Dynamical Systems Orchestration

A blob-driven MPE polysynth controlled by camera and keyboard

COMPLETE BUILD GUIDE · v4.0 · MPE EDITION

macOS · SuperCollider · OpenCV · GUDHI

Persistent homology blob detection · Flow-advected Wasserstein tracking · Physics-informed Kalman · Birth-topology fingerprinting · Thermodynamic hue classification · Zombie voice pool

v4.0: Voice-slot architecture · MPE SynthDef · Computer-keyboard performance + edit modes · Bidirectional blob↔voice binding · Controller-abstraction layer for Phase-2 external MIDI surface

Contents

Version History — Iteration Roadmap

Dynamical Systems Orchestration (working title: Audible Lava) has evolved through five architectural iterations. Each addresses a concrete failure mode or capability gap observed in the previous version. The progression has moved from generic multi-object tracking, through domain-specific physics-grounded methods that exploit the lava lamp's unique dynamics, into the present v4.0 reframing in which blobs no longer simply *generate* music but *populate the voice slots of a player-controllable polyphonic synthesiser*.

v1.0 — Initial Implementation

OpenCV SimpleBlobDetector with HSV colour mask. Naive distance-threshold Hungarian assignment. Constant-velocity 4-state Kalman. Single hardcoded MIN_BLOB_AREA threshold. OSC bridge to SuperCollider with pitch / amplitude / pan mappings. No identity fingerprinting; spurious birth / death events on every transparency fluctuation. No warm-up gate, so hundreds of ghost IDs spawned during MOG2 stabilisation.

v2.0 — Robust Four-Layer Tracking

SimpleBlobDetector replaced by GUDHI CubicalComplex H0 persistence — a topological significance filter in place of an area threshold. MOG2 background subtraction became the primary foreground filter. Contour moment enrichment (orientation, elongation, Hu invariants) added to PersistentBlob. CamShift colour-histogram refinement added as a third layer (later identified as ineffective for monochromatic wax). Kalman + Hungarian stack. Coast-and-retire with MAX_COAST=8 frames. Three SuperCollider SynthDefs (blobVoice, blobAngle, blobStrike). Elongation → resonance, orientation → pan, birth → cutoff. MOG2 warm-up gate eliminated ghost IDs during model build.

v3.0 / v3.1 — Physics-Grounded, Flow-Advected Architecture

Optical flow mask advection replaced per-frame assignment as the primary identity carrier — identity is the mask, solved once at birth, not every frame. Wasserstein-inspired topological matching (birth-death plane, 75% weight) resolves ambiguous overlaps. A 5-state physics Kalman [x, y, vx, vy, ay] with buoyancy mean-reversion and directional ay priors handles turning-point dynamics. Birth topology fingerprinting enables re-identification after transparency gaps. Circular mean hue is classified into rising / falling / turning direction and drives attack / release envelopes and oscillator detuning. LampProfile dataclass persists all per-lamp calibration. The calibration GUI derives every parameter from freehand-drawn blob regions.

v3.2 — Stability Against Turning-Point Storms

Four targeted changes to eliminate the cascade of phantom births, deaths, and release-tail stacking that occurred when wax blobs reversed direction at the lamp top or bottom.

- **MAX_COAST raised 8 → 20 frames** so tracks survive longer detection gaps — roughly 1.3 s at 15 fps instead of 0.5 s.
- **MOG2 learning rate throttled to 1e-4 after warm-up**, effectively freezing the background model, so stationary wax pooling at turning points is no longer absorbed into the background and dropped from the foreground mask.
- **Hue term down-weighted in the fingerprint distance** (divisor 25 → 60), so a full thermal reversal (≈70° hue shift) no longer blows past the match threshold on its own and re-identification can correctly recover a blob through its reversal.
- **Zombie voice slot-pool with resurrection window** in the voice manager: when a tracker ID disappears, its voice is held at a soft sustain for up to 4 s rather than released; the next nearby blob with compatible persistence resurrects it in place. This eliminates release-tail stacking and preserves sonic identity across detector gaps that the tracker itself could not bridge.

v4.0 — Current: MPE Edition (Voice Slots and Player Agency)

v4.0 reframes the relationship between blob and voice. Through v3.2 the system was a one-way translator — blobs produced sound. v4.0 turns the voice into a first-class object that the blob is one of several possible drivers of. The other driver is the human at a computer keyboard.

- **Voice-slot abstraction.** A fixed pool of 16 voice slots replaces the implicit blob_id → voice mapping of v3.2. Each slot carries its own synth state (pitch, wavetable, envelope, filter, mod matrix) and a lifecycle state of *IDLE*, *BLOB_ONLY*, *KEYBOARD_ONLY*, *HYBRID*, or *ZOMBIE*. The v3.2 zombie pool is preserved as a sub-state of BLOB_ONLY voices.
- **MPE-friendly SynthDef.** The v3.2 \blobVoice (single filtered saw) is replaced by \mpeVoice — a wavetable oscillator with a multi-mode state-variable filter, full ADSR, and a 6-source × 8-destination internal modulation matrix. Per-voice OSC namespace /voice/<slot>/...
- **Computer-keyboard MPE controller.** A pygame-based input layer turns the laptop keyboard into a 16-voice MPE surface with two coexisting modes — Performance (play notes, slot-select) and Edit (scroll the synth parameter tree of the selected slot). All keyboard input flows through a Controller abstraction.
- **Bidirectional blob ↔ voice binding.** A small BindingManager makes blob-driven and player-driven voice creation symmetric. Blobs can spawn voices (the v3.2 default), the user can create empty voices and then tie them to the next-born blob, and any voice can be in HYBRID mode where the user plays notes and the blob continuously modulates the synth state.
- **Routing matrix.** Each voice carries an explicit (source → destination, depth, curve) routing list. The sources are blob CV features — y_norm, vel_y, persistence, hue, elongation, orientation, age — and the destinations are the synth's mod inputs. Default routings reproduce v3.2 behaviour; users rewrite them in real time.
- **Controller abstraction.** All input is funnelled through a Controller protocol that emits ControllerEvent objects. Phase 2's external MIDI surface adds a single new adapter — the rest of the system does not change.
- **Removed.** The plinth, ESP32 feedback speaker, three-camera system, and the closed-loop acoustic feedback concept are no longer part of the project. The musical interest now comes from the player-blob duet, not from the lamp's sound modulating its own behaviour.

High-Level Overview

Dynamical Systems Orchestration is a generative, performable music instrument built around a lava lamp. A camera films the lamp continuously. Computer vision software analyses each frame in real time, extracting the size, position, velocity, lifecycle state, topological significance, shape geometry, and thermodynamic phase of every wax blob. These properties are routed into a 16-voice polyphonic SuperCollider synthesiser, where each blob occupies — or shares — one of the synth's voice slots.

What the system produces is not entirely automatic. A human player at a computer keyboard can select any voice, edit its synth parameters, take over note-playing duties from the blob, or set up empty voices that will bind to the next blob to appear. The blob's CV features always continue to act as continuous modulation on whichever voice it is bound to — the lamp's slow organic motion is, in effect, MIDI CC for the synth — but the musical foreground is a duet between the wax and the player, not a monologue from the wax.

The Central Idea

The lava lamp is a *dynamical score* — a dissipative physical system whose blob lifecycle (birth, death, merge, split, turning) drives compositional events and continuous parameter modulation. The novel methodological contribution sits at the intersection of topological data analysis and performative music: persistent homology is used to quantify the significance of each wax blob as a contrast-against-surroundings measure, rather than as a threshold on raw intensity or pixel area. This yields musically meaningful voice assignment, because a dim but isolated blob (topologically salient) drives a voice, while a bright but indistinct region (topologically noisy) does not.

Why This Architecture Exists

Three timescale mismatches make naïve approaches fail. **First**, blob cycles last 90–180 seconds while MIDI events last milliseconds; bridging requires deliberate multi-layer design, not per-frame mapping. **Second**, position is the least stable feature of a lava lamp blob — wax moves slowly and gets occluded often — while topological signature (birth, death) and thermodynamic state (hue) are comparatively stable. **Third**, detection is inherently intermittent: wax can become momentarily transparent, two blobs can pool into one, background subtraction can absorb stationary material.

v3.0/3.1 fixed the first two by making the binary mask the identity carrier (advected by optical flow, refined by detection) and by matching in topological feature space rather than pixel space. v3.2 addressed the third by recognising that turning points are special: wax decelerates, hue flips, MOG2 forgets the now-stationary material, detection drops out, re-identification fails across the thermal transition, and the voice manager stacks release tails as the same physical blob is born and killed repeatedly. v4.0 builds on top of all of that — none of the tracking work is discarded — and adds *player agency* as a first-class architectural concern.

