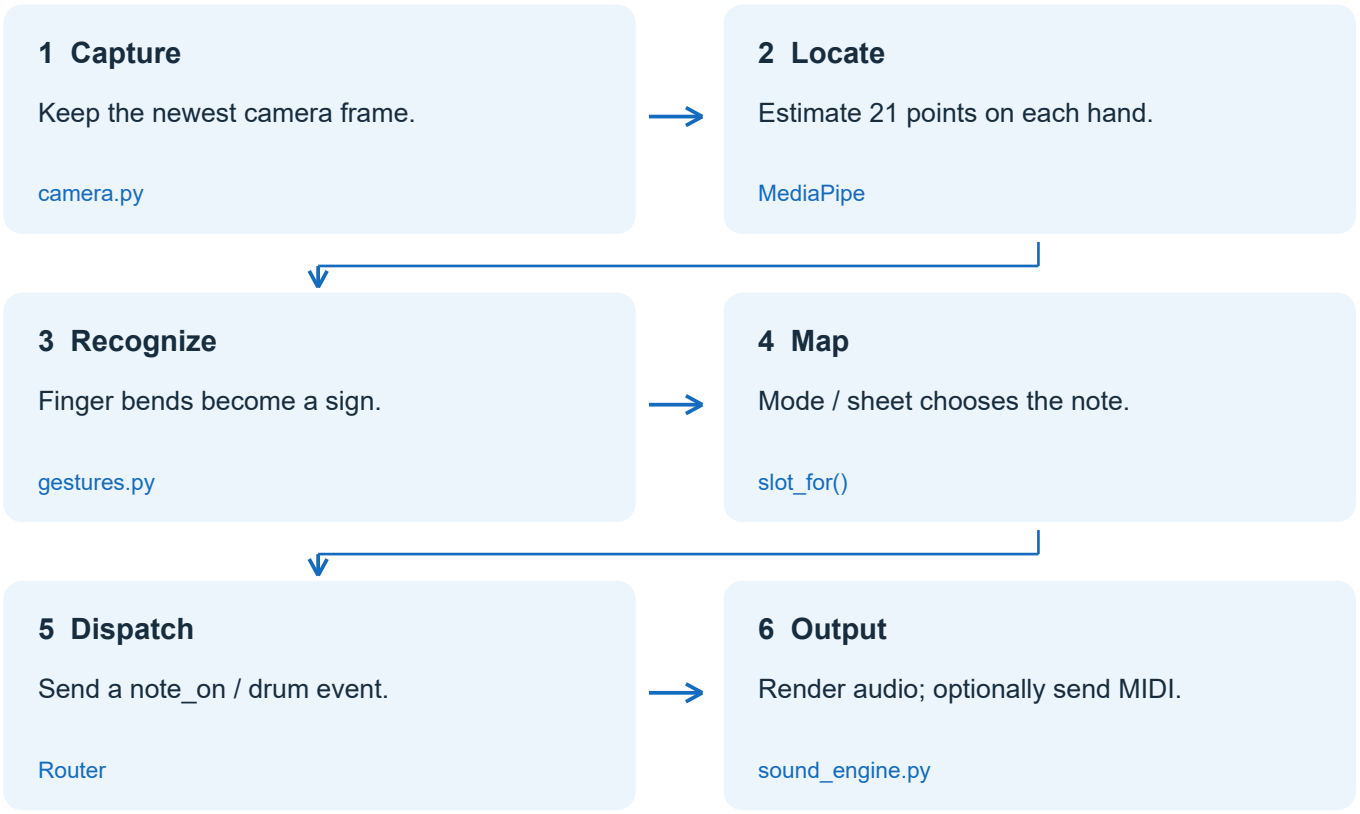
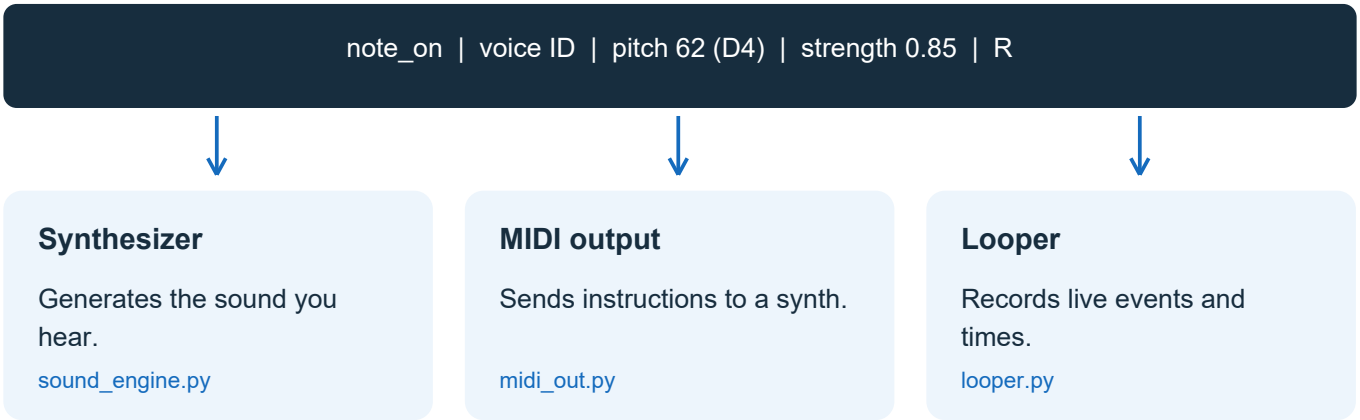
From hand to music

The camera measures your hand. A sign selects a sound. A musical event tells the rest of the program what to do.



One shared event, several destinations

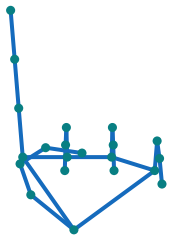


Display branch: background.py blurs a copy for the preview. Hand tracking still receives the sharp image. capture.py records the preview plus audio.

Your hand-sign dictionary

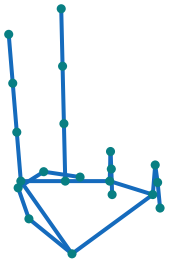
One shape per hand selects one sound slot. The mode or selected song decides what that slot plays.

Slot 1 / 1 finger



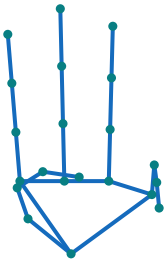
Index

Slot 2 / 2 fingers



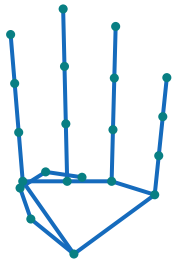
Index + middle

Slot 3 / 3 fingers



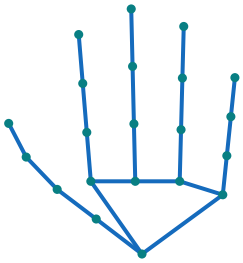
Index + middle + ring

Slot 4 / 4 fingers



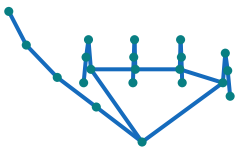
Thumb tucked

Slot 5 / Open hand



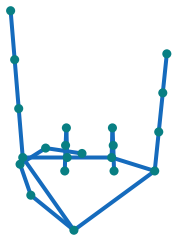
Thumb spread

Slot 6 / Thumbs up



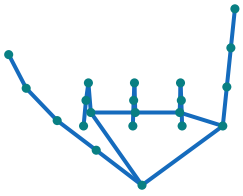
Thumb up, fingers down

Slot 7 / Rock



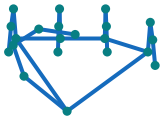
Index + pinky

Slot 8 / Call me



Pinky; thumb optional

Fist

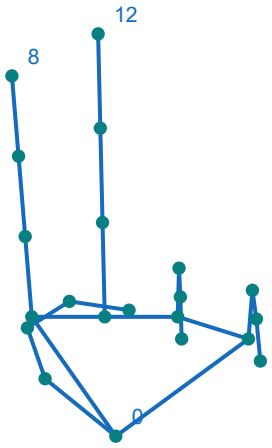


Silence; thumb tucked

Basic: slots 1-4; thumb ignored. **Extended:** slots 1-8.
Keep the palm visible. In Extended mode, the thumb separates 4 fingers from open hand, and fist from thumbs up.

How the program reads a sign

MediaPipe estimates positions; your code turns the geometry into a stable decision. Recognition is a rule-based step after tracking.



21 tracked landmarks

Index Middle Ring Pinky

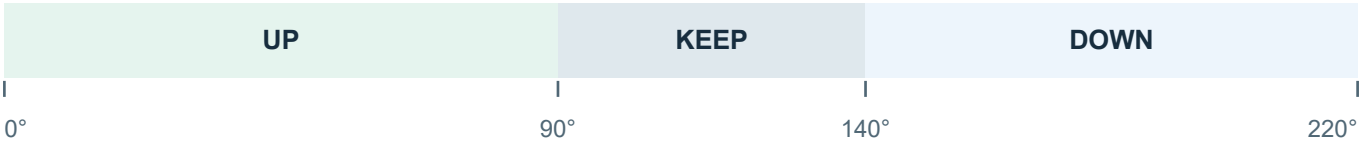
UP UP DOWN DOWN

1 1 0 0

Pattern match: 2 fingers

The thumb is ignored for this sign. An unrecognized pattern does not select a new sound.

A half-bent finger keeps its previous state



Example: 40° → UP; 110° → still UP; 160° → DOWN; back to 110° → still DOWN. Separate thresholds stop flicker near a boundary. 90° and 140° are the defaults; calibration moves both lines for each finger (V9).

Two frames confirm a change

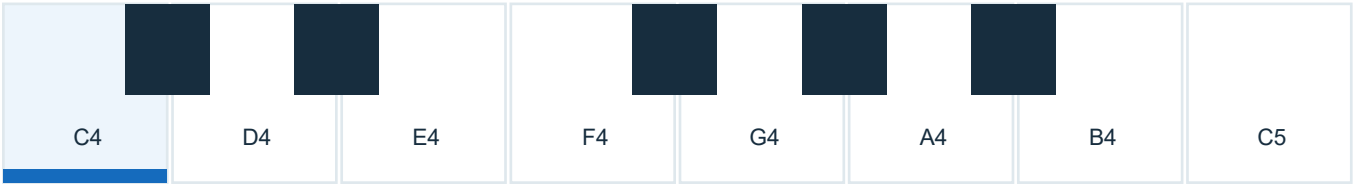
Frame 1	Frame 2	Action
Candidate sign; save onset	Same sign; confirm it	Send note_on

Game timing uses Frame 1, so the confirmation frame does not count against you. This removes the confirmation penalty, not the physical camera or audio delay.

Piano notes, chords and arpeggios

These are three different ways to use pitches. A sign is a control; it does not inherently mean a chord.

A note: one pitch



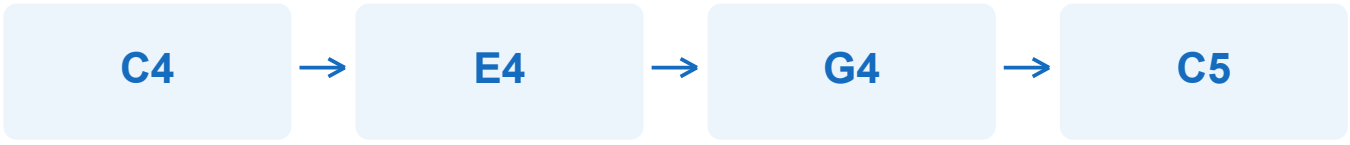
C4 alone → one voice. It is about 261.6 Hz.

A chord: pitches together



C major → C4 + E4 + G4 at the same time.

An arpeggio: pitches one after another



Beat 1 Beat 2 Beat 3 Beat 4

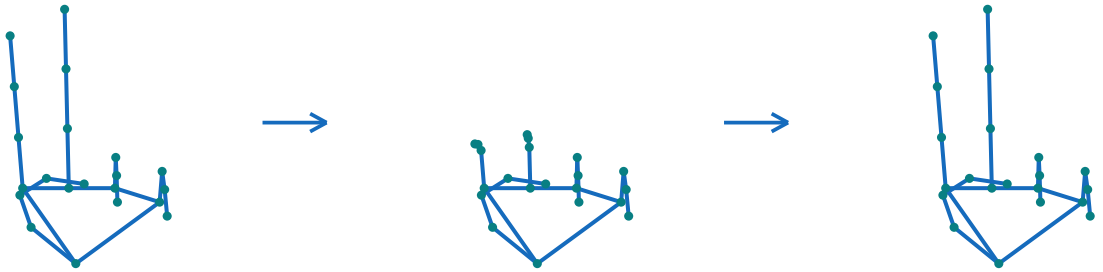
In piano play: each sign selects an individual note. The song sheet tells you which sign to make next. Low notes can be assigned to the left hand and high notes to the right.

Charts from your music: a *Melody* chart gives each sign one note of the tune; a *Chords* chart gives each sign a whole chord. Both are made from your own file (V12).

Strike again without changing the note

Holding the same sign sustains its selection. To repeat it, the program needs a new strike event.

1 HOLD **2 FOLD** **3 STRAIGHTEN**



One note starts. About 40° past baseline. Return within 18°: strike.

note_on → hold → note_off + note_on

Alternative: pump. Keep the sign and flick the whole hand downward. Lift or slow the hand before the next strike. Pump is disabled in Bend mode, where the same movement changes pitch.

A looper remembers actions, not a microphone recording

Beat	1	2	3	4
Record	C4	hold	E4	release
Replay	C4	hold	E4	release
Overdub	Kick	.	Snare	.

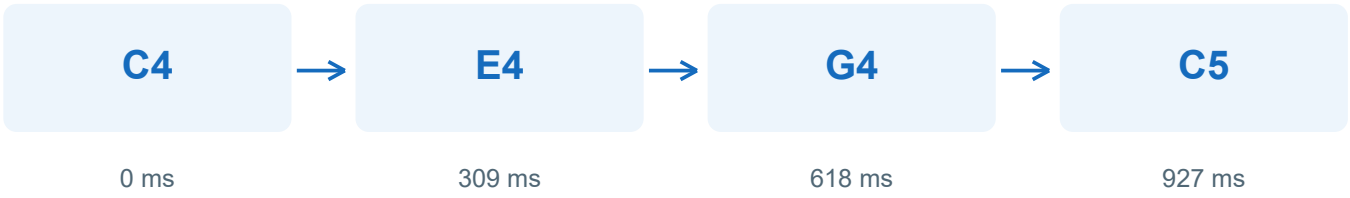
Why events matter: the same loop can play through the built-in synth or MIDI. Demo capture is different: it saves the actual sound and camera view.

Follow the rhythm

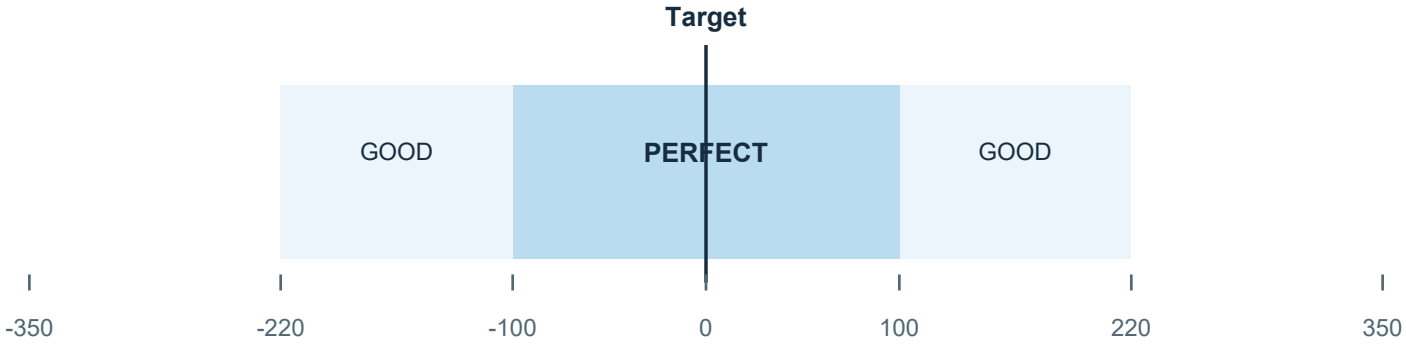
The game checks the hand, the sign and the time it first appeared. The visual line is the target time for each note.

