

OSCILLA

User Manual

v0.4.9

Animated Graphic Score Framework

Rob Canning

oscilla.cc

2026

Contents

Preface	12
1 Getting Started	13
1.0.1 At a Glance	13
1.0.2 What Problems Does Oscilla Solve?	13
1.0.3 What Can You Actually Do With It?	13
1.0.4 Where Does Oscilla Sit in the Bigger Picture?	14
1.0.5 Who Is Oscilla For?	15
1.1 Oscilla Installation Guide	15
1.1.1 Option 1 Standalone Application (Recommended)	15
1.1.2 Option 2 Node.js / npm Installation (Advanced / Development)	16
1.1.3 Windows (Node.js Route)	16
1.1.4 Linux (Node.js Route)	16
1.1.5 macOS (Node.js Route)	17
1.1.6 Test the Demo Project	17
1.1.7 Inkscape Extension (Optional)	17
1.1.8 Troubleshooting	17
1.1.9 Quick Start	17
1.2 Workflow	18
1.2.1 1. Projects and Project Structure	18
1.2.2 2. Creating a New Project	18
1.2.3 3. Editing the Score in Inkscape	18
1.2.4 4. Adding Behaviour Using IDs	19
1.2.5 5. Opening and Switching Projects	20
1.2.6 6. Saving and Duplicating Projects	20
1.2.7 7. Exporting and Importing Projects (.oscilla)	20
1.2.8 8. Iteration Loop (Typical Use)	21
1.2.9 9. Rehearsal and Performance	21
1.2.10 Summary	21
1.3 Working with Parts	21
1.3.1 Approach 1: Separate Score Files	21
1.3.2 Approach 2: Stacked Layers	22
1.3.3 Combining Both Approaches	23
1.3.4 View Modes	23
1.3.5 Project Structure Summary	24
1.3.6 Quick Reference	24
2 Session Layer	25
2.1 Playhead & Playzone	25
2.1.1 Adjustable Playhead Position	25
2.1.2 Visibility Toggle	26
2.1.3 Appearance	26
2.1.4 Cue Interaction	26
2.1.5 Repeat Indicator	26
2.1.6 Edge Behaviour	27

2.1.7	Parts & Layers per-performer layer filtering	27
2.2	Performer Annotations	28
2.2.1	Overview	28
2.2.2	Entering Annotation Mode	29
2.2.3	Creating an Annotation	29
2.2.4	Editing an Annotation	29
2.2.5	Moving an Annotation	29
2.2.6	Keyboard Behaviour While Editing	29
2.2.7	Local vs Shared Annotations	29
2.2.8	Visibility	30
2.2.9	What Annotations Do <i>Not</i> Do	30
2.2.10	Notes on Use	30
2.2.11	Future Changes	30
2.3	Freehand Annotation	30
2.3.1	Overview	30
2.3.2	Entering Draw Mode	31
2.3.3	Drawing	31
2.3.4	Session Grouping	31
2.3.5	Toolbar	31
2.3.6	Select and Move	32
2.3.7	Eraser	32
2.3.8	Undo	32
2.3.9	Exiting Draw Mode	32
2.3.10	Keyboard Shortcuts	32
2.3.11	Local vs Shared	33
2.3.12	Persistence	33
2.3.13	Coordinates and Scaling	33
2.3.14	Stroke Appearance	33
2.3.15	What Freehand Annotations Do <i>Not</i> Do	34
2.3.16	Technical Notes	34
2.3.17	Future Directions	35
2.4	Markers	35
2.4.1	No SVG authoring required	35
2.4.2	What markers are for	35
2.4.3	Pre-performance planning	36
2.4.4	During rehearsal	36
2.4.5	Shared and local markers	36
2.4.6	Colors	36
2.4.7	Marker controls	36
2.4.8	Relationship to other timing tools	37
2.4.9	Navigation targets	37
2.4.10	Keyboard and transport navigation	38
2.4.11	The session layer	38
2.4.12	Typical workflow	38
2.5	Timers	39
2.5.1	Overview	39
2.5.2	Accessing the Timer	39
2.5.3	Countdown Controls	39
2.5.4	Swapping Timers	40
2.5.5	Countdown Sequences	40
2.5.6	Network Synchronization	42
2.5.7	Import & Export	42

2.5.8	Use Cases	43
2.5.9	Technical Notes	43
2.5.10	Keyboard Shortcuts	44
2.5.11	Summary	44
2.6	drag() Moveable Score Elements	44
2.6.1	Quick Start	44
2.6.2	Syntax	45
2.6.3	Combining with Animations	45
2.6.4	Nesting	45
2.6.5	With propagate	45
2.6.6	Use Cases	45
2.6.7	Persistence	46
2.6.8	Resetting Positions	46
2.6.9	Workflow	46
2.7	Live Console Live Coding & Inspection	46
2.7.1	Opening the Console	46
2.7.2	Panel Layout	47
2.7.3	Usage Examples	47
2.7.4	Keyboard Behaviour	48
2.7.5	Signal Monitor	48
2.7.6	Tips	48
2.7.7	Related	48
3	Cues	49
3.1	stop(...) Toggle Playback (On / Off)	49
3.1.1	Syntax	49
3.1.2	Behaviour	49
3.1.3	Related	49
3.2	pause() Temporarily Halt Playback	50
3.2.1	Syntax	50
3.2.2	Daisy-chaining with next(...)	50
3.3	speed()	50
3.3.1	Syntax	50
3.3.2	Parameters	50
3.3.3	Base Speed Multiplier (Preferences)	51
3.3.4	Two Types of Speed Cues	51
3.3.5	Position Awareness	51
3.3.6	Behaviour	52
3.3.7	Examples	52
3.3.8	Project Setup	52
3.3.9	Server Synchronization	52
3.3.10	Debugging	53
3.3.11	Notes	53
3.4	stopwatch (<i>with autostart support</i>)	53
3.4.1	Autostart (NEW)	54
3.5	metronome (<i>with autostart support</i>)	54
3.5.1	page() navigate pages or scrolling positions in the score	56
3.6	nav() Navigation and Structural Movement	58
3.6.1	Syntax	58
3.6.2	Parameters	58
3.6.3	Examples	58
3.6.4	UID rule	59
3.6.5	Notes	59

3.7	button() Clickable Buttons Inside the Score	59
3.7.1	Basic Syntax	59
3.7.2	Trigger	59
3.7.3	Style Block	59
3.7.4	Automatic Label Fallback	60
3.7.5	Positioning	60
3.7.6	Containers and Page Mode	61
3.7.7	Click Behaviour	61
3.7.8	Managing Buttons Programmatically	61
3.7.9	Summary	61
3.8	propagate() Group-Level Parameter Propagation	62
3.8.1	Purpose	62
3.8.2	Syntax	62
3.8.3	UID Handling	62
3.8.4	Example	62
3.8.5	Evaluation Rules	63
3.8.6	Non-Recursive	63
3.8.7	Execution Order	63
3.8.8	Supported Contexts	63
3.8.9	When to Use	63
3.8.10	When Not to Use	63
3.8.11	Notes	64
3.8.12	Version Compatibility	64
3.9	Reusable Blocks with use(name)	64
3.9.1	Defining a Reusable Block	64
3.9.2	Using the Block Elsewhere	64
3.9.3	Placement Modes	64
3.9.4	Behaviour and Guarantees	65
3.9.5	Summary	65
3.10	cue:text OscillaScore Text Cue Reference	65
3.11	1. Basic Syntax	65
3.12	2. Content Source	65
3.13	3. Unit Construction	66
3.14	4. Duration & Gap	66
3.15	5. Ordering	66
3.16	6. Looping	66
3.17	7. Crossfade System	67
3.18	8. Slot-Based Vertical Layout	67
3.19	9. Positioning	68
3.20	10. Examples	68
3.21	11. Interaction	68
3.22	12. Stopping All Text	69
3.23	13. Summary Table	69
3.24	End of Document	69
3.25	Audio Cues audio, audioPool, audioImpulse	69
3.25.1	1. audio(...) Play a Single File	69
3.25.2	2. audioPool(...) One-Shot Selection From a Folder	70
3.25.3	3. audioImpulse(...) Stochastic Repeating Process	71
3.25.4	Fade Values	72
3.25.5	Random Expressions	72
3.25.6	OSC Output	72

3.25.7 Stopping	73
3.25.8 Visual Overlays	73
3.25.9 Quick Examples	73
3.25.10 video()	74
3.26 synth() Web Audio Synth Cue	74
3.26.1 Oscilla “Native WebAudio Utility Synth”	74
3.26.2 Basic Usage	75
3.26.3 Wave Types	75
3.26.4 Lifetime and Duration	75
3.26.5 Amplitude Envelope (ADSR)	75
3.26.6 Chords	75
3.26.7 Patterned Parameters	76
3.26.8 Filters	76
3.26.9 Delay	76
3.26.10 Reverb	77
3.26.11 Glide (Frequency Only)	77
3.26.12 Interpolation Mode	77
3.26.13 Pan	77
3.26.14 OSC Mirroring	77
3.26.15 Stopping a Synth	77
3.26.16 Summary	78
3.27 cue:osc Discrete OSC Event Cue	78
3.27.1 BASIC FORM	78
3.27.2 EVENT BEHAVIOUR	78
3.27.3 VISUAL PARAMETER SOURCES (NORMALISED)	78
3.27.4 SEMANTIC PITCH VALUES	79
3.27.5 OPTIONAL ROOT OVERRIDE	79
3.27.6 PITCH PRIORITY	79
3.27.7 TRIGGERS	79
3.27.8 UID (OPTIONAL)	79
3.27.9 OSC OUTPUT FORMAT (AUTHORITATIVE)	79
3.27.10 USE WITH propagate()	80
3.27.11 DESIGN NOTES	81
3.27.12 SUMMARY	81
3.28 oscCtrl Continuous Control Lanes (OSC)	81
3.28.1 Basic Syntax	81
3.28.2 What gets sent	81
3.28.3 Path Semantics	82
3.28.4 Modes	82
3.28.5 Recommended Uses	82
3.28.6 Not Sent	83
3.28.7 Future Extensions	83
3.28.8 Troubleshooting	83
3.28.9 Summary	83
4 Animation	84
4.1 cue:scale Scaling Animation	84
4.2 BASIC FORMS	84
4.3 TRIGGERING	84
4.4 TRIGGER DELAY	84
4.5 PRESTART VISIBILITY	84
4.6 SEQUENCES	85
4.7 CONTINUOUS SCALING	85

4.8	UID Live Updates	85
4.9	FULL PARAMETER LIST	85
4.10	EXAMPLES	85
4.11	rotate() Rotation Cue	85
4.11.1	BASIC FORMS	86
4.11.2	TRIGGERING	86
4.11.3	TRIGGER DELAY tdelay:<seconds>	86
4.11.4	PRESTART VISIBILITY prestate:<show hide ghost>	86
4.11.5	SEQUENCE PARAMETERS	86
4.11.6	CONTINUOUS ROTATION	86
4.11.7	UID Live Updates	87
4.11.8	OSC OUTPUT (optional)	87
4.11.9	FULL PARAMETER LIST	87
4.11.10	EXAMPLES	88
4.11.11	Summary	88
4.12	cue:o2p Object-to-Path Animation	88
4.13	BASIC FORM	88
4.14	TIMING	88
4.15	PRESTART VISIBILITY	89
4.16	DIRECTION MODES	89
4.17	PATH SEGMENTS	89
4.18	ROTATION	89
4.19	LOOPING	90
4.20	OSC OUTPUT	90
4.21	TRIGGERING	90
4.22	TOUCH MODE (Interactive Control)	90
4.23	FADER GROUPS AND PRESETS	91
4.23.1	Grouping Faders	91
4.23.2	Launcher Bar	91
4.23.3	Preset Manager Panel	91
4.23.4	Preset Data	91
4.23.5	Console API	91
4.24	ROTATION HANDLES	92
4.24.1	hmode:<continuous limited>	92
4.24.2	rotrange:<degrees>	92
4.24.3	handle:<elementId>	92
4.25	LIVE UPDATE (uid:)	92
4.26	FULL PARAMETER LIST	92
4.27	EXAMPLES	93
4.28	cueTraverse Animate Objects Between SVG Points	93
4.28.1	Syntax	93
4.28.2	Parameter Reference	93
4.28.3	Examples	94
4.28.4	Notes	94
4.29	color() / colour() Colour Animation Cue	94
4.29.1	Quick Examples	94
4.29.2	Basic Syntax	95
4.29.3	Parameters	95
4.29.4	Colour Interpolation	97
4.29.5	Patterns	97
4.29.6	Examples	98

4.29.7	Named Colours Reference	98
4.29.8	Tips	99
4.29.9	See Also	99
4.29.10	<code>cue:fade()</code> fade or pulse an SVG element	99
4.30	<code>ui()</code>	100
4.30.1	Syntax	100
4.30.2	What it affects	100
4.30.3	Built-in UI toggle button	100
4.30.4	Parameters	101
4.30.5	Examples	101
4.30.6	Design notes	101
5	Control and Interaction	102
5.1	Oscilla Control Plane Architecture & Usage Guide	102
5.1.1	Overview	102
5.1.2	Quick Start	102
5.1.3	Signal Reference Syntax	103
5.1.4	Published Signals by Cue Type	103
5.1.5	Bindable Parameters	104
5.1.6	Architecture	104
5.1.7	Integration Guide	105
5.1.8	Parser Integration	107
5.1.9	Debugging	107
5.1.10	Common Patterns	108
5.1.11	Limitations & Notes	108
5.1.12	Summary Checklist	108
5.1.13	Version History	109
5.2	controlXY Multitouch XY Control & Score Animation System	109
5.2.1	Oscilla OSC controller design in Inkscape	109
5.2.2	Core Concept: Score Animation Through Spatial Control	109
5.2.3	Syntax	109
5.2.4	Parameters	110
5.2.5	Coordinate System	110
5.2.6	Rotation Control (Optional)	111
5.2.7	Published Signals	111
5.2.8	Setup Examples	112
5.2.9	Animation Workflow: Creating Score Choreography	113
5.2.10	Rotation Control Use Cases	114
5.2.11	Complete Animation Example: Verse-Chorus-Bridge	115
5.2.12	Binding to Score Elements: Creating Visual Animations	116
5.2.13	Advanced: Programmatic Animation via Console	117
5.2.14	Easing Functions	117
5.2.15	Complete DSL Action Reference	118
5.2.16	OSC Output (Bonus Feature)	119
5.2.17	Preset Panel UI	120
5.2.18	Storage & Persistence	120
5.2.19	CSS Styling	120
5.2.20	Complete API Reference	121
5.2.21	Compositional Patterns	121
5.2.22	Technical Notes	122
5.2.23	Summary	122
5.2.24	Quick Start Checklist	122
6	Tools	124

6.1	OSCILLA Inkscape Extension	124
6.1.1	Overview	124
6.1.2	Installation	124
6.1.3	First-Time Setup: Keyboard Shortcut	124
6.1.4	Components	124
6.1.5	Workflow	125
6.1.6	Using the Smart Cues Editor	125
6.1.7	Presets	126
6.1.8	Troubleshooting	127
6.1.9	Files	128
7	Developer Reference	129
7.1	Oscilla Complete System Documentation	129
7.1.1	Table of Contents	129
7.1.2	System Overview	129
7.1.3	Architecture	129
7.1.4	Server Components	130
7.1.5	Client Components	131
7.1.6	Cue System	132
7.1.7	Transport & Synchronization	133
7.1.8	OSC Integration	134
7.1.9	Project Structure	135
7.1.10	File Reference	135
7.1.11	Dependencies	138
7.1.12	Quick Reference: Adding a New Cue Type	138
7.2	Oscilla Architecture Overview	139
7.2.1	Purpose	139
7.2.2	High-Level Entry Points	139
7.2.3	Frontend Architecture (public/js/)	139
7.2.4	Runtime Data Flows	141
7.2.5	Server & Build System	141
7.2.6	Documentation	141
7.2.7	Immediate Technical Notes	142
7.2.8	Summary	142
7.3	Oscilla Multi-Client Visual Synchronization	142
7.3.1	Overview	142
7.3.2	Key Terms	142
7.3.3	How It Works	142
7.3.4	DOM Structure	142
7.3.5	Critical CSS Requirements	143
7.3.6	Coordinate Extraction CRITICAL	143
7.3.7	Core Functions	144
7.3.8	Server State server.js	145
7.3.9	File Reference	145
7.3.10	Debugging	146
7.3.11	Interaction Layer Elements	147
7.3.12	Coordinate System Summary	148
7.4	Oscilla Developer Guide: Adding a New Cue Type	148
7.4.1	Overview	148
7.4.2	Step 1: Define the Token (Parser)	149
7.4.3	Step 2: Add Parser Rule (Parser)	149
7.4.4	Step 3: Add AST Extraction (Parser)	149
7.4.5	Step 4: Create the Handler Module	150

7.4.6	Step 5: Register in Dispatcher	.153
7.4.7	Step 6: Register in Animation Assignment	.153
7.4.8	Common Parameters	.154
7.4.9	Pattern Support	.154
7.4.10	OSC Integration	.155
7.4.11	Hit Labels	.155
7.4.12	Testing Your Cue	.155
7.4.13	Checklist	.156
7.4.14	File Locations Summary	.156
7.4.15	Example: Complete color() Implementation	.156
7.5	Control Plane & Cross-Cue Modulation Developer Guide	.157
7.5.1	Architecture at a Glance	.157
7.5.2	File Map	.157
7.5.3	The Signal Pipeline in Detail	.158
7.5.4	How the Parser Creates Signal References	.159
7.5.5	Dual-Path Addressing	.159
7.5.6	Published Channels by Source	.160
7.5.7	Bindable Parameters (Consumer Side)	.161
7.5.8	How to Add a New Publisher	.161
7.5.9	How to Add a New Consumer	.162
7.5.10	OSC-In Target Registration (Not Yet Active)	.162
7.5.11	Debugging	.162
7.5.12	Known Limitations	.163
7.6	New Project Flow - Developer Documentation	.163
7.6.1	Overview	.163
7.6.2	Table of Contents	.163
7.6.3	Architecture Overview	.164
7.6.4	Splash Screen System	.164
7.6.5	New Project Creation Flow	.165
7.6.6	Project Browser Flow	.167
7.6.7	Preferences System	.168
7.6.8	Pin State Management	.169
7.6.9	File Locations	.170
7.6.10	API Endpoints	.171
7.6.11	Development Tips	.172
7.6.12	Summary	.172
7.7	Oscilla OSC System Developer Guide	.173
7.7.1	What is OSC in Oscilla?	.173
7.7.2	Architecture Overview	.173
7.7.3	Key Files	.173
7.7.4	Sending OSC	.174
7.7.5	Receiving OSC	.174
7.7.6	OSC Mute	.175
7.7.7	ParamBus Integration	.175
7.7.8	Control Router	.175
7.7.9	Visual Overlays	.175
7.7.10	Message Types	.176
7.7.11	Debugging	.176
7.8	drag() Developer Reference	.176
7.8.1	Architecture Overview	.176
7.8.2	File Location	.176
7.8.3	Module Exports	.176

7.8.4	Transform Layering	.177
7.8.5	ID Splitting	.177
7.8.6	Pointer Handling	.177
7.8.7	Data Flow	.177
7.8.8	Socket Protocol	.178
7.8.9	Server Persistence	.178
7.8.10	Persistence Key	.179
7.8.11	OSC Output	.179
7.8.12	Global Registry	.179
7.8.13	Console API	.179
7.8.14	Known Interactions	.180
7.8.15	Integration Points	.180
7.9	Marker Navigation Architecture	.180
7.9.1	Resolution Order	.180
7.9.2	Unified Nav Points	.181
7.9.3	Jump Mechanics	.181
7.9.4	Marker Coordinate Space	.182
7.9.5	Name Collisions	.182
7.9.6	Timer onComplete Integration	.182
7.9.7	Rehearsal Popup	.183
7.9.8	File Map	.183
7.10	Freehand Annotation Layer Developer Guide	.183
7.10.1	Architecture Overview	.183
7.10.2	Files Involved	.184
7.10.3	Key Design Decision: SVG over Canvas	.184
7.10.4	Coordinate System	.185
7.10.5	Scroll Integration	.185
7.10.6	Session Grouping	.185
7.10.7	Pointer Event Flow	.186
7.10.8	Data Model Integration	.186
7.10.9	Select and Drag	.187
7.10.10	Network Sync	.187
7.10.11	Eraser Implementation	.188
7.10.12	Draggable Toolbar	.188
7.10.13	Vector-effect: non-scaling-stroke	.188
7.10.14	Touch Disambiguation	.189
7.10.15	Point Thinning	.189
7.10.16	Path Smoothing	.189
7.10.17	Pressure Data	.189
7.10.18	Server Changes	.189
7.10.19	What is not implemented (yet)	.190
7.11	Developer Guide: Live Console	.190
7.11.1	Architecture	.190
7.11.2	Dependencies	.191
7.11.3	Files	.191
7.11.4	Integration	.191
7.11.5	Exports	.191
7.11.6	Execution Model	.191
7.11.7	Target Resolution	.192
7.11.8	Element Picker	.192
7.11.9	Signal Monitor	.192
7.11.10	Keyboard Isolation	.192
7.11.11	Index	.193

7.11.1 Future Extensions	.193
7.11.1 Related	.193
7.12 Playhead Offset Developer Guide	.193
7.12.1 Architecture Overview	.193
7.12.2 Files Involved	.194
7.12.3 Data Flow	.194
7.12.4 Score Positioning Logic	.195
7.12.5 Collision Pipeline (Unchanged)	.195
7.12.6 Drag Handle UI	.196
7.12.7 Playzone Tracking	.196
7.12.8 Public API	.196
7.12.9 Integration Points	.197
7.12.10 Testing Checklist	.197
7.12.11 Common Issues	.197

Preface

This manual documents Oscilla, an open-source browser-based framework for creating animated, cue-driven graphic scores. It is generated from the same source files as the online documentation at oscilla.kompot.si/docs.

The manual covers installation, score authoring, the cue DSL, animation, control systems, session features, and developer reference. For the most up-to-date version, consult the online documentation.

Chapter 1

Getting Started

Oscilla is an open-source framework for creating animated, cue-driven graphic scores that run in the browser. Scores are authored as SVG documents in Inkscape, with performance behaviour embedded directly into the graphic elements using a lightweight cue syntax. The result is a single document that is simultaneously a visual composition and an executable performance environment.

This section covers what Oscilla is, how to install it, and how to move from a blank SVG to a working score. Whether you are a composer exploring graphic notation, an improviser looking for networked coordination tools, or a performer encountering an Oscilla score for the first time, these pages will orient you within the system.

At a Glance

Oscilla is a framework for creating, performing, and sharing animated, cuedriven graphic scores in the web browser.

It sits between **notation**, **performance**, and **live electronic control**, allowing composers to design scores that move, react, and synchronize across performers devices while remaining readable, editable, and shareable.

If you can draw it in SVG, Oscilla can turn it into a timebased, interactive score.

What Problems Does Oscilla Solve?

Many contemporary scores:

- rely on **static PDFs** that cannot express temporal or interactive processes
- require **custom software** or fragile patches to realise
- separate **notation**, **rehearsal coordination**, and **electronic control** into different tools

Oscilla addresses this by providing:

- **notationcentric** workflow (the score is the interface)
 - **browserbased performance** (no installation for performers)
 - **explicit time, cue, and process control** embedded directly in the score
-

What Can You Actually Do With It?

1. Create Animated Graphic Scores

- Draw notation in Inkscape (or any SVG editor)
- Attach behaviours to graphical elements using simple, readable IDs
- Animate rotation, scaling, traversal, and visibility over time

The score becomes **dynamic**, without becoming opaque or procedural.

2. Coordinate Ensemble Performance

- Synchronise scores across multiple devices via a local server
- Use shared playheads, rehearsal marks, and cues
- Support fixed form, open form, or hybrid structures

Each performer sees the same temporal structure, while still interpreting locally.

3. Work With Composed Improvisation

Oscilla is particularly suited to:

- controlled improvisation
- modular or sectional forms
- scores that evolve differently on each run

Cues can:

- branch
- repeat
- pause
- trigger other events

This allows the score to **guide behaviour** rather than prescribe outcomes.

4. Control Electronics (Without Becoming a DAW)

Oscilla allows graphical elements in the score to function directly as **control structures** for electronics.

In practice this means:

- paths, shapes, and points drawn in Inkscape can be interpreted as **trajectories, timelines, or control curves**
- these structures can drive time, density, position, or other parameters
- the same graphical material serves both as **notation** and **control logic**

This creates a direct visual link between what performers see and how electronics behave.

Oscilla's built-in audio system is intentionally simple:

- basic sample playback
- simple synthesis
- time-aligned audio cues placed directly in the score

This makes it possible to realise works for **instruments and fixed media ("tape") entirely in the browser**, without external software or complex setup.

Where more advanced processing is required, Oscilla can:

- send OSC messages to SuperCollider, Pure Data, Max, lighting systems, or custom software
- act as a **notational and organisational layer** for complex electronic setups

Oscilla is not a replacement for audio environments it is the **score that coordinates them**.

5. Support Rehearsal and Collaboration

- Performers can add private or shared annotations
- Scores can be shared via URL
- No special software is required beyond a modern browser

This lowers barriers in rehearsal contexts and supports distributed ensembles.

Where Does Oscilla Sit in the Bigger Picture?

Oscilla belongs to a lineage of systems exploring the space between:

- graphic notation
- live electronics
- networked performance

What distinguishes Oscilla is that:

- the **score itself** is the primary interface
- everything runs using **open web technologies**

- composers work visually, not programmatically

Rather than inventing a new notation language, Oscilla **extends existing graphic practices into time, motion, and interaction.**

Who Is Oscilla For?

Oscilla is designed for:

- composers working with graphic or hybrid notation
- performers comfortable with nontraditional scores
- ensembles using electronics, live processing, or networked coordination
- researchers and educators exploring new scoreperformance relationships

It is not aimed at replacing traditional notation tools, but at **augmenting what scores can do.**

Oscilla Installation Guide

Oscilla is a browserbased system for creating and performing synchronized graphic scores. It can be run **either as a standalone desktop application** (recommended for most users) **or** via **Node.js** for development and advanced workflows.

Option 1 Standalone Application (Recommended)

Standalone builds are available for **Linux, macOS (Intel & Apple Silicon), and Windows.**

No Node.js, npm, or Git required.

Download

Download the latest release from:

<https://github.com/robcaning/oscilla/releases>

Choose the build for your operating system:

- **Linux** AppImage
- **macOS** .app (Intel or Apple Silicon)
- **Windows** .exe

Run

- **Linux:** Make the AppImage executable and run it

```
chmod +x Oscilla-*.AppImage
```

```
./Oscilla-*.AppImage
```
- **macOS:** Open the .app
 - You may need to rightclick **Open** the first time
- **Windows:** Doubleclick the .exe

Oscilla will start its local server automatically and open in your browser.

Open Oscilla

If it does not open automatically, visit:

<http://localhost:8001>

Requirements for Standalone Use

Tool	Purpose
Inkscape	Create and edit SVGbased score projects
Modern web browser	Chrome, Firefox, Safari, or Edge

Option 2 Node.js / npm Installation (Advanced / Development)

This option is intended for: - contributors and developers - users modifying Oscilla source code - custom server integrations

Requirements

Tool	Purpose
Node.js + npm	Run the Oscilla local server
Inkscape	Create and edit SVG score projects
Modern web browser	View and perform scores
Git (optional)	Clone and update the repository

Windows (Node.js Route)

1. Install Inkscape

<https://inkscape.org/release/windows/>

2. Install Node.js

<https://nodejs.org/en/download>

Choose the **Windows Installer (.msi)** and ensure: - **Add to PATH** is enabled

Verify:

```
node -v
npm -v
```

3. Get Oscilla

Option A Git

```
git clone https://github.com/robcanning/oscilla.git
cd oscilla
```

Option B ZIP 1. Download from <https://github.com/robcanning/oscilla/releases> 2. Extract the ZIP 3. Open PowerShell in the extracted folder

4. Install & Run

```
npm install
npm start
```

Open:

<http://localhost:8001>

Linux (Node.js Route)

1. Install Inkscape

```
sudo apt update
sudo apt install inkscape
```

2. Install Node.js (20.x recommended)

```
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs
```

Verify:

```
node -v
npm -v
```

3. Get Oscilla

```
git clone https://github.com/robcanning/oscilla.git
cd oscilla
```

Or download and extract the ZIP from the releases page.

4. Install & Run

```
npm install
npm start
```

Open:

<http://localhost:8001>

macOS (Node.js Route)

1. Install Inkscape

<https://inkscape.org/release/macos/>

2. Install Node.js

<https://nodejs.org/en/download>

Verify:

```
node -v
npm -v
```

3. Get Oscilla

```
git clone https://github.com/robcaning/oscilla.git
cd oscilla
```

Or download and extract the ZIP from the releases page.

4. Install & Run

```
npm install
npm start
```

Open:

<http://localhost:8001>

Test the Demo Project

Once Oscilla is running:

<http://localhost:8001/?project=helper-score>

You should see the demo score **helperscore**.

Inkscape Extension (Optional)

Oscilla includes an optional **Inkscape extension** to assist with: - creating score templates - inserting cue elements - managing IDs and layers

See the documentation for installation and usage details.

Troubleshooting

Problem	Solution
App does not start	Ensure port 8001 is free
Browser shows blank screen	Check browser console for errors
npm not found	Reinstall Node.js
Inkscape SVG not loading	Ensure SVG is saved as Plain SVG
Port conflict	Change port in <code>server.js</code>

Quick Start

Download the latest release from:

<https://github.com/robcaning/oscilla/releases>

OR

Clone the repository and start the local server:

```
git clone https://github.com/robcaning/oscilla.git
cd oscilla
npm install
npm start
```

Then open your browser at:

<http://localhost:8001>

The demo project can be launched directly from the splash screen.

For full setup instructions (including Inkscape, Node.js, and platform-specific installation), see the [Full Installation Guide](#).

title: workflow layout: docs_layout.njk

Workflow

This page describes the current OscillaScore workflow, including **creating, loading, saving, and sharing projects**. It reflects the newer project lifecycle tools available from the UI (New Project, Save As, Import / Export).

The core idea remains the same: **draw in Inkscape, perform in the browser**, but project management is now handled explicitly by Oscilla.

1. Projects and Project Structure

All Oscilla projects live inside:

```
public/scores/
```

Each project is a self-contained folder. At minimum, a project contains:

```
myProject/
├─ score.svg
```

This is enough for a project to load and run.

A typical project created by Oscilla looks like:

```
myProject/
├─ score.svg           # main scrolling or hybrid score
├─ preferences.json    # project preferences (auto-generated)
├─ pages/              # optional page-mode SVGs
├─ audio/              # optional local audio files
├─ texts/              # optional external text cues
└─ videos/             # optional local video files
```

You normally do **not** create this structure by hand anymore Oscilla does it for you.

2. Creating a New Project

Recommended method (UI)

1. Open Oscilla in your browser:

```
http://localhost:8001
```

2. Open the **hamburger menu** (top-right)

3. Choose:

```
File → New Project...
```

4. Enter a project name

Oscilla will:

- create a new project folder inside `public/scores/`
- copy the **project template** (including `score.svg` and helper files)
- load the project automatically

After creation, Oscilla shows a short hint explaining where the score file lives on disk and that it should be edited in Inkscape.

3. Editing the Score in Inkscape

Oscilla does not provide a built-in score editor. **Inkscape is the primary authoring tool.**

1. Open Inkscape

2. Open:

```
public/scores/myProject/score.svg
```

3. Draw shapes, text, paths, and layout the score visually

4. Save the file

Refresh the browser and the changes appear immediately.

Page size conventions

Oscilla expects specific page dimensions depending on score type:

Score Type	Width	Height	Use Case
Scrolling	~40000px	1024px	Continuous horizontal scores
Paged	1366px	1024px	Discrete pages (tablet-friendly)

Set this in Inkscape via:

File Document Properties Page Size

Notes:

- 1024px height is optimized for tablet landscape display
- For scrolling scores, width directly affects playback duration
- Playback speed can be adjusted in `preferences.json`

4. Adding Behaviour Using IDs

Behaviour is encoded by editing SVG element IDs (via the XML Editor):

- **Ctrl + Shift + X** (Inkscape)

Examples:

- `pause(dur:12, count:true)`
- `audio(file:intro_theme.wav)`
- `o2p(path:p01)`
- `scale([1,1.3,1])`

Using the Oscilla Inkscape Extension (recommended)

Editing complex IDs manually can be errorprone. Oscilla therefore provides an **Inkscape extension** that makes working with cue and animation IDs significantly easier.

The extension:

- provides structured input fields instead of raw text editing
- helps avoid syntax errors
- exposes common cue and animation parameters
- speeds up authoring for larger or more complex scores

The extension is **optional**, but strongly recommended for regular use.

Documentation and installation instructions:

https://robcaning.github.io/oscilla/docs/inkscape_extension/

You can freely mix workflows:

- use the extension for most edits
- finetune or experiment directly in the XML editor when needed

5. Opening and Switching Projects

You can open projects in several ways.

From the loader

1. Visit:

`http://localhost:8001`

2. Select a project from the loader dialog

Direct URL

`http://localhost:8001/?project=myProject`

From the menu

Use:

File → Projects

to switch between existing projects without restarting the server.