> *Through v3.2 the lamp had a voice. In v4.0 it has a place in the orchestra.*

Methodology at a Glance

A single camera frame flows through six conceptual stages. All six are active in Phase 1 (single-camera + keyboard). Phase 2 replaces stage 5's input source — the keyboard — with an external MIDI controller surface, without touching stages 1–4 or stage 6. Each stage below is summarised in one paragraph; full technical detail follows in the Deep Dive section after the Phase 1 walkthrough.

Stage 1 — Calibration (Once at startup)

The calibration GUI captures a frozen camera frame and lets the user draw freehand around visible wax blobs plus a rectangle for the lamp ROI. From the drawn pixels it derives HSV range, area bounds, lamp geometry, and thermodynamic hue thresholds (35th / 65th percentile split). Written to lamp_profile.json and loaded automatically on subsequent runs.

Stage 2 — Detection (Every frame)

MOG2 foreground ∩ wax-colour HSV mask → morphological clean-up → GUDHI CubicalComplex H0 persistence on the masked greyscale → filter by persistence threshold → locate connected component at death threshold → contour moments (orientation, elongation, Hu invariants) → circular mean hue → thermodynamic direction (rising / falling / turning). Output: a list of PersistentBlob objects sorted by persistence. MOG2 learning rate is frozen to ~1e-4 after the warm-up period so stationary wax stays in the foreground.

Stage 3 — Flow Advection and Matching (Every frame)

Farneback dense optical flow between previous and current greyscale. Every existing track's mask is inverse-warp remapped forward by the flow. IoU between advected masks and detection masks determines primary matches. Ambiguous overlaps are resolved by Wasserstein-inspired distance in the (birth, death) plane. Unmatched detections are checked against the fingerprint buffer for re-identification before a birth event fires. Every 5 frames the mask is blended with the detection contour. Coasting tracks survive up to MAX_COAST=20 frames before a death event.

Stage 4 — Physics Kalman Update (Every frame)

Each matched track's 5-state Kalman filter [x, y, vx, vy, ay] is corrected with the detection measurement. Measurement noise adapts based on y-position: near the lamp top / bottom, noise is high (trust physics model); in mid-column, noise is low (trust detection). The ay state decays toward zero (mean-reversion coefficient 0.9) and is initialised with a directional prior from the detected thermodynamic direction. Coasting tracks receive predict-only steps.

Stage 5 — Voice Management and Player Input (Every frame)

This is the v4.0 stage. The VoiceManager maintains 16 voice slots, each holding full synth state plus a lifecycle state. Blob lifecycle events drive slot allocation (birth → spawn into IDLE slot or resurrect zombie; death → zombify or detach; merge → fade to silence; split → silent emergence). The Controller layer (KeyboardController in Phase 1) emits ControllerEvents that select slots, play notes, edit synth parameters, and create/tie/detach voices. The routing matrix multiplies blob CV features by per-voice depths and writes the result to the SC node's mod inputs. All parameters smooth toward their targets via first-order lag.

Stage 6 — Synthesis and Output (Every frame)

SuperCollider's scsynth runs one \mpeVoice instance per active slot, plus the existing \blobAngle (orientation-panned sine pad) and \blobStrike (transient on birth) where enabled. All voices route through a single stereo bus and out the audio interface. There is no plinth speaker, no ESP32, and no acoustic feedback path; the loop closes through the player's ears and hands, not through the lamp.

Section 1 — Project Description & Goals

Artistic Goals

• Create a generative musical instrument whose source material is a physical, non-digital process — heated wax in a glass vessel — but which is *performable*, not autonomous.

• Demonstrate that thermodynamic entropy — the unpredictable organic motion of heated wax — can serve as a musically rich and meaningful continuous modulation source within a multi-voice synthesiser.

• Produce music that evolves on the same slow timescales as the lamp: minutes, not milliseconds. A blob rising for 90 seconds should contribute to a 90-second sweep across the synth state of its bound voice.

• Give each wax blob a persistent, audible identity — listeners should be able to follow individual voices, hear them interact, and perceive their birth, development, and death.

• Allow a human player to engage musically with the lamp's voices: select any blob's voice, take it over, edit its timbre, play notes against it, or set up new voices that wait for the next blob to appear.

Technical Goals

• Stable blob identity tracking robust to transparency fluctuations, occlusion, and proximity ambiguity using flow-advected masks, topological matching, fingerprint re-identification, and a zombie voice pool at the sonic layer.

• Real-time video processing at 15 fps with sub-50 ms vision latency.

• Temporal coherence: blob lifecycle events — not per-frame pixel values — are the primary compositional triggers.

• Per-lamp calibration from a single freehand-drawing session, with all parameters derived automatically from drawn pixel statistics.

• 16 polyphonic voices in SuperCollider, each with independently editable wavetable / envelope / filter / mod-matrix state, addressable by a namespaced OSC address.

• A clean controller abstraction that lets Phase 2 swap the keyboard for an external MIDI surface without rewriting the voice or routing layers.

• Each architectural decision grounded in domain-specific physics (buoyancy dynamics, thermodynamic phase) rather than generic multi-object-tracking heuristics.

Section 2 — Pipeline Architecture

The pipeline runs as a continuous loop, processing one camera frame per cycle at approximately 15 fps. Each cycle passes data through the six stages summarised earlier. The v3.0+ tracking layer is fundamentally different from v1.0 / v2.0 in that blob identity is no longer solved as a per-frame assignment problem: it is carried forward by the flow-advected mask and only re-evaluated on ambiguity. v3.2 added a second identity layer at the voice manager via the zombie pool. **v4.0 adds a third identity layer above that**: voice slots live independently of blobs and can be driven by either a blob, a player, or both.

ASCII System Diagram

```
USB Camera (1280x960 @ 15 fps)
        |
        v
blob_detection.py  [BlobDetector - stateful, owns MOG2]
    MOG2 (history=500, varThreshold=16)  <- learning_rate=1e-4 post-warmup
    && Wax colour HSV mask (from lamp_profile)
    -> Morphological clean (7x7 close, 5x5 open)
    -> GUDHI CubicalComplex H0 persistence (downscale=4)
    -> Filter by pers_thresh
    -> Contour moments (orientation, elongation, hu[7])
    -> Circular mean hue (wrap-safe)
    -> Direction: rising | falling | turning
        |
        v  [PersistentBlob] sorted by persistence desc
blob_tracker.py  [BlobTracker]
    A. Farneback dense flow
    B. Advect all track masks (inverse-warp)
    C. IoU matching (threshold=0.25)
    D. Wasserstein resolution (topo:0.75, pos:0.15, hue:0.10)
    E. Mask refinement every 5 frames (50/50 blend)
    F. Physics Kalman correct [x, y, vx, vy, ay]
    G. Coast / retire (max_coast=20)
    H. Fingerprint re-ID (hue weight /60)
        |
        v  [TrackedBlob] + events: birth | death | merge | split
binding_manager.py  [BindingManager]                  <-- v4.0
    on_blob_birth -> match waiting voice / resurrect zombie / spawn IDLE
    on_blob_death -> zombify / detach (depending on voice state)
    bind_request  -> tie KEYBOARD_ONLY voice to blob
        |
        v  slot allocation events
voice_manager.py  [VoiceManager - 16 slot pool]       <-- v4.0
    Slot states: IDLE, BLOB_ONLY, KEYBOARD_ONLY, HYBRID, ZOMBIE
    Each slot holds: synth_params + routing_table + sc_node
        ^                                                  |
        | ControllerEvents                                 |
        |                                                  v
controller.py  [Controller]                        /voice/<n>/...
    KeyboardController  (Phase 1)                        |
    MIDIController      (Phase 2)                        v
    -> NOTE_ON / NOTE_OFF                       SuperCollider scsynth
    -> SELECT_SLOT                               \mpeVoice (per slot)
    -> SET_PARAM(family, value)                  \blobAngle (optional)
    -> CREATE_EMPTY / TIE / DETACH               \blobStrike (transient)
    -> LEARN                                             |
                                                         v
                                              Focusrite Scarlett -> Speakers
```

Temporal Coherence — Four Layers

The central design challenge is ensuring the music evolves on the same timescale as the lamp. A blob typically takes 90–180 seconds to rise from base to top. v4.0 has four mechanisms for preserving temporal coherence:

- **Flow-advected identity at the tracker** — the mask is always present, even when the detector loses the blob momentarily. (v3.0)
- **Fingerprint re-identification** — lost tracks are recoverable for 15 seconds by topological fingerprint. (v3.1)
- **Zombie voice pool** — sonic identity persists for a further 4 seconds beyond the tracker gap, smoothing over any detection failures that the tracker itself could not bridge. (v3.2)
- **Voice slots independent of blobs** — a voice can outlive its blob indefinitely if the player has bound to it (HYBRID or KEYBOARD_ONLY). The blob's modulation simply ceases when it dies; the voice itself is kept alive by the player. (v4.0)

Section 3 — Prior Art & Related Works

The following works constitute the primary precedent landscape. This project combines simultaneously: a real lava lamp as live source, camera-based analysis, topological blob tracking with thermodynamic enrichment, a voice-slot architecture with bidirectional player ↔ blob binding, and real-time MPE-style polyphonic synthesis with sonic-identity continuity — a combination not present in any prior known work.

