PFA Studio

User Manual — adaptive audio for games

September 2026 · beta · figures rendered from the live editor

PFA Studio — User Manual

PFA is an adaptive audio suite for games. You author interactive music and sound effects visually in **PFA Studio**, export one folder of encoded audio plus a `config.json`, and the **PFA Engine** (available for Web / JavaScript and for Godot) plays it back in your game — switching intensity levels, crossfading sections, randomizing variations and positioning sounds in space, all driven by a handful of one-line calls from game code.

The workflow in one sentence:

> **Author in Studio → Export for Game → `trigger()` / `setLevel()` from your game.**

Chapters

#	Chapter	What it covers
1	Getting Started	Install, the main window, projects, transport, shortcuts
2	Events & Groups	Music vs SFX events, the event tree, groups
3	Levels	Intensity tiers, level crossfades, per-level state
4	Tracks, Samples & Clips	The timeline, waveform trimming, loops, sub-levels
5	Variations & Randomization	Sample variations, play chance, per-trigger randomization
6	Advanced Mode: the Node Graph	Sections, transitions, tempo changes, stingers
7	Event Settings	Fades, loop modes, spatial attenuation, reverb send, intensity
8	Reverb Zones	The project reverb library, Web/Godot characters
9	Preview & Mixing	Transport preview, the event mixer, meters
10	Live Connect	Real-time tweaking against a running game
11	Export for Game	Formats, quality, SFX→WAV, sync folder, .pfabank
12	Packs	Sharing and importing .pfapack event libraries
13	Tools & Settings	Collect Samples, themes, the AI Automation Bridge, update notifications, troubleshooting
14	Appendix	config.json reference, shortcuts, engine parity notes

The demo projects

The figures in this manual are captured from the PFA demo projects — each one exists as a real playable scene and shows a feature set in context:

Project	Demonstrates
Adaptive Music	The node graph: five sections, per-direction transitions, a stinger, free-tempo and fixed-tempo sections (ch. 6)
Footsteps	SFX variations and surface switching via levels (ch. 3, 5)
Fox Run	A complete game: grouped events, three musical states, sub-levels, 2D spatial SFX (ch. 2, 4)
3D Simple	3D spatial loops (seamless vs smart), reverb sends, a reverb zone (ch. 7, 8)
Billiards	Intensity (hit strength), 3D spatial one-shots, the pack workflow (ch. 7, 12)

Conventions

- **Event** — one named sound your game talks to (`music_forest`, `cue_strike`).
- **Level** — an intensity tier / alternate state inside an event.
- **Track** — one lane of audio inside a level.
- **Variation** — one of several interchangeable samples on a track.
- **Clip** — the trimmed window of a sample that actually plays.
- Keyboard shortcuts are written mac-style (`Cmd+S`); PFA Studio ships for macOS and Windows — on Windows read `Cmd` as `Ctrl`.

1. Getting Started

1.1 What PFA is

PFA replaces the "export a stereo loop, hope for the best" workflow with an adaptive one. In PFA Studio you build **events** — a piece of interactive music with intensity levels and transitions, or a sound effect with variations, randomization and spatial behavior. **Export for Game** writes a folder of encoded audio plus a `config.json` describing all of it. In the game, the PFA Engine (JS for web, GDScript addon for Godot) loads that folder and your code stays tiny:

```
PFA.play("music_forest")           # start the music event
PFA.set_level("music_forest", 2)   # raise intensity
PFA.trigger("footsteps")           # one footstep, auto-varied
```

```
pfa.play('music_forest');
pfa.setLevel('music_forest', 2);
pfa.trigger('footsteps');
```

Everything you author — fades, probabilities, loop lengths, attenuation curves — lives in the exported config, not in game code. Change the mix, re-export (or press **Sync**), and the game sounds different without touching a line of it.

1.2 Installing

PFA Studio is in public beta on macOS and Windows. It has been tested on macOS 26 (Tahoe) and macOS 27.0.1 on Apple Silicon and on Windows 10 (22H2, 64-bit). The license terms are shown on first launch and stay one click away under **Help** → **Beta License (EULA)**; this manual describes the product, not the terms.

macOS

1. Download `PFA-Studio-<version>.zip` and unzip it.
2. Drag **PFA Studio.app** to Applications (or run it from anywhere).
3. Double-click to launch. Beta builds aren't signed with an Apple Developer ID yet, so the first time macOS stops the app with **"PFA Studio" Not Opened** — click **Done** (not Move to Bin).
4. Open **System Settings** → **Privacy & Security** and scroll to **Security**: next to "*PFA Studio*" *was blocked to protect your Mac*, click **Open Anyway**.
5. macOS asks once more — **Open "PFA Studio"?** — click **Open Anyway** (and confirm with your password or Touch ID if asked).

This is needed once per build — a new version asks again; after that it launches normally.

Windows

1. Download and run `PFA-Studio-Setup-<version>-win64.exe`.
2. Follow the installer (it associates `.pfa` project files on the way).

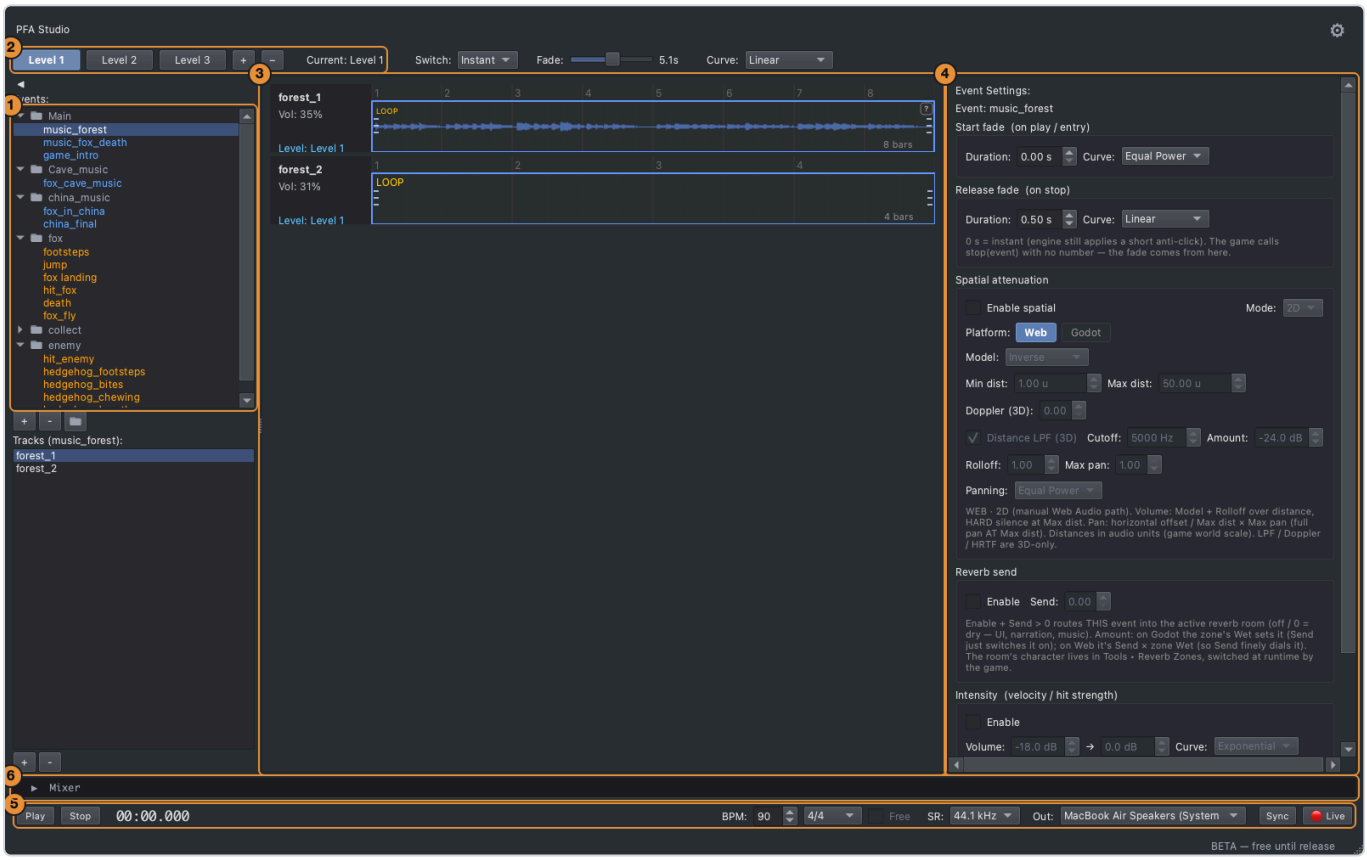
3. Launch **PFA Studio** from the Start Menu. If SmartScreen objects to an unsigned app, choose **More info** → **Run anyway** once.

On Windows 11 with **Smart App Control** on, Windows blocks unsigned installers and has no way to allow a single app. Installing means turning it off under **Windows Security** → **App & browser control** → **Smart App Control settings**; recent Windows updates let you turn it back on later without reinstalling Windows.

If anything goes wrong, the app writes a crash log — attach it to a bug report:

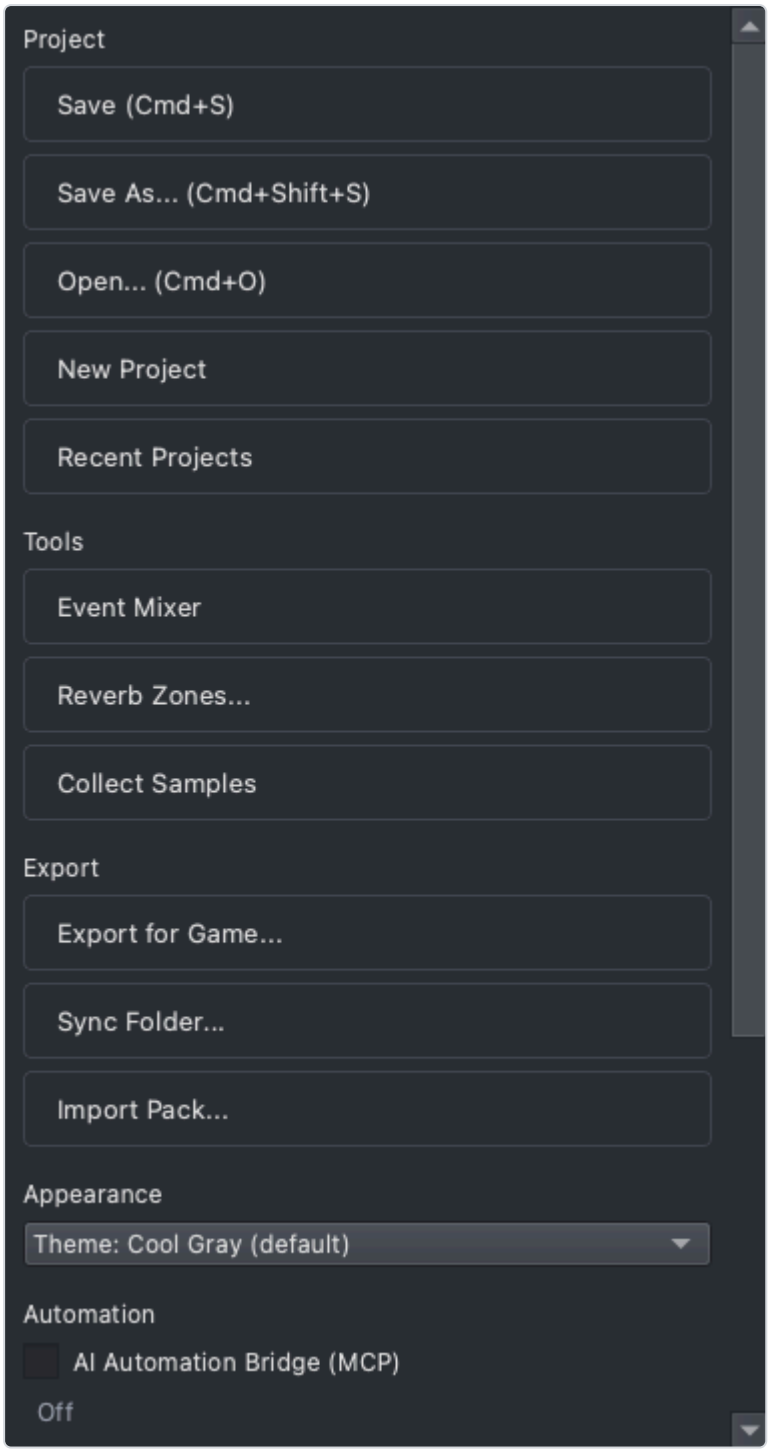
- macOS: `~/Library/Logs/PFA Studio/crash_log.txt`
- Windows: `%LOCALAPPDATA%\PFA Studio\Logs\crash_log.txt`

1.3 The main window



#	Area	What it does
1	**Events**	Every sound in the project — music and SFX, organized into groups (ch. 2)
2	**Levels bar**	The selected event's intensity levels + level-switch fade controls (ch. 3)
3	**Timeline**	Tracks and waveforms of the current level; trim, loop and fade clips here (ch. 4)
4	**Inspector**	Event Settings for the selected event, or Track Details when a track is selected (ch. 5, 7)
5	**Transport**	Play/Stop, position, tempo, sample rate, output device, Sync and Live Connect
6	**Mixer panel**	Collapsible track mixer for the selected event, with a master strip and LUFS metering (ch. 9)

All project commands live behind the ⚙ button in the top-right corner (the menu bar only carries the standard macOS About entry and a Help menu — manual, online docs, licenses, bug reports):



- **Project** — `Save (Cmd+S)`, `Save As… (Cmd+Shift+S)`, `Open… (Cmd+O)`, `New Project`, `Recent Projects`
- **Tools** — `Event Mixer`, `Reverb Zones…`, `Collect Samples`
- **Export** — `Export for Game…`, `Sync Folder…`, `Import Pack…`
- **Appearance** — theme preset (five presets, applied on restart — see ch. 13)
- **Automation** — the AI Automation Bridge toggle (drive the editor from Claude Code — see ch. 13.4)
- **Info** — license status, the **Check for new versions** toggle (ch. 13.5), About, manual and online documentation
- Bottom row — `Restart App`

1.4 Projects

A PFA project is a folder: `MyGame/MyGame.pfa` plus a `samples/` directory next to it. The `.pfa` file is JSON — friendly to git and to backups.

- **New Project** starts an untitled project with one Music and one SFX event to explore, at 48 kHz.
- **Save** the first time acts as Save As and creates the `<name>/<name>.pfa` folder layout.
- **Open** reads `.pfa` (and legacy `.amp`) files; recently used projects are in **Recent Projects**.
- The window title shows the project name with a dirty marker while there are unsaved changes (macOS shows a dot in the close button). Closing, opening another project or restarting with unsaved edits asks **Save / Discard / Cancel** — you can't lose work silently.

Sample paths are stored relative to the project folder whenever the files sit inside it. Run ⚙ → **Tools** → **Collect Samples** to copy every referenced file into `<project>/samples/` — after that the project folder is fully self-contained and moves between machines (or into a cloud folder) safely. Collect Samples requires the project to be saved first.

1.5 Transport



#	Control	Notes
1	**Play**	`Space` toggles. Previews the selected event as the engines will play it — with the differences ch. 9.1 lists
2	**Stop**	`Esc` also stops; ends a long or looping SFX preview too
—	**Position**	`MM:SS.mmm`
3	**BPM**	60–200. The event's default tempo; node-graph sections can override it with a Tempo node (ch. 6)
4	**Time signature**	`4/4 3/4 6/8 2/4 5/4 7/8` or `Custom…` (any beats-per-bar 1–32 over 2/4/8/16)
5	**Free**	Free-tempo indicator — the loop follows file length instead of a BPM grid and the timeline switches to seconds. Free tempo is authored per section with a **Tempo node** (ch. 6); in Simple mode this box is read-only
6	**SR**	Project *preview* sample rate, 44.1/48 kHz. Source files are never modified; export quality is chosen separately at export time. New projects default to 48 kHz
7	**Out**	Output device for preview
8	**Sync**	One-click incremental export to your game folder (ch. 10–11)
9	🔴 **Live**	Live Connect — real-time parameter sync with a running game on port 9264 (ch. 10)

1.6 Keyboard shortcuts (main window)

Shortcut	Action
`Space`	Play / Stop
`Esc`	Stop
`Cmd+S`	Save
`Cmd+Shift+S`	Save As
`Cmd+O`	Open
`Delete` / `Backspace`	Remove selected track (when the track list has focus)
`Ctrl+wheel`	Zoom the timeline (1–8×)

The node editor adds its own shortcuts — see ch. 6.

1.7 Where to go next

- Building interactive **music**? Read ch. 3 (Levels) and ch. 4 (Tracks), then graduate to ch. 6 (Node Graph).
- Designing **SFX**? Read ch. 4 and ch. 5 (Variations), then ch. 7 for loops, spatial and intensity.
- Ready to **ship**? Ch. 11 (Export) and ch. 10 (Live Connect) close the loop with your game.

2. Events & Groups

2.1 What an event is

An **event** is the unit your game talks to. Everything else in PFA — levels, tracks, variations, fades — lives *inside* an event. There are two kinds:

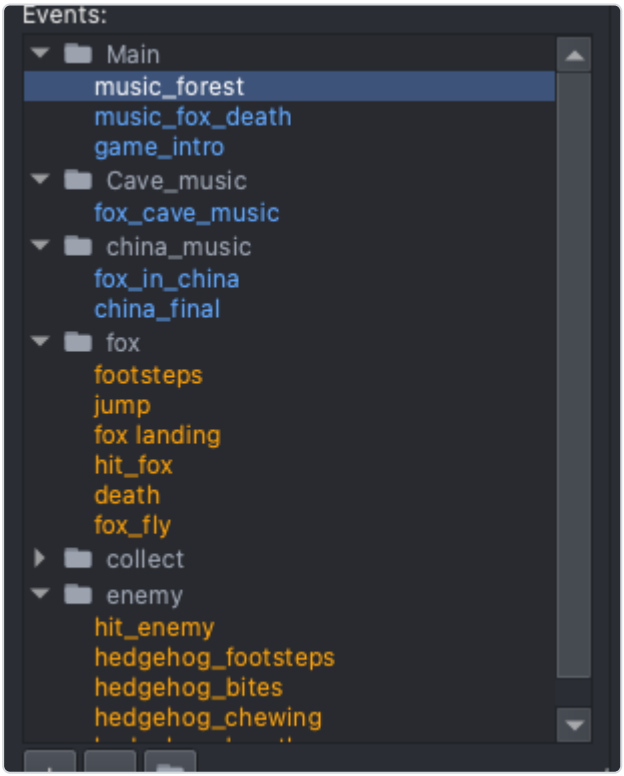
- **Music** — continuous, tempo-aware, organized in intensity **levels** (simple mode) or as a **node graph** of sections and transitions (advanced mode, ch. 6). The game starts/stops it and moves it between states.
- **SFX** — triggered sounds: one-shots (footsteps, hits, UI) or sustained loops (campfire, engine). The game calls `trigger()` per occurrence, or creates an emitter for a positioned loop.