194 BPM

One beat ≈ 309 ms | Four beats ≈ 1.24 s



When does the sign land?



Timing error in milliseconds: early ← 0 → late

+60 ms late	Perfect: within 100 ms
+170 ms late	Good: within 220 ms
+260 ms late	Miss: outside the window

Camera offset example: first seen at +260 ms, minus a 60 ms offset = +200 ms, so the matching sign becomes Good. Adjust the offset only to compensate for consistent delay.

Tempo is not note density. At 194 BPM, one note per beat is 3.23 notes/s; one every two beats is 1.62 notes/s. Fast notes alone do not establish the BPM.

Shape the sound, protect the timing

Note choice, volume, pitch bend and instrument sound are independent controls.

PITCH

Which note? C4 ≈ 261.6 Hz.

VOLUME

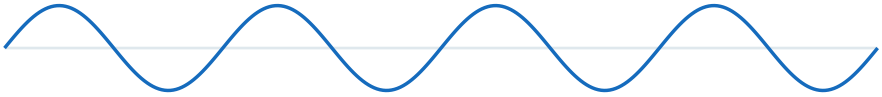
How loud? A faster strike, or a higher hand.

TIMBRE

What character? 10 instruments, one per hand.

A waveform explains the difference

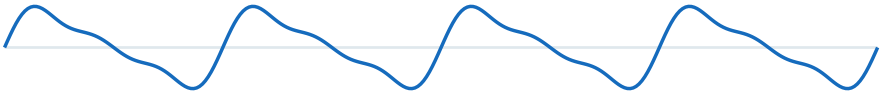
Base note



Same pitch, quieter



Same pitch, more overtones



Pitch bend: moving the hand changes frequency smoothly, up to ±2 semitones. **MIDI:** sends note and expression messages; the receiving synth generates its own sound.

Two paths share the sharp frame

Tracking path

Sharp image → landmarks → events

Display path

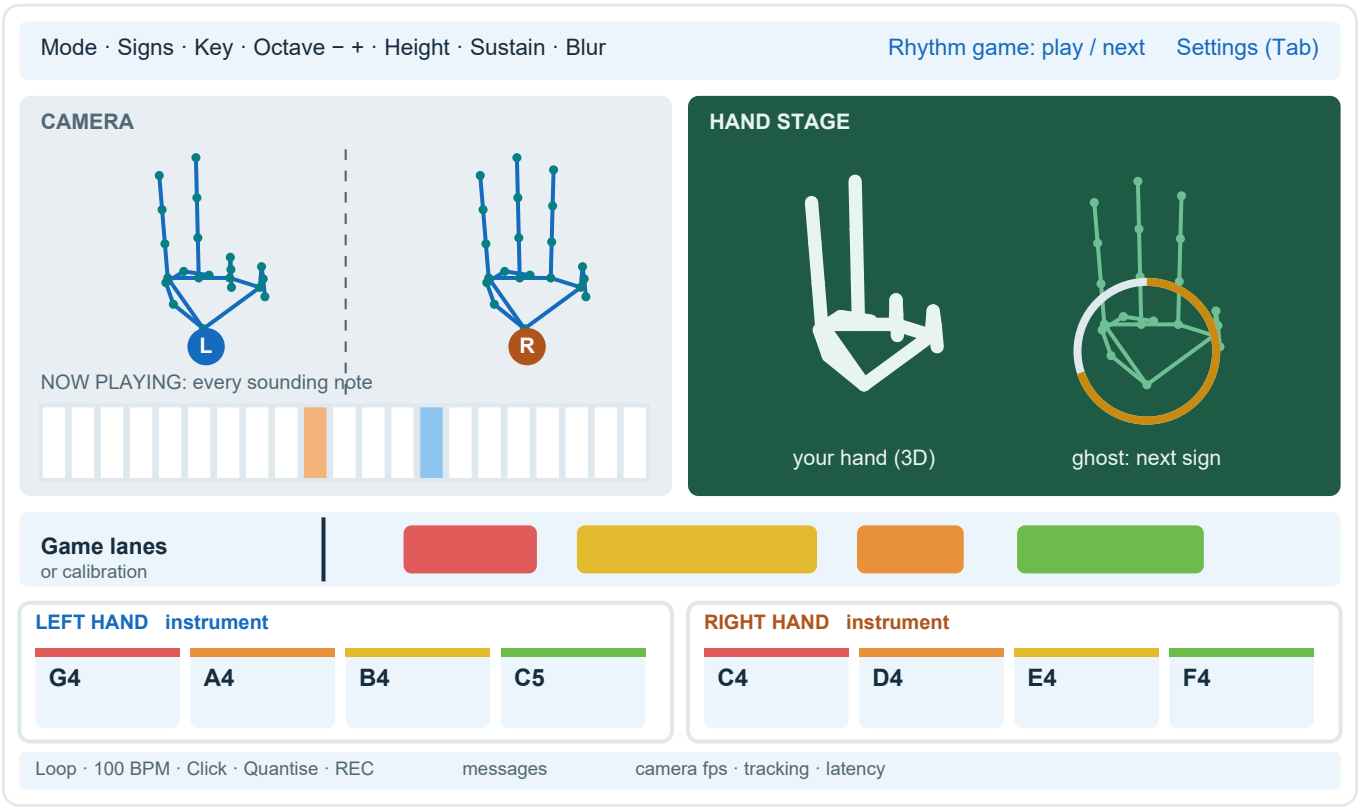
Person mask + blur → preview

Run alongside one another

Delay adds up: camera sampling + tracking + confirmation + audio output. A camera at 10 fps supplies a frame only every 100 ms; at 30 fps, every 33 ms. Bright light helps maintain frame rate.

Your window at a glance

Everything the program knows is on screen, and every chip, pad and button can also be clicked.



Click anything

Chips cycle their setting, a pad plays its sound, and the footer chips run the looper.

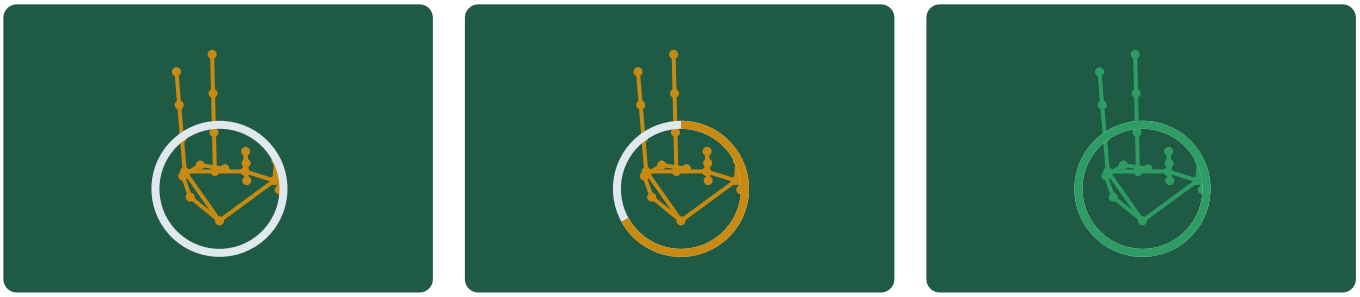
Hand stage

Your hands rebuilt from 3D landmarks on a plain colour: front, three-quarter or side, with trails.

Settings (Tab)

Instruments · Key mapping · Looper & MIDI · Sessions · Songs · Calibrate · Display · Keys

Ghost hand: the next sign, before it is due



3.0 s before: appears | 1.0 s before: ring fills | on time and matched

The ghost sits over your own hand, labelled with its sign and sound: *Next: 2 fingers → C5*.

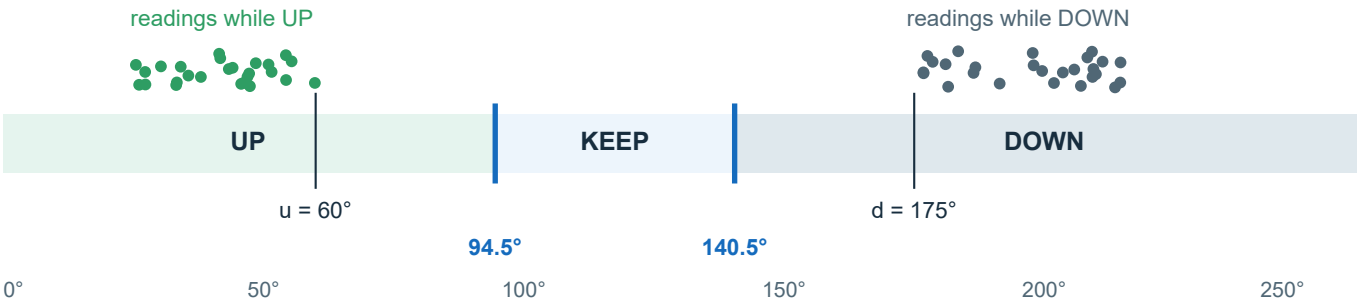
Calibrate once, read every sign reliably

Hold each sign for two seconds. The program learns how your own fingers bend and moves both thresholds to fit them.



Basic: 5 steps (about 18 s). Extended adds open hand, thumbs up, rock and call me (9 steps) and also fits the thumb. Both hands are measured at once; **Skip** passes a sign you cannot make.

Fitting one finger



u = 95th percentile of the UP readings, **d** = 5th percentile of the DOWN readings (outliers ignored). The gap between them is split 30% / 40% / 30%: **up below u + 0.3·gap**, **down above d − 0.3·gap**, and the middle 40% keeps the last state.

A stiffer finger that reads up to 110° when raised gets *up below 122°*: with the default 90° line it would often sit in the KEEP zone and be misread.

Check before you save

fist 100% | **1 finger 100%** | **2 fingers 98%** | **3 fingers 76%** | **4 fingers 94%**

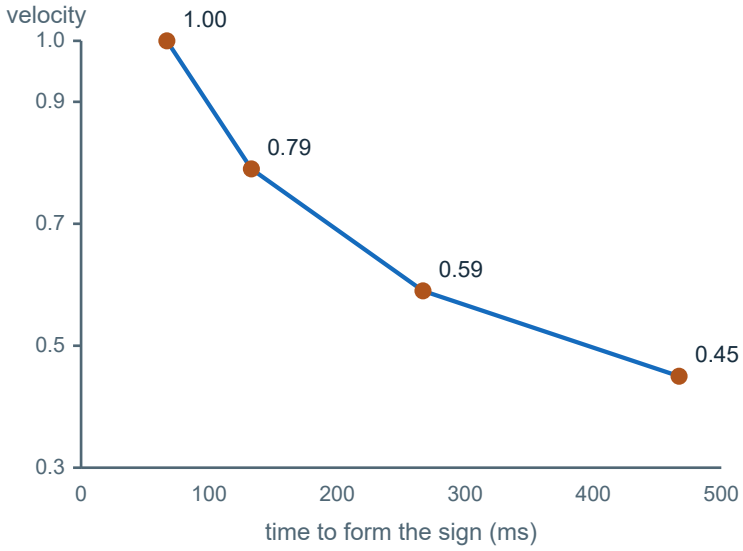
Every collected frame is read again with the new lines. Under 90% (amber) that sign may still be misread: try again, or skip it. Fingers whose UP and DOWN readings overlap are flagged.

Saved per hand

settings.json keeps both lines for every finger and the thumb. Settings > Calibrate shows them, and can reset to 90° / 140°.

Strike harder, play louder; mix each hand

Velocity comes from how fast your fingers moved just before the sign. Each hand then has its own volume and reverb.



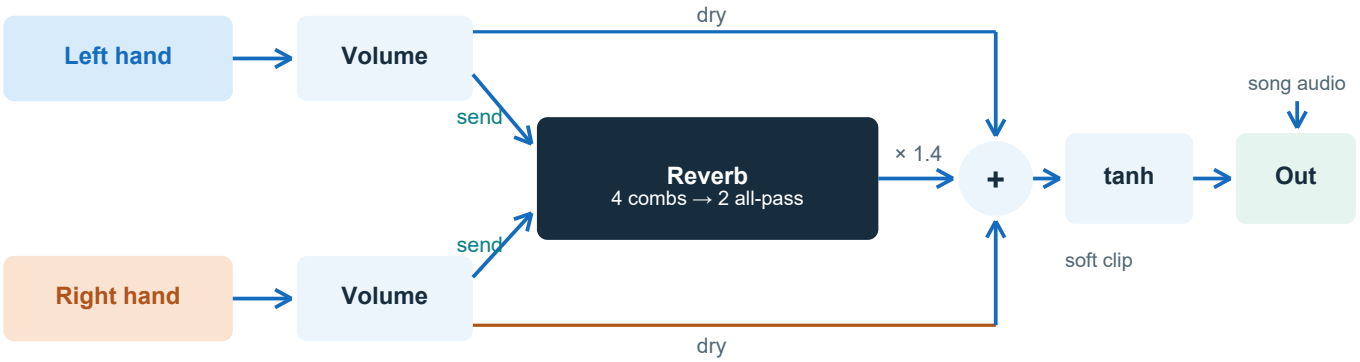
Fingers: the fastest change in finger bend over the last 0.15 s (degrees per second). **Pump:** the hand's speed (hand lengths per second).

strength = (speed − floor) ÷ scale

velocity = 0.3 + 0.7 × strength

The floor ignores tracking jitter, so a slow sign is soft. **i** turns dynamics off.

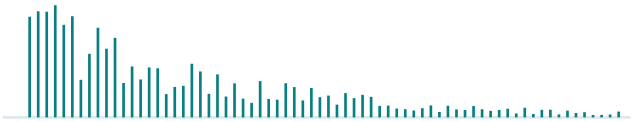
The mixer: volume and reverb per hand



What the reverb does to one click



comb filters: a train of echoes, each quieter and duller



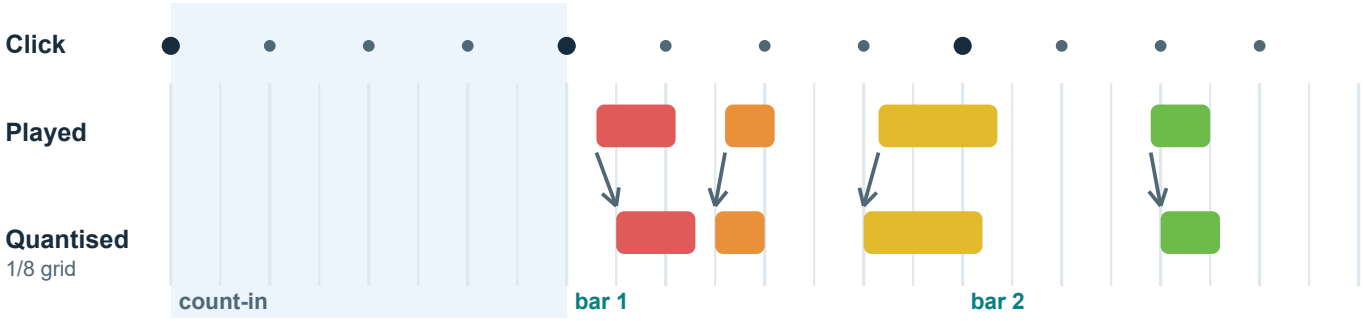
all-pass filters: echoes smeared into a smooth wash

Every delay (441 to 1,356 samples) is longer than one 256-sample audio block, so each block is computed with a few whole-array NumPy operations: fast enough for real time.

The song's own recording joins after the soft clip, since it is already mixed.

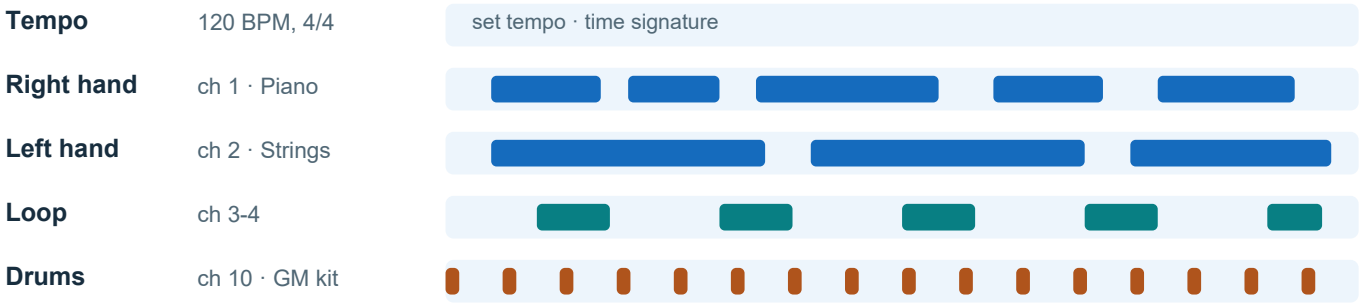
Loop in time, keep it as MIDI

A metronome and count-in put you on the beat, quantise tidies what you played, and any performance can be saved as a MIDI file.