Work	Lava Lamp	Live Camera	Blob Analysis	Music Output	Player Agency
Jenna Sutela — I Magma (2019)	Yes	Yes	Motion analysis	—	—
Daniel Rapp — SpectroFace	—	Yes	Row scan	Yes	—
VOSIS / VOSIS Pro (iOS)	—	Yes	Pixel scan	Yes	Touch surface
Virtual ANS (WarmPlace.ru)	—	—	Static image	Yes	Drawing
Susi Gelb — Core 02/03, Melt 01	Yes	Yes	Motion data	—	—
Refik Anadol — Unsupervised	—	—	Simulated	—	—
Aphex Twin — Windowlicker	—	—	Embedded image	Pre-composed	—
Chen et al. — Images That Sound	—	—	Generative	Inverse	—
This project (v4.0)	Yes	Yes	TDA + flow + hue + slots	Yes — MPE polysynth	Keyboard / MIDI controller

Novelty Summary

To the best of the author's knowledge, this is the first work to combine: (1) a physical lava lamp as live input, (2) H0 persistent homology for topological blob significance filtering, (3) flow-advected mask identity with Wasserstein topological matching, (4) physics-informed buoyancy Kalman with directional priors, (5) birth topology fingerprinting for re-identification, (6) thermodynamic hue as a phase indicator driving musical parameters, (7) a zombie voice pool preserving sonic identity across detection gaps, and (8) a bidirectional voice-slot architecture in which blob CV features and player input share the same MPE-style synthesiser as continuous-modulation and discrete-event sources respectively.

Section 4 — Parts List

Phase 1 — Core (Single Camera + Keyboard)

Component	Qty	Purpose	Est. Cost (GBP)
MacBook Pro (M-series recommended)	1	Central compute: vision pipeline, GUDHI TDA, SuperCollider, keyboard input	Existing or £1,099+
Logitech Brio 4K or C922 Pro (USB-A)	1	Primary camera (calibrated at 1280×960)	£75–95
Articulated camera arm / desk clamp	1	Position camera 30–40 cm from lamp	£8–12
Focusrite Scarlett Solo (3rd gen)	1	Audio interface: stereo SC output	£110–130
Mathmos Astro lava lamp (500 ml)	1	Primary installation object	£35–50
TRS 3.5 mm to XLR cables (pair)	2	Scarlett outputs to powered monitors	£8–12
Powered studio monitors (e.g. Yamaha HS5)	2	Main audio output	£120–300

Phase 1 — Optional

Component	Qty	Purpose	Est. Cost (GBP)
Behringer DeepMind 6 Module (used)	1	Optional analogue voice for one or two slots; driven via Scarlett MIDI out	£130–165
DIN-5 MIDI cable 1 m	1	Required only if DeepMind 6 is in the rig	£5–8

Phase 2 — External MIDI Controller

Component	Qty	Purpose	Est. Cost (GBP)
MIDI controller (knob/button surface, see candidates list)	1	Replaces the keyboard as the player input device. Recommended primary: MIDI Fighter Twister (16 RGB push-encoders × 4 banks).	£100–400
USB-C / USB-A cable to laptop	1	Class-compliant MIDI over USB; no driver needed on macOS	£5–10
(Optional) Pad controller for triggering	1	e.g. Akai LPD8 — useful for slot-select / chord trigger if primary surface is encoders-only	£40–80

Phase 2 candidate controllers

Controller	Layout	Notes
MIDI Fighter Twister	16 endless RGB push-encoders × 4 banks (= 64 logical)	Best overall match. RGB feedback for state recall, banks page across voice families. ~£200.

Controller	Layout	Notes
Faderfox EC4	16 endless encoders + OLED labels, 4 setups	Most engineered controller in this class. ~£400. Premium choice.
Faderfox PC12	12 endless encoders + buttons, 16 setups, OLED	Cheaper alternative to EC4 with more buttons.
Novation Launch Control XL	24 absolute knobs + 8 faders + 16 buttons + 8 templates	Faders give a 'voice strip' mixer-style control of all active slots. ~£170.
Akai MIDImix	24 absolute knobs + 9 faders + 16 buttons	Cheapest functional option (~£100). No LED feedback on knobs — recall harder.
Korg nanoKONTROL2	8 knobs + 8 faders + buttons	Ubiquitous, ~£60. Probably too few controls for 16 slots; useful as secondary surface.
Native Instruments Maschine Mikro MK3	16 pads + 8 knobs + screen	Adds sampler/sequencer ambition. Loosely tied to NI host software.
OXI One	8 endless knobs + deep sequencer + display	Standalone, premium (~£600). Cleanest fit if sampler/sequencer extension matures.

Section 5 — Software

Package	Version	Role	Install
opencv-python	>=4.8.0	MOG2, Farneback flow, contour moments, HSV masking, Kalman	pip install opencv-python
numpy	>=1.24.0	Array math, flow remap, circular mean hue	pip install numpy
gudhi	>=3.7.0	CubicalComplex H0 persistence (core TDA engine)	pip install gudhi
scipy	>=1.10.0	linear_sum_assignment for Hungarian fallback	pip install scipy
python-osc	>=1.8.0	OSC UDP client → SuperCollider	pip install python-osc
pygame	>=2.5	Cross-platform keyboard polling for the Phase 1 controller layer (no extra OS permissions on macOS)	pip install pygame
mido + python-rtmidi	latest	MIDI I/O for Phase 2 (external controller) and the optional DeepMind 6 output	pip install mido python-rtmidi
torch / torchvision	2.x	Reserved for future RAVE neural-audio-decoder voices; MPS on Apple Silicon	pip install torch torchvision
sounddevice	latest	Reserved for future RAVE waveform output to Scarlett in real time	pip install sounddevice
SuperCollider	3.13+	scsynth synthesis server; \mpeVoice, \blobAngle, \blobStrike SynthDefs	supercollider.github.io

Environment note. Python 3.11.9 is pinned for macOS compatibility with GUDHI and current OpenCV wheels. A Conda environment named dso is the reference setup; a plain venv also works. All modules live in a flat directory with no __init__.py required.

Removed since v3.2. PlatformIO and pyserial are no longer required — the ESP32 firmware path has been retired with the plinth and feedback speaker. They can be reinstated as an optional output target in a future revision but are not part of v4.0.

Section 6 — Project Timeline & Milestones

Solo developer, 10–15 hours per week. Phase 1 is a complete, performable instrument on its own. Phase 2 is an additive upgrade — the system runs the entire time, gaining a new input source. The timeline assumes v3.2 tracking is already working.

Week	Phase	Tasks	Milestone
1	P1 setup	Verify v3.2 tracking is healthy (test_setup, test_detection, test_tracker). Confirm calibration, MOG2 freeze, zombie pool.	
2	P1 — voice slots	Refactor VoiceManager to slot-pool architecture. Add Voice class with synth state + routing list. Preserve all v3.2 zombie behaviour as BLOB_ONLY sub-state.	M1: Slot abstraction landed
3	P1 — overlay	Voice-ID overlay on video GUI: ring colour by state, label with slot ID + binding state, side panel listing all 16 slots.	
4	P1 — SC SynthDef	\mpeVoice SynthDef in SC: wavetable osc, SVF, ADSR, internal 6×8 mod matrix. /voice/<n>/ namespace. Manual sanity tests.	M2: SC layer ready
5	P1 — keyboard	pygame-based KeyboardController. Performance + Edit modes. Slot select, note input, parameter scrolling.	
6	P1 — routing	Routing matrix runtime: source → destination, depth, curve. Default routings reproducing v3.2 behaviour.	M3: Blob CC modulating SC voices
7	P1 — binding	BindingManager: blob birth / death / merge / split routed into slot lifecycle. CREATE_EMPTY / TIE / DETACH commands.	M4: Phase 1 complete
8	P1 polish	Smoothing tuning, default-routing audition, 4-hour stability test.	
9	P2 setup	Choose controller; configure mido port; build MIDIController adapter behind the same Controller protocol.	
10	P2 — learn	MIDI Learn mode: select param in Edit mode, send a knob/button — bind stored. Persist learn maps to disk.	M5: External controller online
11	P2 — workflow	Live-set workflow tuning: bank/page conventions, slot-select ergonomics, recall.	
12	P2 polish	Optional: explore sampler / sequencer extensions on the same controller.	M6: Performable instrument

Phase 1 — Keyboard MPE Walkthrough

Phase 1 is a complete, self-contained, performable instrument requiring only a MacBook Pro, one USB camera, a Focusrite Scarlett Solo, and powered speakers. It implements the full v4.0 pipeline: calibration GUI, flow-advected tracking with topological matching and fingerprint re-identification, physics Kalman, thermodynamic hue classification, voice-slot pool, MPE-style SuperCollider synth, computer-keyboard controller, and bidirectional binding. All Phase 2 extension points are marked in the code for later activation without rewrites.