In game code:

```
PFA.play("music_forest")         # music: start the event (entry fade applies)
PFA.set_level("music_forest", 1) # music: switch intensity level
PFA.trigger("footsteps")         # sfx: play once (variations handled by PFA)
PFA.stop("music_forest")         # stop with the event's release fade
```

```
pfa.play('music_forest');
pfa.setLevel('music_forest', 1);
pfa.trigger('footsteps');
pfa.stop('music_forest');
```

2.2 The event tree



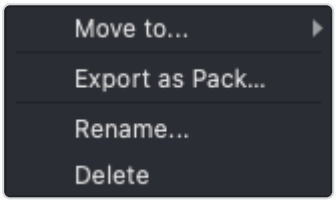
The left panel lists every event, foldered into groups (here: the Fox Run demo — `Cave_music`, `china_music`, `fox`, `collect`, `enemy`). Click an event to load it into the timeline and inspector. Clicking the event again after working in a track always returns the inspector to Event Settings.

Below the tree:

- `+` — add an event. You choose the type once at creation: `Music (Simple)`, `Music (Advanced)` (starts with a node graph seeded `Intro → Loop 1 → Loop 2`, ready to rewire), or `SFX`.
- `–` — delete the selected event.
- the **folder icon** (tooltip *Add group*) — add a group.

Drag events to reorder them or drop them into a group.

2.3 The context menu



Right-click an event:

- **Move to…** — submenu with `(No group)` and every existing group.
- **Export as Pack…** — package this one event as a `.pfapack` (ch. 12).
- **Rename…**
- **Delete**

Right-click a group for **Export Group as Pack…**, **Rename…** and **Delete Group** (its events return to the root — they are not deleted).

2.4 Groups

Groups are folders with a mixer role: each group has its own **volume, mute and solo** — visible as group headers in the event mixer (ch. 9). Use them the way you'd use buses in a DAW: `music`, `ui`, `enemies`, `ambience`.

Group volume is exported with the config, so a game-wide "music at –6 dB" decision is an authoring decision, not a code change. Over Live Connect (ch. 10) an **event's** solo ducks everything else in the running game; a **group's** solo stays local to Studio monitoring — ch. 10 lists exactly what travels.

Groups also define the natural unit for sharing: **Export Group as Pack** turns a curated group (say, `Footsteps`) into a single file another PFA user can import (ch. 12).

3. Levels

3.1 The idea

A **level** is one state of an event. For music, levels are intensity tiers — *calm → build → peak* — each with its own tracks and mix. For SFX, levels are alternate sample sets behind one trigger — the classic example is footstep surfaces:



The Footsteps demo's `footsteps_walk` event has four levels — one per surface. The game keeps triggering the same event; switching the level switches what plays:

```
PFA.set_level("footsteps_walk", 2)   # stepped onto stone
PFA.trigger("footsteps_walk")        # each step
```

```
pfa.setLevel('footsteps_walk', 2);
pfa.trigger('footsteps_walk');
```

Levels are indexed from 0 in game code, left to right as shown in the level bar.

3.2 Working with levels

The level bar sits above the timeline:

- Click a level tab to select it — the timeline below always shows the **current level's** tracks. The active tab is highlighted, and the `Current:` label confirms it.
- `+` adds a level, `–` removes the last one *(Simple Music and SFX; an Advanced Music event manages levels as Section nodes — add and remove them in the node editor, ch. 6)*.
- Each track decides per level what to play (per-level sample sets) or can play the same material on every level — that's the track's *level behavior*, ch. 5.

During preview, switching tabs is the same operation your game performs with `set_level()` — you hear the transition the players will get, with the differences ch. 9.1 lists (among them: the tab that already plays, clicked while a switch waits, cancels that switch here and is ignored in-game).

3.3 Level switching (music)

For music events the level switch is a musical crossfade, controlled by the right side of the level bar:

- **Switch** — quantization of the moment the switch happens: on the next **Bar**, next **Beat**, or **Instant**.
- **Fade** — crossfade duration, 0–10 s (double-click the slider to reset to 2.0 s), with a live readout.
- **Curve** — `Equal Power`, `Linear`, `S-Curve`, `Exponential`, `Logarithmic`.

These three settings are **global for the event** — every level switch uses the same crossfade. When you need different transitions per direction (e.g. a fast rise into combat, a slow drop out of it), that is authored with fade nodes in the node graph (ch. 6), which give you a fade per *transition* rather than per level.

Two details that make switches feel professional:

- **Phase-synced entry.** Levels stay aligned to the same musical timeline: when you enter a level mid-bar, it continues *from the equivalent position*, not from its beginning. Fresh-start behavior is an explicit opt-in per node in the node graph (`Restart on entry`, ch. 6).
- **Per-level playback state.** Each level remembers its own variation rotation and play-chance dice, and that state keeps advancing in the background while another level is audible — re-entering a level doesn't restart its musical "story". One difference to know: the game engines seed every level's dice at `play()` and tick them all from the start, while the editor's preview seeds a level the first time it is heard and simulates only levels it has already played — a level's *first* entry can land on a different variation in preview than in the game; later re-entries agree (ch. 9.1).

3.4 Simple mode vs the node graph

The level system *is* adaptive music for most games: a handful of tiers, one crossfade rule, `set_level()` from code. When you need different transitions per direction, tempo changes between sections, one-shot stingers on top of transitions, or probability-driven routing — switch the event to the **node graph** (ch. 6). Levels remain the foundation: each node in the graph plays one level's tracks.

4. Tracks, Samples & Clips

4.1 Tracks

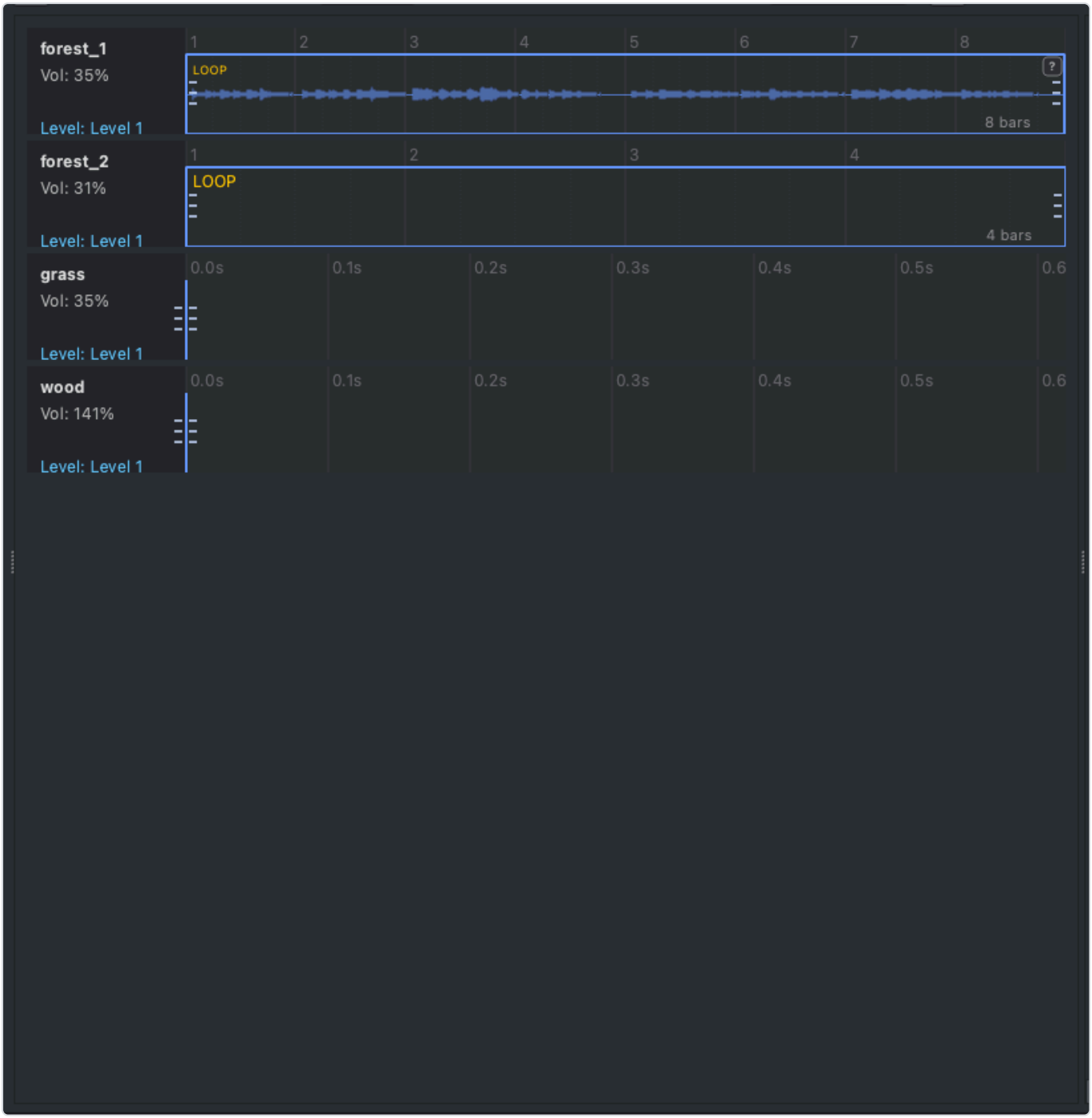
Tracks are the lanes inside a level. A music level might have `drums`, `bass`, `strings`; an SFX event often has just one track with many variations.

- `+` / `−` under the **Tracks:** list add and remove tracks.
- Click a track in the list — or click its lane in the timeline — to open **Track Details** in the inspector (volume with double-click-to-0 dB reset, **Mute** / **Solo**, and everything in ch. 5).
- **Drag tracks in the list to reorder them**; the timeline lanes follow. Order is purely visual — it doesn't change the mix.

For SFX events the track panel also carries two event-wide voice rules:

- **Cooldown** (0–5000 ms) — minimum time between two triggers of this event; protects against machine-gun retriggering.
- **Max voices** (1–32, default 8) — polyphony cap. When the cap is hit, the **least audible** voice is released to make room (quietest first, the oldest on a tie), and a new trigger quieter than every playing voice is rejected instead of evicting one. An evicted voice ends through a short fixed anti-click seam (20 ms; 30 ms for Godot spatial voices), not the event's Release fade — Release belongs to the game's `stop()` and to emitter teardown (ch. 7.2).

4.2 The timeline



Each lane shows the track header (name, volume, `MUTED` / `SOLO` state, level) and the waveform of the current level's active sample.

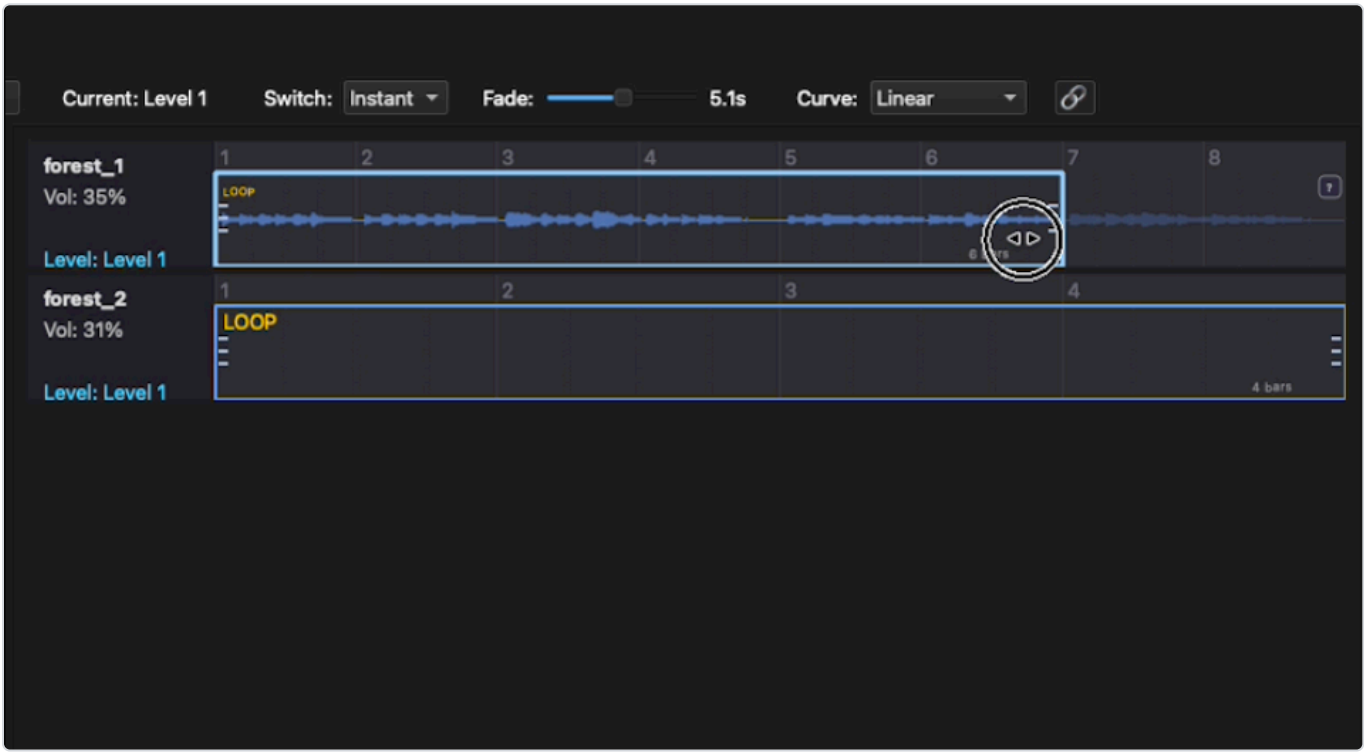
- The ruler shows **bars & beats** for music, or **seconds** for SFX and free-tempo events.
- `Ctrl+wheel` zooms 1–8×.
- The red playhead tracks preview playback, loop-aware.

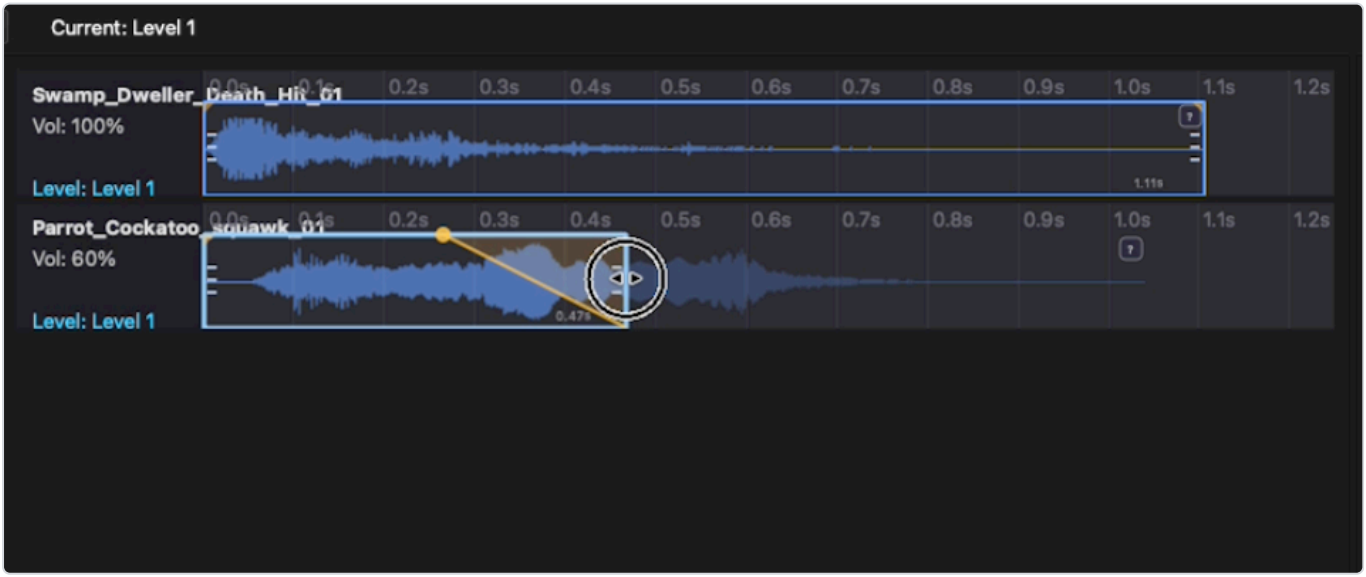
4.3 The clip frame



Every sample plays through a **clip** — a trim window over the source file. The waveform is drawn full-length; the part inside the bright frame is what plays, the dimmed part is trimmed away.

- **Drag the frame edges** to trim the start/end. The full source stays on disk untouched — trims are non-destructive and re-editable.
- **Fade handles** (the orange corner markers on SFX tracks) drag out a fade in / fade out inside the clip.
- `LOOP` appears top-left when looping is enabled for this region.
- Hover the `?` badge (bottom-right) to see the source filename.





4.4 Samples

Add files with **Add…** in the track's sample list (ch. 5), or drag them from Finder straight onto the window. On import PFA:

- up-mixes mono → stereo,
- resamples to the project preview rate if needed (fast polyphase; cached),
- keeps WAV files untouched on disk — the project references them in place (**Collect Samples**, ch. 13.1, copies them into the project when you want that).