Each start moves to the nearest grid step; the note keeps its length. With the click or quantise on, the first take is rounded to whole bars, so the loop stays in time with the click.

A MIDI file of what you played



F7 records everything you play live; **F8** saves the loop. Times are stored in ticks: **tick = seconds × BPM ÷ 60 × 480**. Velocities come from your strike speed, and each hand's instrument becomes a General MIDI program.

Hand sessions: replay without a camera

F5 record
Only the 21 points of each hand, 30 times a second.

→

.gmsession
Compressed text; about 20 KB per second; no images.

→

F6 / --demo
Replays in real time, looping.

→

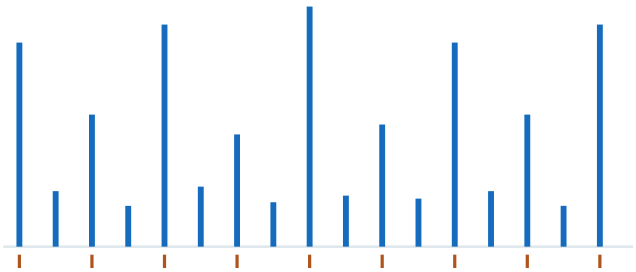
Same pipeline
Fed to process() exactly like the camera.

Replay a misread sign with the debug view on, try a new setting on exactly the same movement, or show the instrument with no webcam: **--demo** plays a 20-second Band-mode performance.

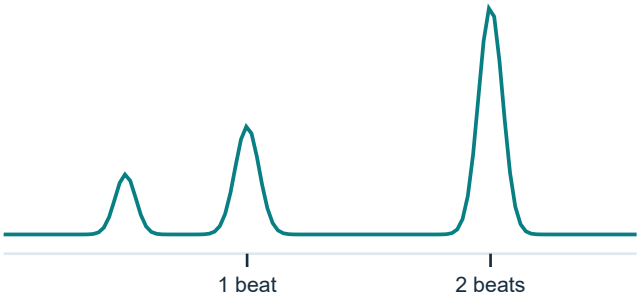
Turn your own music into a chart

The program listens to the file: where the beats are, which chords sound and what the tune is. You check it by ear, then pick a chart.

Audio file → Spectrogram → Onsets → Tempo + beats → Notes → Song sheet



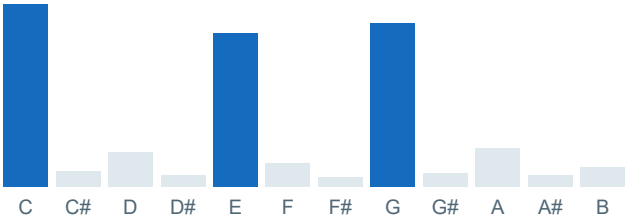
Onset envelope: sudden increases in the spectrum (spectral flux). Orange ticks: the beat grid found.



Autocorrelation: how alike the envelope is to itself when shifted. A kick on beats 1 and 3 peaks at 2 beats; when the 1-beat pulse is also clear, the faster tempo wins.

Notes from the spectrum

Chroma of one beat → C major



Chords: each beat's 12 note energies are compared with 24 major and minor chord shapes. A Viterbi pass charges a small cost for every change, so passing melody notes don't make the chords flicker.
Key: the song's note totals against major and minor key profiles.
Melody: the highest strong pitch every half beat, since the tune usually sits on top.

Check it by ear, then choose a chart

Listen with clicks

A rim click on every beat, louder on bar 1. Clicks drifting: fix the tempo. Steady but off: move bar 1.

Fix it, or tap it

±0.1 / ±1 / ×2 / ÷2 BPM; ±0.01 s / ±1 beat. Or tap Space on the beats: a straight line through your taps.

Pick a chart

Melody, Chords or Rhythm; Easy, Normal or Hard. A preview shows the pads and first bars.

Melody: low notes on the left hand, high on the right, like a keyboard. **Chords:** one chord per sign. **Rhythm:** kick and snare on the left, hi-hat on the right.

Tempo, bar 1, key and chords are reliable with a steady beat; the melody is a starting point in a full mix. The sheet is plain text, so any note can be fixed.

Connect the pictures to your code

Read the visual idea first, then jump into the handbook for the implementation and theory.

CONCEPT	CODE OWNER
Get a recent camera image	camera.py
Measure, confirm and time hand signs	gestures.py
Fit thresholds to your hands	calibration.py
Choose notes and dispatch events	gesture_music.py
Draw the window; mouse and keys	ui.py
Generate sound; mix and reverb	sound_engine.py / drums.py
Send external instrument messages	midi_out.py
Write and read MIDI files	midi_file.py
Record and replay timed actions	looper.py
Record and replay hand sessions	session.py / handmodel.py
Parse charts, schedule and judge	game.py / songs/
Analyse music; write song sheets	audio_analysis.py / songmaker.py
Remember settings and presets	settings.py
Blur the preview; record a demo	background.py / capture.py

Jump into the full handbook

Play: window, signs, sounds Handbook p. 10
Window, signs, modes, instruments, dynamics, presets

Calibrate and record Handbook p. 14
Calibration, looper, MIDI files, hand sessions

Understand the architecture Handbook p. 21
Data flow, events, threads and every module

Understand gesture geometry Handbook p. 30
Angles, thresholds, calibration fit, strike speed

Understand sound Handbook p. 35
Partials, envelopes, mixer and reverb

Understand music analysis Handbook p. 43
Tempo, beats, chords and melody from a recording

The next page begins the complete handbook. Its printed page numbers are unchanged; PDF page numbers are 13 higher.

HANDBOOK

Gesture Music

A webcam instrument played with hand signs. How to use it, how the code is built, and the theory behind every part.

Python 3.13 · OpenCV · MediaPipe · NumPy · pygame · sounddevice · soundfile · mido

Project folder: gesture-music · 18 modules, about 6,000 lines, plus tests

Edition of 3 November 2026: calibration, hand sessions, metronome and quantise, MIDI files, strike dynamics, mixer and reverb, key-mapping presets, ghost-hand hints, songs from your own music

Contents

1. What this project is 6
Features at a glance 6
How the pieces fit, in one paragraph 6
2. Setting up and running 8
Requirements 8
Install 8
Libraries and what each is for 8
Command-line options 8
Running the tests 9
3. Playing the instrument 10
The window 10
Hand signs 10
Which hand is which 11
Repeating the same note 11
Modes 11
Instruments, mixer and reverb 11
Dynamics: strike speed sets loudness 12
Custom mode: your own sound on every sign 12
Hand height: volume and pitch bend 12
Sustain, capture and blur 12
Pinch style 13
4. Calibrating your hands 14
Running the wizard 14
Reading the results 14
5. The rhythm game 15
How to play 15
Ghost-hand hints 15
Judging and scoring 15
Adding your own music 15
Writing your own song sheets 16
6. Recording: looper, MIDI files and hand sessions 17
The looper 17
MIDI files 17
Hand sessions and demo mode 17
7. MIDI and music software 18

8. Controls reference 19
9. Troubleshooting and tuning 20
It feels slow or jumpy 20
A sign isn't recognised 20
Left and right are swapped 20
10. Architecture 21
The pipeline 21
Events: the instrument's vocabulary 21
Threads 21
One frame, step by step 22
11. Module reference 23
gesture_music.py — the App (997 lines) 23
ui.py — the window (1,984 lines) 23
gestures.py — reading the hands (406 lines) 24
calibration.py — the calibration wizard (134 lines) 24
handmodel.py — a synthetic hand (83 lines) 25
session.py — hand sessions (109 lines) 25
midi_file.py — MIDI files (103 lines) 25
camera.py — always the newest frame (60 lines) 25
sound_engine.py — the real-time synthesizer (284 lines) 26
drums.py — the drum kit (103 lines) 26
midi_out.py — live MIDI output (116 lines) 26
looper.py — record and loop (242 lines) 26
game.py — the rhythm game (409 lines) 27
audio_analysis.py — listening to a recording (285 lines) 27
songmaker.py — from a music file to a song sheet (421 lines) 28
settings.py — remembering your choices (164 lines) 28
background.py, capture.py — blur and demo videos (67 and 72 lines) 28
tools/ and tests/ 28
12. Computer vision: finding hands 29
MediaPipe Hand Landmarker 29
Mirroring and handedness 29
13. Gesture geometry 30
Angles from vectors 30
Hysteresis: two thresholds instead of one 30
Classifying and confirming a sign 30

The thumb 30
Repeats: relative measurement and windowed speed 30
Calibration: fitting the thresholds to your hands 31
Strike speed: how loud 31
Pinch style 32
14. Signal smoothing and filtering 33
Exponential smoothing 33
The One Euro filter 33
15. Latency: from movement to sound 34
16. Sound synthesis 35
Digital audio basics 35
From note number to frequency 35
Additive synthesis and timbre 35
Envelopes and vibrato 35
Pitch bend with a phase accumulator 36
Mixing, reverb and soft clipping 36
The reverb 36
Drum synthesis 36
17. MIDI 38
Standard MIDI files 38
18. Music theory used in the project 39
Notes, semitones and octaves 39
Scales 39
Intervals and chords 39
Chords in a key (Roman numerals) 39
Voicing 40
Tempo, beats and bars 40
Notes, chords and arpeggios 40
Keys and how to recognise one 40
19. Timing: the looper and the rhythm game 41
Recording events, not audio 41
Metronome, count-in and whole bars 41
Quantise 41
Scheduling the game 41
Judging a press 42
20. Music analysis: from a recording to a song sheet 43

1. A picture of the sound: the spectrogram 43
2. When does something start? Spectral flux 43
3. The tempo: autocorrelation 43
4. The beats: dynamic programming 43
5. Bar 1: the downbeat 44
6. Which notes: chroma, key and chords 44
7. The melody: harmonic salience 44
8. Correcting by ear: tap tempo 44
9. Writing the chart 44
21. Image processing: the background blur 46
Appendix A. Measured performance 47
Appendix B. Glossary 48

1. What this project is

Gesture Music turns a webcam into a musical instrument. The camera watches both of your hands; the shape you make with each hand (a **hand sign**) chooses a note, a chord or a drum, how fast you make it sets how loud it plays, and moving your hand bends or swells the sound. Everything you play can be looped over a metronome, saved as a MIDI file, sent to other music software, recorded as a demo video, or scored in a rhythm game that can build its songs from your own music files.

Features at a glance

Feature	What it does
Hand signs	Basic set: hold up 1–4 fingers. Extended set: 8 signs per hand (adds open hand, thumbs up, rock, call me). A fist is silence.
Calibration	Hold each sign for 2 seconds; the finger thresholds are fitted to your hands, camera and light, with a score per sign.
Two hands, clearly labelled	The hand on the left half of the screen is your left hand. Each hand is drawn in its own colour with an L/R badge.
The window	Camera view, a hand stage (your hands redrawn in 3D on a plain background), clickable pads and a settings panel.
Modes	Notes, Chords, Drums, Band (chords + drums), Custom (your own sound on every sign) and Song (a song sheet chooses every sound).
Key-mapping presets	Save, load, import and export named Custom mappings as small JSON files.
Repeating a note	Tap (fold the raised fingers halfway and straighten) or pump (a quick downward flick) replays the note without changing sign.
Dynamics	A quick, snappy sign plays louder than a slow one: velocity from finger or hand speed.
Hand-height expression	Volume: raise your hand for louder. Bend: move up or down while holding to bend the pitch up to ±2 semitones.
Real-time synthesizer	10 instruments generated live, chosen per hand, with per-hand volume and reverb; a 12-piece synthesized drum kit (or your own WAV samples).
Looper	Record a phrase, stack overdub layers, undo, stop and clear; metronome, one-bar count-in and quantise.
Rhythm game	Timed song sheets scroll toward a hit line; make the right sign on the beat. A ghost hand shows the next sign. Perfect/Good/Miss, combo, grades, best scores.
Songs from your music	Add an audio or MIDI file: tempo, bar start, key, chords and melody are detected and a song sheet is written. The recording plays along.
MIDI	Live MIDI out to a DAW or hardware synth, and standard MIDI files of a performance or the loop.
Hand sessions	Record your hand movements (landmarks only) and replay them for debugging, tests, or a demo without a camera.
Background blur	Keeps you sharp and softens the room behind you, on screen and in demo videos.
Demo capture	Saves the whole window and the exact audio you heard, merged into one MP4 when ffmpeg is installed.

How the pieces fit, in one paragraph

A background thread reads the newest webcam frame (or a recorded session replays one). MediaPipe finds 21 landmarks on each hand. `gestures.py` turns those landmarks into a sign (which sound), a strike speed (how loud) and a hand height (how to shape it), using thresholds the calibration fitted to your hands. Every musical action becomes a small **event** such as `("note_on", id, note, velocity, hand, instrument)`, which a router sends to the synthesizer, the MIDI port, the looper and the MIDI-file recorder at the same

time. The rhythm game listens to the same presses to judge your timing, and `ui.py` draws it all in a pygame window. Part II explains each step in detail and Part III explains the theory behind it.

2. Setting up and running

Requirements

- Windows 10/11 with a webcam (the code also runs on macOS/Linux with small camera changes).
- Python 3.13 (MediaPipe does not yet support 3.14).
- Speakers or headphones. Optional: a MIDI device or the free loopMIDI virtual cable for a DAW; ffmpeg for merged demo videos.
- No webcam yet? `--demo` replays a recorded performance, so everything except live tracking can be tried.

Install

```
py -3.13 -m venv .venv
.venv\Scripts\python -m pip install -r requirements.txt
.venv\Scripts\python gesture_music.py
```

On the first run the program downloads two small model files from Google into the project folder: `hand_landmarker.task` (about 7.8 MB, the hand tracker) and `selfie_segmenter.tflite` (about 250 KB, used for the background blur).

Libraries and what each is for

Library	Used for
mediapipe	Hand landmark detection (21 points per hand, in 2D and 3D) and selfie segmentation for the blur.
opencv-python (cv2)	Reading the webcam, image conversion, blur and blending, video writing.
pygame	The window: drawing, text, mouse and keyboard. Its bundled PortMidi also sends MIDI (no compiler needed on Windows).
numpy	All sound synthesis (arrays of samples), drums, reverb, music analysis (FFT), calibration statistics, maths on landmarks.
sounddevice	Real-time audio output through PortAudio, using a callback that is asked for the next block of samples.
soundfile	Reading music files (WAV, MP3, FLAC, OGG, AIFF) for the song importer.
mido	MIDI messages for live output, and reading and writing standard MIDI files.
tkinter (built in)	The native Open/Save dialogs for music files and key-mapping presets.

Command-line options

Option	Effect
`--camera 1`	Use a different webcam (default 0).
`--root 48`	Start an octave lower. The root is a MIDI note number; 60 is middle C.
`--signs extended`	Start with 8 signs per hand instead of 4.
`--style pinch`	Start in pinch style (thumb touches a fingertip) instead of hand signs.
`--confirm 2`	Frames a new sign must be seen before it plays. 1 is fastest, 3 is steadiest.
`--game-offset 60`	Rhythm game: your camera's delay in milliseconds, subtracted when judging timing.
`--no-blur`	Start with the background blur off.
`--midi / --midi loopMIDI`	Send MIDI to a virtual cable or device (optionally one whose name contains the text).
`--list-midi`	List MIDI output ports and exit.
`--mute`	Turn off the built-in synthesizer (MIDI only).

Option	Effect
`--lookahead 40`	Pinch style only: trigger fast pinches this many ms early to hide camera delay.
`--window 1600x900`	Window size in pixels (default: fits your screen).
`--demo`	Replay the built-in demo session instead of using the camera (Band mode; your saved mode is kept).
`--replay FILE`	Replay a recorded hand session instead of using the camera.
`--no-camera`	Stay without a camera when a replay is stopped.

Running the tests

```
.venv\Scripts\python -m unittest discover tests -v
```

The 14 tests need no camera or sound card and write only to temporary folders. They use the synthetic hand model and synthesized music, so they always give the same result: every sign on both hands, strike speed ordering, calibrated thresholds, the calibration fit (including overlapping readings and an unseen hand), looper quantise and export, a MIDI file round trip, a session record and replay, preset validation, and the whole song import (tempo, bar start, key, chords, melody, kick and snare, and a playable sheet for every chart and difficulty).

