {
        CoreAudioHost::new().expect("the default host should always be
available").into()
    }
}
```

Dos módulos con el mismo nombre serían normalmente un error. Pero `#[cfg(...)]` es un atributo de **compilación condicional**: cuando su condición es falsa, el elemento que tiene debajo se quita antes de que el compilador siquiera lo mire. En Linux solo existe el primer módulo; en un Mac, solo el segundo. Las condiciones pueden preguntar por:

- el sistema para el que compilas: `target_os = "windows"`, `target_os = "macos"`, `target_family = "unix"` (o solo `unix`), `target_vendor = "apple"`;
- el tipo de compilación: `debug_assertions` está activo en las compilaciones de depuración, `test` mientras se ejecuta `cargo test` (cada `#[cfg(test)] mod tests` lo usa);
- las **features** de un crate: `feature = "wayland"`;
- y combinaciones, con `any(..)`, `all(..)` y `not(..)`.

Cargo puede hacer lo mismo con las dependencias. El `Cargo.toml` de `cpal` pide la biblioteca de ALSA solo en los sistemas que la tienen:

```
[target.'cfg(any(target_os = "linux", target_os =
"freebsd"))'.dependencies]
alsa = "0.11"
```

También hay una macro `cfg!(...)`, que da `true` o `false` como un valor corriente. La partitura guía la usa:

```
let mut command = if cfg!(target_os = "windows") {
```

La diferencia importa. Con `cfg!`, todas las ramas se compilan igual, así que todas tienen que ser válidas en todos los sistemas; con `#[cfg]`, el código de los otros sistemas no se compila nunca, y por eso puede llamar a funciones que solo existen allí. Ese es también su peligro: un error de tipeo en una función `#[cfg(windows)]` pasa inadvertido en Linux, porque el compilador nunca la lee. El código que no puedes compilar es código en el que no puedes confiar, y por eso conviene tenerlo dentro de unos pocos crates bien probados, y por eso la integración continua de más abajo compila en todos los sistemas.

Features: elegir las partes de un crate

La línea de `rfd` en el `Cargo.toml` del workspace apaga parte del crate:

```
rfd = { version = "0.17", default-features = false, features = ["xdg-
portal"] }
```

Una **feature** es una parte opcional de un crate, nombrada en su `Cargo.toml`. Algunas están activas salvo que digas lo contrario: las features `default`. `rfd` normalmente activa dos: `xdg-portal`, que le pide al escritorio de Linux que muestre su propio diálogo de archivos (por D-Bus, el sistema de mensajes del escritorio, así no hay que instalar GTK), y `wayland`, que le dice al portal a qué ventana pertenece el diálogo. Ferrous Lineup no se lo dice, así que `default-features = false` quita `wayland`, y `features = ["xdg-portal"]` deja la que hace falta. Menos features significa menos crates que compilar. En macOS y Windows la feature de Linux no cambia nada: los `#[cfg]` dentro de `rfd` dejan su código afuera.

Las features solo pueden *agregar*. Si dos crates de un programa usan `rfd`, uno con `wayland` y otro sin, Cargo lo compila una vez, con `wayland`. Así que una feature

nunca debe quitar nada: un crate con ella tiene que seguir haciendo todo lo que hacía sin ella.

Sin ventana de consola en Windows: `cfg_attr`

En Windows, cada programa es o un programa de *consola*, que abre una ventana de terminal al arrancar, o un programa de *ventanas*, que no la abre. Rust hace programas de consola salvo que se le diga otra cosa, así que hacer doble clic en `ferrous-lineup.exe` abriría una ventana negra vacía junto a la aplicación. Una línea al principio de `main.rs` dice otra cosa:

```
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]
```

- `windows_subsystem = "windows"` pide el segundo tipo. En los otros sistemas el atributo no hace nada.
- `#![...]`, con `!`, se aplica a aquello dentro de lo que está, aquí el crate entero. `#[...]` sin `!` se aplica al elemento siguiente.
- `cfg_attr(condition, attribute)` aplica el atributo solo cuando la condición es verdadera. `not(debug_assertions)` es verdadera en las compilaciones release.

Así `cargo run`, una compilación de depuración, conserva la consola, donde aparecen los `println!` y los mensajes de pánico mientras trabajas, y `cargo build --release` hace el programa silencioso que le das a otras personas. Esos mensajes entonces no van a ninguna parte en Windows, que es una razón más para que la aplicación muestre los errores de audio y los mensajes «No se pudo abrir» y «No se pudo guardar» dentro de su propia ventana.

Un script de compilación para el `.exe`

`crates/lineup-app/build.rs` es un **script de compilación**: Cargo lo compila y lo ejecuta antes de compilar el crate. En Windows pone el icono, la versión y el copyright dentro del `.exe`, para que el Explorador y la barra de tareas muestren el icono de la aplicación y la página de Propiedades del archivo su nombre:

```
#[cfg(windows)]
{
    let version = env!("CARGO_PKG_VERSION");
```

Aquí `#[cfg(windows)]` describe la computadora que compila, porque un script de compilación se ejecuta allí: el compilador de recursos al que llama es el de Microsoft, que solo tiene Windows. El flujo de trabajo de versiones compila el programa de Windows en Windows, así que el icono siempre está. macOS y Linux toman sus iconos de los paquetes y los archivos de escritorio de `packaging/` (ver Paquetes de cada versión).

Pequeñas diferencias que el código ya resuelve

- **Atajos:** `Modifiers::COMMAND` es Ctrl en Linux y Windows y ⌘ en macOS.
- **Rutas:** `PathBuf` y `Path` manejan `/` y `\`, las letras de unidad y los nombres de archivo que no son texto válido. `std::env::args_os` pasa el archivo que hay que abrir tal como lo dio el sistema.
- **Reemplazar un archivo:** `std::fs::rename` reemplaza un archivo que ya existe en todos los sistemas, y de eso depende guardar sin riesgo.
- **Sonido:** cpal le da a la función de audio de cada sistema el mismo `&mut [f32]`, así el hilo de audio nunca sabe dónde se ejecuta.

Tres flujos de trabajo

Un **flujo de trabajo** (workflow) es un archivo en `.github/workflows/` que le dice a GitHub qué ejecutar, en qué máquinas y cuándo. Ferrous Lineup tiene tres:

Flujo de trabajo	Se ejecuta en	Hace
CI (`ci.yml`)	cada pull request y cada push a main	revisa el formato y los lints, prueba y compila en Linux, macOS y Windows
Release (`release.yml`)	cada pull request, push a main y etiqueta `v`, y a mano	compila las descargas para cada sistema, y las publica para una etiqueta, subida con push o escrita en una ejecución a mano (ver Paquetes de cada versión)
Book (`book.yml`)	los cambios en los libros y los manuales	compila los sitios web y los PDF, y los publica desde main (ver Cómo se construyen los libros)

Un pull request muestra una marca para cada trabajo, verde si pasó y roja si falló, con un enlace a su registro.

Un trabajo, tres máquinas: la matriz

El flujo de trabajo CI tiene un solo trabajo, `check`, que se ejecuta tres veces:

```yaml
check:
  name: Build and test (${{ matrix.name }})
  strategy:
    fail-fast: false
    matrix:
      include:
        - name: Linux
          os: ubuntu-latest
          artifact: ferrous-lineup-linux
          binary: target/release/ferrous-lineup
        - name: macOS
          os: macos-latest
          artifact: ferrous-lineup-macos
          binary: target/release/ferrous-lineup
        - name: Windows
          os: windows-latest
          artifact: ferrous-lineup-windows
          binary: target/release/ferrous-lineup.exe
  runs-on: ${{ matrix.os }}
```

- Una **matriz** convierte un trabajo en varios. Cada entrada bajo `include` es una copia del trabajo, con sus propios valores para `name`, `os`, `artifact` y `binary`.
- `${{ matrix.os }}` se reemplaza por el valor de la entrada, así `runs-on` elige una máquina Linux, Mac o Windows, y el nombre del trabajo dice cuál («Build and test (macOS)»).
- `fail-fast: false` deja que las otras copias terminen cuando una falla. Sin eso, un fallo en Windows cancelaría el trabajo del Mac, y un segundo problema aparecería solo después de arreglar el primero.

Cada copia ejecuta después los mismos pasos:

1. **Instalar las bibliotecas del sistema**, solo en Linux: los archivos de desarrollo de ALSA, xkbcommon y Wayland. macOS y Windows traen los suyos.
2. **Rust**, el estable más nuevo, con `rustfmt` y `clippy`.
3. **Una caché** de las descargas de Cargo y de las dependencias compiladas (`Swatinem/rust-cache`), así una ejecución compila solo lo que cambió.
4. **Revisar el formato**, solo en Linux: el formato es igual en todas partes.
5. **Lint**: `cargo clippy --workspace --all-targets -- -D warnings`.

6. **Tests**: `cargo test --workspace`, en una compilación de depuración (ver Ejecutarlos).
7. **Compilar en release**, y subir el programa como artefacto de la ejecución.

Los pasos que solo tienen sentido en un sistema lo dicen:

```
- name: Install system libraries
  if: runner.os == 'Linux'
```

Clippy y los tests se ejecutan en los tres: clippy porque cada sistema compila un código un poco distinto, y los tests porque un test que pasa en un sistema puede fallar en otro (una ruta escrita con `/`, un archivo que no se puede renombrar mientras está abierto).

En la página de una ejecución, bajo **Artifacts**, están los tres programas. GitHub comprime cada uno en un zip, y el zip olvida que el programa era ejecutable, así que en Linux y macOS hace falta `chmod +x ferrous-lineup` después de descomprimirlo. Las descargas de verdad, con instaladores, salen del flujo de trabajo Release.

Comprobar el código de Windows desde Linux

Esperar a que el CI encuentre un error de Windows es lento. Rust puede comprobar el código para otro sistema sin salir de Linux:

```
rustup target add x86_64-pc-windows-gnu
cargo clippy --workspace --all-targets --target x86_64-pc-windows-gnu -- -
D warnings
```

Un **target** es el sistema *para* el que compilas, a diferencia de aquel *en* el que compilas. `rustup target add` descarga la biblioteca estándar de Rust compilada para Windows. `cargo clippy` y `cargo check` solo ejecutan las comprobaciones del compilador y nunca enlazan un programa, así que no necesitan ni el enlazador de Windows ni sus bibliotecas del sistema. Eso atrapa la mayoría de los errores, un `#[cfg]` solo de Windows que no compila o un crate que no admite Windows en absoluto, aunque no lo que solo pasa cuando el programa se ejecuta. Lo mismo funciona para los Mac con `aarch64-apple-darwin` (los procesadores de Apple) o `x86_64-apple-darwin` (los Mac con Intel), siempre que ninguna dependencia necesite las herramientas propias de Apple para compilar.

Lo que las comprobaciones no pueden demostrar

Las máquinas del CI no tienen tarjeta de sonido ni pantalla. Los trabajos demuestran que el programa compila y que la lógica musical es correcta en cada sistema, no que se oiga algo: los tests de ventana dibujan en memoria (ver Tests de ventana), y los tests de audio renderizan en un búfer. Ejecutarlo en un Mac o una computadora con Windows de verdad sigue siendo la prueba real del sonido y del dibujo.

Paquetes de cada versión

Archivos: `.github/workflows/release.yml` y las carpetas de `packaging/`

La mayoría de quienes usen Ferrous Lineup nunca van a instalar Rust. El flujo de trabajo **Release** compila el programa para cuatro tipos de computadora y prepara siete descargas:

Paquete	Sistema
`ferrous-lineup-…-linux-x86_64.AppImage`	Linux en una PC (Intel o AMD): un solo archivo, nada que desempaquetar
`ferrous-lineup-…-linux-x86_64.tar.gz`	Linux en una PC, como una carpeta corriente
`ferrous-lineup_…_amd64.deb`	Debian, Ubuntu y sus derivadas
`ferrous-lineup-…-windows-x86_64-setup.exe`	Windows en una PC, con desinstalador y la asociación de archivos `.lineup`
`ferrous-lineup-…-windows-x86_64.zip`	Windows en una PC, portable: descomprimir y ejecutar
`ferrous-lineup-…-macos-arm64.dmg`	Mac con Apple Silicon (M1 y posteriores)
`ferrous-lineup-…-macos-x86_64.dmg`	Mac con procesador Intel

Los `…` son la versión en una publicación (`v0.4.1`), o los siete primeros caracteres del commit en cualquier otra compilación (`1a2b3c4`), así una descarga siempre dice qué es. Cómo abrir cada una en su sistema está en Instalarlo, en el manual; este capítulo trata de cómo se hacen.

Dónde están las descargas

- **Cada ejecución** del flujo de trabajo las produce: cada pull request, cada push a main, y cada ejecución a mano (**Actions > Release > Run workflow** compila cualquier rama). Los paquetes son los **artefactos** (Artifacts) de la ejecución, al final de su página. Los artefactos necesitan una cuenta de GitHub para descargarlos, y GitHub los borra a los 90 días. Así un pull request se puede probar en un Mac o en Windows antes de fusionarlo.

- **Una etiqueta** que empieza con `v` también publica los paquetes en la página **Releases** del repositorio, con un `SHA256SUMS.txt` al lado. Una publicación se queda ahí para siempre, y cualquiera puede descargarla. La etiqueta se puede subir con un push, o el flujo de trabajo puede crearla él mismo en una ejecución a mano (ver más abajo).

Publicar una versión

Una versión es una etiqueta: un nombre pegado a un commit. Hacer la versión 0.5.0 lleva un pull request y una etiqueta.

El pull request de la versión cambia:

- la versión en el `Cargo.toml` raíz, bajo `[workspace.package]`, que cada crate toma con `version.workspace = true`;
- `Cargo.lock`, que lista la versión de cada crate, los nuestros también (un `cargo build` lo actualiza);
- las notas de la versión en el archivo AppStream, `packaging/linux/io.github.naelolaiz.FerrousLineup.metainfo.xml`, en inglés y en español, que muestran los centros de software;
- la versión por defecto en el script del instalador de Windows, `packaging/windows/ferrous-lineup.iss`;
- la versión en las recetas de AUR y la etiqueta en el manifiesto de Flathub;
- si la publicación debe empezar con unas palabras propias, un archivo `packaging/release-notes/v0.5.0.md` que las contenga, en Markdown.

La etiqueta, una vez fusionado eso, llega de una de dos maneras. Desde un clon, con un push:

```
git fetch origin
git tag -a v0.5.0 -m "Ferrous Lineup 0.5.0" origin/main
git push origin v0.5.0
```

O, en GitHub, se abre **Actions > Release > Run workflow**, se deja la rama en `main`, se escribe `v0.5.0` en el campo **tag** y se ejecuta. Esto solo necesita el permiso de ejecutar flujos de trabajo, no el de hacer push al repositorio, y el flujo de trabajo crea la etiqueta él mismo, en el commit que compiló.

De cualquiera de las dos maneras, cuando todos los trabajos de compilación pasaron, aparece la publicación, llamada «Ferrous Lineup v0.5.0». Sus notas son el archivo de notas, si lo hay, seguido de la lista de pull requests fusionados desde la versión

anterior que escribe GitHub; después se pueden editar en GitHub. Un último pull request pequeño fija lo que solo existe una vez que existe la etiqueta: la suma de comprobación del archivo de la etiqueta que hace GitHub, en la receta de AUR, y el commit de la etiqueta en el manifiesto de Flathub.

Tres detalles:

- Una etiqueta con un guion, como `v0.5.0-beta.1`, hace una **prepublicación** (pre-release), que nunca se muestra como la última. Así escribe el versionado semántico una versión que todavía no es definitiva, y Cargo lee las versiones de la misma manera.
- Si una etiqueta subida con push y `Cargo.toml` no coinciden, la publicación se hace igual, con un aviso en la página de la ejecución: la etiqueta ya existe, y detenerse solo la dejaría sin descargas. La etiqueta decide los nombres de los archivos, pero **Ayuda > Acerca de Ferrous Lineup** dice lo que decía `Cargo.toml`.
- Una ejecución a mano comprueba antes de crear nada: si la etiqueta pedida no es `v` más la versión de `Cargo.toml`, o la ejecución está en otra rama que main, el trabajo de publicación se detiene con un error, y no se crea ninguna etiqueta.

Targets: para qué computadora es un programa

Un programa compilado es código máquina para un tipo de procesador, en el formato de archivo de un sistema operativo. Rust nombra cada combinación con una **target triple**, arquitectura-fabricante-sistema(-entorno):

Target	Procesador	Sistema
`x86_64-unknown-linux-gnu`	Intel o AMD de 64 bits	Linux, con la biblioteca C de GNU
`x86_64-pc-windows-msvc`	Intel o AMD de 64 bits	Windows, con las herramientas de Microsoft
`aarch64-apple-darwin`	ARM de 64 bits (Apple Silicon)	macOS («Darwin» es su núcleo)
`x86_64-apple-darwin`	Intel de 64 bits	macOS

`rustc --version --verbose` muestra el target de tu propia computadora en su línea `host:`. `cargo build --target <triple>` compila para otro, una vez que `rustup target add <triple>` descargó la biblioteca estándar para él, y deja el programa en

`target/<triple>/release/`, así las compilaciones para distintos targets nunca se mezclan.

En el flujo de trabajo cada target es una línea de una matriz (ver Un trabajo, tres máquinas):

```
- name: Linux x86_64
  os: ubuntu-22.04
  target: x86_64-unknown-linux-gnu
  platform: linux-x86_64
  appimage: true
```

y la compilación es

```
cargo build --release --locked --target ${{ matrix.target }}
```

`--locked` compila exactamente las versiones de las bibliotecas que dice `Cargo.lock`, y se detiene con un error si el archivo tuviera que cambiar. Una versión se debe compilar a partir de lo que se probó, no de las versiones más nuevas que haya el día de la etiqueta. Por eso también el pull request de la versión tiene que incluir `Cargo.lock`: con el archivo viejo, `--locked` se niega a compilar.

El Mac con Intel, compilado en un Mac con Apple Silicon

Las máquinas macOS de GitHub son Apple Silicon. Compilar para otro procesador que aquel en el que compilas es **compilación cruzada**, y en un Mac es fácil: los compiladores y las bibliotecas del sistema de Apple cubren los dos procesadores. Así que el trabajo de Intel corre en el mismo tipo de máquina que el de Apple Silicon; solo cambia su target. Los trocitos de C de la compilación, como la criptografía de `ring` (para descargar el SoundFont por HTTPS), los compila el crate `cc`, que le pasa el target al compilador de C de Apple. El flujo de trabajo no puede ejecutar el programa de Intel, y no hace falta: los tests se ejecutan en el CI, y el código es el mismo.

Por qué Linux compila en un Ubuntu más viejo

Un programa de Linux llama a la biblioteca C, glibc, para los archivos, la memoria y los hilos. Cada función de glibc lleva una versión, y un programa anota la más nueva que usó: compilado en Ubuntu 24.04, Ferrous Lineup pediría `GLIBC_2.39`. Una glibc más vieja se niega a arrancarlo («version `GLIBC_2.39' not found»), mientras que una más nueva lo ejecuta sin problema. Así que la compilación de Linux corre en el Ubuntu

más viejo que ofrece GitHub, 22.04, con glibc 2.35, y el programa funciona en Ubuntu 22.04, Debian 12 y cualquier cosa más nueva.

`ldd` lista las bibliotecas que un programa carga al arrancar:

```
$ ldd target/release/ferrous-lineup
    libasound.so.2 => /lib/x86_64-linux-gnu/libasound.so.2
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

Solo está ALSA. Las bibliotecas de ventanas (Wayland o X11, xkbcommon para el teclado, OpenGL) no faltan: los crates las abren mientras el programa se ejecuta, con `dlopen`, y usan las que tenga el escritorio. Por eso una sola compilación de Linux funciona en escritorios Wayland y X11.

El runtime de C dentro del programa de Windows

Los programas de Rust en Windows usan el runtime de C de Microsoft para algunas cosas básicas. Por defecto es una DLL aparte que viene con el «Visual C++ Redistributable», que tienen la mayoría de las computadoras con Windows, pero no todas. La línea de Windows de la matriz agrega una opción del compilador:

```
rustflags: -C target-feature=+crt-static
```

`crt-static` copia el runtime dentro del `.exe` (que crece un poco), así el programa arranca en un Windows que nunca instaló nada. La variable de entorno `RUSTFLAGS` le pasa opciones a cada `rustc` que ejecuta Cargo; en las otras líneas está vacía.

Empaquetar

Para Linux y Windows, un paso **Package** copia el programa, el README y la licencia en una carpeta con el nombre de la etiqueta, y la empaqueta:

```
if [[ -n "$RELEASE_TAG" ]]; then
  label="$RELEASE_TAG"
else
  label="${COMMIT::7}"
fi
name="ferrous-lineup-$label-${{ matrix.platform }}"
```

`RELEASE_TAG` se define una vez, para todos los trabajos, al principio del archivo:

```
env:
  RELEASE_TAG: ${{ github.ref_type == 'tag' && github.ref_name ||
inputs.tag }}
```

Las expresiones de GitHub no tienen «si … si no», así que `a && b || c` hace sus veces: cuando la ejecución es por una etiqueta subida con push, el valor es el nombre de la etiqueta; si no, es el campo **tag** de una ejecución a mano, que está vacío en cualquier otra ejecución (`inputs` no tiene nada cuando nadie llenó un formulario). En Bash, `-n` es verdadero para un texto que no está vacío.

`${COMMIT::7}` es Bash para «los primeros siete caracteres». `COMMIT` es `${{ github.event.pull_request.head.sha || github.sha }}`: para un pull request GitHub construye un commit de fusión propio, que nadie ve nunca, así que la etiqueta toma el último commit de la rama; `||` elige el primer valor que no esté vacío. Windows recibe un `.zip`, que abre por sí solo. Linux recibe un `.tar.gz`, porque un archivo tar conserva el permiso de «ejecutable» del programa.

Para macOS, **Package for macOS** construye un `Ferrous Lineup.app` de verdad: el programa en `Contents/MacOS`, `Info.plist` y el icono de `packaging/macos/`, y la versión de `Cargo.toml` escrita en el plist. Lo firma **ad hoc** (`codesign --force --sign -`), lo pone en una carpeta con un enlace a `/Applications`, y `hdiutil` hace el `.dmg`, así la gente arrastra la aplicación a Aplicaciones.

El plist declara además el **tipo de documento** `.lineup`: una entrada `CFBundleDocumentTypes` que nombra el tipo que abre la aplicación, y una entrada `UTExportedTypeDeclarations` que define ese tipo, `io.github.naelolaiz.FerrousLineup.project`, como los archivos que terminan en `.lineup`. Con las dos, el Finder abre un proyecto con la aplicación. En Linux y Windows el gestor de archivos arranca el programa con la ruta del archivo en la línea de comandos, que lee `main`. macOS lo arranca sin argumentos y manda los archivos en un evento de Apple, «abrir documentos» (`aevt` / `odoc`), que llega también a una aplicación que ya está abierta. El manejador propio de AppKit para ese evento le pide al delegado de la aplicación que abra los archivos; el delegado es de winit, la biblioteca de ventanas, y no abre nada. Así que `open_files.rs` pone su propio manejador, en el momento que dan las reglas de Apple, cuando la aplicación está por terminar de arrancar (después de que AppKit pone el suyo, antes de que llegue el evento del arranque). El manejador solo junta las rutas:

```
#[unsafe(method(openDocuments:withReplyEvent:))]
fn open_documents(
    &self,
    event: &NSAppleEventDescriptor,
    _reply: &NSAppleEventDescriptor,
) {
    // SAFETY: `paramDescriptorForKeyword:` takes a four-letter
    // code and gives a descriptor or nil.
    let files: Option<Retained<NSAppleEventDescriptor>> =
        unsafe { msg_send![event, paramDescriptorForKeyword:
DIRECT_OBJECT] };
    let Some(files) = files else {
        return;
    };
    // A list of files, counted from 1, or a single one.
    let count = files.numberOfItems();
    let items: Vec<Retained<NSAppleEventDescriptor>> = if count == 0 {
        vec![files]
    } else {
        (1..=count)
            .filter_map(|index| files.descriptorAtIndex(index))
            .collect()
    };
    super::arrived(
        items
            .iter()
            .filter_map(|item| item.fileURLValue())
            .filter_map(|url| url.to_file_path()),
    );
}
```

`objc2` llama a Objective-C desde Rust: `define_class!` crea una clase cuyos métodos AppKit puede llamar, y `msg_send!` manda un mensaje por su nombre, que el compilador no puede comprobar, de ahí el `unsafe`. `arrived` guarda las rutas en un `Mutex<Vec<PathBuf>>` y le pide a la ventana que se vuelva a dibujar; `handle_dropped_files` las toma en el cuadro siguiente y abre el archivo como si lo hubieran soltado sobre la ventana. Todo está bajo `#[cfg(target_os = "macos")]`, así que en Linux y Windows `listen` no hace nada.

Cada paquete se sube con `archive: false`:

```
- uses: actions/upload-artifact@v7
  with:
    name: ${{ env.NAME }}
    path: ${{ env.PACKAGE }}
    archive: false
```

Por defecto un artefacto se comprime en zip al subirlo, así que un `.tar.gz` se descargaría como un zip que contiene un `.tar.gz`. Con `archive: false` el archivo

se guarda tal cual, y el artefacto lleva el nombre del archivo.