6. Saving and Duplicating Projects

Save As

Use:

File → Save Project As...

This:

- duplicates the current project
- assigns a new project name
- preserves all SVGs, media, and preferences

The new project is loaded automatically.

This is the recommended way to create variations or rehearsal versions.

7. Exporting and Importing Projects (.oscilla)

Oscilla supports bundling a complete project into a single file for sharing.

Export

File → Export Project (.oscilla)

This creates a .oscilla file containing:

- `score.svg`
- `preferences.json`
- pages, audio, texts, and videos (if present)

You can send this file to collaborators or archive it.

Import

File → Import Project (.oscilla)

When importing:

- you choose a **new project name**
- the project is unpacked into `public/scores/`
- the project is loaded automatically

This allows projects to be shared without manually copying folders.

8. Iteration Loop (Typical Use)

In practice, work looks like this:

1. Edit `score.svg` in Inkscape
2. Save
3. Refresh the browser
4. Test cues, timing, and layout
5. Repeat

Oscilla never locks files the filesystem is always the source of truth.

9. Rehearsal and Performance

For rehearsal and performance:

- open the same project on all devices
- use cues for timing and navigation
- optionally synchronize multiple clients

Each performer reads from the same authored score, rendered locally in their browser.

Summary

- Projects are **folders**, not opaque files
- Inkscape is the **authoring environment**
- Oscilla handles **loading, duplication, import/export, and playback**
- `.oscilla` files are for **sharing and archiving**, not daily editing

This separation keeps the system transparent, hackable, and robust.

Working with Parts

Oscilla supports two approaches to per-performer parts: **separate score files** and **stacked layers within a single SVG**. Both are configured in the **Parts** tab of the preferences dialog and stored per-browser, so each performer on a shared network can independently choose their own view.

The two approaches can be combined. A project might use separate score files for radically different parts (e.g. a conductor score and a performer score) while also using layers within each file for finer distinctions (e.g. violin I and violin II sharing a string parts score).

Approach 1: Separate Score Files

Each part is authored as an independent scrolling SVG. All files live in the project root alongside `score.svg`:

```
myProject/
├── score.svg                # full score (always present)
├── score-part-violin.svg    # violin part
├── score-part-violola.svg   # viola part
├── score-part-cello.svg     # cello part
├── score-conductor.svg      # conductor view
├── preferences.json
└── pages/
```

Naming Convention

Files must match the pattern `score*.svg`:

- `score.svg` the main or full score (always listed first)
- `score-part-violin.svg` a part score
- `score-conductor.svg` a conductor score
- `score-electronics.svg` an electronics part

The prefix `score` is required. Everything after it becomes the display label in the preferences dialog. Hyphens become spaces: `score-part-violin.svg` appears as **part violin**.

When to Use Separate Files

Separate files are appropriate when parts differ substantially in content, layout, or length:

- different notation for different instruments
- a conductor score with rehearsal annotations that players should not see
- an electronics part showing control data rather than musical notation
- parts at different zoom levels or page sizes
- simplified parts for less experienced performers

Authoring

Each file is a standard Inkscape SVG. All the usual cue syntax works identically cues are embedded in element IDs just as in `score.svg`. The scrolling playhead, transport, timing, and synchronisation all work the same regardless of which score file is loaded.

One important consideration: if you use cue IDs that reference specific elements (e.g. `nav(scroll@A)`), the rehearsal marks referenced must exist in the currently loaded score file for that performer. If mark **A** exists in `score.svg` but not in `score-part-violin.svg`, navigation cues targeting it will not work for a performer who has selected the violin part.

Performer Selection

Each performer selects their score file in **Preferences > Parts > Score**. The selection is stored in the browser's `localStorage` and takes effect on the next project load. This means each performer on a different device (or browser) can independently choose which score file they see, while all remaining synchronised via the shared transport.

Approach 2: Stacked Layers

A single `score.svg` contains all parts as Inkscape layers. Each performer can highlight their own layer and dim the others.

```
score.svg
├─ [layer] part-violin
├─ [layer] part-violola
├─ [layer] part-cello
├─ [layer] shared-notation    ← always visible (no "part" in name)
└─ [layer] background        ← always visible (no "part" in name)
```

Naming Convention

Oscilla scans Inkscape layers (groups with `inkscape:groupmode="layer"`) and includes only those whose label contains the word **part** (case-insensitive). Layers without “part” in their name are always visible to all performers and are not affected by the filter.

This means you can freely mix filterable and non-filterable layers:

Layer Name	Filterable?	Why
part-violin	Yes	contains “part”
part-cello	Yes	contains “part”
conductor-part	Yes	contains “part”
shared-notation	No	no “part” in name
background	No	no “part” in name
grid	No	no “part” in name

Creating Layers in Inkscape

1. Open `score.svg` in Inkscape
2. Open the Layers dialog: **Layer > Layers** (or Shift+Ctrl+L)
3. Create a layer for each part, naming it with “part” in the name

- 4. Draw each instruments notation on its own layer
- 5. Put shared elements (barlines, rehearsal marks, time signatures, background) on a layer without “part” in the name
- 6. Save

When to Use Layers

Layers work well when parts share the same horizontal timeline and general layout but differ in vertical content:

- ensemble scores where all parts scroll at the same speed
- scores where performers benefit from seeing each others parts in context
- pieces where the spatial relationship between parts is meaningful
- situations where you want a single file to maintain

Performer Selection

Each performer selects their layer in **Preferences > Parts > My Part**. The **Other Parts** slider controls how visible the remaining part layers are (0% = invisible, 100% = fully visible). The default is 15% enough to see the shape of other parts without them dominating the view.

Like score file selection, layer preferences are stored per-browser and apply immediately.

Combining Both Approaches

A project can use both methods simultaneously. For example:

```
myProject/
├─ score.svg                # full score with all layers
│   └─ [layer] part-violin-I
│   └─ [layer] part-violin-II
│   └─ [layer] part-violola
│   └─ [layer] shared
├─ score-part-cello.svg     # cello has its own score entirely
├─ score-conductor.svg     # conductor view with annotations
└─ preferences.json
```

Here the string players share a layered score (each highlighting their own part), while the cellist and conductor each have dedicated score files.

The Parts tab in preferences adapts to what is available: if multiple score files exist, it shows the Score dropdown. If the currently loaded score contains layers with “part” in the name, it shows the layer filter controls. If both are present, both controls appear.

View Modes

The **Default View** setting in **Preferences > Settings** interacts with parts:

Mode	Behaviour
hybrid (default)	Loads the selected score file for scrolling. Page navigation also available via cues or the mode toggle.
scroll	Same as hybrid. Score file is required.
page	Starts in page mode. If a score file exists, the mode toggle allows switching to scroll view. If no score*.svg files exist, pure page mode with no scroll option.

In hybrid and scroll modes, the selected score file determines what scrolls. In page mode, score file selection is irrelevant unless the performer switches to scroll view.

Project Structure Summary

myProject/	
— score.svg	# main scrolling score (required for scroll/hybrid)
— score-part-*.svg	# optional per-performer score files
— preferences.json	# project settings
— pages/	# page-mode SVGs (optional)
— home.svg	
— section-a.svg	
— audio/	# audio files (optional)
— texts/	# text cue content (optional)
— videos/	# video files (optional)

Quick Reference

What you want	What to do
Different notation per instrument	Create separate score-part-*.svg files
Same notation, highlight own part	Use Inkscape layers with “part” in the name
Both	Combine: layers inside score files, plus separate files for very different parts
Shared elements visible to all	Put them on a layer without “part” in the name
Performer chooses their part	Each performer opens Preferences > Parts
Settings persist across sessions	Automatic stored in browser localStorage
Settings sync across devices	They do not each browser has its own selection, which is the intended behaviour

Chapter 2

Session Layer

The session layer encompasses features that operate on top of the score during rehearsal and performance. These are per-performer tools that require no SVG authoring they work with any score, or even with no score at all.

The playhead provides a configurable look-ahead window that performers can adjust based on notation density and personal preference. Instrumental parts allow performers to highlight their own material while keeping the ensemble context visible. Annotations provide a text-and-audio compositional layer created directly in the browser, from simple rehearsal notes to executable audio triggers. Freehand drawing, drop markers, and sequence timers offer further ways to sketch structure onto a shared timeline without interrupting the flow of a rehearsal or performance.

Together, these tools form a flexible interaction surface that sits between the authored score and the live performance moment.

Playhead & Playzone

The playhead is the vertical line that marks the current reading position in a scrolling score. As playback advances, the score moves beneath the playhead and cues are triggered when their bounding box crosses the playhead line.

The playzone is a translucent highlight region centred on the playhead. It gives performers a wider visual context of “where we are now” and can be styled per project.

Adjustable Playhead Position

By default, the playhead sits at the horizontal centre of the screen. This means roughly equal amounts of upcoming and past notation are visible at any moment. But different musical situations call for different reading positions, and performers often have strong preferences about how much lead time they need.

Why Move the Playhead?

Preparation time. In a densely notated score with precise rhythmic cues or complex extended technique instructions, performers need to see material well in advance. Shifting the playhead to the left say 25% or 30% from the screen edge gives significantly more screen space to the right, meaning upcoming notation stays visible for longer before it reaches the intersection point. This extra read-ahead time can be the difference between a confident entrance and a scrambled reaction.

Loose timing and sustained material. In freer passages where timing is approximate and material might be held, stretched, or revisited, a centred or even right-of-centre playhead keeps the current material on screen longer. Performers who are sustaining a texture or interpreting a passage freely may not want the notation they're working with to disappear off the left edge too quickly. Keeping the playhead further right means the “now” zone lingers.

Ensemble coordination. In networked performance where multiple performers share the same scrolling score, different instrumentalists may need different amounts of preparation depending on their part. A percussionist reading rapid triggering cues might want maximum look-ahead, while a vocalist sustaining long tones might prefer a centred view. Since the playhead offset is a per-device setting, each performer can adjust independently without affecting synchronisation the underlying score position stays locked across all clients.

Audience projection. When a separate display shows the score to an audience, a centred playhead often makes the most visual sense, giving equal context on either side. Meanwhile the performers tablets might

have the playhead pulled left for more look-ahead. This is a natural configuration in live performance setups.

Rehearsal vs. performance. During rehearsal, a performer might want a centred playhead for a balanced overview while learning the piece. In performance, they might shift it left once they know the material and need that extra preparation window for confident execution. The drag handle makes this a quick adjustment between runs.

Using the Drag Handle

1. **Hover** near the playhead line. A small grip handle and a lock icon appear at the top of the line.
2. **Click the lock icon** to unlock the playhead position.
3. **Drag the grip handle** left or right to reposition. The playzone follows automatically.
4. **Click the lock icon** again to lock the position and prevent accidental movement during performance.

The offset is saved per device and persists across page reloads.

Using Preferences

Open the project preferences dialog (hamburger menu) and look for **Playhead Position %** in the Appearance section. The slider ranges from 10% (far left) to 90% (far right), with 50% being the default centre position.

The preferences value is saved per project on the server, while the drag handle saves per device in the browser. The drag handle setting takes priority on load, since screen size and performer preference are inherently local.

Visibility Toggle

The playhead and playzone can be cycled through four visibility states using the toggle button in the transport controls (bottom-right grid):

- **Both** playhead line and playzone visible (default)
- **Playhead only** just the line
- **Playzone only** just the highlight region
- **None** both hidden

Appearance

Playhead and playzone colours and the playhead line width can be configured in the project preferences dialog under **Appearance**:

- **Playhead Color** colour of the vertical line
- **Playzone Color** colour of the highlight region (supports transparency via hex-with-alpha)
- **Playhead Width** thickness of the line in pixels

Cue Interaction

Cues are triggered when the playhead crosses their left edge during forward playback. The collision system reads the playhead elements actual screen position, so adjusting the playhead offset does not break cue triggering cues always fire at the playhead line, wherever it is on screen.

This applies to all cue types: navigation, audio, synth, OSC, animation triggers, pause, stop, and continuous control lanes (oscCtrl).

Repeat Indicator

When a cueRepeat region is active, the playhead line turns red and a count box appears near the bottom showing the current repeat iteration. Clicking the count box cancels the repeat and resumes normal playback.

Edge Behaviour

At the very start and end of the score, the playhead departs from its configured screen position to avoid showing empty space beyond the score boundaries:

- **Near the start:** the scores left edge stays at the screens left edge and the playhead moves rightward from the left toward its configured offset.
- **Near the end:** the scores right edge stays at the screens right edge and the playhead moves rightward from its configured offset toward the right.

During the main body of the score, the playhead stays fixed at its offset position and the score scrolls beneath it.

Parts & Layers per-performer layer filtering

Allows each performer to select their own instrumental part from the Inkscape layers in a score, dimming other parts to a configurable opacity. This is a client-side, per-browser preference each device remembers its own part selection independently.

Authoring in Inkscape

Organise your score using Inkscape's layer system. Each layer represents one instrumental part or section of the ensemble. Layer names become the labels shown in the Oscilla preferences dialog.

Layer panel in Inkscape:

```
Part: Violin      <-- contains "part", detected
Part: Viola       <-- contains "part", detected
Part: Electronics <-- contains "part", detected
Background       <-- no "part", always visible
Shared           <-- no "part", always visible
```

Each layer corresponds to a `<g>` element in the SVG with Inkscape's layer attributes:

```
<g inkscape:groupmode="layer"
  inkscape:label="Part: Violin"
  id="layer1">
  <!-- score content for violin -->
</g>
```

The only naming requirement is that the layer label contains the word “part” (case-insensitive). Everything else about the name is freeform. Layers without “part” in their name are never filtered, making them suitable for shared notation, grids, or background elements.

Layer detection

When a score is loaded, Oscilla scans the SVG for all `<g>` elements with `inkscape:groupmode="layer"` whose label contains the word “part” (case-insensitive). Layers without “part” in their name are left untouched by the filter and always render at their authored visibility.

This means layers like “Part 1 Violin”, “viola part”, or “PART: Electronics” are all detected, while layers named “Background”, “Grid”, or “Shared” are excluded and remain fully visible regardless of the performers selection.

Preferences UI

The “Parts” section appears automatically in the Preferences dialog when a score contains Inkscape layers. It provides two controls:

Control	Description
My Part	Dropdown listing all detected layers, plus “All (no filter)”
Other Parts	Opacity slider (0100%) for non-selected layers

Changes apply immediately as a live preview. The setting is also saved when the Preferences dialog Save button is pressed.

Behavior

- Selecting a part sets that layer to full opacity and all other layers to the configured “Other Parts” opacity.
- When a part is selected, all layers are forced to `display:inline`, overriding any Inkscape visibility state. This ensures hidden layers become visible (at reduced opacity) so the performer sees the full score context.
- Selecting “All (no filter)” restores all layers to their authored visibility and clears any opacity overrides.
- The opacity slider at 0% fully hides other parts. At 100% all parts are equally visible (useful for rehearsal or conducting).

Storage

Layer filter preferences are stored in `localStorage`, keyed per project:

```
oscilla_layerFilter_<projectName>
```

This means each browser or device maintains its own part selection. A violinists tablet remembers “Part: Violin” while the cellists tablet remembers “Part: Cello”, even when viewing the same score.

The preference is not saved to `preferences.json` on the server and does not affect other connected clients.

Authoring tips

- Include the word “part” in any layer name that should appear in the performers dropdown (e.g. “Part: Violin I”, “Electronics Part”, “Percussion part”). The match is case-insensitive.
- Place shared notation (rehearsal marks, structural cues, tempo markings) on a layer whose name does not contain “part” it will always remain visible regardless of the performers selection.
- Layer ordering in Inkscape determines rendering order in the SVG. Place shared/background layers below part layers.
- Cue elements (animations, navigation, audio triggers) work normally regardless of layer filtering opacity changes are purely visual.
- If a layer is hidden in Inkscape (`display:none`), it will remain hidden until a performer actively selects a part, at which point all layers become visible at their respective opacities.

Technical details

The feature is implemented in `layerFilter.js` with integration points in `projectLoader.js` (layer scanning on score load) and `oscillaPreferences.js` (UI section in preferences dialog).

Layer detection uses both namespace-aware attribute access (`getAttributeNS`) and a plain attribute fallback (`getAttribute("inkscape:groupmode")`) to handle differences in how browsers resolve XML namespaces when SVG is embedded in an HTML document.

Notes

- Only Inkscape layers whose label contains “part” (case-insensitive) are detected. Other layers and plain SVG `<g>` groups are not affected by the filter.
- The filter applies to scroll mode. Page mode loads individual SVG files which may have their own layer structure.
- All animations, cue triggers, and OSC output continue to function normally on filtered layers the filter only affects visual opacity.

Performer Annotations

Overview

Performer Annotations allow performers to add **personal notes directly onto the score view** while working with Oscilla.

Annotations are intended for: - reminders - listening cues - coordination notes - interpretive decisions - rehearsal observations

Annotations **do not change the score** and **do not affect playback or cues**.

They exist as a separate, optional layer that can be shown or hidden at any time.

***Note:** Because annotations can be placed freely in time and space, Oscilla can be used in a **minimal, annotation-only mode** (without an SVG score). In this case, annotations function as a very lightweight textual score. This mode is intentionally limited, but can be useful for sketches, instructions, or rehearsal frameworks.*

Entering Annotation Mode

1. Click the **pen icon** in the toolbar
2. The cursor changes to indicate annotation mode
3. Click anywhere on the score area to add a new annotation

Annotation mode automatically exits after saving a note.

Creating an Annotation

When you click the score area in annotation mode:

1. A text editor appears
2. Enter your note (multi-line text is supported)
3. Choose whether the note is:
 - **Local** visible only on your device
 - **Shared** visible to other connected clients
4. Click **Save**

The annotation appears as: - a small dot (anchor) - a text label displaying the note

Editing an Annotation

- Click **either the dot or the text label** to open the editor
- You can:
 - edit the text
 - change Local / Shared status
 - adjust font size
 - delete the annotation

Click **Save** to apply changes.

Moving an Annotation

- Click and **drag the text label** to reposition the annotation
- Release to confirm the new position
- The position is saved automatically

The dot remains visually tied to the text.

Keyboard Behaviour While Editing

While typing in the annotation editor: - All global keyboard shortcuts are temporarily disabled - This prevents accidental playback or navigation - Press **Escape** to close the editor

Normal keyboard controls resume once the editor is closed.

Local vs Shared Annotations

Local

- Stored only in your browser
- Not visible to others
- Suitable for personal rehearsal notes

Shared

- Broadcast to other connected clients
- Read-only for other performers
- Useful for ensemble coordination

Shared annotations do not override local ones.

Visibility

Annotations: - can be globally shown or hidden - move with the score when scrolling or paging - remain aligned to the score content

They never affect timing, cues, or navigation.

What Annotations Do *Not* Do

Annotations: - do not trigger playback - do not alter the score - do not affect synchronization - do not replace cues or instructions

They are purely informational.

Notes on Use

Annotations are best used for: - marking sections to listen closely - noting transitions or handovers - tracking rehearsal decisions - personal memory aids

They are intentionally lightweight and non-authoritative.

Future Changes

This feature is evolving. Possible future additions include: - filtering annotations by author - export / import of annotation files - visibility modes (rehearsal vs performance)

Freehand Annotation

Overview

The freehand annotation layer allows performers and composers to mark up the score directly in the browser using finger, stylus, or mouse. Strokes are captured as vector paths anchored in score coordinates, persisted locally, and optionally synchronised across all connected clients via WebSocket.

Freehand annotations serve the same role as ink on a printed part: rehearsal marks, breath indicators, dynamic contours, circled passages, conductor gestures, or any other visual markup a musician might add during preparation or performance. They coexist with the SVG score and the structured annotation layer without affecting playback, cues, or timing.

Strokes drawn in a single session are automatically grouped. Groups can be selected, repositioned, deleted, and have their network scope toggled between local and shared.

Three Coexisting Layers

SVG Score Layer	Annotation Layer	Freehand Layer
Authored in Inkscape	Text + audio triggers	Freehand strokes
Embedded DSL cues	Click-to-execute	Visual markup only
Fixed at authoring time	Editable in browser	Editable in browser
Playhead-driven	Performer-driven	Performer-driven

All three layers move together with the score during scrolling and playback. None interfere with each other.

Entering Draw Mode

1. Click the **paintbrush icon** in the top bar (next to the annotation pen icon)
2. The button highlights to indicate draw mode is active
3. A **toolbar** appears at the bottom of the screen with colour, width, select, eraser, and share controls
4. Draw directly on the score with finger, stylus, or mouse

Draw mode captures all pointer input on the score area. Score dragging is disabled while drawing. During playback the score continues to auto-scroll normally.

***Tip:** On a tablet with a stylus, draw mode gives you a natural ink-on-paper feel. Pressure data is captured for future variable-width rendering.*

Drawing

When draw mode is active:

- **Touch down / mouse down** on the score begins a stroke
- **Move** to extend the stroke it renders live as you draw
- **Lift / release** to finish the stroke
- The stroke is **saved automatically** to localStorage (and broadcast to other clients if sharing is enabled)

Very short taps (accidental touches) are discarded.

Strokes are rendered as smooth curves using quadratic bezier interpolation. Points are thinned during capture to keep data compact.

Session Grouping

All strokes drawn in a single draw-mode session belong to the same **group**. A new group is created each time draw mode is activated (clicking the paintbrush icon or pressing D). When you exit draw mode and re-enter, a new group begins.

This means drawing a word, symbol, or phrase produces a single group that can later be selected and moved as a unit.

To split work into separate groups, toggle draw mode off and on again between drawing sessions.

Toolbar

The floating toolbar appears when any drawing sub-mode (draw, select, or erase) is active. It can be **dragged** to a different position by the grip handle on its right edge.

Colour

A row of colour swatches. Click a swatch to change the stroke colour and return to draw mode. The active colour is indicated by a white border.

Default colours: white, red, blue, green, yellow, pink, grey.

Width

A slider controlling stroke width (120 units). Adjusts the thickness of subsequent strokes. Existing strokes are not affected.

Select / Move

A toggle button for select mode (see below). Highlighted in blue when active.

Eraser

A toggle button for eraser mode (see below). Highlighted in red when active.

Share

A toggle button controlling whether new strokes are shared with all connected clients or kept local. Highlighted in green when sharing is enabled (the default).

Drag Handle

The small grip bar on the right edge of the toolbar. Drag it to reposition the toolbar anywhere on screen.

Select and Move

Click the **move icon** in the toolbar (or press **V** while in draw or eraser mode) to enter select mode.

In select mode:

- Hover over any stroke to highlight its entire group (blue glow)
- **Click** a stroke to select its group a dashed blue bounding box appears
- **Shift-click** another group to add it to the selection (or remove it if already selected)
- **Drag** to reposition all selected groups together
- **Click empty space** to deselect

The bounding box encompasses all selected groups. On drag release, all stroke coordinates are updated and saved.

Keyboard shortcuts in select mode

Key	Action
Delete / Backspace	Delete all selected groups
S	Toggle scope of selected groups between local and shared
D	Return to draw mode
E	Switch to eraser
Escape	Exit select mode

Eraser

Click the **eraser button** in the toolbar (or press **E** while in draw or select mode) to enter erase mode.

In erase mode:

- Hover over a stroke to highlight it in red
- Click the stroke to delete it
- The deletion is saved immediately

Erasing works at the stroke level each drawn line is a separate object that can be individually removed. To delete an entire group at once, use select mode and press Delete.

Press **D** to return to draw mode.

Undo

Press **Ctrl+Z** (or Cmd+Z on Mac) while in draw or erase mode to undo the most recent stroke by the current author. Repeat to undo further strokes.

Exiting Draw Mode

Any of:

- Click the **paintbrush icon** again to toggle off
 - Press **Escape** from any sub-mode
 - The toolbar disappears and normal score interaction resumes
-

Keyboard Shortcuts

Key	Action	Context
Escape	Exit all drawing modes	Any drawing mode
D	Switch to draw mode	Select or eraser mode
V	Switch to select mode	Draw or eraser mode
E	Toggle eraser	Draw or select mode
S	Toggle local/shared scope on selection	Select mode with group(s) selected
Delete / Backspace	Delete selected group(s)	Select mode with group(s) selected
Ctrl+Z / Cmd+Z	Undo last stroke	Draw or erase mode

All shortcuts are suppressed when a text input is active (e.g. the annotation editor).

Local vs Shared

Shared (default)

- Broadcast to all connected clients via WebSocket
- Visible to everyone viewing the same project
- Stored on the server for late-joining clients
- Useful for conductor markings, collaborative annotation, or ensemble rehearsal notes

Local

- Stored only in the current browser
- Not visible to other performers
- Suitable for personal rehearsal markup

The share toggle on the toolbar controls the default scope for new strokes. The scope of existing strokes can be changed after the fact: select a group and press **S** to toggle all its strokes between local and shared. Shared strokes are transmitted as compact point-array data, not as images. All clients render strokes locally from the same world coordinates, so they align correctly regardless of screen size or resolution.

Persistence

Freehand annotations are stored as part of the annotation data in localStorage, keyed by project name. They persist across browser sessions and page refreshes.

They are included in annotation import/export (JSON). When exporting annotations, strokes are exported alongside text annotations, markers, and triggers in the same file.

Coordinates and Scaling

Strokes are stored in **world coordinates** the same coordinate space as the SVG scores viewBox. This means:

- Strokes remain anchored to the correct position in the score regardless of screen size or zoom level
- A stroke drawn on a tablet appears in the same score position on a laptop
- Shared strokes align correctly across devices with different resolutions

The drawing overlay uses an SVG element with a matching viewBox, so coordinate conversion is handled automatically by the browser's SVG rendering.

Stroke Appearance

By default, strokes use `vector-effect: non-scaling-stroke` in CSS. This means the visual thickness of a line stays constant regardless of how the score is scaled the same way a pen mark on paper looks the same width whether you hold the page close or far away.

To make strokes scale with the score instead (thicker when zoomed in, thinner when zoomed out), remove the vector-effect rule from `oscillaDrawing.css`.

What Freehand Annotations Do *Not* Do

Freehand annotations:

- do not trigger playback or cues
- do not alter the SVG score
- do not affect synchronisation or timing
- do not carry executable behaviour

They are purely visual markup. For executable score elements, use structured annotations with triggers.

Technical Notes

Data Model

Strokes are stored as annotation items with `kind: "stroke"`:

```
{
  "id": "ann_m3k...",
  "kind": "stroke",
  "scope": "shared",
  "anchor": { "mode": "scroll" },
  "placement": { "space": "score" },
  "stroke": {
    "points": [
      { "x": 1234.5, "y": 456.2, "p": 0.72 },
      { "x": 1238.1, "y": 458.0, "p": 0.68 }
    ],
    "color": "#ffffff",
    "width": 3,
    "opacity": 1,
    "groupId": "grp_01JK..."
  }
}
```

The `p` field records pointer pressure (01) from stylus input. This data is captured but not yet used for variable-width rendering.

The `groupId` field links strokes drawn in the same session. All strokes sharing a `groupId` are selected and moved together.

Module Location

`public/js/interaction/drawing.js`

Integrated via `interactionSurface.js` alongside the annotation editor, markers, and trigger system.

Window API

```
// Mode control
window.oscillaDrawing.toggleDrawMode()
window.oscillaDrawing.setDrawMode(true)
window.oscillaDrawing.setEraserMode(true)
window.oscillaDrawing.setSelectMode(true)
window.oscillaDrawing.toggleSelectMode()

// Stroke settings
window.oscillaDrawing.setStrokeColor("#ff4444")
window.oscillaDrawing.setStrokeWidth(5)

// Sharing
window.oscillaDrawing.toggleShareDrawing() // toggle default scope
window.oscillaDrawing.toggleSelectedScope() // toggle selected groups

// Edit operations
window.oscillaDrawing.undoLastStroke()
window.oscillaDrawing.clearLocalStrokes()
```

```

window.oscillaDrawing.deleteSelectedGroup()    // delete all selected groups

// State queries
window.oscillaDrawing.isDrawMode()
window.oscillaDrawing.isEraserMode()
window.oscillaDrawing.isSelectMode()
window.oscillaDrawing.isShareDrawing()
window.oscillaDrawing.getStrokeColor()
window.oscillaDrawing.getStrokeWidth()

```

Also accessible via the annotation API:

```

window.oscillaAnnotations.toggleDrawMode()
window.oscillaAnnotations.setDrawMode(true)

```

Future Directions

- **Pressure-sensitive rendering:** variable-width strokes using captured pressure data
- **Page mode support:** freehand annotation in page view (currently scroll mode only)
- **Stroke simplification:** Ramer-Douglas-Peucker path reduction for long sessions
- **Drawing layers:** multiple named layers (e.g. “rehearsal 1”, “rehearsal 2”) with independent visibility
- **Full colour picker:** integration with the existing `colorPicker.js` component

Markers

Markers are visual waypoints that performers and composers can drop onto the score during playback, rehearsal, or pre-performance planning. Unlike rehearsal marks (which are authored into the score itself), markers are created in the moment by anyone in the ensemble a lightweight way to sketch structure onto a timeline without interrupting the flow.

No SVG authoring required

Markers are part of Oscillas **interaction layer** features that work on top of any score without requiring you to edit SVG files or learn cue syntax. You can:

- Load any image or simple graphic as a score
- Drop markers to create structure
- Navigate between markers with arrow keys and transport buttons
- Use the timer system for durational cues, with automatic jumps to markers on completion
- Annotate with pins and text

This makes markers an accessible entry point for ensembles who want networked coordination and shared visual structure without diving into Oscillas more advanced SVG-based cue system. The score becomes a backdrop; markers become the notation. Combined with countdown timer sequences and `onComplete` actions, markers can form a complete temporal framework a **session layer** built entirely from the interaction tools.

For groups working with graphic scores, text scores, or fully improvised sets, this approach lets you use Oscilla as a **shared timeline and coordination tool** rather than a notation rendering engine. Wire markers to timer sequences and you have automated section navigation without writing a single line of SVG.

What markers are for

Markers serve as **compositional scaffolding** for loosely-defined works. In graphic notation, open scores, and improvised music, the timeline is often a container rather than a prescription. Markers let you populate that container with signposts:

- **Structural landmarks** “climax here”, “transition begins”, “return to opening texture”
- **Solo assignments** designate who plays when across a 20-minute set
- **Moments to revisit** flag a passage during a run-through without stopping
- **Cues for coordination** visible anchors that all performers can see and anticipate
- **Navigation targets** named markers can be jumped to with arrow keys, transport buttons, `nav()` cues, or timer completion actions

Think of markers as sticky notes on a shared timeline. Theyre fast to place, easy to move, and visible to everyone (or just yourself, if you prefer). Unlike plain sticky notes, though, theyre wired into the transport you can jump between them, and other systems can target them.

Pre-performance planning

Before a performance, markers can sketch out a flexible roadmap:

1. **Set duration** You have a 20-minute set. The score provides material but not strict form.
2. **Drop structural markers** Place markers for major sections: intro, development, climax, wind-down, ending.
3. **Assign solos** Each performer drops markers in their own color indicating where theyll take the lead.
4. **Mark transitions** Add markers where the ensemble should shift texture, density, or energy.
5. **Size for visibility** Drag important markers to the center of the score and increase their font size so theyre visible from across the stage.

This creates a shared map thats visible on every connected client. During performance, the playhead moves through the score and performers see their markers approachinggentle reminders rather than rigid cues.

During rehearsal

Markers shine as a **non-interruptive annotation tool**:

- Spot something interesting or problematic? Drop a marker without stopping.
- After the run-through, use Arrow Up/Down to jump between markers, or click any marker label to jump back and discuss.
- The marker label (editable) can hold a quick note: “bass too loud”, “nice texture”, “try again”.

This keeps rehearsals flowing. Instead of “stop, lets go back to where was it?”, you have a trail of bread-crumbs to follow and keyboard shortcuts to follow them instantly.

Shared and local markers

By default, markers are **shared** they appear on all connected clients scores in real-time. This supports collaborative planning and ensemble-wide visibility.

Toggle the network icon in the marker tools to switch to **local** mode. Local markers are private to your screen, useful for personal notes you dont need to broadcast.

Individual markers can also be toggled between shared and local in the marker editor (click any marker label to open it).

Colors

Each connected client is assigned a unique color, visible in the client list in the top bar. When you drop a marker, it inherits your client color by defaultmaking it easy to see at a glance who marked what.

Colors can be changed per-marker in the editor. Use them however makes sense for your ensemble:

- Per-performer (each player has their color)
- Per-function (red = problem, green = landmark, blue = solo)
- Per-section (color-code movements or scenes)

The system doesnt impose meaning on colorstheyre a visual vocabulary for you to define.

Marker controls

The marker tools appear in the top bar:

Button	Function
(arrow)	Drop marker at current playhead position
M (marker icon)	Toggle visibility of all markers on/off
(network icon)	Toggle sharing when active (blue), new markers sync to all clients

Button	Function
/ (transport)	Fast forward / rewind between markers and rehearsal marks
Arrow Up / Down	Keyboard navigation between markers and rehearsal marks

Dragging markers

Markers can be **dragged in two dimensions**:

- **Horizontal drag** moves the marker along the timeline
- **Vertical drag** slides the label up or down along the marker line

This lets you position a label anywhere on the score at the top for unobtrusive markers, or in the center of the page for prominent waypoints visible from across the room.