3. Playing the instrument

The window

```
+- top bar: mode, signs, key, octave -/+, height, sustain, blur, song, Settings -+
|------------- CAMERA --------------|------------ HAND STAGE ------------------|
| mirrored camera view, skeletons,  | your tracked hands redrawn as 3D models  |
| L/R badges, the sound per hand    | on a plain background, with trails;      |
| NOW PLAYING: 5-octave keyboard    | ghost hand = next sign in the game       |
|------- rhythm-game lanes (during a song) or the calibration wizard ---------|
|-- LEFT HAND: instrument, pads ----|-- RIGHT HAND: instrument, pads ----------|
+- footer: looper, BPM, Click, Quantise, REC, messages, fps / latency --------+
```

- **Everything is clickable.** The chips in the top bar cycle their setting. Clicking a pad plays it, so you can try sounds without the camera. The footer's BPM, Click and Quantise chips control the looper.
- **Camera view:** your mirrored image (optionally with the background blurred), each hand's skeleton in its colour with an L/R badge, and the sound each hand is playing. Below it, a five-octave keyboard lights every sounding note in the hand's colour.
- **Hand stage:** your hands redrawn from MediaPipe's 3D world landmarks on a single-colour background (5 colours; k), seen from the front, three-quarter or side (o), with motion trails. Raised fingers are bright, lowered ones dim, half-bent ones amber; the active sign's fingertips glow in the pad's colour; a ring flashes on a tap or pump.
- **Pads:** one strip per hand with its instrument menu and a pad per sign, showing the sound, the sign's name and a small picture of it. A bar on a lit pad shows how hard it was struck.
- **Settings (`Tab`):** tabs down the left for Instruments, Key mapping, Looper & MIDI, Sessions, Songs, Calibrate, Display and Keys. Long tabs scroll with the mouse wheel.
- The window follows Windows display scaling (sharp on high-DPI screens), is resizable and goes full screen with `F11`. Choices are saved in `settings.json`.

Hand signs

Keep your palm facing the camera with your fingers spread a little. Each hand shows one sign at a time; changing sign changes the sound. A **fist** is silence.

Slot	Sign	Fingers up (index, middle, ring, pinky)	Thumb	Basic	Extended
1	1 finger	● ○ ○ ○	any	✓	✓
2	2 fingers (peace)	● ● ○ ○	any	✓	✓
3	3 fingers	● ● ● ○	any	✓	✓
4	4 fingers	● ● ● ●	Basic: any. Extended: tucked across the palm	✓	✓
5	Open hand	● ● ● ●	spread out		✓
6	Thumbs up	○ ○ ○ ○	up		✓
7	Rock	● ○ ○ ●	any		✓
8	Call me	○ ○ ○ ●	any		✓
–	Fist	○ ○ ○ ○	Extended: wrapped over the fingers	silence	silence

Press x to switch between the Basic and Extended sets. The thumb only matters in two places: 4 fingers versus open hand, and fist versus thumbs up. That is deliberate: the thumb is the finger the camera reads least reliably.

Which hand is which

- The view is mirrored, so your left hand appears on the left. **The hand on the left half of the screen is treated as your left hand.** A dashed line marks the centre.
- With two hands in view, the one further left is L and the other is R.
- With one hand near the centre line, it keeps the label it had in the previous frame, so it cannot flicker while crossing.
- If tracking loses a hand for up to 0.2 s, its notes keep playing instead of cutting out.
- Each hand is drawn in its colour (left blue, right orange) with an L or R badge on the wrist. Press `h` if they are ever backwards.

Repeating the same note

Changing sign plays a new note, but melodies often repeat a note. Without changing sign you can:

Gesture	How	When it plays
Tap	Fold your raised fingers about halfway forward, then straighten them. For thumbs up, bend and straighten the thumb.	As the fingers straighten again
Pump	A quick downward flick of the whole hand, like knocking. Lift the hand back up before the next one. Off in Bend mode.	On the downward flick
Fist	Close to a fist and open again. If the fist is too quick to register, it counts as a tap.	When the sign reappears

The screen flashes TAP or PUMP at your wrist when a repeat is recognised.

Modes

Press `m` to cycle through the modes. In each mode the same signs play different things:

Mode	Right hand	Left hand
Notes	Basic: scale notes 1–4. Extended: scale notes 1–8.	Basic: scale notes 5–8. Extended: notes 1–8 an octave lower.
Chords	Basic: chords I, IV, V, vi (C, F, G, Am in C major). Extended: I, ii, iii, IV, V, vi, vii°, I.	Melody notes, an octave above the chords.
Drums	Kick, snare, hat, open hat (Extended adds clap, low tom, high tom, crash).	Clap, low tom, high tom, crash (Extended: ride, rim, cowbell, shaker, clap, toms, crash).
Band	Chords, as in Chords mode.	Kick, snare, hat, clap (Extended adds open hat, crash, toms).
Custom	Your own mapping (Settings > Key mapping).	Your own mapping.
Song	Whatever the selected song sheet's `right:` line assigns.	Whatever its `left:` line assigns.

`s` changes the scale (Major, Minor, Pentatonic, Blues) and `[` and `]` move an octave down or up.

Instruments, mixer and reverb

Each hand has its own instrument: click the instrument button above its pads, use Settings > Instruments, or press `t` (right hand) and `y` (left hand). Loops keep the instrument they were recorded with, and over MIDI each hand's channel gets the matching General MIDI program.

Instrument	Character	GM program
Piano	Struck strings that fade; upper partials die first	0

Instrument	Character	GM program
Electric piano	Soft, bell-like keys with a slight shimmer	4
Organ	Steady, never fades; sub-octave for weight	16
Strings	Slow swell (0.25 s attack) with vibrato	48
Pluck	Short guitar-like notes	24
Bass	Plays an octave lower	33
Synth lead	Bright and buzzy, with delayed vibrato	81
Bell	Long metallic ring from inharmonic partials	14
Marimba	Wooden, quick decay	12
Flute	Soft, breathy, gentle vibrato	73

Settings > Instruments also has a **volume** and a **reverb** slider for each hand. The reverb is a small room (section 16) on a send: each hand's dry sound plus its share of the room.

Dynamics: strike speed sets loudness

Like a piano key, a sign struck quickly plays louder than one formed slowly. When a sign starts, the program looks back 0.15 s and takes the fastest finger movement (or hand movement, for a pump) and maps it to a velocity between 0.3 and 1.0 (section 13 has the formula). The velocity sets the synth's loudness and the MIDI note velocity, and a bar on the pad shows it. Press `i` to switch it off and play every sign at the same loudness. Hand-height Volume mode replaces it while it is on.

Custom mode: your own sound on every sign

Press `Tab`, open **Key mapping**, pick a hand and a sign, then choose a **Note** (click a key on the two-octave keyboard), a **Chord** (a root plus a type: maj, m, 7, maj7, m7, 6, m6, sus2, sus4, dim, aug, add9) or a **Drum**. Each choice plays a preview and switches to Custom mode. *Start from a built-in layout* copies Notes, Chords, Drums or Band into Custom as a starting point. A mapping has 8 sounds per hand; signs 5–8 need the Extended set.

Presets. *Save current as...* stores the whole mapping (both hands, 16 signs) under a name, as a JSON file in the `mappings` folder. Click a preset's name to select it, then **Load** it, **Export** it anywhere, or **Delete** it (click twice to confirm). **Import** reads a preset file someone sent you; invalid pads in a file are dropped rather than breaking the app.

Hand height: volume and pitch bend

Press `e` to cycle Off, Volume and Bend. In **Volume** mode, a hand near the top of the frame plays loud and near the bottom plays quiet; the height when you start a note sets how hard it is struck. In **Bend** mode, moving your hand up or down while holding a sign bends the pitch by up to two semitones, measured from where your hand was when the phrase started, so every note begins in tune. A meter next to your hand shows the current value.

Sustain, capture and blur

Key	What it does
space	Sustain on/off. While on, released notes keep ringing; turning it off stops them.
v	Start or stop a demo capture of the whole window plus the audio, saved in the `demos` folder.
b	Background blur on/off.

The looper, MIDI files and hand sessions have their own section (6).

Pinch style

Press g to switch to the original pinch style: touch your thumb to the index, middle, ring or pinky fingertip. It has four sounds per hand and supports playing two fingers at once, but pinches are harder for the camera to see because the thumb often hides the fingertip.

4. Calibrating your hands

Every hand bends differently, and cameras and lighting change what the tracker sees. A finger that one person holds straight at 40° may read 85° for someone else, close to the default 90° line, and that is the main reason a sign is misread. Calibration measures your hands and fits the thresholds to them, separately for each finger of each hand.

Running the wizard

1. Press `F2` (or Settings > Calibrate, and choose Basic or Extended). A strip appears above the pads; the camera and hand stage stay visible.
2. It shows a picture of each sign in turn: a fist first, then every sign of the set (5 steps for Basic, 9 for Extended).
3. **Get ready** (1.5 s): form the sign with both hands, each on its own side of the centre line.
4. **Hold** (2 s): keep the sign while moving the hand a little (turn and tilt it), so the measurement covers how you really play. A dot per finger turns green when that finger already reads correctly, and a frame counter shows how much was collected.
5. **Skip** passes over a sign you can't make; **Esc** cancels.
6. The results card appears. Choose **Save**, **Try again** or **Discard**.

Reading the results

- For each finger, a green bar shows the range of bend measured while it should have been **up** and a grey bar the range while it should have been **down**; two white lines mark the new thresholds.
- A warning replaces the numbers when a finger's up and down readings overlap (it was half-bent in some signs). Retry with clearer shapes, or skip the sign that causes it.
- **Signs read correctly** re-checks every collected frame with the new thresholds: green ≥ 90%, amber ≥ 70%, red below. A low score points at exactly the sign to practise or skip.
- Extended calibration also fits the thumb (in and out lines), which separates 4 fingers from open hand and fist from thumbs up.

Saved thresholds are stored per hand in `settings.json` and used from the next frame on. Settings > Calibrate shows them and can reset to the defaults (90° and 140°). Section 13 explains the fitting.

5. The rhythm game

How to play

1. Add a song (below). Press `n` to choose a song; its title appears at the top right. Press `j` to start.
2. Notes scroll from right to left in one lane per hand. Each block shows what it plays (for example `Am`) and the sign that plays it (`4 fingers`).
3. Make the sign as the block reaches the white line. The strip under the lanes shows this bar and the next ones, like a lead sheet.
4. Your song's recording (or a synthesized backing track) plays along. At the end you get a grade and your best score is saved in `scores.json`.

Ghost-hand hints

While a song plays, a see-through **ghost hand** on the hand stage shows each hand's next sign, drawn over your own hand (or on its side of the stage when the hand is out of view), with a label such as *Next: 2 fingers → C5 (in 0.8 s)*. A ring around its palm fills up as the note approaches and is full when the note is due. The ghost turns green as soon as you are making the right sign. Ghosts appear 3 seconds ahead, so you have time to read them and form the shape.

Judging and scoring

Judgement	Timing	Points
Perfect	within ±0.10 s of the note	100 × combo bonus
Good	within ±0.22 s	50 × combo bonus
Miss	no matching sign by 0.22 s after the note	combo resets to 0
Wrong sign	a different sign near a note	shown on screen, no penalty

$$combo\ bonus = min(4,\ 1 + \lfloor combo \div 8 \rfloor)$$

$$accuracy = (perfect + 0.5 \times good) \div notes\ judged \times 100\%$$

Grades: **S** ≥ 95%, **A** ≥ 85%, **B** ≥ 70%, **C** ≥ 55%, otherwise **D**.

> **Fair timing** Timing is judged from the camera frame where your sign *first appeared*, not the frame where it was confirmed, so the two-frame confirmation does not make you late. If you are still consistently late, your camera has extra delay: press `.` to shift the timing window (shown as offset) or start with `--game-offset 60`. `,` shifts it back.

Adding your own music

No songs come with the app: you bring the music you want to play. Press `a` (or Settings > Songs > **Add a music file**) and choose an audio file (WAV, MP3, FLAC, OGG, AIFF) or a MIDI file.

1. **Analysis** (a few seconds). The app finds the tempo, where bar 1 starts, the key, the chord on every beat, the melody on every half beat, and the kick and snare hits (section 20 explains how). A MIDI file skips the guessing: its notes and tempo are read exactly.
2. **Check the beat.** *Listen with clicks* plays 20 seconds of the music with a rim click on every beat, louder on bar 1. Clicks that drift away from the music mean the tempo is off: use ±0.1 or ±1 BPM, or ×2 / ÷2 if it found half or double speed. Steady clicks that sit early or late mean bar 1 is off: use ±0.01 s or ±1 beat.

3. **Or tap it.** While listening, tap `Space` on every beat, starting on the first beat of a bar. After 4 taps a straight line is fitted through your taps, which sets the tempo and bar start together.
4. **Choose a chart.** *Melody* (the tune note by note: the lower half of its notes on the left hand, the upper half on the right, like a keyboard), *Chords* (the chord changes, up to 8 per hand) or *Rhythm* (kick and snare on the left hand, hi-hat on the right, Basic signs). **Easy**, **Normal** or **Hard** set how often a new note comes. A preview shows the pads and the first bars.
5. **Rename** the song if you like, then **Create song**. The sheet is written to `songs/<title>.txt` and the music copied to `songs/audio/`. In the game the music starts early enough for the first notes to scroll in, and bar 1 of the sheet lands exactly on the beat the analysis found.

> **How accurate is it?** The tempo, bar start, key and chords are reliable on music with a steady beat. The melody is the hardest part: in a full mix the voice, solos and harmony all compete, so the melody chart is a starting point. Sheets are plain text, so open the file and fix any note; the game reloads it next time.

Writing your own song sheets

A song is a plain text file in the `songs` folder. Lines are `name: value`. A comment starts with `#` (hash followed by a space); a `#` inside a chord name such as `D#m` is part of the chord.

```
title: Four Chord Jam
bpm: 80
mode: Band            # Notes, Chords, Drums, Band or Song
scale: Major          # Major, Minor, Pentatonic or Blues
root: 60              # MIDI note of the key (60 = C4)
signs: Basic          # Basic (4 per hand) or Extended (8 per hand)
steps per beat: 1     # 2 = each token is an eighth note
count in: 4
metronome: on

play r: C - - - | G - - - | Am - - - | F - - - |
play l: ( Kick . Snare . ) x4
backing: C3 - - - | G2 - - - | A2 - - - | F2 - - - |
```

Token	Meaning
a sound	A note (`E4`), chord (`Am`, `D#m6/F#`) or drum (`Kick`, `Open_hat`).
-	Keep holding the previous sound for another step.
.	Rest.
|	Bar line, only for readability.
(...) x4	Repeat the bracketed part 4 times. Groups can be nested; brackets need spaces around them.

- `play:` lines are what you play. Each sound is matched to the hand and sign that plays it in the song's mode; `play r:` or `play l:` forces a hand. Lines with the same name join end to end.
- `backing:` lines play automatically. Use different names (`backing bass:`, `backing drums:`) for parts that play at the same time.
- `mode: Song` with `right:` and `left:` lines lets a song choose the sound of every sign (up to 8 per hand), for songs whose chords do not fit the built-in modes.
- Single notes use octaves 0–5 (`C4`, `F#2`). A name ending in 6, 7 or 9 is always a chord, so `E6` means E sixth, not a high E.
- `audio: audio/my_song.mp3` plays a recording with the song and `audio start: 1.286` says where bar 1 begins in it (seconds). Generated sheets use these with `count in: 0`, since the music leads you in. If the notes feel early or late against the music, change `audio start` by a few hundredths; if they drift apart over the song, the `bpm` is slightly off.

6. Recording: looper, MIDI files and hand sessions

The looper

`r` starts recording; `r` again sets the loop length and starts playback; after that `r` toggles overdub (adding a layer over what plays). `p` stops or restarts, `u` removes the last layer, `c` clears. Settings > Looper & MIDI has buttons for all of these.