Step 1 — Environment Setup

1.1 Install Python and create virtual environment

```
brew install python@3.11
mkdir ~/lava && cd ~/lava
python3.11 -m venv .venv
source .venv/bin/activate
```

1.2 Install all Python dependencies

```
pip install "opencv-python>=4.8.0"
pip install "numpy>=1.24.0" "scipy>=1.10.0"
pip install "gudhi>=3.7.0"
pip install "python-osc>=1.8.0"
pip install pygame
pip install mido python-rtmidi      # Phase 2 prep + optional DeepMind output
```

1.3 Verify environment

```
python test_setup.py
# All checks should print OK before proceeding.
```

Install SuperCollider from supercollider.github.io (macOS: .dmg, drag to Applications). Boot the server and verify with s.boot; s.meter; in the SC IDE.

Step 2 — Camera and Physical Setup

Mount the Logitech Brio 4K or C922 Pro on an articulated arm 30–40 cm from the lamp, centred on the full lamp height. **Lock exposure manually** to prevent auto-exposure adjusting to lamp colour changes — this is essential, because auto-exposure re-baselines MOG2 during the day / night cycle and causes mass phantom births. The calibrated profile uses camera index 0 at 1280×960.

```
# Verify camera index
python3 -c "
import cv2
for i in range(4):
    cap = cv2.VideoCapture(i)
    if cap.read()[0]:
        print(f'Camera {i}: OK')
    cap.release()
"
# Cover the lens with your hand to confirm which index is the lamp camera.
```

Step 3 — Calibration GUI

On first run, audible_lava.py opens calibration automatically. To re-calibrate explicitly: python calibration_gui.py

3.1 Procedure

(a) Wait 30–60 minutes after power-on so wax blobs are visible and in their characteristic colour range. (b) Press SPACE to freeze a frame. (c) Left-drag freehand around each visible wax blob — one region per distinct mass;

the panel on the right updates HSV statistics and a colour swatch live. (d) Right-drag a rectangle around the interior of the lamp glass (excluding heater base and cap); this becomes the lamp ROI used for pitch normalisation. (e) Press ENTER or S to save.

3.2 Example saved profile

```
{
  "wax_hue_lo": 0,          "wax_hue_hi": 5,
  "wax_sat_min": 245,       "wax_val_min": 52,
  "lamp_x1": 198,           "lamp_y1": 243,
  "lamp_x2": 638,           "lamp_y2": 792,
  "blob_area_min": 5020.5, "blob_area_max": 80586.0,
  "n_blobs_expected": 4,
  "frame_h": 960,           "frame_w": 1280,
  "rising_hue_threshold":  0.0,
  "falling_hue_threshold": 0.0,
  "observed_hue_mean":     0.004,
  "expected_persistence_mean": 15,
  "expected_persistence_std":  3.0
}
```

Key thresholds must match the actual image range. The expected persistence mean is calibrated from the drawn regions; a common pitfall is leaving a legacy value (e.g. 220) that cannot be achieved on a masked image where maximum actual persistence is 60–80, producing zero blobs. Re-run calibration if blob counts are surprising.

Step 4 — Detection & Tracker Calibration

```
python test_detection.py
# Keys: +/- = persistence threshold * c = masks
#       h/H = hue * s/S = sat * v/V = val

python test_tracker.py
# Each blob should keep the same random colour (= same ID) as it moves.
# IDs should not flicker. The coast counter (coast=N) should reset to 0 when
# a blob is re-detected. If a blob disappears briefly and reappears with the
# same ID (re-identified from the fingerprint buffer), no birth event prints.
```

Press **c** in test_detection to open the three mask windows: MOG2, Wax colour, Combined. The Wax colour mask should show white only where wax pixels are. Direction ring colour: green = rising, blue = falling, yellow = turning. Copy the final persistence value into expected_persistence_mean in lamp_profile.json to make it persistent.

Step 5 — \mpeVoice SynthDef in SuperCollider

Replace the v3.2 \blobVoice with the v4.0 \mpeVoice. The skeleton:

```
SynthDef(\mpeVoice, {
    |out=0, freq=220, bend=0, amp=0.3, gate=1, pan=0,
     wave=0, bank=0,
     attack=1.5, decay=0.4, sustain=0.7, release=2.5,
     cutoff=2000, res=0.4, fmode=0,
     mod1=0, mod2=0, mod3=0, mod4=0, mod5=0, mod6=0,
     m1_pitch=0, m1_wave=0, m1_cut=0, m1_res=0, m1_a=0, m1_r=0, m1_amp=0, m1_pan=0,
     m2_pitch=0, m2_wave=0, m2_cut=0, m2_res=0, m2_a=0, m2_r=0, m2_amp=0, m2_pan=0,
     m3_pitch=0, m3_wave=0, m3_cut=0, m3_res=0, m3_a=0, m3_r=0, m3_amp=0, m3_pan=0,
     m4_pitch=0, m4_wave=0, m4_cut=0, m4_res=0, m4_a=0, m4_r=0, m4_amp=0, m4_pan=0,
     m5_pitch=0, m5_wave=0, m5_cut=0, m5_res=0, m5_a=0, m5_r=0, m5_amp=0, m5_pan=0,
     m6_pitch=0, m6_wave=0, m6_cut=0, m6_res=0, m6_a=0, m6_r=0, m6_amp=0, m6_pan=0|

    var modSum, pitchMod, waveMod, cutMod, resMod, aMod, rMod, ampMod, panMod;
    var p, env, osc, sig;
```

```
    pitchMod = (mod1*m1_pitch) + (mod2*m2_pitch) + (mod3*m3_pitch)
             + (mod4*m4_pitch) + (mod5*m5_pitch) + (mod6*m6_pitch);
    waveMod  = (mod1*m1_wave)  + (mod2*m2_wave)  + (mod3*m3_wave)
             + (mod4*m4_wave)  + (mod5*m5_wave)  + (mod6*m6_wave);
    cutMod   = (mod1*m1_cut)   + (mod2*m2_cut)   + (mod3*m3_cut)
             + (mod4*m4_cut)   + (mod5*m5_cut)   + (mod6*m6_cut);
    // (...resMod, aMod, rMod, ampMod, panMod analogously)

    p   = freq * (2 ** ((bend + (pitchMod*1200)) / 1200));
    env = EnvGen.kr(
        Env.adsr(attack + aMod, decay, sustain, release + rMod),
        gate, doneAction:2);
    osc = VOsc.ar((bank.clip(0,3) * 4) + (wave + waveMod).clip(0,3),
                  freq:p, mul: amp + ampMod);
    sig = SVF.ar(osc, (cutoff + cutMod*4000).clip(40,16000),
                      (res + resMod).clip(0.1, 1.0), fmode);
    Out.ar(out, Pan2.ar(sig * env, (pan + panMod).clip(-1,1)));
}).add;
```

Wavetables. Pre-load 4 banks × 4 wavetables into VOsc buffers (Buffer.allocConsecutive). Bank 0 = analogue shapes (saw / square / pulse / triangle), bank 1 = harmonic series, bank 2 = formants, bank 3 = noise / metallic — easy to expand later.

Filter mode. fmode = 0 = LP, 0.5 = BP, 1 = HP. SVF.ar interpolates internally; CC values can be continuous.