Editing markers

Click any marker label to open the **marker editor**:

- **Label** edit the text
- **Show vertical line** toggle the full-height line on/off
- **Share with all clients** switch between shared and local
- **Size** adjust font size (1032px) for prominence
- **Color** pick from presets or choose a custom color
- **Delete** remove the marker

Large font sizes combined with central vertical positioning create bold structural landmarks that performers can see at a glance during performance.

Relationship to other timing tools

Markers work alongside Oscillas other timing features:

- **Rehearsal marks** Navigation points defined in the SVG score using Oscillas cue syntax. These require authoring `cue_rehearsal(...)` IDs in your SVG file. Markers offer similar landmark functionality without any SVG editing they live in the interaction layer, not the score file. Since both rehearsal marks and markers share the same navigation system (arrow keys, fast forward/rewind, rehearsal popup), they function interchangeably as structural waypoints the difference is only in how they are created.
- **Countdown timer sequences** For non-timeline-based structures, the timer system can trigger sequences of countdowns (e.g., “5 minutes 3 minutes 2 minutes done”). Each timer slot has an optional **onComplete** field that accepts any cue expression. Setting `nav(scroll@intro)` or `nav(mark@bridge)` as a slot's onComplete action causes the playhead to jump to that marker when the countdown finishes. This means you can build an entire performance structure from markers and timers alone no SVG authoring required.
- **nav() cues** Named markers can be targeted from SVG cue syntax or from timer onComplete fields. See *Navigation targets* below.

Navigation targets

Named markers (any marker whose label has been edited from the default “m”) can be jumped to programmatically using Oscillas cue syntax:

Syntax	Behavior
<code>nav(scroll@myMarker)</code>	Jumps to rehearsal mark “myMarker” first; if not found, jumps to the user marker with that name. Playback auto-resumes.
<code>nav(mark@myMarker)</code>	Jumps directly to the user marker (skips rehearsal marks). Playback auto-resumes.

Syntax	Behavior
<code>nav(markPaused@myMarker)</code>	Jumps to the user marker and stays paused.
<code>nav(scrollPaused@myMarker)</code>	Jumps to rehearsal mark or marker and stays paused.

These expressions work anywhere a cue string is accepted:

- **In SVG cue IDs** embed navigation in the score itself
- **In timer onComplete fields** jump to a marker when a countdown slot finishes
- **In button triggers** wire a `button()` cue to jump to a specific marker

When multiple markers share the same name, the leftmost one (by score position) is used. A console warning is logged when duplicates are found.

Keyboard and transport navigation

Markers integrate into Oscillas transport navigation alongside rehearsal marks:

Control	Action
Arrow Up	Jump to the next navigation point (marker or rehearsal mark) after the current playhead position
Arrow Down	Jump to the previous navigation point before the current playhead position
Fast Forward button	Same as Arrow Up
Fast Rewind button	Same as Arrow Down
Rehearsal popup	Shows all rehearsal marks and named markers; click any to jump

All navigation points rehearsal marks and markers are sorted by their position on the score. Arrow keys and transport buttons walk this unified list based on where the playhead currently is, not on a stored index. This means navigation stays correct regardless of how the playhead got to its current position (manual seeking, cue jumps, timer actions, etc.).

Default unnamed markers (“m”) are included in keyboard navigation. All markers with a valid position are navigable.

The session layer

Markers, timers, and the `nav()` cue together form what could be called a **session layer** a way to construct temporal structure on top of any score without editing SVG files.

A typical session-layer workflow:

1. **Load any image or graphic as a score** it doesn't need cue IDs, rehearsal marks, or any Oscilla-specific markup.
2. **Drop and name markers** create structural waypoints: “intro”, “bridge”, “climax”, “ending”.
3. **Build a countdown sequence** create timer slots matching your planned section durations.
4. **Wire onComplete actions** set each slot's `onComplete` to `nav(scroll@nextSection)` so the playhead automatically jumps forward when each timer finishes.
5. **Optionally add ui() calls** use `ui(#layerName, visible:true)` in `onComplete` fields to show/hide visual layers at section boundaries.

The result is a fully navigable, timed performance structure built entirely from the interaction layer. The score provides visual material; markers and timers provide the temporal framework.

Typical workflow

Before rehearsal: 1. Load the score 2. Set the duration/tempo 3. Drop markers to outline the set structure 4. Assign colors or labels as needed 5. Optionally: build a countdown sequence with `onComplete` actions targeting your markers

During rehearsal: 1. Start playback 2. Drop markers on-the-fly when something notable happens 3. Use Arrow Up/Down to jump between markers without stopping 4. Dont stop keep playing

After the run-through: 1. Click markers (or use arrow keys) to jump back to flagged moments 2. Discuss, adjust, make notes 3. Delete or reposition markers as the interpretation evolves

Before performance: 1. Clear rehearsal markers (or hide them) 2. Keep only the structural landmarks you want visible 3. Perform with a shared map that guides without dictating

Markers are intentionally simple to create drop, name, drag, done. But they connect into Oscillas deeper systems: theyre navigable by keyboard and transport controls, targetable by `nav()` cues, and triggerable from timer completion actions. This makes them a bridge between casual annotation and structured performance start with sticky notes on a timeline, and wire them into a full temporal framework when youre ready.

Timers

Overview

Oscillas Timer system provides flexible time-tracking tools for performers, conductors, and ensembles. At its core is the **Auxwatch** an auxiliary timer that can function as a simple stopwatch, countdown timer, or sequenced section timer for structuring musical time.

The Timer system is designed for: - **Solo performers** tracking practice or performance duration - **Ensembles** synchronizing timed sections across multiple devices - **Improvisers** using time-based structures without traditional notation - **Conductors** managing sectional rehearsals or timed cues - **Session-layer workflows** where timer slots trigger navigation to markers, creating automated performance structures

Accessing the Timer

Click the **stopwatch display** in the top bar to open the fullscreen Timer view.

Fullscreen Timer Display

The fullscreen view shows three columns at the bottom:

Element	Description
Mini Timer	Swappable display (Performance Time or Timer)
Clock	Current time of day
Countdown Controls	Play button and sequence selector with red ring indicator

The large central display shows elapsed time or active countdown.

View Modes

Click the **button** (top left) to cycle through display modes:

1. **Solid** Dark background, light text. Clean, high-contrast display.
2. **Blur** Score visible but blurred behind timer. Maintains context.
3. **Transparent** Timer overlays score directly. Ideal for minimal cue scores.

Press **Escape** or click to exit fullscreen.

Countdown Controls

The rightmost column in the fullscreen view contains the countdown controls:

Element	Description
/	Play/Stop button for the selected sequence or cue
Selection text	Shows currently selected sequence or cue name (click to change)
Red ring	Click to open the Countdown Sequences editor

Quick Playback

1. Click the **selection text** to open the dropdown
2. Choose a sequence or individual cue
3. Click to start

The selection remembers your last choice between sessions.

Playlist Behavior

The countdown controls work like a playlist: - After a countdown completes, the selection automatically advances to the next item - When reaching the end of the list, it wraps back to the beginning - This makes it easy to step through multiple sequences or cues manually

Swapping Timers

Click the **Mini Timer** box (left column) to swap positions: - Performance Time and Timer swap between main and mini display - Choose which time reference dominates the display

Countdown Sequences

Click the **red ring** to open the Countdown Sequences editor.

Creating Sequences

A **sequence** is a named group of timed sections that play consecutively. Each section can optionally trigger an action when it finishes.

Example Sonata Form:

Sequence: "Sonata"

— Exposition	120s	onComplete: nav(scroll@development)
— Development	180s	onComplete: nav(scroll@recap)
— Recapitulation	90s	

Editor Controls

Control	Function
+ Add Sequence	Create a new sequence group
Sequence Name	Text field to name the sequence
(on sequence)	Play entire sequence from start
	Delete sequence
Loop	Number of times to repeat (0 = infinite)
Then	Chain to another sequence when complete
+ Add Cue	Add a timed section to sequence
Cue Name	Name displayed during countdown
Duration (s)	Length in seconds
onComplete	Cue expression to trigger when this slot finishes (e.g. nav(scroll@B))
(on cue)	Play single cue only
	Delete cue

Looping

The **Loop** control determines how many times a sequence repeats:

Value	Behavior
1	Play once (default)
2, 3,	Repeat that many times
0	Loop infinitely until manually stopped

Example: A 3-cue sequence with Loop = 2 plays:

Cue 1 → Cue 2 → Cue 3 → Cue 1 → Cue 2 → Cue 3 → stop

Chaining (Sequence of Sequences)

The **Then** dropdown lets you chain sequences together:

"Warm-up" (Loop: 1, Then: "Main Set")
→ plays once, then automatically starts "Main Set"

"Main Set" (Loop: 3, Then: "Cool-down")
→ plays 3 times, then starts "Cool-down"

"Cool-down" (Loop: 1, Then: – stop –)
→ plays once, then stops

This creates a **playlist** of sequences without manual intervention.

Per-Cue Completion Actions (onComplete)

Each cue slot has an optional **onComplete** field a cue expression that fires when that slots countdown reaches zero. This connects the timer system to Oscillas navigation and UI systems.

onComplete expression	What happens
nav(scroll@B)	Jump to rehearsal mark or marker “B”, auto-resume playback
nav(mark@bridge)	Jump to user marker “bridge” (skips rehearsal marks)
nav(markPaused@ending)	Jump to marker “ending”, stay paused
ui(#brass, visible:true)	Show a visual layer
page(C)	Switch to page C
audio(src:gong.wav)	Play a sound

Any valid Oscilla cue expression works in the onComplete field. The action fires on **all connected clients** simultaneously (since the server owns the timer).

Example timed sections with automatic navigation:

```
Sequence: "Performance"
├─ Intro          120s  onComplete: nav(scroll@development)
├─ Development    180s  onComplete: nav(scroll@climax)
└─ Climax         90s   onComplete: nav(scroll@ending)
```

When each sections countdown finishes, the playhead automatically jumps to the next marker. The score scrolls itself through the performance structure.

onComplete actions also work with single cues (not just sequences). If you start a standalone countdown that has an onComplete expression, it fires when that countdown reaches zero.

Playback Behavior

When a countdown runs: - Main display shows **section name** as label - Time counts down in **MM:SS** format - Sequences auto-advance to next section - **onComplete** action (if set) fires when each section finishes - Single cues return to normal timer when complete - Play button changes to (stop)

Network Synchronization

The Timer system automatically synchronizes across all connected clients.

How It Works

- When any client starts a countdown, **all clients** see the same countdown
- Sequences are shared automatically when clients connect
- Time is synchronized using server timestamps all displays show the same remaining time

What Syncs

Event	Behavior
Client connects	Receives current sequences from server
Sequence created/edited	Changes broadcast to all clients (including onComplete fields)
Countdown started	All clients show synchronized countdown
Cue slot completes	Server broadcasts onComplete action; all clients execute it simultaneously
Countdown stopped	All clients return to normal display

Server-Owned Timer

The countdown timer is **server-owned**, meaning:

- The server maintains the authoritative timer state
- Clients send commands (start/stop) to the server
- Server broadcasts state to all clients
- Late-joining clients receive current countdown state immediately

This ensures all performers see exactly the same time, regardless of when they connected.

Import & Export

Sequences can be saved and loaded as JSON files.

Export

Click **Export** to download countdown-sequences.json containing all sequences.

Example JSON:

```
[
  {
    "name": "Warm-up",
    "loop": 1,
    "chain": 1,
    "cues": [
      { "name": "Stretch", "seconds": 120, "onComplete": null }
    ]
  },
  {
    "name": "Sonata",
    "loop": 1,
    "chain": null,
    "cues": [
      { "name": "Exposition", "seconds": 120, "onComplete": "nav(scroll@development)" },
      { "name": "Development", "seconds": 180, "onComplete": "nav(scroll@recap)" },
      { "name": "Recapitulation", "seconds": 90, "onComplete": null }
    ]
  }
]
```

Import

Click **Import** and select a JSON file to load sequences.

Note: Import merges with existing sequences by ID. Sequences are automatically shared to other connected clients after import.

Use Cases

1. Structured Improvisation

Create a sequence of timed sections:

```
"Free Improv Structure"
├─ Sparse Texture    60s
├─ Build             90s
├─ Peak              45s
└─ Decay             120s
```

All performers see the same countdown, enabling synchronized structural changes without traditional cues.

2. Session Layer Markers + Timers

Use markers and timer onComplete actions to build a complete performance structure without editing SVG:

1. Load any image as a score
2. Drop and name markers at key positions: “intro”, “development”, “climax”, “ending”
3. Create a countdown sequence with onComplete actions:

```
"Performance"
├─ Intro            120s  onComplete: nav(scroll@development)
├─ Development      180s  onComplete: ui(#overlay, visible:true)
├─ Climax           90s   onComplete: nav(scroll@ending)
└─ Ending           60s
```

The playhead jumps between markers automatically as each section finishes. Layers can appear/disappear at section boundaries. No SVG cue authoring needed.

3. Rehearsal Sectionals

Time individual sections during rehearsal:

```
"Rehearsal Plan"
├─ Warm-up          300s
├─ Exposition work   600s
├─ Break             300s
└─ Development work  600s
```

4. Performance Timing

For pieces with strict durational requirements:

```
"Competition Piece"
├─ Movement I       480s
├─ Pause             30s
└─ Movement II      360s
```

5. Minimal Cue Scores

Use **Transparent mode** with countdown sequences as a time-based score: - Score contains only markers - Timer overlays showing current section - onComplete actions handle navigation between sections - Performers follow time structure, not traditional notation

Technical Notes

Storage

- Sequences stored in `localStorage` as `oscilla.countdownSequences`
- Each cue object contains `name`, `seconds`, and optional `onComplete` (cue expression string or null)
- Selected sequence/cue stored as `oscilla.countdownControls.type` and `.index`
- Persists between sessions

Timer Precision

- Display updates every 250ms (syncd with server broadcast)
- Network sync uses `Date.now()` timestamps
- Typical sync accuracy: 50-200ms depending on network

Performance Time vs Timer

Timer	Behavior
Performance Time	Linked to transport. Pauses on manual stop, continues during musical pauses (cue-triggered).
Timer (Auxwatch)	Independent stopwatch. Can be used for countdowns. Swappable with Performance Time.

Keyboard Shortcuts

Key	Action
Escape	Exit fullscreen timer
Click stopwatch	Enter fullscreen
Click mini timer	Swap timer positions
Click red ring	Open countdown editor
Click selection text	Open sequence/cue dropdown

Summary

The Timer system transforms Oscilla into a flexible time-structuring tool:

- **Simple:** Click stopwatch fullscreen timer
- **Quick access:** Play button and dropdown right in the fullscreen view
- **Sequenced:** Create named countdown sections with looping and chaining
- **Actionable:** Per-cue `onComplete` fields trigger navigation, UI changes, or any cue expression
- **Synchronized:** All clients automatically stay aligned
- **Portable:** Import/export sequences as JSON

Combined with markers, the Timer system enables a **session layer** approach building complete performance structures from the interaction layer without editing score files. Whether for strict durational works, timed improvisations, or rehearsal management, the Timer provides a unified interface for musical time.

drag() Moveable Score Elements

Make any SVG group draggable in the browser. Author a palette of visual components in Inkscape, then arrange and rearrange them during rehearsal or performance. Positions persist and sync across all connected clients.

Quick Start

In Inkscape, give a group the ID `drag(1)`:

```
<g id="drag(1)">
  <circle cx="100" cy="100" r="30" fill="red" />
</g>
```

Open in browser click and drag the group anywhere on the score.

Syntax

```
drag(1) // basic draggable
drag(1, osc:1) // + send position via OSC
drag(1, osc:/my/custom/addr) // + custom OSC address
```

Parameter	Description
1	Enable drag (required)
osc:1	Send x,y position via OSC to /oscilla/drag/{id}
osc:/addr	Send to a custom OSC address

Combining with Animations

Drag works alongside any cue or animation. Use a **compound ID** (space-separated) to combine drag with scale, rotate, o2p, etc:

```
// drag + scale
<g id="scale(values:[1,1.5,1], dur:2) drag(1)">

// drag + rotate + OSC
<g id="rotate(dir:1, dur:4) drag(1, osc:1)">

// drag + slow rotate with uid
<g id="rotate(dir:1, dur:30, uid:myThing, osc:1) drag(1)">
```

The animation plays on the element while you can still grab and reposition the whole group. They don't interfere with each other.

Nesting

Use the documented nesting pattern to keep cues separate:

```
<g id="drag(1)">
  <g id="scale(values:[1,2,1], dur:3)">
    <rect ... />
  </g>
</g>
```

Behaviour stacks outer inner. The outer group is draggable, the inner group scales. Shapes inherit all enclosing cues.

With propagate

Each child in a propagated group gets its own independent drag handler:

```
<g id="propagate(rotate(dir:1, dur:rnd(2,8)) drag(1))">
  <circle cx="50" cy="50" r="20" />
  <circle cx="150" cy="50" r="20" />
  <circle cx="250" cy="50" r="20" />
</g>
```

Each circle rotates independently AND can be dragged independently.

Use Cases

Modular score layout Author a collection of musical fragments as separate groups in Inkscape. Add drag(1) to each. In the browser, arrange them spatially to create the performance layout. Positions are saved close the browser and reopen, everything is where you left it.

Performer interaction Performers drag their assigned elements during performance, creating a collaboratively arranged visual score in real time. All clients see the changes immediately.

Spatial control via OSC Use `drag(1, osc:1)` and route x,y position to synthesis parameters drag a visual element to pan a sound source, control filter frequency by vertical position, etc.

Rehearsal tool Quickly rearrange sections of a score during rehearsal without going back to Inkscape. Reset all positions when you're done: open the browser console and type `resetDragPositions()`.

Persistence

Drag positions are automatically saved and shared:

- **Across page reloads** positions survive browser refresh
- **Across clients** drag on one screen, all connected screens update
- **Across server restarts** positions saved to JSON in your project folder
- **Per project** each project has its own saved positions

Positions are stored in `scores/myProject/drag-positions.json` alongside your other project files. This file is auto-generated you don't need to create or edit it.

Resetting Positions

To move everything back to the original authored positions:

```
// Browser console
resetDragPositions()
```

This clears all drag positions for the current project on all connected clients and deletes the saved JSON file. Elements return to where they were placed in Inkscape.

Workflow

1. Author shapes & groups in Inkscape
2. Add `drag(1)` to group IDs (XML Editor: Ctrl+Shift+X)
3. Optionally combine with animations:
`rotate(dir:1, dur:4) drag(1)`
4. Save SVG → Open in browser
5. Drag elements to arrange the score
6. Positions sync to all clients + saved to disk
7. `resetDragPositions()` to start fresh

Live Console Live Coding & Inspection

The Live Console lets you type and execute Oscilla DSL expressions in real time, directly against elements in the running score. It also provides a signal monitor showing all active control-plane values.

Use it for:

- **live coding** type DSL, execute it, hear/see the result
- **testing** try parameter changes without editing the SVG
- **debugging** inspect signal flow and animation state
- **performance** build and modify animations during a show

Opening the Console

Click the `>` button in the top bar. The console panel opens on the right side of the screen. Click the button again (or the X) to close it.

Panel Layout

The panel has four sections, top to bottom:

Target

Shows which SVG element your DSL will be applied to. You can set the target by:

- **Picking** click “pick”, then click any element in the score. The element is highlighted and its uid populates the target field.
- **Typing** enter a uid or element id in the target field and press Enter. Partial matches against the animation registry are supported.

The info line below the field shows the elements tag, id, uid, kind (rotate/scale/o2p/), and whether it is currently running.

Animation cues (rotate, scale, o2p, color, fade) require a target element. Other cues (synth, audio, speed, nav, osc, stop, pause) do not they execute without a target.

Editor

A text area where you type DSL expressions. Execute with:

- **Ctrl+Enter** run the current line (where the cursor is)
- **Ctrl+Shift+Enter** run all lines

When you pick an element, its existing DSL expression (from its SVG id) is pre-filled in the editor so you can modify and re-apply.

Output

Shows the result of each execution: green for success, red for parse errors or missing targets.

Signals

A live-updating view of all values on the ParamBus (the control plane). Updated at 5 fps. Use the filter field to narrow by path prefix e.g.type rotate to see only rotation signals, or a uid to see a specific elements outputs.

Usage Examples

Modify a running rotation

1. Click **pick**, click the rotating element in the score
2. The editor fills with the current DSL, e.g. rotate(dir:1 dur:30 uid:spinner1 osc:1)
3. Change dur:30 to dur:5 and press **Ctrl+Enter**
4. The element now rotates at the new speed

Start a synth (no target needed)

Type directly in the editor:

```
synth(freq:440 amp:0.3 wave:sine)
```

Press Ctrl+Enter. No target element required.

Change playback speed

```
speed(0.5 dur:4)
```

Ramps playback to half speed over 4 seconds.

Navigate to a rehearsal mark

```
nav(B)
```

Jumps to rehearsal mark B.

Trigger an audio cue

```
audio(clicks.wav vol:0.6 loop:1)
```

Layer multiple cues

Write multiple lines and run all with Ctrl+Shift+Enter:

```
rotate(dir:-1 dur:8 uid:orb1)
synth(freq:orb1.norm[200,800] amp:0.4)
speed(1.5)
```

Keyboard Behaviour

While the cursor is in any field or the editor inside the live console panel, all keyboard events are captured by the panel. Arrow keys, spacebar, and other keys will not trigger transport controls or score navigation. Keyboard shortcuts resume normal behaviour when focus leaves the panel.

Signal Monitor

The signal monitor displays all values currently published to the ParamBus. Signals are published by running animations:

```
o2p:slider1.t      0.4523
rotate:spinner.norm 0.7210
synth:drone1.freq   442.00
```

Use the filter to focus on signals you care about. This is useful for verifying that cross-cue modulation sources are producing expected values before wiring them to targets.

Tips

- You can re-pick a different element at any time without closing the panel.
 - The DSL you type follows exactly the same syntax used in SVG element IDs in Inkscape the same parser handles both.
 - Animation cues applied via the console replace the previous animation on that element, just as re-triggering from the playhead would.
 - The console does not modify the SVG file. Changes exist only in the running session.
-

Related

- [Control & Modulation](#)
- [Cue System](#)
- [rotate\(\)](#), [scale\(\)](#), [o2p\(\)](#), [synth\(\)](#)

Chapter 3

Cues

Cues are the core mechanism through which an Oscilla score becomes executable. A cue is a short instruction embedded in the ID attribute of an SVG element. The most common trigger is the scrolling playhead when it crosses an element, the cue fires. But cues can also be triggered by interactive buttons embedded in the score, by timer sequences on completion of each unit, or typed directly into the live console during performance.

The cue syntax is designed to be readable and compact a minimal, accessible form of text-based authoring rather than traditional programming. A single element can carry multiple cues, and cues can reference other elements, enabling navigation, conditional branching, and structural control within the score. Cues are typed into Inkscape's Object Properties dialog, or built using the Oscilla Inkscape extension which provides a graphical interface for assembling cue expressions.

This section documents each cue type with its syntax, parameters, and usage examples.

stop(...) Toggle Playback (On / Off)

The `stop(...)` cue toggles the transport:

- if playback is running it stops
- if playback is already stopped it starts again

It does not end the score and it does not freeze the system.

Syntax

```
stop()
stop(uid:NAME)
stop(next:CUEID)
```

Parameters

param	required	description
uid		Only needed if multiple identical <code>stop(...)</code> cues appear
next		Trigger another cue immediately after <code>stop(...)</code> runs

Behaviour

- toggles playback (stop start)
- briefly ignores sync echoes
- `next(...)` triggers another cue **immediately** (does not wait for resume / spacebar)

Examples

```
stop()
stop(uid:s1)
stop(next:nav(End))
```

Related

- `pause(...)`
- `nav(...)`

pause() Temporarily Halt Playback

The `pause(...)` cue temporarily stops playback for a fixed duration. It may optionally display a count-down overlay and can automatically trigger another cue when finished.

Playback and synchronization are completely frozen during a pause.

Syntax

```
pause(dur:SECONDS, count:BOOL, next:CUEID, uid:NAME)
```

Daisy-chaining with next(...)

`next(...)` does **not** make the pause jump anywhere by itself.

Instead, it tells Oscilla:

“When the pause finishes, trigger this other cue.”

```
pause(dur:4, next:sectionB)
```

What happens:

1. pause
2. countdown (if enabled)
3. the cue named `sectionB` is triggered

If `sectionB` happens to be a navigation cue, the score may jump but it is the **navigation cue** doing the jump, not the pause system.

speed()

Controls the playback scroll speed of the score.

The `speed()` cue sets the playback speed multiplier. Speed values are **relative to the base speed** set in project preferences. Speed changes may be applied instantly or gradually over time.

Syntax

```
speed(
  value:<multiplier>,
  add:<offset>,
  dur:<seconds>,
  ease:<easing>,
  uid:<id>
)
```

Shorthand (positional):

```
speed(<multiplier>)
```

Parameters

- **value**
Speed multiplier relative to the base speed.
1 = base speed, 0.5 = half base speed, 2 = double base speed.
- **add**
Relative adjustment applied to the current speed multiplier.
- **dur**
Duration of the speed change in seconds.
If omitted, the speed change is applied immediately.

- **ease**
Easing curve used during a timed speed change.
Defaults to linear.
- **uid**
Optional unique identifier for synchronization.

Base Speed Multiplier (Preferences)

The **base speed multiplier** is set in your projects preferences.json:

```
{
  "defaultPlaybackSpeed": 0.3
}
```

All speed() cues multiply against this base value:

Preference	Cue	Actual Speed
0.3	speed(1)	0.3 (1 0.3)
0.3	speed(2)	0.6 (2 0.3)
0.3	speed(3)	0.9 (3 0.3)
1.0	speed(0.5)	0.5 (0.5 1.0)

This allows you to: - Set a **global tempo** for the entire piece in preferences - Use speed() cues for **relative tempo changes** within the score - Adjust overall tempo without editing individual cues

Two Types of Speed Cues

1. Position-Based Speed Markers

Simple speed markers embedded in the SVG that apply **instantly** when the playhead crosses them:

```
speed(1)
speed(2)
speed(0.5)
```

These are extracted from SVG elements with IDs starting with speed(. They: - Apply immediately when crossed - Are position-aware (rewinding past a marker reverts to the previous speed) - Automatically sync with the server - Multiply against baseSpeedMultiplier

2. Triggered Speed Cues (cueSpeed)

Full cue syntax with ramping support:

```
cueSpeed(value:2, dur:3)
```

These support gradual transitions and are triggered like other cues.

Position Awareness

When you **rewind or seek** to a position before a speed marker, the system automatically applies the correct speed for that position:

```
Position:    0 ----- 5000 ----- 10000 ----- 15000
Markers:     speed(1)  speed(3)      speed(1)
```

Playhead at 7000 → speed = 3 × base

Rewind to 3000 → speed = 1 × base (automatically updated)

This ensures consistent playback regardless of navigation direction.

Behaviour

- Speed values multiply against `baseSpeedMultiplier` from preferences
 - Position-based markers (`speed(x)`) apply instantly when crossed
 - Triggered cues (`cueSpeed`) support ramping with `dur` parameter
 - Rewinding past a speed marker reverts to the previous speed
 - Speed changes sync to all connected clients via the server
 - A new speed cue replaces any previously active speed change
 - Speed changes affect scrolling, timing, and all time-based cues
-

Examples

Simple Position Markers (in SVG)

`speed(1)`

Base playback speed.

`speed(0.5)`

Half of base speed.

`speed(2)`

Double base speed.

Triggered Cues with Ramping

`cueSpeed(value:1.25, dur:3)`

Gradually change to 1.25 base speed over 3 seconds.

`cueSpeed(add:-0.5, dur:2)`

Gradually reduce speed multiplier by 0.5 over 2 seconds.

`cueSpeed(value:2, dur:6, ease:inOutQuad)`

Smooth, eased speed transition to 2 base speed.

Project Setup

preferences.json

```
{
  "defaultPlaybackSpeed": 0.3,
  "duration_minutes": 10,
  "darkMode": false
}
```

SVG Score

Place speed markers as elements with appropriate IDs:

```
<circle id="speed(1)" cx="100" cy="50" r="5" fill="blue"/>
<circle id="speed(3)" cx="5000" cy="50" r="5" fill="red"/>
<circle id="speed(1)" cx="10000" cy="50" r="5" fill="blue"/>
```

The visual element (circle, rect, path, etc.) marks the position; only the `id` matters for the cue system.

Server Synchronization

Speed changes are broadcast to all connected clients:

```
{
  type: "set_speed_multiplier",
  multiplier: 0.9,      // actual speed (cue × base)
  cueSpeed: 3,         // raw cue value
  baseMultiplier: 0.3, // from preferences
  source: "position_watch"
}
```

The server maintains the authoritative speed state and syncs it at ~4Hz.

Debugging

Check speed state in browser console:

```
console.log("baseSpeedMultiplier:", window.baseSpeedMultiplier);
console.log("speedMultiplier:", window.speedMultiplier);
console.log("speedCueMap:", window.speedCueMap); // position-based markers
```

Watch speed changes in real-time:

```
[speedWatch] pos=5000, cue=3, base=0.3, actual=0.90, current=0.30
[speedWatch] □ SPEED CHANGE: 0.30 → 0.90 (cue=3 × base=0.3)
```

Notes

- Speed cues modify global playback behaviour
- Only one speed is active at a time
- The performer sees only a cue symbol in the score
- The microsyntax appears only in the SVG `id` field
- Position-based speed is automatically restored when seeking/rewinding
- Base speed from preferences allows global tempo control

stopwatch (with autostart support)

Displays a stopwatch overlay above the score. Stopwatch overlays may show the **main stopwatch time** or create their own **independent stopwatch**.

This cue now supports **autostart**, meaning it can begin running automatically when the SVG is loaded in both page and scroll modes without requiring the playhead to cross.

Syntax

```
stopwatch(source:<main|new>, hold:<seconds>, scroll:<true|false>,
          offsetX:<pixels>, style:<"css rules">, trig:<auto|edge>)
```

Arguments

Arg	Values	Default	Behaviour
source	"main" / "new"	"main"	Use global clock or create/reset local stopwatch
hold	seconds	0	When >0, overlay fades+removes at timeout
scroll	true / false	false	Follows scrolling score or fixed overlay
offsetX	px	0	X-offset from cue location
style	CSS		Inline CSS, use ; separated
trig	"auto" / "edge"	"edge"	auto means autostart on load

Behaviour

- `source:main` displays the **global** stopwatch time
- `source:new` starts (or resets) a **private timer**
- If `hold > 0`, the stopwatch fades after expiry
- If `hold = 0`, click dismisses it
- Multiple independent stopwatches can run concurrently
- Autostart requires no playhead crossing

Autostart (NEW)

Autostart activates when:

`trig:auto`

Example:

```
stopwatch(source:new,style:"font-size:3em;color:red",trig:auto)
```

Autostart behaviour:

- Works in **page overlay** and **scroll** modes
- Executed **after SVG and DOM are fully initialized**
- Overlay appears immediately
- Does not require playhead position
- Does not use trigger dedupe

Autostart Examples**Main stopwatch, always visible**

```
stopwatch(source:main,style:"font-size:2.5em;",trig:auto)
```

Independent timer with 10s hold

```
stopwatch(source:new,hold:10,style:"font-size:2em;",trig:auto)
```

Scrolling stopwatch

```
stopwatch(source:new,scroll:true,trig:auto)
```

Stylised overlay

```
stopwatch(source:new,
  style:"font-size:4em;color:#0ff;text-shadow:0 0 5px #000;",
  trig:auto)
```

Edge-triggered Examples (non-autostart)

```
stopwatch(source:new)
```

```
stopwatch(source:main,scroll:true)
```

```
stopwatch(source:new,hold:8,offsetX:-30)
```

Developer Notes

- Autostart runs **once**, after initialization
- Stopwatch overlay is a **DOM absolute positioned element**
- High stacking ensured using:
`div.style.zIndex = "99999"`
- Autostart does not use `triggeredCues`

Known Limitations

- Overlap of multiple autostarts is a user-design choice
- Very large fonts may clip on small screens

metronome (*with autostart support*)

Triggers a visual and/or audible metronome marker at the cues location or a specified target: element. The metronome can follow scrolling objects, stay fixed to the viewport, and optionally send OSC messages.

The metronome **can also autostart**, meaning it runs automatically when the SVG is loaded in either page mode or scroll mode without needing a cue trigger crossing.

Syntax

```
metro(bpm:<number>,beats:<number>,visual:<type>,audio:<0|1>,
      position:<fixed|scrolling>,osc:<0|1>,uid:<string>,
      target:<uid>,hideTarget:<0|1>,showcount:<0|1>,hold:<seconds>,
      colour:<csscolour>,size:<pixels>,trig:<auto|edge>)
```

Parameters

Parameter	Type	Default	Description
bpm	number	120	Beats per minute. Controls metronome speed.
beats	number	4	Number of beats per cycle.
visual	string	"circle"	Visual style (circle, hex, future shapes).
audio	0 / 1	0	Enables click sound.
position	"fixed" / "scrolling"	"fixed"	Follow-scrolling or fixed overlay.
osc	0 / 1	0	Publishes OSC beat messages.
uid	string	"default"	Unique metronome identifier.
target	uid		Anchor element (scrolling mode only).
hideTarget	0 / 1	1	Hide target element when starting.
showcount	0 / 1	1	Show count number inside shape.
hold	seconds	0	Stop after defined duration.
colour	CSS color	"red"	Circle colour.
size	pixels	50	Circle diameter.
trig	"auto" / "edge"	"edge"	Whether to autostart or trigger when playhead crosses.