Setting	What it does
Tempo (`- =`, ±1 / ±10)	Sets the metronome, count-in and quantise grid, and the tempo written into MIDI files. 40–240 BPM.
Metronome (`z`)	A click on every beat while recording and looping, louder on the first beat of a bar.
Count-in	The first recording starts after one bar of clicks, so you come in on the beat.
Quantise (`w`)	Off, 1/4, 1/8 or 1/16: each new note's start moves to the nearest grid step, and its release moves with it so the note keeps its length.
Whole bars	With the metronome or quantise on, the first take's length is rounded to whole bars, so the loop stays in time with the click.

MIDI files

`F7` starts recording everything you play live; `F7` again saves it as `recordings/performance-<date>-<time>.mid`. `F8` saves the loop (every layer, one cycle) as `recordings/loop-<date>-<time>.mid`. The files are standard MIDI (type 1): one track per channel (left hand, right hand, loop layers, drums on channel 10), with the General MIDI program of each instrument, note velocities from your strike speed, pitch bend and expression, at the looper's tempo. Drag them into any DAW or notation program. Section 17 describes the format.

Hand sessions and demo mode

`F5` records a **session**: the 21 tracked points of each hand, 30 times a second, written to `sessions/session-<date>-<time>.gmsession` (gzip-compressed JSON lines, about 20 KB per second). No camera image is ever stored. `F5` again saves it.

- **Debugging:** replay the moment a sign was misread with the `d` view on, change a setting or recalibrate, and replay exactly the same movement.
- **Tests:** a session is repeatable input; the same frames always give the same events.
- **Demo without a camera:** `--demo` replays the built-in demo (20 s, Band mode at 96 BPM: C G Am F on the right hand, kick and snare with finger taps on the left). `--replay FILE` replays any session. `F6` replays your latest session or stops a replay. Settings > Sessions lists every session with a Replay button.

During a replay the camera panel shows the file name and position instead of the picture; everything else (stage, pads, sound, MIDI, looper, game) works exactly as with live hands, because the replay feeds the same `process()` step the camera does.

7. MIDI and music software

With `--midi`, every event is also sent as MIDI, so the gestures can play any software instrument.

1. Install loopMIDI (a free virtual MIDI cable for Windows) and create a port.
2. Run `gesture_music.py --midi loopMIDI --mute`.
3. In your DAW (Ableton, FL Studio, Reaper...), enable loopMIDI as a MIDI input and arm a track for each channel you want.

Source	MIDI channel
Right hand, live	1
Left hand, live	2
Looper playback (right / left)	3 / 4
Rhythm game backing	5
Drums (General MIDI)	10

> **Tip** Windows' built-in Microsoft GS Wavetable Synth works for a quick test (`--midi "GS Wavetable" --mute`) but adds 100–200 ms of delay, so the program never picks it automatically.

8. Controls reference

Key	Action
m / M	Mode: Notes / Chords / Drums / Band / Custom / Song (next / previous)
x	Sign set: Basic (4 per hand) / Extended (8 per hand)
g	Gesture style: hand signs / pinch
s	Scale: Major, Minor, Pentatonic, Blues
[ ]	Octave down / up
t / T	Right hand: next / previous instrument
y / Y	Left hand: next / previous instrument
e	Hand height: Off / Volume / Bend
i	Strike speed sets loudness: on / off
space	Sustain on / off (taps the beat while listening to a song in the importer)
r	Record; then overdub on / off
p / u / c	Loop play-stop / undo the last layer / clear
z / w	Loop metronome on / off; quantise Off, 1/4, 1/8, 1/16
- =	Loop tempo −1 / +1 BPM
n / j	Rhythm game: next song / start or stop
a	Add a song from a music or MIDI file
, .	Rhythm game timing offset −10 / +10 ms
F2	Calibrate your hands
F5 / F6	Record a hand session / replay the latest (or stop the replay)
F7 / F8	Record a MIDI file / export the loop as a MIDI file
b	Background blur on / off
k / o	Hand stage colour / view angle
l	Skeleton on the camera view on / off
v	Demo capture start / stop
d	Debug view: live finger measurements against the thresholds
h	Swap left and right hands
Tab	Settings panel
F11	Full screen
Esc	Cancel calibration, close the settings panel or menu; quits if nothing is open
q	Quit
mouse	Click chips, pads and buttons; drag sliders; scroll long settings tabs

9. Troubleshooting and tuning

It feels slow or jumpy

- **Add light.** Webcams lengthen their exposure in dim rooms, dropping from 30 to 10–12 frames per second, which adds about 100 ms between frames and blurs moving fingers. The app shows a warning when the camera drops below 24 fps.
- **Keep both hands in view.** MediaPipe runs its slower whole-image hand search every frame until it has found both hands (about 35 ms instead of 10–16 ms).
- **Don't play live through the GS Wavetable synth** (100–200 ms of extra delay).
- `--confirm 1` makes new signs play one frame sooner, at the cost of letting single-frame glitches through.

A sign isn't recognised

Calibrate first (`F2`, section 4): it fixes most misreadings, and its score per sign tells you which shape is the problem. To look closer, press `d`. For each hand, a bar shows how bent each finger is against its up and down lines; green means up, grey down, amber in between (keeps the last state). Extra rows show the thumb's distance from the pinky knuckle, the tap fold angle and the pump speed. Record a session of the problem (`F5`) and replay it (`F6`) to watch the same movement again. The defaults below are used for any hand that is not calibrated:

Constant	Value	Meaning
`EXTENDED_MAX_CURL`	90°	A finger bending less than this in total counts as up.
`CURLED_MIN_CURL`	140°	A finger bending more than this counts as down.
`CONFIRM_FRAMES`	2	Frames a new sign must be seen before it plays.
`THUMB_OUT_MIN`	0.85	Thumb counts as out above this tip-to-pinky-knuckle distance (in hand lengths)...
`THUMB_STRAIGHT_MAX`	50°	...and only if it bends less than this.
`THUMB_IN_MAX / THUMB_BENT_MIN`	0.72 / 75°	Thumb counts as in below this distance or above this bend.
`TAP_DIP / TAP_RETURN`	40° / 18°	Fold past your straight position to arm a tap; straighten back within this to fire it.
`PUMP_SPEED / PUMP_REARM_SPEED`	3.0 / 0.8	Downward speed (hand lengths per second) that fires a pump / re-arms it.
`PRESS_RATIO / RELEASE_RATIO`	0.32 / 0.48	Pinch style: thumb-to-tip distance (hand lengths) that presses / releases.

Left and right are swapped

Press `h`. Hands are labelled by screen side, so this only happens if the view is not mirrored.

10. Architecture

The pipeline

Data flows in one direction, from camera to sound, and every musical action is expressed as an event:

```
webcam → camera.py → MediaPipe landmarks ─┐        session.py (record ← / replay →)
                                          ▼
                          assign L/R by screen side → calibration.py (while calibrating)
                                          │
                      gestures.py (sign, strike speed, hand height; calibrated thresholds)
                                          │ events
                                          ▼
     game.py (backing, music, judging) →  Router (gesture_music.py) → midi_file.py (F7)
                                  ┌───────┼──────────────────┐
                                  ▼       ▼                  ▼
                 sound_engine.py       midi_out.py          looper.py
                 (synth, mixer,        (DAW / synth)        (records, then replays
                  reverb, music)                             events to the Router)

        ui.py draws everything from the App's state; songmaker.py + audio_analysis.py
        turn a music file into a song sheet for game.py
```

Events: the instrument's vocabulary

Event	Meaning
`("note_on", voice_id, midi_note, velocity, group, instrument)`	Start a note. The voice id lets the matching note_off find it; group is the hand (or loop layer) for bend, volume and the mixer; instrument names the sound.
`("note_off", voice_id)`	Release that note (it fades out over the instrument's release time).
`("drum", name, velocity, hand)`	Play a one-shot drum sound, through that hand's mixer channel.
`("expr", group, bend_semitones, volume)`	Set the pitch bend and volume for every note in a group.
`("all_off", prefix)`	Release every voice whose id starts with prefix ("live", "loop", "game", "preview").
`("track", samples, start_time)`	Play a song's recording, starting exactly at start_time (a perf_counter time).
`("track_stop",)`	Stop the recording.

Because the synthesizer, MIDI output, looper and MIDI-file recorder all accept the same events, one design gives several features for free: a loop plays back over MIDI, a performance becomes a MIDI file by simply keeping the events with their times, the game's backing track uses the same synthesizer, and adding a new output means writing one more `send(event)` method.

```
class Router:
    def play(self, event):          # to the speakers and MIDI
        self.engine.send(event)
        if self.midi:
            self.midi.send(event)

    def live(self, event):          # from your hands: also looped and recorded
        self.play(event)
        if self.looper:
            self.looper.record(event)
        if self.performance is not None:          # F7: recording a MIDI file
            self.performance.append((time.perf_counter(), event))
```

Threads

Several jobs need their own timing, so they run on separate threads and talk through small, safe channels:

Thread	Job	How it communicates
Main	Hand tracking (or session replay), gesture logic, calibration, drawing the window, mouse and keyboard.	Calls router.live(event); reads frames from the camera thread.
Camera reader	Reads the webcam continuously and keeps only the newest frame.	A threading.Condition guards the latest frame and its sequence number.
Audio callback	Generates each 256-sample block of sound (PortAudio calls it about 172 times a second).	Pulls events from a lock-free collections.deque.
Looper timer	Every 2 ms, plays loop events and metronome clicks whose time has come.	Lock around the layer list; dispatches via router.play.
Game timer	Every 2 ms, plays scheduled backing, music start and count-in events.	Lock around the schedule; dispatches via router.play.
Importer	Analyses a music file in the background; a 2 ms timer plays the preview clicks.	Sets state and message fields the window reads; schedule under a lock.
Blur worker	Blurs the background of the current frame.	ThreadPoolExecutor future, collected before drawing.

Why threads help here Python's Global Interpreter Lock lets only one thread run Python code at a time, but OpenCV, MediaPipe and NumPy release it while their native code runs. That is why the blur can run alongside hand tracking and cost about 1 ms instead of 12 ms.

One frame, step by step

`App.step()` gets the next hands; `App.process()` turns them into sound; `UI.draw()` shows the result.

1. **Input.** Live: wait for a frame newer than the last one (`Camera.read`), mirror it, hand it to the blur worker, and run MediaPipe's hand landmarker (`detect_for_video`) with a strictly increasing timestamp. Replay: the `SessionPlayer` waits until the next recorded frame is due and returns its landmarks instead.
2. Label each detected hand L or R by screen side (`assign_hands`). If a session is being recorded, write the landmarks to it.
3. For each hand, update its `HandState` with that hand's thresholds: which sign or pinch is active, whether a tap or pump happened, how fast it was struck, and the hand height.
4. **While calibrating**, stop here: feed the hands' finger curls to the `Calibrator` and play nothing.
5. Send an `expr` event if bend or volume changed. For every released slot, release its notes; for every pressed slot, choose its velocity (hand height, strike speed or a default) and start its notes or drum, and tell the rhythm game about the press.
6. Release the notes of any hand that has been missing for more than 0.2 s.
7. Measure frame-to-sound latency; update the rhythm game (misses, end of song).
8. Collect the blurred frame and draw the window: top bar, camera view and keyboard, hand stage (with ghost hints), game lanes or calibration strip, pads, footer, and the settings panel or popups on top.
9. Save the window to the demo capture if recording, then handle mouse and keyboard events.

11. Module reference

Every file in the project, with its main classes and functions. Private helpers start with an underscore.

gesture_music.py — the App (997 lines)

State and one method per action, the per-frame step, hand sides, modes and chords, Custom mapping, sustain and the event router. It never draws: ui.py reads its state.

Key settings: MODES, SCALES, CHORD_DEGREES, DRUM_LAYOUT, SIDE_DEAD_ZONE = 0.08, DUPLICATE_DIST = 0.06, LOST_GRACE = 0.20 s, DEFAULT_VELOCITY = 0.85, VELOCITY_SOFT = 0.3, CHORD_VELOCITY = 0.7.

Function or class	What it does
`ensure_model()`	Downloads hand_landmarker.task on first run.
`build_chord(root, key, degree)`	The triad on a scale degree plus a bass note an octave below; returns notes and a name such as "Am".
`pad_sound(pad), slot_for(mode, hand, slot, ...)`	What a sign plays: ('notes', [midi...], label), ('drum', name, label) or ('none', ...).
`assign_hands(result, hands, swap)`	Labels detections L/R by screen side, merges duplicates, keeps a lone hand's label near the centre.
`make_lookup(song, nslots)`	For the game: maps sheet tokens ('Am', 'E4', 'Kick') to the hand and sign that play them.
`Router.play / Router.live`	Fans events out to the synth and MIDI; live events also go to the looper and the MIDI-file recording.
`App.step() / App.process(detections, stamp)`	Get the next hands (camera or replay) / turn them into events (or calibration samples).
`App.press(), release(), pad_down(), pad_up()`	Start or stop the sound for a sign (from a hand or a mouse click), with velocity and sustain.
`App.set_instrument, set_mix, toggle_dynamics`	Per-hand instrument, volume and reverb; strike-speed dynamics.
`App.loop_action, set_loop_bpm, toggle_metronome, cycle_quantise`	Looper controls.
`App.toggle_midi_recording(), export_loop()`	F7 and F8: write MIDI files with midi_file.export.
`App.start_calibration(), finish_calibration(save)`	Run the wizard; save the fitted thresholds into each HandState and settings.
`App.toggle_session_recording(), start_replay(), stop_replay()`	F5 / F6: hand sessions.
`App.add_song(), create_song(), select_song()`	The song importer and the song list.
`App.save_preset ... export_preset`	Key-mapping presets (settings.py does the file work).
`App.save()`	Writes every choice to settings.json.
`parse_args(), main()`	Command-line options; build the App and the UI and run.

ui.py — the window (1,984 lines)

Everything you see and click, drawn with pygame each frame from the App's state. Clickable areas are recorded as hit rectangles while drawing, so layout and input can never disagree.

Key settings: GAME_PPS = 240 px per second, TRAIL_SECONDS = 0.6, colours and fonts at the top of the file.

Function or class	What it does
`UI.draw()`	One frame: layout, top bar, camera, stage, game lanes or calibration strip, pads,

Function or class	What it does
	footer, settings, popups.
`handle_events(), on_click(), on_key()`	Mouse (down, up, drag, wheel) and keyboard, including text entry for names.
`draw_camera(), now_playing()`	The camera view with skeletons and labels; the five-octave keyboard.
`draw_stage(), project()`	The hand stage: world landmarks rotated by the view angle, depth-sorted bones, trails, glows.
`draw_ghosts(), pose_points()`	In-game hints: a synthetic hand in the next sign's pose, with a countdown ring.
`draw_pads(), draw_popup(), draw_footer()`	Pads with strike bars, the instrument menu, looper and recording status.
`draw_game(), draw_results()`	The scrolling lanes, sheet strip, count-in and results card.
`draw_settings() and tab_*()`	The settings panel: one method per tab, drawn into a scrolled, clipped area.
`draw_calibration(), draw_calibration_results()`	The wizard strip and the results card.
`button(), slider(), toggle_row(), row_buttons(), text_box()`	Small widgets.
`icon(), sign_icon()`	Vector icons and the little sign pictures (drawn as shapes, so no font is needed).

gestures.py — reading the hands (406 lines)

Turns landmarks into musical intent: which sign or pinch is active, repeats, strike speed and hand height.

Key settings: the defaults in section 9, plus STRIKE_WINDOW = 0.15 s, STRIKE_FINGER_FLOOR / FAST = 250 / 2000 °/s, STRIKE_HAND_FLOOR / FAST = 1 / 5 hand lengths per second, BEND_RANGE = 2 semitones.

Function or class	What it does
`sign_names(sign_set), classify_sign(...)`	Sign names; match the up/down pattern (plus the thumb where needed) to a slot, REST, or None.
`bend(a, b, c), finger_curl(...)`	Joint bend in degrees; the sum over a finger's three joints.
`OneEuroFilter`	Adaptive smoothing for the hand height (section 14).
`HandState(calibration)`	Everything known about one hand across frames, with its own thresholds.
`HandState.calibrate(cal)`	Use fitted thresholds (or None for the defaults); kept across reset().
`HandState.update_signs(...)`	Sign style: finger states with hysteresis, thumb, classification, confirmation, onset, repeats, strike speed.
`HandState._track_motion(), _strike()`	Keep 0.15 s of finger and hand positions; the fastest movement as a 0..1 strength.
`HandState._detect_repeat(...)`	Finger tap (relative fold) and hand pump (windowed speed).
`HandState.update(...)`	Pinch style: thumb-to-tip ratios, hysteresis, fast-pinch prediction.
`HandState.lost(), expression(mode)`	Release on losing the hand; current (bend, volume).

calibration.py — the calibration wizard (134 lines)

Steps through the signs, collects finger curls and thumb ratios per hand, fits thresholds and scores them.