Step 6 — VoiceManager Refactor

```
# voice_manager.py (sketch -- v4.0)

class VoiceState(Enum):
    IDLE          = 0
    BLOB_ONLY     = 1
    KEYBOARD_ONLY = 2
    HYBRID        = 3
    ZOMBIE        = 4

@dataclass
class Voice:
    slot_id:     int
    state:       VoiceState = VoiceState.IDLE
    blob_id:     int | None = None
    sc_node:     int | None = None
    synth:       SynthState = field(default_factory=SynthState.default)
    routing:     list[Routing] = field(default_factory=list)
    last_note:   int | None = None
    last_played: float = 0.0  # for HYBRID->KEYBOARD_ONLY transition on blob death

class VoiceManager:
    def __init__(self, profile, n_slots=16, ...):
        self.voices_by_slot = [Voice(slot_id=i) for i in range(n_slots)]
        self.voices_by_blob = {}
        self.selected_slot  = 0
        self.learn_target   = None      # (slot, param_name) for MIDI Learn
        # zombie/sustain settings preserved from v3.2

    # ---- lifecycle (called from BindingManager) ----
    def spawn_for_blob(self, blob): ...
    def zombify(self, slot_id):    ...   # existing v3.2 behaviour
    def resurrect(self, slot_id, blob): ...
    def detach(self, slot_id):     ...   # blob died but voice has been played

    # ---- player API (called from Controller layer) ----
    def select_slot(self, slot_id): self.selected_slot = slot_id
    def note_on(self, slot_id, note, velocity): ...
```

```
    def note_off(self, slot_id, note): ...
    def set_param(self, slot_id, family, key, value): ...
    def add_routing(self, slot_id, source, dest, depth, curve='LIN'): ...
    def remove_routing(self, slot_id, source, dest): ...
    def create_empty(self) -> int: ...
    def tie_to_blob(self, slot_id, blob_id): ...

    # ---- per-frame (called from main loop) ----
    def process_frame(self, tracked_blobs):
        for v in self.voices_by_slot:
            if v.state in (VoiceState.BLOB_ONLY, VoiceState.HYBRID):
                blob = self._lookup_blob(v.blob_id, tracked_blobs)
                if blob is not None:
                    self._apply_routing(v, blob)
            self._smooth_and_send(v)
        self._expire_zombies()
```

Step 7 — KeyboardController

```
# controller.py (sketch -- v4.0)

@dataclass
class ControllerEvent:
    type:  str          # 'NOTE_ON', 'NOTE_OFF', 'SELECT_SLOT', 'SET_PARAM',
                        # 'CREATE_EMPTY', 'TIE_BLOB', 'DETACH', 'LEARN'
    slot:  int | None  = None
    param: str | None  = None
    value: float | None= None
    note:  int | None  = None
    velocity: int | None= None

class KeyboardController:
    NOTE_KEYS_LOW  = "zsxdcvgbhnjm,"      # white+black, octave 1
    NOTE_KEYS_HIGH = "q2w3er5t6y7ui"      # white+black, octave 2
    SLOT_KEYS      = "1234567890"         # slots 0..9
    EDIT_TOGGLE    = pygame.K_TAB
    PARAM_FAMILY   = {pygame.K_p:'pitch', pygame.K_w:'wave',
                      pygame.K_e:'env',   pygame.K_f:'filter',
                      pygame.K_m:'mod'}

    def __init__(self):
        self.mode = 'PERFORMANCE'
        self.octave = 4
        self.current_family = None
        # ...

    def poll(self) -> list[ControllerEvent]:
        events = []
        for ev in pygame.event.get():
            if ev.type == pygame.KEYDOWN:
                if ev.key == self.EDIT_TOGGLE:
                    self.mode = 'EDIT' if self.mode == 'PERFORMANCE' else 'PERFORMANCE'
                elif ev.unicode in self.SLOT_KEYS:
                    events.append(ControllerEvent('SELECT_SLOT',
                        slot=int(ev.unicode) if ev.unicode != '0' else 9))
                elif self.mode == 'PERFORMANCE':
                    note = self._note_for(ev.unicode)
                    if note is not None:
                        events.append(ControllerEvent('NOTE_ON',
                            note=note, velocity=100))
                else:  # EDIT
                    events.extend(self._handle_edit(ev))
            elif ev.type == pygame.KEYUP:
                # NOTE_OFF for performance keys
```

```
                ...
        return events
```

Step 8 — BindingManager

```
# binding_manager.py (sketch -- v4.0)

class BindingManager:
    def __init__(self, voices: VoiceManager):
        self.voices = voices
        self.pending_keyboard_voices = []  # KEYBOARD_ONLY voices waiting to tie

    def on_blob_birth(self, blob):
        # 1. Try to tie to a waiting keyboard-created empty voice
        if self.pending_keyboard_voices:
            slot = self.pending_keyboard_voices.pop(0)
            self.voices.tie_to_blob(slot, blob.id)
            return
        # 2. Try to resurrect from the zombie pool (v3.2 logic)
        if self.voices.try_resurrect(blob):
            return
        # 3. Spawn into next IDLE slot as BLOB_ONLY
        self.voices.spawn_for_blob(blob)

    def on_blob_death(self, blob_id):
        v = self.voices.voices_by_blob.get(blob_id)
        if v is None: return
        if v.state == VoiceState.HYBRID and self._recently_played(v):
            self.voices.detach(v.slot_id)        # -> KEYBOARD_ONLY
        elif v.state == VoiceState.BLOB_ONLY:
            self.voices.zombify(v.slot_id)        # v3.2 zombie pool
        else:  # KEYBOARD_ONLY shouldn't happen here, but keep voice
            self.voices.detach(v.slot_id)
```

Step 9 — Main Loop

```
cd ~/lava && source .venv/bin/activate
python audible_lava.py

# Startup sequence (v4.0):
[Startup] Loaded lamp profile from lamp_profile.json
[Startup] Camera: 1280x960   MOG2 warm-up: 35 s
[Startup] Voice slots: 16   Default routings: 5
[Startup] OSC -> 127.0.0.1:57110
[Startup] Keyboard controller ready
Keys: TAB=edit/perf  1-0=select  ZSXDCVGBH=notes
      P/W/E/F/M=param family  B=claim  N=new empty  T=tie

[MOG2 warm-up 0s/35s] ...
+ Birth blob #4   -> spawn slot 4 (BLOB_ONLY)
+ Birth blob #7   -> resurrect slot 2 (was zombie 3.1s)
~ Selected slot 4
> Edit slot 4: filter cutoff 1200 -> 1850
+ B claimed slot 4 (HYBRID)
+ NOTE_ON  slot 4 note 60
+ Birth blob #11  -> tied to slot 9 (was KEYBOARD_ONLY, pending)
```

Phase 2 — External MIDI Controller

Phase 2 swaps the laptop keyboard for an assignable knob/button MIDI surface. The rest of the system does not change. The change is contained in three places: a new MIDIController adapter, a LearnMode mechanism in VoiceManager, and a small JSON file that persists the learn map.

MIDIController adapter

```
# controller.py (Phase 2 addition)

import mido
from collections import defaultdict

class MIDIController:
    def __init__(self, port_name=None, learn_map_path='learn_map.json'):
        self.port = mido.open_input(port_name or mido.get_input_names()[0])
        self.learn_map = self._load(learn_map_path)   # (channel,cc) -> (slot,param)
        self.learn_target = None  # (slot,param) waiting for next message

    def poll(self) -> list[ControllerEvent]:
        events = []
        for msg in self.port.iter_pending():
            key = (msg.channel, msg.control) if msg.type == 'control_change' else None
            if self.learn_target is not None and key is not None:
                self.learn_map[key] = self.learn_target
                self._save()
                self.learn_target = None
                continue
            if msg.type == 'note_on':
                # Map note -> slot select OR play note depending on bank
                ...
            elif msg.type == 'control_change':
                bound = self.learn_map.get(key)
                if bound:
                    slot, param = bound
                    events.append(ControllerEvent('SET_PARAM',
                        slot=slot, param=param, value=msg.value/127.0))
        return events
```

Recommended bank layout (MIDI Fighter Twister)

16 encoders × 4 banks = 64 logical controls. The recommended layout maps naturally to the Phase 1 edit-mode parameter families:

Bank	Encoders 1–4	Encoders 5–8	Encoders 9–12	Encoders 13–16
Bank A — Pitch & envelope	Slot select × 4 (with push-to-select)	ADSR of selected slot	Tune + bend + scale + chord	Pan + amp + slot solo + slot mute
Bank B — Wavetable & filter	Wave bank × 4	Wave pos + cutoff + res + fmode	Mod1 source × 4	Mod1 depth × 4 destinations
Bank C — Mod matrix	Mod 2..3 sources	Mod 2..3 depths	Mod 4..5 sources	Mod 4..5 depths
Bank D — Macro & transport	Macros × 4 (per-voice user-defined)	Global filter + reverb + delay + master	Bind / detach / tie / new empty	Tap tempo + record + transport