La AppImage

Una **AppImage** es un programa y lo que necesita, empaquetados en un solo archivo que arranca como un programa. El archivo son dos cosas pegadas: un pequeño programa de arranque, el **runtime**, seguido de una imagen de disco comprimida de una carpeta, el **AppDir**. Al arrancarla, el runtime monta la imagen como una carpeta (con FUSE, que permite que un programa normal ofrezca un sistema de archivos) y arranca el programa que hay dentro. No se instala nada, así que borrar el archivo lo desinstala. El AppDir está organizado como el `/usr` de un sistema:

```
AppDir/
├── AppRun -> usr/bin/ferrous-lineup          lo que arranca el runtime
├── io.github.naelolaiz.FerrousLineup.desktop -> usr/share/…  la entrada
del menú
├── io.github.naelolaiz.FerrousLineup.svg -> usr/share/…      el icono
└── usr/
    ├── bin/ferrous-lineup
    ├── lib/libxkbcommon.so.0, libxkbcommon-x11.so.0, libxcb-xkb.so.1
    └── share/applications/, icons/, mime/, metainfo/
```

La **entrada de escritorio** (`packaging/linux/io.github.naelolaiz.FerrousLineup.desktop`) es el archivo de texto que lee cada menú de Linux: el nombre de la aplicación, su icono, su categoría y cómo arrancarla (`Exec=ferrous-lineup %f`, donde `%f` es un archivo que abrir). El archivo de tipo MIME le enseña al escritorio qué es un archivo `.lineup`, y el archivo **metainfo** de AppStream describe la aplicación y sus versiones para los centros de software. El flujo de trabajo revisa los tres con `desktop-file-validate` y `xmllint`, porque un error de tipeo en ellos falla en silencio en un escritorio.

`appimagetool` convierte el AppDir en la AppImage. Él y el runtime se descargan fijados a una versión y se comprueban contra el SHA-256 escrito en el flujo de trabajo:

```
echo "$RUNTIME_SHA256  runtime" | sha256sum --check
```

Si alguien reemplazara el archivo en GitHub, la suma de comprobación no coincidiría y la compilación se detendría, en lugar de publicar lo que fuera que el archivo se hubiera vuelto.

Llevar una biblioteca consigo

En un sistema sin `libxkbcommon-x11`, el programa se detiene antes de abrir una ventana. `ldd` no lista esa biblioteca, porque winit la abre con `dlopen` cuando se crea la ventana. Una biblioteca abierta así se busca como cualquier otra: en las carpetas de `LD_LIBRARY_PATH`, luego en la **RUNPATH** escrita en el programa, luego en las carpetas del sistema. Así que el flujo de trabajo copia las tres bibliotecas del teclado en `AppDir/usr/lib` y escribe una RUNPATH con `patchelf`:

```
patchelf --set-rpath '$ORIGIN/../lib' AppDir/usr/bin/ferrous-lineup
patchelf --set-rpath '$ORIGIN' AppDir/usr/lib/libxkbcommon-x11.so.0
```

`$ORIGIN` quiere decir «la carpeta en la que está este archivo», calculada cuando el programa arranca, así la ruta vale dondequiera que se monte la AppImage. Las comillas simples impiden que la shell reemplace `$ORIGIN` por una variable vacía. Lo que la AppImage **no** lleva: glibc, ALSA y los controladores gráficos. Esos tienen que corresponder a la computadora, y la comunidad de AppImage mantiene una lista de las bibliotecas que una AppImage nunca debe llevar.

El paquete de Debian

El `.deb` lo construye `cargo-deb`, fijado a una versión, a partir del mismo programa y la misma entrada de escritorio, icono, tipo MIME y archivo AppStream:

```
cargo deb -p lineup-app --no-build --target x86_64-unknown-linux-gnu
```

`--no-build` toma el programa que la matriz acaba de compilar. La tabla `[package.metadata.deb]` de `crates/lineup-app/Cargo.toml` lista los archivos y aquello de lo que depende el paquete (ALSA, Wayland, xkbcommon, el portal del escritorio y las herramientas que registran entradas de menú, iconos y tipos MIME), así `apt` también los instala.

El instalador de Windows

El programa de instalación lo hace **Inno Setup**, que las máquinas Windows de GitHub ya tienen, a partir del script `packaging/windows/ferrous-lineup.iss`. El flujo de trabajo le pasa la versión y la etiqueta:

```
"$iscc" /DAppVersion="$version" /DBuildLabel="$LABEL"
packaging/windows/ferrous-lineup.iss
```

El script instala el programa para quien lo ejecuta, sin necesitar permisos de administrador, y agrega una entrada en el menú Inicio, un acceso directo opcional en el escritorio, un desinstalador y la asociación de archivos `.lineup`.

Publicar: un trabajo que espera a los demás

El segundo trabajo, `release`, se ejecuta solo para una etiqueta subida con push o una ejecución a mano con etiqueta, y solo cuando todos los trabajos de compilación pasaron:

```
release:
  name: Publish the release
  if: github.ref_type == 'tag' || (github.event_name ==
'workflow_dispatch' && inputs.tag != '')
  needs: build
```

`workflow_dispatch` es el nombre que GitHub le da a una ejecución a mano. El formulario que muestra sale del disparador al principio del archivo:

```
workflow_dispatch:
  inputs:
    tag:
      description: "Publish a release with this tag (v0.4.1), made on the
commit built. Empty: only build."
      required: false
      default: ""
```

`needs: build` espera a cada línea de la matriz. Si una compilación falla, no hay publicación en absoluto, en lugar de una publicación sin Windows.

Los `permissions` del trabajo se refieren al **token** que recibe cada ejecución, `GITHUB_TOKEN`, una contraseña que caduca cuando termina el trabajo. Al principio del archivo, `permissions: contents: read` deja que cada trabajo lea el código y nada más, así un paso de compilación no puede cambiar el repositorio. Solo este trabajo pide `contents: write`, que necesita una publicación.

Descarga los paquetes, escribe sus sumas de comprobación con `sha256sum`, y se lo pasa todo a `gh`, la herramienta de línea de comandos de GitHub:

```
if gh release view "$TAG" > /dev/null 2>&1; then
  gh release upload "$TAG" release/* --clobber
else
```

- `gh release view` tiene éxito cuando la publicación ya existe, hecha en el sitio web o por una ejecución anterior. Entonces los archivos se suben a ella, y `--clobber` reemplaza los que tengan el mismo nombre.
- Si no, `gh release create` la crea, con `--generate-notes` para las notas. Para una etiqueta subida con push agrega `--verify-tag`, que se detiene si la etiqueta no está en GitHub en lugar de crear una etiqueta nueva en el commit más reciente. Para una ejecución a mano agrega `--target "$GITHUB_SHA"`, y GitHub crea la etiqueta justo en el commit con que se compilaron los paquetes, aunque main haya avanzado desde entonces.
- `--notes-file packaging/release-notes/<etiqueta>.md` se agrega cuando ese archivo existe; `gh` pone su texto antes de las notas generadas.
- `--prerelease` se agrega cuando la etiqueta tiene un guion.

Una etiqueta creada con el propio `GITHUB_TOKEN` de la ejecución no pone en marcha ningún otro flujo de trabajo: lo que una ejecución hace con su propio token no inicia ejecuciones nuevas (solo pasa una «ejecución a mano» pedida explícitamente por la API de GitHub), para que un flujo de trabajo no pueda dispararse a sí mismo sin fin. Por eso la ejecución que crea la etiqueta es la que la publica; ninguna otra lo hará.

Una **suma de comprobación** (checksum) es una huella de 64 caracteres de los bytes de un archivo: cambia un byte y la huella es completamente distinta. `SHA256SUMS.txt` le permite a cualquiera comprobar que una descarga llegó entera.

Probar un flujo de trabajo antes de fusionarlo

Un flujo de trabajo de publicación solo hace su trabajo de verdad con una etiqueta, que es un mal momento para encontrar un error de tipeo. Dos cosas atrapan los errores antes:

- Cada pull request lo ejecuta, así un cambio que rompe la compilación de Windows o del Mac aparece en su pull request. Solo se salta el trabajo `release`, porque un pull request no es una etiqueta.

- actionlint revisa los archivos de flujos de trabajo y, con ShellCheck, los scripts de shell que contienen:

```
pip install actionlint-py shellcheck-py
actionlint .github/workflows/release.yml
```

`concurrency` deja una sola ejecución por rama: cuando un pull request recibe un push nuevo, se cancela la ejecución anterior, porque la nueva la contiene. Las publicaciones nunca se cancelan, ni por una etiqueta subida con push ni por una ejecución a mano con etiqueta:

```yaml
concurrency:
  group: release-${{ github.ref }}-${{ inputs.tag }}
  cancel-in-progress: ${{ github.ref_type != 'tag' && !inputs.tag }}
```

La etiqueta forma parte del nombre del grupo, así un push a main mientras se compila una publicación desde main pone en marcha su propia ejecución en lugar de esperar detrás de ella. `!inputs.tag` es verdadero cuando el campo está vacío.

Otras puertas de entrada: AUR y Flathub

En `packaging/` hay dos recetas más, para gestores de paquetes que compilan el programa ellos mismos a partir del código:

- `packaging/aur/` tiene dos **PKGBUILD** para el repositorio de usuarios de Arch Linux: `ferrous-lineup`, compilado a partir de la etiqueta de una versión, y `ferrous-lineup-git`, compilado a partir de main.
- `packaging/flathub/` tiene un manifiesto de **Flatpak** para Flathub, con `cargo-sources.json`, cada crate de `Cargo.lock` con su URL y su suma de comprobación: Flathub compila sin red, así que cada descarga tiene que estar listada de antemano. `generate-sources.py` vuelve a escribir ese archivo después de que cambie `Cargo.lock`.

Las dos instalan la misma entrada de escritorio, icono, tipo MIME y archivo AppStream que la AppImage y el `.deb`. Sus README explican cómo publicarlas.

Lo que los paquetes todavía no hacen

- Los programas no están **firmados** del todo. La aplicación de macOS tiene una firma ad hoc, que Apple Silicon exige, pero no está **notarizada**: eso necesita un Developer ID de Apple de pago, así que Gatekeeper pregunta la primera vez. Los

programas de Windows no están firmados, así que SmartScreen avisa. Instalarlo, en el manual, muestra cómo pasar cada aviso.

Cómo se construyen los libros

Archivos: `.github/workflows/book.yml`, `book-es/book.toml`, `book-es/print.css`, `book-es/print-pdf.py` y `book-es/index-words.txt`

En el repositorio viven cuatro libros, junto al código que describen:

Carpeta	Libro	En la web
`book/`	este libro, en inglés	https://naelolaiz.github.io/ferrous-lineup/
`book-es/`	este libro	https://naelolaiz.github.io/ferrous-lineup/es/
`manual/`	el manual de usuario, en inglés	https://naelolaiz.github.io/ferrous-lineup/manual/
`manual-es/`	el manual de usuario	https://naelolaiz.github.io/ferrous-lineup/es/manual/

Cada uno es un sitio web y un PDF con índice de contenido, números de página, índice alfabético y marcadores. Un cambio en la aplicación y las páginas que lo describen van en el mismo pull request, o en un pull request del libro justo después, así los libros nunca describen un programa que ya no existe.

mdBook

mdBook convierte una carpeta de archivos Markdown en un sitio web. `book.toml` guarda la configuración, y `src/SUMMARY.md` el orden de los capítulos, que se convierte en la barra lateral:

```
# Índice

[Introducción](introduction.md)

[Compilarlo desde el código fuente](building.md)

# Parte I: La forma del programa

- [Cinco crates, y qué puede saber cada uno](program/crates.md)
```

Un enlace en una línea propia, antes de la primera parte, es un capítulo sin número, como la introducción. Una línea que empieza con `#` es el título de una parte, y cada `- [...](...)` un capítulo numerado (uno con sangría sería un capítulo dentro del de arriba). `mdbook build` escribe el sitio web en la carpeta `book/` del libro, y `mdbook serve --open` lo construye, lo abre, y lo vuelve a construir cada vez que se guarda una página: la manera de leer una página antes de subirla.

`site-url` en `book.toml` es la ruta desde la que GitHub Pages sirve el libro, `/ferrous-lineup/es/` (o `/ferrous-lineup/` en inglés). La mayoría de las páginas encuentran sus estilos con enlaces relativos, pero la página de «no encontrada» se puede mostrar en cualquier dirección, así que necesita saber dónde empieza el libro.