Behaviour

- Runs as **overlay DOM** element (not part of the SVG).
- Follows scrolling or stays screen-fixed depending on `position`.
- Supports **audio click** and **OSC output**.
- Supports **multiple concurrent** metronomes (UID required).
- Can target another SVG element and hide it on activation.

Autostart Mode (NEW)

When is a metronome autostarted?

Whenever the cue contains:

```
trig:auto
```

Example:

```
cue:metro(bpm:90,visual:hex,uid:m21,trig:auto)
```

When does autostart happen?

- In **page overlay mode** (during PDF-style view)
- In **scroll mode** (timeline view)
- **After the SVG and DOM are initialized**

- **Before** the animation clock starts

What does autostart actually do?

Autostart metronomes fully initialise as if they had been triggered by the scrolling playhead:

- Overlay DOM is created
- Beat scheduler starts
- UI is visible
- OSC messages begin
- Audio (if enabled) clicks on beat 1,2,3,4

Multiple autostarts supported

Each metronome UID becomes its **own independent instance**.

Example cues

Description	Example
Autostart metronome at load	<code>metro(bpm:90,visual:hex,trig:auto,uid:m1)</code>
Two autostarts	<code>metro(bpm:70,trig:auto,uid:A) + cue:metro(bpm:110,trig:auto,uid:B)</code>
Standard (on crossing), no autostart	<code>metro(bpm:100,visual:circle,uid:M)</code>
Scroll-following autostart	<code>metro(bpm:120,position:scrolling,target:x1,trig:auto)</code>

OSC Message Example

```
/oscilla/metro
{
  "uid": "m21",
  "client": "rob",
  "bpm": 90,
  "beat": 3
}
```

Implementation Notes (for developers)

- Autostart hooks run **after** DOM insertion and SVG setup.
- Autostart routines are separate from cue crossing logic:
 - No tracking in `triggeredCues`
 - No playhead position checks required
- Overlay visibility ensured via:


```
div.style.zIndex = "99999";
```

Known Limitations

- Multiple autostarts in high BPM (>200) may need scheduler improvements.
- OSC feedback not currently aware of phase differences between UIDs.
- Appearance position is based on CSS absolute, not SVG hitbox.

`page()` navigate pages or scrolling positions in the score

Triggers navigation between SVG score pages or scroll positions.

The cue can load a single page, sequence multiple pages, loop sections, choose randomly from options, randomize order, or define composite playlists using pattern-based syntax. It forms the structural backbone for cue-based navigation and timeline control in Oscilla scores.

Syntax

`page()` `page(Pseq([:,:],))` `page(Pxrand([:,:],))` `page(Pshuf([:,:],))` `page(Prand([:,:],))` `page(Pchoose([:,:])` `page(Pseq([page1:3, page2:3],1), after:mode(scroll@F))` `page(Pseq([Pseq([page1:2,page2:2],2),page3:4],1))`

Arguments

Argument	Description
page	ID of a single page or scroll section to load
Pseq	sequential playback of listed pages or subpatterns
Prand	random selection with replacement
Pxrand	random order, reshuffled each repeat cycle
Pshuf	random order, shuffled once and repeated as fixed order
Pchoose	choose one random page from a list (per trigger)
repeats	number of cycles for the pattern (<code>inf</code> = infinite)
:	optional duration for each page (in seconds)
after:	optional post-action (e.g. <code>mode(scroll@F)</code>)
mode:	switch playback display mode (<code>scroll</code> or <code>page</code>)
@uid	rehearsal mark or cue ID to jump to after sequence completes

Behavior

- `page(page1)` jumps immediately to the given page or scroll position.
- `page(Pseq([page1:2,page2:2],3))` loops between pages 1 and 2 three times.
- `page(Prand([page1,page2,page3],5))` selects five random pages with replacement.
- `page(Pshuf([page1,page2,page3],2))` shuffles once and repeats the same order twice.
- `page(Pxrand([page1,page2,page3],2))` reshuffles the list for each repetition.
- `page(Pchoose([page1,page2,page3]))` chooses one random page each time the cue is triggered.
- Durations can be specified using `pageId:seconds` (e.g.`page1:5`).
- The `after:` clause defines what happens when the sequence ends (e.g.`returning to scroll mode`).
- `after:mode(scroll@F)` jumps to scroll mode at rehearsal mark F and resumes playback.
- `after:mode(scrollPaused@F)` jumps to scroll mode at mark F and stays paused.
- All navigation is synchronized across connected clients.

Examples

`page(page1) page(Pseq([page1:2,page2:2],3)) page(Prand([page1,page2,page3],4)) page(Pxrand([pageA,pageB,pageC],2)) page(Pshuf([pageIntro,pageMid,pageEnd],1)) page(Pchoose([pageA,pageB,pageC])) page(Pseq([page1:3],1), after:mode(scroll@F)) page(Pseq([page1:3],1), after:mode(scrollPaused@F)) page(Pseq([Pseq([page1:2,page2:2],2),page3:4],1))`

Notes

- Pattern parentheses `()` are required for lists and sequences.
- Repetition counts follow the pattern syntax (`Pseq([...],<count>)`).

- Durations follow the page name with a colon (page1:4 = 4 s).
- The @ suffix defines a rehearsal mark or cue ID to jump to after completion.
- Pshuf and Pxrand follow SuperCollider semantics:
 - **Pshuf**: shuffled once, same order repeated.
 - **Pxrand**: reshuffled on each repeat.
- after: supports scroll and scrollPaused modes; :hold(n) extension planned.
- All page and mode changes are fully synchronized across connected clients.

nav() Navigation and Structural Movement

Navigates to a page, rehearsal mark, or scroll mode location.

Syntax

```
nav(<actionOrPage>[ @ <target> ], repeats:<N>?, uid:<id>?)
```

The first argument determines the navigation behavior. It may be: - scroll = switch to scrolling mode and continue playback - scrollPaused = switch to scrolling mode but do not resume playback - <pagename> = jump directly to a page or rehearsal mark

The optional @<target> is used when scroll mode needs to jump to a specific place.

Parameters

Parameter	Description
actionOrPage	"scroll", "scrollPaused", or a page/rehearsal name (e.g., page3, Coda)
target	Optional rehearsal/page anchor used only with scroll modes
repeats	Optional number of times to re-trigger a scroll jump before traversal continues
uid	Unique cue identifier required for repeats; recommended whenever nav() may re-trigger

Examples

```
// direct page navigation
nav(page3)
nav(page3, uid:p3)
nav(Coda)

// scroll navigation
nav(scroll@A)
nav(scrollPaused@B)
nav(scroll)
nav(scrollPaused)

// repeatable scroll navigation
nav(scroll@G, repeats:3, uid:g1)
// jump to G three times; afterwards traversal continues normally
```

UID rule

UID is optional unless repeats are used or multiple identical `nav()` expressions appear. It uniquely identifies navigation state across jump cycles.

Notes

- The performer sees only the visual cue symbol in the score.
- The microsyntax remains in the SVG id and is not shown in the visible notation.

button() Clickable Buttons Inside the Score

The `button()` helper lets you place **HTML buttons** aligned to SVG markers in your score.

Each `button()` marker is an SVG element with an ID that encodes a CueDSL expression; at runtime, Oscilla replaces that marker with a positioned HTML button which can trigger any cue.

Basic Syntax

```
id="button(
  trigger:audio(src:noise, amp:0.9, loop:6, toggle:false, uid:noiseA),
  style(size:"120x45", color:"#333", textcolor:"#fff", fontsize:36, active:"flash", uid:btnNoise)
)"
```

General form:

```
button(
  trigger: <cue expression>,
  style(...optional style keys...),
  offsetX:<number>,
  offsetY:<number>
)
```

- **trigger:** required. Any valid CueDSL expression (e.g.`nav(...)`, `audio(...)`, `page(...)`, `repeat(...)`, etc.).
- **style(...)** optional styling for the HTML button.
- **offsetX**, **offsetY** optional pixel offsets from the SVG marker position.

The `button()` expression must appear as the **ID** of an SVG object:

```
<rect
  id="button(
    trigger:audio(src:noise, amp:0.9, loop:6, toggle:false, uid:noiseA),
    style(size:"120x45", color:"#333", textcolor:"#fff", fontsize:36, active:"flash", uid:btnNoise)
  )"
  x="100" y="40" width="20" height="20"
/>
```

Trigger

The `trigger:` field receives the full cue expression that should run when the button is clicked:

```
button(
  trigger:nav(page:2)
)

button(
  trigger:pause(4)
)

button(
  trigger:audio(src:noise, amp:0.9, loop:6, toggle:false, uid:noiseA)
)
```

At runtime, clicking the button calls `handleCueTrigger(...)` with the parsed trigger AST.

Style Block

The `style(...)` block controls the appearance and behaviour of the HTML button. For example:

```
button(
  trigger:audio(src:noise, amp:0.9, loop:6, toggle:false, uid:noiseA),
```

```

style(
  label:\ "Noise A\ ",
  size:\ "120x45\ ",
  color:\ "#333\ ",
  textcolor:\ "#fff\ ",
  font:\ "Inter\ ",
  fontsize:36,
  active:\ "flash\ ",
  uid:btnNoise
)
)

```

Supported style keys

Key	Type	Description
label	string	Button text override
size	"WxH"	Width height in pixels, e.g. "120x45"
color	string	Background color (CSS color)
textcolor	string	Text color
font	string	Font-family name
fontsize	number	Font size in px
active	string	"flash" to briefly highlight on click
uid	string	Identifier for debugging / CSS hooks

Notes:

- If `style(label:...)` is present, it overrides any auto-generated label.
- If `style(active:\ "flash\ ")` is set, the button gets the `oscilla-button-flashable` class and flashes briefly on click.

Automatic Label Fallback

If no `style(label:...)` is provided, Oscilla derives a label from the trigger:

Priority (roughly):

1. `style(label:\ "...\" )` override
2. `trigger.uid` (if present)
3. `trigger.action` (e.g. "nav", "audio", "page")
4. `trigger.page` (for page cues)
5. "button" as a final fallback

This means even minimal buttons like:

```
button(trigger:nav(page:3))
```

will still get a readable label.

Positioning

Each `button()` marker is an SVG element that defines where the HTML button should appear.

Runtime behaviour:

1. The engine looks for a suitable container element in the DOM:
 - `#singlePage-content`
 - `#pageOverlay`
 - `#scoreInner`

2. It computes the SVG markers bounding box and transform, converts that to screen coordinates, then to container-local coordinates.
3. It creates a `<button>` element with `position:absolute` and sets its `left` and `top` to match the marker position.
4. If the marker is **not** part of a `reuse(...)` clone, the SVG marker is hidden (via `visibility:hidden`) so only the HTML button is visible.

Offsets

You can nudge the button relative to the marker with `offsetX` and `offsetY` in pixels:

```
button(
  trigger:nav(page:4),
  style(label:"Next Page"),
  offsetX:10,
  offsetY:-8
)
```

Containers and Page Mode

Buttons are designed primarily for **page mode** overlays:

- In page mode, `button()` elements will be placed inside `singlePage-content` or `pageOverlay`.
- If the chosen `containerEl` accidentally resolves to an SVG element, Oscilla falls back to `#singlePage-content`.

This ensures that buttons remain correctly aligned even when the score scrolls or resizes.

Click Behaviour

On click, a button:

1. Prevents default and stops event propagation.
2. Optionally flashes (`active:"flash"`).
3. Invokes `handleCueTrigger(triggerAst, false, true, markerElement)` with the trigger AST.

This means `button()` is essentially a **visual trigger surface** for any existing cue type.

Managing Buttons Programmatically

Destroy all generated buttons

```
destroyAllCueButtons();
```

- Removes all `.oscilla-cue-button` elements from the DOM.
- Calls an internal `_destroyCueButton` if present (for cleanup).
- Useful when unloading or reloading a score.

Hide all button() markers in the SVG

```
hideAllButtonPlaceholders(svgRoot);
```

- Finds all markers with `id^="button("` and sets:
 - `opacity: 0`
 - `pointer-events: none`
 - This hides the original SVG placeholders (including clones) while leaving the HTML buttons visible.
-

Summary

- Use `button(trigger:..., style(...))` to add clickable controls into Oscilla scores.
- The SVG marker defines **where** the button appears.
- The `trigger:` field defines **what** happens when the button is clicked.
- The `style(...)` block refines **how** the button looks and responds.
- Buttons integrate seamlessly with existing CueDSL expressions, making them a flexible way to expose navigation, audio, or structural cues as tactile UI widgets.

propagate() Group-Level Parameter Propagation

`propagate(...)` is a pre-parser macro that expands a cue or animation template across all direct children of an SVG `<g>` element. Each child receives a unique instance of the template with independently evaluated parameters.

Purpose

To apply variations of an animation or cue to *multiple similar objects* without manually writing separate IDs. Typical uses:

- evenly distributing random timing or values across many elements
- giving each object slightly different speed, scale, rotation, or other parameters
- generating unique `uid:` values for each item in a group

`propagate(...)` runs **before cue/animation parsing** and rewrites child element IDs into fully expanded DSL strings.

Syntax

```
propagate(
  TEMPLATE,
  ARG1,
  ARG2,
  ...
)
```

TEMPLATE

A new-DSL animation or cue definition, e.g.:

```
scale(values:[$1, $2], mode:alternate, dur:$3, uid:circs)
```

`$n` placeholders

Inside the template, `$1`, `$2`, `$3`, are replaced with the **evaluated values** of ARG1, ARG2, ARG3, etc. Each **occurrence** is evaluated independently, allowing fresh randomisation.

UID Handling

- If `uid:NAME` appears in the template becomes `uid:NAME_0`, `uid:NAME_1`,
- If no `uid:` is present one is appended automatically:

```
uid:prop_GROUPINDEX_CHILDINDEX
```

Example: `uid:prop_3_2`

Example

Group of 6 circles with unique scale range and timing between **12**:

```
propagate(
  scale(values:[$1, $2], mode:alternate, dur:$3, uid:circs),
  rnd(1,2),          // $1: min scale
  rnd(1,2),          // $2: max scale
  rnd(0.4,1.2)       // $3: duration
)
```

Possible expanded IDs:

```
scale(values:[1.14,1.92], mode:alternate, dur:0.66, uid:circs_0)
scale(values:[1.83,1.21], mode:alternate, dur:1.07, uid:circs_1)
scale(values:[1.05,1.99], mode:alternate, dur:0.48, uid:circs_2)
...
```

Evaluation Rules

- Arguments (ARG1, ARG2,) may contain:
 - `rnd(min,max)`
 - `rnd([a,b,c])`
 - numeric literals
 - string literals
- Each `${n}` in the template triggers an independent evaluation of ARGn.

This enables effects like:

```
scale(values:[${1},${1}], dur:${2})
```

Where the two `${1}` occurrences receive **different random values**.

Non-Recursive

Only **direct children** of the `<g id="propagate(...)">` are expanded.

Nested groups are left as-is unless you manually apply propagate to them.

Execution Order

`propagate(svgElement)` must run **before any animation or cue parsing**, typically inside `initializeSVG()`:

```
// Expand propagate(...) groups before parsing
propagate(svgElement);
```

If used in page overlays, it must run before `animationAssign(...)` and cue scanning.

Supported Contexts

- Main score SVG
 - Subpages / page mode SVG
 - Cue-triggered dynamic page loads
-

When to Use

Use `propagate(...)` when:

- you have many similar shapes needing slight variation
 - you want per-object timing/scale/rotation differences
 - you want a template-based param generator for groups
 - you want unique uids for each instance automatically
-

When Not to Use

Avoid propagate when:

- objects require coordinated timing (use shared `seqdur` instead)
 - nested group structure must remain untouched
 - ordering or indexing inside the SVG is unpredictable
-

Notes

- `propagate(...)` is **not** parsed by the DSL directly.
- It is a *preprocessor* that rewrites IDs into pure DSL expressions.
- After expansion, each child is parsed as if you wrote it manually.

Version Compatibility

This documentation refers to the **new Oscilla DSL (2025)** with:

- function-style cues
- named parameters
- canonical `uid`: handling
- no legacy `_uid(...)` microsyntax

Reusable Blocks with `use(name)`

`use(name)` allows you to define a visual or interactive structure once and reuse it anywhere in the score or on any page. This avoids copy/paste in Inkscape and keeps layouts consistent.

Defining a Reusable Block

Any SVG group whose ID begins with:

`reuse-`

is treated as a reusable block.

Example:

```
<g id="reuse-mainMenu">
  <!-- Buttons, text, shapes, animations -->
</g>
```

This block is registered automatically when its SVG is loaded.

Using the Block Elsewhere

To insert the reusable block somewhere else, place a placeholder group:

```
<g id="use(mainMenu)"></g>
```

When the score initializes, the placeholder is replaced with a cloned instance of the reusable block, preserving:

- `cueButtons`
- `cueText` elements
- animations and object-followers
- internal transforms and visual structure

The placeholder itself is removed.

Placement Modes

There are two placement modes depending on how `use()` is written:

Syntax	Placement Behaviour
<code>use(name)</code>	Inserts the block at its original authored coordinates (as designed in its source SVG).
<code>use(name)@self</code>	Inserts the block at the location of the placeholder , inheriting the placeholders transform.

Example (@self Placement)

```
<g transform="translate(300,200)">
  <g id="use(mainMenu)@self"></g>
</g>
```

Result: mainMenu appears exactly at (300,200).

Behaviour and Guarantees

- The injected block receives **unique IDs** to avoid collisions.
 - The placeholder `<g id="use(...)" />` is **removed** automatically.
 - Reusable blocks can be defined in **any SVG inside /pages/**.
 - All internal behaviour is preserved, including:
 - cue triggers
 - animations (rotate, scale, obj2path)
 - cue buttons and overlays
-

Summary

Define once:

```
<g id="reuse-name">...</g>
```

Use anywhere:

```
<g id="use(name)"></g>
```

Or place at placeholder position:

```
<g id="use(name)@self"></g>
```

This system keeps Oscilla scores modular, clean, and visually authored directly in Inkscape no handwritten layout code required.

cue:text OscillaScore Text Cue Reference

This document describes all features of the OscillaScore text engine, including **slot-based layout**, **two-layer crossfades**, **duration/gap logic**, **looping**, and **ordering**.

1. Basic Syntax

```
text(
  src:<file or inline>,
  order:<seq|rnd>,
  dur:<seconds or range>,
  gap:<seconds or range>,
  cross:<0|1>,
  crossfade:<ms or "%">,
  loop:<0|N>,
  target:<center|self|elementId>,
  offsetX:<px>,
  offsetY:<px>,
  yslots:<N>,
  yoffset:<px>,
  yslotmode:<sequence|singlestep|random|shuffle>,
  autostart:<0|1>,
  style:"CSS..."
)
```

All parameters are optional unless stated.

2. Content Source

src:tzara.txt

Loads a .txt file from the projects text directory.

src:"Hello world"

Inline text string.

Lines may be separated with: - newline () - semicolon (;)

3. Unit Construction

mode:line (*default*)

Each line is shown as a unit.

mode:word

Each word becomes a unit.

mode:char

Each character becomes a unit.

4. Duration & Gap

dur:3

Each unit shows for **3 seconds**.

Ranges

dur:2-4

gap:0.5-1.2

- Duration is chosen per unit.
 - Gaps produce blank time between units **only when cross:0**.
-

5. Ordering

order:seq

Units shown in natural order.

order:rnd

Units shuffled every cycle.

6. Looping

The loop behaviour:

Setting	Meaning
loop:0	Infinite loop
loop:N	Play N cycles
No loop param, order:seq	Play once
No loop param, order:rnd	Loop infinitely

Example infinite loop:

text(src:foo.txt, loop:0)

7. Crossfade System

OscillaScore now uses a **two-layer crossfade design**:

- The **old line** fades out in its slot
- The **new line** fades in in its own slot
- Both remain visible simultaneously during overlap
- No teleporting, no slot-shifting during fade

cross:1

Enables overlapping crossfade.

cross:0

Fade-out blank gap fade-in.

crossfade:"60%"

Fade time = 60% of unit duration.

crossfade:800

Fade time = 800ms.

8. Slot-Based Vertical Layout

Slots override vertical placement completely.

This system enables structured multi-line presentation.

yslots:N

Enable slot layout, N vertical “lanes”.

Example:

yslots:3

Creates:

top

center

bottom

yoffset:<px>

Distance between slots. Default: 100px.

Slot Modes

yslotmode:sequence

1 → 2 → 3 → 1 → 2 → 3...

yslotmode:singlestep

Bounces:

center → next → center → previous → center...

yslotmode:random

Each unit goes to a random slot.

yslotmode:shuffle

Uses each slot once per cycle:

(3,1,2) → (2,3,1) → ...

Behavior

- Each line lives in exactly **one** slot.

- Crossfade overlaps use **two different layers**, one per slot.
- With high crossfade %, multiple lines may be visible across slots.

This allows the reader to finish reading an old line while a new one appears.

9. Positioning

Slots override Y-position entirely.

Horizontal center with offsetX

`left = 50% + offsetX`

If yslots is not enabled:

- `target:self` anchors to cue element
 - `target:<id>` anchors to an SVG element
 - `target:center` (default) centers onscreen
-

10. Examples

Basic

```
text(src:foo.txt, dur:3, autostart:1)
```

Infinite random cycling

```
text(src:foo.txt, order:rnd, dur:2, loop:0)
```

Crossfade with 60% overlap

```
text(src:foo.txt, dur:3, cross:1, crossfade:"60%")
```

Slot-based layout (3 lanes)

```
text(
  src:foo.txt,
  dur:3,
  cross:1,
  crossfade:"70%",
  yslots:3,
  yoffset:130,
  yslotmode:singlestep,
  autostart:1
)
```

Sequential slot cycling

```
text(src:foo.txt, yslots:3, yslotmode:sequence)
```

Random slots, no crossfade

```
text(
  src:foo.txt,
  yslots:3,
  yslotmode:random,
  cross:0,
  gap:1,
  dur:2
)
```

11. Interaction

Clicking the text overlay cancels the sequence.

persist:1

Prevents the overlay from being removed during `stopAllCueTexts()`.

12. Stopping All Text

`stopAllCueTexts()`

Removes all **non-persistent** text overlays.

13. Summary Table

Param	Description
<code>src</code>	File or inline text
<code>order</code>	seq or rnd
<code>dur</code>	seconds or range
<code>gap</code>	seconds or range (non-cross only)
<code>cross</code>	1=overlap, 0=fade-out/in
<code>crossfade</code>	ms or percent
<code>loop</code>	0 infinite, N cycles
<code>mode</code>	line / word / char
<code>yslots</code>	number of vertical lanes
<code>yoffset</code>	px between lanes
<code>yslotmode</code>	sequence, singlestep, random, shuffle
<code>offsetX/Y</code>	px offsets
<code>autostart</code>	begin immediately
<code>persist</code>	survive global stop

End of Document

Audio Cues `audio`, `audioPool`, `audioImpulse`

Oscillas audio system provides three related cue types:

- **`audio()`** play a specific file
- **`audioPool()`** select one file from a discovered folder
- **`audioImpulse()`** stochastic repeated triggering from a pool

All cues use generic `key:value` parameter syntax.

1. `audio(...)` Play a Single File

Lowest-level audio playback building block.

Example

```
audio(src:kick, amp:0.9, loop:1, fade:0.3, pan:-0.5, pitch:1.2)
```

Parameters

Key	Meaning
<code>src</code> (required)	filename/stem, .wav auto-added
<code>amp</code>	gain 01 (default 1)
<code>pan</code>	stereo position -1 (left) to 1 (right), default 0

Key	Meaning
pitch	playback rate multiplier (default 1). Values < 1 slow down, > 1 speed up
loop	1=once, N>1 repeats, 0=infinite
fade	applies to both fadeIn and fadeOut
fadeIn	fade-in seconds or percentage (e.g. "20%")
fadeOut	fade-out seconds or percentage (e.g. "50%")
toggle	second trigger stops instead of restarting
uid	playback identity (default = src)

Files load from the project /audio folder, falling back to shared /audio.

Playback state is tracked so UI and buttons can reflect on/off.

2. audioPool(...) One-Shot Selection From a Folder

A pool is built dynamically by scanning a folder you never list filenames manually.

Example

```
audioPool(
  path:sfx/birds,
  format:wav,
  mode:shuffle,
  amp:rand(0.2, 0.8),
  pan:rand(-1, 1),
  pitch:rand(0.8, 1.2),
  fadein:0.05,
  fadeout:"30%",
  poly:4,
  uid:birdsA,
  osc:1,
  oscaddr: "/audio/pool/birds"
)
```

Behaviour

- Server enumerates files in path:
- Every trigger selects **one** file
- Optional randomisation per trigger
- Overlays show filename and evaluated params (amp, pan, pitch)
- Optional OSC mirror sends each hit to external software

Parameters

Key	Meaning
path (required)	folder inside project audio
glob	optional filter hint
format	extension (default wav)
mode	shuffle (no repeat until exhausted) or rand (pure random)
amp	number or rand(a, b)
pan	stereo -1 to 1, or rand(-1, 1)
pitch	playback rate, or rand(a, b)
fadein	seconds or percentage string
fadeout	seconds or percentage string
fade	shorthand for both fadein and fadeout

Key	Meaning
loop	loop the selected file
poly	overlapping voices (default 1, 0=unlimited)
uid	identity of this pool
osc	enable OSC mirroring (1=on)
oscaddr	custom OSC address

Polyphony applies **per pool**.

3. audioImpulse(...) Stochastic Repeating Process

Uses the same pool logic, but runs autonomously and keeps firing hits at a configurable rate.

Example

```
audioImpulse(
  path:sfx/rain,
  rate:40,
  jitter:0.4,
  amp:rand(0.1, 0.5),
  pan:rand(-1, 1),
  pitch:rand(0.5, 2),
  fadein:0.1,
  fadeout:rand("10%", "60%"),
  poly:6,
  lifetime:region,
  uid:rainfall,
  osc:1,
  oscaddr:"/audio/impulse/rain"
)
```

Timing Parameters

Key	Meaning
rate	events per minute
jitter	timing randomisation 01 (0=steady, 1=fully random)
poly	overlapping voices (default 6, 0=unlimited)

Sound Parameters

Key	Meaning
amp	gain 01, or rand(a, b)
pan	stereo -1 to 1, or rand(-1, 1)
pitch	playback rate, or rand(a, b)
fadein	seconds or percentage
fadeout	seconds or percentage
fade	shorthand for both

Lifetime Modes

Value	Behaviour
process	runs until explicitly stopped

Value	Behaviour
region	runs only while playhead is inside the cue element

In region mode, overlays update live and disappear when leaving the region.

OSC Parameters

Key	Meaning
osc	enable OSC mirroring (1=on)
oscaddr	custom OSC address (default /oscilla/audio/impulse)

Fade Values

Fades can be specified as:

Format	Example	Meaning
Seconds	fadeout:0.5	0.5 second fade
Percentage	fadeout:"50%"	50% of the files duration
Random seconds	fadeout:rand(0.1, 0.5)	random between 0.10.5 seconds
Random percentage	fadeout:rand("10%", "60%")	random between 10%60% of duration

Percentage-based fades automatically adjust to each files actual duration (accounting for pitch changes).

Random Expressions

Evaluated **per hit**:

```
amp:rand(0.2, 0.9)      // random amplitude
pan:rand(-1, 1)         // random stereo position
pitch:rand(0.5, 2)      // random playback speed
fadeout:rand(0.05, 0.3) // random fade in seconds
fadeout:rand("10%", "50%") // random fade as percentage
```

For integer random values, use irand(a, b).

OSC Output

When osc:1 is enabled, each audio hit sends an OSC message.

Message Format

```
/oscilla/audio/impulse sfffff filename amp pan pitch fadeIn fadeOut
/oscilla/audio/pool    sfffff filename amp pan pitch fadeIn fadeOut
```

Custom Address

Use oscaddr to specify a custom OSC path:

```
audioImpulse(path:sfx, osc:1, oscaddr:"/myapp/texture/rain")
```

This sends to /oscilla/myapp/texture/rain.

Example OSC Output

```
/oscilla/audio/impulse/x1 sfffff "sfx/birdgone.wav" 0.45 -0.32 1.2 0.2 0.8
```

Arguments: filename (string), amp, pan, pitch, fadeIn, fadeOut (all floats)

Stopping

- Internal stop on fade completion or toggle
 - Region exit automatically cleans up the process
 - Programmatic helpers:
 - `stopAudioImpulse(uid)` stop a specific impulse process
 - `stopAllAudio()` stop all audio playback
-

Visual Overlays

Audio cues display overlays showing:

- **audioPool**: filename and params, auto-destroys after 1.5s
 - **audioImpulse**: persistent overlay showing current file, amp, pan, pitch updates each hit, destroys on region exit
-

Quick Examples

```
// Simple drone with slow fade
audio(src:drone, loop:0, fade:2)

// Percussive one-shots with stereo spread
audioPool(path:sfx/wood, mode:shuffle, pan:rand(-0.8, 0.8), poly:5)

// Pitched-down atmosphere
audio(src:atmosphere, pitch:0.5, loop:0, fadeIn:3, fadeOut:5)

// Rainfall texture inside a passage with OSC output
audioImpulse(
  path:sfx/rain,
  rate:30,
  jitter:0.5,
  amp:rand(0.2, 0.6),
  pan:rand(-1, 1),
  pitch:rand(0.8, 1.3),
  fadeout:"40%",
  lifetime:region,
  osc:1,
  oscaddr:"/texture/rain"
)

// Bird sounds with percentage-based random fades
audioImpulse(
  path:sfx/birds,
  rate:20,
  amp:rand(0.4, 1),
  pan:rand(-1, 1),
  pitch:rand(0.1, 2),
  fadeIn:0.2,
  fadeout:rand("1%", "60%"),
  lifetime:region,
  uid:birdgroup,
  osc:1,
  oscaddr:"/audio/birds"
```

)

video()

Purpose: Spawn and control HTML5 video overlays during a score.

Required - file: Filename with optional extension (.mp4 default if missing). Supports mp4|webm|ogg.

Placement & Positioning - location: fixed (default) | scroll - fixed pinned to viewport. - scroll follows target/score scroll. - target: <elementId> centers the video on the target (if not fullscreen). - offsetX: / offsetY: pixel offsets applied after centering.

Size - size: - fs OR fullscreen covers viewport at (0,0) with 100vw×100vh and **passes clicks through** by default. - <W> (e.g.640) width in px, auto height. - <W>×<H> (e.g.640×360).

Audio (Default Muted) - audio: 0|1|true|false **default is muted**. Set audio:1 (or true) to unmute.

Spawning / Reuse - By default, cues **reuse** an existing video matching the same file + target. - uid: <string> OR new:1 **force a new concurrent instance** (even if one exists). - Every instance receives data-uid for tracking; data-key = file_target.

Timing & Playback - in: seconds (seek start) - out: seconds (optional fade-out scheduling; pairs well with fadeOut) - hold: seconds (auto-remove after this duration, regardless of loop) - loop: 0 = infinite; N = loop count; omit = play once - speed: playback rate (e.g.0.5, 1.25) - opacity: 0..1 (default 1) - fadeIn: seconds; fadeOut: seconds

Interaction - Fullscreen (size:fs): uses pointer-events:none so the score behind remains draggable/clickable. - Add clickable:1 to allow clicks **on** the fullscreen video (e.g., to close on click). - Non-fullscreen videos remain clickable; single-click removes them by default.

Removal - Ends when: - playback completes (no loop) **or** - loop count is reached **or** - hold expires **or** - user clicks (non-fs) / double-click overlay (if you implement) / explicit cue cleanup.

Examples

1. Fullscreen, pass-through clicks, but allow click-to-close

```
cue:video(file:intro.mp4,size:fs,clickable:1,audio:1,fadeIn:0.5)
```

2. Windowed, anchored to a target, follows scroll, infinite loop

```
cue:video(file:clip.webm,target:markerA,location:scroll,size:640x360,loop:0,opacity:0.9)
```

3. Spawn a second independent instance via uid

```
cue:video(file:cam.mp4,uid:stageLeft,in:5,fadeIn:1,fadeOut:1,hold:20)
```

4. Force new instance without specifying uid

```
cue:video(file:teaser.mp4,new:1,in:2,out:10,fadeIn:0.5,fadeOut:0.5,speed:1.25)
```

***Note:** size:fs always positions at (0,0) and ignores target geometry; all other sizes center on target (or the cue position) with optional offsetX/offsetY. Default audio is muted; use audio:1 to unmute. By default, same file+target cues reuse the existing element unless uid or new:1 is supplied.*

synth() Web Audio Synth Cue

Oscilla “Native WebAudio Utility Synth”

The synth() cue creates a lightweight Web Audio sound source inside Oscilla. It is intended for **reference tones, drones, textures, chords, and simple patterned sound processes**, integrated directly into the score timeline.

The design prioritises:

- deterministic, cue-scoped behaviour
- readable parameter syntax
- low default amplitudes and click-free envelopes
- optional pattern sequencing
- optional OSC mirroring

The synth is not intended as a full synthesiser environment. For full electroacoustic work, Oscilla is designed to operate in conjunction with external audio systems via OSC (e.g.SuperCollider, Pure Data,

Max). The built-in synth provides a bounded, score-aligned sound source intended for rehearsal contexts away from a complete setup, as well as for simple tone cues, drones, and basic animation- or pattern-driven sound sequences embedded directly in the score.