Importing MP3, OGG, FLAC and friends

PFA edits and exports from WAV, but the import door takes the common compressed formats too: **MP3 / OGG / OPUS / FLAC / AIFF / M4A / AAC / WMA / CAF** — and an *All Files* pick of something more exotic still gets an honest try (the decoder sniffs the file's content, not its extension).

A non-WAV file is decoded **once** into a 24-bit WAV copy named `<name>_<format>_<source-id>.wav` (`boom.mp3` → `boom_mp3_1f3a9c02.wav`; the short hex tail fingerprints the source's location, so same-named files from different library folders don't fight over one copy), and the project uses only that copy from then on. The one-time copy is deliberate: lossy decoders differ by a few tens of milliseconds between builds, so decoding fresh on every load could shift your trims — the copy freezes the decode forever, and the project stops depending on a Downloads folder surviving. The original file is never modified. Re-importing the same file reuses the existing copy; replace the source with a newer version and the next import re-converts.

Where the copy lands:

- with a saved project — in the project's `samples/` folder, next to collected samples;
- before the first save there is no project folder yet, so the copy goes to the studio's user folder (`~/.pfa_studio/imported_samples/`). After you save, run **Collect Samples** (ch. 13.1) to gather those into `samples/`.

The source sample rate is kept, so no extra resample generation is added. At export the copy encodes exactly like any WAV source (ch. 11). One honest cost: MP3 in → OGG out at export is a second lossy generation — inaudible for typical SFX, but a WAV or FLAC master is always the better source when you have one.

Per-sample **volume** is set on the sample row; at export it is *baked into the rendered audio* together with clip volume, so engines don't pay a runtime cost (ch. 11).

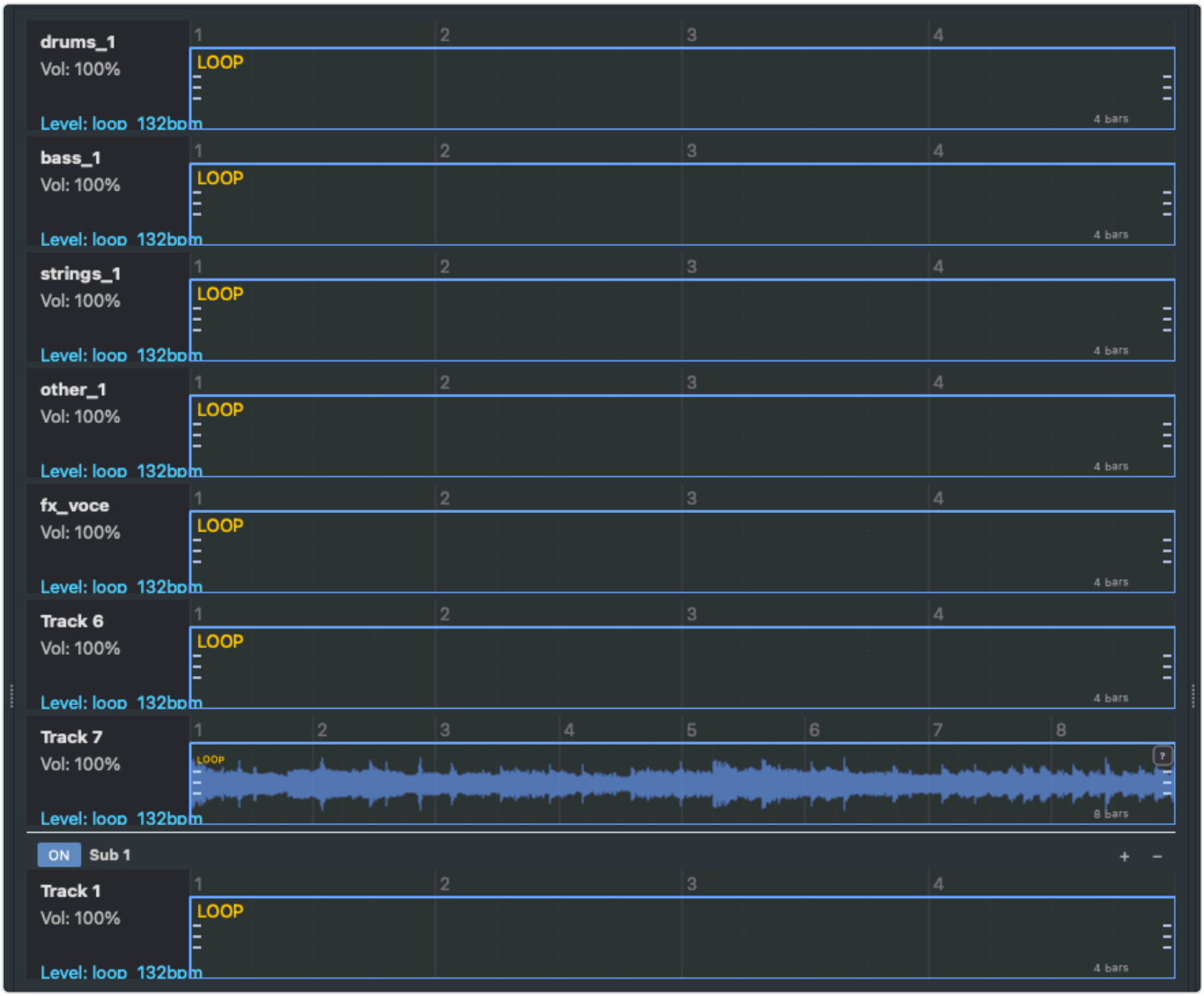
4.5 Loops & loop length

For music tracks, the **Region / Loop length** control in Track Details sets the musical length of the loop: pick `1–64 bars` from the combo or type any bar count. The cyan auto-label shows the length PFA detects from the trimmed clip at the event's BPM — with a clean trim, the detected value is already right.

The exported loop boundary is always computed from the **clip's trimmed length at the event's tempo**, so what loops in your game is what loops in the preview. While a transition fades a looping clip out, the clip keeps looping under the fade on all three players — audio beyond the trimmed window (a tail after the loop point) is not played during the fade.

SFX loops (sustained sounds like fires and hums) are enabled per event, not per clip — see **Loop (sustained SFX)** in ch. 7.

4.6 Sub-levels

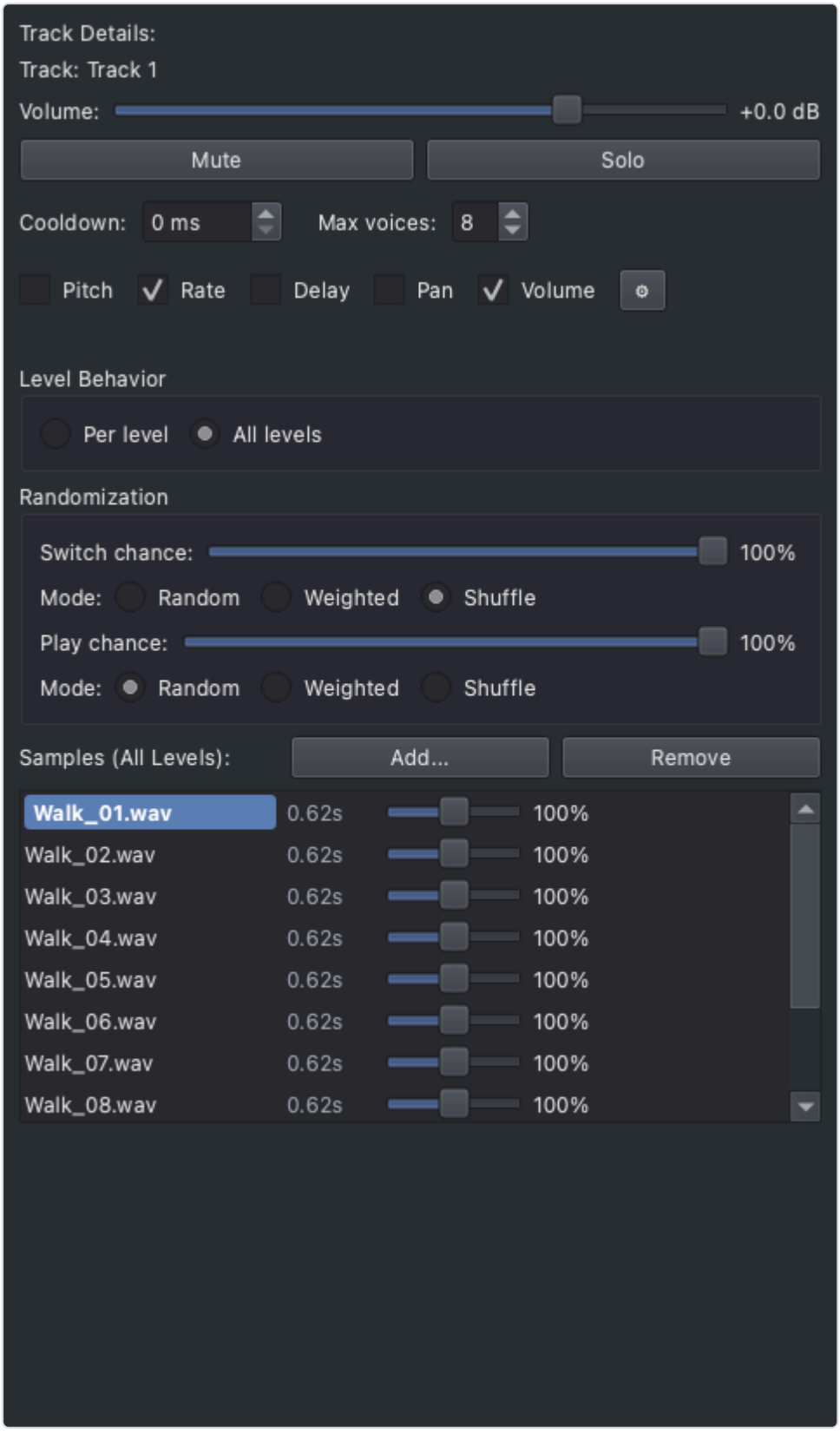


A **sub-level** is a layered variant hanging under a level: the base level keeps playing and the sub adds (or swaps in) material on top — hierarchical intensity inside one level. Sub-levels are created in the node editor (`+ Sub`, ch. 6.7); their tracks appear in the timeline as extra lanes under the parent with an `ON` / `OFF` toggle per sub.

At export, sub-levels are flattened into the config, and the game toggles them at runtime like a mini-level inside the level.

5. Variations & Randomization

Repetition is the enemy. This chapter is everything PFA does so that the hundredth footstep doesn't sound like the first — all of it authored per track in **Track Details** (select a track to see it).



5.1 Level behavior

Two radio buttons decide where a track's samples live:

- **Per level** — each level has its own sample set (the footsteps demo: four surfaces × several step recordings each).
- **All levels** — one shared set that plays on every level (a heartbeat that persists across intensity tiers).

5.2 Variations

Samples: lists the current level's variations — add several takes of the same sound with **Add…**. Each row has its own volume; in *Weighted* mode a weight controls how often it wins the draw.

Switch chance (0–100%, double-click resets to 100) is the probability that the *next cycle picks a new variation* instead of keeping the current one. The switching algorithm:

- **Random** — uniform, repeats allowed.
- **Weighted** — respects per-sample weights.
- **Shuffle** — no-repeat bag: every variation plays once before any repeats; the musical default.

Play sample `N` before switching keeps a chosen variation for N cycles — useful when a musical phrase needs to establish itself before rotating.

5.3 Play chance

Play chance (0–100%) makes a track *sometimes stay silent* — the core trick of generative layering: give a percussion overlay 40% and it enters and leaves organically.

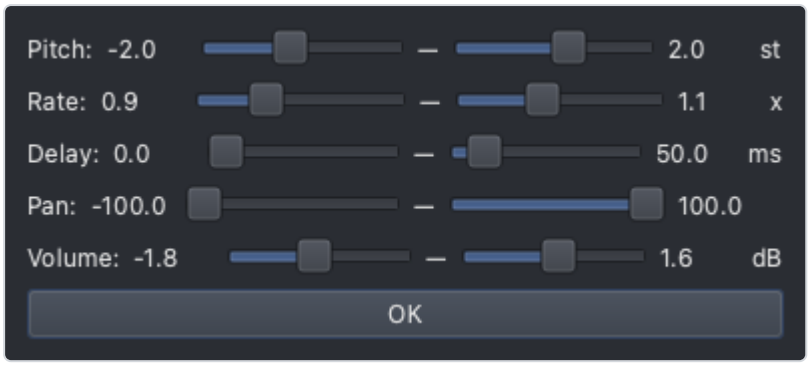
- Modes: **Random** (true random — every roll independent), **Weighted** (weighted history — misses raise the odds, avoids long droughts), **Shuffle** (shuffle bag — a ten-slot bag holding the rounded share of hits, so every full cycle lands on the nearest 10%: 40% plays exactly 4 of 10, 25% plays 3 of 10).
- **Reroll every `N`** — for music, how many loop cycles between dice rolls.
- The ⚙ next to it opens **Chance Transition**: when a roll silences or un-silences the track, apply it **Instant (10 ms anti-click)** or as a smooth **Fade** (0.1–5.0 s, default 0.5 s).

> **Engine note.** On both engines this fade plays in both directions (when a roll silences *and* when it un-silences the track) no matter how many variations the track has. A variation switch that lands inside the fade hands it over seamlessly — the ramp finishes on its original schedule.

Per-level playback state (ch. 3.3) means each level's bag, history and dice keep advancing even while another level is audible.

5.4 SFX per-trigger randomization

For SFX, four per-trigger randomizers vary every single trigger. Enable each dimension with its checkbox in Track Details, then set ranges in the ⚙ **Randomization Settings** dialog:



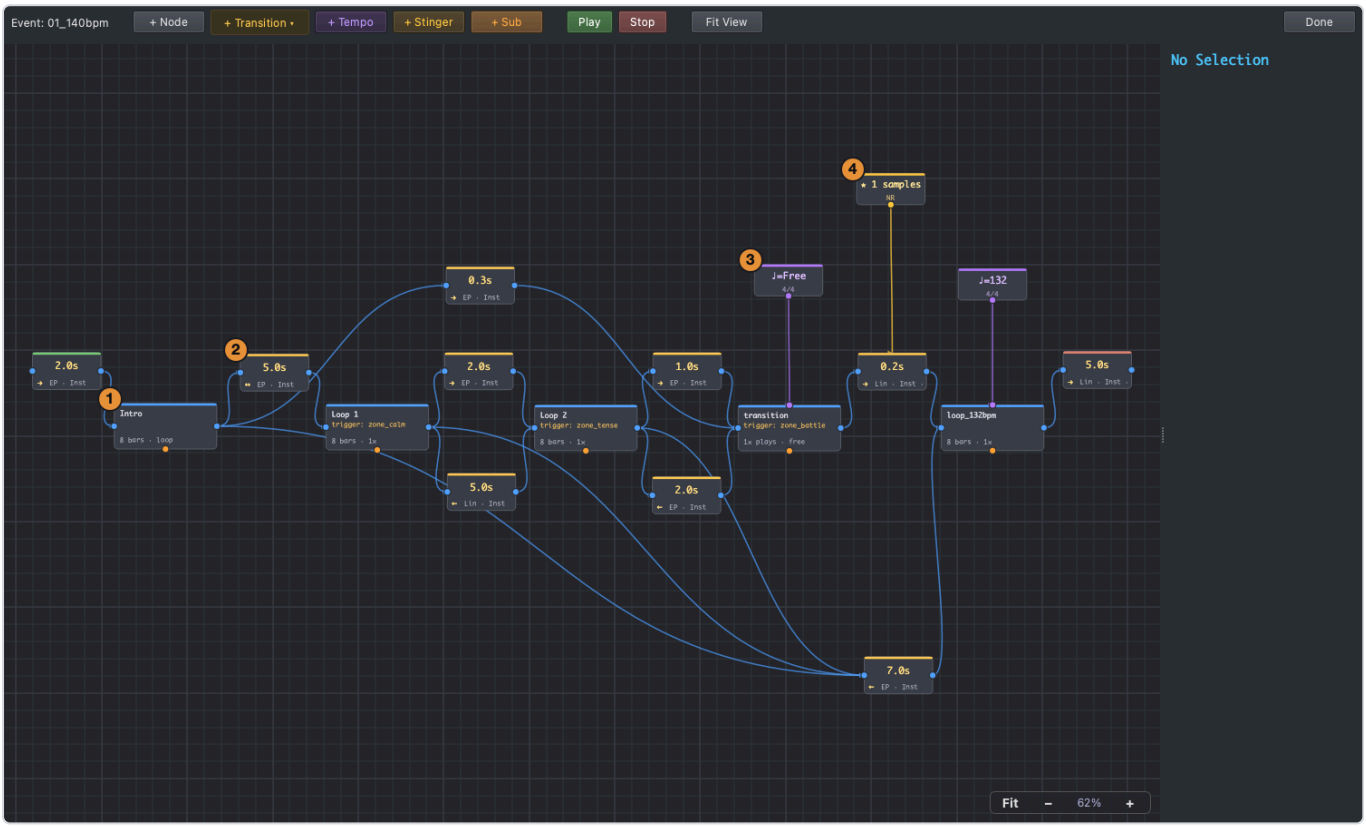
Dimension	Range
Pitch	±12 semitones
Rate	0.5–2.0× playback speed
Delay	0–500 ms pre-trigger delay
Volume	±24 dB

Each trigger rolls fresh values inside the enabled ranges — a footstep event with ±2 st pitch and ±3 dB volume never plays the same twice even with a single sample. Randomization stacks *on top of* variation selection, and intensity (ch. 7.6) stacks on top of both.

For **smart loops** (ch. 7.3), each loop cycle counts as a trigger: the seam re-rolls randomization and can rotate variations — a campfire that never audibly loops.

6. Advanced Mode: the Node Graph

Levels + one crossfade rule cover a lot of games. The node graph is for when music needs *structure*: different transitions per direction, tempo changes between sections, stingers over transitions, probability routing, intros and outros. You lay the music out as a map — sections as nodes, transitions as the roads between them — and the game just names destinations.