Una página que se mueve conserva su dirección vieja. `book.toml` lista las rutas viejas, cada una con adónde fue su página, y mdBook escribe en la ruta vieja una pequeña página que lleva al navegador a la nueva:

```
[output.html.redirect]
"/step-9/band.html" = "../band/band.html"
```

El flujo de trabajo

El flujo de trabajo **Book** tiene dos trabajos. **build** se ejecuta en cada push a main y en cada pull request que cambia un libro, un manual o el propio flujo de trabajo; **deploy** se ejecuta solo después de un push a main, o cuando se lanza a mano (**Actions > Book > Run workflow**).

```
pull_request:
  paths:
    - "book/**"
    - "book-es/**"
    - "manual/**"
    - "manual-es/**"
    - ".github/workflows/book.yml"
```

`paths` es la razón por la que un cambio en el código de la aplicación no construye los libros, y una página del libro no espera los tests de la aplicación. `pull_request` no nombra ninguna rama, así que también se construye una página del libro apilada sobre otro pull request.

1. **mdBook**, fijado a una versión, lo descarga ya compilado `taiki-e/install-action`, que comprueba su suma de comprobación. La misma versión en cada ejecución construye los mismos libros, cambie como cambie mdBook después.

2. **Los sitios web:** `mdbook build` en cada una de las cuatro carpetas.
3. **Tipografías.** Los libros escriben silencios y notas con símbolos musicales (𝄾, ♩) y tienen algunos emoji. Una tipografía sin ellos imprime un recuadro vacío, así que el trabajo instala FreeSerif y Noto Color Emoji.
4. **pypdf**, una biblioteca de Python para archivos PDF, también fijada. El Python de Ubuntu no instala paquetes para todo el sistema, así que va en un **entorno virtual**: una carpeta con un Python y paquetes propios.
5. **Los PDF.** mdBook también escribe `print.html`, todos los capítulos en una página, que es lo que abre el botón de imprimir del sitio web. Cada libro se construye una segunda vez, en `pdf/`, con el tema claro, porque el sitio web es oscuro pero el papel es blanco. mdBook lee una opción de una variable de entorno con su nombre, así que `MDBOOK_OUTPUT__HTML__DEFAULT_THEME=light` reemplaza `default-theme` bajo `[output.html]` solo para esa construcción (`__` entre los niveles). Después `print-pdf.py` imprime esa página en un PDF.
6. **Subidas.** Cada PDF es una descarga de la ejecución, también en los pull requests. Los cuatro sitios web, con sus PDF, se juntan en un solo sitio: el libro en inglés en la raíz, el español bajo `es/`, y cada manual bajo `manual/` en su idioma.
7. **Publicar.** `deploy-pages` le entrega el sitio a GitHub Pages. Ese trabajo es el único que pide `pages: write` e `id-token: write`, para publicar y para demostrar qué flujo de trabajo publica; todos los demás trabajos solo leen el código. Se hace una publicación a la vez (`concurrency: group: pages`): una más nueva espera en lugar de cancelar otra a medias.

Orientarse en el PDF

La página de impresión, impresa tal cual, son cientos de páginas sin números, sin índice de contenido y sin índice alfabético. En el sitio web la barra lateral y el cuadro de búsqueda encuentran un capítulo; en papel no lo encontraría nada. `print-pdf.py` agrega lo que necesita un libro impreso:

Qué	Hecho a partir de
Una portada y el **índice de contenido**, cada capítulo con la página donde empieza	`SUMMARY.md`
Un **número de página** al pie de cada página	`print.css`
Un **índice alfabético** al final: palabras, y las páginas que las explican	los glosarios de música y de Rust, e `index-words.txt`

Qué	Hecho a partir de
Marcadores, el esquema que muestra un lector de PDF junto a las páginas	las partes, los capítulos y sus secciones

Números de página: CSS para el papel

Cuando un navegador imprime, vuelca la página en hojas de papel, y la regla `@page` dice cómo son esas hojas:

```
@page {
    size: A4;
    margin: 18mm 16mm;

    @bottom-center {
        content: counter(page);
        font-family: "Open Sans", sans-serif;
        font-size: 9pt;
        color: #555;
    }
}
```

Dentro de ella, las *cajas de margen* nombran los espacios alrededor del texto: `@bottom-center` es el centro del margen inferior, y `counter(page)` el número de la página que se imprime. La portada no tiene número: el script agrega una regla solo para la primera página, `@page :first`.

El índice de contenido: imprimir dos veces

El índice de contenido tiene que decir en qué página empieza cada capítulo antes de imprimir el libro, pero solo imprimir decide dónde cae un capítulo: la longitud de cada página anterior depende de las tipografías, las imágenes y dónde se corta cada línea. El script hace lo que hace LaTeX: imprime dos veces.

1. La primera vez, cada número de página del índice de contenido y del índice alfabético es `0000`.
2. El script lee en ese PDF dónde cayó cada capítulo y cada palabra del índice.
3. La segunda vez, los números de verdad ocupan el lugar de los blancos. `0000` es tan ancho como cualquier número de página del libro, así que ninguna línea crece, nada se mueve, y los números son correctos. El script lee también ese PDF, y si algo se movió, vuelve a imprimir, hasta cuatro veces.

Dónde cayó un capítulo se lee a través de sus enlaces. mdBook le da a cada título un `id` y un enlace a sí mismo, y cuando Chrome imprime una página con un enlace a `#la-hoja-de-acordes`, escribe un **destino con nombre** en el PDF: el nombre, la página, y dónde en la página. pypdf los vuelve a leer:

```
def destinations(reader: PdfReader) -> dict[str, tuple[int, object]]:
    found = {}
    for name, dest in reader.named_destinations.items():
        id = urllib.parse.unquote(name.lstrip("/"))
        found[id] = (reader.get_destination_page_number(dest), dest)
    return found
```

Los nombres vuelven como `/la-hoja-de-acordes`, con cada letra fuera de ASCII escrita como códigos `%` (la `ó` de `canción` es `%C3%B3`), así que `unquote` los convierte de nuevo en los `id` del HTML. Chrome solo escribe un destino para un `id` al que va algún enlace. Cada título se enlaza a sí mismo, y el índice de contenido y el alfabético enlazan a lo que listan; cuando el índice alfabético muestra un solo número para dos lugares de la misma página, una lista oculta de enlaces (`<div hidden>`) sigue apuntando a los dos.

El índice alfabético

Los libros ya tienen listas de palabras: sus glosarios, tablas con la palabra en negrita, lo que significa, y un enlace «Se explica en». El script los lee de `print.html` y le da a cada palabra:

- la página adonde va su enlace «Se explica en», en **negrita**;
- cada página donde un capítulo escribe la palabra en negrita, que es como estos libros presentan una palabra. Cada una de esas palabras en negrita recibe un `id`, como `pdf-ix-17`, para que el índice la enlace.

El plural también cuenta, y una palabra que es toda código solo coincide con código: el `windows` del glosario, un método de Rust, no es el Windows en negrita de «Linux, macOS y Windows». `index-words.txt` agrega las palabras que no tiene ningún glosario, sobre todo las partes de la aplicación, con los capítulos que las explican:

```
Hoja de acordes: band/chord-sheet.md, ...
Modulación: véase Cambio de tonalidad
Función: no program/sync.md
```

- Un lugar es un capítulo, por el archivo que enlaza `SUMMARY.md`, o una sección suya, por el `id` de su título después de `#`.
- `véase` manda al lector a otra entrada.

- `no` deja afuera un capítulo donde la palabra en negrita quiere decir otra cosa: ahí «función» es una función de Rust, no lo que hace un acorde en una tonalidad.

Una línea que nombra un capítulo o una sección que el libro no tiene detiene el script con un error, así un título renombrado hace fallar el flujo de trabajo en lugar de mandar a los lectores a la página equivocada.

Marcadores y enlaces

Chrome no escribe un esquema para esta página, así que el script agrega uno con pypdf después de la última impresión: las partes arriba, cada capítulo bajo su parte, y las secciones `##` de un capítulo bajo el capítulo.

```
writer = PdfWriter(clone_from=reader)
add_bookmarks(writer, chapters, dests)
point_links_at_pages(writer)
writer.page_mode = "/UseOutlines"
```

`/UseOutlines` le pide al lector que abra con los marcadores a la vista. `point_links_at_pages` arregla los enlaces mismos. Chrome escribe un enlace como «ir al lugar llamado `la-hoja-de-acordes`», y una tabla al final del PDF le da un lugar a cada nombre. Los lectores de Chrome y de Firefox buscan el nombre, pero no todos lo hacen, y en esos un enlace así no hace nada. Así que el script busca él mismo cada nombre y escribe el lugar en el enlace, «ir a la página 543, a esta altura», que todos los lectores siguen:

```
link[NameObject("/Dest")] = ArrayObject(place)
```

Un script por idioma

El libro en español tiene su propio `print-pdf.py`, una copia con las palabras en español: «Contenido» e «Índice alfabético» para los títulos, `véase` y `no` en `index-words.txt`, los glosarios en español, y dos reglas que el inglés no necesita. Una palabra terminada en *-ión* tiene su plural en *-iones* (inversión, inversiones), y las tildes no mueven una palabra en el índice: «Índice» se ordena con las íes.

Los manuales no tienen script propio. Cada uno se imprime con el script de su idioma, al que se le da la carpeta del manual como tercer argumento, donde el script encuentra el `book.toml` y el `index-words.txt` del manual.

Imprimir un PDF por tu cuenta

Nada de esto necesita GitHub. Para imprimir el PDF del libro como lo hace el flujo de trabajo, necesitas mdBook, Python 3.11 o más nuevo, y Chrome o Chromium (para este libro, en `book-es/` en lugar de `book/`):

```
python3 -m venv ~/pdf-tools
~/pdf-tools/bin/pip install pypdf
cd book
MDBOOK_OUTPUT__HTML__DEFAULT_THEME=light \
  MDBOOK_OUTPUT__HTML__PREFERRED_DARK_THEME=light \
  mdbook build --dest-dir pdf
~/pdf-tools/bin/python print-pdf.py pdf book/ferrous-lineup-book.pdf
```

Si Chrome no se llama `google-chrome` en tu sistema, di dónde está: `CHROME=chromium ~/pdf-tools/bin/python print-pdf.py ...`.

La aplicación es Rust; los libros se construyen con lo que hace cada tarea de la manera más simple: Markdown para las páginas, YAML para el flujo de trabajo, CSS para el papel, Python y pypdf para el PDF.

Formato del archivo de proyecto

Un proyecto de Ferrous Lineup es un archivo de texto en **RON**, «Rusty Object Notation», que se parece a la propia sintaxis de structs y enums de Rust. La aplicación lo escribe mediante `serde`, lo lee mediante los mismos tipos, y lo guarda con la extensión `.lineup`.

```
(
    format: 29,
    tempo: 130.0,
    scale: (root: C, kind: Major),
    channels: [
        (name: "Kick", kind: Sampler(Builtin(Kick)), volume: 0.8, muted:
false),
        // ...
    ],
    // ...
)
```

Guardar explica RON, serde y el struct `ProjectFile` que media entre el archivo y el `Project` en vivo. Esta página es la referencia del número de arriba: `FORMAT_VERSION` es **29** en `crates/lineup-core/src/file.rs`.

El número de formato

El número dice qué características puede contener un archivo. No es la versión de la aplicación.

`format_for` escribe el formato **más bajo** que puede contener el proyecto, no siempre el 29. Un proyecto sencillo se queda en el formato 1, así que cualquier versión de Ferrous Lineup puede abrirlo. Una canción con hoja de acordes pero sin banda es el formato 12; una banda es el formato 13; un canal bloqueado es al menos el formato 23. El número es el de la característica más alta que usa la canción, y cada formato puede contener todo lo que contienen los de debajo.

Cada campo que añade un formato tiene un valor por defecto de serde. Un campo en su valor por defecto se omite al escribir, y un campo ausente recibe su valor por defecto al leer. Así, una canción que no usa nada nuevo se escribe exactamente igual que antes, y un archivo no necesita ningún paso de conversión al abrirse: lleva solo lo que usa.