Basic Usage

```
synth(uid:refA, wave:sine, freq:440, amp:0.1)
```

Pitch may be specified as Hz or note name:

```
synth(uid:tuning, wave:sine, freq:A4, amp:0.08)
```

Wave Types

The following wave values are supported:

```
sine
square
saw
triangle
noise
```

Oscillator waveforms map directly to standard Web Audio oscillator types.

noise produces a broadband noise source generated from a looping audio buffer. No spectral colouring is applied.

Lifetime and Duration

Region-based lifetime (default)

When `synth()` is attached to a cue element, the synth starts when the playhead enters the cues bounding box and stops when the playhead exits it.

```
synth(uid:regionTone, wave:sine, freq:220, amp:0.06)
```

Explicit duration

```
synth(uid:fixedDur, wave:sine, freq:220, dur:5, amp:0.07)
```

The synth stops automatically after the specified number of seconds.

Persistent process lifetime

```
synth(uid:persist, wave:saw, freq:110, lifetime:process, amp:0.05)
```

The synth continues until explicitly stopped.

Amplitude Envelope (ADSR)

```
synth(
  uid:env1,
  wave:saw,
  freq:220,
  amp:0.12,
  env:{a:0.5, d:0.2, s:0.7, r:1.2}
)
```

Envelope parameters:

- a attack (seconds)
 - d decay (seconds)
 - s sustain level (01)
 - r release (seconds)
-

Chords

If `freq` is an array, multiple oscillators are created and summed as a chord.

```
synth(
  uid:pad3,
  wave:sine,
  freq:[440, 477, 644, 777],
  env:{a:1.5},
  amp:0.12
)
```

All voices share the same envelope and effects chain.

Patterned Parameters

The following parameters may be static values, arrays, or pattern functions:

- `freq`
- `amp`
- `dur`
- `filter.freq`
- `filter.q`
- `pan`

Frequency sequence

```
synth(
  uid:seq1,
  wave:triangle,
  freq:Pseq(220, 330, 440),
  dur:Pseq(1, 1, 2),
  amp:0.08
)
```

Patterned chord progression

```
synth(
  uid:chordSeq,
  wave:saw,
  freq:Pseq(
    [220, 330, 440],
    [247, 370, 494],
    [196, 294, 392]
  ),
  dur:1.5,
  amp:0.1
)
```

Random pitch

```
synth(
  uid:randFreq,
  wave:square,
  freq:Prand(200, 400, 600),
  dur:0.8,
  amp:0.09
)
```

Filters

```
synth(
  uid:filterSeq,
  wave:saw,
  freq:330,
  filter:{type:lp, freq:Pseq(400, 800, 1600, 800), q:0.7},
  dur:0.5,
  amp:0.08
)
```

Supported filter types:

```
lp
hp
bp
notch
```

Delay

```
synth(
  uid:delayLead,
  wave:square,
  freq:550,
  delay:{time:0.25, fb:0.35, mix:0.2},

```

```
    amp:0.12
  )
```

Reverb

```
synth(
  uid:revDrone,
  wave:sine,
  freq:110,
  reverb:{mix:0.3, time:2, damp:3000},
  amp:0.07
)
```

Glide (Frequency Only)

```
synth(
  uid:glide1,
  wave:sine,
  freq:Pseq(220, 330, 440),
  glide:0.1,
  dur:1,
  amp:0.1
)
```

glide specifies the time (seconds) used to interpolate frequency changes between steps. It applies only to pitch.

Interpolation Mode

```
synth(
  uid:stepSeq,
  wave:triangle,
  freq:Pshuf(300, 450, 600),
  interp:step,
  dur:1,
  amp:0.08
)
```

- `interp:smooth` ramps between values
 - `interp:step` immediate value changes
-

Pan

```
synth(
  uid:panTest,
  wave:sine,
  freq:440,
  pan:-0.6,
  amp:0.09
)
```

Pan may be static or patterned but does not support continuous modulation.

OSC Mirroring

```
synth(
  uid:oscLead,
  osc:1,
  oscAddr:/synth/lead,
  wave:saw,
  freq:440,
  amp:0.1
)
```

Stopping a Synth

```
synthStop(uid:seq1, rel:0.5)
```

The optional `rel` parameter specifies release time in seconds.

Summary

The `synth()` cue provides a bounded, score-aligned sound source suitable for reference tones, drones, chords, and simple patterned textures. All behaviour is deterministic and cue-scoped, and integrates directly with Oscillas timing and animation model.

cue:osc Discrete OSC Event Cue

`osc(...)` sends a **single OSC message** when triggered.

It is **event-based**, not continuous. It turns drawn objects into **discrete control events**, typically consumed by environments like SuperCollider, Pd, or Max.

It works especially well with `propagate()`, because a simple drawing can become many individual OSC triggers.

BASIC FORM

`osc(addr:<name>, <param>:<source>, ...)`

Required:

Key	Meaning
<code>addr</code>	OSC address name (without leading /)

Example:

`osc(addr:voice, pitch:y, env:size)`

EVENT BEHAVIOUR

- Sends **one OSC message per trigger**
- No looping, no animation, no scheduler
- Values are sampled **at the moment of triggering**
- Cue completes immediately

`osc()` is a *logical cue*, not an animation cue.

VISUAL PARAMETER SOURCES (NORMALISED)

All visual sources are normalised to 0–1.

Source	Meaning
<code>x</code>	Object centre X position
<code>y</code>	Object centre Y position
<code>size</code>	Max(width, height) mapped to viewport
<code>scale</code>	Current visual scale value
<code>rotation</code>	0360 mapped to 01
<code>opacity</code>	Computed opacity
<code>fill</code>	Numeric colour hash

Example:

`osc(addr:voice, pitch:y, amp:size)`

Here the Y-axis produces a **continuous control pitch**, interpreted downstream.

SEMANTIC PITCH VALUES

Beyond visual values, `osc()` supports **typed pitch declarations**.

Hz (absolute)

```
osc(addr:voice, pitch:hz(440))
```

MIDI note number

```
osc(addr:voice, pitch:midi(60))
```

Scale degree + octave

```
osc(addr:voice, pitch:deg(5,3))
```

Represents:

- scale degree = 5
- absolute octave = 3

The receiver chooses how to interpret the scale.

OPTIONAL ROOT OVERRIDE

You may specify a **root** in the cue:

```
osc(addr:voice, pitch:deg(2,4), root:48)
```

If omitted, the receiver uses its global default (`~oscRoot` in SC).

PITCH PRIORITY

When multiple forms are present, resolution is:

1. `deg(d,o)` (scale-based symbolic pitch)
2. `hz()` / `midi()` (absolute pitch)
3. visual mapping (e.g., `pitch:y`)

Only one pitch representation is used per event.

TRIGGERS

Key	Meaning
<code>trig:auto</code>	Send immediately
<code>trig:playhead</code>	Fire when playhead intersects
<code>trig:click</code>	Fire on user click
<code>prestate:ghostClickable</code>	Arm interaction ahead of time

Example:

```
osc(addr:voice, pitch:deg(0,4), trig:playhead)
```

UID (OPTIONAL)

```
osc(addr:voice, pitch:y, uid:v1)
```

UID can be used downstream to associate events with voices/players/etc.

OSC OUTPUT FORMAT (AUTHORITATIVE)

`osc()` currently sends **positional arguments**, not keyed pairs.

GENERAL PACKET FORMAT

```
/oscilla/<addr>
  pitchType, pitchA, pitchB, size, env, density, [optional root]
```

Where:

Field	Meaning
pitchType	0=none, 1=deg/oct, 2=hZ, 3=midi, 4=y-control
pitchA	meaning depends on pitchType
pitchB	octave (deg mode only)
size	01
env	01
density	01 (area-derived)
root	optional per-event root override

Examples

Control pitch from Y

```
/oscilla/voice
1 5 3 0.41 0.41 0.12
```

Absolute pitch

```
/oscilla/voice
2 440 0 0.18 0.22 0.10
```

Degree + octave + root override

```
/oscilla/voice
1 5 3 0.33 0.20 0.15 48
```

Receivers are expected to decode according to this positional layout.

USE WITH propagate()

```
propagate(
  osc(
    addr:voice,
    pitch:y,
    env:size,
    trig:playhead
  )
)
```

Degree constellation examples

```
propagate(
  osc(
    addr:pontalist,
    pitch:deg(irand(0,11), irand(0,2)),
    env:size
  )
)
```

```
propagate(
  osc(
    addr:pontalist,
    pitch:deg(${1}, ${2}),
    env:size,
    root:48
  )
)
```

```

),
rnd([0,2,4,5,7,9,11]),
rnd([0,1,2,3,4,5])
)

```

DESIGN NOTES

- stateless
 - no DOM ID use
 - no continuous OSC streaming
 - no scale-logic baked into DSL
 - musical interpretation lives downstream
 - arguments are positional, compact, and receiver-focused
-

SUMMARY

osc() converts visual gestures into **one-shot OSC control events**, supporting symbolic, absolute, and control-driven pitch while leaving musical semantics to the instrument.

oscCtrl Continuous Control Lanes (OSC)

oscCtrl lets you draw a **path in the score** and use it as a live control lane for audio / electronics via OSC. As the playhead passes over the path, values are mapped and streamed to the audio engine. The path acts like a **breakpoint automation curve** tightly aligned to instrumental notation.

Basic Syntax

```

oscCtrl(
  addr: "/fx/ring/freq",
  min: 60,
  max: 800,
  uid: ringModFreq
)

```

Parameters

Param	Required	Type	Description
addr		string	OSC address to control (joined under /oscilla/control)
min		number	Minimum mapped value (default 0)
max		number	Maximum mapped value (default 1)
uid		string	Unique label (clientside only, debugging)
mode		enum	event (default) or continuous

The cue **must be placed directly on a <path> element**.

What gets sent

Given:

```
addr: "/fx/ring/freq"
```

Oscilla sends:

```
/oscilla/control/fx/ring/freq <value> <t>
```

Where:

- **value** mapped Y position between min max
- **t** normalized X position along the path (01)

Example:

```
/oscilla/control/fx/ring/freq 726.06 0.27
```

Path Semantics

The path defines:

- horizontal = **time alignment** with notation
- vertical = **control value**
- Top of path higher values
- Bottom of path lower values

Zoom, scale, and SVG transforms are handled automatically.

Modes

mode:event (default)

Send only when the value **changes meaningfully**.

```
oscCtrl(
  addr: "/fx/pan",
  min: -1,
  max: 1
)
```

Reduces controller spam and CPU noise.

mode:continuous

Send regularly while the playhead is inside the lane:

```
oscCtrl(
  addr: "/fx/pan",
  min: -1,
  max: 1,
  mode: continuous
)
```

Useful for:

- scrubbing
- FM / modulation
- granular playback positions

Messages are still **throttled to ~30/s**.

Recommended Uses

ring modulation frequency
 panning automation
 filter cutoff / resonance
 spatialization
 cueing electronics transitions

Not Sent

`oscCtrl` does **not** apply smoothing.

This should happen in the **audio client**, where it belongs.

Future Extensions

- `oscCtrlNode()` breakpoint nodes & interpolation
 - custom easing curves (exp / log)
 - curve visualization overlays
-

Troubleshooting

Nothing happens?

Ensure the cue is attached to the **path element itself**, not a group.

OSC not received?

Check the server receives:

type: "osc_control"

and that `/oscilla/control/*` is routed correctly.

Summary

`oscCtrl` brings automationstyle control inside the score while remaining modular and OSCnative.

Compose gestures visually let the audio engine interpret.

Chapter 4

Animation

Animation cues bring graphic score elements to life during playback. Objects can scale, rotate, follow paths, change colour, and fade in or out all driven by the same embedded cue syntax used for transport and media control.

These transformations are not decorative. In an animated graphic score, movement, colour shift, and visual emphasis carry musical meaning: a rotating arrow might indicate which material a performer should play next; a scaling gesture might suggest dynamic intensity; a colour transition might signal a shift in texture or instrumentation. The animation system allows composers to encode these interpretive signals directly into the visual objects that performers read.

Animations can be layered and combined on a single element, and their parameters can be modulated in real time through the control system described in the next section.

cue:scale Scaling Animation

`scale()` adjusts the size of SVG objects using sequences, patterns, continuous pulsing, or independent X/Y scaling.

BASIC FORMS

Uniform scaling

```
scale(values:[1,1.5,1], dur:2)
```

Non-uniform (XY)

```
scaleXY([1,1.3],[1,0.6], dur:1)
```

Continuous pulse

```
scale(min:1, max:1.3, dur:2)
```

TRIGGERING

- automatic page-mode
 - scroll edge
 - cue activation
-

TRIGGER DELAY

tdelay:<seconds>

Delays scaling start.

```
scale(values:[1,1.4,1], tdelay:3)
```

PRESTART VISIBILITY

prestate:<show|hide|ghost>

```
scale([1,1.5,1], tdelay:4, prestate:ghost)
```

Initial scale is applied immediately.

SEQUENCES

Key	Meaning
values	uniform scale list
x, y	independent XY sequences
dur	duration per step
mode	loop / once / alternate
interp	smooth / step
hold	pause (smooth only)

Pattern forms (Pseq, Prand, etc.) supported for all parameters.

CONTINUOUS SCALING

```
scale(min:1, max:1.4, dur:1.5, loop:0)
```

UID Live Updates

```
scale(values:[1,1.2,1], uid:s1)
scale(uid:s1, dur:0.5)
```

FULL PARAMETER LIST

Key	Description
values	scalar sequence
x, y	axis-specific sequences
dur	step duration
hold	pause
ease	tween curve
mode	loop logic
interp	smooth/step
uid	animation identity
trig	trigger
tdelay	delay after trigger
prestate	pre-start visual state

EXAMPLES

```
scale([1,2,1], dur:2, tdelay:3)
scaleXY([1,1.4],[1,0.6], dur:1)
scale(Pseq([1,1.5,1],inf), dur:Prand([0.5,1],inf))
```

rotate() Rotation Cue

rotate(...) rotates an SVG element using either:

- **continuous rotation**, or

- a **sequence / pattern of angle values**

Rotation can be:

- smooth interpolated
- stepped
- looping, alternating, or oneshot
- optionally sending OSC messages (with live overlays).

BASIC FORMS

Continuous

```
rotate(dir:1, dur:1)
```

Sequence

```
rotate(values:[0,120,240], dur:2)
```

Pattern sequences

```
rotate(values:Pseq([0,45,10],inf), dur:Pseq([1,0.2,2],inf))
```

TRIGGERING

- auto (page load / mode)
- edge (playhead collision)
- cue activation

TRIGGER DELAY `tdelay:<seconds>`

Delays motion after trigger:

```
rotate(values:[0,120,240], tdelay:2)
```

PRESTART VISIBILITY `prestate:<show|hide|ghost>`

Controls appearance before rotation begins.

```
rotate(values:[0,90,180], tdelay:4, prestate:ghost)
```

The element adopts the **initial rotation angle immediately**, then waits.

SEQUENCE PARAMETERS

Key	Meaning
values	list of angles or pattern
dur	seconds per step (or pattern)
mode	loop, once, alternate
interp	smooth OR step
hold	pause after tween (smooth only)

CONTINUOUS ROTATION

```
rotate(dir:1, dur:2)
```

```
rotate(dir:-1, dur:3, ease:"easeInOutSine")
```

- dir:1 clockwise
- dir:-1 counterclockwise

- loop:0 (default) = infinite

UID Live Updates

UID lets multiple cues address the same animation:

```
rotate(values:[0,90,180], uid:r1)
rotate(uid:r1, dur:0.5)
```

OSC OUTPUT (optional)

Rotation can send OSC every frame or perstep.

Enable via:

```
osc:1
```

Specify an OSC address:

```
rotate(values:[0,90,180], osc:1, oscaddr: "/spatialisation/source1")
```

Payload format

Every message includes:

Field	Meaning
deg	wrapped degrees 0360
rad	radians ($\text{deg} * \pi/180$)
norm	normalized 01 ($\text{deg}/360$)
uid	animation UID
timestamp	ms since epoch

Example conceptual payload:

```
{
  "type": "osc_rotate",
  "uid": "r1",
  "deg": 135.0,
  "rad": 2.356,
  "norm": 0.375,
  "addr": "/spatialisation/source1"
}
```

*NOTE: Internally angles may accumulate, but outgoing OSC **always wraps to 0360**.*

Onscreen overlays

When OSC is enabled, a small overlay shows:

```
/spatialisation/source1 - deg:135.0
```

Overlays can be globally toggled.

FULL PARAMETER LIST

Key	Description
values	angle sequence or pattern
dur	duration or pattern
mode	loop logic
interp	step or smooth
hold	pause after tween
dir	direction (continuous mode)
ease	easing curve

Key	Description
uid	animation identity
trig	trigger mode
tdelay	trigger delay
prestate	visual prestart state
osc	enable OSC (0/1/2)
oscaddr	explicit OSC address

EXAMPLES

```
rotate(values:[0,120,240], dur:4, tdelay:2)
```

```
rotate(values:Pshuf([0,180],inf),
      dur:1,
      mode:alternate,
      osc:1,
      oscaddr:"/fx/rot1")
```

```
rotate(dir:-1, dur:3, prestate:hide, tdelay:1)
```

Summary

- declarative rotation cue
- works in page + scroll modes
- full control of timing + interpolation
- OSCready with wrapped degrees, radians, and normalized values
- optional debug overlays

cue:o2p Object-to-Path Animation

`o2p(...)` animates an SVG object along a named `<path>`.

It supports path traversal, directional modes, rotation modes, looping, OSC output, partial-path motion, trigger-based activation, delayed start, visual pre-start states, **interactive touch/drag control**, and **grouped fader presets with launcher bars**.

BASIC FORM

```
o2p(path:<id>, dur:<seconds>, mode:fwd)
```

Required:

Key	Meaning
path	The ID of the <code><path></code> element to follow

TIMING

dur:<seconds>

Duration of one full traversal of the path.

```
o2p(path:orbit, dur:6)
```

In mode:alt, the full ABA cycle lasts dur seconds.

tdelay:<seconds>

Delays the start of the animation after the cue triggers.

```
o2p(path:orbit, tdelay:3)
```

tdelay is real-time and independent of score position.

PRESTART VISIBILITY**prestate:<show|hide|ghost>**

Controls how the object looks before the animation begins:

- show fully visible immediately
- hide invisible until start
- ghost semi-transparent until start

The object is positioned at its correct starting location before motion begins.

```
o2p(path:orbit, tdelay:4, prestate:hide)
```

DIRECTION MODES

Mode	Meaning
forward / fwd	0 1 along path
reverse / rev	1 0
alternate / alt	back-and-forth motion

```
o2p(path:curve, mode:alt, dur:8)
```

PATH SEGMENTS

```
o2p(path:ring, start:0.2, end:0.8)
```

The object travels only between normalized positions `start` and `end`.

ROTATION

rotate:	Description
none	no rotation
aligned	align to tangent direction
locked	fixed heading using <code>rotlock</code>
spin	continuous rotation (<code>rotspeed</code> , <code>rotmdir</code>)

Additional keys:

Key	Meaning
rotoffset	add angular offset
rotlock	fixed heading in degrees
rotspeed	seconds per full rotation

Key	Meaning
rotDir	1 direction multiplier

LOOPING

```
o2p(path:orbit, loop:3)
o2p(path:orbit, loop:0) // infinite
```

OSC OUTPUT

```
o2p(path:orbit, osc:true)
Emits:
/o2p/<uid> <pathT> <normX> <normY> <angle>
Custom OSC address:
o2p(path:orbit, osc:true, oscAddr:myController)
Emits:
/myController <pathT> <normX> <normY> <angle>
```

TRIGGERING

Trigger	Behavior
trig:auto	starts automatically in page mode
trig:edge	starts when the playhead intersects in scroll mode
trig:touch	no auto-animation; object is draggable along path

Cues may also be activated programmatically.
tdelay applies after the trigger is detected.

TOUCH MODE (Interactive Control)

The trig:touch mode transforms the o2p animation into an **interactive controller**. Instead of automatically animating, the object waits for user interaction.

```
o2p(path:fader, trig:touch, osc:true)
```

Behavior: - Object is positioned at the start point (or start: position) - No automatic animation occurs
- User can **grab and drag** the hit-label overlay - Object follows the users finger/mouse along the path - If osc:true, every movement emits OSC values

Use Cases: - Browser-based OSC faders/sliders - XY pads with constrained motion - Rotary knobs (using circular paths) - Custom UI controllers that follow any SVG path shape

Example Linear Fader:

```
o2p(path:verticalLine, trig:touch, osc:true, oscAddr:fader1)
```

Example Circular Knob:

```
o2p(path:knobArc, trig:touch, start:0.1, end:0.9, osc:true, oscAddr:knob1)
```

Visual Indicator: Touch mode objects display an **orange** hit-label ring (instead of purple) to indicate they are draggable.

FADER GROUPS AND PRESETS

Touch-mode faders can be collected into named **groups**. A group enables saving and recalling all fader positions as named presets, tweening between presets with easing, and sequencing preset recalls over time.

Grouping Faders

Add `group:<name>` and `uid:<id>` to each touch-mode fader:

```
o2p(path:track1, trig:touch, group:mixer1, uid:vol, osc:true)
o2p(path:track2, trig:touch, group:mixer1, uid:pan, osc:true)
o2p(path:track3, trig:touch, group:mixer1, uid:send, osc:true)
```

All three faders register under `window._o2pTouchGroups["mixer1"]` and can be saved/recalled as a unit. The `group` value is freeform any string.

Launcher Bar

When a group registers, a **launcher bar** appears below the groups bounding box. It provides direct access to presets and sequences without opening the full preset panel.

The launcher includes:

- **Preset/Sequence slots** left-click to recall, long-press to store current positions
- **Bank navigation** arrow buttons to page through banks of slots
- **Mode toggle** (P/S) switch between preset and sequence slot modes
- **Tween toggle** (~) smooth interpolation or instant jump on recall
- **Sequence transport** play/stop buttons for sequence playback
- **Settings gear** opens the full o2p Preset Manager panel

An **orange toggle circle** appears to the right of the group. Click it to show or hide the launcher bar.

Preset Manager Panel

The preset panel (keyboard shortcut **Alt+Shift+O**, or gear button on launcher) provides full preset management:

- **Presets tab** save, recall, delete named presets with tween duration and easing
- **Sequences tab** define ordered lists of presets with per-step timing
- **Import/Export** save and load preset collections as JSON files

Console access: `window.o2pPresetUI.toggle()`

Preset Data

Each preset stores the normalized position of every fader in the group:

```
{
  "mixer1": {
    "vol": { "t": 0.75 },
    "pan": { "t": 0.3 },
    "send": { "t": 0.5 }
  }
}
```

If a fader has a rotation handle (`hmode:`), a `p` value is also stored:

```
{ "vol": { "t": 0.75, "p": 0.6 } }
```

Console API

```
window.o2pPresets.save("intro")           // save current positions
window.o2pPresets.recall("intro")         // instant recall
window.o2pPresets.recall("intro", { dur: 2, ease: "easeInOutSine" }) // tween
window.o2pPresets.list()                  // list all preset names
window.o2pPresets.delete("intro")         // delete a preset

// Sequences
window.o2pPresets.defineSequence("drift", [
  { preset: "intro", dur: 3 },
  { preset: "verse", dur: 2 },
  { preset: "chorus", dur: 4 }
])
```

```
], { loop: true })
window.o2pPresets.playSequence("drift")
window.o2pPresets.stopSequence()
```

ROTATION HANDLES

Touch-mode faders can have a secondary **rotation handle** that adds a second control dimension. This is useful for parameters like pan, filter cutoff, or any value that benefits from a rotary control alongside the linear fader.

hmode: <continuous|limited>

Determines the rotation behavior:

- continuous full 360-degree rotation with wraparound (suitable for azimuth, phase)
- limited constrained arc with hard stops like a physical potentiometer (suitable for volume, gain)

rotrange: <degrees>

Sets the arc range for hmode:limited. Default is 270 degrees.

```
o2p(path:track1, trig:touch, group:mixer1, uid:vol, hmode:limited, rotrange:270, osc:true)
```

handle: <elementId>

Reference a user-drawn SVG element as the rotation handle instead of the auto-generated one:

```
o2p(path:track1, trig:touch, group:mixer1, uid:vol, hmode:limited, handle:knob1, osc:true)
```

Where knob1 is the ID of an existing SVG element in the score.

LIVE UPDATE (uid:)

```
o2p(path:ring, uid:a1)
```

```
o2p(uid:a1, mode:alt)
```

Later cues with the same uid modify running animations.

FULL PARAMETER LIST

Key	Description
path	required path reference
dur	motion duration
mode	direction mode
loop	repeat count (0 = infinite)
start, end	normalized path range
rotate	rotation behavior
rotoffset, rotlock, rotspeed, rotmdir	rotation parameters
ease	easing preset
osc	OSC emission
oscAddr	custom OSC address (without leading /)
uid	animation identity tag
trig	trigger mode (auto, edge, touch)
tdelay	time delay after trigger
prestate	show/hide/ghost before start
group	fader group name (touch mode only)
hmode	rotation handle mode: continuous OR limited (touch mode only)

Key	Description
rotrange	arc range in degrees for <code>hmode:limited</code> (default 270)
handle	ID of user-drawn SVG element to use as rotation handle

EXAMPLES

```

o2p(path:orbitA, dur:8, tdelay:3, prestate:hide)
o2p(path:spiral, rotate:aligned, rotoffset:-90)
o2p(path:ring, start:0.2, end:0.9, mode:alt)
o2p(path:circle, rotate:spin, rotspeed:2, rotdir:-1)

// Touch mode controllers
o2p(path:faderTrack, trig:touch, osc:true, oscAddr:volume)
o2p(path:knobPath, trig:touch, start:0.15, end:0.85, osc:true)

// Grouped faders with preset system
o2p(path:track1, trig:touch, group:mixer1, uid:vol, osc:true)
o2p(path:track2, trig:touch, group:mixer1, uid:pan, osc:true)
o2p(path:track3, trig:touch, group:mixer1, uid:send, osc:true)

// Grouped fader with rotation handle
o2p(path:track1, trig:touch, group:mixer1, uid:vol, hmode:limited, rotrange:270, osc:true)

// Grouped fader with user-drawn rotation handle
o2p(path:track1, trig:touch, group:mixer1, uid:vol, hmode:continuous, handle:knob1, osc:true)

```

cueTraverse Animate Objects Between SVG Points

The `cueTraverse(...)` cue animates an SVG object along a path defined by a list of point elements (`<circle>`, `<rect>`, or `<ellipse>`). Its useful for moving symbols, playheads, or visual elements dynamically across the score.

Syntax

```
cue_traverse_p(p1,p2,p3)_o(objectId)_s(speed)_d(direction)_x(repeats)_e(easeType)_h(holdMs)_next(targetCue)
```

Each parameter is enclosed in its own parenthesis. The cue name and parameters can be written as a single underscore-delimited string.

Parameter Reference

Param	Meaning
<code>p(...)</code>	List of point IDs (e.g. <code>p(p1,p2,p3)</code>)
<code>o(...)</code>	Object ID to move
<code>s(...)</code>	Speed in pixels per second (e.g. <code>s(2.5)</code>)
<code>d(...)</code>	Direction: 0=forward, 1=reverse, 2=pingpong, 3=random
<code>x(...)</code>	Number of repeats (0 = infinite)
<code>e(...)</code>	Easing type (0 = linear, 19 = anime.js presets)
<code>h(...)</code>	Hold time at each point in milliseconds
<code>_next(...)</code>	Optional cue to trigger after traversal ends

Examples

Traverse between 3 points once:

```
cue_traverse_p(p1,p2,p3)_o(dot)_s(1.5)_x(1)
```

Infinite pingpong movement:

```
cue_traverse_p(p1,p2,p3)_o(dot)_d(2)_x(0)
```

Traverse with hold, ease, and jump to another cue:

```
cue_traverse_p(A,B,C)_o(cursor)_s(3)_e(2)_h(300)_next(cue_done)
```

Notes

- Objects must be <circle>, <ellipse>, or <rect> to be correctly positioned.
 - p(...) must contain **at least two** valid point IDs.
 - _next(...) cue will only be triggered after **final traversal loop** finishes (or never, if infinite).
 - Traversal will pause playback if isPlaying is false at start.
-

Triggering Other Objects via Intersections

As the animated object (defined via o(...)) moves through the points listed in p(...), the system checks whether any point ID matches the data-id of other SVG elements.

When a match is found:

- The corresponding element is **animated using its data-id definition**
- This allows cueTraverse to function like a **trigger engine**
- Its useful for triggering visual pulses, spins, or dynamic elements connected to spatial locations in the score

Example

```
c-t_p(p1,p2,p3)_o(pulsetrig01)
```

If you have an element like:

```
<circle id="pulse" data-id="pulsetrig01 obj_scale_seq[1,1.5,1]_bounce" />
```

When the animated object reaches a point named pulsetrig01, the element above will animate according to its data-id.

color() / colour() Colour Animation Cue

Animates the visual colour of SVG elements over time using HSL interpolation.

Both spellings are fully equivalent:

```
color(...) □ colour(...)
```

Quick Examples

```
// Cycle through three colours
```

```
color(vals:[#f00, #0f0, #00f], dur:3)
```

```
// Continuous rainbow hue rotation
```

```
color(vals:hue, dur:10, loop:0)
```

```
// Alternating palette (ping-pong)
```

```
color(vals:[#f80, #08f], mode:alt, dur:1.2)
```

```
// Pattern-driven sequence
```

```
color(vals:Pseq([red, yellow, cyan], 4), dur:0.5)
```

Basic Syntax

```
color(
    vals:[...],      // colour values or hue keyword
    dur:seconds,     // duration per transition
    mode:mode,       // interpolation mode
    loop:n           // repetitions (0 = infinite)
)
```

Parameters**vals: (required)**

Defines the colour values or behaviour.

Discrete Colour Values

Supported formats: - **Hex:** #f00, #ff8800 - **RGB:** rgb(255, 0, 0) - **HSL:** hsl(120, 80%, 50%) - **Named:** red, cyan, magenta

```
color(vals:[#f00, #0f0, #00f], dur:2)
color(vals:[red, yellow, green], dur:3)
```

Hue Cycling (Continuous)

Full 360 hue rotation:

```
color(vals:hue, dur:6)
```

Range-limited cycle:

```
color(vals:hue(120, 240), dur:4)
```

This cycles only through the green-to-blue range.

Pattern Sequences

```
color(vals:Pseq([#f00, #ff0, #0ff], 3), dur:1)
color(vals:Prand([#f80, #08f, #8f0], inf), dur:0.8)
```

dur:

Duration in seconds for one transition (or one full hue cycle).

```
color(vals:[#f00, #0f0], dur:2)           // 2 seconds per transition
color(vals:hue, dur:10)                   // 10 seconds per full rotation
```

Duration can also be a pattern:

```
color(vals:[#f00, #0f0, #00f], dur:Pseq([1, 0.5, 2], inf))
```

mode:

Controls interpolation behaviour.

Mode	Behaviour
linear	Smooth interpolation (default)
alt	Ping-pong (forward then reverse)
step	Hard switching, no interpolation

```
// Smooth transitions
color(vals:[#f00, #0f0], dur:2, mode:linear)
```

```
// Bounce back and forth
color(vals:[#f00, #0f0], dur:2, mode:alt)
```

```
// Instant colour changes
color(vals:[#f00, #0f0, #00f], dur:0.5, mode:step)
```

loop:

Number of repetitions.

```
loop:3    // repeat three times then stop
loop:0    // infinite (default for hue cycling)
loop:1    // play once
```

uid:

Unique identifier for the animation instance.

```
color(uid:myColor, vals:hue, dur:5)
```

trig:

Trigger mode.

Value	Behaviour
auto	Start immediately on load (default)
playhead	Start when playhead reaches element

```
color(vals:[#f00, #00f], dur:2, trig:playhead)
```

tdelay:

Delay before animation starts (seconds).

```
color(vals:hue, dur:6, tdelay:2)    // wait 2 seconds then start
```

prestate:

Initial visibility state before animation starts.

Value	Effect
show	Visible immediately (default)
hide	Hidden until triggered
ghost	Semi-transparent (30% opacity)
fadein(ms)	Fade in over specified milliseconds
ghostClickable(ms)	Ghost until clicked

```
color(vals:hue, dur:5, prestate:hide, trig:playhead)
color(vals:[#f00, #0f0], dur:2, prestate:fadein(1000))
```

target:

Specifies which SVG property to animate.

Value	Effect
both	Animate fill and stroke (default)
fill	Animate fill only
stroke	Animate stroke only

```
// Animate fill only
color(vals:[#f00, #00f], dur:2, target:fill)

// Animate stroke only (useful for outlines)
color(vals:hue, dur:10, target:stroke)

// Animate both (default)
color(vals:[#f00, #00f], dur:2, target:both)
```

osc:

Enable OSC output.