Key settings: GET_READY = 1.5 s, HOLD = 2.0 s, MIN_FRAMES = 12, MIN_GAP = 20°.

Function or class	What it does
`steps_for(sign_set)`	Fist, then every sign of the set.
`Calibrator(sign_set, now)`	Phase ready → hold → next step; samples[hand][step].
`Calibrator.update(now, states)`	Advance the phases; while holding, store each seen hand's curls and thumb ratio.
`Calibrator.skip(now), progress(now), frames(hand)`	Skip a sign; the progress bar; frames collected.

Function or class	What it does
`_fit(up, down, ...)`	Two thresholds inside the gap between up and down readings (section 13).
`compute(steps, samples)`	Per hand: thresholds, a report per finger and the thumb, and scores per sign.
`_step_scores(...)`	How many of each sign's frames the new thresholds read correctly.

handmodel.py — a synthetic hand (83 lines)

Builds the 21 landmarks of a right or left hand making any sign, with optional partial curl and turn.

Key settings: Used by the ghost hints, the calibration pictures, the demo session generator and the tests.

Function or class	What it does
`pose(pattern, thumb, fold)`	21 (x, y, z) points in hand lengths; fold sets each finger's curl from 0 (straight) to 1.
`landmarks(pattern, thumb, hand, center, size, fold, yaw)`	Screen and world landmarks shaped like MediaPipe's, for tests and sessions.
`thumb_mode(pattern, thumb)`	Where the thumb goes for a sign: out, in, up or fist.

session.py — hand sessions (109 lines)

Records and replays tracked hands. A session is gzip-compressed JSON lines: a header, then one line per frame.

Function or class	What it does
`SessionRecorder.start(), add(stamp, detections), stop()`	Write frames (time from the first frame, hands with screen and world points).
`load(path), list_sessions()`	Read a session; the saved sessions, newest first.
`SessionPlayer(path, loop).read()`	Wait until the next frame is due and return it, looping at the end.

midi_file.py — MIDI files (103 lines)

Writes timed events as a standard MIDI file and reads MIDI files for the song importer.

Key settings: TICKS = 480 per beat; files go to recordings/.

Function or class	What it does
`export(timed_events, path, bpm, name)`	Type-1 file: tempo track plus one track per channel; notes, drums, program changes, pitch wheel, CC 11.
`read_notes(path)`	(bpm, [(start s, length s, pitch, velocity, channel)]) from any MIDI file.

camera.py — always the newest frame (60 lines)

Reads the webcam on its own thread so the main loop never processes a stale, buffered frame.

Function or class	What it does
`Camera(index, width, height)`	Opens the camera at 1280×720 with a 1-frame buffer and automatic exposure, and starts the reader thread.
`Camera.read(after_seq)`	Waits for a frame newer than after_seq; returns (seq, frame, arrival time).
`Camera.release()`	Stops the thread and closes the camera.

sound_engine.py — the real-time synthesizer (284 lines)

Generates audio block by block inside the sounddevice callback, so a note's pitch and volume can change while it plays.

Key settings: SAMPLE_RATE = 44,100 Hz, BLOCK = 256 samples (5.8 ms), MASTER_GAIN = 0.35, reverb send 0.15 per hand by default.

Function or class	What it does
`Instrument, INSTRUMENTS`	Partials, attack, release, vibrato, transpose, GM program and gain for the 10 instruments.
`hand_of(group)`	Which hand's mixer channel a note belongs to (live or looped).
`Reverb.process(x)`	Freeverb-style room: 4 damped comb filters and 2 all-pass filters, a whole block at a time.
`Voice.render(frames, bend, volume)`	Additive synthesis for one block with pitch bend, vibrato and the envelope.
`SoundEngine.send(event)`	Thread-safe: queues an event for the audio thread.
`SoundEngine._apply(event)`	Audio thread: voices, drums, bend and volume targets, the song's music track.
`SoundEngine._callback(...)`	Audio thread: per-hand mixes with volume and reverb send, soft clip, music, output and capture.

drums.py — the drum kit (103 lines)

Synthesizes 12 drum sounds with NumPy, or loads your WAV files instead.

Key settings: DRUMS (12 names), LABELS, GM_NOTES (General MIDI drum notes).

Function or class	What it does
`synth_kit()`	Builds all 12 sounds (sweeps, filtered noise, partials) and normalises them.
`load_wav(path), load_kit()`	Reads a 16-bit WAV, mixes to mono and resamples; samples/<name>.wav replaces a sound.

midi_out.py — live MIDI output (116 lines)

Translates events into MIDI messages for a DAW or hardware synth.

Function or class	What it does
`list_ports(), MidiOut(port_hint)`	MIDI outputs; open one (never the slow GS Wavetable synth unless named).
`channel_for(group)`	Right hand → 1, left → 2, loop → 3/4, game → 5 (shown 1-based).
`MidiOut.send(event)`	Note on/off with a program change when a channel's instrument changes, drums on channel 10, pitch wheel, CC 11.
`MidiOut.close()`	Releases held notes and resets the pitch wheels.

looper.py — record and loop (242 lines)

Records events with timestamps and replays them in a loop, with overdub layers, a metronome and quantise.

Key settings: QUANTISE = Off, 1/4, 1/8, 1/16.

Function or class	What it does
`Looper(dispatch, bpm, metronome, count_in, quantise)`	Starts the 2 ms playback thread.

Function or class	What it does
`toggle_record()`	Count-in → recording → close the loop (whole bars when in time) → overdub on/off.
`toggle_play(), undo(), clear()`	Stop or restart; remove the last layer; remove everything.
`record(event)`	Store a live event with its time in the loop.
`_commit_take(), _quantised(take)`	Close held notes; snap starts to the grid, keeping lengths.
`_click(now)`	Metronome clicks on the beat grid, accented on bar starts.
`export_events(cycles)`	The loop as timed events for a MIDI file.
`_as_loop_event(layer, event)`	Rename voices and groups so loop notes never clash with live ones (keeps the instrument).

game.py — the rhythm game (409 lines)

Song sheet parser, chord builder, timing and scoring, backing track and music start. Drawing lives in ui.py.

Key settings: PERFECT_WINDOW = 0.10 s, GOOD_WINDOW = 0.22 s, CHORD_QUALITIES.

Function or class	What it does
`parse_note(token), parse_chord(token)`	'F#2' → MIDI note; 'D#m6/F#' → [bass, chord tones...] for 22 chord types.
`expand_repeats(tokens), to_events(tokens)`	'(...) x3' groups; tokens → [start step, length, token].
`Song, load_song(path)`	The parsed sheet: tempo, mode, signs, tracks, pads, audio file and audio start.
`RhythmGame.refresh(select)`	Re-read the songs folder (after a song is created).
`RhythmGame.start(...)`	Builds the notes, count-in, metronome, backing and music schedule.
`RhythmGame.on_press(hand, slot, onset)`	Judges a press against the nearest matching note.
`RhythmGame.update(now)`	Marks misses and ends the song.
`accuracy, grade, _finish()`	Results, and saving the best score.

audio_analysis.py — listening to a recording (285 lines)

Finds tempo, beats, bar start, key, chords, melody, kick and snare in an audio file, with NumPy only (section 20).

Key settings: SR = 22,050 Hz, N_FFT = 2048, HOP = 512 (23 ms), FRAME_CENTER = 46 ms.

Function or class	What it does
`load(path, sr)`	Decode with soundfile, mix to mono, resample to 22,050 Hz.
`_spectrogram(y), _onsets(mag, freqs, lo, hi)`	Short-time Fourier transform; spectral flux in a frequency band.
`_tempo(env, fps)`	Autocorrelation of the onset envelope with a tempo preference, and the double-speed check.
`_track_beats(env, period)`	Dynamic-programming beat tracker.
`_chroma(), _key()`	Pitch-class energies; key by correlation with major and minor profiles.
`_chords(beat_chroma, key)`	24 triad templates per beat, smoothed by a Viterbi pass.
`_melody(mag, freqs, times, fps)`	Harmonic salience; the highest strong pitch per half beat.
`analyze(path, progress)`	All of the above, returning an Analysis.
`regrid(a)`	Recompute chords, melody, kick and snare after the tempo or bar start is corrected.

songmaker.py — from a music file to a song sheet (421 lines)

Turns an analysed file into sheets, and holds the logic behind the Songs tab.

Key settings: CHARTS, DIFFICULTIES, OUTPUT_LATENCY = 30 ms (tap compensation).

Function or class	What it does
`load_draft(path), from_audio(), from_midi()`	A Draft: title, tempo, bar start, key and the analysis or MIDI notes.
`Draft.per_beat()`	Chords per beat, melody / kick / snare per half beat on the current grid.
`make_sheet(draft, chart, difficulty)`	The sheet text for a Melody, Chords or Rhythm chart.
`create_song(draft, chart, difficulty)`	Write songs/<title>.txt and copy the music to songs/audio/.
`track_samples(path)`	Decoded 44.1 kHz music, cached for the game and the preview.
`Importer.open(path)`	Analyse in a background thread; state and message for the window.
`Importer.set_bpm(), shift_start(), tap(now)`	Corrections, and tap tempo by a straight-line fit.
`Importer.preview(), stop_preview(), create()`	Listen with clicks; create the song.

settings.py — remembering your choices (164 lines)

Loads and saves settings.json, ignoring unknown or invalid values; key-mapping presets.

Function or class	What it does
`load(), save(data)`	Defaults merged with valid saved values; save writes a temporary file and renames it.
`list_presets(), read_preset(path), write_preset(name, pads, path)`	Presets in mappings/; invalid pads dropped.
`load_preset(name), delete_preset(name)`	By name.

background.py, capture.py — blur and demo videos (67 and 72 lines)

Selfie segmentation plus a blurred background; recording the window and the audio.

Function or class	What it does
`BackgroundBlur.apply(frame, timestamp)`	Segments a 320×180 copy, smooths the mask, blurs the background and blends.
`DemoCapture.start(), add_frame(), stop()`	MP4 of the window at a steady 30 fps plus the synth's audio as WAV, merged with ffmpeg.

tools/ and tests/

Helpers that are not part of the running app.

Function or class	What it does
`tools/make_demo_session.py`	Generates sessions/demo.gmsession from the synthetic hand.
`tests/test_gesture_music.py`	14 unit tests (section 2).
`docs/build_handbook.py, docs/build_visual_handbook.py`	Build this handbook and its illustrated Visual Edition.

12. Computer vision: finding hands

MediaPipe Hand Landmarker

MediaPipe's hand landmarker is a two-stage neural network pipeline. A **palm detector** looks over the whole image for hands; then a **landmark model** looks closely at each hand's region and predicts 21 keypoints. In video mode it reuses the previous frame's hand positions to skip the palm search, which is why tracking is faster (10–16 ms) once both hands are found than while it is still searching (about 35 ms).

For each point it gives two sets of coordinates:

- **Normalised image coordinates** (x, y from 0 to 1 across the image). Used for drawing and for screen position (which side, how high).
- **World coordinates** in metres, centred on the hand. These describe the hand's real 3D shape, so they do not change when the hand moves closer, further away, or turns. Distances and angles for signs and pinches use these.

Index	Landmark	Index	Landmark
0	Wrist	9–12	Middle: knuckle (MCP), PIP, DIP, tip
1–4	Thumb: CMC, MCP, IP, tip	13–16	Ring: MCP, PIP, DIP, tip
5–8	Index: knuckle (MCP), PIP, DIP, tip	17–20	Pinky: MCP, PIP, DIP, tip

MCP is the knuckle where the finger meets the palm; PIP and DIP are the middle and end joints. The video mode requires each frame's timestamp to be strictly larger than the last, so the program uses the camera arrival time in milliseconds and bumps it by 1 if two frames share a millisecond.

Mirroring and handedness

The frame is flipped horizontally before tracking so the screen behaves like a mirror. MediaPipe also guesses whether each hand is left or right, but that guess flips easily from frame to frame, which used to stop and restart notes. The program ignores it and uses screen side instead: a rule you can see and predict, with a dead zone of ±0.08 around the centre and duplicate detections closer than 0.06 merged.

13. Gesture geometry

Angles from vectors

To know how bent a joint is, take the two bones meeting at it as vectors and measure the angle between them with the dot product:

$$\cos\theta = (u \cdot v) \div (|u|\,|v|) \qquad bend = 180° - \theta$$

Here u points from the joint back toward the previous joint and v forward toward the next. A straight finger has θ ≈ 180°, so its bend is about 0°. Adding the bends at the knuckle, middle and end joints gives the finger's **curl**:

$$curl = bend(wrist, MCP, PIP) + bend(MCP, PIP, DIP) + bend(PIP, DIP, tip)$$

Because angles do not depend on size, position or rotation, the same sign gives the same numbers whether your hand is near, far, tilted or turned. A straight finger measures roughly 30–60° (the knuckle sits at an angle to the wrist, plus tracking noise); a fist about 200–220°.

Hysteresis: two thresholds instead of one

With one threshold, a finger measured right at the boundary would flip between up and down every frame because of noise. The program uses two: below 90° the finger is **up**, above 140° it is **down**, and in between it **keeps its previous state** (these are the defaults; calibration fits a pair for every finger, below). This idea, called hysteresis, appears throughout the project: pinch press and release ratios (0.32 / 0.48), thumb in and out, tap arm and fire, pump fire and re-arm.

Classifying and confirming a sign

- The four finger states form a pattern such as (up, up, down, down). `classify_sign` looks it up in the sign set; the thumb is checked only where the set needs it.
- A pattern that is not a sign (for example only the middle finger up, or an in-between shape while changing sign) is ignored: nothing changes.
- A new sign must be seen in **2 consecutive frames** before it plays (debouncing). A one-frame glitch therefore never makes a sound.
- The **onset** recorded for the game is the time of the first of those frames, so confirmation does not count against your timing.

The thumb

The thumb is out when its tip is far from the pinky knuckle **and** the thumb is straight; it is in when the tip is close **or** the thumb is bent. Distances are divided by the hand size (wrist to middle knuckle), so they are measured in hand lengths. Typical values: tucked across the palm 0.5, wrapped over a fist 0.65, thumbs up 0.9, spread open 1.3.

$$thumb\ ratio = |thumb\ tip - pinky\ knuckle| \div |wrist - middle\ knuckle|$$

Repeats: relative measurement and windowed speed

Tap. The average curl of the raised fingers is smoothed (each new reading counts 50%) and compared with a **baseline**: the straightest the fingers have been during this sign. The baseline follows the fingers down

immediately but drifts up only slowly, so it tracks your own natural finger position without needing calibration. Folding 40° past the baseline arms a tap; coming back within 18° fires it.

Pump. The hand's downward speed is measured over the last 0.1 s rather than between two frames, and in hand lengths per second so distance from the camera does not matter:

$$speed = (y_now - y_0.1\ s\ ago) \div average\ hand\ size \div elapsed\ time$$

Above 3.0 hand lengths per second it fires; it re-arms when the speed drops below 0.8. Measuring over a window averages out tracking jitter: with single-frame speed, holding still produced dozens of false pumps in tests; with the window, none.

Calibration: fitting the thresholds to your hands

The defaults (90° and 140°) are a compromise for everyone. During calibration the program knows, for every frame, which sign you were asked to make, so it knows whether each finger *should* be up or down. For each finger it collects two sets of curl readings, **U** (should be up) and **D** (should be down), from all the steps. Both sets have outliers (a frame where tracking slipped), so it uses percentiles rather than the extremes:

$$u = 95th\ percentile\ of\ U \qquad d = 5th\ percentile\ of\ D \qquad gap = d - u$$

If the gap is at least 20°, the two thresholds go 30% of the way into the gap from each side, leaving the middle 40% as the keep-last-state zone (hysteresis):

$$up\ below = u + 0.3 \cdot gap \qquad down\ above = d - 0.3 \cdot gap$$

If the readings overlap (gap under 20°), the finger was not clearly up or down in some sign. The thresholds are then centred on the midpoint, 10° apart, and the finger is flagged so you can retry. The same fit, on the thumb's tip-to-pinky-knuckle ratio, gives the thumb's in and out lines (Extended calibration only, since only those signs say where the thumb should be). Finally every collected frame is classified again with the new thresholds, and the share read correctly is the score per sign.