The figure shows the **Adaptive Music** demo: ① loop sections (`Intro → Loop 1 → Loop 2 → transition → loop_132bpm`), ② transition nodes carrying the fades, ③ tempo nodes (one `♩=Free`, one `♩=132`), ④ a stinger attached to a transition. Three things to notice before the reference:

- **Rises and falls are different.** Each step up uses its own short transition (0.2–2 s); the falls use long ones (5–7 s). Games rise abruptly and cool down slowly — the graph makes that trivial to author.
- **Transitions are reusable.** Several sections route their way down through *one shared 7-second fade node*. Author the descent once; connect every source to it.
- **Sections own their time.** The `transition` section is free-tempo (a riser with no grid); `loop_132bpm` runs at its own 132 BPM. The graph re-quantizes as playback crosses them.

6.1 Opening the editor

Create the event as **Music (Advanced)**, or press the **Nodes** button in Event Settings on an advanced event. The window is three parts: toolbar, canvas, and a properties panel on the right that shows the selected node.

6.2 Toolbar

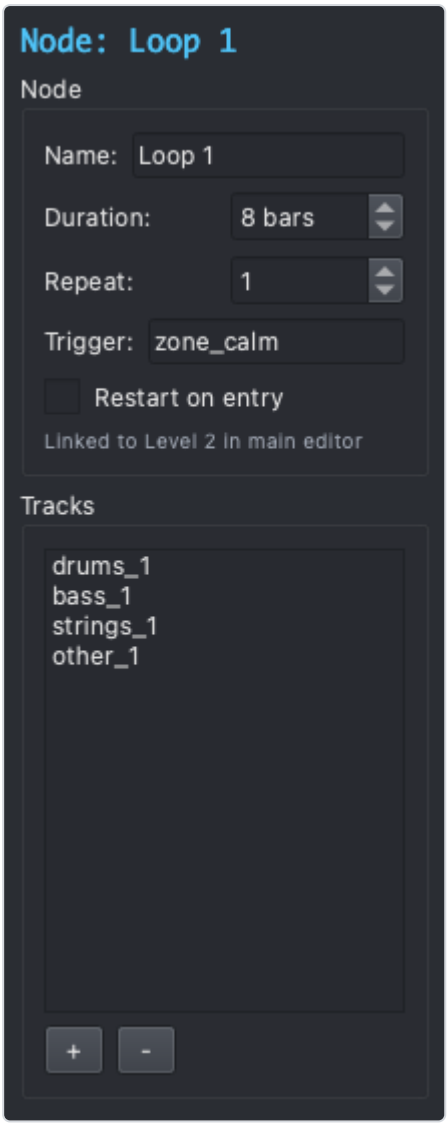
Button	What it places
`+ Node`	A loop section
`+ Transition ▾`	`Crossfade` / `Fade In` / `Fade Out` node
`+ Tempo`	A tempo override node
`+ Stinger`	A stinger (one-shot over a transition)
`+ Sub`	A sub-level under the selected section (ch. 6.7)
`Play` / `Stop`	Preview with live node highlighting
`Fit View`	Frame the whole graph
`Done`	Close (the graph lives in the event; save the project as usual)

Placement is ghost-style: click the button, a translucent ghost follows the cursor, click to drop it, right-click or `Esc` to cancel. The empty-canvas right-click menu carries a placement subset — `Add Section`, the three `Transitions`, `Add Sub-level Node` (plus `Fit to View`); tempo and stinger nodes are placed from the toolbar.

Canvas navigation — pan with middle-drag; zoom with the bottom-right overlay (`Fit – 100% +`) or the keyboard: `+` / `=` in, `–` out, `0` fit. `Space` toggles preview. Delete a selected node or connection with `Delete` / `Backspace`.

6.3 Loop nodes (sections)

A loop node plays one level's tracks. Its face reads like `8 bars · loop` (infinite), `8 bars · 1×` (plays once), or `loop · free` / `1× plays · free` when a free-tempo node is attached.



Property	Meaning
Name	Section name shown on the node
Duration	Section length in bars (1–128)
Repeat	How many passes before the section is "done" (0 = Infinite)
Trigger	Trigger name the game sends via `navigate()` (ch. 6.8) to steer playback toward this section
Restart on entry	Start this section from position 0 instead of phase-synced entry. Use for hard cuts or when the section's tempo/length differs from its source

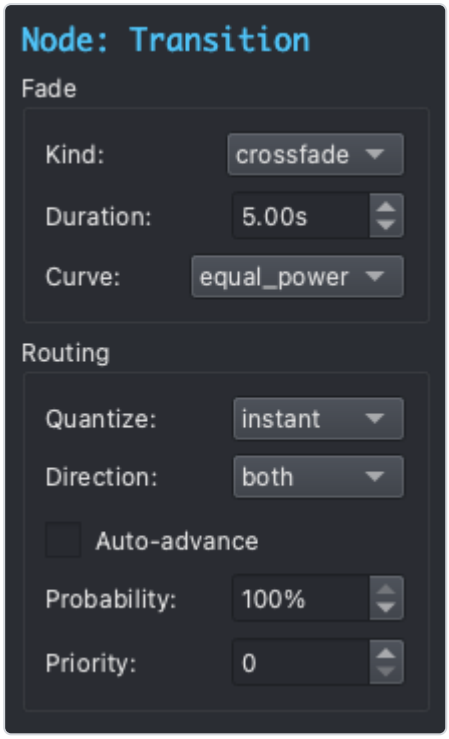
By default entry is **phase-synced** (ch. 3.3): arriving mid-bar continues the musical timeline. `Restart on entry` is the opt-out.

Connect sections by dragging from a node's right (output) port to another node's left (input) port — normally *through* a transition node.

6.4 Transition nodes

Transitions are nodes, not line properties: drop a `Crossfade`, `Fade In` or `Fade Out` between two sections and wire `source → transition → target`. The face shows everything at a glance — `5.0s`, a direction glyph

(`↔` both / `→` forward / `←` reverse), and the details line, e.g. `EP · Inst` (curve `EP/Lin/S/Exp/Log` · quantize `Bar/Beat/Inst` · `A` if auto-advance · `%` if probability < 100).



Fade group:

Property	Values
Kind	`crossfade` (both sides overlap) / `fade_in` (new side fades in, old cuts at the boundary) / `fade_out` (old side fades out; new side enters clean)
Duration	0–30 s
Curve	`equal_power`, `linear`, `s_curve`, `exponential`, `logarithmic`

Routing group:

Property	Meaning
Quantize	When the transition may start: next `bar`, next `beat`, or `instant`
Direction	`both` / `forward` / `reverse` — restricts which travel direction may use this transition
Auto-advance	Fire automatically when the source section finishes its repeats — this is how intros chain into loops
Probability	0–100% — chance this route is taken when eligible
Priority	0–99 — tie-breaker when several routes are eligible (higher wins)

> **Quantize is per transition, and preview obeys it.** A transition's own Quantize wins over the event-level **Switch** setting (Event Settings) — in the in-app preview and in the exported game alike; the config carries the value per connection and both engines (Web and Godot) resolve it the same way. Two rules override it on both sides: a section with **Restart on entry** always waits for the next bar (beat 1 at an

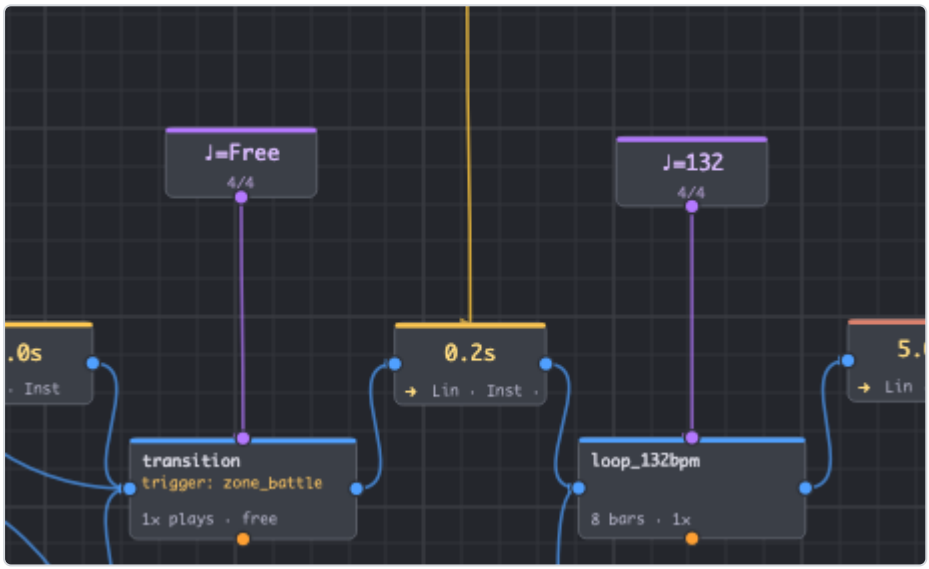
> arbitrary sample isn't a thing), and a **Free tempo** section has no grid to snap to, so its transitions land instantly — leaving a Free tempo section, there is no bar to wait for even into a Restart-on-entry target.

Two **dangling-fade** conventions round out the system:

- A `Fade In` node wired into the *first* section with no source = the event's **entry fade**. (Event Settings shows "⚠ *Controlled by the node graph (entry node)*" on Start fade in that case — the graph owns it.)
- A `Fade Out` node hanging off a section with no target = a **terminal fade**: when the section finishes its repeat count, it fades to silence and the event ends musically on its own.

6.5 Tempo nodes

A tempo node hangs *under* a section: drag from the tempo node's bottom port to the loop node's top port (the port turns purple). The face shows `♩=132` or `♩=Free`, plus the time signature.



Property	Values
BPM	1–999 (disabled when Free is on)
Free	Free tempo — no bar/beat grid for this section; loops follow the clip's trimmed window (a section's repeats are counted in clip cycles, on all three players); fades run on wall-clock time
Time Sig	`4/4 3/4 6/8 2/4 5/4 7/8` or `Custom…`

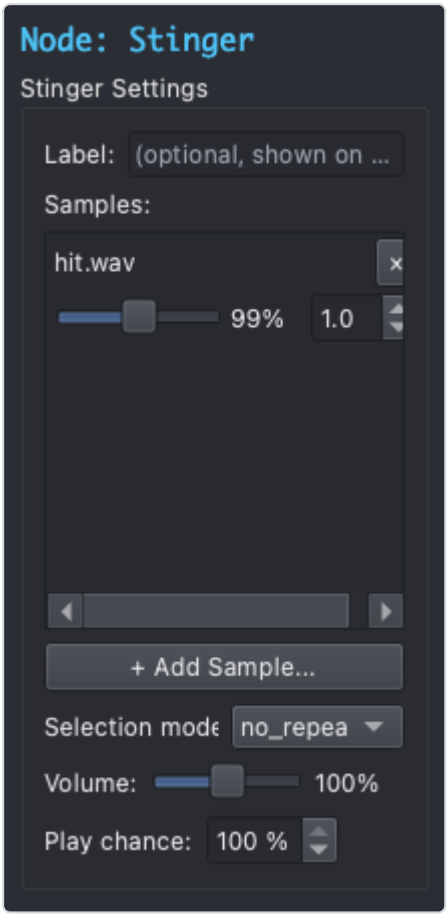
Free tempo is available **only** through a tempo node — the transport-bar **Free** box is a read-only indicator, editable here in the node graph.

Without a tempo node, a section uses the event's transport BPM / signature. With them, one event can travel `140 BPM → free-time riser → 132 BPM` — the demo does exactly that.

6.6 Stinger nodes

A stinger is a one-shot flourish (cymbal swell, whoosh, vocal hit) fired **at the moment its transition starts** — for a `Crossfade` or `Fade In`, that is t=0 of the fade; when a `Fade Out` chains into a `Fade In`, the stinger

fires when the fade-in begins (the outgoing fade finishes first). Drop a `+ Stinger`, fill it with samples, and wire its bottom port to a `Crossfade` or `Fade In` transition node. `Fade Out` nodes don't take stingers: the editor refuses the wire, and refuses switching a stinger-carrying transition's Kind to `Fade Out` — the reason appears right under the Kind field. One stinger can feed several transitions — author the flourish once, reuse it across the graph.



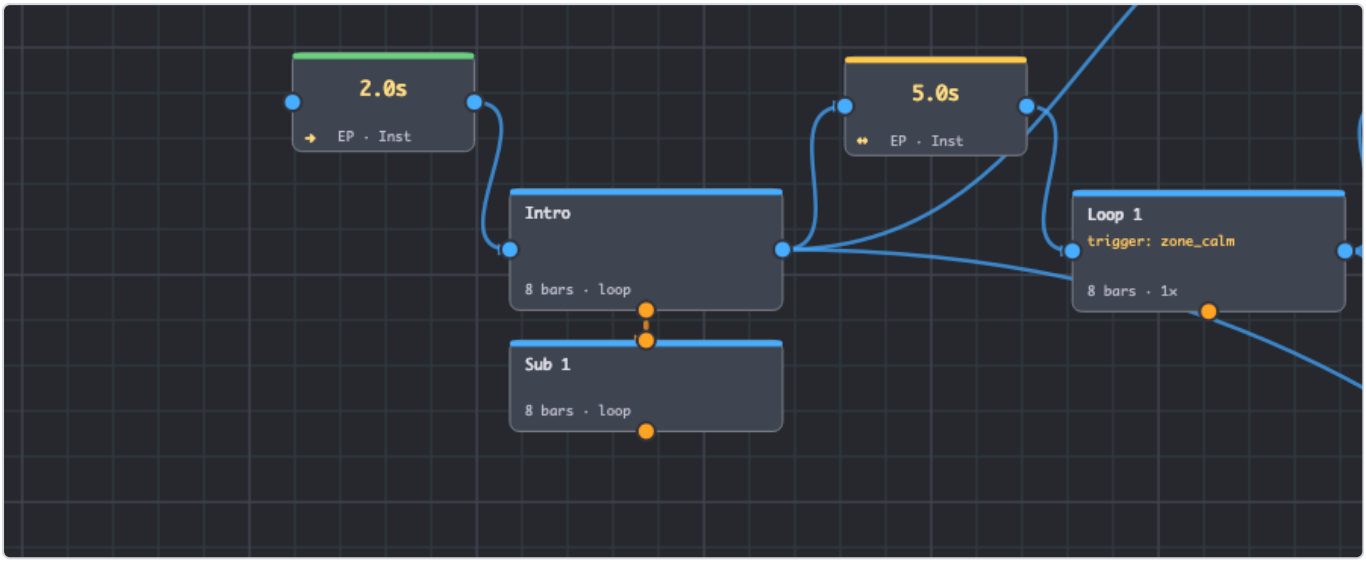
Property	Values
Label	Optional text shown on the node (otherwise `★ N samples`)
Samples	Any number of variants — per-sample volume and weight; drag-and-drop WAV/OGG works
Selection mode	`random` / `weighted` / `no_repeat`
Volume	0–200%
Play chance	0–100% per transition fire

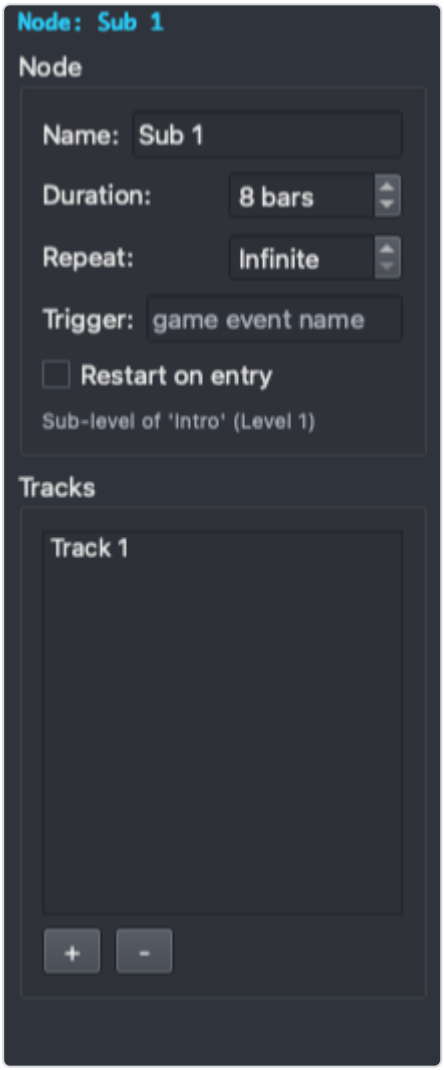
Double-click a stinger node to audition it (the node glows gold). Face badges: `NR` / `RND` / `WTD` for the mode, plus `%` when play chance < 100.

6.7 Sub nodes (layered sub-levels)

`+ Sub` (toolbar, or right-click a section → **Add Sub-level**) chains `Sub 1 → Sub 2 → …` *below* the parent section with vertical connections. Each sub is a layer variant of its parent (ch. 4.6): its tracks appear in the main timeline under the parent level, and the game toggles subs at runtime for intensity-within-the-section. In the game every section entry starts with its subs **off** — after a transition, toggle the sub on again if it

should still play; the editor's preview keeps a sub's switch as you authored it and restores it when you return to the section (ch. 9.1).





A sub's inspector mirrors a section's (duration, repeat, trigger, restart-on-entry), names the parent it hangs under, and lists the sub's own tracks — `+` / `–` add or remove them right from the graph.

6.8 Preview & how the game drives it

Press `Play` (or `Space`): the active section glows green and, during a transition, the transition node glows orange — you can watch the music walk the map. To rehearse the real thing, run your game on the exported pack — Live Connect (ch. 10) carries your mix moves editor → game while you play.