Value	Behaviour
0	Disabled (default)
1	Continuous output

```
color(vals:hue, dur:10, osc:1)
```

OSC output includes: - h Hue (0360) - s Saturation (0100) - l Lightness (0100) - hNorm Normalised hue (01)
- sNorm Normalised saturation (01) - lNorm Normalised lightness (01)

oscaddr:

Custom OSC address.

```
color(vals:hue, dur:8, osc:1, oscaddr:/visuals/bg/hue)
```

Colour Interpolation

All interpolation occurs in **HSL colour space**:

- **Hue**: Circular interpolation (shortest angular path)
- **Saturation**: Linear interpolation
- **Lightness**: Linear interpolation

This means transitions between colours feel natural and avoid muddy intermediate tones.

Shortest Path Hue

When transitioning between hues, Oscilla takes the shortest path around the colour wheel:

```
#f00 → #00f    // Red to Blue: goes via magenta (shorter)
                // NOT via yellow/green (longer)
```

Patterns**Pseq Sequential**

Plays values in order, repeating a specified number of times.

```
color(vals:Pseq([#f00, #ff0, #0f0], 3), dur:1)
// Plays: red, yellow, green, red, yellow, green, red, yellow, green
```

Prand Random

Selects randomly, may repeat the same value consecutively.

```
color(vals:Prand([#f00, #0f0, #00f], inf), dur:0.5)
```

Pxrand Exclusive Random

Selects randomly but never repeats the same value twice in a row.

```
color(vals:Pxrand([#f00, #0f0, #00f], inf), dur:0.5)
```

Pshuf Shuffle

Shuffles the values, plays through, then reshuffles.

```
color(vals:Pshuf([#f00, #ff0, #0f0, #0ff], 2), dur:0.8)
```

Examples**Discrete Colour Melody**

```
color(vals:[#f00, #ff0, #0ff, #f0f], dur:1.5)
```

Warm to Cool Transition

```
color(vals:[#f80, #fa0, #ff0, #8f0, #0f8, #0ff], dur:6)
```

Breathing Glow

```
color(vals:[hsl(200,80%,30%), hsl(200,80%,60%)], mode:alt, dur:2, loop:0)
```

Continuous Spectrum

```
colour(vals:hue, dur:20, loop:0)
```

Limited Hue Range

```
color(vals:hue(0, 60), dur:4, loop:0)    // Red to yellow only
color(vals:hue(180, 300), dur:5)        // Cyan to magenta
```

Rhythmic Colour Pulses

```
color(vals:[#fff, #000], mode:step, dur:0.25, loop:0)
```

Pattern with Variable Timing

```
color(
  vals:Pseq([#f00, #0f0, #00f], inf),
  dur:Pseq([1, 0.5, 2], inf)
)
```

Playhead-Triggered Fade

```
color(
  vals:[#333, #f80],
  dur:3,
  trig:playhead,
  prestate:fadein(500)
)
```

With OSC Output

```
colour(
  vals:hue,
  dur:15,
  loop:0,
  osc:1,
  oscaddr:/stage/wash/hue
)
```

Named Colours Reference

Oscilla recognises these named colours:

Name	Hue
red	0
orange	30
yellow	60

Name	Hue
green	120
cyan	180
blue	240
purple	270
magenta	300
pink	330
white	
black	
gray / grey	

For precise control, use hex, RGB, or HSL notation.

Tips

1. **Use HSL for control:** `hsl(h, s%, l%)` gives direct control over hue, saturation, and lightness.
2. **Hue cycling for ambience:** `vals:hue` creates slow, evolving colour fields without discrete steps.
3. **Step mode for rhythm:** `mode:step` creates hard cuts useful for rhythmic visual accents.
4. **Alt mode for breathing:** `mode:alt` with two colours creates organic pulsing effects.
5. **Combine with other cues:** Colour animations work alongside scale, rotate, and o2p animations on the same element.

See Also

- `rotate()` Rotation animation
- `scale()` Scale animation
- `o2p()` Object-to-path animation
- [Patterns](#) Pattern system reference

cue:fade() fade or pulse an SVG element

Fades the opacity of the triggering SVG object (or a specified target) between two opacity values. Can be used for smooth fade-ins, fade-outs, or pulsing effects. The transition speed, easing, and timing can all be controlled.

Syntax

```
cue:fade(mode:<in|out|pulse|blink>, dur:<seconds>,
         from:<0-1>, to:<0-1>, ease:<easing>,
         time:<seconds>, delay:<seconds>, hold:<seconds>,
         target:<id>)
```

Arguments

Argument	Description
mode	type of fade: "in", "out", "pulse", or "blink"
dur	duration of the fade in seconds
from	starting opacity (0 = transparent, 1 = opaque)
to	ending opacity
ease	easing type (e.g. <code>linear</code> , <code>easeInOutSine</code> , etc.)
time	start time offset in seconds (optional, for cue scheduling)
delay	delay before fade starts (seconds)
hold	time to hold at final opacity before reset or removal

Argument	Description
target	optional target element ID; if omitted, applies to the cue element itself

Behavior

- `cue:fade(mode:in)` smoothly fades in from `from` to `to` opacity.
- `cue:fade(mode:out)` fades out in reverse.
- `cue:fade(mode:pulse)` fades in and out continuously while active.
- `cue:fade(mode:blink)` alternates visibility quickly (use short `dur`).
- The cue supports timing offsets and can be combined with OSC or other cue types.
- When `target` is not provided, the fade applies to the object containing the cue.
- The `hold` argument determines how long the element stays visible before reverting or fading away.

Examples

```
cue:fade(mode:in, dur:2, from:0, to:1)
cue:fade(mode:out, dur:3, ease:easeOutSine)
cue:fade(mode:pulse, dur:1.5, from:0.2, to:1)
cue:fade(mode:blink, dur:0.5, time:10)
cue:fade(mode:in, dur:2, from:0, to:1, target:circle1)
cue:fade(mode:pulse, dur:2, hold:5, target:obj_light)
```

ui()

`ui()` applies **performer-facing UI changes** (HTML/CSS) during playback most commonly to **hide or reveal interface elements** such as the playhead, play zone, overlays, or control panels.

It targets **HTML elements (not SVG)** using a CSS selector and applies a controlled set of visual changes, optionally animated over time.

The UI cue system was **primarily designed to manage visibility of the playhead and play zone**, allowing scores to shift between guided, open, or reduced-UI performance states.

Syntax

```
ui("<css selector>", opacity:<number>, scale:<number>, x:<px>, y:<px>, rotate:<deg>, visible:<bool>, color:<css>,
↳ bg:<css>, z:<number>, dur:<seconds>)
```

What it affects

- Works on: HTML UI elements
- Does not affect: SVG score elements
- Multiple matches: All matching elements are updated

Built-in UI toggle button

Oscilla includes a **GUI button in the bottom-right corner** that toggles:

- Playhead
- Play zone

on and off.

This button uses the same internal UI logic as the `ui()` cue and exists to allow fast switching between visible guidance and reduced interface modes during performance.

Parameters

Transition

- `dur` (seconds, default 0)

Visibility

- `visible:true|false`

Opacity

- `opacity` (number)

Transform

- `x, y` (px)
- `scale` (number)
- `rotate` (degrees)

Transforms are stateful and combined.

Color

- `color`
- `bg`

Layering

- `z` (z-index)
-

Examples

```
ui("#playhead", opacity:0, dur:0.3)
ui("#playhead", visible:false, dur:0.3)
ui("#playhead", #playzone, visible:false, dur:0.25)
ui("#playhead", #playzone, visible:true, dur:0.25)
```

Design notes

- Intended for performance-state control
- Touch- and tablet-friendly
- Reduces visual clutter
- Suitable for audience-facing projections

Chapter 5

Control and Interaction

The control layer allows performers to influence score behaviour in real time. While cues define what happens when the playhead crosses an element, the control system determines how those behaviours can be shaped during performance through live input.

At the centre of this system is a source-agnostic parameter bus. Control signals can originate from the controlXY touchpad interface, from external OSC sources, or from other cues within the score. These signals are routed to cue parameters through a binding system that allows any control source to modulate any parameter animation speed, colour, audio volume, OSC output values without the cue needing to know where the signal comes from.

This architecture supports a range of performance practices, from a single performer adjusting their own score to a conductor modulating parameters across the entire ensemble via a networked control surface.

Oscilla Control Plane Architecture & Usage Guide

Published signals are shared across the system, so animations can control audio parameters, slider-like interfaces can control animations, and multiple cues can influence each other in any combination. Because values are updated continuously, this also allows feedback systems where motion, sound, and interaction form coupled, dynamic behaviours within the score.

```
synth(uid:pad, freq:"fadeSineFreq.t[90,2000]", env:{a:4})
```

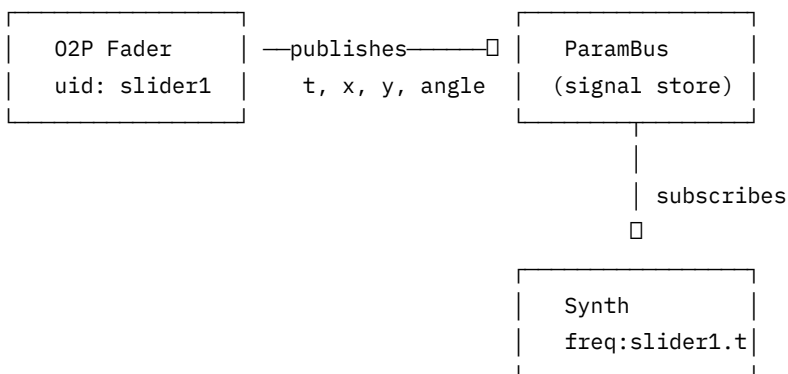
```
o2p(path:fadeSineFreq, trig:touch, osc:1, uid:fadeSineFreq, oscAddr:fadeSineFreq)
```

Overview

The Oscilla Control Plane enables **bidirectional signal flow** between cues. Any animation can publish its values as signals, and any synth or audio cue can subscribe to those signals to control its parameters in real-time.

This transforms Oscilla from a trigger-based score system into a **dynamic, signal-driven, executable score environment**.

Core Concept



Quick Start

1. Create a Controller (O2P Fader)

```
o2p(path:faderTrack, trig:touch, uid:myFader)
```

This fader automatically publishes: - `o2p:myFader.t` position along path (0-1) - `o2p:myFader.x` normalized X position (0-1) - `o2p:myFader.y` normalized Y position (0-1) - `o2p:myFader.angle` tangent angle in degrees

2. Bind a Synth Parameter

```
synth(uid:pad, freq:myFader.t[200,800], amp:0.2)
```

The `freq:myFader.t[200,800]` syntax means: - Subscribe to signal `o2p:myFader.t` - Map the 0-1 value to 200-800 Hz

3. Result

Moving the fader changes the synths frequency in real-time!

Signal Reference Syntax

Basic Binding

```
param:source.channel
```

- **param** The parameter to control (freq, amp, pan, etc.)
- **source** The uid of the publishing cue
- **channel** Which signal to subscribe to (t, x, y, angle, etc.)

Binding with Range

```
param:source.channel[min,max]
```

Maps the signal (assumed 0-1) to the specified output range.

Examples

DSL	Meaning
<code>freq:slider.t</code>	Freq follows slider position (0-1)
<code>freq:slider.t[200,2000]</code>	Freq mapped to 200-2000 Hz
<code>amp:fader.y[0,0.5]</code>	Amplitude mapped to 0-0.5
<code>pan:knob.x[-1,1]</code>	Pan mapped to full left-right
<code>cutoff:wheel.t[200,8000]</code>	Filter cutoff mapped to range

Published Signals by Cue Type

O2P Animation

Signal	Description	Range
<code>o2p:{uid}.t</code>	Position along path	0-1
<code>o2p:{uid}.x</code>	Normalized X in frame	0-1
<code>o2p:{uid}.y</code>	Normalized Y in frame	0-1
<code>o2p:{uid}.angle</code>	Tangent angle	degrees

Rotate Animation

Signal	Description	Range
<code>rotate:{uid}.angle</code>	Current angle	0-360
<code>rotate:{uid}.rad</code>	Angle in radians	0-2
<code>rotate:{uid}.norm</code>	Normalized angle	0-1

Scale Animation

Signal	Description	Range
scale:{uid}.sx	Scale X factor	varies
scale:{uid}.sy	Scale Y factor	varies
scale:{uid}.uniform	Average scale	varies

Bindable Parameters

Synth

Parameter	Default Range	Description
freq / frequency	20-20000	Oscillator frequency (Hz)
amp / amplitude	0-1	Output amplitude
pan	-1 to 1	Stereo position
cutoff / filterFreq	20-20000	Filter cutoff (Hz)
q / resonance	0.1-30	Filter resonance
detune	-1200 to 1200	Detune in cents

Audio / AudioPool / AudioImpulse

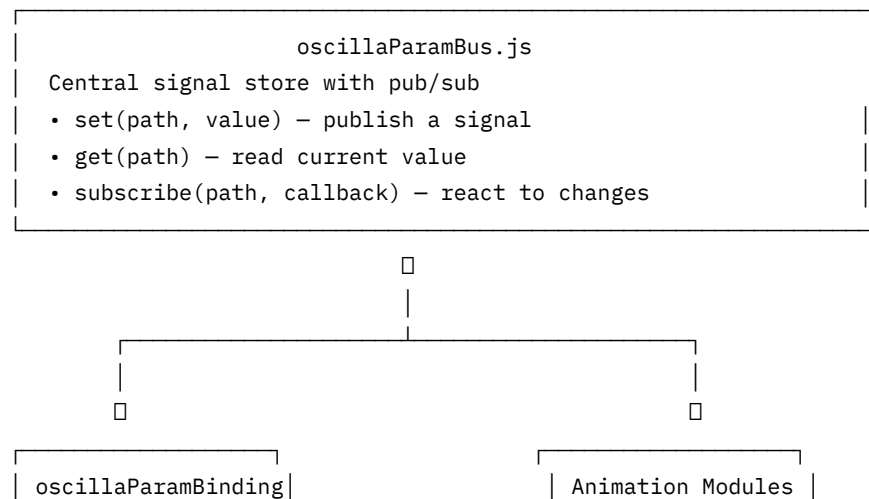
Parameter	Default Range	Description
amp	0-1	Playback amplitude
pan	-1 to 1	Stereo position
pitch / rate	0.25-4	Playback rate

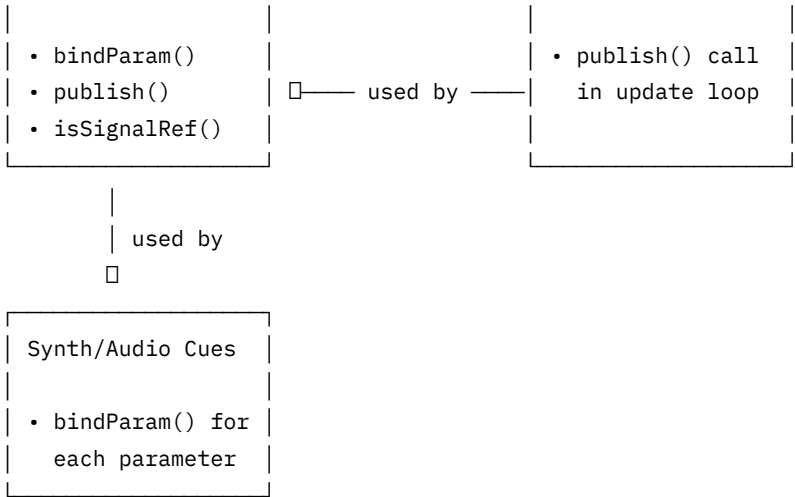
Effects (within synth)

Parameter	Default Range	Description
delayTime	0-2	Delay time (seconds)
feedback / fb	0-0.99	Delay feedback
mix / wet	0-1	Effect mix level

Architecture

Module Overview





File Descriptions

File	Purpose
oscillaParamBus.js	Central signal store with pub/sub
oscillaParamBinding.js	bindParam() and publish() helpers
oscillaParserSignalRef.js	Parser extension for signal ref syntax

Integration Guide

Adding Signal Publishing to a Cue

To make any animation publish signals, add **one line** to its update callback:

```
import { publish } from './oscillaParamBinding.js';

// In your animation's update callback:
update: () => {
  // ... existing animation logic ...

  // ADD THIS LINE:
  publish("o2p", cfg.uid, { t: pathT, x: normX, y: normY, angle });
}
```

Example: O2P Animation

Location: oscillaAnimationO2p.js, in startContinuousO2P(), around line 492

```
// BEFORE (existing code):
emitO2POsc({ cfg, uid: cfg.uid, path, point, pathT });

// AFTER (add this line):
import { publish } from './oscillaParamBinding.js';
// ... then in update callback:
emitO2POsc({ cfg, uid: cfg.uid, path, point, pathT });
publish("o2p", cfg.uid, { t: globalT, x: normX, y: normY, angle });
```

Note: Youll need to calculate normX and normY from the path bbox:

```
const bbox = path.getBBox();
const normX = (point.x - bbox.x) / bbox.width;
const normY = (point.y - bbox.y) / bbox.height;
```

Example: Rotate Animation

Location: oscillaAnimationRotate.js, in handleRotateContinuous(), around line 548

```
import { publish } from './oscillaParamBinding.js';

update: () => {
  // ... existing code ...
  const angle = getCurrentAngle(animEl, 0);

  // ADD THIS:
```

```
    publish("rotate", cfg.uid, { angle: angle });
  }
}
```

Example: Scale Animation

Location: `oscillaAnimationScale.js`, in `handleScaleContinuous()`, around line 586

```
import { publish } from './oscillaParamBinding.js';

update: () => {
  // ... existing code ...
  const sx = parseFloat(tr.match(/scale\(((^,)+),/)?.[1] || 1);
  const sy = parseFloat(tr.match(/,\s*(^)+\)/)?.[1] || sx);

  // ADD THIS:
  publish("scale", cfg.uid, { sx, sy, uniform: (sx + sy) / 2 });
}
```

Adding Signal Binding to a Cue

To make parameters bindable in a cue, use `bindParam()`:

```
import { bindParam, isSignalRef } from './oscillaParamBinding.js';

function startMyCue(params) {
  const unbinders = [];

  // Bind frequency - works with both static and signal ref
  const freqBinding = bindParam(
    params.freq,
    (hz) => oscillator.frequency.setTargetAtTime(hz, 0, 0.02),
    { min: 20, max: 20000, default: 440 }
  );
  unbinders.push(freqBinding.unbind);

  // Use initial value
  oscillator.frequency.value = freqBinding.value;

  // ... later, on cleanup:
  unbinders.forEach(fn => fn());
}
```

Example: Synth Integration

Location: `oscillaSynth.js`, in `startSynthVoice()`, around line 673

```
import { bindParam } from './oscillaParamBinding.js';

function startSynthVoice(uid, ast, cueElement, opts) {
  const ctx = sharedAudioCtx;
  const params = extractParams(ast);
  const unbinders = [];

  // Frequency binding
  const freqBinding = bindParam(
    params.freq,
    (hz) => {
      if (voice.source?.kind === 'osc') {
        voice.source.node.frequency.setTargetAtTime(hz, ctx.currentTime, 0.02);
      }
    },
    { min: 20, max: 20000, default: 440 }
  );
  unbinders.push(freqBinding.unbind);

  // Amplitude binding
  const ampBinding = bindParam(
    params.amp,
    (amp) => {
      voice.graph.gain.gain.setTargetAtTime(amp, ctx.currentTime, 0.02);
    },
    { min: 0, max: 0.5, default: 0.1 }
  );
}
```

```

unbinders.push(ampBinding.unbind);

// ... create voice with initial values ...
const freqHz = freqBinding.value;
const amp = ampBinding.value;

// Store unbinders on voice for cleanup
voice._unbinders = unbinders;
}

function cleanupVoice(uid, voice) {
  // Unbind all signal subscriptions
  voice._unbinders?.forEach(fn => fn());
  // ... rest of cleanup ...
}

```

Parser Integration

Modifying the Parser

The parser needs to recognize signal references in parameter values. Add this to the value extraction logic:

Location: `oscillaParser.js`, in `cstToAst()` `synth/audio` sections

```

import { maybeConvertToSignalRef } from './oscillaParserSignalRef.js';

// When extracting a parameter value:
let val = extractRawValue(valueNode);
val = maybeConvertToSignalRef(val, paramName);

```

What the Parser Outputs

Input DSL:

```
synth(uid:pad, freq:fader1.t[200,800], amp:0.2)
```

Output AST:

```

{
  type: "cueSynth",
  args: [
    { type: "uid", value: "pad" },
    {
      type: "freq",
      value: {
        type: "signalRef",
        source: "fader1",
        channel: "t",
        range: [200, 800]
      }
    },
    { type: "amp", value: 0.2 }
  ]
}

```

Debugging

Enable Debug Mode

```

// In browser console:
oscillaParamBus.setDebugMode(true);

```

This logs all signal changes.

Inspect Current Signals

```

// List all signals:
oscillaParamBus.list();

// List signals from a specific source:
oscillaParamBus.list("o2p:");

// Get current value:
oscillaParamBus.get("o2p:fader1.t");

```

```
// Get snapshot of all values:
oscillaParamBus.snapshot();
Test Signal Publishing
// Manually publish a signal:
oscillaParamBus.set("o2p:test.t", 0.5);

// Watch a signal:
const unsub = oscillaParamBus.subscribe("o2p:test.t", (value, path) => {
  console.log(`${path} = ${value}`);
});

// Later: unsub() to stop watching
```

Common Patterns

XY Pad Synth

```
o2p(path:xyPad, trig:touch, uid:xy)
synth(uid:pad, freq:xy.x[200,2000], amp:xy.y[0,0.5])
```

Rotation Filter

```
rotate(dur:4, loop:0, uid:wheel)
synth(uid:drone, freq:220, cutoff:wheel.norm[200,4000])
```

Multiple Controllers One Synth

```
o2p(path:freqSlider, trig:touch, uid:fSlider)
o2p(path:ampSlider, trig:touch, uid:aSlider)
synth(uid:lead, freq:fSlider.t[100,1000], amp:aSlider.t[0,0.3])
```

One Controller Multiple Synths

```
o2p(path:masterFader, trig:touch, uid:master)
synth(uid:bass, freq:master.t[50,200], amp:0.2)
synth(uid:mid, freq:master.t[200,800], amp:0.15)
synth(uid:high, freq:master.t[800,4000], amp:0.1)
```

Limitations & Notes

1. **Signal Publishing Rate:** Signals are published at ~60fps to avoid overwhelming the system.
 2. **Binding Lifecycle:** Bindings are automatically cleaned up when the cue stops (if you call the unbind functions).
 3. **No Circular Dependencies:** The system doesn't prevent circular signal routing. Avoid creating feed-back loops unless intentional.
 4. **Static Parameters:** Some parameters can't be changed after cue start (e.g., wave type, audio src). These will use the initial value even if bound.
 5. **Signal Range:** All signals are assumed to be in the 0-1 range. Use the `[min,max]` syntax to map to your desired output range.
-

Summary Checklist

To Make an Animation Publish Signals:

1. Import `publish` from `oscillaParamBinding.js`
2. Add `publish("type", uid, { channel: value })` to update callback
3. Done!

To Make a Cue Accept Signal Bindings:

1. Import `bindParam` from `oscillaParamBinding.js`
2. Wrap parameter initialization with `bindParam()`
3. Store unbind functions for cleanup

4. Call unbind functions when cue stops
 5. Update parser to recognize signal refs (one-time)
-

Version History

- **v1.0** Initial control plane implementation
 - ParamBus signal store
 - bindParam/publish helpers
 - Signal reference syntax support

controlXY Multitouch XY Control & Score Animation System

controlXY transforms your score into an **interactive animation canvas**. It defines persistent, multi-touch XY control surfaces that enable composers to create **complex choreographed animations** directly within the score the **score-as-instrument, instrument-as-score**.

Unlike traditional time-based cues, controlXY provides **continuous spatial control** that can be: - **Pre-programmed** as animated sequences (choreography) - **Performed live** as a tactile control surface - **Hybrid** switching between automation and manual control

As a bonus, all movements can simultaneously **transmit OSC** for external synthesis and media control.

Oscilla OSC controller design in Inkscape

Your browser does not support the video tag.

The controlXY multitouch OSC controller created in Inkscape. Each control object can be dragged freely across the canvas, sending X, Y, and rotation (twist) values via OSC. The orange rings indicate touch mode (trig:touch) is active. Multiple simultaneous touches are supported, with each object independently sending its position and rotation state. This example shows the page view with preset management and parameter display.

Core Concept: Score Animation Through Spatial Control

Think of controlXY as a **spatial modulation source** embedded in your score:

1. **Define control pads** with draggable handles
2. **Bind handle positions** to visual/sonic parameters
3. **Animate via presets & sequences** OR **perform live**
4. **Record/playback** spatial gestures as part of the composition

This inverts the traditional “playhead reads static notation” model instead, **notation becomes dynamic**, responding to spatial control in real-time.

Syntax

```
controlXY(
  uid: <string>,
  touchpt: <element-id> | [<id1>, <id2>, ...],
  handle: <element-id> | [<id1>, <id2>, ...], // optional: rotation handles
  hmode: continuous | limited | [mode1, mode2, ...], // optional: rotation behavior
  rotRange: <degrees>, // optional: rotation range for limited mode
  bounds: <element-id> | "self",
  label: <bool>,
  osc: <bool|number>,
  oscAddr: <string>
)
```

Note: For backwards compatibility, if only `handle:` is specified (without `touchpt:`), it refers to XY control elements with no rotation.

Parameters

Parameter	Type	Required	Default	Description
uid	string	yes		Unique identifier for the control pad. Used for publishing and parameter binding.
touchpt	element id or array	yes*		ID(s) of SVG element(s) for XY position control. Use array syntax for multitouch: [dot1, dot2, dot3]
handle	element id or array	no		ID(s) of SVG element(s) for rotation control. Can be single shared handle or parallel array. Omit for XY-only control.
hmode	continuous limited array	no	continuous	Rotation behavior: continuous wraps at 360, limited stops at min/max like a potentiometer. Use array for per-handle modes.
rotRange	number	no	270 (limited)360 (continuous)	Rotation range in degrees for limited mode. Default 270 provides 7 oclock to 4 oclock range.
bounds	element id or "self"	no	self	ID of the SVG element that defines the XY constraint area. Defaults to the element the DSL is attached to.
label	boolean	no	false	Show live value labels above each handle. Useful for debugging and performance.
osc	true false number	no	false	Enable OSC output. If a number is given, it specifies throttle interval in ms. Default throttle 30 ms.
oscAddr	string	no	controlXY/<uid>	Custom OSC address (without leading /).

*For backwards compatibility, `handle:` can be used instead of `touchpt:` when not using rotation.

Coordinate System

- **X axis**

- Left = 0.0
- Right = 1.0
- **Y axis** (musical convention)
 - Bottom = 0.0
 - Top = 1.0

All values are **normalized** to the bounding box, making animations resolution-independent.

Rotation Control (Optional)

In addition to XY positioning, handles can have **rotation control** - either as a shared twist handle or individual knobs per touchpoint.

Rotation Modes: hmode Parameter

continuous (default) - Full 360 rotation with wraparound - Rotating past 360 wraps to 0 - Natural for: azimuth, phase, circular parameters - Provides endless rotation without hard stops

limited - Hard stops at minimum and maximum - like a real potentiometer - Default range: 270 (7 oclock to 4 oclock) - **Cannot wrap** from maximum to minimum or vice versa - Safe for: volume, gain, pan - prevents accidental jumps - Custom range via rotRange: parameter (e.g., rotRange:180 for half rotation)

Rotation Coordinate System

- **0** = 7 oclock position (minimum)
- **90** = 10 oclock position
- **180** = 1 oclock position
- **270** = 4 oclock position (maximum for default limited mode)
- Rotation proceeds **clockwise**

Example Behaviors

Continuous mode (default):

Rotate clockwise: 350° → 360° → 0° → 10° (wraps smoothly)
Perfect for spatial direction, phase rotation, etc.

Limited mode:

At 0°: Try rotating counterclockwise → STOPS (hard stop at minimum)
At 270°: Try rotating clockwise → STOPS (hard stop at maximum)
Perfect for volume faders, mix levels, etc.

Published Signals

XY Only (No Rotation)

```
controlXY:<uid>.x      // 0.0 - 1.0
controlXY:<uid>.y      // 0.0 - 1.0
controlXY:<uid>.handle // handle element id
```

XY + Rotation

```
controlXY:<uid>.x      // 0.0 - 1.0
controlXY:<uid>.y      // 0.0 - 1.0
controlXY:<uid>.p      // 0.0 - 1.0 (rotation, normalized)
controlXY:<uid>.handle // handle element id
```

Multiple Handles

```
controlXY:<uid>.<handleId>.x // 0.0 - 1.0
controlXY:<uid>.<handleId>.y // 0.0 - 1.0
controlXY:<uid>.<handleId>.p // 0.0 - 1.0 (rotation, if handle has rotation)
```

Rotation normalization: - **Continuous mode:** $p = \text{angle} / 360$ (full circle = 0.0 to 1.0) - **Limited mode:**

$p = \text{angle} / \text{rotRange}$ (range = 0.0 to 1.0)

These signals can be bound to **any parameter** in the system, creating direct connections between spatial control and visual/sonic outcomes.

Setup Examples

Basic Single Handle (XY Only)

```
<rect id="pad1" x="100" y="100" width="400" height="300"
  fill="#222" stroke="#666"
  cue="controlXY(uid:pad1, touchpt:dot1)"/>
```

```
<circle id="dot1" cx="0" cy="0" r="12" fill="#ff4444"/>
```

XY + Shared Rotation Handle

```
<rect id="synthPad" x="100" y="100" width="400" height="300"
  fill="#222" stroke="#666"
  cue="controlXY(uid:synth, touchpt:[dot1, dot2], handle:knob, label:true)"/>
```

```
<circle id="dot1" cx="0" cy="0" r="12" fill="#ff4444"/>
```

```
<circle id="dot2" cx="0" cy="0" r="12" fill="#44ff44"/>
```

```
<circle id="knob" cx="0" cy="20" r="6" fill="#ffaa00"/>
```

One rotation handle shared by both touchpoints.

Volume Mixer with Hard Stops

```
<rect id="mixer" x="100" y="100" width="600" height="400"
  fill="#111"
  cue="controlXY(uid:mixer,
    touchpt:[vol1, vol2, vol3, vol4],
    handle:fader,
    hmode:limited,
    label:true)"/>
```

```
<circle id="vol1" cx="0" cy="0" r="10" fill="#ff4444"/>
```

```
<circle id="vol2" cx="0" cy="0" r="10" fill="#44ff44"/>
```

```
<circle id="vol3" cx="0" cy="0" r="10" fill="#4444ff"/>
```

```
<circle id="vol4" cx="0" cy="0" r="10" fill="#ffff44"/>
```

```
<circle id="fader" cx="0" cy="15" r="5" fill="#fff"/>
```

Rotation stops at min/max - safe for volume control, no wraparound.

Mixed Rotation Modes (Array Syntax)

```
<rect id="hybrid" x="100" y="100" width="600" height="300"
  cue="controlXY(uid:hybrid,
    touchpt:[vol, phase, pan],
    handle:knob,
    hmode:[limited, continuous, limited],
    label:true)"/>
```

Volume and pan use limited (hard stops), phase uses continuous (wraps).

Multi-Handle (Multitouch, XY Only)

```
<rect id="mixer" x="100" y="100" width="600" height="400"
  fill="#111"
  cue="controlXY(uid:mixer, touchpt:[fader1,fader2,fader3,fader4], label:true)"/>
```

```
<circle id="fader1" cx="0" cy="0" r="10" fill="#ff4444"/>
<circle id="fader2" cx="0" cy="0" r="10" fill="#44ff44"/>
<circle id="fader3" cx="0" cy="0" r="10" fill="#4444ff"/>
<circle id="fader4" cx="0" cy="0" r="10" fill="#ffff44"/>
```

With OSC Output (XY + Rotation)

```
<rect id="synthControl" x="50" y="50" width="300" height="300"
  cue="controlXY(uid:synth, touchpt:dot1, handle:knob, osc:true, oscAddr:synth/xy)"/>
```

Sends X, Y, and rotation values via OSC.

Animation Workflow: Creating Score Choreography

The true power of `controlXY` emerges when you **pre-program spatial animations** as part of your composition.

Step 1: Save Spatial States as Presets

Using the Preset UI: 1. Press **Alt+Shift+P** to open the preset manager (or click on the pad) 2. Move handles to desired positions 3. Type a preset name (e.g., “intro_position”) 4. Click Save

Via DSL (triggered by playhead or buttons):

```
<!-- Save current state when playhead passes -->
<rect x="100" y="0" width="2" height="600"
  cue="ui(action:'controlXYSave', preset:'stateA')"
  fill="red" opacity="0.3"/>

<!-- Button to save state -->
<g cue="button(
  trigger:ui(action:'controlXYSave', preset:'verse1'),
  style(label:'Save Verse', x:10, y:10)
)"/>
```

Via Console (for experimentation):

```
// Save all controlXY instances
window.controlXYPresets.save('stateA');

// Save specific pad only
window.controlXYPresets.save('stateB', 'pad1');
```

Step 2: Recall States with Animation

Instant recall (no tween):

```
<rect x="500" y="0" width="2" height="600"
  cue="ui(action:'controlXYRecall', preset:'stateA')"
  fill="blue" opacity="0.3"/>
```

Smooth tween (2 seconds, easing):

```
<rect x="1000" y="0" width="2" height="600"
  cue="ui(action:'controlXYRecall', preset:'stateB', dur:2, ease:'easeInOutSine')"
  fill="green" opacity="0.3"/>
```

Via button:

```
<g cue="button(
  trigger:ui(action:'controlXYRecall', preset:'chorus', dur:3, ease:'easeOutElastic'),
  style(label:'□ Chorus', x:10, y:50)
)"/>
```

Step 3: Define Sequences (Choreographed Animation)

Sequences are **playlists of presets** that play automatically.