Formato	Qué añade	Qué haría mal una versión que solo lee los formatos de debajo	Se explica en
1	La base: canales, patrones, pistas, clips, tempo y tonalidad.	Nada; toda versión lo lee.	Guardar
2	Canales que tocan un sintetizador MIDI externo.	No conocería el canal `MidiOut` y no enviaría las notas.	MIDI
3	Nueve baterías integradas más, patrones de más de cuatro compases, e indicaciones de compás.	No conocería las baterías, cortaría los patrones largos sin avisar, y dibujaría cada compás en cuatro.	Tempo, compás, swing y sensación
4	Secciones, clips que se repiten, y una región de bucle.	Tocaría cada repetición una sola vez y perdería el bucle.	El proyecto
5	Marcas de compás, que cambian el compás a mitad de una canción.	Dibujaría cada compás en el primer compás de la canción.	Tempo, compás, swing y sensación
6	Una serie dodecafónica o un conjunto de clases de altura como tonalidad.	No conocería ninguno de los dos tipos de tonalidad y no podría abrir el archivo.	Series dodecafónicas, Conjuntos de clases de altura
7	Cambios de tempo a lo largo de una canción, y la soltura.	Tocaría toda la canción a su primer tempo y pondría cada nota en la cuadrícula.	Tempo, compás, swing y sensación

Formato	Qué añade	Qué haría mal una versión que solo lee los formatos de debajo	Se explica en
8	Cambios de tonalidad a lo largo de una canción.	Mostraría y tocaría toda la canción en su primera tonalidad.	Escalas, modos y tonalidades
9	Un instrumento General MIDI en un canal.	Tocaría cada Lineup Osc con el mismo sonido y no enviaría cambios de programa.	SoundFonts y General MIDI
10	Notas que empiezan o terminan entre pasos, como los tresillos de un archivo MIDI importado.	Las movería a la cuadrícula y cambiaría el ritmo.	MIDI
11	Pulsos agrupados de forma irregular, como 7/8 contado 2+2+3.	Leería el 7/8 pero dibujaría una línea de pulso en cada corchea.	Tempo, compás, swing y sensación
12	La hoja de acordes: la armonía de la canción, compás a compás.	Perdería la hoja de acordes sin avisar al guardar.	La hoja de acordes
13	La banda que toca la hoja de acordes, y los patrones que escribe.	Conservaría los patrones de la banda como patrones normales y dejaría de seguir la hoja.	Cómo toca la banda
14	La forma de la canción de la banda: una intro, vueltas repetidas y un final.	Tocaría la canción una vez, tal como está escrita, sin repeticiones ni final.	Forma

Formato	Qué añade	Qué haría mal una versión que solo lee los formatos de debajo	Se explica en
15	La segunda variación del estilo, elegida con marcas en la hoja de acordes.	Tocaría toda la canción en la primera variación.	Forma
16	El swing, las corcheas tocadas largo-corto.	Tocaría cada par de corcheas parejo.	Tempo, compás, swing y sensación
17	La guitarra, las cuerdas y el órgano de la banda.	Solo conocería el piano, el bajo y la batería, y no podría leer sus partes.	Cómo toca la banda
18	Golpes, sostenidos, silencios y anticipaciones en la hoja de acordes.	Perdería las marcas y tocaría el estilo de corrido.	Golpes, sostenidos, silencios y anticipaciones
19	El solista de la banda.	No conocería ni al músico solista ni su parte.	El solista
20	El ritmo de una banda tomado de un estilo y sus instrumentos de otro, el estilo de armonía exacto, y solo los acentos.	Usaría un único estilo para todas las partes y perdería el ajuste de solo acentos.	Cómo toca la banda, La orquesta
21	El master del mezclador, el solo y la reverberación estéreo.	Tocaría la canción con el master y la reverberación por defecto, no con la mezcla que guardaste.	El mezclador
22	Tus propios estilos, guardados con la banda.	Recurriría al estilo que conoce por su nombre, así que la	Tus propios estilos

Formato	Qué añade	Qué haría mal una versión que solo lee los formatos de debajo	Se explica en
		banda tocaría otra cosa.	
23	Canales bloqueados.	Dejaría que los estilos y la banda escribieran encima de la parte bloqueada.	Forma
24	Un estilo distinto para una parte de una canción.	Tocaría toda la canción en el estilo propio de la banda.	Forma
25	Dos acordes a la vez en la hoja de acordes, escritos `Eb7` sobre `Fb`.	No podría leer un poliacorde, así que no podría abrir el archivo.	Oír la armonía
26	Un balance y un envío de reverberación en cada canal.	Tocaría cada canal en el centro y con todo su sonido en la reverberación.	El mezclador
27	Enviar una batería integrada por MIDI.	Tocaría la batería ella misma en lugar de enviarla al sintetizador externo.	MIDI
28	Una sección de vientos en la banda, y EQ, compresión y delay en un canal.	No tocaría vientos, y tocaría cada canal sin sus efectos.	Cómo toca la banda, El mezclador
29	Una banda que toca la batería de otro estilo que sus partes de acordes y bajo, o que deja la batería a la canción.	Tocaría la batería del estilo propio de la banda.	Cómo toca la banda, La biblioteca

Dos registros pueden estar en un archivo de cualquier formato, porque una versión que se los salta sigue tocando la canción exactamente igual. Uno dice de dónde vino cada parte de la canción (La biblioteca). El otro es una tonalidad escrita al revés, como sol bemol mayor en lugar de fa sostenido mayor: la tonalidad se guarda como `(root: FSharp, kind: Major, respelled: true)`, y una versión que no conoce `respelled` la lee como fa sostenido mayor, con las mismas notas (Escalas, modos y tonalidades).

Archivos de una versión más nueva

Un Ferrous Lineup más nuevo puede escribir un formato que este no ha visto nunca. Un archivo así puede contener nombres y campos que esta versión no sabe leer, e incluso si los lee, su significado puede haber cambiado. Así que el lector comprueba el número de formato y rechaza todo lo que esté por encima de `FORMAT_VERSION`:

```
this file uses format 30, made by a newer version of Ferrous Lineup (this one reads up to format 29)
```

Primero se lee el archivo entero, y el número se comprueba después. Cuando el texto no se puede leer en absoluto, el lector busca el número al principio sin leer el resto, así que un archivo de una versión más nueva se explica por su número de formato y no por un desconcertante campo desconocido (Guardar).

Guardar con seguridad

`file::save` escribe en un archivo temporal junto al destino, terminado en `.lineup.tmp`, y luego lo renombra. Un cierre inesperado o un disco lleno dejan, por tanto, el proyecto anterior como estaba, y un archivo a medio escribir nunca se confunde con uno guardado; si el guardado falla, el archivo temporal se elimina. Los finales de línea son siempre `\n`, sea cual sea el sistema, y una marca de orden de bytes al principio de un archivo se ignora al leer (Guardar sin perder el archivo viejo).

Glosario de Rust

Definiciones breves de los términos de Rust usados en este libro, con el capítulo que explica cada uno.

Término	Significado	Se explica en
Anonymous lifetime (`'_`)	Un lifetime escrito sin nombre, que significa «el lifetime del argumento prestado».	Tempo, compás, swing y sensación
`Arc<T>`	Un puntero compartido con conteo de referencias que es seguro de usar entre hilos; los datos se liberan cuando se va el último `Arc`.	Instrumentos
Array	Un número fijo de valores guardados en el propio lugar, como `[u8; 12]`; la longitud forma parte del tipo.	Series dodecafónicas y sus reglas
Array `map`	`[T; N]::map` llama a un closure con cada elemento de un array y devuelve un array nuevo de la misma longitud, sin `Vec` y sin `collect`.	Series dodecafónicas y sus reglas
`array::from_fn`	Construye un array llamando a un closure una vez por cada elemento.	Instrumentos
`as` **cast**	Convierte entre tipos numéricos, o un enum sin campos a su número: `PitchClass::D as u8` es 2.	Cinco crates, y qué puede saber cada uno
Associated function	Una función en un bloque `impl` sin `self`, que se llama como `Type::name()`. `Note::new` es una.	Cinco crates, y qué puede saber cada uno
Associated type	Un tipo que elige cada implementación de un trait, como `type Error = FileError;` en un `impl TryFrom`.	Guardar

Término	Significado	Se explica en
Atomic	Un valor (como `AtomicUsize`) que los hilos pueden leer y escribir sin un lock.	Dos hilos
`AtomicU64` for an `f64`	No hay float atómico, pero `f64::to_bits` convierte un float en el `u64` con los mismos bits, y `f64::from_bits` lo devuelve a float, así que un `AtomicU64` puede compartir uno.	MIDI
Backtracking	Construir una respuesta pieza a pieza, y quitar la última pieza cuando no lleva a ninguna parte; normalmente una función que se llama a sí misma.	Series dodecafónicas y sus reglas
Big-endian / `to_be_bytes`	Escribir primero el byte más significativo de un número; `6u32.to_be_bytes()` es `[0, 0, 0, 6]`. Los archivos MIDI lo usan.	MIDI
Bit operations	`&` (y), `|` (o), `!` (no), `<<` y `>>` (desplazamiento) trabajan sobre los bits de un entero.	Clases de altura, notas y su escritura
`bool::then` / `then_some`	Convierten una condición en un `Option`: `cond.then_some(x)` es `Some(x)` si `cond` es true y `None` si no; `cond.then(|| ...)` solo ejecuta el closure cuando es true.	Herramientas de edición como funciones simples
Borrow	Préstamo: usar un valor mediante una referencia (`&T` compartida, `&mut T` mutable) sin tomar su propiedad.	Dos hilos
Borrow checker	La parte del compilador que impone: un `&mut` *o* cualquier número de `&` a la vez, y ninguna referencia que sobreviva a su valor.	Decirle al hilo de audio lo que ha cambiado
`Borrow` trait	Una promesa de que un tipo con propiedad (como `PathBuf`)	Decirle al hilo de audio lo

Término	Significado	Se explica en
	hace hash y compara exactamente como el tipo que guarda (`Path`), de modo que un `HashMap<PathBuf, _>` se puede buscar con un `&Path` .	que ha cambiado
`Box<T>`	Un puntero con propiedad a un valor en el montículo.	Dibujar las ventanas con egui
Byte string (`b"..."`)	Un literal de string que es un `&[u8; N]` de códigos ASCII en lugar de un `&str` , como `b"MThd"` .	MIDI
Capacity	Cuántos elementos puede guardar un `Vec` antes de tener que reservar más memoria.	Dos hilos
Cargo	La herramienta de compilación y el gestor de paquetes de Rust.	Cinco crates, y qué puede saber cada uno
`Cell<T>`	Una caja cuyo valor se puede reemplazar a través de una referencia compartida, moviendo valores enteros hacia dentro y hacia fuera con `get` y `set` (para tipos `Copy`); nada puede tener una referencia a lo de dentro, así que nada lo lee mientras cambia.	Números romanos, las teclas de armonía y las funciones de los acordes
`cfg` / `cfg_attr`	Compilación condicional: `#[cfg(target_os = "windows")]` conserva el elemento siguiente solo cuando la condición es cierta, y `cfg_attr` aplica un atributo solo entonces.	Tres sistemas e integración continua
`#[cfg(test)]` **on an item**	Conserva una función (o `use` , o struct) solo en las compilaciones de test, para que un ayudante al que solo llaman los tests no dé avisos de no usado en una compilación normal.	Micromodos

Término	Significado	Se explica en
`checked_add_signed`	Suma un número con signo a uno sin signo, como `usize + isize`, y da `None` en lugar de bajar de cero o desbordar.	Herramientas de edición como funciones simples
`chunk_by`	Divide un slice en tramos de vecinos que van juntos, sin copiar.	Los editores
`chunks` / `chunks_exact`	Dividen un slice en trozos de n elementos; `chunks_exact` deja fuera el último trozo si queda corto.	Series dodecafónicas y sus reglas
`chars` / `Chars::as_str`	`text.chars()` recorre un string de a un carácter completo («♭» son tres bytes pero un solo `char`); `as_str()` es la parte que aún no se ha recorrido, prestada del mismo texto.	Clases de altura, notas y su escritura
Closure	Una función anónima, `|args| body`, que puede capturar variables de su entorno.	Dos hilos
Closure captures fields	Desde la edición 2021, un closure que usa `self.plays` pide prestado solo ese campo, no todo `self`, así que los demás campos se pueden pedir prestados a la vez.	El secuenciador
`cmp::Reverse`	Envuelve un valor para que compare al revés: ordenar por `Reverse(count)` pone primero el conteo más grande.	Micromodos
Collecting `Option`s	Recolectar un iterador de `Option<T>` en un `Option<Vec<T>>` da `Some` de todos los valores, o `None` en cuanto un elemento es `None`. `Result` funciona igual.	Herramientas de edición como funciones simples
`const fn`	Una función lo bastante simple como para ejecutarse durante la	Guardar

Término	Significado	Se explica en
	compilación, de modo que sirve para definir un `const`.	
Constant promotion	Escribir `&[1, 2, 3]` de valores constantes simples da un préstamo `&'static`, porque el compilador guarda los valores con el programa. Una llamada a función dentro de los corchetes, incluso a un `const fn`, no se promueve.	La biblioteca
`Copy`	Un trait marcador para valores pequeños que se copian en lugar de moverse, como los números y `Note`; `let held = list[i];` copia entonces el elemento fuera.	Cinco crates, y qué puede saber cada uno
`count_ones`	Cuántos bits de un entero son 1: el tamaño de un conjunto guardado como bits.	Clases de altura, notas y su escritura
Crate	Una biblioteca o un programa; la unidad que Cargo compila y publica.	Cinco crates, y qué puede saber cada uno
`cycle`	Repite un iterador para siempre. Es seguro porque los iteradores son perezosos: otra cosa, como `zip`, decide cuándo parar.	Herramientas de edición como funciones simples
Declarative macro (`macro_rules!`)	Una macro definida por patrones: `macro_rules! hit { ($at:expr, ...) => { ... } }` hace match con lo que recibe y lo pega en una plantilla, antes de que el código se compile.	La biblioteca
`Default`	Un trait para «el valor por defecto» de un tipo; en un enum, `#[derive(Default)]` necesita `#[default]` en una variante.	Cinco crates, y qué puede saber cada uno