Example: an index finger measured 25–60° when up and 175–215° when down gives u = 60, d = 175, gap = 115, so it counts as up below 94.5° and down above 140.5°. A stiffer ring finger measured 50–110° up and 150–200° down gives up below 122° and down above 138°, where the default 90° would have read it as half-bent in many frames.

Strike speed: how loud

Each frame the hand state keeps the finger curls and the hand's height, with their times, for the last 0.15 s. When a new sign is confirmed, it finds the fastest change in that window:

$$finger\ speed = max\ over\ fingers\ and\ frames\ of\ |\Delta curl| \div \Delta t\ \ (degrees\ per\ second)$$

$$hand\ speed = |\Delta y| \div hand\ size \div \Delta t\ \ (hand\ lengths\ per\ second)$$

$$strength = clip(max((finger\ speed - 250) \div 2000, (hand\ speed - 1) \div 5), 0, 1)$$

$$velocity = 0.3 + 0.7 \times strength$$

The floors (250 °/s and 1 hand length per second) ignore tracking jitter, so a sign formed slowly or held reads as soft; opening the fingers within about two frames (67 ms) reaches full strength. Measured on the synthetic hand: a sign formed in 67 ms gives strength 1.0, in 133 ms 0.70, in 267 ms 0.41 and in 467 ms 0.21. Pinch style uses the speed of the thumb-to-fingertip distance instead.

Pinch style

$$ratio = |thumb\ tip - fingertip| \div |wrist - middle\ knuckle|$$

A pinch presses below 0.32 and releases above 0.48. To hide camera delay, a finger closing quickly (faster than 1 hand length per second) is predicted forward by 40 ms using its current speed (linear extrapolation), and only the closest finger may start a pinch in each frame, which stops a neighbouring finger from triggering by accident.

14. Signal smoothing and filtering

Exponential smoothing

The simplest smoother mixes each new reading with the previous smoothed value:

$$\hat{x}_n = \hat{x}_{n-1} + \alpha\,(x_n - \hat{x}_{n-1})$$

A small α is steady but lags behind; a large α follows quickly but passes noise through. The tap detector uses α = 0.5; the background mask uses 0.5 between frames.

The One Euro filter

Hand height needs to be steady when you hold still and responsive when you move. The One Euro filter (Casiez, Roussel and Vogel, 2012) solves this by changing α with speed. It first estimates the speed dx̂, then sets a cutoff frequency that rises with speed:

$$f_c = f_min + \beta \cdot |d\hat{x}| \qquad \tau = 1 \div (2\pi\, f_c) \qquad \alpha = 1 \div (1 + \tau \div \Delta t)$$

Slow movement gives a low cutoff (heavy smoothing, no jitter); fast movement gives a high cutoff (almost no lag). The project uses f_min = 1.0 Hz, β = 6.0 and a 1.0 Hz cutoff for the speed estimate.

15. Latency: from movement to sound

Latency is the time from moving your hand to hearing the result. It is the sum of several stages:

Stage	Typical time	What the project does about it
Camera exposure and frame interval	33 ms at 30 fps; 83–100 ms at 10–12 fps in dim light	Warns when the camera drops below 24 fps; asks for auto exposure.
Waiting in the camera buffer	up to 1–2 frames	The reader thread keeps only the newest frame (measured 0.4 ms old when picked up).
Hand tracking	10–16 ms (35 ms while searching)	Keep both hands in view.
Sign confirmation	1 extra frame	Game timing uses the first frame; `--confirm 1` removes it.
Background blur	~1 ms extra	Runs in parallel with tracking (12 ms if run afterwards).
Audio block	5.8 ms	256-sample blocks (was 11.6 ms).
Audio output	23 ms measured	Uses WASAPI instead of Windows' default MME driver.

The on-screen line `camera fps | tracking ms | frame-to-sound ms` shows the live values; frame-to-sound is measured from the frame's arrival to the moment its events are sent.

16. Sound synthesis

Digital audio basics

Sound is stored as a list of numbers (samples) describing the speaker position over time. At 44,100 samples per second, frequencies up to 22 kHz can be represented, covering human hearing. The sound card asks for audio in **blocks**; sounddevice calls the program's callback for each 256-sample block (5.8 ms), and the callback must fill it in time or the audio glitches. Rendering six voices takes about 0.43 ms, around 7% of the budget.

From note number to frequency

Western tuning divides the octave (a doubling of frequency) into 12 equal semitones, with A4 = 440 Hz as the reference and MIDI note 69:

$$f = 440 \times 2\text{^}((n - 69) \div 12)$$ e.g. n = 60 (C4) → 261.6 Hz

Additive synthesis and timbre

Every musical tone is made of a fundamental frequency plus **partials** (overtones) at multiples of it. The mix of partials and how quickly each fades gives an instrument its character, or timbre. The synthesizer adds sine waves:

$$s(t) = \Sigma_i\, a_i \cdot e\text{^}(-d_i\, t) \cdot \sin(r_i \cdot \varphi(t))$$

where r_i is the partial's frequency ratio, a_i its amplitude (normalised so they sum to 1) and d_i its decay rate per second:

Instrument	Partials (ratio, amplitude, decay)	Envelope and extras
Piano	(1, 1.0, 2.5) (2, 0.5, 3.5) (3, 0.25, 4.5) (4, 0.12, 5.5) (5, 0.06, 6.5)	5 ms attack, 0.25 s release; upper partials die first, like a struck string.
Electric piano	(1, 1, 1.4) (2, 0.45, 2.6) (3, 0.12, 4) (4, 0.06, 6) (7, 0.05, 9)	The 7th partial gives the bell-like tine; slight vibrato.
Organ	(0.5, 0.4, 0) (1, 1.0, 0) (2, 0.6, 0) (3, 0.3, 0) (4, 0.2, 0)	No decay; a sub-octave (ratio 0.5) adds weight; 0.08 s release.
Strings	harmonics 1–10 at 1/n (sawtooth-like), no decay	0.25 s attack (bowed swell), 0.6 s release, vibrato 5.5 Hz.
Pluck	harmonics 1–8, amplitude 1/n, decay 3n	2 ms attack; partials vanish quickly, like a plucked string.
Bass	(1, 1, 1) (2, 0.5, 2) (3, 0.25, 3.5) (4, 0.1, 5)	Transposed down an octave.
Synth lead	harmonics 1–12 at 1/n, slowly decaying	Bright and buzzy; vibrato after 0.2 s.
Bell	(1, 1, 1.2) (2, 0.6, 1.6) (2.76, 0.5, 2.2) (5.4, 0.3, 3.5) (8.9, 0.2, 5)	Inharmonic ratios (2.76, 5.4, 8.9) make the metallic ring; 1 s release.
Marimba	(1, 1, 4) (3.9, 0.35, 9) (9.2, 0.12, 16)	A wooden bar's partials (near 4× and 9×), all decaying fast.
Flute	(1, 1, 0) (2, 0.25, 0) (3, 0.08, 0)	Nearly a pure tone; 80 ms attack, gentle vibrato.

Envelopes and vibrato

Each note's loudness follows an envelope: a linear **attack** (2–250 ms depending on the instrument; starting a sine wave instantly at full volume makes a click), the instrument's own **decay** per partial, and a linear **release** after you let go. A voice is removed once its release has finished. **Vibrato** is a slow sine wobble of the pitch, added to the pitch bend after a short delay:

$$\text{pitch offset} = \text{depth} \cdot \sin(2\pi \cdot \text{rate} \cdot (t - \text{delay}))$$ e.g. 5.5 Hz, ±0.12 semitones

A note's velocity scales its amplitude, so the strike speed (section 13) is heard directly.

Pitch bend with a phase accumulator

To bend a note while it plays, each voice keeps its own **phase** ϕ and advances it every sample by an amount that depends on the current frequency:

$$\varphi[k] = \varphi[k-1] + 2\pi \cdot f \cdot 2^{(bend[k] \div 12)} \div 44{,}100$$

Because the phase is accumulated rather than computed from time, the frequency can change smoothly without a jump in the waveform. Bend and volume are also ramped linearly across each block instead of jumping at block boundaries, which would cause a buzzing artefact called zipper noise. The phase is wrapped at 4π (not 2π) so the organ's 0.5-ratio sub-octave stays continuous. Measured: A4 rendered at 441 Hz moved to 495 Hz at +2 semitones (target 493.9 Hz, within the measurement's resolution).

Mixing, reverb and soft clipping

Voices and drums are summed into three buses: the left hand, the right hand (live and looped notes alike) and everything else (game backing, previews, clicks). Each hand bus is scaled by its **volume** slider, and a share of it, its **reverb send**, goes into one shared reverb:

$$dry = g_L \cdot L + g_R \cdot R + other \qquad wet = Reverb(s_L \cdot g_L \cdot L + s_R \cdot g_R \cdot R + 0.1 \cdot other)$$

$$out = \tanh(0.35 \times (dry + 1.4 \cdot wet))$$

tanh is almost linear for small values but rounds off peaks smoothly instead of clipping them harshly when many notes play at once. A song's recording is added after the soft clip (it is already mixed) at 0.7 of its level, then the output is limited to ±1.

The reverb

A room's reverberation is thousands of echoes arriving ever later and softer. The Freeverb design imitates it cheaply with two kinds of filter:

- **Comb filters** (4 in parallel, delays of 1116, 1188, 1277 and 1356 samples, 25–31 ms): each feeds its output back into itself after its delay, `y[n] = x[n] + 0.8 · lowpass(y[n − D])`, making a train of decaying echoes. The low-pass (damping 0.35) makes the echoes duller as they repeat, as walls absorb high frequencies. Delays with no common factor keep the echo trains from lining up into a metallic ring.
- **All-pass filters** (2 in series, 556 and 441 samples): `y[n] = −g·x[n] + x[n − D] + g·y[n − D]` with g = 0.5. They pass every frequency at the same level but smear the echoes in time, turning the comb's distinct repeats into a smooth wash.

Every delay is longer than one 256-sample audio block, so a whole block's output depends only on samples from earlier blocks. That lets each filter process a block with a few NumPy array operations instead of a Python loop per sample, which would be far too slow for real time.

Drum synthesis

Drum	Recipe
Kick	Sine sweeping from 160 Hz down to 45 Hz, decaying in about 0.5 s, plus a short noise click.
Snare	Bright noise plus a 185 Hz tone, both decaying fast.
Hat / open hat	Bright noise decaying very fast (closed) or slowly (open).
Clap	Three noise bursts 11 ms apart, then a short tail.

Drum	Recipe
Toms	Sine sweeps 140 → 90 Hz (low) and 230 → 150 Hz (high).
Crash	1.8 s of bright noise.
Ride	Five inharmonic partials (523, 811, 1325, 2010, 3150 Hz) over a soft noise wash.
Rim	A 1700 Hz click (also the game's metronome).
Cowbell	Two square waves at 540 and 800 Hz.
Shaker	Noise with a 10 ms fade-in and a fast decay.

"Bright noise" is white noise passed through a first-difference filter, y[n] = x[n+1] − x[n], a very simple high-pass filter that removes low frequencies. A falling sine sweep is how real kick drums sound: the drum skin's pitch drops as its vibration dies down.

17. MIDI

MIDI is a standard language for musical instruments. It sends small messages, not sound:

Message	Data	Used for
Note on / note off	channel, note 0–127, velocity 1–127	Starting and stopping notes.
Pitch wheel	channel, −8192 to +8191	Bend: ±2 semitones maps to the full wheel range.
Control change 11	channel, 0–127	Expression: hand-height volume.
Program change	channel, program 0–127	Selecting the sound: sent whenever a channel's instrument changes (General MIDI numbers, section 3).
Set tempo (meta)	microseconds per beat	MIDI files only: the tempo, 60,000,000 ÷ BPM.

$$wheel = round(clip(bend \div 2, -1, 1) \times 8191) \qquad velocity = max(1, round(v \times 127))$$

Pitch bend in MIDI applies to a whole channel, so each hand gets its own channel; otherwise bending one hand would bend the other. Drums use channel 10, where General MIDI defines a fixed note for each drum: kick 36, rim 37, snare 38, clap 39, closed hat 42, low tom 45, open hat 46, crash 49, high tom 50, ride 51, cowbell 56, maracas/shaker 70.

Standard MIDI files

A MIDI file stores the same messages with times. Times are not in seconds but in **ticks**, fractions of a beat (480 per beat here), so a DAW can change the tempo without moving notes off the grid. Each message carries a *delta time*: the ticks since the previous message in its track.

$$tick = round(seconds \times BPM \div 60 \times 480) \qquad delta = tick - previous\ tick$$

`midi_file.export` takes the router's timed events, starts the clock at the first one, converts every time to ticks at the looper's tempo, and writes a type-1 file (several tracks played together): a tempo track, then one track per channel holding its program changes, notes, pitch wheel and CC 11. Messages at the same tick are ordered so a note's off comes before the next note's on. Notes still held at the end are closed. `read_notes` does the reverse for the song importer, following tempo changes to turn ticks back into seconds.

18. Music theory used in the project

Notes, semitones and octaves

There are 12 note names per octave: C, C#, D, D#, E, F, F#, G, G#, A, A#, B. A sharp (#) raises a note by one semitone; a flat (b) lowers it, so C# and Db are the same key (enharmonic names). In MIDI, middle C is note 60 and is called C4; the octave number is $\lfloor n \div 12 \rfloor - 1$.

Scales

A scale is a set of notes chosen from the 12, written as semitone steps from the root:

Scale	Semitones from the root (8 notes, one per sign)	In C
Major	0 2 4 5 7 9 11 12	C D E F G A B C
Natural minor	0 2 3 5 7 8 10 12	C D Eb F G Ab Bb C
Major pentatonic	0 2 4 7 9 12 14 16	C D E G A C D E
Blues	0 3 5 6 7 10 12 15	C Eb F F# G Bb C Eb

Intervals and chords

An interval is the distance between two notes. A chord stacks intervals on a root note:

Interval	Semitones	Interval	Semitones
Minor third	3	Major sixth	9
Major third	4	Minor seventh	10
Perfect fourth	5	Major seventh	11
Tritone (diminished fifth)	6	Octave	12
Perfect fifth	7	Ninth	14

Chord type	Written	Semitones	Example (root C)
Major	C	0 4 7	C E G
Minor	Cm	0 3 7	C Eb G
Diminished / augmented	Cdim / Caug	0 3 6 / 0 4 8	C Eb Gb / C E G#
Sixth / minor sixth	C6 / Cm6	0 4 7 9 / 0 3 7 9	C E G A / C Eb G A
Dominant seventh	C7	0 4 7 10	C E G Bb
Major / minor seventh	Cmaj7 / Cm7	0 4 7 11 / 0 3 7 10	C E G B / C Eb G Bb
Suspended	Csus2 / Csus4	0 2 7 / 0 5 7	C D G / C F G
Seventh suspended	C7sus	0 5 7 10	C F G Bb
Ninth chords	C9 / Cm9 / Cmaj9	add 14 to the seventh chord	C E G Bb D ...
Slash chord	C/E	chord over a different bass note	C E G with E in the bass

Chords in a key (Roman numerals)

Building a triad on each note of a major scale, using only notes of that scale, gives seven chords. Roman numerals name them by scale position; capitals are major, lower case minor, ° diminished:

Numeral	I	ii	iii	IV	V	vi	vii°
In C major	C	Dm	Em	F	G	Am	Bdim
In Ab major	Ab	Bbm	Cm	Db	Eb	Fm	Gdim

Basic Chords mode uses I, IV, V and vi, the four chords behind a huge number of pop songs; Extended uses all seven plus I an octave up. `build_chord` makes these by taking every other note of the key (degree, degree + 2, degree + 4) and adding the root an octave below as a bass note.

Voicing

A voicing is the choice of which octave each chord note goes in. `parse_chord` keeps every chord in a similar register so changes sound smooth: the root is placed between G3 and F#4, the other notes stacked above it, and the bass (the slash note if there is one) between E2 and D#3. For example `D#m6/F#` becomes F#2 under D#4 F#4 A#4 C5.

Tempo, beats and bars

$$\textit{beat length} = 60 \div \textit{BPM seconds} \qquad \textit{bar} = 4 \textit{ beats}$$

At 120 BPM a beat lasts 0.5 s and a bar 2 s. A sheet's `steps per beat` sets the grid: 1 means each token is a beat (a quarter note), 2 an eighth note. Tempo is not the same as how busy the music is: a song at 97 BPM with notes on every eighth sounds as fast as one at 194 BPM with notes on every beat. That ambiguity is why the song importer offers ×2 and ÷2 (section 20).