DeepMind 6 — Why It Is Not the Central Engine

The Roland/Behringer DeepMind 6 was originally chosen for the project because slow CC sweeps on its analogue ladder filter produce *timbral change*, not just amplitude or pitch change — which matches the 90–180 second blob timescale beautifully. A blob slowly rising can audibly modulate a real ladder filter in a way digital subtractive synthesis only approximates. For the fully realised v4.0 instrument, however, it has structural limits:

- **Monotimbral.** All six voices share one patch. The blob-as-agent concept needs each voice to potentially have its own timbre — a brass blob next to a granular blob next to a cello blob. The DeepMind cannot produce that simultaneously.
- **Six voices is not enough.** A modest lava lamp routinely shows 4–8 active blobs, and the v3.2 zombie pool can hold several more. v4.0's 16-slot pool will exceed six voices regularly.
- **No native MPE.** Per-voice CC isolation requires multitimbral channel splitting, which the DeepMind doesn't really do. SuperCollider provides per-voice CC for free.
- **Single fixed signal flow.** VCO → VCF → VCA. No flexible mod-matrix at the voice level, no wavetable scanning, no granular, no sampling — all of which the SC \mpeVoice provides.
- **DIN MIDI throughput.** With many simultaneous CCs across voices, the 31250-baud serial bus becomes the bottleneck. SC over OSC is effectively unconstrained.
- **Jitter.** scsynth is sample-accurate; DIN MIDI is not.

The right framing is: the DeepMind 6 remains a great single-timbre option that one or two voices in the SC orchestra can drive as a hardware *voice*, addressed by routing the assigned slots' notes/CCs out the Scarlett's MIDI port instead of (or alongside) the SC graph. That is a natural extension of the same VoiceManager — the SC node ID for the slot becomes a (channel, note, CC group) tuple sent to mido.

Technical Deep Dives

This section provides rigorous technical detail on each algorithm in the pipeline. The treatment assumes familiarity with OpenCV, numpy, and basic control theory.

1. Topological Data Analysis: Persistent Homology

Persistent homology, computed via GUDHI's CubicalComplex on a pixel grid, is the foundational significance filter for blob detection. Rather than detecting blobs by thresholding intensity or filtering by area — both of which conflate size with significance — persistence measures how prominent a blob is relative to its local surroundings across a continuous range of intensity thresholds.

The H0 (connected components) persistence diagram treats the greyscale image as a scalar function. As a threshold sweeps from high to low intensity, connected components 'born' at blob peaks 'die' when they merge with a brighter neighbouring region. A blob's persistence is the intensity gap between its peak (birth) and the saddle point where it merges (death). High persistence indicates a blob that is genuinely isolated from its surroundings — a topologically significant wax mass rather than a noise spike or gentle intensity gradient in the fluid.

Implementation note. The input is inverted (255.0 - small.astype(float64)) so OpenCV-style 'bright = foreground' becomes 'low value = foreground' matching GUDHI's sublevel-set sweep. Downscale to ~320×240 keeps CubicalComplex under ~40 ms on an M1 Mac. The persistence threshold must match the actual masked image range; on a wax-masked image, realistic maxima are 60–80, not 220.

2. Dense Optical Flow Mask Advection

Farneback dense optical flow estimates a displacement vector at every pixel between consecutive greyscale frames. Each track owns a binary mask initialised at birth from the blob's detection contour. Every frame the mask is remapped forward using the inverse-warp of the flow field:

```
map_x = x - flow[..., 0]
map_y = y - flow[..., 1]
advected = cv2.remap(mask.astype(float32), map_x, map_y, cv2.INTER_LINEAR)
new_mask = (advected > 0.4).astype(uint8)
```

The 0.4 threshold prevents the mask from drifting too far outside the wax boundary on ambiguous flow fields. Every REFINE_INTERVAL=5 frames the mask is 50/50 blended with the current detection contour, counteracting accumulated flow drift. A blob that briefly becomes semi-transparent and disappears from the persistence diagram does not fire a death event — the mask continues to exist and be advected forward, and when the blob reappears it is matched back by IoU overlap.

3. Wasserstein-Inspired Topological Matching

When multiple advected masks overlap a single detection, the system resolves the ambiguity using distance in the persistence diagram's birth-death plane rather than pixel space. The cost function:

```
cost = 0.75 * topo_dist(blob, det)   # hypot(d_birth, d_death)
     + 0.15 * pixel_dist(blob, det)  # centroid distance
     + 0.10 * hue_cost(blob, det)
# hue_cost: circular hue distance, plus direction penalty if
# both blob and det are non-TURNING and their directions disagree.
```

This is not full Wasserstein optimal transport — rather, a Wasserstein-inspired cost that prioritises topological feature space over ambient pixel space. Two blobs at similar positions but different thermal states (one rising, one falling, hence different birth–death coordinates) are distinguished correctly; naïve position-based Hungarian would assign them arbitrarily.

4. Physics-Informed Kalman with Buoyancy Dynamics

The constant-velocity Kalman model assumes blobs move at fixed velocity between frames. Lava lamp blobs do not: they decelerate to near zero at the top and bottom as buoyancy force reverses, and accelerate through the middle column. The v3.x Kalman adds a fifth state variable ay (vertical acceleration), governed by a mean-reverting dynamics model and initialised with a directional prior:

```
transitionMatrix =
[[1, 0, 1, 0, 0.0],   # x  += vx
 [0, 1, 0, 1, 0.5],   # y  += vy + 0.5 * ay   (Euler integration)
 [0, 0, 1, 0, 0.0],   # vx  = vx
 [0, 0, 0, 1, 1.0],   # vy += ay
 [0, 0, 0, 0, 0.9]]   # ay decays  (mean-reversion, buoyancy weakens mid-column)

ay_initial = {RISING: +0.15, FALLING: -0.15, TURNING: 0.0}[direction]
```

Measurement noise adapts based on y-position. Near the lamp top / bottom (within 12% of lamp height), measurement noise is 2.0 — the filter trusts its physics model more than the noisy detection. In mid-column it is 0.05 — detection dominates. This produces accurate predictions at the physically most important moments: turning points where identity ambiguity is highest because multiple blobs cluster at similar heights.

5. Birth Topology Fingerprinting and Re-identification

When a track is retired (coast_frames > MAX_COAST = 20), its fingerprint is stored in the ReidentificationBuffer rather than being discarded. The fingerprint records (birth_value, persistence, area, y_norm, mean_hue, direction) at the moment of loss. New detections not matched to any existing mask are checked against the buffer before a birth event fires. Fingerprint distance weights:

```
distance = 0.40 * |d_birth|       / 50
         + 0.25 * |d_persistence| / 50
         + 0.15 * |d_area|        / area_avg
         + 0.12 * circular_d_hue  / 60.0     # v3.2: was /25
         + 0.08 * |d_y_norm| * 2
         + 0.30 if directions disagree (and neither is TURNING) else 0
match_threshold = 0.35
```

v3.2 hue change. The old divisor of 25 meant a 70° hue flip contributed 0.12 × (70/25) = 0.34 to the distance, almost saturating the 0.35 threshold on its own. Under the new divisor of 60, the same flip contributes 0.14 — leaving headroom for topological and positional terms to carry the match through a thermal reversal. Records expire after 15 seconds, long enough to cover a transparency gap but short enough that a genuine death does not linger.

6. Thermodynamic Hue as Phase Indicator

Lava lamp wax undergoes a measurable hue shift across its thermal cycle. Wax near the heat source (about to rise) is hotter and slightly more yellow-orange; wax that has risen, cooled, and is falling is darker and more red. This hue shift encodes thermodynamic phase without requiring velocity history.

Circular mean hue is essential because OpenCV's H channel is periodic at 0 / 180 — a naïve arithmetic mean of H=2 and H=178 yields H=90 (green), whereas the correct circular mean is H=0 (red). The computation:

```
angles   = radians(hue_vals * 2.0)   # double-angle trick for [0,180] periodicity
sin_mean = mean(sin(angles))
cos_mean = mean(cos(angles))
mean_hue = degrees(atan2(sin_mean, cos_mean)) / 2.0 % 180.0
```

Direction drives three downstream behaviours: (a) Kalman ay is initialised with ±0.15 for rising / falling, 0 for turning. (b) Voice manager attack / release defaults: rising = 1.5 / 2.5 s, falling = 3.0 / 5.0 s, turning = 2.0 / 3.5 s — these are now *defaults at slot spawn*, freely overrideable by the player in Edit mode. (c) Fingerprint-buffer direction penalty (only when both sides are non-TURNING), making re-identification of

mismatched-thermal-state blobs less likely.