Término	Significado	Se explica en
Deref coercion	Rust convierte `&Vec<T>` en `&[T]` (o `&String` en `&str`) automáticamente donde hace falta.	MIDI
Dereference (`*`)	Sigue una referencia hasta el valor al que apunta, para leerlo o escribirlo.	El proyecto
Derive	`#[derive(Trait)]` pide al compilador que implemente traits comunes automáticamente.	Cinco crates, y qué puede saber cada uno
Derive macro	Código que escribe la implementación de un trait para un tipo; el `#[derive(Serialize)]` de serde viene de un crate, no del compilador.	Guardar
`Display`	El trait detrás de `{}` en `format!` y de `to_string()`. Implementa su método `fmt` una vez, y todos los lugares escriben el valor de la misma manera.	Cinco crates, y qué puede saber cada uno
Disjoint borrows	Pedir prestados dos campos distintos de un struct a la vez, incluso uno de forma mutable; permitido porque no pueden solaparse.	Dibujar las ventanas con egui
`div_ceil`	División entera que redondea hacia arriba: `16.div_ceil(6)` es 3, mientras que `16 / 6` es 2.	Micromodos
`drain(..)`	Saca todos los elementos de un `Vec`, en orden, y conserva la memoria del `Vec` para más adelante.	MIDI
`dyn Trait`	Un objeto de trait: algún tipo que implementa `Trait`, elegido en tiempo de ejecución.	Guardar
Edition	Una versión del lenguaje (2015, 2018, 2021, 2024) elegida por	Cinco crates, y qué puede

Término	Significado	Se explica en
	cada crate.	saber cada uno
Enum	Un tipo cuyo valor es una de varias variantes, cada una de las cuales puede guardar datos.	Dos hilos
Enum size	Un enum ocupa tanto como su variante más grande; poner los datos de una variante grande en un `Box` mantiene pequeño cada valor. El lint `large_enum_variant` de Clippy lo señala.	MIDI
`env!` y `concat!`	Macros que se ejecutan mientras se compila el programa: `env!("CARGO_PKG_VERSION")` se convierte en la versión que Cargo leyó de `Cargo.toml`, y `concat!` une literales en un solo `&'static str`, así que ambas pueden construir una `const`.	Dibujar las ventanas con egui
`expect`	Como `unwrap`, pero entra en pánico con el mensaje dado. Afirma una promesa («esto siempre es `Some`»), que conviene respaldar con un test.	Conjuntos de clases de altura, y una armonía con cuatro nombres
Feature	Una parte opcional de un crate, que se activa o desactiva en `Cargo.toml`; los features solo pueden añadir, nunca quitar.	Guardar
`find_map`	Llama a un closure con cada elemento y devuelve el primer `Some` que dé.	Acordes
`flat_map`	Convierte cada elemento en un iterador y une todos en un único iterador.	Los editores
Float to integer `as`	`x as usize` descarta la fracción y *satura*: los números negativos	MIDI

Término	Significado	Se explica en
	y NaN se vuelven 0, los números demasiado grandes se vuelven el máximo del tipo. Nunca entra en pánico.	
`fn` pointer	El tipo de una función simple, como `fn(usize) -> Change`. Las variantes de enum que guardan datos son funciones así: `Change::RemovePattern` construye un `Change` a partir de un `usize`.	Decirle al hilo de audio lo que ha cambiado
`Fn` / `FnMut` / `FnOnce`	Los traits que implementan las closures: solo lee, muta sus capturas, o solo se puede llamar una vez.	Instrumentos
`From` / `Into`	Traits de conversión: implementar `From<A> for B` también da `a.into()`. El operador `?` usa `From` para convertir errores.	El proyecto
`FromIterator`	El trait detrás de `collect()`; impleméntalo para recolectar en tu propio tipo.	Clases de altura, notas y su escritura
`FromStr` / `str::parse`	El trait para «se puede leer desde texto»: implementa `from_str`, y `"Am7".parse::<ChordSymbol>()` funciona, como con los números.	Acordes
Generic	Código escrito para un tipo comodín `T`, rellenado en tiempo de compilación.	Dos hilos
Glob import	`use super::*;` trae todos los nombres que el otro módulo puede ver, incluidos los elementos privados de un padre cuando se usa desde un hijo. Dos imports con glob que traen el mismo nombre lo hacen ambiguo.	Cómo se escribe una demostración

Término	Significado	Se explica en
`HashMap<K, V>`	Un mapa de claves a valores que encuentra una clave más o menos en el mismo tiempo tenga las entradas que tenga.	Decirle al hilo de audio lo que ha cambiado
`if let` / `while let`	Ejecutan un bloque si (o mientras) un valor coincide con un patrón.	Dibujar las ventanas con egui
`ilog2`	El logaritmo entero en base 2: cuántas veces se puede partir un número por la mitad antes de llegar a 1. `8u8.ilog2()` es 3. Entra en pánico con 0.	Tempo, compás, swing y sensación
`impl Trait`	Como tipo de argumento: cualquier tipo con este trait. Como tipo de retorno: un tipo con este trait, sin nombre.	El proyecto
Inner attribute (`#![...]` **)**	Un atributo que se aplica a aquello dentro de lo cual está escrito, como todo el crate al principio de `main.rs`.	Tres sistemas e integración continua
Invariant	Una regla que siempre se cumple sobre un valor, normalmente protegida por campos privados.	Series dodecafónicas y sus reglas
`is_some_and`	`true` si un `Option` es `Some` y su valor pasa una prueba, `false` para `None`.	Números romanos, las teclas de armonía y las funciones de los acordes
Iterator	Una secuencia perezosa de valores, transformada con métodos como `map`, `filter`, `count`.	Cómo se prueba el código
`JoinHandle<T>`	Lo que devuelve `thread::spawn`. `join()` espera a que el hilo acabe y devuelve lo que devolvió su closure, o un `Err` si entró en pánico.	Exportar

Término	Significado	Se explica en
Lazy	Los iteradores no trabajan hasta que algo extrae elementos de ellos.	Herramientas de edición como funciones simples
Leading underscore (`_name`)	Dice al compilador que una variable o campo no se usa a propósito, para que no avise de ello.	Exportar
`let ... else`	Desestructura un patrón, o ejecuta un bloque `else` que debe salir de la función.	Decirle al hilo de audio lo que ha cambiado
Let chain	`if cond && let Some(x) = value { ... }` : condiciones y coincidencias de patrones en un solo `if` (Rust 1.88).	Dos hilos
Literal in a pattern	`Event::Key { repeat: false, .. }` : un valor escrito en un patrón solo coincide cuando el campo es igual a él.	MIDI
`match`	Compara un valor con patrones; debe cubrir todos los casos.	Dos hilos
Match guard	`pattern if condition =>` : el brazo coincide solo si la condición también es cierta.	Acordes
`matches!`	Una macro que es `true` si un valor coincide con un patrón.	Clases de altura, notas y su escritura
`mem::replace`	Pone un valor nuevo en un lugar y devuelve el viejo.	Dos hilos
`mem::swap`	Intercambia los valores detrás de dos referencias `&mut` , sin copiar ninguno.	Deshacer y rehacer
`mem::take`	Saca un valor de un lugar y deja detrás el valor por defecto del tipo (como un `Vec` vacío).	El secuenciador
Method	Una función en un bloque `impl` que recibe `self` , `&self` o `&mut`	Cinco crates, y qué puede

Término	Significado	Se explica en
	`self`.	saber cada uno
`min_by_key`	El elemento de un iterador con la clave más pequeña, o `None` si no hay elementos.	Conjuntos de clases de altura, y una armonía con cuatro nombres
Module	Un espacio de nombres dentro de un crate, normalmente un archivo, declarado con `mod`.	Cinco crates, y qué puede saber cada uno
Monomorphization	El compilador hace una copia especializada del código genérico para cada tipo usado.	Dos hilos
Move	Transferir la propiedad de un valor; el nombre antiguo deja de poderse usar.	Dos hilos
`move` **closure**	Un closure que toma la propiedad de todo lo que usa en lugar de pedirlo prestado; necesario para un closure que se entrega a otro hilo.	Exportar
`mpsc` **channel**	Una cola de la biblioteca estándar con muchos emisores (`Sender` se puede clonar) y un receptor; `try_iter` toma lo que está esperando sin esperar más.	MIDI
`Mutex<T>` **and poisoning**	Un lock que deja a un solo hilo a la vez llegar al `T` de dentro. Si un hilo entra en pánico mientras lo sostiene, el mutex queda *envenenado*, y `lock` devuelve un error hasta que tomas los datos de todos modos con `PoisonError::into_inner`.	MIDI
Newtype	Un struct con un solo campo que envuelve otro tipo para darle su propio significado.	Clases de altura, notas y su escritura

Término	Significado	Se explica en
`?` **operator**	Sale de la función devolviendo el error si un `Result` es `Err` (o un `Option` es `None`).	Dos hilos
`Option::and_then`	Llama a un closure que devuelve un `Option` sobre el valor, si lo hay; `None` sigue siendo `None`.	Los editores
`Option::as_mut`	Convierte `&mut Option<T>` en `Option<&mut T>`, para usar el valor de dentro sin moverlo.	Dos hilos
`Option::filter`	Conserva un `Some` solo si un closure dice que sí, y da `None` si no.	MIDI
`Option<T>`	`Some(value)` o `None`; el sustituto de Rust para null.	Dos hilos
`Option::take`	Mueve el valor fuera de un `Option` y deja `None` en su lugar; útil para tomar la propiedad de algo guardado en un campo.	Exportar
Or-pattern	`A | B` en un patrón coincide con `A` o con `B`.	Dibujar las ventanas con egui
Ownership	Cada valor tiene exactamente un dueño, y se libera cuando el dueño desaparece.	Dos hilos
`partition`	Divide un iterador en dos con un closure, en una pasada: los elementos a los que dice que sí y el resto, cada uno recolectado en su propia colección.	Micromodos
`partition_point`	En un slice ordenado de modo que una prueba es `true` para una primera parte y `false` para el resto, el índice donde empieza la parte `false`, hallado por búsqueda binaria.	Conjuntos de clases de altura, y una armonía con cuatro nombres
Panic	Un error irrecuperable que detiene el hilo actual (`panic!`, `assert!` fallido).	Cinco crates, y qué puede

Término	Significado	Se explica en
		saber cada uno
Property-based test	Un test que comprueba una regla que debe cumplirse para cualquier entrada («seguir los changes da el proyecto nuevo»), en lugar de una respuesta para una entrada.	Decirle al hilo de audio lo que ha cambiado
`pub(crate)`	Visible en todo el interior del crate actual, pero no para otros crates.	El proyecto
`pub(super)`	Visible para el módulo padre y todo lo que hay dentro de él.	Guardar
Pure function	Una función cuyo resultado depende solo de sus argumentos y que no cambia nada más; el tipo de código más fácil de probar.	Herramientas de edición como funciones simples
`Rc<T>`	Un puntero con conteo de referencias para un solo hilo: clonarlo suma a un conteo en lugar de copiar el valor, que se libera cuando se va el último `Rc`. `Rc<[T]>` comparte un slice.	Conjuntos de clases de altura, y una armonía con cuatro nombres
RAII / `Drop`	Ser dueño de un valor significa ser dueño de su recurso; al hacer drop de él se libera el recurso.	Dos hilos
Range	`a..b`, los números desde `a` hasta `b` sin incluirlo; un valor con métodos como `contains`.	El secuenciador
Recursion	Una función que se llama a sí misma sobre un trozo más pequeño del problema, hasta que el trozo es lo bastante pequeño para responder directamente.	Series dodecafónicas y sus reglas
`rem_euclid`	Un resto que nunca es negativo: `(-1_i32).rem_euclid(12)` es 11, mientras que `-1 % 12` es -1.	Clases de altura, notas y su escritura