Notes, chords and arpeggios

A **note** is one pitch. A **chord** is several pitches together (C major: C, E, G). An **arpeggio** (broken chord) plays a chord's notes one after another, a common piano accompaniment. A hand sign is only a control: what it plays depends on the mode or the song sheet. A melody chart gives each sign a single note, with the lower notes on the left hand and the higher ones on the right, like the two halves of a keyboard; a chords chart gives each sign a whole chord.

Keys and how to recognise one

Music in a key uses mostly the notes of its scale, and some notes more than others: the root and the fifth most, then the third. Krumhansl and Kessler measured how well each of the 12 notes "fits" a key by asking listeners; the result is a **key profile**, 12 numbers per key type (major: 6.35 for the root, 5.19 for the fifth, 4.38 for the third ... down to 2.23 for the note a semitone above the root). The importer finds a song's key by comparing how much of each note the song contains with the profile shifted to each of the 12 roots, for major and minor, and picking the best match (section 20).

19. Timing: the looper and the rhythm game

Recording events, not audio

The looper stores events with their time inside the loop instead of recording sound. That keeps loops perfectly clean, lets them play over MIDI, and lets them keep their notes when you change scale or sound. Loop position is time modulo loop length:

$$position = (now - loop\ start) \bmod loop\ length$$

Every 2 ms the playback thread sends every event whose time lies between the previous position and the current one. When the position wraps past the end, it sends the events from the old position to the end and then from the start to the new position. In a test, a 0.52 s loop replayed its kick at 0.53, 1.05 and 1.57 s.

Metronome, count-in and whole bars

The looper's tempo defines a beat grid. The metronome sends a rim click each time the clock passes a grid line (louder on every fourth, the bar start), from the same 2 ms thread that plays the loop. With the count-in on, pressing record sets the loop's start one bar in the future and clicks that bar, so your first note lands on beat 1. When the first take is closed with the metronome or quantise on, its length is rounded to whole bars:

$$loop\ length = \max(1, \mathrm{round}(take\ length \div bar)) \times bar \qquad bar = 4 \times 60 \div BPM$$

Quantise

Quantising moves each note's start to the nearest grid line, with the grid step g = 1, ½ or ¼ beat for 1/4, 1/8 and 1/16:

$$snapped = \mathrm{round}(t \div g) \times g$$

The note's release is moved by the same amount, so its length is unchanged; drums snap the same way. For example at 120 BPM with 1/8 (g = 0.25 s), a note played at 0.27 s and released at 0.61 s becomes 0.25 s to 0.59 s. Quantise only changes new takes; switching it off never moves notes already recorded.

Scheduling the game

When a song starts, every backing note, drum, count-in click and the start of the song's recording is converted to an absolute time and sorted into a schedule. A 2 ms timer thread plays each event when its time arrives, independent of the camera's frame rate; measured clicks landed within 6 ms of the beat.

$$note\ time = start\ step \times 60 \div BPM \div steps\ per\ beat$$

With a recording, bar 1 of the sheet must line up with the moment `audio start` seconds into the music. The music is scheduled to begin early enough for the first notes to scroll in (at least 2 s of lead), and the sheet's clock starts `audio start` seconds after the music does:

$$music\ at = \max(now + 0.3, now + 2 - audio\ start) \qquad song\ start = music\ at + audio\ start$$

The audio engine starts the recording at the exact sample that corresponds to *music at*, so the music and the scrolling notes share one clock.

Judging a press

$$t = onset - offset - song\ start$$

The game looks for the nearest unjudged note for that hand within ±0.22 s. If it needs the sign you made, it is Perfect (|t − note time| ≤ 0.10 s) or Good; if a different sign was needed, the screen shows WRONG SIGN. Notes nobody played by 0.22 s after their time become misses.

20. Music analysis: from a recording to a song sheet

A recording is just a list of samples; nothing in it says where the beats are or which notes play. `audio_analysis.py` recovers them with classic signal processing in NumPy, no machine-learning model needed. The steps below run in about a third of a second for a three-minute song.

1. A picture of the sound: the spectrogram

The file is decoded, mixed to mono and resampled to 22,050 Hz. It is cut into overlapping windows of 2048 samples (93 ms), one every 512 samples (23 ms), each multiplied by a Hann window (a smooth bump that avoids false high frequencies at the cut edges). A Fourier transform of each window gives the strength of every frequency at that moment:

$$X[m, k] = \Sigma_n \, x[n + 512m] \cdot w[n] \cdot e^{-2\pi i \, kn \div 2048}$$

$$\text{frequency of bin } k = k \times 22{,}050 \div 2048 \approx 10.8 \text{ Hz} \times k$$

The magnitudes |X| over time form the spectrogram: 43 columns per second, 1025 frequency rows.

2. When does something start? Spectral flux

A drum hit or a new note makes some frequencies suddenly louder. **Spectral flux** adds up those increases between consecutive windows, ignoring decreases (half-wave rectification), on a compressed (logarithmic) scale so quiet instruments still count:

$$flux[m] = \Sigma_k \, max(0, \, log(1 + 100|X[m,k]|) - log(1 + 100|X[m-1,k]|))$$

Subtracting a moving average (16 windows) and clipping at zero leaves sharp peaks at onsets: the **onset envelope**. Restricting the sum to 30–150 Hz gives a bass-drum envelope, and 1.5–5 kHz a snare and hi-hat envelope. A window registers an onset when the sound reaches its middle, so every time read from the envelope is corrected by half a window (46 ms); without it, bar 1 came out 60 ms early.

3. The tempo: autocorrelation

Music repeats at the beat period, so the onset envelope looks most like itself when shifted by one beat. **Autocorrelation** measures that similarity for every shift (lag) τ between 50 and 220 BPM:

$$r(\tau) = \Sigma_m \, env[m] \cdot env[m + \tau] \qquad BPM = 60 \times 43.07 \div \tau$$

Shifts of two or three beats also match well, so the scores are weighted by a gentle preference for common tempos (a log-normal curve centred at 115 BPM, one octave wide), and the best lag is refined between neighbouring lags with a parabola. One more rule fixes a common mistake: in a plain rock beat, the kick on beats 1 and 3 repeats every two beats and can outweigh the beat itself, so the tempo comes out half as fast. If the result is under 90 BPM and the double tempo also shows a clear pulse (at least 30% as strong), the double is used. The importer's ×2 and ÷2 buttons fix any remaining octave error.

4. The beats: dynamic programming

Knowing the period is not enough: real beats drift slightly and must sit on actual onsets. The beat tracker (Ellis, 2007) scores every possible beat time t as the onset strength there plus the best score of a previous beat, minus a penalty for a gap that differs from the period P:

$$score(t) = env(t) + max \text{ over } P/2 \le \Delta \le 2P \text{ of } [\, score(t - \Delta) - 100 \cdot (log(\Delta \div P))^2 \,]$$

This is **dynamic programming**: each score reuses the best scores already computed, so the best chain of beats through the whole song is found in one pass and read back from the end. A straight line fitted through the beat times (least squares) then gives one steady tempo and a grid without wobble, since the game needs a fixed BPM.

5. Bar 1: the downbeat

Bars usually begin with a strong bass note or kick. For each of the four positions a beat can have in a bar, the bass envelope is summed over every fourth beat; the position with the most bass energy is beat 1. The earliest such beat in the music becomes `audio start`.

6. Which notes: chroma, key and chords

Every spectrogram bin between 55 Hz and 2 kHz is assigned to its nearest note name, ignoring the octave, and the energy is summed into 12 values per window: the **chroma** vector (how much C, C#, D ... is sounding). The key comes from correlating the song's total chroma with the 24 key profiles (section 18) and keeping the best.

For chords, the chroma is averaged over each beat and compared with 24 templates, a major and a minor triad on each of the 12 roots (1 on the chord's three notes, 0 elsewhere), by cosine similarity, plus a small bonus (0.04) for chords that belong to the key. Picking the best template per beat would flicker whenever a melody note passes through, so a **Viterbi** pass chooses the whole sequence at once: staying on a chord is free and changing costs 0.08, so a change is only made when the music clearly supports it for a while.

$$\textit{best path} = \textit{argmax}\, \Sigma_\textit{beats}\, [\, \textit{similarity}(\textit{chord_b}, \textit{beat b}) - 0.08 \cdot (\textit{chord_b} \neq \textit{chord}_{b-1}) \,]$$

7. The melody: harmonic salience

For every half beat and every candidate note from G3 to B5, the importer adds up the spectrum at the note's frequency and its next two harmonics, weighting the fundamental most (1, 0.5, 0.3; summing many harmonics with equal weight favours the octave below):

$$\textit{salience}(f_0) = |X(f_0)| + 0.5 \cdot |X(2f_0)| + 0.3 \cdot |X(3f_0)|$$

Among the notes with at least 60% of the strongest salience, the **highest** is taken, because the tune usually sits on top of the harmony. A half beat with no note standing out (2.5 times the median) is a rest, and single-step blips between two equal notes are removed. This is the least certain step: voice, solos and accompaniment overlap in a mix.

8. Correcting by ear: tap tempo

If you tap Space on the beats while listening, tap k happens at time t_k. Fitting a straight line $t_k = a + P \cdot k$ by least squares (after numbering the taps by the median gap, so a missed tap does not break the count) gives the period P and phase a, with tapping errors averaged out:

$$BPM = 60 \div P \qquad \textit{bar 1} = a \bmod 4P$$

Taps are corrected by the 30 ms it takes a scheduled sound to reach your ears. After any correction the chords, melody and drum hits are re-read on the new grid (`regrid`) without analysing the file again.

9. Writing the chart

Chart	Easy	Normal	Hard
Melody	a note every 2 beats	a note every beat	a note every half beat

Chart	Easy	Normal	Hard
Chords	a chord per bar	every 2 beats	every beat
Rhythm	kick/snare on the beat	+ hi-hat	+ off-beat hits

- Each step takes the most common value in its group of half beats (or beats), and repeated values become - (hold), so long notes are held instead of re-struck.
- Melody notes are moved by octaves into C2–B5 (sheets read names like C6 as chords). If more than 16 different notes remain, the 16 most frequent are kept and the others move to the nearest kept note. The lower half goes to the left hand, the upper half to the right, so pitch rises from left to right as on a keyboard.
- Chords: the 16 most common chords get a sign each (right hand first); any other chord is replaced by the kept chord sharing the most notes with it.
- Rhythm: a half beat whose kick or snare strength passes a threshold (0.5, 0.35 or 0.25 by difficulty) becomes Kick or Snare, whichever is stronger.
- MIDI sources use their exact notes and get a synthesized backing of chords and bass, since there is no recording to play.

21. Image processing: the background blur

1. Shrink the frame to 320×180 (16× fewer pixels).
2. Run MediaPipe's selfie segmenter on it (about 3.7 ms). It returns a confidence mask: 1 where it sees a person, 0 for background.
3. Soften the mask's edge with a 7×7 Gaussian blur and average it with the previous frame's mask, so the outline does not flicker.
4. Blur the small frame with a 21×21 Gaussian kernel (about 85 pixels at full size) and scale it and the mask back up to 1280×720.
5. Blend the sharp and blurred images pixel by pixel using the mask as the weight.

$$output = \alpha \cdot sharp + (1 - \alpha) \cdot blurred \qquad \alpha = person\ confidence$$

A Gaussian blur replaces each pixel with a weighted average of its neighbours, with weights following a bell curve. Blurring a small image and enlarging it gives almost the same result as blurring the full image, at a fraction of the cost. The blend first cost 28 ms with NumPy floating-point arrays; OpenCV's `blendLinear` does it in about 2 ms. Running the whole blur on a worker thread alongside hand tracking brings its cost down to about 1 ms per frame.

Appendix A. Measured performance

These were measured on the development laptop, mostly with offline tests that run the code on recorded or simulated data.

What	Result
Camera frame rate	30 fps in good light; 10–20 fps in a dim room
Hand tracking	10–16 ms per frame with hands found; about 35 ms while searching
Background blur	35 ms (first version) → 12 ms (OpenCV blend) → about 1 ms extra (in parallel)
Synthesizer	0.43 ms to render 6 voices for a 5.8 ms block
Audio output latency	23 ms (WASAPI)
Pitch bend accuracy	441 Hz → 495 Hz at +2 semitones (target 493.9 Hz)
Basic sign recognition, simulated hands at random angles	99–100% within 2 frames at up to 1.8 mm tracking noise; 100% within 6 frames at 2.7 mm
Extended set, 9 shapes, simulated	100% of 1,080 trials at 0.9 mm and at 1.8 mm noise
False repeats while holding still	0 in 30 s with heavy simulated jitter (was 40 before the fixes)
Game metronome timing	clicks within 6 ms of the beat
Looper timing	0.52 s loop replayed at 0.53 s, 1.05 s, 1.57 s
Synthetic hand model	18 of 18 shapes (9 per hand, including fist) recognised
Strike speed (sign formed in 67 / 133 / 267 / 467 ms)	strength 1.00 / 0.70 / 0.41 / 0.21, so velocity 1.00 / 0.79 / 0.59 / 0.45
Calibration on synthetic hands (Extended, both hands)	every sign read correctly with the fitted thresholds (100%)
Song analysis, synthetic 112 BPM song (WAV and MP3)	112.00 BPM; bar 1 at 1.286 s (true 1.300 s); C major; chords 8 of 8 bars; melody 8 of 8 beats; 0.3 s per analysis
Tempo on plain kick-and-snare songs	75, 90, 112 and 140 BPM found as 75.2, 89.7, 112.2 and 139.8 (112 and 140 came out at half speed before the double-tempo check)
Window drawing	about 7 ms per frame at 30 fps (demo mode)
Unit tests	14 tests in about 1.3 s

Appendix B. Glossary

Term	Meaning
Landmark	One of 21 tracked points on a hand, such as a fingertip or knuckle.
World coordinates	MediaPipe's 3D landmark positions in metres, independent of where the hand is in the image.
Slot	One sound position on a hand: sign 1 to 4 (Basic) or 1 to 8 (Extended).
Curl	The total bend of a finger across its three joints, in degrees.
Hysteresis	Using separate on and off thresholds so a value near the boundary cannot flicker.
Debounce / confirmation	Requiring a change to persist for a few frames before acting on it.
Onset	The moment a sign first appeared; used for game timing.
Event	A small tuple describing a musical action, shared by synth, MIDI, looper and game.
Voice	One sounding note inside the synthesizer.
Partial	One sine component of a sound; partials together make the timbre.
Envelope	How a note's loudness changes over time: attack, decay, release.
Phase accumulator	A running phase advanced each sample, which lets frequency change smoothly.
Block / callback	A chunk of audio samples, and the function the sound card calls to get the next one.
Latency	Delay between an action and its result.
BPM	Beats per minute.
Voicing	Which octave each note of a chord is placed in.
Arpeggio	A chord played one note at a time.
Overdub	Recording a new loop layer on top of what is already playing.
General MIDI (GM)	A standard that fixes instrument and drum numbers so files sound similar on any synth.
Segmentation mask	An image saying, for each pixel, how likely it belongs to a person.
Calibration	Measuring your hands while you make known signs, and fitting the thresholds to the readings.
Percentile	The value below which a given share of readings fall; the 95th percentile ignores the top 5% outliers.
Velocity	How hard a note is struck (MIDI 1–127); here set by strike speed.
Reverb send	The share of a sound sent to the reverb; the rest stays dry.
Comb / all-pass filter	Feedback delays used to imitate a room: repeating echoes, then smeared echoes.
Quantise	Moving note starts to the nearest step of a timing grid.
Count-in	Clicks before recording or playing starts, so you come in on the beat.
Session	A recording of tracked hand landmarks (no images) that can be replayed as input.
Spectrogram	Frequency content over time, from Fourier transforms of short overlapping windows.
Spectral flux	How much the spectrum grows from one window to the next; peaks mark onsets.
Autocorrelation	How similar a signal is to a shifted copy of itself; peaks reveal repetition periods.
Chroma	The energy of each of the 12 note names, ignoring octave.
Key profile	How strongly each of the 12 notes belongs to a key, from listening experiments.
Viterbi / dynamic programming	Finding the best sequence by building on the best partial solutions.
Downbeat	The first beat of a bar.
Ticks	MIDI file time units: fractions of a beat (480 per beat here).