7. MOG2 Post-Warmup Freeze (v3.2)

MOG2 (Mixture of Gaussians v2) with default parameters continuously updates its background model using a learning rate automatically selected as 1 / min(frameCount, history). This is appropriate for surveillance scenes but catastrophic for lava lamp tracking: when wax decelerates to near zero at turning points, MOG2 absorbs the now-stationary material into the background within seconds. The detector loses the blob, coast_frames accumulates, and after MAX_COAST the track dies — while the wax is still physically there, just not moving fast enough to count as foreground.

The fix is a post-warmup learning rate of 1e-4 (effectively frozen). During warm-up (first 35 s), learning rate stays at the OpenCV default so MOG2 converges to a stable background. Afterwards, audible_lava.py passes learning_rate=1e-4 to detector.detect().

8. Voice Slot-Pool with Zombie Window (v3.2 → v4.0)

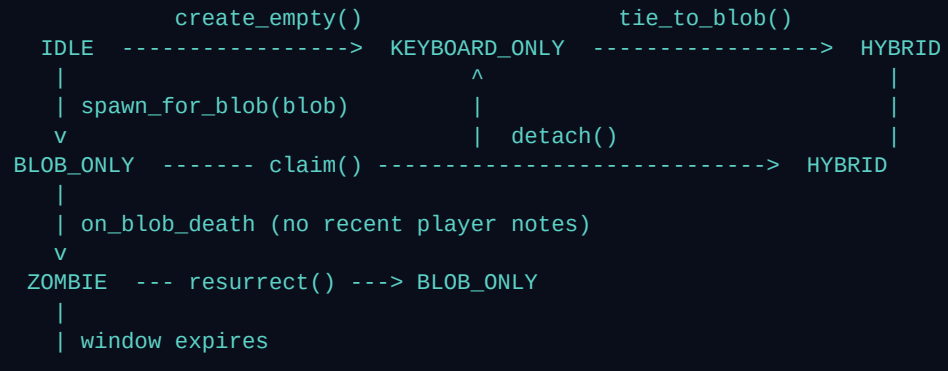
The zombie pool is the sonic analogue of the tracker's flow-advected mask: SC synth nodes are preserved across short-lived tracker gaps, giving each blob a continuous audible identity even when detection fails entirely. In v3.2 the zombie pool was the only voice-survival mechanism. In v4.0 the zombie pool is preserved as the lifecycle behaviour of *BLOB_ONLY-state voices when their blob disappears*; HYBRID and KEYBOARD_ONLY voices have different behaviours (they can outlive their blobs indefinitely) so the pool's role is reduced but unchanged.

Lifecycle (BLOB_ONLY → ZOMBIE → resurrected | expired). When the blob disappears, the slot transitions BLOB_ONLY → ZOMBIE; the SC node stays alive (gate=1) but its amp is multiplied by zombie_sustain_amp_scale (default 0.35), audibly softening without triggering release. Before spawning a fresh voice for any new blob, the BindingManager scans the zombie pool for a match within zombie_match_radius pixels (default: 15% of lamp height, floor 60 px) whose persistence is within a factor of 3× (sanity gate). On match, the zombie is popped, its blob_id is rewritten to the new ID, state returns to BLOB_ONLY, and smoothing morphs the new target values in seamlessly. Zombies older than zombie_window seconds (default 4) are released normally.

Why three filters (distance + persistence + direction)? Distance prevents a zombie in the top zone from being resurrected by a newly-rising blob at the base — different physical material, different voice. Persistence ratio prevents a low-persistence ghost detection from grabbing a high-persistence zombie slot. The fingerprint direction penalty already lives in the tracker's re-id, so the zombie pool doesn't duplicate that logic — but because the pool runs after fingerprint re-id has had its shot, it effectively functions as a second-chance matcher with different (centroid-based rather than topological) criteria. The two layers complement each other.

9. Voice-Slot Architecture (v4.0)

v4.0's central new abstraction. A Voice is no longer implicitly a 1:1 image of a blob — it is a slot in a fixed pool, holding full synth state, a routing matrix, and a lifecycle state that is independent of any given blob's existence. The state machine:



```
    v
  IDLE  (release SC node)
```

The state determines what events the slot accepts. IDLE accepts only spawn_for_blob or create_empty. BLOB_ONLY runs the routing matrix and forwards no keyboard input. KEYBOARD_ONLY accepts notes / edits but ignores blob updates. HYBRID accepts both — the player's notes set base values, the routing matrix adds modulation on top. ZOMBIE is a transient state with a timer.

10. MPE Routing Matrix (v4.0)

Every Voice carries a list of Routing tuples: (source, dest, depth, curve). Sources read blob CV features each frame; destinations write to the SC node's mod1..mod6 inputs and per-source depth controls. Default routings on BLOB_ONLY voices reproduce v3.2 behaviour:

```
DEFAULT_BLOB_ROUTING = [
    Routing(src='y_norm',      dest='pitch',     depth=+24, curve='LIN'),
    Routing(src='persistence', dest='amp',       depth=+1.0, curve='SQRT'),
    Routing(src='mean_hue',    dest='wave_pos',  depth=+1.0, curve='LIN'),
    Routing(src='elongation',  dest='res',       depth=+0.6, curve='LIN'),
    Routing(src='orientation', dest='pan',       depth=+1.0, curve='LIN'),
    # plus a discrete event:
    EventRoute(src='direction_change', dest='attack_release_preset'),
]
```

Sources include continuous features — y_norm, x_norm, vy, persistence, area, mean_hue, elongation, orientation, age — and discrete events — direction_change, merge, split. Destinations include the synth's mod inputs (mod1..mod6) and direct synth-param writes (pitch, cutoff, res, attack, release, amp, pan, wave_pos). Curves are pre-tabulated lookup arrays (LIN, LOG, EXP, SCURVE, SQRT) — cheap to evaluate and trivial for the user to extend.

11. Controller Abstraction (v4.0, Phase-2-ready)

All player input is funnelled through a Controller protocol that emits ControllerEvent dataclasses. The VoiceManager and BindingManager only see ControllerEvents; they do not know whether the source is a keyboard, a MIDI controller, an OSC stream from a tablet, or a future voice-to-MIDI bridge. The Phase 1 KeyboardController and Phase 2 MIDIController are interchangeable adapters.

Why this is worth the small upfront cost. The temptation in Phase 1 is to wire pygame keys directly to VoiceManager methods. That works, but Phase 2 then becomes a refactor instead of an add. A 60-line Controller protocol now makes Phase 2 a 100-line MIDIController plus a small JSON learn-map file. The same protocol can later host a Push 3 adapter, a TouchOSC layout, or a tablet UI without disturbing anything downstream of it.

12. Extended Coast Window (v3.2)

MAX_COAST controls how many consecutive frames a track can be unmatched before its death event fires. At 15 fps, the v3.0 value of 8 frames meant ~0.5 s of tolerance; the v3.2 value of 20 gives ~1.3 s. The effect is significant at turning points, where detection dropouts of several hundred milliseconds are typical while wax reverses through its pooling phase. A longer coast window also means the physics Kalman has more time to predict through the gap, and because its measurement noise is high near the top / bottom of the lamp, predict-only steps there are reasonably accurate. The zombie pool and MAX_COAST together bracket the problem: MAX_COAST covers gaps that the tracker can bridge, the zombie pool covers gaps it cannot, and v4.0's voice-slot independence covers the player-claimed case where the voice should outlive the blob entirely.