Término	Significado	Se explica en
`Result<T, E>`	`Ok(value)` o `Err(error)`; la manera de Rust de devolver errores.	Dos hilos
`retain`	Conserva solo los elementos de un `Vec` para los que un closure devuelve `true`, en una pasada.	El proyecto
RON	Rust Object Notation, un formato de texto que se parece a la propia sintaxis de struct de Rust; los archivos de proyecto de Ferrous Lineup lo usan.	Guardar
`rsplit_once` / `split_once`	Divide un string en el último (o primer) lugar donde aparece un patrón, en las partes de antes y de después, o `None` si no aparece.	Acordes
`rposition`	El índice del último elemento de un slice que coincide, buscando desde el final; `position` busca desde el principio.	Micromodos
`saturating_sub`	Resta que se detiene en el valor más pequeño (0 para `usize`) en lugar de desbordar.	Herramientas de edición como funciones simples
`Send`	Un trait marcador: los valores de este tipo se pueden mover a otro hilo.	Dos hilos
serde	El crate habitual para convertir valores de Rust en texto o bytes y viceversa, mediante los traits `Serialize` y `Deserialize`.	Guardar
serde container attributes	`#[serde(try_from = "String", into = "String")]` en un tipo lo guarda como el `String` que da su `From`, y lo lee de vuelta a través de su `TryFrom<String>`, de modo que un valor leído de un archivo se comprueba como cualquier otro.	Series dodecafónicas y sus reglas

Término	Significado	Se explica en
serde field attributes	`#[serde(default)]` rellena un campo que falta con su `Default` al leer; `#[serde(skip_serializing_if = "path")]` deja un campo fuera al escribir si esa función lo dice.	Guardar
Slice	Una vista prestada de parte de un array o un `Vec`: `&[T]` o `&mut [T]`.	Dos hilos
Slice pattern	`let [a, b, c, ..] = *bytes else { ... }`: coincide con un slice por su forma, nombrando los primeros elementos, de modo que indexar no puede entrar en pánico.	MIDI
Slice pattern with `@`	`[first @ .., last]` coincide con un slice de dos o más elementos y nombra todo menos el último como el slice `first`.	Micromodos
State machine	Código que mantiene un estado actual, a menudo un enum, y pasa de un estado a otro según suceden cosas.	Instrumentos
`'static`	Un lifetime que significa «vive tanto como haga falta»; para closures, «no pide prestado nada temporal».	Dos hilos
`static` **item**	`static NAME: T = ...;` es un valor en una dirección fija para todo el programa. Un `const`, en cambio, se pega en cada lugar donde se usa.	La biblioteca
`step_by`	Toma un elemento de cada n de un iterador: `(0..8).step_by(2)` es 0, 2, 4, 6.	Herramientas de edición como funciones simples
Struct update syntax	`NoteEvent { length: 2, ..note }`: un struct nuevo con	Dibujar las ventanas con

Término	Significado	Se explica en
	algunos campos cambiados y el resto tomados de `note`.	egui
Target	El tipo de sistema para el que se compila un programa, como `x86_64-pc-windows-gnu`; `rustup target add` instala la biblioteca de Rust para uno.	Tres sistemas e integración continua
`thread::Builder`	Arranca un hilo con opciones como un nombre; su `spawn` devuelve un `Result` en lugar de entrar en pánico cuando el sistema se niega.	MIDI
`thread_local!`	Declara un valor del que cada hilo tiene su propia copia, al que se llega con `.with(...)`; la app guarda así sus ajustes de números romanos, nombres de notas e idioma, para que las pruebas que corren a la vez nunca vean los de las demás.	Números romanos, las teclas de armonía y las funciones de los acordes
Trait	Un conjunto de métodos que un tipo promete proporcionar, como una interfaz.	Cinco crates, y qué puede saber cada uno
Trait bound	Una exigencia sobre un tipo genérico, como `T: FromSample<f32>`.	Dos hilos
`TryFrom` / `TryInto`	Conversiones que pueden fallar, devolviendo un `Result`.	Guardar
Tuple	Un grupo fijo de valores, posiblemente de tipos distintos, como `(usize, Note)`; se puede hacer match y desmontar.	Clases de altura, notas y su escritura
Tuple comparison	Las tuplas comparan elemento a elemento: el segundo elemento solo decide cuando los primeros son iguales. Una tupla como clave de orden da desempates en orden.	El mezclador

Término	Significado	Se explica en
Tuple struct	Un struct cuyos campos no tienen nombre, como `struct Note(u8)`; los campos se leen como `.0`, `.1`…	Cinco crates, y qué puede saber cada uno
Turbofish (`::<>`)	Escribir el tipo de una función genérica en la llamada, como en `collect::<Vec<Note>>()`, cuando Rust no puede deducirlo.	Dos hilos
Type alias	Un segundo nombre para un tipo existente: `type Hit = (u8, i32);`.	La biblioteca
`VecDeque<T>`	Una cola de doble extremo: un búfer circular al que es barato añadir y del que es barato quitar por ambos extremos.	Deshacer y rehacer
`windows`	Cada tramo de n vecinos de un slice, solapándose: `[1, 2, 3].windows(2)` da `[1, 2]` y `[2, 3]`.	Cómo se prueba el código
`wrapping_add` / `wrapping_mul`	Aritmética que da la vuelta al desbordar en lugar de detener una compilación de depuración, para cuando dar la vuelta es lo que quieres, como en los números aleatorios.	Series dodecafónicas y sus reglas
Workspace	Un grupo de crates compilados juntos con un solo `Cargo.lock`.	Cinco crates, y qué puede saber cada uno
`zip`	Empareja los elementos de dos iteradores, `(a, b)`, y para cuando cualquiera de los dos se acaba.	Decirle al hilo de audio lo que ha cambiado

Dónde aprender más

- *The Rust Programming Language* («el libro»): se instala con Rust; ábrelo sin conexión con `rustup doc --book`.
- *Rust by Example*: `rustup doc --rust-by-example`.
- La documentación de la biblioteca estándar: `rustup doc --std`.

Glosario de música

Este libro usa las palabras que usan los músicos. Esta página da a cada una un significado breve y el capítulo que la muestra en acción. El Glosario de Rust está aparte: explica el lenguaje, no la música.

Término	Significado	Se explica en
Comping	Los acordes que un pianista o un guitarrista toca debajo de la melodía, y el ritmo y los voicings que usan. Como verbo, comp es acompañar con acordes: tocar esos acordes.	Cómo toca la banda una hoja de acordes
Voicing	Qué notas de un acorde suenan, en qué octavas y cómo están repartidas. Un buen voicing conserva las notas comunes y deja sitio a la melodía.	Cómo toca la banda una hoja de acordes
Conducción de voces (*voice leading*)	Mover cada voz de un acorde a la voz más cercana del siguiente, para que la armonía se una con suavidad en lugar de saltar.	Números romanos, las teclas de armonía y las funciones de los acordes
Changes	La progresión de acordes de una canción, compás a compás. «Los changes» son lo que contiene la hoja de acordes.	Cómo toca la banda una hoja de acordes
Head	La melodía escrita, tocada al principio y otra vez después de los solos. En una actuación de jazz, el head es la melodía que todos conocen.	Forma, secciones, variaciones y bloqueos
Chorus	En jazz, una pasada completa por la forma, desde el principio del chart hasta el último compás. En una canción pop, el estribillo, la parte que se repite después de cada estrofa.	Forma, secciones, variaciones y bloqueos
Turnaround	Los acordes al final de una sección que llevan de vuelta a su	Forma, secciones, variaciones y

Término	Significado	Se explica en
	principio, como I–vi–ii–V.	bloqueos
Walking bass	Una línea de bajo de una nota por pulso: la fundamental, notas del acorde y un paso hacia la fundamental del acorde siguiente.	Cómo toca la banda una hoja de acordes
Acorde con barra (*slash chord*)	Un acorde con barra: un símbolo de acorde con una nota de bajo después de una barra, como `C/E`. El bajo puede ser una nota del acorde (una inversión) o una nota ajena a él.	Acordes
Inversión (*inversion*)	Un acorde con una nota distinta de su fundamental en el bajo. En una tríada, la primera inversión lleva la tercera como nota más grave y la segunda, la quinta; en un acorde de séptima, la tercera inversión lleva la séptima.	Acordes
Bajo cifrado (*figured bass*)	Una taquigrafía del continuo: números encima o debajo de la nota de bajo nombran los intervalos que hay que tocar sobre ella, como 6, 6/4 o 6/5. Ferrous Lineup escribe las cifras después del número romano, como `I⁶` y `V⁶₅`.	Números romanos, las teclas de armonía y las funciones de los acordes
Número romano (*roman numeral*)	Un acorde nombrado por el grado de la tonalidad sobre la que está construido: mayúsculas para mayor, minúsculas para menor. En Do mayor, G7 es V7.	Números romanos, las teclas de armonía y las funciones de los acordes
Tonalidades enarmónicas (*enharmonic keys*)	Dos nombres para las mismas teclas del piano, como Fa♯ mayor (seis sostenidos) y Sol♭ mayor (seis bemoles). Suenan igual; el compositor elige el nombre que encaja con la música que lo rodea.	Escalas, modos y tonalidades
Función (*function*)	Lo que hace un acorde en la tonalidad: la tónica es el hogar, la subdominante aleja, la dominante	Números romanos, las teclas de armonía

Término	Significado	Se explica en
	es tensión que tira hacia el hogar. Las teclas de armonía las colorean de azul, verde y naranja.	y las funciones de los acordes
Acorde prestado (*borrowed chord*)	Un acorde tomado de la tonalidad homónima, la mayor o menor sobre la misma tónica, como La♭ (♭VI) en Do mayor, tomado en préstamo de Do menor. También se llama **mixtura modal**.	Números romanos, las teclas de armonía y las funciones de los acordes
Tercera de picardía (*Picardy third*)	Un acorde de tónica mayor al final de una pieza en tonalidad menor: la tercera mayor tomada en préstamo de la mayor.	Números romanos, las teclas de armonía y las funciones de los acordes
Dominante secundaria (*secondary dominant*)	La dominante de un acorde de la tonalidad distinto de la tónica, escrito V/V («dominante de la dominante»), V7/vi y así. D7 en Do mayor es V7/V, la dominante de Sol.	Números romanos, las teclas de armonía y las funciones de los acordes
Tonicización (*tonicization*)	Hacer que un acorde de la tonalidad suene por un momento como una tónica, tocando antes su propia dominante o su acorde de sensible, sin cambiar de tonalidad.	Números romanos, las teclas de armonía y las funciones de los acordes
Cuarta y sexta cadencial (*cadential six-four*)	La tríada de tónica en segunda inversión sobre el bajo de la dominante, como C/G antes de G7 en Do mayor. Suena como la dominante retenida, y avanza hacia V.	Números romanos, las teclas de armonía y las funciones de los acordes
Sexta napolitana (*Neapolitan sixth*)	Un acorde mayor sobre el segundo grado rebajado, normalmente en primera inversión: D♭/F en Do mayor, escrito N^6. Casi siempre va a la dominante.	Números romanos, las teclas de armonía y las funciones de los acordes
Acorde pivote (*pivot chord*)	Un acorde que pertenece a la tonalidad vieja y a la nueva en una modulación, oído en la	Números romanos, las teclas de armonía

Término	Significado	Se explica en
	primera y releído después en la segunda: Am en Do mayor es vi, y ii en Sol mayor.	y las funciones de los acordes
Grado conjunto, salto, salto grande (*step, skip, leap*)	Hasta dónde se mueve una parte: una segunda es un grado conjunto, una tercera un salto y una cuarta o más, un salto grande.	Conducción de voces y movimiento de las partes
Movimiento contrario (*contrary motion*)	Dos partes que se mueven en direcciones opuestas. Entre la línea superior y el bajo las mantiene como dos líneas independientes.	Conducción de voces y movimiento de las partes
Movimiento oblicuo (*oblique motion*)	Una parte mantiene su nota mientras la otra se mueve.	Conducción de voces y movimiento de las partes
Movimiento paralelo (*parallel motion*)	Dos partes que se mueven en la misma dirección y conservan el mismo número de intervalo, como sextas que van a sextas. Las quintas y octavas paralelas son lo que evita la escritura clásica de partes.	Conducción de voces y movimiento de las partes
Posición cerrada (*close position*)	Un acorde con sus notas tan juntas como es posible, todas dentro de una octava, como lo toca una sección de vientos.	Cómo toca la banda una hoja de acordes
Ecualizador (EQ) (*equalizer*)	Un filtro que sube o baja rangos de frecuencias, como los graves, los medios y los agudos de un canal. Cambia el color de un sonido, no sus notas.	El mezclador
Compresor (*compressor*)	Un efecto que baja un sonido cuando supera un umbral, según un ratio, para que una parte suene a un nivel más constante.	El mezclador
Delay	Un efecto que vuelve a tocar un sonido un poco más tarde, y otra	El mezclador