Define sequence via DSL:

```
<!-- Define a 3-state sequence -->
<rect x="100" y="0" width="2" height="600"
  cue="ui(action:'controlXYDefineSequence',
    name:'intro_dance',
    steps:'stateA,stateB,stateC')"
  fill="yellow" opacity="0.3"/>
```

Play the sequence:

```
<!-- Auto-play when playhead reaches this point -->
<rect x="200" y="0" width="2" height="600"
  cue="ui(action:'controlXYSequence',
```

```

        seq:'intro_dance',
        dur:2,
        ease:'easeInOutSine',
        loop:false)"
    fill="cyan" opacity="0.3"/>

```

Stop sequence:

```

<rect x="1500" y="0" width="2" height="600"
    cue="ui(action:'controlXYSequenceStop')"
    fill="red" opacity="0.5"/>

```

Via buttons (interactive control):

```

<!-- Define sequence button -->
<g cue="button(
    trigger:ui(action:'controlXYDefineSequence',
        name:'verse_pattern',
        steps:'v1,v2,v3,v1'),
    style(label:'Define Pattern', x:10, y:10)
)"/>

<!-- Play button -->
<g cue="button(
    trigger:ui(action:'controlXYSequence',
        seq:'verse_pattern',
        dur:1.5,
        loop:true),
    style(label:'▶ Play', x:10, y:50)
)"/>

<!-- Stop button -->
<g cue="button(
    trigger:ui(action:'controlXYSequenceStop'),
    style(label:'◻ Stop', x:10, y:90)
)"/>

```

Rotation Control Use Cases

When to Use `hmode:limited` (Hard Stops)

Perfect for: - **Volume/Gain controls** - prevents accidental jump from loud to silent - **Mix levels** - fader-style controls - **Pan controls** - left-center-right with hard stops - **Envelope parameters** - attack, decay, sustain, release - **Filter cutoff** - when you need precise min/max boundaries

Example:

```

<rect cue="controlXY(uid:mixer,
    touchpt:[ch1, ch2, ch3],
    handle:volKnob,
    hmode:limited,
    rotRange:270)"/>

```

Provides 270 of rotation (7 oclock to 4 oclock) with hard stops at both ends.

When to Use `hmode:continuous` (Wrapping)

Perfect for: - **Spatial positioning** - azimuth, direction, angle - **Phase controls** - where 0 = 360 naturally - **LFO rate/frequency** - continuous spinning - **Cyclical parameters** - anything that wraps by nature - **Visual rotation** - spinning objects, wheels, orbits

Example:

```

<rect cue="controlXY(uid:spatial,
    touchpt:[src1, src2],
    handle:direction,
    hmode:continuous)"/>

```

Full 360 rotation with smooth wraparound at the 0/360 boundary.

Mixed Modes for Hybrid Control

Different parameters on the same pad can have different rotation behaviors:

```

<rect cue="controlXY(uid:synth,
    touchpt:[cutoff, resonance, phase, gain],
    handle:knob,
    hmode:[limited, limited, continuous, limited])"/>

```

- **cutoff, resonance, gain:** Hard stops (safe for audio levels)
- **phase:** Wraps naturally (0 = 360)

Complete Animation Example: Verse-Chorus-Bridge

Heres a **full composition workflow** showing how to choreograph spatial animation:

```
<!-- ===== SETUP: Define the control pad ===== -->
<rect id="controlPad" x="100" y="100" width="600" height="400"
  fill="#111" stroke="#333"
  cue="controlXY(uid:mixer, handle:[dot1,dot2,dot3], label:true)"/>

<circle id="dot1" cx="0" cy="0" r="12" fill="#ff4444"/>
<circle id="dot2" cx="0" cy="0" r="12" fill="#44ff44"/>
<circle id="dot3" cx="0" cy="0" r="12" fill="#4444ff"/>

<!-- ===== PLAYHEAD TRIGGERS: Auto-save states ===== -->
<!-- Measure 1: Save intro position -->
<rect x="100" y="0" width="1" height="600"
  cue="ui(action:'controlXYSave', preset:'intro')"
  fill="transparent"/>

<!-- Measure 5: Save verse position -->
<rect x="500" y="0" width="1" height="600"
  cue="ui(action:'controlXYSave', preset:'verse')"
  fill="transparent"/>

<!-- Measure 13: Save chorus position -->
<rect x="1300" y="0" width="1" height="600"
  cue="ui(action:'controlXYSave', preset:'chorus')"
  fill="transparent"/>

<!-- Measure 21: Save bridge position -->
<rect x="2100" y="0" width="1" height="600"
  cue="ui(action:'controlXYSave', preset:'bridge')"
  fill="transparent"/>

<!-- ===== PLAYHEAD AUTOMATION: Recall with tweens ===== -->
<!-- M9: Tween to verse over 2 seconds -->
<rect x="900" y="0" width="2" height="600"
  cue="ui(action:'controlXYRecall', preset:'verse', dur:2, ease:'easeInOutSine')"
  fill="blue" opacity="0.4"/>

<!-- M17: Quick snap to chorus -->
<rect x="1700" y="0" width="2" height="600"
  cue="ui(action:'controlXYRecall', preset:'chorus', dur:0.5, ease:'easeOutQuad')"
  fill="green" opacity="0.4"/>

<!-- M25: Elastic bounce to bridge -->
<rect x="2500" y="0" width="2" height="600"
  cue="ui(action:'controlXYRecall', preset:'bridge', dur:3, ease:'easeOutElastic')"
  fill="purple" opacity="0.4"/>

<!-- ===== SEQUENCES: Complex multi-step animations ===== -->
<!-- M29: Define and play verse pattern -->
<rect x="2900" y="0" width="1" height="600"
  cue="ui(action:'controlXYDefineSequence',
    name:'verse_pattern',
    steps:'verse,intro,verse')"
  fill="transparent"/>

<rect x="2950" y="0" width="2" height="600"
  cue="ui(action:'controlXYSequence',
    seq:'verse_pattern',
    dur:1,
    loop:false)"
```

```

    fill="yellow" opacity="0.4"/>

<!-- M40: Stop all automation, return to intro -->
<rect x="4000" y="0" width="2" height="600"
    cue="ui(action:'controlXYSequenceStop')"
    fill="red" opacity="0.5"/>

<rect x="4020" y="0" width="2" height="600"
    cue="ui(action:'controlXYRecall', preset:'intro', dur:4, ease:'easeInOutSine')"
    fill="cyan" opacity="0.4"/>

<!-- ===== MANUAL OVERRIDE: Buttons for live performance ===== -->
<!-- Performers can override automation at any time -->
<g cue="button(
    trigger:ui(action:'controlXYRecall', preset:'intro', dur:2),
    style(label:'□ Intro', x:10, y:500, width:80)
)"/>

<g cue="button(
    trigger:ui(action:'controlXYRecall', preset:'verse', dur:1.5),
    style(label:'V Verse', x:100, y:500, width:80)
)"/>

<g cue="button(
    trigger:ui(action:'controlXYRecall', preset:'chorus', dur:1),
    style(label:'C Chorus', x:190, y:500, width:80)
)"/>

<g cue="button(
    trigger:ui(action:'controlXYSequenceStop'),
    style(label:'□ Stop Auto', x:280, y:500, width:100)
)"/>

```

Binding to Score Elements: Creating Visual Animations

This is where **spatial control** becomes **visual transformation**.

Position Control

```

<!-- Dot X position controls rectangle X position -->
<rect id="box1"
    cue="scale(uid:box1, tx:mixer.dot1.x[-200,200])"/>

```

Size/Scale Control

```

<!-- Dot Y controls vertical scale -->
<rect id="box2"
    cue="scale(uid:box2, sy:mixer.dot1.y[0.5,2.0])"/>

```

Rotation Control

```

<!-- Map X position to rotation angle -->
<g id="spinner"
    cue="rotate(uid:spinner, values:mixer.dot1.x[0,360])"/>

```

Binding Handle Rotation (p)

```

<!-- Map the handle's twist (p) to element rotation -->
<g id="dial"
    cue="rotate(uid:dial, values:mixer.ch1.p[0,360])"/>

```

```

<!-- Use p for filter cutoff via OSC binding -->
<rect id="filterPad"
    cue="controlXY(uid:filter, touchpt:dot1, handle:knob, hmode:limited, osc:true)"/>

```

When handles have rotation, the *p* value (0.01.0) is saved in presets, tweened between presets, and published alongside *x* and *y*.

Color Control

```

<!-- Y position controls hue -->
<circle id="colorDot"
    cue="color(uid:colorDot, hue:mixer.dot1.y[0,360])"/>

```

Opacity Control

```

<!-- X position fades element -->
<rect id="fader"
    cue="fade(uid:fader, opacity:mixer.dot1.x)"/>

```

Multi-Parameter Complex Animation

```
<!-- Dot1 controls rotation, Dot2 controls scale, Dot3 controls opacity -->
<g id="complexShape">
  <rect cue="rotate(uid:r1, values:mixer.dot1.x[0,360])
    scale(uid:s1, sx:mixer.dot2.x[0.5,2], sy:mixer.dot2.y[0.5,2])
    fade(uid:f1, opacity:mixer.dot3.y)"/>
</g>
```

Spatial Navigation

```
<!-- Two handles define start/end points of a loop region -->
<g cue="o2p(path:loopA, start:mixer.dot1.x, end:mixer.dot2.x)"/>
```

Advanced: Programmatic Animation via Console

For complex choreography, you can script animations directly:

Tweening Without Presets

```
// Tween specific handles to target positions over 2 seconds
window.controlXYPresets.tweenTo({
  pad1: {
    dot1: { x: 0.5, y: 0.5 },
    dot2: { x: 0.2, y: 0.8 }
  }
}, { dur: 2, ease: 'easeInOutSine' });

// Include rotation (p) - tweens smoothly alongside x/y
window.controlXYPresets.tweenTo({
  mixer: {
    ch1: { x: 0.3, y: 0.7, p: 0.75 },
    ch2: { x: 0.8, y: 0.2, p: 0.25 }
  }
}, { dur: 3, ease: 'easeOutElastic' });
```

Complex Sequences with Per-Step Timing

```
// Define sequence with variable timing
window.controlXYPresets.defineSequence('complex_dance', [
  { preset: 'position1', dur: 2 },
  { preset: 'position2', dur: 1 },
  { preset: 'position3', dur: 3 },
  { preset: 'position1', dur: 2 }
]);

// Play with custom options
window.controlXYPresets.playSequence('complex_dance', {
  loop: true,
  onStep: (step, preset) => console.log(`Now: ${preset}`)
});
```

Per-Handle Timing Control

```
// Staggered animation - each handle moves independently
window.controlXYPresets.recall('myPreset', {
  handles: {
    dot1: { dur: 2, delay: 0, ease: 'easeInOutSine' },
    dot2: { dur: 1.5, delay: 0.5, ease: 'easeOutQuad' },
    dot3: { dur: 1, delay: 1, ease: 'easeOutElastic' }
  }
});
```

Easing Functions

Choose from 15 easing functions to shape your animations:

Number	Name	Character
0	linear	Constant speed
1	easeInSine	Slow start
2	easeOutSine	Slow end
3	easeInOutSine	Smooth both ends

Number	Name	Character
4	easeInQuad	Accelerate
5	easeOutQuad	Decelerate
6	easeInOutQuad	Smooth acceleration
7	easeInCubic	Strong acceleration
8	easeOutCubic	Strong deceleration
9	easeInOutCubic	Powerful smooth
10	easeInBack	Anticipation (goes backward first)
11	easeOutBack	Overshoot
12	easeInOutBack	Anticipation + overshoot
13	easeInElastic	Elastic snap-in
14	easeOutElastic	Bouncy arrival

Use by name or number in DSL:

```
<!-- By name -->
cue="ui(action:'controlXYRecall', preset:'state1', dur:2, ease:'easeOutElastic')"
```

```
<!-- By number -->
cue="ui(action:'controlXYRecall', preset:'state1', dur:2, ease:14)"
```

Complete DSL Action Reference

All controlXY preset actions work through `ui(action:...)` syntax.

1. Save Preset

```
<!-- Save all pads -->
ui(action:"controlXYSave", preset:"stateName")
```

```
<!-- Save specific pad -->
ui(action:"controlXYSave", preset:"stateName", uid:"pad1")
```

Examples:

```
<!-- Playhead trigger -->
<rect x="500" cue="ui(action:'controlXYSave', preset:'verse1')"/>
```

```
<!-- Button -->
<g cue="button(
  trigger:ui(action:'controlXYSave', preset:'myState'),
  style(label:'Save State', x:10, y:10)
)"/>
```

2. Recall Preset

```
<!-- Instant -->
ui(action:"controlXYRecall", preset:"stateName")
```

```
<!-- With tween -->
ui(action:"controlXYRecall", preset:"stateName", dur:2, ease:"easeInOutSine")
```

Examples:

```
<!-- Playhead trigger with animation -->
<rect x="1000"
  cue="ui(action:'controlXYRecall', preset:'chorus', dur:3, ease:'easeOutElastic')"/>
```

```
<!-- Button with quick snap -->
<g cue="button(
  trigger:ui(action:'controlXYRecall', preset:'breakdown', dur:0.5),
  style(label:'□ Breakdown', x:10, y:50)
)"/>
```

3. Define Sequence

```
ui(action:"controlXYDefineSequence", name:"seqName", steps:"preset1,preset2,preset3")
```

Examples:

```
<!-- Define on load -->
<rect x="100"
  cue="ui(action:'controlXYDefineSequence',
    name:'intro_pattern',
```

```

        steps:'a,b,c,a')"/>

<!-- Button to create sequence -->
<g cue="button(
    trigger:ui(action:'controlXYDefineSequence',
        name:'verse_loop',
        steps:'verse1,verse2,verse1'),
    style(label:'Create Loop', x:10, y:90)
)"/>

```

4. Play Sequence

```
ui(action:"controlXYSequence", seq:"seqName", dur:2, ease:"easeInOutSine", loop:true)
```

Examples:

```

<!-- Auto-play when playhead hits -->
<rect x="2000"
    cue="ui(action:'controlXYSequence',
        seq:'intro_pattern',
        dur:1.5,
        loop:false)"/>

<!-- Button with looping -->
<g cue="button(
    trigger:ui(action:'controlXYSequence',
        seq:'verse_loop',
        dur:2,
        loop:true),
    style(label:'□ Loop Verse', x:10, y:130)
)"/>

```

5. Stop Sequence

```
ui(action:"controlXYSequenceStop")
```

Examples:

```

<!-- Playhead trigger -->
<rect x="4000" cue="ui(action:'controlXYSequenceStop')"/>

<!-- Button -->
<g cue="button(
    trigger:ui(action:'controlXYSequenceStop'),
    style(label:'□ Stop', x:10, y:170)
)"/>

```

OSC Output (Bonus Feature)

While spatial animation is the primary use case, all handle movements can **simultaneously control external software**.

Enable OSC

```

<rect id="pad1"
    cue="controlXY(uid:synth, touchpt:dot1, osc:true)"/>

```

Custom Address & Throttle

```

<rect id="pad1"
    cue="controlXY(uid:synth, touchpt:dot1, osc:50, oscAddr:'max/xy')"/>

```

OSC Messages Sent

XY only (no rotation):

```
/controlXY/synth 0.42 0.87
```

XY + Rotation:

```
/controlXY/synth 0.42 0.87 0.65
```

Third value is rotation (0.0-1.0, normalized by hmode).

Multiple handles:

```
/controlXY/synth/dot1 0.42 0.87 0.33
```

```
/controlXY/synth/dot2 0.15 0.63 0.78
```

Custom address:

```
/max/xy 0.42 0.87 0.65
```

Works with Max/MSP, Pure Data, SuperCollider, TouchOSC, and any OSC-compatible software.

Preset Panel UI

Opening the Panel

1. **Click button** in top-right of any controlXY pad
2. **Keyboard: Alt+Shift+P**
3. **Console:** `window.controlXYPresetUI.toggle()`

Panel Features

- **Save Section:** Type name, click
- **Presets List:** Click to recall, to delete
- **Recall Options:** Set duration (seconds) and easing
- **Sequences:** Play/stop defined sequences
- **Import/Export:** Save/load preset files

Keyboard Shortcut

Alt+Shift+P toggles the panel (changed from Ctrl+Shift+P to avoid Chrome conflict).

Storage & Persistence

Presets auto-save to `scores/<project>/controlxy-presets.json`:

```
{
  "presets": {
    "intro": {
      "pad1": {
        "dot1": { "x": 0.25, "y": 0.75, "p": 0.33 },
        "dot2": { "x": 0.80, "y": 0.30, "p": 0.67 }
      }
    }
  },
  "sequences": {
    "verse_pattern": ["verse1", "verse2", "verse1"]
  },
  "updatedAt": 1704067200000,
  "projectId": "myProject"
}
```

The `p` value stores rotation (0.0-1.0) if the handle has rotation control. It is saved, recalled, and **smoothly tweened** between presets just like `x` and `y`. Loaded automatically when project loads.

CSS Styling

Handle Classes

```
.controlxy-handle {
  cursor: grab;
  transition: opacity 0.15s;
}

.controlxy-handle:hover {
  opacity: 0.8;
}

.controlxy-handle--active {
  cursor: grabbing;
  filter: drop-shadow(0 0 10px rgba(100, 200, 255, 0.9));
}
```

Label Styling

```
.controlxy-label {
  font-family: 'SF Mono', 'Monaco', monospace;
  font-size: 11px;
  fill: #fff;
  filter: drop-shadow(0 1px 2px rgba(0, 0, 0, 0.8));
}
```

Toggle Button

The button is automatically added to each pads top-right corner. Style via:

```
.controlxy-toggle-btn {
  background: rgba(30, 30, 30, 0.9);
  border: 1px solid rgba(255, 255, 255, 0.3);
}

.controlxy-toggle-btn:hover {
  background: rgba(30, 30, 30, 1);
  transform: scale(1.1);
}
```

Complete API Reference

JavaScript Console API

```
// ===== PRESETS =====
controlXYPresets.save(name, uidFilter?)
controlXYPresets.recall(name, options?)
controlXYPresets.delete(name)
controlXYPresets.list() // Returns array of preset names
controlXYPresets.get(name) // Returns preset data

// ===== TWEENING =====
controlXYPresets.tweenTo(positions, options?) // positions: { uid: { handleId: { x, y, p? } } }
controlXYPresets.stopAllTweens()

// ===== SEQUENCES =====
controlXYPresets.defineSequence(name, steps)
controlXYPresets.playSequence(name, options)
controlXYPresets.stopSequence()
controlXYPresets.getActiveSequence() // Returns current sequence info

// ===== SCENES =====
controlXYPresets.saveScene(name) // Save all handle positions + launcher state
controlXYPresets.recallScene(name, options?) // Recall scene (supports dur/ease tweening)
controlXYPresets.deleteScene(name)
controlXYPresets.listScenes()
controlXYPresets.getScene(name)

// ===== PERSISTENCE =====
controlXYPresets.init(projectId) // Called automatically
controlXYPresets.export() // Returns JSON string
controlXYPresets.import(json, merge?)
controlXYPresets.importFromProject(projectId, merge?)

// ===== UI =====
controlXYPresetUI.show()
controlXYPresetUI.hide()
controlXYPresetUI.toggle()
controlXYPresetUI.refresh()
```

Compositional Patterns

Pattern 1: Verse-Chorus Automation

1. Save state at verse start
2. Save state at chorus start
3. Tween between them at transitions
4. Add button overrides for live performance

Pattern 2: Looping Textures

1. Define 3-4 related states
2. Create sequence with varied timing
3. Loop sequence with 'loop:true'
4. Bind to visual parameters for evolving texture

Pattern 3: Build-Tension-Release

1. Start at calm state (center, low values)
2. Sequence through increasingly tense positions
3. Climax: rapid sequence or elastic bounce
4. Release: slow tween back to calm

Pattern 4: Call-Response

1. Manual control for "call" phrase
2. Automated sequence for "response"
3. Alternate between live and programmed

Pattern 5: Polytemporal Layers

1. Multiple pads, each with own sequence
2. Different loop lengths (3, 5, 7 steps)
3. Creates phasing/evolving relationships
4. Each pad controls different visual layer

Technical Notes

- controlXY is a **control-plane cue**, not temporal
- Registered during assignCues(), not via playhead
- Handles initialized at center of bounds (XY) and 0 rotation
- Uses Pointer Events API (works with touch, mouse, stylus)
- All tweening uses requestAnimationFrame for smooth 60fps
- Event propagation stopped to prevent score dragging
- Multi-touch: each pointer controls nearest available handle
- **Rotation handles:**
 - Can be shared (one handle for all touchpoints) or individual
 - Coordinate system: 0 = 7 oclock, clockwise positive
 - limited mode applies hard stops using angle clamping
 - continuous mode normalizes angles to 0-360 with wraparound

Summary

controlXY fundamentally transforms the score from **static notation** into **dynamic canvas**. By combining:

1. **Spatial control surfaces** (the pads)
2. **Parameter binding** (connecting space to parameters)
3. **Preset/sequence system** (programming choreography)
4. **Live performance** (manual override)

composers gain the ability to create **complex, evolving visual/sonic animations** directly within the score itself. The score becomes both **instrument and notation**, collapsing the traditional divide between composition and performance.

As a bonus, OSC output allows the same spatial gestures to control external synthesis and media systems, making controlXY a **unified interface** for all aspects of a multimedia performance.

Score-as-instrument. Instrument-as-score.

Quick Start Checklist

1. Add CSS: <link rel="stylesheet" href="controlxy-preset-ui.css">
2. Define pad: <rect cue="controlXY(uid:pad1, touchpt:dot1)"/>
3. *Optional:* Add rotation: handle:knob, hmode:limited for volume-style controls
4. Open UI: Press **Alt+Shift+P** or click
5. Save states: Move handles, name them, click
6. Animate: Use ui(action: 'controlXYRecall', ...) on playhead triggers

7. Choreograph: Define sequences, play/loop them
8. Bind: Connect handle positions to visual parameters
9. Perform: Override automation with buttons or manual control

Now go create some animated score-instruments!

Chapter 6

Tools

This section covers external tools that extend the Oscilla authoring workflow. While scores can be created entirely by editing SVG element IDs in Inkscape's Object Properties dialog, dedicated tooling can streamline the process of building and applying cues.

OSCILLA Inkscape Extension

A smart cue editor for creating OSCILLA interactive graphic notation within Inkscape.

Overview

The OSCILLA Inkscape Extension provides a graphical interface for applying OSCILLA DSL cues to SVG elements. Instead of manually typing cue syntax into element IDs, you select cue types from organized categories, configure parameters using appropriate widgets, and apply them directly to selected elements.

Installation

Automatic Installation

```
cd oscilla-inkscape-extension
./install.sh
```

The installer detects your operating system and copies files to the correct Inkscape extensions folder.

Manual Installation

Copy all files to your Inkscape extensions folder:

OS	Extensions Path
Linux	~/.config/inkscape/extensions/
macOS	~/Library/Application Support/org.inkscape.Inkscape/config/inkscape/extensions/
Windows	%APPDATA%\inkscape\extensions\

Restart Inkscape after installation.

First-Time Setup: Keyboard Shortcut

For efficient workflow, bind “Apply Queued Cue” to a keyboard shortcut:

1. In Inkscape: **Edit Preferences Interface Keyboard**
2. In the search box, type Apply Queued
3. Click on the “Apply Queued Cue” entry
4. Press your desired shortcut (recommended: Ctrl+Shift+Q)
5. Click **Close**

This shortcut applies cues configured in the Smart Cues Editor to your selected elements.

For detailed keyboard shortcut documentation, see the [Inkscape Keyboard Shortcut Guide](#).

Components

The extension consists of two parts under **Extensions OSCILLA:**

OSCILLA Smart Cues Editor

The main interface for building cues. Opens as a floating window that stays on top of Inkscape while you work.

Note: The editor cannot dock inside Inkscape - it remains a separate floating window. However, it stays on top so you can see both windows simultaneously.

Apply Queued Cue

Applies the configured cue to selected elements in Inkscape. Bind this to a keyboard shortcut for the fastest workflow.

Workflow

Standard Workflow

1. **Open the Smart Cues Editor:** Extensions OSCILLA OSCILLA Smart Cues Editor
2. **Keep the editor open** - it floats above Inkscape
3. **Configure your cue:**
 - Select a category (Timing, Animation, Visual, etc.)
 - Select a cue type
 - Optionally select a preset
 - Adjust parameters as needed
4. **Click “Apply to Selection”** - this queues the cue
5. **In Inkscape:** Select one or more elements
6. **Press your keyboard shortcut** (e.g., Ctrl+Shift+Q) - cue is applied!

Quick Workflow with Presets

Once youve saved presets for your commonly used cues:

1. Open Smart Cues Editor (keep it open)
2. Select preset from dropdown
3. Click “Apply to Selection”
4. Select element in Inkscape Press Ctrl+Shift+Q

Stacking Multiple Cues

To add multiple cues to one element:

1. Apply the first cue normally
2. Check **Append to existing ID**
3. Configure and apply additional cues

Result: `pause(dur:4) scale(values:[1,1.3,1], dur:2)`

Using the Smart Cues Editor

Interface Elements

Category Dropdown

Select from seven cue categories:

Category	Cue Types
Timing & Navigation	stop, pause, speed, nav, page, stopwatch, metro
Animation	scale, scaleXY, rotate, o2p
Visual	color, fade, text
Audio / Video	audio, audioPool, audioImpulse, video
Synth	synth, synthStop
OSC	osc, oscCtrl
Interaction	button, reuse, use

Cue Type Dropdown

Shows available cues for the selected category. When you select a cue type, only the relevant parameters appear below.

Preset Dropdown

Load pre-configured parameter combinations. Select “Custom ” to configure parameters manually.

Parameters Area

Dynamic parameter widgets appropriate to each parameter type:

Widget	Used For
Text entry	Free text, file paths, color values
Spin button	Integer values with increment/decrement
Float spinner	Decimal values with precision
Dropdown	Fixed option selection
Checkbox	Boolean on/off values
Color button	Color picker (alongside text entry)

Preview

Shows the cue string as you configure parameters. Updates in real-time. Wraps long cues across multiple lines.

Append Checkbox

When checked, the cue is added to the elements existing ID rather than replacing it. Use this to stack multiple cues on one element.

Buttons

Button	Action
Save Preset	Save current parameters as a reusable preset
Copy to Clipboard	Copies the cue string for manual pasting
Apply to Selection	Queues the cue for application via keyboard shortcut

Presets

Using Presets

Presets provide quick access to common cue configurations. Select from the Preset dropdown to auto-fill parameters, then adjust as needed.

Saving Presets

1. Configure parameters as desired
2. Click **Save Preset**
3. Enter a name for the preset
4. Click **Save**

The preset is immediately available in the Smart Cues Editor dropdown.

Customizing Presets Manually

Presets are stored in `oscilla_presets.json` in your extensions folder. You can edit this file directly.

File Location

OS	Path
Linux	<code>~/.config/inkscape/extensions/oscilla_presets.json</code>
macOS	<code>~/Library/Application Support/org.inkscape.Inkscape/config/inkscape/extensions/oscilla_presets</code>
Windows	<code>%APPDATA%\inkscape\extensions\oscilla_presets.json</code>

Structure

```

{
  "category_id": {
    "cue_name": [
      {
        "name": "Preset Display Name",
        "params": {
          "param_name": value
        }
      }
    ]
  }
}

```

Example: Adding Pause Presets

```

{
  "timing": {
    "pause": [
      {
        "name": "Short Pause (4s)",
        "params": {
          "dur": 4
        }
      },
      {
        "name": "Long Pause with Countdown",
        "params": {
          "dur": 12,
          "count": true
        }
      }
    ]
  }
}

```

Category IDs

Use these exact category IDs in the JSON:

Display Name	Category ID
Timing & Navigation	timing
Animation	animation
Visual	visual
Audio / Video	audio
Synth	synth
OSC	osc
Interaction	interaction

Cue Names

Use the exact cue name as shown (case-sensitive):

stop, pause, speed, nav, page, stopwatch, metro, scale, scaleXY, rotate, o2p, color, fade, text, audio, audioPool, audioImpulse, video, synth, synthStop, osc, oscCtrl, button, reuse, use

Validating Your Presets

After editing, verify the JSON is valid:

```
python3 -m json.tool oscilla_presets.json > /dev/null && echo "Valid JSON"
```

Restart the Smart Cues Editor to load updated presets.

Troubleshooting

Extension not appearing in menu

1. Restart Inkscape completely
2. Verify files are in the correct extensions folder
3. Check for Python errors: `python3 -c "import oscilla_smart_cues_gtk"`

Editor wont open

Check the launcher script is executable and Python 3 with GTK is available:

```
python3 -c "import gi; gi.require_version('Gtk', '3.0'); from gi.repository import Gtk; print('GTK OK')"
```

Cue not applying to element

1. Ensure an element is selected in Inkscape
2. Ensure youve clicked “Apply to Selection” in the Smart Cues Editor
3. Press your keyboard shortcut (e.g., Ctrl+Shift+Q)
4. If still not working, check /tmp/oscilla_cue.txt exists after clicking Apply

Keyboard shortcut not working

1. Verify the shortcut is bound: Edit Preferences Interface Keyboard search “Apply Queued”
2. Make sure Inkscape window is focused when pressing the shortcut
3. Try a different shortcut combination if theres a conflict

Presets not loading

1. Validate JSON syntax: `python3 -m json.tool oscilla_presets.json`
2. Check category and cue names match exactly (case-sensitive)
3. Restart the Smart Cues Editor

Editor not staying on top

On some systems/window managers, “always on top” may not work. Try: - Right-click the window title bar “Always on Top” (if available) - Use your window managers settings to pin the window

Files

File	Purpose
<code>oscilla_smart_cues_gtk.py</code>	Main GTK editor application
<code>oscilla_smart_cues_launcher.py</code>	Inkscape extension that launches editor
<code>oscilla_smart_cues_launcher.inx</code>	Menu entry definition
<code>oscilla_apply_cue.py</code>	Extension that applies cue to selection
<code>oscilla_apply_cue.inx</code>	Menu entry definition
<code>oscilla_presets.json</code>	User-editable preset configurations

Chapter 7

Developer Reference

This section provides technical documentation for developers working on Oscilla itself or extending it with new cue types, control systems, or integrations. It covers the internal architecture, module organisation, synchronisation protocol, and the patterns used throughout the codebase.

These pages assume familiarity with the user-facing features documented in the preceding sections. Where a user doc explains what a feature does, the corresponding developer doc explains how it is implemented and how to modify or extend it.