In the game, music graphs are steered with `navigate(event, trigger)`: give sections trigger names (`zone_calm`, `zone_battle`), then name the music event plus the destination — transitions are the roads between destinations, not destinations themselves. `trigger()` is the SFX call, not a music one: it starts an SFX event, which then follows its authored Loop (ch. 7.3) — a plain event plays once, a looping event plays until `stop()`:

```
PFA.navigate("music_forest", "zone_battle")  # route to the battle section
```

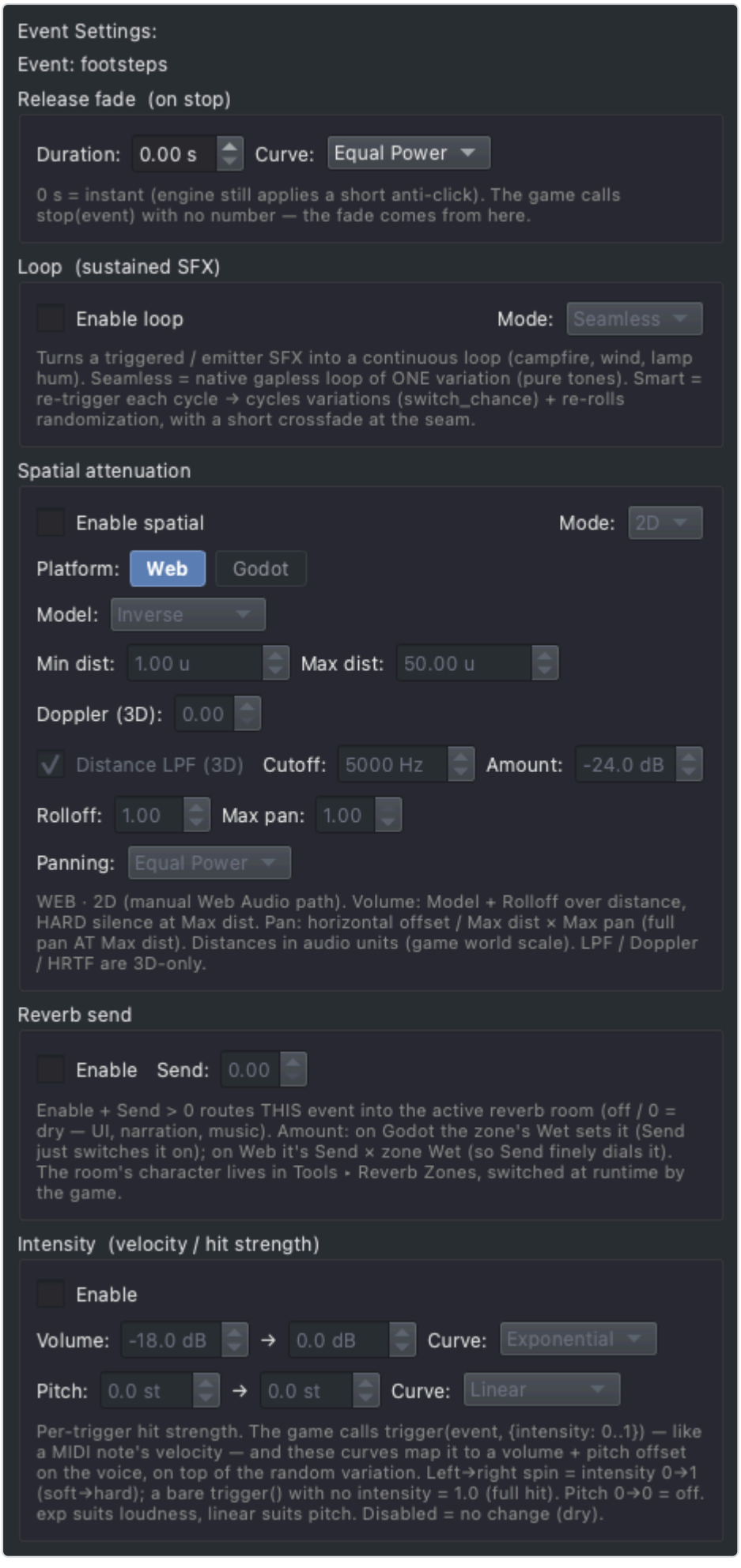
```
pfa.navigate('music_forest', 'zone_battle');
```

It returns `true` when the engine accepted the request. A route carrying that trigger out of the current section is ranked with its siblings by **Priority** and rolled on **Probability** — `true` means a roll passed. With no such route, a section that carries the trigger is entered directly and nothing is rolled: the fade settings are borrowed from the first road leaving the current section, whichever section that road leads to, or are the event's own when no road leaves it. `false` means nothing was scheduled. `true` says the switch is scheduled, not that it has happened — the `node_changed` signal / `onNodeChanged` callback marks the moment it applies. Quantize, curves, probability, priority, stingers — all resolved by the engine from your graph. The game names a destination; the graph decides how to get there musically.

Subs are not destinations: the game switches a sub-level on and off with `toggle_sub_level(event, sub_name, enabled)` / `toggleSubLevel(event, subName, enabled)` (ch. 6.7) — an exported sub carries its name, not a trigger, and the Trigger field is disabled on a Sub node.

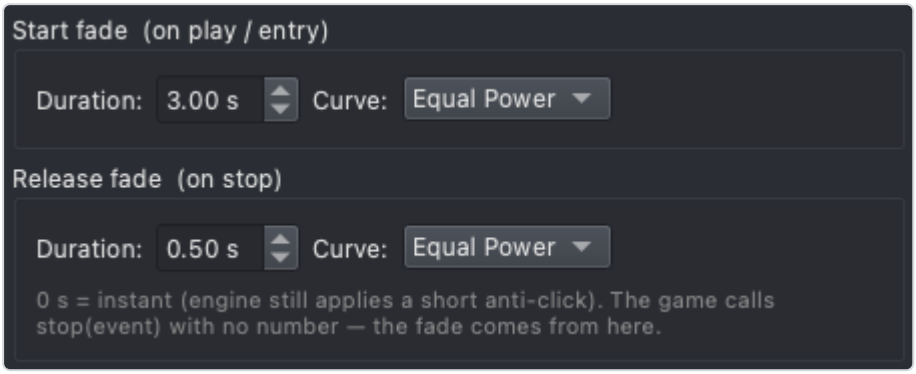
7. Event Settings

Select an event and the inspector shows **Event Settings** — the per-event behaviors the engines execute at runtime. Top to bottom: fades, loop mode, spatial attenuation, reverb send, intensity.


Event Settings:
Event: footsteps
Release fade (on stop)
Duration: 0.00 s
Curve: Equal Power
0 s = instant (engine still applies a short anti-click). The game calls stop(event) with no number — the fade comes from here.
Loop (sustained SFX)
Enable loop
Mode: Seamless
Turns a triggered / emitter SFX into a continuous loop (campfire, wind, lamp hum). Seamless = native gapless loop of ONE variation (pure tones). Smart = re-trigger each cycle → cycles variations (switch_chance) + re-rolls randomization, with a short crossfade at the seam.
Spatial attenuation
Enable spatial
Mode: 2D
Platform: Web
Godot
Model: Inverse
Min dist: 1.00 u
Max dist: 50.00 u
Doppler (3D): 0.00
Distance LPF (3D)
Cutoff: 5000 Hz
Amount: -24.0 dB
Rolloff: 1.00
Max pan: 1.00
Panning: Equal Power
WEB · 2D (manual Web Audio path). Volume: Model + Rolloff over distance, HARD silence at Max dist. Pan: horizontal offset / Max dist × Max pan (full pan AT Max dist). Distances in audio units (game world scale). LPF / Doppler / HRTF are 3D-only.
Reverb send
Enable
Send: 0.00
Enable + Send > 0 routes THIS event into the active reverb room (off / 0 = dry — UI, narration, music). Amount: on Godot the zone's Wet sets it (Send just switches it on); on Web it's Send × zone Wet (so Send finely dials it). The room's character lives in Tools ▸ Reverb Zones, switched at runtime by the game.
Intensity (velocity / hit strength)
Enable
Volume: -18.0 dB
→
0.0 dB
Curve: Exponential
Pitch: 0.0 st
→
0.0 st
Curve: Linear
Per-trigger hit strength. The game calls trigger(event, {intensity: 0..1}) — like a MIDI note's velocity — and these curves map it to a volume + pitch offset on the voice, on top of the random variation. Left→right spin = intensity 0→1 (soft→hard); a bare trigger() with no intensity = 1.0 (full hit). Pitch 0→0 = off. exp suits loudness, linear suits pitch. Disabled = no change (dry).

7.1 Start fade (on play / entry)

Fade-in applied when the event starts. **Duration** 0–10 s, **Curve** `Equal Power / Linear / S-Curve / Exponential / Logarithmic`.



If the event is an advanced-mode music event whose node graph has an entry fade node, the panel shows "⚠ *Controlled by the node graph (entry node)*" — author the intro fade in the Node Editor instead (ch. 6.4).

7.2 Release fade (on stop)

Fade-out applied when the game stops the event:

```
PFA.stop("music_forest")    # no duration argument — it comes from here
```

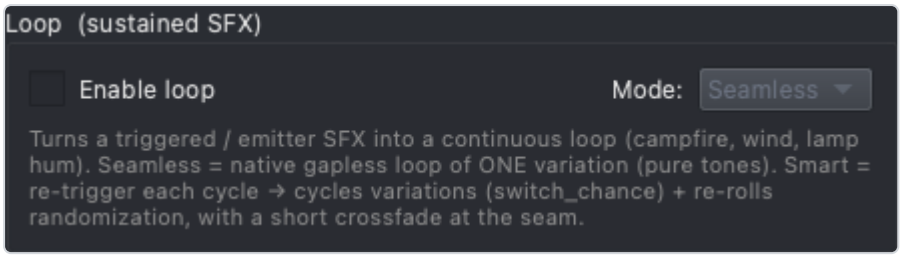
The game never passes a fade time; **the release fade is authored, not coded**. `0 s` means instant — the engine still applies a short anti-click ramp so hard stops don't pop. The release fade is also what the engine uses when the game retires a voice without naming a fade: destroying a positioned emitter, stopping a spatial loop. Polyphony eviction (ch. 4.1) is the exception — an evicted voice ends through a short fixed anti-click seam so the cap can act at once.

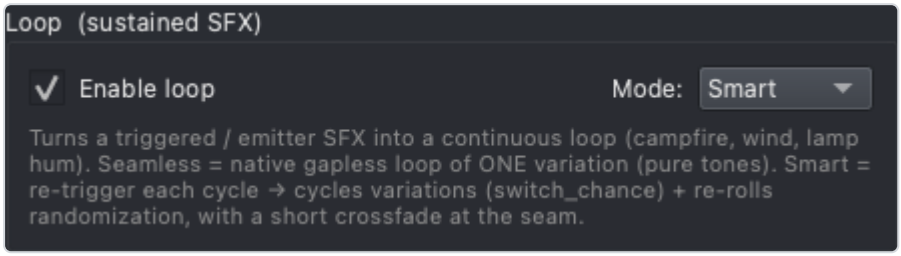
For sustained/looping SFX give this a real value (0.15–0.6 s) — a bare cut in the middle of a waveform is audible even with the anti-click floor.

7.3 Loop (sustained SFX)

Turns a triggered SFX into a continuous loop (campfire, wind, lamp hum, rolling ball). **Enable loop** + **Mode**:

Mode	Behavior	Use for
Seamless	Native gapless loop of ONE variation	Pure tones and steady textures (hums, drones, fire beds)
Smart	Re-triggers each cycle with a short crossfade at the seam — rotates variations (switch chance) and re-rolls randomization	Anything organic: wind gusts, crowd, chimes, crickets





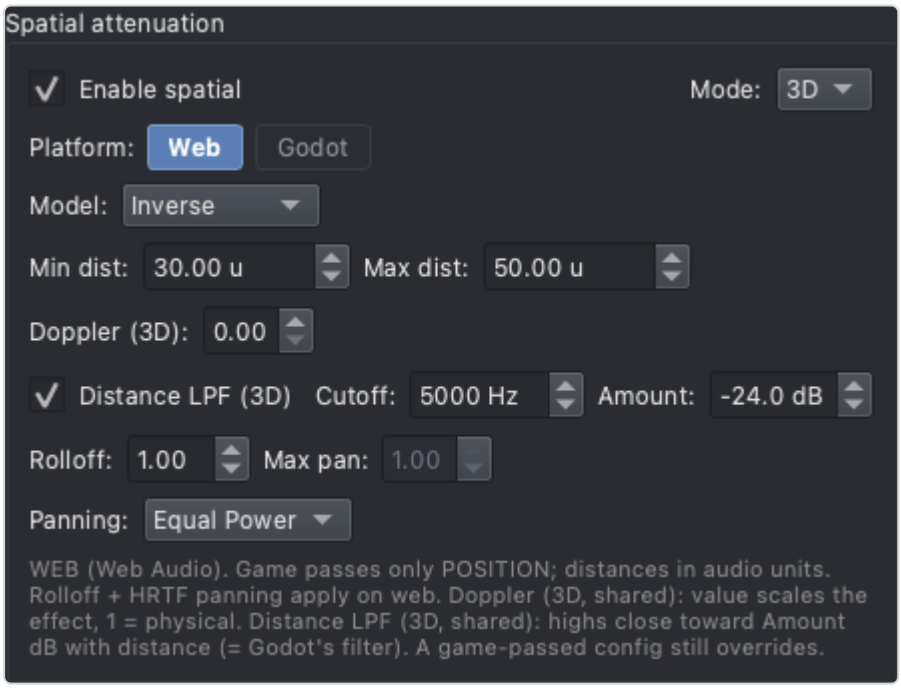
The 3D demo uses both on purpose: `fireplace` is seamless; `windchimes` and `cricket` are smart so their cycles never repeat literally.

The loop you trim is the loop that ships: on export a looping clip's window is baked into the audio file itself (ch. 11.2), so the seam you audition in the editor is the seam the game plays — the same window on both engines. The 80 ms crossfade at a Smart seam is shaped per player (equal-power in the editor, linear gain in the web engine, dB-linear in Godot): the same character, not a bit-identical curve.

The game stops a looping SFX with `stop(event)` (release fade applies), or by destroying its emitter for positioned loops.

7.4 Spatial attenuation

Positional audio, authored per event — the game only supplies positions.



- **Enable spatial** + **Mode** `2D` / `3D`.
- **2D** — for flat games: horizontal panning + distance attenuation in a plane. Pan = horizontal offset / max distance × **Max pan**.

- **3D** — full distance model with z; adds HRTF panning (web), Doppler and distance low-pass.
- **Model** — attenuation curve over distance (`Inverse` / `Linear` / `Exponential`; the Godot tab offers Godot's native equivalents).
- **Min dist / Max dist** — attenuation starts at min and reaches silence at max. Units are *your game's world units* (the engine takes a world-scale factor at init), so author distances in the scale you actually use. Godot 2D has no min distance — attenuation starts at the source.
- **Rolloff** *(web · Godot 2D)* — steepness multiplier for the model. Godot 3D has no rolloff exponent — shape the curve there with the native **Model** and **Min dist** instead.
- **Max pan** — 0–1, how hard the sound may pan at the edges (web 3D ignores it — the `PannerNode` always pans fully).
- **Panning** *(web)* — `Equal Power` or `HRTF (3D)` for binaural rendering.
- **Doppler (3D)** — 0–2, pitch shift from relative motion. The value scales the web effect; on Godot any value > 0 enables the engine's native physical doppler.
- **Distance LPF (3D)** — air absorption: **Cutoff** (100–20000 Hz, default 5000) and **Amount** (–60–0 dB, default –24) reached at max distance.

Web and Godot tabs. The Web values are the canonical character. Godot follows them automatically until you edit anything on the **Godot** tab — that authors a Godot-specific override (Godot's native attenuation sounds different, so matching by ear per platform is sometimes worth it). **↺ Follow Web** removes the override and re-links.



In the game, a positioned one-shot is a single call; a looping source is an emitter — create it, feed it positions, trigger it, and its voices follow. Everything else comes from this panel:

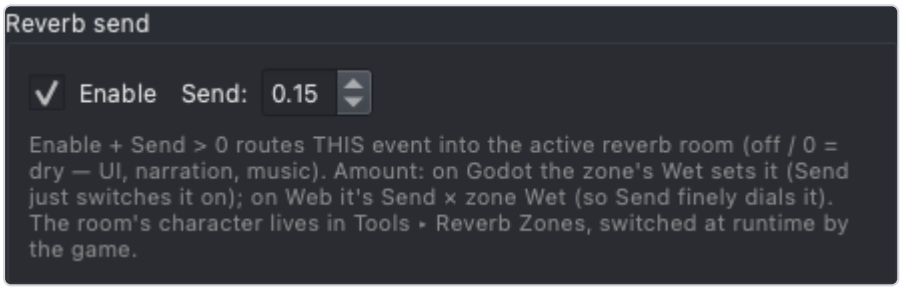
```
PFA.trigger_spatial("ball_hit", pos.x, pos.y)  # one-shot at a position
var em = PFA.create_emitter_2d("fireplace")    # looped emitter
em.set_position_2d(pos.x, pos.y)
em.trigger()
```

```
pfa.triggerSpatial('ball_hit', x, y);   // one-shot at a position
const em = pfa.createEmitter();         // looped emitter — no event yet…
em.setPosition2D(x, y);
em.trigger('fireplace');                // …the web dialect names it here
```

These are the 2D calls; the 3D twins are `trigger_spatial_3d`, `create_emitter_3d` and `set_position_3d` (web: pass a `z`, use `setPosition3D`). A moving object keeps updating its emitter's position and live voices follow. `em.destroy()` stops them with the authored release fade (ch. 7.2).

Note: the editor preview plays events dry/centered — spatial placement is a runtime behavior. Audition it in the demo games or over Live Connect.

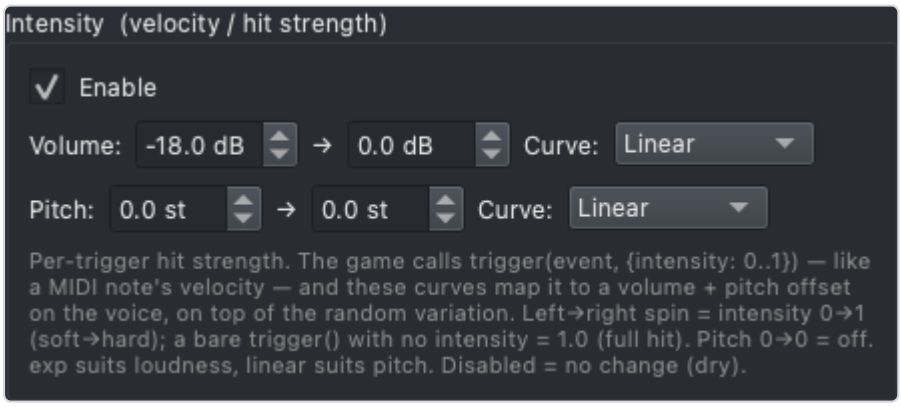
7.5 Reverb send



Enable + **Send** (0–1) routes this event into the active reverb zone. Off (or 0) = dry — the right choice for UI, narration and most music. The *character* of the room lives in the project's Reverb Zones library (ch. 8); the game switches zones at runtime.