Término	Significado	Se explica en
	vez, cada eco más bajo; en la app, a tiempo con la canción.	
Poliacorde (*polychord*)	Dos acordes que suenan a la vez, escritos uno encima del otro, como `Eb7` sobre `Fb` . El oído oye los dos acordes, no un acorde grande.	Oír la armonía
Conjunto de clases de alturas (*pitch-class set*)	Un grupo de notas nombrado por clase de alturas, ignorando octava y orden. Todo acorde es uno, pero el término también cubre grupos que no tienen nombre de acorde.	Conjuntos de clases de altura, y una armonía con cuatro nombres
Forma prima (*prime form*)	La transposición o inversión más compacta de un conjunto de clases de alturas, usada para comparar conjuntos. El catálogo de Forte lista cada clase de conjunto con su forma prima y su vector de intervalos.	Conjuntos de clases de altura, y una armonía con cuatro nombres
Serie dodecafónica (*twelve-tone row*)	Un ordenamiento de las doce clases de alturas usado como material de una pieza. Sus formas son la prima, la inversión, la retrogradación y la inversión retrógrada.	Series dodecafónicas y sus reglas
Micromodo (*micromode*)	Uno de los doce grupos de tres notas de Francisco Kröpfl, clasificados por su contenido interválico y oídos como un color. La app puede usar uno como escala o dibujar con él.	Micromodos
Swing	Tocar la segunda corchea de cada pulso tarde, de modo que las parejas van larga-corta. Un swing completo, al 100 %, es un tresillo: dos tercios y un tercio.	Tempo, compás, swing y sensación
Golpe (*shot*)	Un golpe de acorde corto y acentuado que toca toda la banda, seguido de silencio. También se llama stab, hit o kick.	Golpes, sostenidos, silencios y anticipaciones

Término	Significado	Se explica en
Anticipación (*push*)	Un acorde atacado antes de su pulso y sostenido sobre él, escrito `^F`; también llamado anticipación. Hace que la música se incline hacia delante.	Golpes, sostenidos, silencios y anticipaciones
Sostenido (*hold*)	Un acorde atacado y sostenido hasta la siguiente armonía, con el pulso siguiendo debajo. No es un calderón.	Golpes, sostenidos, silencios y anticipaciones
Silencio (*rest*)	La banda deja de tocar durante un compás o más; en jazz, un break. En la app, un silencio también deja muda la batería.	Golpes, sostenidos, silencios y anticipaciones
Afinación de concierto (*concert pitch*)	Música en afinación de concierto, escrita como suena, «en Do»: lo que leen el piano, la guitarra, el bajo, la voz, la flauta y el violín. La tonalidad de concierto de una canción es la tonalidad que oye la banda.	La partitura guía
Instrumento transpositor (*transposing instrument*)	Un instrumento cuyas notas escritas no son las notas que suena, nombrado por la nota que suena un Do escrito: una trompeta en Si♭, un saxofón alto en Mi♭, una trompa en Fa. Su música se escribe esa distancia más aguda, de modo que una canción en Do está en Re para una trompeta.	La partitura guía
Solfeo (*solfège*)	Nombrar las notas con sílabas, Do Re Mi Fa Sol La Si, en lugar de con las letras C D E F G A B, como en español, italiano, francés y portugués. Un acorde se nombra igual: Am7 es Lam7 (*cifrado latino*, frente al *cifrado americano* con letras).	Dos idiomas
Do fijo (*fixed do*)	Solfeo en el que Do es siempre la misma nota, sea cual sea la tonalidad, como lo usan la app y	Dos idiomas

Término	Significado	Se explica en
	las escuelas de España y América Latina. En el **do móvil**, que usan algunos métodos de entrenamiento auditivo, *do* es la tónica de la tonalidad.	

Ideas para lo que viene

Nada de esto está planificado todavía. Son ideas, más o menos de menor a mayor, y cada una sería un buen proyecto para quien lea este libro:

- **Más banda**: más estilos, y más instrumentos que pueda traer a cualquiera de ellos, como ahora trae los vientos (Una sección de vientos).
- **Un solista que ya no sea un experimento** (El solista).
- **Grabar en modo canción**: hoy Grabar pasa al modo patrón y graba en el patrón que se repite (ejercicio 22 de la Parte II).
- **Clips de audio** en las pistas, junto a los patrones, colocados y movidos como cualquier clip. Una grabación importada suena ahora como una sola nota larga del sampler en el primer paso de la canción, que el motor empieza por la mitad cuando la reproducción empieza más tarde (Empezar por la mitad).
- **Automatización**: movimientos de mandos grabados o dibujados a lo largo del tiempo, sobre las pistas.

Índice alfabético

Las palabras de los glosarios, las partes de la aplicación y los capítulos que las explican. Un número en **negrita** es la página a la que el glosario te envía para esa palabra; los demás son donde un capítulo la presenta o trata de ella.

A

Acorde con barra, **456**
Acorde pivote, 468, **484**
Acorde prestado, **481**
Afinación de concierto, **834**
Ajustar a la escala, 443
Anonymous lifetime (`'_`), **229**
Anticipación, 640, 770, **772**, 775
`Arc<T>`, **237**
Archivo de proyecto, 80, 942
Archivo MIDI, 330, 336, 891
Arpegio, 496
Array, **542**
Array `map`, **542**
`array::from_fn`, **237**
`as` cast, **18**
Associated function, **17**
Associated type, **80**
Atajos de teclado, 78, 125, 150
Atomic, **22**
`AtomicU64` for an `f64`, **298**

B

Backtracking, **543**, 544
Bajo cifrado, 464, **470**
Batería, 239, 291, 643
Biblioteca, 639
Big-endian / `to_be_bytes`, **298**
Bit operations, **424**
`bool::then` / `then_some`, **487**
Borrow, **22**
Borrow checker, **54**
`Borrow` trait, **51**
`Box<T>`, **101**
Byte string (`b"..."`), **298**

C

Cambio de tonalidad, 437, 441, 517
Capacity, **22**
Cargo, **12**
`Cell<T>`, **473**
`cfg` / `cfg_attr`, **914**
`#[cfg(test)]` on an item, **575**
Changes, **697**
`chars` / `Chars::as_str`, **422**
`checked_add_signed`, **487**
Chorus, **749**
`chunk_by`, **131**
`chunks` / `chunks_exact`, **543**
Clase de altura, 414
Clip, 36, 161
Closure, **22**, 28, 54, 753, 785, 843
Closure captures fields, **205**
`cmp::Reverse`, **578**
Collecting `Option`s, **487**
Compás, 210
Compases irregulares, 213, 732
Comping, 640, **697**, 698
Compresor, 283, **285**
Conducción de voces, **477**, 537, 556, 587, 634, 636, 698
Conjunto de clases de alturas, **522**, 528, 550, 556, 634
`const fn`, **80**
Constant promotion, **641**
`Copy`, **18**
`count_ones`, **424**

Crate, **12**
Cuarta y sexta cadencial, **481**
`cycle`, **487**

D

Declarative macro (`macro_rules!`), **641**
`Default`, **18**
Delay, 283, **286**, 404
Demo, 874, 891
Deref coercion, **302**
Dereference (`*`), **36**
Derive, **18**
Derive macro, **80**
Deshacer, 70
Disjoint borrows, **105**
`Display`, **18**
`div_ceil`, **579**
Do fijo, **172**
do móvil, *véase* Do fijo
Dominante secundaria, 463, 468, **481**
`drain(..)`, **298**
`dyn Trait`, **89**

E

Ecualizador (EQ), **284**
Edition, **12**
Editor de estilos, 793
Encontrar la tonalidad, 514
Entrenador de velocidad, 804
Entrenamiento auditivo, 804
Enum, **22**, 23
Enum size, **320**
`env!` y `concat!`, **127**
Envolvente, 250
Escala, 428
Estilo, 642, 697, 759
Estilo por sección, 759
Estructura de la canción, 45
`expect`, **526**
Exportación, *véase* Exportación WAV, Exportación de video
Exportación de video, 350
Exportación WAV, 350

F

Feature, **80**, 83, 605, 915, 916
`find_map`, **446**
`flat_map`, **131**
Float to integer `as`, **298**
`Fn` / `FnMut` / `FnOnce`, **240**
`fn` pointer, **51**
Forma de la canción, 747
Forma prima, **522**, 523, 566
`From` / `Into`, **42**
`FromIterator`, **414**
`FromStr` / `str::parse`, **457**
Función, 467, **475**

G

Generic, **27**
Glob import, **876**
Golpe, 640, 770, **771**
Grabar desde un teclado MIDI, 315
Grado conjunto, salto, salto grande, **587**
Guardado automático, 98

H

`HashMap<K, V>`, **51**
Head, **749**
Hilo de audio, 22, 51, 196
Hoja de acordes, 529, 661

I

Idioma, 171
`if let` / `while let`, **101**
`ilog2`, **234**
`impl Trait`, **42**
Importar una grabación, 600
Inner attribute (`#![...]`), **914**
Instrumento transpositor, **834**, 846
Integración continua, 914
Invariant, **539**

Inversión, 447, 452, 456, **464**, 487, 820, 825, 826
`is_some_and`, **483**
Iterator, **908**

J

`JoinHandle<T>`, **350**

L

Lazy, **493**
Leading underscore (`_name`), **350**
`let ... else`, **59**, 60
Let chain, **22**, 29, 61, 143, 307, 595, 790
Lineup Osc, 249
Lista de reproducción, 36, 161
Literal in a pattern, **312**

M

Marca de tonalidad, 441
`match`, **22**
Match guard, **462**
`matches!`, **414**
`mem::replace`, **22**
`mem::swap`, **70**
`mem::take`, **196**
Method, **17**
Métrica, *véase* Compás
Mezclador, 276
Micromodo, 530, 532, **565**, 634, 887
`min_by_key`, **524**
mixtura modal, *véase* Acorde prestado
Modulación, *véase* Cambio de tonalidad
Module, **12**
Monomorphization, **22**
Move, **22**
`move` closure, **350**
Movimiento contrario, **588**
Movimiento oblicuo, **588**
Movimiento paralelo, **588**
`mpsc` channel, **298**
Música latinoamericana, 874, 891
`Mutex<T>` and poisoning, **298**

N

Newtype, 200, 232, **417**, 539
Nombres de acordes, 446
Nomenclatura de notas, 171
Número romano, 467, **468**

O

Oír acordes, 501
`?` operator, **22**
`Option::and_then`, **147**
`Option::as_mut`, **29**
`Option::filter`, **298**
`Option::take`, **350**
`Option<T>`, **22**, 26
Or-pattern, **102**
Orquesta, 732
Ownership, **22**

P

Panic, **19**, 475
Paquetes de cada versión, 922
`partition`, **573**
`partition_point`, **534**
Partitura guía, 834
Patrón, 36
Piano roll, 131
Pista, 161
Pista bloqueada, 763
Poliacorde, **501**
Posición cerrada, **724**
Práctica, 804
Preferencias, 122, 268
Property-based test, **62**
`pub(crate)`, **41**
`pub(super)`, **80**
Pure function, **487**

R

Rack de canales, 131

RAII / `Drop`, **22**
Range, **198**
`Rc<T>`, **534**
Reconocimiento de acordes, *véase* Oír acordes
Recursion, **543**
`rem_euclid`, **414**
`Result<T, E>`, **22**, 26
`retain`, **39**
RON, **80**, 82, 942
`rposition`, **579**
`rsplit_once` / `split_once`, **456**

S

Salida MIDI, 320, 324
Sampler, 243
`saturating_sub`, **496**
Sección de vientos, 722
`Send`, **28**, 359
serde, **80**, 82
serde container attributes, **539**
serde field attributes, **82**
Serie dodecafónica, **537**, 550, 556, 634
Sexta napolitana, **481**
Silencio, 770, **771**
Slice, **28**, 683, 774
Slice pattern, **298**
Slice pattern with `@`, **575**
Solfeo, 171, **172**
Solista, 779
Sostenido, 770, **771**
SoundFont, 256
State machine, **237**
`'static`, **28**, 359
`static` item, **642**
`step_by`, **496**
Struct update syntax, **102**
Swing, **229**, 399, 649, 761, 779, 794, 798, 860, 861, 863, 866

T

Tango, 874, 891
Target, **914**, 920
Teclado MIDI, 303
Teclas de armonía, 475, 528
Tempo, 210
Tercera de picardía, **481**
`thread::Builder`, **298**
`thread_local!`, **473**, 474
Tonalidades enarmónicas, **434**
Tonicización, **481**
Traducción, *véase* Idioma
Trait, **18**, 845
Trait bound, **22**, 27, 72
Transporte, 206, 228
Traza de reproducción, 383
`TryFrom` / `TryInto`, **80**
Tuple, **416**
Tuple comparison, **294**
Tuple struct, **16**, 17
Turbofish (`::<>`), **27**
Turnaround, **749**
Type alias, **643**

V

Variaciones, 753
`VecDeque<T>`, **70**
Velocidad, 156, 886
Ventana Acerca de, 127
Versión, 127, 922
Voicing, **697**

W

Walking bass, **697**
`windows`, **908**
Workspace, **12**, 14
`wrapping_add` / `wrapping_mul`, **543**

Z

`zip`, **51**
Zoom, 111