Appendices

Appendix A — Parameter Reference (v4.0)

Parameter	Current value	Effect of increasing
wax_hue_lo / wax_hue_hi	0 / 5 (with wrap to 175-180)	Narrowing captures more specific wax hue; reduces fluid bleed-through
wax_sat_min	245	Higher: reject less-saturated fluid pixels. Lower: catch dimmer wax
wax_val_min	52	Higher: reject dark background. Lower: catch cool / dim wax
blob_area_min	5020 px	Higher: discard small features. Lower: catch small nascent blobs
expected_persistence_mean	15	Higher threshold = fewer, more prominent blobs detected
expected_persistence_std	3.0	Wider std: threshold more tolerant of variation
pers_thresh (computed)	max(8, 15 - 0.5*3) = 13.5	Raise: stricter, fewer blobs. Lower: permissive, more blobs
DOWNSCALE	4	Raise to 6 for faster GUDHI (~8-15 ms); reduces localisation precision
TARGET_FPS	15	Higher: more CPU; more GUDHI calls per second
MAX_COAST	20 frames (v3.2)	Higher: blobs survive longer occlusion. Lower: faster death events
MOG2_WARMUP_SECS	35 s	Higher: more audio-silent warm-up; 30 s minimum for stable model
MOG2_POST_WARMUP_LR	1e-4 (v3.2)	Lower: freezer background; stationary wax preserved in foreground
smoothing	0.12	Higher: faster parameter response. Lower: slower, smoother evolution
rising_hue_threshold	from calibration	Lower: fewer blobs classified as rising
falling_hue_threshold	from calibration	Higher: fewer blobs classified as falling
iou_threshold (tracker)	0.25	Higher: stricter mask overlap required for match
topo_threshold (tracker)	35.0	Lower: tighter topological similarity required for match
reident_threshold	0.35	Lower: stricter fingerprint required for re-identification
hue fingerprint divisor	60 (v3.2, was 25)	Lower: hue mismatches more penalising; higher: re-id survives thermal reversal
zombie_window	4.0 s (v3.2)	Higher: longer grace period for resurrection
zombie_sustain_amp_scale	0.35 (v3.2)	Higher: louder hold. 0: silent. 1: no softening

Parameter	Current value	Effect of increasing
n_voice_slots (v4.0)	16	Higher: more polyphony / more concurrent player+blob voices. CPU cost ~linear.
default_routing_smoothing	0.10 (v4.0)	Per-routing smoothing alpha; lower = slower CC sweep, higher = snappier
blob_recent_play_window	5.0 s (v4.0)	How long a HYBRID voice 'remembers' player notes when blob dies; controls HYBRID → KEYBOARD_ONLY transition vs zombify

Appendix B — Troubleshooting

Symptom	Most likely cause	Fix
No blobs for first 35 s	MOG2 warm-up (expected)	Wait. Reduce MOG2_WARMUP_SECS if environment is stable.
Mass births (50+ / frame) after warm-up	persistence_threshold too low	test_detection.py; raise with + until count matches visual
Blobs disappear at turning points	MOG2 absorbing stationary wax	Confirm MOG2_POST_WARMUP_LR=1e-4 is applied after warm-up
IDs churn on every reversal	Fingerprint hue too heavily weighted	Confirm hue divisor is 60, not 25
Tracks die prematurely during brief occlusion	MAX_COAST too low	Confirm MAX_COAST=20; raise to 30 for very noisy lamps
Release-tail stacking during turns	Voice manager lacks zombie pool	Confirm BLOB_ONLY voices transition through ZOMBIE on blob death
Voices never resurrect from zombie pool	zombie_match_radius too tight	Lower zombie_match_radius_frac to 0.10 or raise zombie_min_match_radius
Zombie pool grows without bound	zombie_window too long	Lower zombie_window from 4.0 to 2.0; Z:N on HUD should oscillate
All blobs classified as TURNING	Hue thresholds cover full range	Re-calibrate drawing blobs at different temperatures
Wax colour mask empty	wax_sat_min too high for pale wax	Lower to 80–100 in lamp_profile.json
Re-identification not triggering	Fingerprint distance too strict	Lower reident_threshold from 0.35 to 0.5
Physics Kalman predicts wrong direction	ay prior not matching actual	Verify direction classification via test_detection.py overlay
SC not receiving OSC	SC not booted or wrong port	Check s.boot in IDE; verify SC_PORT=57110 matches
SC error: Input channel mismatch	USB webcam sample rate != SC rate	Set s.options.numInputBusChannels = 0
SC node ID collisions between runs	Stale nodes from previous session	Add /g_freeAll at VoiceManager startup
GUDHI hanging / KeyboardInterrupt	Grid too large (downscale=2)	Use downscale=4 or 6
Keyboard not responding (v4.0)	OpenCV window has focus instead of pygame window	pygame.event.pump() each frame; ensure pygame window stays foreground OR poll OS-level keyboard via pynput
Keyboard notes play on wrong slot (v4.0)	selected_slot not updated by SLOT_KEY	Confirm KeyboardController emits SELECT_SLOT before NOTE_ON; check VoiceManager.selected_slot in HUD
New blob always re-binds to old voice (v4.0)	BindingManager finds zombie before checking pending_keyboard_voices	Verify on_blob_birth checks pending_keyboard_voices first, then resurrect, then spawn

Symptom	Most likely cause	Fix
MIDI controller knobs do nothing (Phase 2)	Learn map empty / wrong MIDI port	Confirm mido.get_input_names() lists controller; trigger Learn mode and turn each knob

Appendix C — File Reference

File	Purpose	v4.0 changes
lamp_profile.py	Calibration dataclass + JSON I/O	Unchanged
calibration_gui.py	Freehand drawing calibration interface	Unchanged
blob_detection.py	MOG2 + GUDHI + contour moments + hue	Unchanged from v3.2 (detect() accepts learning_rate kwarg)
blob_tracker.py	Flow-advected tracker, Wasserstein, Kalman, fingerprint buffer	Unchanged from v3.2 (max_coast=20, hue divisor 60)
merge_tree.py	GUDHI merge tree with shape/hue leaf enrichment	Unchanged
voice_manager.py	Voice slot pool with full synth state + routing matrix	Major refactor — VoiceState enum, Voice dataclass, slot pool, routing matrix, controller-event API. v3.2 zombie pool preserved as BLOB_ONLY → ZOMBIE → BLOB_ONLY transition.
controller.py	Controller protocol + KeyboardController	NEW (v4.0). MIDIController stub for Phase 2.
binding_manager.py	Lifecycle bridge between BlobTracker and VoiceManager	NEW (v4.0).
routing.py	Routing tuple + curve lookup tables	NEW (v4.0).
audible_lava.py	Main loop: calibration, all pipeline stages, GUI overlay	Updated GUI overlay (voice IDs, slot HUD), Controller poll integration, BindingManager wiring. MAX_COAST=20, MOG2 freeze unchanged.
blob_synth.scd	SuperCollider SynthDefs	\mpeVoice replaces v3.2 \blobVoice. \blobAngle and \blobStrike preserved as optional auxiliary voices.
test_setup.py	Environment verification	Adds pygame, mido, python-rtmidi to checks
test_detection.py	Live detection calibration with mask windows	Unchanged
test_tracker.py	Live tracker with mask advection overlay	Unchanged
test_voices.py	Live voice-slot test: spawn, edit, claim, tie, detach	NEW (v4.0)

Appendix D — Call Graph Summary

```
audible_lava.main()
  +-- LampProfile.load()  (or calibration_gui.run_calibration())
  +-- BlobDetector(profile=...)
  +-- BlobTracker(profile=..., max_coast=20)
  +-- VoiceManager(profile=..., n_slots=16, ...)          <-- v4.0
  +-- BindingManager(voices)                              <-- v4.0
  +-- KeyboardController()                                <-- v4.0
  +-- (Phase 2) MIDIController(port=...)                  <-- v4.0
```

```
+-- loop:
     +-- detector.detect(frame, lr=...)   (lr -1 during warm-up, 1e-4 after)
     +-- tracker.update(blobs, frame)
     |    +-- _compute_flow -> advect masks
     |    +-- IoU + Wasserstein matching
     |    +-- Kalman predict / correct
     |    +-- coast / retire (>max_coast -> reident.record_loss)
     |    +-- reident.try_reidentify (hue divisor 60)
     +-- compute_merge_tree(...) (visualisation; phase-2 harmonic)
     +-- for event in tracker.events:                    <-- v4.0
     |    +-- binding.on_blob_birth/death/merge/split
     |       +-- voices.spawn_for_blob | resurrect | zombify | detach | tie
     +-- controller.poll() -> [ControllerEvent]          <-- v4.0
     |    +-- voices.dispatch(event):
     |       +-- select_slot / note_on / note_off / set_param
     |       +-- create_empty / tie_to_blob / detach
     |       +-- add_routing / remove_routing / learn
     +-- voices.process_frame(tracked_blobs)
     |    +-- for each voice in slot pool:
     |       +-- if BLOB_ONLY or HYBRID: apply_routing(voice, blob)
     |       +-- _smooth_and_send(voice)
     |    +-- _expire_zombies()
     +-- gui.draw_overlay(frame, blobs, voices)          <-- v4.0
```

Dynamical Systems Orchestration — Complete Build Guide, version 4.0 (MPE Edition). The lamp's wax keeps moving. The voice slot waits. The player decides whether to listen, take over, or play alongside.