Platform nuance (from the panel hint): on Web the heard amount is `Send × zone Wet` (fine per-event control); on Godot the zone's Wet sets the level and Send effectively switches the routing on.

7.6 Intensity (velocity / hit strength)



Intensity is per-trigger hit strength — a MIDI-velocity analog. The game passes `0..1` with the trigger; these curves map it onto the voice:

- **Volume** — dB at intensity 0 → dB at intensity 1 (–60…+24 dB), with a curve (`Linear / Exponential / Logarithmic / S-Curve`).
- **Pitch** — semitone offset range (–24…+24 st), with its own curve. `0 → 0` = pitch untouched.

```
PFA.trigger("ball_hit", {"intensity": impact_speed / max_speed})
```

```
pfa.trigger('ball_hit', { intensity: impactSpeed / maxSpeed });
```

A bare `trigger()` means intensity 1.0 (full hit). Exponential suits loudness, linear suits pitch. Intensity stacks *on top of* per-trigger randomization (ch. 5.4): randomization gives life, intensity gives meaning.

The Billiards demo drives all five events this way — cue strength, ball impacts, cushion hits — from `−18 dB → 0 dB` on an exponential curve: soft kisses whisper, breaks crack.

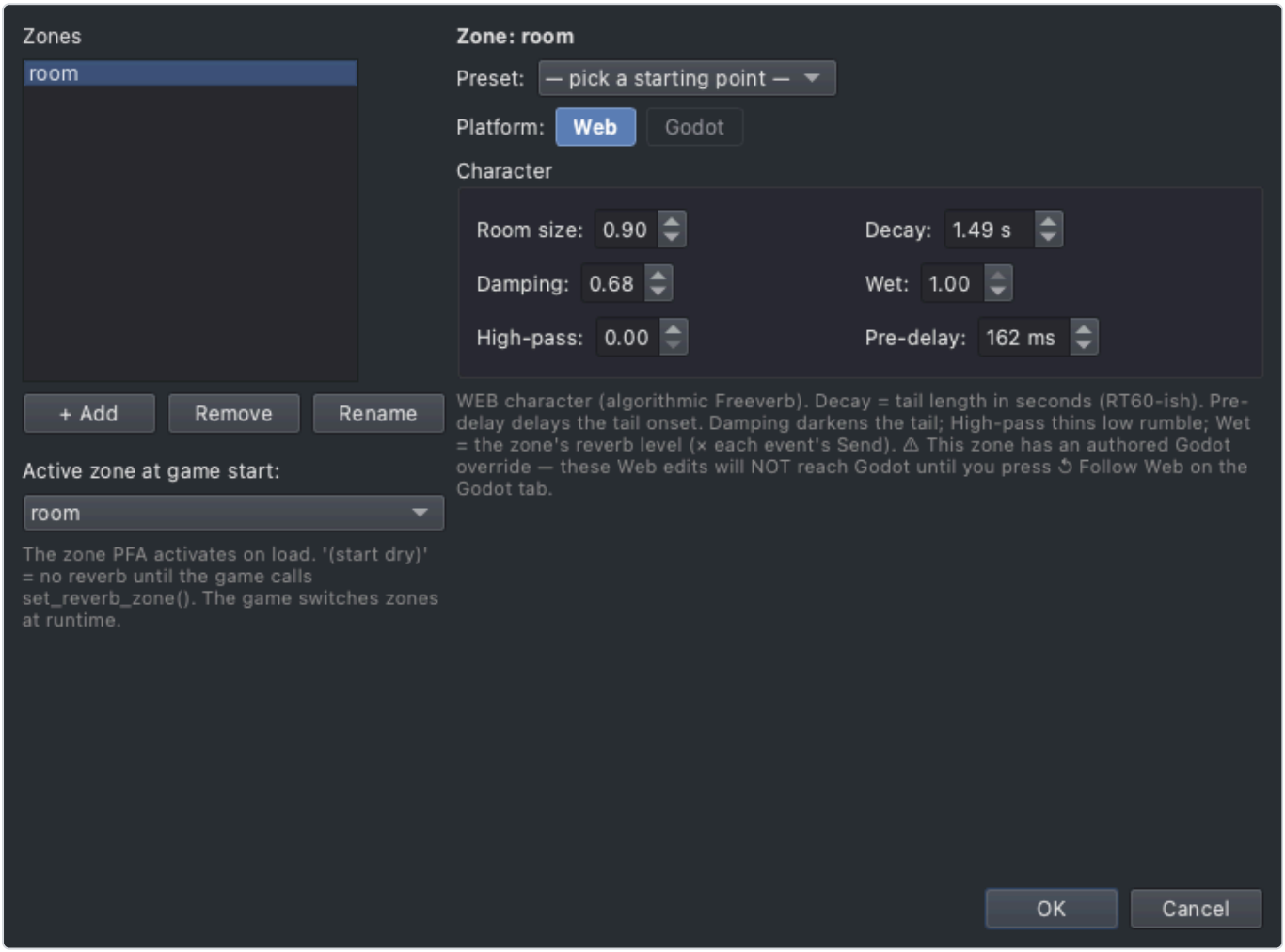
7.7 Cooldown & polyphony

Cooldown (0–5000 ms) and **Max voices** (1–32) are edited in the track panel for SFX events but apply to the whole event — see ch. 4.1.

8. Reverb Zones

Reverb in PFA is split in two: **events** say *how much* they feed the room (the send, ch. 7.5); the **project** owns a library of rooms — the reverb zones. The game switches the active zone as the player moves through spaces; every sending event immediately sits in the new room.

Open the library from ⚙ → **Tools** → **Reverb Zones…**



8.1 Zones

- **Zones** list — `+ Add`, `Remove`, `Rename`. A zone is a named room character: `room`, `cave`, `hangar`…
- **Active zone at game start** — which zone the engines activate on load, or `(start dry)` for no reverb until the game asks. Renaming/removing a zone keeps references tidy automatically.
- **Preset:** — 16 I3DL2/EAX-style starting points: *Dry (no reverb), Padded Cell (dead), Small Room, Bathroom (tiled), Large Stone Room, Concert Hall, Deep Cave, Arena / Stadium, Aircraft Hangar, Stone Corridor, Alley (urban), Open Forest, Open Plain / Outdoor, Parking Garage, Sewer Pipe, Underwater.* Pick one, then shape it with the knobs.

8.2 The character knobs

Knob	Range	What it does
Room size	0–1	Size of the simulated space (reflection density)
Decay	0.1–12 s	Tail length, RT60-ish *(Web only — see below)*
Damping	0–1	High-frequency absorption; higher = darker tail
Wet	0–1	The zone's reverb level (× each event's Send on Web)
Dry	0–1	Direct-signal level *(Godot tab)*
Spread	0–1	Stereo width of the tail *(Godot tab)*
High-pass	0–1	Thins low rumble out of the tail
Pre-delay	0–200 ms	Gap before the tail starts — clarity for transients *(Web only)*

8.3 Web and Godot tabs

Like spatial settings (ch. 7.4), a zone has one **canonical Web character** (algorithmic Freeverb) and an optional **Godot override** (Godot's native `AudioEffectReverb`). Godot mirrors the Web values until you edit any knob on the Godot tab — with one deliberate exception: picking any preset except `Dry` seeds a Godot override immediately. The engines run different send topologies (Web's reverb return is full-wet with per-event sends; Godot's zone wet *is* the bus mix), so the preset keeps the perceived wet as a Godot value while Web stores `wet = 1.0`. **↺ Follow Web** removes the override — the seeded compensation included — and re-links.

The two engines expose slightly different physics, and the dialog's hint text explains the differences as you switch tabs:

- Godot has **no Decay knob** — tail length rides on Room size (no RT60 parameter in the native effect).
- Godot has **no Pre-delay** — its native predelay is a feedback comb (tonal), not a true gap, so PFA doesn't use it.
- Godot adds native **Dry** and **Spread** knobs.

If you authored a Godot override, the Web tab warns you that Web edits no longer reach Godot — press ↺ Follow Web to re-link when you want one character again.

8.4 In the game

The exported config carries the whole library. Engines apply the start zone on load; switching is one call:

```
PFA.set_reverb_zone("cave")
```

```
pfa.setReverbZone('cave');
```

Only events with **Reverb send** enabled react — UI and music stay dry by default. Note that the editor preview doesn't run the reverb DSP; audition rooms in the demo games or over Live Connect (ch. 10).

One packaging note: reverb zones are a *project* library, so they intentionally do **not** travel inside `.pfapack` files (ch. 12) — pack READMEs tell the buyer to create a matching zone.

9. Preview & Mixing

9.1 Preview

`Space` plays the selected event through the same logic the engines use — loops, level crossfades, node-graph routing, variations, play-chance dice all behave as they will in the game, with the differences listed at the end of this section. Start and Release fades (ch. 7), SFX loop modes (ch. 7.3), a transition's own Quantize (ch. 6) and the intensity curve are auditioned here too, so what you tune is what ships. Switch level tabs or click sections in the Node Editor while playing to rehearse transitions. `Esc` stops (a long or looping SFX preview too) — with a Release fade authored, Stop rings the tail out first, as `stop()` will in-game. Double-click a stinger node to audition it (ch. 6.6).

What preview intentionally does *not* reproduce — these are runtime behaviors of the game engines; audition them in the demo games or over Live Connect (ch. 10):

- spatial positioning, distance rolloff, distance LPF, Doppler (events preview dry and centered);
- intensity *between* its endpoints: preview triggers at full strength (1.0), like a bare `trigger()` in game, so you hear the top of the authored curve — the game is what passes the values in between;
- reverb DSP (sends are authored here, heard in-game).

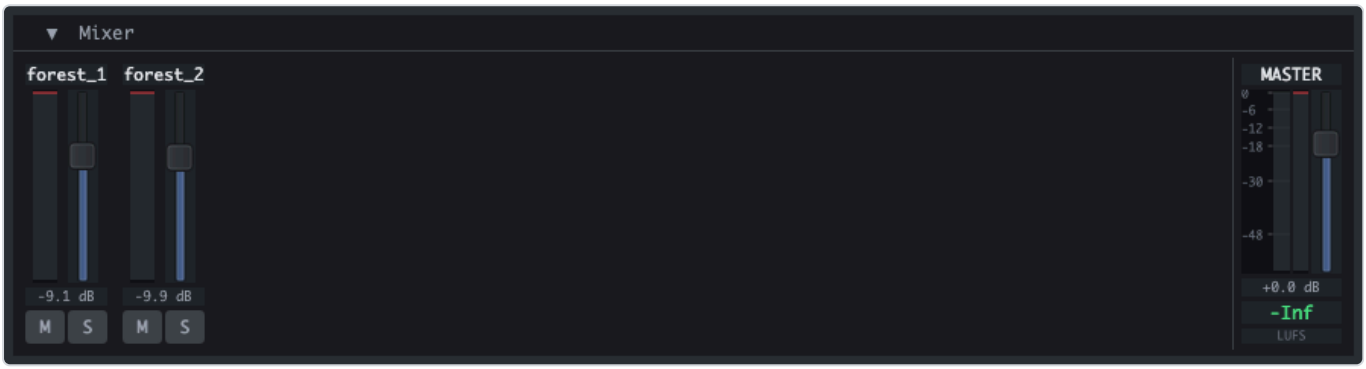
And a few places where the preview and the game engines differ by design or by construction — know them before you trust the preview for them:

- **a level's first visit**: the engines seed every level's variation and play-chance dice at `play()` and tick them all; the preview seeds a level the first time it is heard (ch. 3.3);
- **subs across a transition**: the preview keeps a sub's switch as authored and restores it when you return to its section; the engines start every section entry with subs off (ch. 6.7);
- **a click's route**: clicking a section in the Node Editor (or a level tab) takes the first route to it without rolling **Probability** or **Priority** — the roll is a trigger's: `navigate` in the game ranks the routes leaving the current section and rolls them (ch. 6.8);
- **per-player playback details**: the curve of a Smart-loop seam (ch. 7.3), the end of a one-shot's trimmed window in Godot (ch. 11.2), and a Smart event whose clip is set to `one_shot` — it plays once in the editor and on a flat web trigger, while emitters (web and Godot) re-arm it every cycle;
- **the level that already plays, requested while a switch waits** (a `set_level` call, ch. 3.3): the preview cancels the waiting switch and stays where it is; the engines ignore a request for the level they already play, so the waiting switch still lands. Through the node graph the rule is different: `navigate()` with the playing section's own trigger (ch. 6.8) enters the section again in the engines, and that entry replaces the waiting switch — the road it was waiting on is dropped, the section starts over if **Restart on entry** is set. The preview drops the waiting switch too, but does not re-enter (see the last bullet);
- **Stop pressed twice**: a second Stop during a Release fade (ch. 7.2) cuts the sound at once in the preview; in the engines a second `stop()` starts the Release again from where the fade stands, so the tail runs its full length once more;
- **a terminal fade interrupted**: switching away from a section while its terminal Fade Out (ch. 6.4) is running cancels the ending in the preview — the new section plays on; the engines let the ending run, so the event still stops when that fade would have ended;

- **a section entered from itself**: a road that leads a section back into itself, or its own trigger repeated, is a no-op in the preview — no restart, no stinger; the engines enter the section again, so **Restart on entry** starts it over and the road's stinger fires.

9.2 The two mixers

The Mixer panel — the collapsible strip at the bottom of the main window — is the *track* mixer of the selected event: one DAW-style channel strip per track (fader, meter, **M**/**S**) plus a **MASTER** strip with stereo metering and a LUFS loudness readout. It's where you balance layers inside an event.



The Event Mixer window (⚙ → Event Mixer) is the *project* mixer: one horizontal strip per event — type badge (`[M]` music / `[S]` SFX), name, ▶ preview, **M**ute, **S**olo, fader with dB readout and a mini peak meter — with collapsible group headers (▼/▶) carrying their own **M / S** and group fader. It's where you balance events and groups against each other.



Fader behavior (all faders in PFA):

- Range **–60 dB … +12 dB**, unity (0 dB) at 75% of the travel.
- **Double-click resets to 0 dB.**

- Readout shows `+3.0` / `−6.5` style values, `−inf` at the bottom.

Meter colors on the bottom strip mixer: green up to −18 dB, yellow −18…−6, orange −6…−3, red above −3 dB, with a peak-hold line. The Event Mixer's mini meters are simpler by design — green/yellow/red, no hold — enough to see a strip is alive. The master strip adds an integrated LUFS loudness readout (green below −20, yellow −20…−14, red above −14 LUFS).

9.3 Mixing decisions that ship

Everything on these strips — event volumes, group volumes — is authoring data: it exports into the config (baked into rendered audio where possible, carried as gains otherwise). Mute/solo are session tools for your ears and do not export as muted audio; over Live Connect (ch. 10) event solo/mute and group mutes duck the running game in real time — soloing an event is the fastest way to audition one layer inside the real mix. Group solo is local Studio monitoring (ch. 10 lists exactly what travels).

10. Live Connect

Mixing against your memory of the game is guesswork. Live Connect puts PFA Studio and a *running* game on a WebSocket so authoring changes land in the game in real time — fader moves, level switches, mutes/solos — while you play it.

10.1 Starting it

Click 🔴 **Live** in the transport. The button turns green (🟢 **Live**) and the status bar confirms `Live Connect started on port 9264`. When a game connects, the button shows the client count — 🟢 **Live (1)**.

The PFA engines connect automatically in development builds when Live Connect is enabled on their side (`ws://localhost:9264`); see the engine integration docs of your platform. Both demo games ship with it wired.

If the port is busy the status bar says so — usually a second Studio instance already running.

10.2 What flows over the wire

While connected, the editor broadcasts your **mix moves**:

- **volume** — event faders, group faders and the master, from the *Event Mixer* window (ch. 9.2). The bottom strip mixer rides tracks *inside* the selected event; track levels are baked at export, not sent live;
- **mute** — per event; a group mute is fanned out over its members;
- **solo** — solo ducks everything else in the running game, the fastest "what is this layer doing" tool there is. Games have no solo concept of their own, so Studio translates it into ducks; group solo stays in Studio;
- **level switches** — click level tabs in Studio, hear the game change.

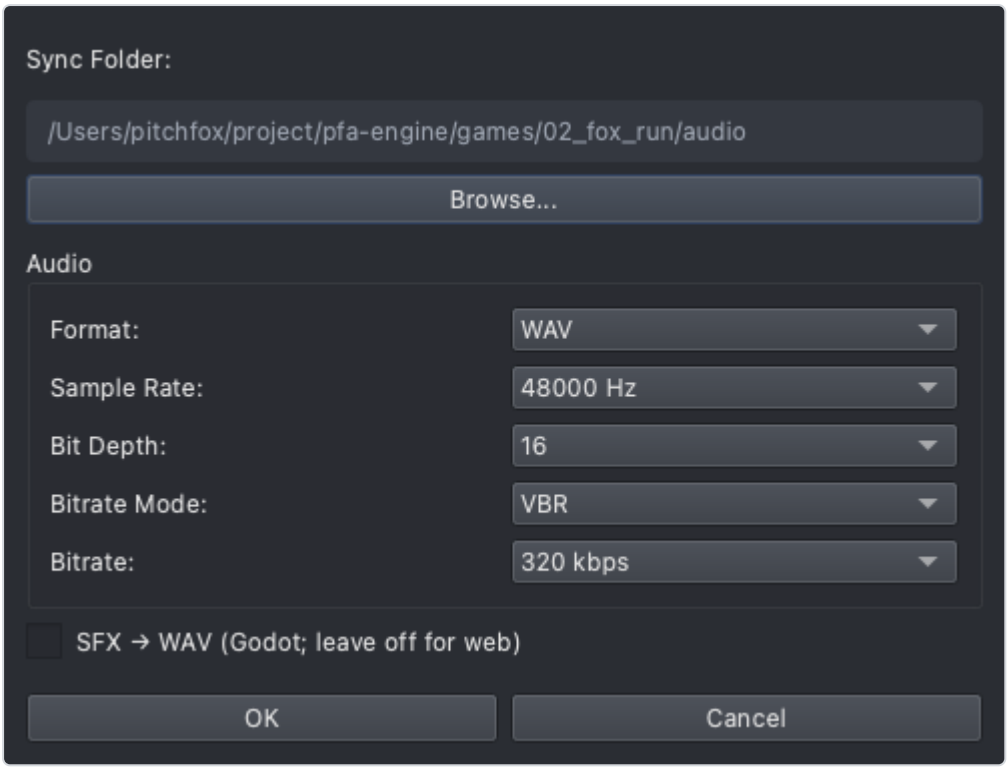
Direction is editor → game: you play the game normally and sculpt from Studio.

Structural edits don't travel live. New events, renames, moves, added tracks — the running game learns about them on **Sync** (10.3), which re-exports `config.json` and tells the client to reload the whole pack. Live messages address an event by its exported key, so a rename mid-session parks that event's faders until you sync.

10.3 Sync — one-click export

Live Connect tweaks gains in memory; **Sync** makes the state real on disk. The **Sync** button runs an incremental export into your configured game folder and tells connected clients to reload: `Syncing…` → ✅ `Synced!`.

Configure the target once in ⚙ → **Export** → **Sync Folder…**:



- **Sync Folder** — usually your game project's audio folder (e.g. `game/audio/`).
- **Audio** — format / sample rate / bit depth / bitrate mode + bitrate, and the same **SFX → WAV** switch as the export dialog (ch. 11.3). These persist in the project.
- The dialog shows the last sync time.

Sync is incremental: a content hash per sample (source bytes + settings + volume) means unchanged files are skipped — re-syncs after a small tweak take a moment, not a re-encode of the whole project. The status line reports what was newly encoded, skipped and removed.

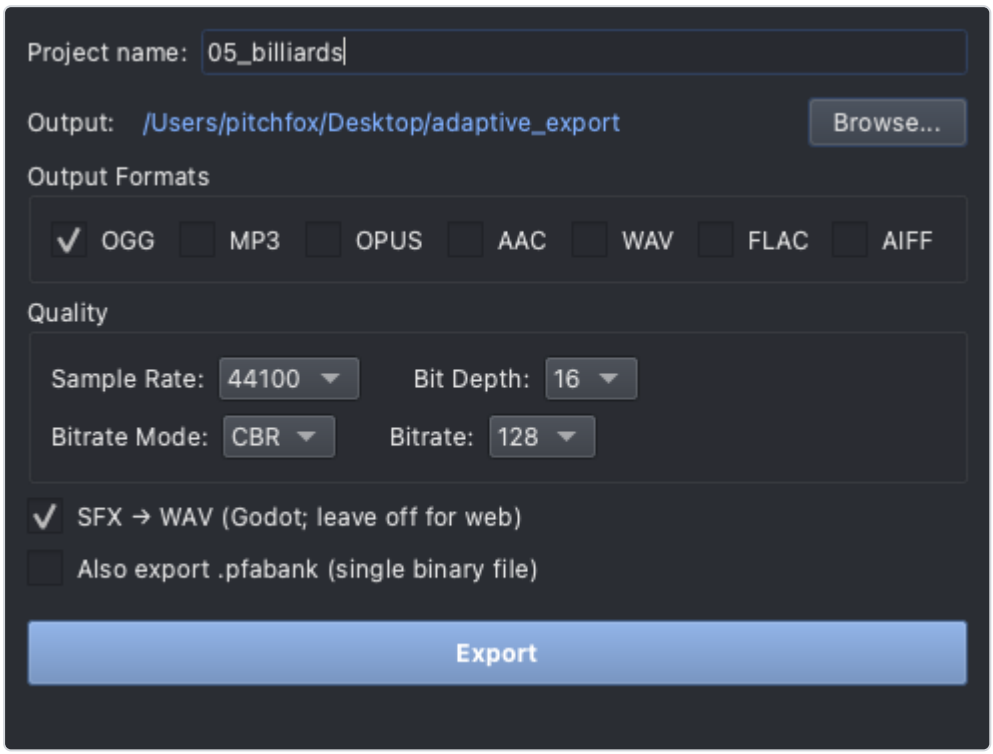
⚠ **Author against the loose `config.json`, not a `.pfabank`.** Mix moves reach a bank-loaded game fine, but Sync only ever writes loose files (ch. 11.4) and the reload that follows re-reads `config.json` from wherever the pack was loaded. A game booted from a bank has no such path: the Godot client logs *nothing to reload* and keeps playing, while the web client stops the audio and then fails to fetch. Load the folder in dev builds and ship the bank in release — or point the client's pack path at the loose config before syncing.

10.4 A working rhythm

1. Start the game (dev build), press 🟢 **Live**.
2. Play; solo events, ride faders, switch levels until it feels right.
3. Press **Sync** — the mix is now on disk and the game reloaded it.
4. Commit your game repo; the audio folder is part of the game.

11. Export for Game

Export renders the project into what ships: encoded audio files plus a `config.json` the PFA engine loads. ⚙ → **Export** → **Export for Game…**



11.1 The dialog

- **Project name** — used inside the config metadata.
- **Output** — target folder (`Browse…`).
- **Output Formats** — check any of `OGG` (default) / `MP3` / `OPUS` / `AAC` / `WAV` / `FLAC` / `AIFF`. Each checked format renders a complete set; OGG is the safe cross-platform default for both engines. The config (and a `.pfabank`) points at the **first** checked format — and the Godot addon decodes only OGG / MP3 / WAV (Godot 4 has no runtime decoders for the other four), so for a Godot game keep one of those three first in the list. The web engine plays whatever the browser decodes.
- **Quality** — Sample Rate `22050 / 44100 / 48000`; Bit Depth `16 / 24 / 32` (for PCM formats); Bitrate Mode `CBR / VBR / ABR` with bitrates `64–320 kbps` (for lossy formats). Export quality is independent of the project preview rate (ch. 1.5). OPUS always encodes at 48 kHz — the codec accepts no other rate family; decoders resample transparently, so this is inaudible plumbing, not a quality change.
- **SFX → WAV (Godot; leave off for web)** — see 11.3.
- **Also export .pfabank (single binary file)** — see 11.4.
- **Export** — runs with a `[current/total]` progress readout and a summary when done.

Encoding is done by the ffmpeg bundled inside the app — no system installs needed. (Running from source, the editor uses the repo's ffmpeg so dev and app exports stay bit-identical.)

11.2 What gets baked

Export is not a file copy — it renders your authoring into runtime-ready assets:

- **Volume baking.** Sample volume × clip volume is baked into the rendered audio (one ffmpeg volume pass), and the exported clip volume is reset to 1.0 — engines don't spend a runtime multiply, and quiet layers can't be un-mixed by accident. Stinger samples are the exception: a stinger clip's volume ships in `config.json` and the engines apply it at fire time.
- **Trim baking.** Clips play as their trimmed windows on both engines. For a **looping SFX** (ch. 7.3) the window is baked into the rendered file itself — the shipped asset *is* the authored loop (Godot native-loops only whole files), and the exported clip is normalized to match. **One-shot** windows ship in `config.json` and are applied at trigger time — sample-accurate on the web engine (the window is the buffer's play length); in Godot the start is sample-accurate and the window's end lands on a frame timer (a frame late at most, released click-free). Their files stay full, so re-exports stay fast.
- **Loop math.** Each music track's `loop_bars` is computed from the clip's trimmed length at the event's own BPM — the loop boundary in the game is exactly the one you previewed (ch. 4.5).
- **Node graphs flatten.** Tempo, transition and stinger nodes bake into an engine-ready structure (ch. 14 for the schema).
- **Master volume.** The master fader (ch. 9.2) ships in `config.json` (written only when it is not at unity) — both engines apply it on load and re-apply it on a live reload, so the game plays exactly as loud as the mix you approved. No need to fake headroom on event faders.
- `config.json` carries everything else: events, levels, fades, spatial, reverb zones, intensity curves, groups.

11.3 SFX → WAV

A Godot-specific optimization, off by default:

> Export one-shot SFX as uncompressed WAV — punchy transients and no Godot native-WAV-loop click. Looping SFX and music keep the formats above. Leave OFF for web builds (WAV is far larger).

Background: Godot decodes WAV with the lowest latency (good for one-shots), but its native WAV *looping* has a one-frame overread bug (godotengine

119778) that clicks at the seam — so PFA routes only one-shot SFX to

WAV and keeps loops in the compressed format, which loops cleanly. For web builds leave it off; download size wins.

11.4 .pfabank

`Also export .pfabank` additionally packs config + all audio of the base format into **one binary file** with an indexed data section. For web games that means one HTTP request instead of dozens. Both engines hold

the bank in memory and slice entries out of it on demand; once what you need is preloaded, the raw bank buffer can be released. Format details in the appendix (ch. 14.3).

A bank is a shipping format: Sync never rewrites one, so a dev build you iterate on with Live Connect should load the loose folder instead (ch. 10.3).

⚠ **Godot: a bank needs the addon enabled (or one export setting).** Godot's exporter ships resources plus whatever the export preset's *Filters to export non-resource files/folders* names — a `.pfabank` is neither, so a bank sitting in `res://` reaches the editor and **not** the built game. The PFA addon from **0.11.1** adds banks to the export by itself; with an older addon, or with the plugin disabled, add `*.pfabank` to that filter. Loose packs need nothing: `config.json` ships as a resource and Godot imports OGG / MP3 / WAV. Formats Godot has no importer for (OPUS, AAC, FLAC, AIFF) do not reach the build either — one more reason ch. 11.1 puts OGG first for Godot targets.

11.5 Export vs Sync

Same renderer, two intents:

	Export for Game…	**Sync**
Target	Any folder (a release build, a handoff zip)	The configured game folder
Formats	One or several, picked in the dialog	The project's export settings
When	Milestones, releases	Constantly, while iterating (pairs with Live Connect)

Both run the same hash-incremental renderer: unchanged samples are skipped via a content-hash cache stored in the destination folder, so re-exporting to the same folder converts only what changed — and a fresh, empty folder is always a full render.

Both write the same `config.json` + audio layout the engines load:

```
PFA.load_pack("res://audio/config.json")
```

```
await pfa.loadPack('audio/config.json');   // or loadBank('audio.pfabank')
```

Both loaders answer `true` / `false` and never raise: input that isn't a pack this engine can read — JSON that is not a config, a damaged bank, a config schema newer than the engine — is refused with one error in the log, and whatever was loaded before keeps playing.

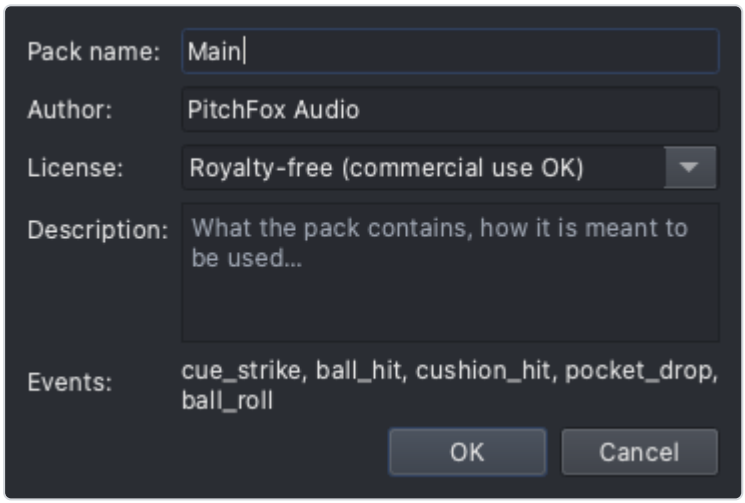
12. Packs (.pfapack)

A **pack** is a portable library of PFA events: one file containing event definitions *in full editable form*, the raw source samples, and a generated README with integration snippets. Packs are how you ship a footsteps library, hand a colleague your UI kit, or sell curated event sets — the buyer imports the pack, tweaks anything, and bakes it with **their** export.

12.1 Exporting a pack

Right-click in the event tree:

- an **event** → **Export as Pack…** (Tier 1 — single event), or
- a **group** → **Export Group as Pack…** (Tier 2 — the natural unit: a whole footsteps set, a UI kit).



The metadata dialog:

Field	Notes
Pack name	Pre-filled from the event/group name
Author	Persists across projects — type it once
License	Pick or type: `CC0`, `CC-BY 4.0`, `Royalty-free (commercial use OK)`, `All rights reserved`, or your own
Description	"*What the pack contains, how it is meant to be used…*"
Events	Read-only summary of what's going in

Save as `<name>.pfapack`. The status bar reports events and samples packed.

What goes inside:

- events in **project form** — clips, fades, variations, spatial, intensity, node graphs, stingers, sub-levels — fully editable after import;
- **raw source samples** (WAV as you authored them), deduplicated;
- an auto-generated **README.md** with the pack's trigger names and copy-paste GDScript + JS snippets: `trigger()`, `set_level()` with the pack's level table, intensity and spatial variants where the events use

them.

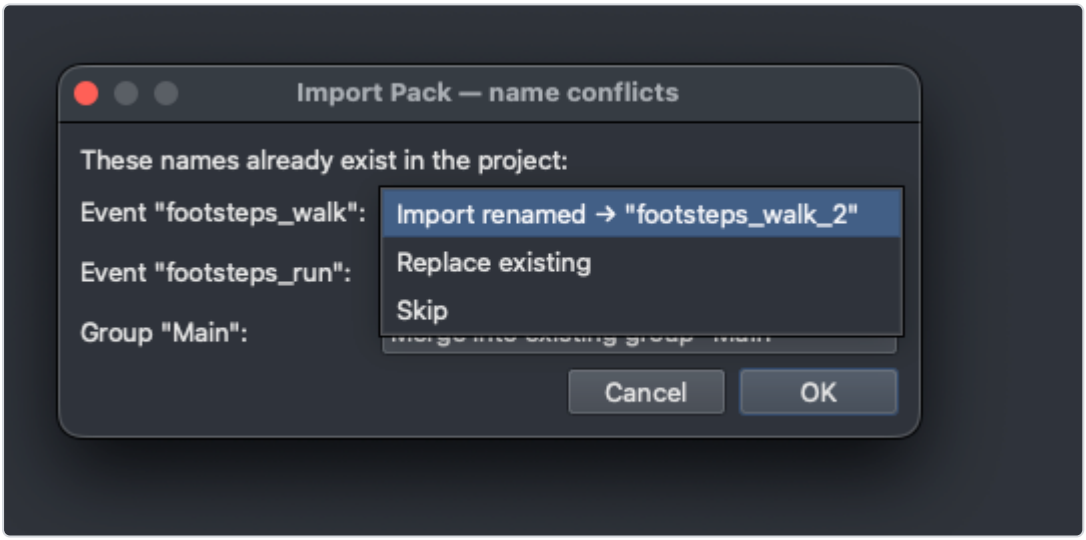
Missing sample files abort the export with a clear error — a pack is always complete.

12.2 Importing a pack

⚙ → **Export** → **Import Pack…** (save your project first — the importer copies the pack's samples into `<project>/samples/`; it will remind you: "*Save the project first — the pack's samples are copied into the project's samples folder.*").

The pack is **fully validated before anything is extracted** — a broken archive changes nothing in your project. Then:

- Samples land flat in `<project>/samples/`. A file that already exists byte-identically is reused (re-importing a pack is idempotent); a different file with the same name gets a `_2` suffix.
- If names collide with existing events, the conflict dialog offers, per event: **Import renamed** → **"name_2"** (default) / **Replace existing** / **Skip** — and for the pack's group: **Merge into existing group** / **Create renamed group**. Names are compared the same way exported file names are — case and spacing don't matter (`"Loop"` and `"loop"` count as the same name), so an import can never plant a hidden clash that would only surface at export time.



After import the events are ordinary events in your tree — reshape, remix, re-export as your own game audio.

12.3 Two design notes

- **Reverb zones don't travel in packs** — they're a project library (ch. 8), and your project's rooms should win. When a pack's events use a send, the pack README says so and points the buyer at ⚙ → **Tools** → **Reverb Zones…**; with no zones in the project those events simply play dry.
- **Packs carry authoring, not bakes.** Sample volumes, trims and loop math bake at the *importing* project's export, with its formats and quality. That's why packs stay editable end to end.

13. Tools & Settings

Everything in this chapter lives behind the ⚙ button (ch. 1.3).

13.1 Collect Samples

⚙ → **Tools** → **Collect Samples** copies every sample the project references into `<project>/samples/` and re-points the project at the copies (relative paths). Result: the project folder is self-contained — safe to move, zip, back up or sync between machines. Requires a saved project; reports `Collected N samples to: …` when done (plus any files it couldn't copy).

Run it whenever you've been auditioning files straight from a downloads folder or a sample-library drive.

13.2 Appearance — themes

⚙ → **Appearance** selects the UI theme preset:

- **Cool Gray (default)** — neutral dark gray with a cold touch
- **Graphite** — flat, fully neutral
- **Indigo (classic)** — the original PFA look
- **Warm Graphite**
- **Lifted** — lighter surfaces

Themes apply on restart (the app offers to restart right away). Semantic colors — meters, Solo gold, record red, Music/SFX tints — stay identical across presets.

13.3 Event Mixer & Restart

- **Event Mixer** opens the standalone mixer window (ch. 9.2).
- **Restart App** — confirm → clean stop → relaunch. With unsaved changes you get the standard **Save / Discard / Cancel** prompt first.

13.4 AI Automation Bridge (MCP)

⚙ → **Automation** → **AI Automation Bridge (MCP)** lets an AI copilot — Claude Code, or any MCP-capable client — drive the editor for you: create and author events, wire node graphs, import samples, mix, tune spatial settings, export, sync a running game and take screenshots of the UI. Authoring happens in the visible window — the same widgets move as if you were clicking, so you watch every step (and can record it); file operations (open/save, export, sync, packs) run dialog-free and report structured results.

The checkbox starts and stops the bridge immediately and is remembered for future launches; the status line underneath shows `Running on 127.0.0.1:<port>` while it's on. It is **off by default**.

Hooking up Claude Code

Register the bridge once per machine. With the app installed on macOS:

```
claude mcp add pfa-studio --scope user -- python3 \
  "/Applications/PFA Studio.app/Contents/Resources/automation/mcp_server.py"
```

On Windows, point it at `C:\Program Files\PFA Studio\_internal\automation\mcp_server.py` (and use `python` if `python3` isn't on your PATH). Running from source, use `<repo>/automation/mcp_server.py`. The helper is a small standalone script — any Python 3.9+ works; it doesn't use the app's bundled runtime.

Then, in a **new** Claude Code session, just ask: "*open my project and play the boss music*", "*build a three-level combat event from these WAVs*", "*export the project and sync the game*" — the tools are discovered automatically.

- MCP clients cache the tool list when a session starts, so enable the bridge (or update Studio) **before** starting the session you want to drive it from.
- Studio itself can start or restart at any time — the connection details are re-read on every call.
- If **two Studios run at once**, the one that started last owns the bridge: it takes over the connection details, and the copilot's next call goes to it without announcing the change. Keep a single instance open while a copilot is driving, and ask it for editor status if in doubt.

Any other MCP-capable client registers the same way — e.g. OpenAI Codex: `codex mcp add pfa-studio -- python3 "<same path as above>"`.

What the copilot can do

54 tools, mirroring what you could do with the mouse:

Area	Tools
Observe	editor status, event list & full event details (levels, clips, per-track settings), node-graph read, reverb-zone list, screenshots (window / node editor / mixer)
Projects	new, open, save, save as, project sample rate
Authoring	groups, events (music / advanced / SFX) incl. delete and move-to-group, tracks incl. per-level vs all-levels behaviour and music repeat count, sample import (WAV in place; MP3/OGG/FLAC/AIFF/M4A… auto-convert to an in-project WAV copy), levels, clip trim/fades/volume, event BPM & time signature, variation behaviour (play/switch chance, selection modes, chance fade), SFX randomization ranges (pitch / rate / delay / volume), SFX cooldown & voice limits
Node graph	sections, transitions, tempo nodes, stingers (incl. per-sample volume/weight), sub-levels & sub tracks, wiring
Mix & settings	event-mixer moves, master fader, track volume / M / S, spatial (Web & Godot tabs), reverb enable & send, reverb zones (create / edit / presets / start zone), intensity response curves, event fades, SFX loop modes
Delivery	preview play/stop, Export for Game (formats, rates, bitrate mode), Live-Connect sync, pack export & import
Recording	pacing control — slows the bridge's moves down so screen recordings are watchable

Ground rules

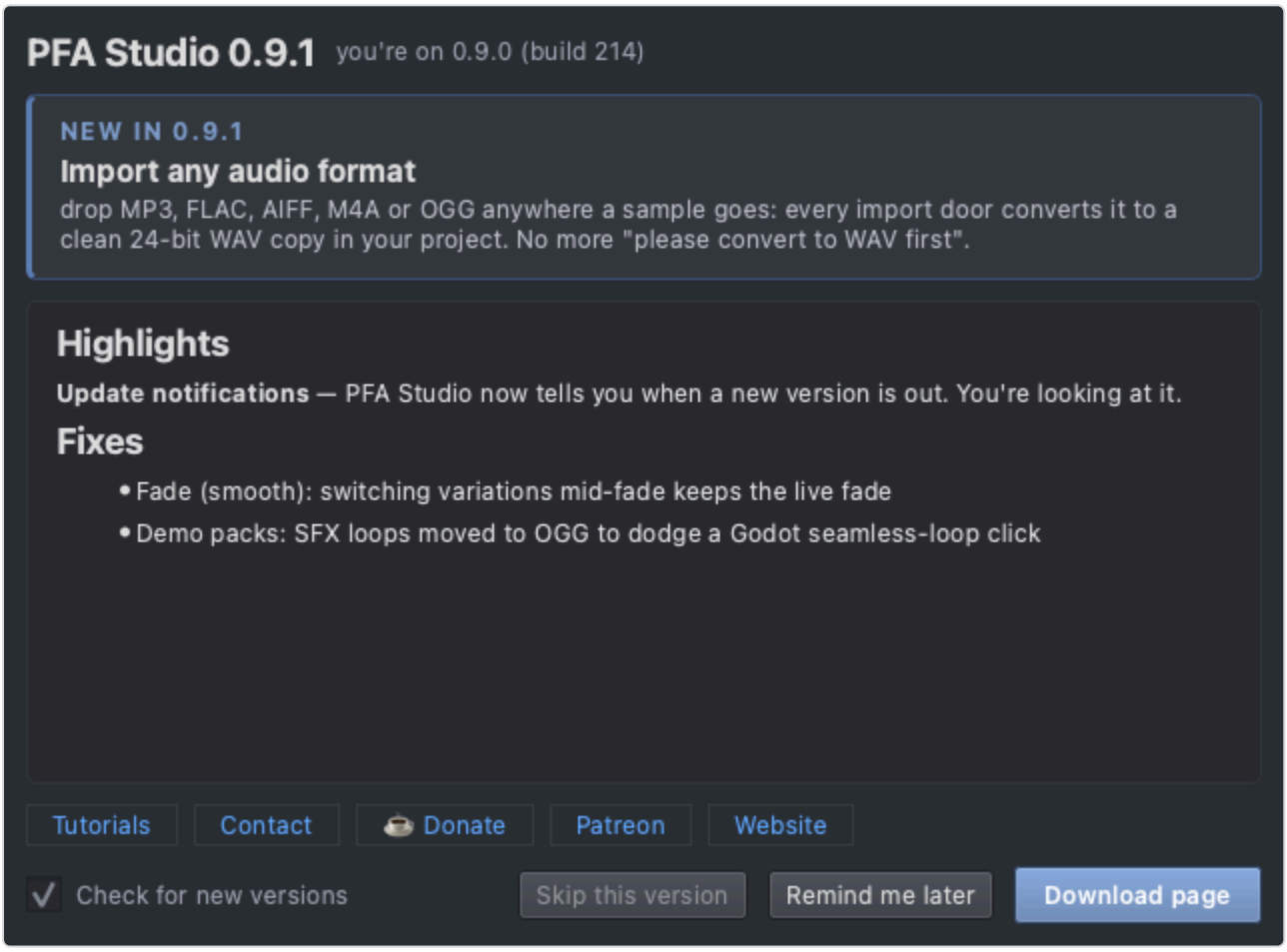
- The bridge binds to **127.0.0.1 only** and requires a random per-launch token, which clients pick up from `~/.pfa_studio_automation.json` (created with owner-only permissions while the bridge is on). Nothing

is reachable from the network.

- Copilot edits are **real edits** — the same as your own clicks, and the editor has no undo. Unsaved-changes gates stay in force (tools refuse rather than pop dialogs), but save deliberately.
- The bridge never opens modal dialogs: anything a tool can't do comes back to the copilot as a structured error.

13.5 Update notifications

Once a day, on launch, PFA Studio asks GitHub Releases whether a newer version is out. When one is, a **New version available** panel shows the release notes — the top highlight as a banner, the full changelog under it — with a **Download page** button. Nothing installs itself: you download and drag to Applications as usual.



Closing is your choice:

- **Remind me later** — the panel returns on the next launch.
- **Skip this version** — silence until the release after that one.

Either way a quiet `↑ x.y.z available` link stays in the status bar; click it to reopen the panel any time.

The check is a single anonymous request (no account, no identifiers, one attempt per day) and fails silently with no network — the app never waits on it. Turn it off with **Check for new versions** — in the panel itself or under ⚙ → **Info**.

13.6 Files PFA writes

What	Where
Projects	Wherever you save them — `<name>/<name>.pfa` + `samples/`
Crash log (app)	`~/Library/Logs/PFA Studio/crash_log.txt`
Crash log (running from source)	`crash_log.txt` in the repo
Editor preferences (last-used file-dialog folders)	`~/.pfa_editor_prefs.json`
App settings (theme, pack author/license, EULA acceptance)	`~/.pfa_studio/settings.ini`
Editor settings (incl. window geometry) & recent projects	`~/.pfa_studio/settings.json`, `recent_projects.json`
Automation handshake (port + token, while the bridge is on)	`~/.pfa_studio_automation.json`
Per-project sync settings	Inside the `.pfa` file itself

13.7 Troubleshooting quick list

- **App won't open ("PFA Studio" Not Opened)** — click **Done**, then **System Settings** → **Privacy & Security** → **Open Anyway**, and **Open Anyway** once more in the dialog (1.2); needed once per build.
- **No sound in preview** — check the **Out:** device combo in the transport; system default may have changed.
- **Export says FFmpeg missing** *(source runs only — the app bundles it)* — the repo copy resolves first; otherwise install via `brew install ffmpeg`.
- **Game doesn't hear Sync** — confirm the Sync Folder path (ch. 10.3) and that the game loads that folder's `config.json`.
- **Claude can't reach PFA Studio** — turn on the bridge (13.4) and check the status line shows a port; then retry from the copilot side.
- **Claude is missing recently added tools** — start a new Claude session; the tool list is cached when a session begins.
- **Something crashed** — grab the crash log (13.6) and file an issue with it; the log contains the full traceback.

14. Appendix

14.1 config.json field reference

What Export writes and the engines read. Optional blocks are simply absent when disabled — engines assume the safe default.

Top level

Key	Type	Meaning
`version`	string	Config schema version
`project`	string	Project display name
`audio_format`	string	Primary codec hint — the format picked first in the export dialog: `ogg`, `mp3`, `opus`, `aac`, `wav`, `flac`, `aiff`
`bpm`	number	Informational — the project tempo at export. Each music event carries its own `bpm`; an event without one defaults to 120 in both engines
`sample_rate`	int	Informational; engines read real rates from audio headers
`groups`	array	`{name, volume, events[]}` — volume is linear (1.0 = 0 dB)
`events`	object	Event definitions, keyed by safe name (lowercase; spaces and unsafe filename characters become `_`)
`reverb_zones`	object	*(optional)* `{default_zone, zones{name → {room_size, decay, damping, wet, dry, spread, hipass, predelay_ms, predelay_feedback, godot{room_size, damping, wet, dry, spread, hipass}}}}` — absent = no reverb bus. The web engine reads `decay` and `predelay_ms` (Godot has no such knobs, ch. 8.3); Godot reads `dry`, `spread` and the `godot` override; `predelay_feedback` is written but unused
`master_volume`	number	*(optional)* Master fader gain (ch. 9.2), linear; written only when ≠ 1.0 — engines assume 1.0 when absent

Every event

Key	Notes
`type`	`"music"` or `"sfx"`
`volume`	Linear event gain
`levels`	`[{name}, …]` — index = what `set_level()` takes
`tracks`	See *Track* below
`start_fade` / `release_fade`	`{duration, curve}`; absent/0 = instant (with engine anti-click)
`spatial`	*(optional)* see below
`reverb`	*(optional)* `{enabled, send}`
`intensity`	*(optional)* see below

Music events add: `bpm`, `time_signature [num, den]`, `fade_duration`, `fade_curve`, `switch_quantization` (level-switch defaults), `use_nodes` + `node_graph` (advanced mode), `sub_levels` — `{level_index → [{name, volume, tracks[]}]}`, each sub track in the *Track* shape below.

SFX events add: `cooldown_ms`, `max_instances`, `loop {enabled, mode: "seamless"|"smart"}`.

Track

Key	Notes
`name`, `volume`	
`level_behavior`	`"level_specific"` (per-level sample sets) or `"always_play"` (one shared set)
`levels`	map `level_index → {samples[], clips[], loop_bars?, switch_chance, play_chance}`
`samples` / `clips`	at track root instead, when `always_play`
`loop_bars`	Musical loop length — computed at export from the clip at the event's BPM
`randomization`	SFX: `{pitch, rate, delay, volume}` each `{enabled, min, max}`; a `pan` block is written in the same shape but reserved — no player applies it
`repeat_count`, `chance_reroll_bars`	Variation hold count; loop cycles between play/silence re-rolls (legacy key name — it counts loops, not bars)
`randomization_algo`	`"random"` | `"weighted"` | `"no_repeat"` (variation switching)
`play_chance_mode`	`"true_random"` | `"weighted_history"` | `"shuffle_bag"` (dice behavior)
`chance_fade_mode`, `chance_fade_duration`	`"instant"` | `"fade"` + seconds

Clip

`{source_offset, source_play_length, loop_mode: "loop_source"|"one_shot", fade_in, fade_out, volume, weight}` — after export `volume` is `1.0` (baked into the audio, ch. 11.2); `weight` feeds weighted

variation picks. Inside a **stinger** bundle the same clip record is written, but both engines read only `volume` and `weight` there — trim and fades are reserved fields on a stinger clip, and a stinger clip's `volume` is applied at fire time, not baked (ch. 11.2).

Spatial block

```
"spatial": {
  "enabled": true, "mode": "2D"|"3D",
  "model": "inverse"|"linear"|"exponential",
  "min_distance": 1.0, "max_distance": 50.0, "rolloff": 1.0,
  "panning_model": "equalpower"|"HRTF", "max_pan": 1.0,
  "doppler": 0.0,
  "lpf_enabled": true, "lpf_cutoff_hz": 5000.0, "lpf_db": -24.0,
  "godot": {
    "mode": "2D"|"3D",
    "model": "inverse"|"inverse_square"|"logarithmic"|"linear",
    "min_distance": 2.0, "max_distance": 4.0, "rolloff": 1.0, "max_pan": 1.0
  }
}
```

Flat keys are the canonical (Web) character; the optional `godot` object is the per-platform override authored on the Godot tab (ch. 7.4). It exists only once that tab has been edited, and then carries exactly these six keys: `mode`, `model` (Godot's native attenuation models — `linear` is the plain fade to silence at `max_distance`), `min_distance`, `max_distance`, `rolloff` and `max_pan`. Export writes all six every time; a key the tab did not set is filled from the flat value of the same name (`model` from `inverse`). The Godot engine reads the object in place of the flat keys — an older config without it uses the flat keys, mapping the web model to the nearest native one — and the web engine ignores it. Runtime merge order: engine defaults ← event block ← game-side override; the game itself passes only positions.

Intensity block

```
"intensity": {
  "enabled": true,
  "vol_db": [-18.0, 0.0], "vol_curve": "exponential",
  "pitch_st": [0.0, 0.0], "pitch_curve": "linear"
}
```

Node graph (flattened by export)

`node_graph.entry_node` + `nodes{}` + `connections[]`. Loop nodes carry `id`, `name`, `type`, `level_index`, `duration_bars`, `loop_count` (0 = ∞), per-section `bpm` / `time_signature` / `free_tempo` (from Tempo nodes), `restart_on_entry`, `trigger`, optional `entry_fade` / `terminal_fade` (dangling fade nodes, ch. 6.4 — each `{fade_duration, fade_curve}`). Every connection carries `id`, `source`, `target`, `conn_type` and, on a transition, `trigger`, `auto`, `probability`, `priority`; the `id` names the road — `A>B` for a direct edge, `A>B|X` through transition node X, `C>B|FO+FI` for a serial chain — and the engines key the road's stinger bundle and its no-repeat history by it, so each direction of a bidirectional transition and each road between

the same pair keeps its own; connections flattened from a **fade node** additionally carry `fade_duration`, `fade_curve`, `quantize`, `kind`, `fade_direction` and an optional embedded `stinger {samples, clips, volume, play_chance, mode}`; a flattened `fade_out → fade_in` chain adds `entry_fade_duration` / `entry_fade_curve` for its entry half. One connection per road: two transition nodes between the same two sections give two connections, and the engine picks between them by priority, then probability (ch. 6.4). A **direct** section-to-section edge carries none of the fade fields — the engines fall back to the event-level `fade_duration` / `fade_curve` / `switch_quantization` for it.

Enums

Where	Values
Fade curves	`linear`, `equal_power`, `s_curve`, `exponential`, `logarithmic`
Quantize	`bar`, `beat`, `instant`
Transition kinds	`crossfade`, `fade_in`, `fade_out`
Directions	`both`, `forward`, `reverse`
Loop (SFX)	`seamless`, `smart`
Clip loop	`loop_source`, `one_shot`

Defaults when a block is absent

Absent	Engines assume
`spatial`	A flat `trigger()` plays dry and centered; the positioned calls (`trigger_spatial*` / `triggerSpatial`, emitters) use the engine's default spatial character
`reverb`	No send
`intensity`	Full-strength triggers
`loop` (SFX)	One-shot
`reverb_zones`	No reverb bus at all
`godot` override	Godot mirrors the flat (Web) values

14.2 Keyboard shortcuts

Context	Shortcut	Action
Main	`Space` / `Esc`	Play–stop / stop
Main	`Cmd+S` / `Cmd+Shift+S` / `Cmd+O`	Save / Save As / Open
Main	`Delete` / `Backspace`	Remove selected track (track list focused)
Timeline	`Ctrl+wheel`	Zoom 1–8×
Node editor	`+` / `=` · `−` · `0`	Zoom in · out · fit
Node editor	`Space`	Play / Stop
Node editor	`Delete` / `Backspace`	Delete node / connection
Faders & sliders	Double-click	Reset (0 dB / 100% / 2.0 s)

14.3 .pfabank in one box

A 64-byte header (`PFABANK` magic, version, UIDs, section directory) + a **JSON section** (the entire config, plus an `_audio_entries` index of `path → {offset, length, codec}`) + a **DATA section** with all audio files 16-byte aligned. Codecs: 1 = OGG, 2 = MP3, 3 = WAV, 4 = Opus. One fetch, one parse, zero per-file requests — built for web loading (ch. 11.4).

14.4 Engine parity notes

- Both engines read the same config; the same export drives web and Godot builds.
- **Web extras:** HRTF panning option (3D), reverb send scales as `send × zone wet`.
- **Godot extras:** native attenuation models on the spatial Godot tab; zone wet level comes from the zone (send switches routing); optional SFX→WAV export path for one-shot latency (ch. 11.3).
- Authored once, platform overrides only where the platforms genuinely differ (spatial Godot tab, reverb zone Godot tab) — the ↺ Follow Web buttons re-link to the canonical character.
- Editor preview limits are listed in ch. 9.1 — spatial, reverb DSP and intensity between its endpoints are auditioned in-game (demo games / Live Connect); everything else is auditioned in Studio as the game will play it, with the differences ch. 9.1 lists (seam curves, the Godot one-shot window end, first-visit dice, sub switches, a click's route, and the four arbitration cases — the section that already plays requested while a switch waits, a second Stop, an interrupted terminal fade, a section entered from itself).