Oscilla Complete System Documentation

Version: 0.4.2

Purpose: Browser-based framework for animated, cue-driven graphic scores

Technologies: SVG, JavaScript (ES Modules), Node.js, WebSockets, OSC

Table of Contents

1. [System Overview](#)
 2. [Architecture](#)
 3. [Server Components](#)
 4. [Client Components](#)
 5. [Cue System](#)
 6. [Transport & Synchronization](#)
 7. [OSC Integration](#)
 8. [Project Structure](#)
 9. [File Reference](#)
-

System Overview

Oscilla is a browser-based platform for real-time animated graphic scores. It enables:

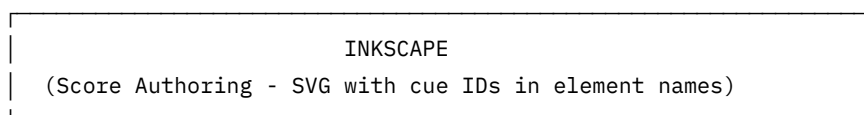
- **Animated and spatial graphic notation** SVG-based scores with cue-driven animations
- **Networked performances** Multiple clients synchronized via WebSocket
- **Cue-driven timing, navigation, and media** DSL for defining behaviors via SVG element IDs
- **OSC integration** Bidirectional communication with external audio/media engines
- **Hybrid fixed-form and open-form works** Support for linear playback and interactive navigation

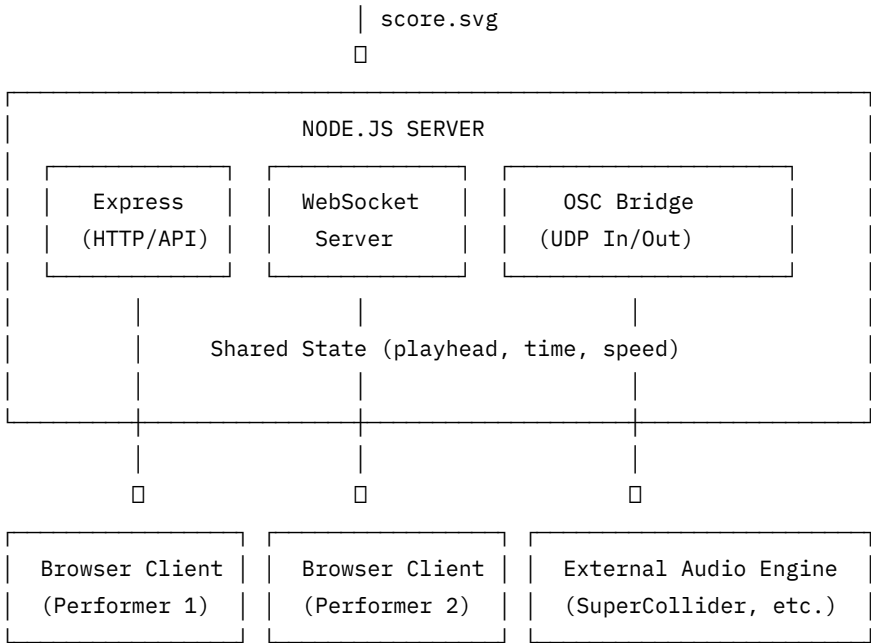
What Oscilla Is NOT

- A DAW (Digital Audio Workstation)
 - A notation engraver (like Sibelius/MuseScore)
 - A collaborative score editor
-

Architecture

High-Level Data Flow





Key Concepts

Term	Description
scoreWidth	Width of SVG in viewBox units (world space)
canonicalRenderedWidth	Pixel width reported by first client (visual authority)
playheadX	Playback position in world units
elapsedTime	Time elapsed in milliseconds
speedMultiplier	Playback speed factor (1.0 = normal)
cue	Behavior definition encoded in SVG element ID

Server Components

server.js (Main Server ~2175 lines)

The central Node.js server providing:

HTTP/Express Endpoints: - GET / Serves static files from /public - GET /scores/:project Project score files - GET /api/version Returns Oscilla version - GET /api/projects Lists available projects - GET /api/audio-tree/:project Audio file browser - POST /api/upload-audio/:project Audio file upload - POST /api/project/new Create new project - POST /api/project/save-as Duplicate project - GET /api/project/export/:name Export as .oscilla - POST /api/project/import Import .oscilla file - POST /save-preferences/:project Save project preferences - GET /config WebSocket configuration for clients

WebSocket Message Types (Inbound from Clients):

Type	Purpose
play	Start playback
pause	Pause playback
jump	Jump to position
cueStop	Stop cue triggered
cue_pause	Pause cue with duration
cue_triggered	Cue was triggered (broadcast to others)
set_speed_multiplier	Change playback speed
set_duration	Set score duration
score_meta	Report score dimensions

Type	Purpose
update_client_name	Change client display name
repeat_update	Sync repeat cue state
osc	Send OSC message
osc_rotate	Rotation OSC message
osc_scale	Scale OSC message
osc_fade	Fade OSC message
osc_value	Generic OSC value
osc_control	Control parameter OSC
osc_audio_*	Audio trigger OSC messages
annotation_*	Shared annotation CRUD
countdown_*	Server-owned countdown timer

WebSocket Message Types (Outbound to Clients):

Type	Purpose
welcome	Client connection acknowledged with name
sync	Authoritative state broadcast (4Hz)
client_list	Connected clients update
jump	Position jump broadcast
cueStop	Stop cue broadcast
cue_pause	Pause cue broadcast
repeat_update	Repeat state sync
repeat_state_map	Full repeat state on connect
annotation_*	Annotation sync to other clients

Shared State Object:

```
sharedState = {
  elapsedTime: 0,           // ms
  isPlaying: false,
  playheadX: 0,            // world units
  duration: null,          // ms (from project)
  speedMultiplier: 1.0,
  startTimestamp: null,    // performance.now() anchor
  scoreWidth: null,
  canonicalRenderedWidth: null,
  countdown: null         // server-owned countdown state
}
```

serverUtils.js

Project management utilities: - createNewProject(name) Copy template to new project - saveProjectAs(source, name) Duplicate existing project - exportProject(name, res, version) Create .oscilla ZIP archive - importProject(buffer, name) Extract and validate .oscilla file

server-gui.js

Electron wrapper for standalone builds: - Spawns server.js as child process - Displays server status in GUI window - Shows connected clients and log output

Client Components

public/js/app.js (Entry Point)

Orchestrates client initialization: - Imports and wires all major subsystems - Establishes global window bindings for cross-module access - Detects platform (desktop/mobile) and loads appropriate CSS - Initializes WebSocket connection - Sets up UI controls and event handlers

Key Global Bindings:

```

window.playheadX      // Current position
window.elapsedTime    // Current time
window.speedMultiplier // Playback speed
window.duration       // Score duration
window.scoreWidth     // Score width in world units
window.isPlaying      // Playback state
window.triggeredCues   // Set of triggered cue IDs
window.socket         // WebSocket connection

```

public/js/system/ (System Modules)

File	Purpose
socket.js	WebSocket connection, message routing, sync handling
RAF.js	RequestAnimationFrame loop management
SVGInit.js	SVG score initialization and layout
UIBindings.js	UI element event bindings
clients.js	Client list UI and audio master designation
paths.js	Path variants and playhead position utilities
projectMenuUI.js	Project selection menu
rehearsalUI.js	Rehearsal marks navigation popup

public/js/oscillaTransport.js (~2225 lines)

Transport control and synchronization: - Play/pause/stop controls - Speed adjustment (0.1x - 4.0x) - Seek bar and keyboard navigation (/ arrows) - Rehearsal mark navigation - Visual scroll synchronization - Dark mode toggle

Synchronization Model:

All clients use identical scale: $\text{canonicalScale} = \text{canonicalRenderedWidth} / \text{scoreWidth}$

Visual positioning via CSS transform (GPU-accelerated):

```

screenX = playheadX * canonicalScale
translateX = (viewportWidth / 2) - screenX
scrollStage.style.transform = translateX(`${translateX}px`)

```

public/js/projectLoader.js

Loads projects from /scores/:project/: - Fetches and parses score.svg - Loads preferences.json - Initializes score dimensions - Reports metadata to server

public/js/projectScoreSetup.js

Score DOM setup: - Extracts score elements - Auto-injects scroll groups - Processes SVG layers - Applies transformations

Cue System

Overview

Cues are behaviors defined by SVG element ID attributes. When the playhead crosses an element, its ID is parsed and the corresponding handler executes.

Parser (public/js/parser/parser.js ~2681 lines)

Chevrotain-based DSL parser supporting:

Cue Syntax Examples:

```

pause(5)           // Pause for 5 seconds
speed(0.5)         // Set speed to 0.5x
speed(2 dur:4)     // Ramp to 2x over 4 seconds
nav(A)             // Jump to rehearsal mark A
page(intro dur:3)  // Show page "intro" for 3 seconds

```

```

audio(file.mp3 vol:0.8)      // Play audio at 80% volume
rotate(360 dur:2 ease:inOut) // Rotate 360° over 2 seconds
scale(2 dur:1)              // Scale to 2x over 1 second
osc(/addr 0.5)              // Send OSC message
synth(freq:440 amp:0.5)     // Trigger synthesizer
text("Hello" dur:3)        // Display text for 3 seconds

```

Cue Dispatcher (public/js/cues/cueDispatcher.js)

Routes parsed AST to appropriate handlers:

```

switch (ast.type) {
  case "cuePause":  return handlePauseCue(ast);
  case "cueSpeed":  return handleSpeedCue(ast);
  case "cuePage":   return handlePageCue(ast);
  case "cueNav":    return handleNavCue(ast);
  case "cueAudio":  return handleAudioCue(ast);
  case "cueRotate": return handleRotateCue(ast);
  case "cueScale":  return handleScaleCue(ast);
  case "cueO2P":    return handleO2PCue(ast);
  case "cueColor":  return handleColorCue(ast);
  case "cueOsc":    return handleOscCue(ast);
  case "cueSynth":  return handleSynthCue(ast);
  case "cueText":   return handleTextCue(ast);
  // ... etc
}

```

Cue Handlers (public/js/cues/)

File	Cue Types	Description
pause.js	pause(N)	Timed pause with countdown overlay
stop.js	stop()	Stop playback
speed.js	speed(N)	Set/ramp playback speed
nav.js	nav(mark)	Jump to rehearsal mark
page.js	page(id)	Fullscreen SVG overlay
audio.js	audio(), audioPool(), audioImpulse()	Audio playback
video.js	video()	Video playback
synth.js	synth()	Web Audio synthesizer
rotate.js	rotate()	Element rotation animation
scale.js	scale()	Element scale animation
o2p.js	o2p()	Object-to-path animation
color.js	color()	Color animation
fade.js	fade()	Opacity animation
text.js	text()	Text display overlay
osc.js	osc()	Send OSC messages
oscCtrl.js	oscCtrl()	OSC control messages
controlXY.js	controlXY()	XY pad control
button.js	button()	Interactive buttons
metro.js	metronome()	Visual/audio metronome
timers.js	stopwatch()	Stopwatch/countdown display
ui.js	ui()	UI control (show/hide elements)

Animation Shared (public/js/cues/animShared.js)

Common animation utilities: - Easing functions - Duration parsing - Loop handling - Prestate management (ghostClickable, fadein)

Transport & Synchronization

Clock Model

The server maintains an authoritative transport clock:

```
// Server calculates elapsed time from wall clock
const wallSeconds = (performance.now() - startTimestamp) / 1000;
const elapsedMs = wallSeconds * speedMultiplier * 1000;

// Playhead position derived from elapsed time
playheadX = (elapsedTime / duration) * scoreWidth;
```

Sync Broadcast (4Hz)

Server broadcasts state every 250ms:

```
{
  type: 'sync',
  state: {
    elapsedTime,
    isPlaying,
    scoreWidth,
    playheadX,
    speedMultiplier,
    startTimestamp,
    canonicalRenderedWidth,
    duration,
    countdown
  },
  serverTime: Date.now()
}
```

Client Sync Handling

Clients apply server state with drift protection: - Small drift (<2% of score): Gradual correction - Medium drift (2-5%): Immediate position update - Large drift (>5%): Teleport mode (suppress cue triggers)

```
// Teleport mode prevents cue cascade during large jumps
window._isTeleporting = true;
window.suppressCueTriggers = true;
window._skipTriggerFrame = 5;
```

OSC Integration

Configuration

```
oscConfig = {
  localAddress: "0.0.0.0", // Listen address
  localPort: 57121, // OSC input port
  remoteAddress: "127.0.0.1", // Destination address
  remotePort: 57120 // OSC output port
}
```

Outbound OSC Messages

Address	Args	Description
/stopwatch	[time_string]	Elapsed time (MM:SS)
/cue/trigger	[cue_number:i]	Cue triggered
/oscilla/rotate/{uid}	[deg:f, rad:f, norm:f]	Rotation value
/oscilla/scale/{uid}	[sx:f, sy:f]	Scale values
/oscilla/fade/{uid}	[value:f]	Fade value
/oscilla/audio/pool	[file:s, amp:f, pan:f, pitch:f, fadeIn:f, fadeOut:f]	Audio pool
/oscilla/audio/impulse	[file:s, amp:f, pan:f, pitch:f, fadeIn:f, fadeOut:f]	Audio impulse
/oscilla/audio/trigger	[file:s, vol:f, loop:i]	Audio trigger
/oscilla/audio/stop	[file:s, fadeOut:f]	Stop audio
/oscilla/control/{addr}	[value:f, t:f]	Control value

Address	Args	Description
/oscilla/o2p/{uid}	[x:f, y:f, angle:f]	Object-to-path position

OSC-In Support

Server can receive OSC and route to clients via WebSocket:

```
case "osc_in":
case "osc_control":
  handleOSCIn(address, args);
```

Project Structure

Score Project Layout

```
public/scores/{project}/
├─ score.svg           # Main score file (required)
├─ preferences.json    # Project settings
├─ audio/              # Audio files
│   └─ samples/
│       └─ recordings/
├─ video/              # Video files
├─ pages/              # SVG page overlays
└─ oscilla.manifest.json # Export manifest (in .oscilla archives)
```

preferences.json

```
{
  "projectName": "my-score",
  "duration": 300,
  "scoreWidth": 40960,
  "enableOSC": true,
  "oscAddress": "/oscilla",
  "theme": "dark"
}
```

Score SVG Structure

```
<svg viewBox="0 0 40960 1080">
  <!-- Main scrolling content -->
  <g id="scroll-content">
    <!-- Notation elements -->
    <rect id="pause(5)" .../>
    <circle id="rotate(360 dur:2)" .../>
    <path id="audio(click.mp3)" .../>
  </g>

  <!-- Rehearsal marks -->
  <g id="rehearsal-marks">
    <text id="mark-A">A</text>
    <text id="mark-B">B</text>
  </g>

  <!-- Reusable blocks -->
  <defs>
    <g id="reuse-pattern1">...</g>
  </defs>
</svg>
```

File Reference

Server Files

File	Lines	Purpose
server.js	~2175	Main server (Express, WebSocket, OSC)

File	Lines	Purpose
serverUtils.js	~145	Project management utilities
server-gui.js	~165	Electron GUI wrapper
gui.html	~123	Server monitor HTML
package.json	~48	Dependencies and build config

Client Core Files

File	Lines	Purpose
public/js/app.js	~250	Client entry point
public/js/oscillaTransport.js	~2225	Transport and sync
public/js/projectLoader.js	~700	Project loading
public/js/projectScoreSetup.js	~900	Score DOM setup
public/js/oscillaPreferences.js	~280	Preferences management
public/js/oscillaTargets.js	~450	Target resolution
public/js/oscillaObserver.js	~250	DOM observation
public/js/oscillaLive.js	~230	Live inspector
public/js/oscillaHitLabels.js	~800	Hit region labels
public/js/oscillaOSCClient.js	~270	OSC client utilities

Parser Files

File	Lines	Purpose
public/js/parser/parser.js	~2681	Chevrotain DSL parser
public/js/parser/parserSignal.js	~170	Signal reference handling
public/js/parser/preProcessPropagate.js	~170	Propagate preprocessing
public/js/parser/preProcessReuse.js	~350	Reuse block preprocessing
public/js/parser/Utils.js	~140	Parser utilities
public/js/parsers.js	~350	Legacy parser support

Cue Handler Files

File	Lines	Purpose
public/js/cues/cueDispatcher.js	~1250	Cue routing and scheduling
public/js/cues/audio.js	~1300	Audio playback
public/js/cues/synth.js	~1400	Web Audio synthesis
public/js/cues/timers.js	~2100	Stopwatch/countdown
public/js/cues/rotate.js	~750	Rotation animation
public/js/cues/scale.js	~800	Scale animation
public/js/cues/o2p.js	~1200	Object-to-path
public/js/cues/color.js	~800	Color animation
public/js/cues/text.js	~800	Text display
public/js/cues/osc.js	~650	OSC sending
public/js/cues/page.js	~350	Page overlays
public/js/cues/pause.js	~350	Pause handling
public/js/cues/metro.js	~650	Metronome
public/js/cues/controlXY.js	~950	XY control
public/js/cues/fade.js	~250	Fade animation
public/js/cues/nav.js	~200	Navigation
public/js/cues/speed.js	~320	Speed control

File	Lines	Purpose
public/js/cues/button.js	~270	Buttons
public/js/cues/video.js	~300	Video playback
public/js/cues/stop.js	~50	Stop handling
public/js/cues/ui.js	~220	UI control
public/js/cues/oscCtrl.js	~160	OSC control
public/js/cues/animShared.js	~670	Shared animation utils
public/js/cues/animation.js	~200	Animation registry

System Module Files

File	Lines	Purpose
public/js/system/socket.js	~490	WebSocket handling
public/js/system/RAF.js	~200	Animation loop
public/js/system/SVGInit.js	~200	SVG initialization
public/js/system/UIBindings.js	~280	UI event bindings
public/js/system/clients.js	~100	Client list
public/js/system/paths.js	~110	Path utilities
public/js/system/projectMenuUI.js	~80	Project menu
public/js/system/rehearsalUI.js	~50	Rehearsal navigation

Interaction Files

File	Lines	Purpose
public/js/interaction/annotationEditor.js	~1500	Annotation editing
public/js/interaction/interactionSurface.js	~700	Contribution surface
public/js/interaction/audioBrowser.js	~550	Audio file browser
public/js/interaction/audioRecorder.js	~1150	Audio recording
public/js/interaction/colorPicker.js	~300	Color picker UI
public/js/interaction/markers.js	~480	Marker management
public/js/interaction/trigger.js	~480	Trigger interactions
public/js/interaction/shared.js	~240	Shared interaction utils

Control Module Files

File	Lines	Purpose
public/js/control/controlXYPreset.js	~1400	XY preset management
public/js/control/controlXYPresetUI.js	~900	XY preset UI
public/js/control/controlXYPresetRoutes.js	~230	XY preset HTTP routes
public/js/control/controlXYPresetRoutes.mjs	~150	XY routes (ESM)
public/js/control/controlRouter.js	~320	Control message routing
public/js/control/controlIntegration.js	~280	Control integration
public/js/control/paramBinding.js	~350	Parameter binding
public/js/control/paramBus.js	~250	Parameter bus

Inkscape Extension Files

File	Purpose
tools/oscilla-inkscape-extension/oscilla_smart_cues_gtk.py	GTK cue editor
tools/oscilla-inkscape-extension/oscilla_apply_cue.py	Apply cue to selection
tools/oscilla-inkscape-extension/oscilla_presets.json	Preset definitions
tools/oscilla-inkscape-extension/install.sh	Installation script

Build/Deploy Scripts

File	Purpose
scripts/deploy.sh	Production deployment
scripts/deploy-local.sh	Local deployment
scripts/convert_score.sh	Score conversion
scripts/convert_to_plain_svg.sh	SVG simplification

Dependencies

Server (package.json)

```
{
  "dependencies": {
    "archiver": "^7.0.1",      // ZIP creation
    "chevrotain": "^11.0.3",  // Parser generator
    "express": "^4.21.2",     // HTTP server
    "multer": "^2.0.2",       // File upload
    "osc": "^2.4.5",          // OSC protocol
    "unzipper": "^0.12.3",    // ZIP extraction
    "ws": "^8.18.0",          // WebSocket
    "yargs": "^17.7.2"        // CLI arguments
  },
  "devDependencies": {
    "@11ty/eleventy": "^2.0.1", // Docs generator
    "@yao-pkg/pkg": "^6.12.0",  // Binary packaging
    "esbuild": "^0.25.0"        // Bundler
  }
}
```

Client (Vendor Libraries)

- anime.min.js Animation library
- svg-path-commander.js SVG path manipulation
- wavesurfer.min.js Audio waveform display

Quick Reference: Adding a New Cue Type

1. **Add token** to parser.js token definitions
2. **Add grammar rule** to parser class
3. **Add CST-to-AST conversion** in cstToAst()
4. **Create handler** in public/js/cues/newcue.js
5. **Add case** to cueDispatcher.js switch statement
6. **Add OSC handler** (if needed) in server.js
7. **Update Inkscape extension** presets (optional)

Oscilla Architecture Overview

This document provides a concise, system-level overview of Oscilla's architecture: major modules, entry points, and runtime data flows. It is intended as a developer-facing map to support onboarding, maintenance, and refactoring.

Purpose

Oscilla is a **browser-based, cue-driven graphic notation platform** built around SVG, WebSockets, and OSC.

- Scores are authored as SVG documents.
- Behaviour is encoded via ID-based cues and mini-DSL syntax.
- The client executes animations, triggers audio, and emits network and OSC events.

The system is designed to support **composed notation**, **live execution**, and **networked contribution** within a single runtime.

High-Level Entry Points

Server

`server.js` (repo root)

- Express server for serving `/public`
 - WebSocket endpoint for client sync
 - OSC bridge to external audio / media engines
 - Project import/export (`.oscilla`)
 - Development and standalone binary lifecycle
-

Client

`public/index.html`

- Main frontend entry point
- Loads UI shell and client JavaScript
- Boots the Oscilla runtime

`public/scores/<project>/score.svg`

- Per-project authored score
 - Contains SVG graphics, cue IDs, reusable blocks
 - Primary data source for runtime behaviour
-

Frontend Architecture (`public/js/`)

`app.js` Client Bootstrap / Orchestration

Role: High-level runtime composition and startup sequence.

Responsibilities: - Imports and wires together all major subsystems - Establishes global runtime bindings on `window` - Orchestrates project loading, score setup, and initialization order - Detects platform (desktop/mobile) and applies conditional UI logic

Key integrations: - Transport and playhead (`oscillaTransport.js`) - Cue dispatch (`oscillaCueDispatcher.js`) - Animation registry (`oscillaAnimation.js`) - Contribution Surface (`oscillaAnnotations.js`) - Audio handling (`oscillaAudio.js`) - Reuse/template system (`reuse.js`) - UI wiring (menus, preferences, buttons)

`app.js` intentionally contains minimal feature logic; it composes modules rather than implementing behaviour.

Animation & Cue Execution

oscillaAnimation.js

Role: Cue parsing, animation assignment, and lifecycle management.

Responsibilities: - Scan SVG DOM for cue-related IDs - Parse mini-DSL expressions - Register animations via `registerAnimation` - Track running animations - Delegate to specific handlers (scale, rotate, path-follow, text)

Key exports: - `registerAnimation` - `registerRunningAnimation` - `clearRunningAnimation` - `animationAssign` - `debugDump`

Touchpoints: - Invoked during SVG initialization - Called by cue dispatcher on playback events

oscillaCueDispatcher.js

Role: Scheduling and dispatch of cue events.

Responsibilities: - Schedule cue start times - Emit cue completion events - Bridge between playhead timing and execution layers

Key functions: - `scheduleCueStart` - `emitCueComplete`

Contribution Surface

oscillaAnnotations.js

Role: Browser-native contribution layer (HTML overlays over SVG).

Responsibilities: - Create, edit, move, and delete contributions - Manage Contribution Surface state - Execute audio triggers (click or playhead) - Manage spatial extent regions - Handle audio-only contributions - Persist contributions (`localStorage`) - Optional WebSocket synchronization

Key concepts: - Contributions as executable, spatial entities - Trigger pools for directory-based audio - Integration with audio and timer subsystems

Exposed API: - `window.oscillaAnnotations`

Audio System

oscillaAudio.js

Role: Audio playback primitives.

Responsibilities: - Play single-file audio - Manage directory-based pools - Run continuous impulse processes - Stop and manage impulse lifecycles

Primary functions: - `handleAudioCue` - `handleAudioPoolCue` - `handleAudioImpulseCue` - `stopAudioImpulse`

Used by: - Contribution Surface - Cue-triggered media execution

OSC Integration

oscillaOSC.js

Role: OSC message construction and transport.

Responsibilities: - Construct OSC messages from visual and cue data - Sample visual geometry (`sampleVisual()`) - Send OSC messages via server bridge

Timing & Utilities

oscillaTimers.js

- Stopwatch and timing utilities
- Shared scheduling primitives

oscillaHitLabels.js

- Visual hit-labels for OSC and interaction feedback
 - UI indicators for triggered events
-

Reuse / Template System

reuse.js

Role: SVG reusable block system.

Responsibilities: - Register reusable `<g id="reuse(...)">` blocks - Inject clones for `<g id="use(...)">` placeholders - Support template-based score construction

Key exports: - `registerReuseBlocks` - `preloadReuseBlocksFromPages` - `autoInjectUseBlocks` - `handleUse` - `buildPageRegistryFromDirIndex`

utils.js

- SVG geometry and transform helpers
 - Bounding box and alignment utilities
 - ID management for reuse blocks
-

Runtime Data Flows

Application Startup

1. `index.html` loads client bundle
 2. `app.js` initializes runtime
 3. Project assets loaded (SVG, pages, audio)
 4. Reuse blocks registered and injected
 5. Contribution Surface initialized
 6. SVG scanned and animations assigned
-

Cue Execution Flow

1. Playhead or UI event triggers cue
 2. `oscillaCueDispatcher` schedules execution
 3. `oscillaAnimation` runs visual behaviour
 4. Contribution Surface may execute audio triggers
 5. OSC messages emitted if configured
-

Network & OSC

- WebSocket sync for shared contributions
 - Server relays messages between clients
 - OSC messages forwarded to external engines
-

Server & Build System

- `server.js`: Express + WebSocket + OSC bridge
 - `build.sh` and scripts: standalone binary packaging
 - Version file maintained at repo root
-

Documentation

- `public/docs/` and `docs/`
 - Cue references, cheatsheets, workflows
 - Inkscape extension documentation
-

Immediate Technical Notes

Low-risk cleanup candidates:

- Consolidate duplicated helpers (e.g. `clamp01`) into shared utils
- Audit module-local debug helpers for removal or centralization
- Consider namespacing global APIs under `window.oscilla`
- Ensure Contribution Surface import/export hooks are exposed in UI if retained

Summary

Oscillas architecture separates **authored structure** (SVG Score Layer) from **live execution and contribution** (Contribution Surface), coordinated through a cue dispatcher and unified runtime.

This modular design supports composed scores, live interfaces, and distributed performance within a single browser-based system.

Oscilla Multi-Client Visual Synchronization

Overview

Oscilla synchronizes a scrolling SVG score across multiple clients so the playhead is always over the **same score content** regardless of screen size or aspect ratio.

Core principle: Sync only **world coordinates** (`playheadX`). Each client computes its own pixel positions locally.

Key Terms

Term	Definition
<code>scoreWidth</code>	SVG width in <code>viewBox</code> units (world space). Shared by all clients.
<code>playheadX</code>	Playback position in world units. This is what gets synced.
<code>localRenderedWidth</code>	Actual pixel width of SVG on this client
<code>localScale</code>	<code>localRenderedWidth / scoreWidth</code> computed per client

How It Works

Server broadcasts: `playheadX = 10000` (world units)

iPad (1024px tall viewport):
SVG renders 8000px wide → `localScale = 0.195`
`screenX = 10000 × 0.195 = 1950px`

Desktop (1200px tall viewport):
SVG renders 9400px wide → `localScale = 0.229`
`screenX = 10000 × 0.229 = 2290px`

Both show the SAME score content under the playhead.

DOM Structure

```
<div id="scoreContainer">      <!-- overflow:hidden, no native scroll -->
  <div id="scrollStage">      <!-- translated via transform -->
    <div id="scoreInner">
      <div class="oscilla-score-inner">
        <svg id="score">...</svg> <!-- height:100vh, width:auto -->
      </div>
```

```

    </div>
  </div>
</div>
<div id="playhead">          <!-- fixed at 50% viewport -->
```

Critical CSS Requirements

```
#scoreContainer {
  position: fixed;
  inset: 0;
  overflow: hidden;
}

#scrollStage {
  display: block;
  width: max-content;
  transform-origin: left top;
  will-change: transform;
}

#scoreInner {
  display: inline-block;
  width: max-content;
}

#scoreInner svg {
  display: block;
  width: auto;
  height: 100vh !important;
}
```

CSS Pitfalls

Problem	Cause	Symptom
SVG position: absolute	Removes SVG from document flow	Parent containers collapse to 0 width, scale calculations may still work but layout breaks
transition on transform	Animates score movement	Visual lag, playhead appears offset during motion
Missing display: block on SVG	Inline element whitespace	Unexpected gaps, measurement errors
max-width constraints	Limits SVG sizing	Score doesnt scale correctly to viewport

Debug check: All parent containers should have non-zero width:

```
const svg = document.querySelector("#scoreInner svg");
const stage = document.getElementById("scrollStage");
const scoreInner = document.getElementById("scoreInner");

// ALL of these should equal the SVG width, NOT zero
console.log("svg:", svg.getBoundingClientRect().width);
console.log("scoreInner:", scoreInner.getBoundingClientRect().width);
console.log("scrollStage:", stage.getBoundingClientRect().width);
```

Coordinate Extraction CRITICAL

The Problem

SVG elements can be positioned in many ways: - x, y attributes - cx, cy attributes (circles) - transform="translate(x, y)" - Nested inside transformed <g> groups - shape-inside text flowing into shapes (Inkscape) - Combinations of all the above

getBBox() and getCTM() do NOT reliably give world coordinates for complex elements like Inkscape text with shape-inside.

Correct Method: Use `getBoundingClientRect()`

Always extract world coordinates by measuring actual rendered position:

```
function getWorldX(element, svgElement) {
  const svgRect = svgElement.getBoundingClientRect();
  const elRect = element.getBoundingClientRect();

  // Screen position relative to SVG left edge
  const screenX = elRect.x - svgRect.x;

  // Convert to world coordinates
  const localScale = svgRect.width / window.scoreWidth;
  const worldX = screenX / localScale;

  return worldX;
}
```

```
function getWorldWidth(element, svgElement) {
  const svgRect = svgElement.getBoundingClientRect();
  const elRect = element.getBoundingClientRect();
  const localScale = svgRect.width / window.scoreWidth;
  return elRect.width / localScale;
}
```

Incorrect Methods (DO NOT USE)

```
// WRONG: getBBox returns local coordinates, CTM doesn't account for all transforms
const bbox = element.getBBox();
const matrix = element.getCTM();
let x = bbox.x + matrix.e; // UNRELIABLE

// WRONG: Parsing transform attribute misses shape-inside offsets
const transform = element.getAttribute("transform");
const match = transform.match(/translate\(([^\d.]+)\)/); // INCOMPLETE

// WRONG: x attribute may not exist (text uses transform instead)
const x = element.x?.baseVal?.value; // MAY BE UNDEFINED
```

Why This Matters

Inkscape often creates text elements like:

```
<text id="rehearsal_A"
      transform="translate(1764.35, 557.56)"
      style="shape-inside:url(#rect122346)">
  A
</text>
```

- The transform says $X = 1764$
- But shape-inside references a rect that adds more offset
- **Actual rendered X = 2032** (268 units different!)
- This error compounds: elements further right have larger errors

Core Functions

`scrollToPlayheadVisual()` `oscillaTransport.js`

Converts world playheadX to screen position and applies transform:

```
export function scrollToPlayheadVisual() {
  const container = window.scoreContainer;
  const stage = document.getElementById("scrollStage");
  const svg = stage?.querySelector("svg");
  if (!container || !stage || !svg || !window.scoreWidth) return;

  container.scrollLeft = 0;
  container.scrollTop = 0;

  const localRenderedWidth = svg.getBoundingClientRect().width;
  const localScale = localRenderedWidth / window.scoreWidth;

  window.localScale = localScale;
  window.localRenderedWidth = localRenderedWidth;
}
```

```

const worldPx = window.playheadX * localScale;
const viewportWidth = container.clientWidth;
const halfViewport = viewportWidth / 2;

let translateX = halfViewport - worldPx;

// Clamp at edges...
stage.style.transform = `translate3d(${translateX}px, 0, 0)`;
}

```

extractScoreElements() oscillaScoreSetup.js

Extracts cue and rehearsal mark positions. **Must use getBoundingClientRect():**

```

export const extractScoreElements = (svgElement) => {
  const svgRect = svgElement.getBoundingClientRect();
  const localScale = svgRect.width / window.scoreWidth;

  const elements = svgElement.querySelectorAll(
    "[id^='rehearsal_']", "[id^='cue']", "[id^='anchor-']"
  );

  elements.forEach((element) => {
    // CORRECT: Use getBoundingClientRect for true position
    const elRect = element.getBoundingClientRect();
    const screenX = elRect.x - svgRect.x;
    const absoluteX = screenX / localScale;
    const worldWidth = elRect.width / localScale;

    if (element.id.startsWith("rehearsal_")) {
      const id = element.id.replace("rehearsal_", "");
      newRehearsalMarks[id] = { x: absoluteX };
    } else if (element.id.startsWith("cue")) {
      newCues.push({ id: element.id, x: absoluteX, width: worldWidth });
    }
  });
};

```

Server State server.js

```

let sharedState = {
  playheadX: 0,           // world units - the key sync value
  scoreWidth: null,       // world units - from SVG viewBox
  elapsedTime: 0,
  duration: null,
  isPlaying: false,
  speedMultiplier: 1.0,
  startTimestamp: null
};

```

Server broadcasts state at ~4Hz. Clients receive playheadX and convert to local pixels.

File Reference

File	Role
oscillaTransport.js	scrollToPlayheadVisual() worldscreen conversion, applies transform
oscillaScoreSetup.js	extractScoreElements() extracts cue/rehearsal world positions
oscillaSystemRAF.js	Animation loop, drift correction
app.js	WebSocket sync handler
server.js	Broadcasts playheadX and scoreWidth to all clients
styles.css	SVG sizing (height: 100vh), container layout

Debugging

Quick Health Check

```
const svg = document.querySelector("#scoreInner svg");
const viewBox = svg.getAttribute("viewBox").split(" ").map(Number);

console.log("=== Sync Health Check ===");
console.log("ViewBox width:", viewBox[2]);
console.log("window.scoreWidth:", window.scoreWidth);
console.log("Match:", viewBox[2] === window.scoreWidth ? "☐" : "X");

console.log("\nRendered width:", svg.getBoundingClientRect().width);
console.log("localScale:", window.localScale);

// Expected width from height
const expectedWidth = window.innerHeight * (viewBox[2] / viewBox[3]);
const actualWidth = svg.getBoundingClientRect().width;
console.log("Expected width:", expectedWidth);
console.log("Actual width:", actualWidth);
console.log("Match:", Math.abs(expectedWidth - actualWidth) < 1 ? "☐" : "X");
```

Test Rehearsal Mark Alignment

```
function testRehearsalAlignment(markId) {
  const cueEl = document.getElementById('rehearsal_' + markId);
  const svg = document.querySelector("#scoreInner svg");
  const svgRect = svg.getBoundingClientRect();
  const cueRect = cueEl.getBoundingClientRect();

  // Get true world position
  const screenX = cueRect.x - svgRect.x;
  const actualWorldX = screenX / window.localScale;

  console.log(`=== Testing rehearsal_${markId} ===`);
  console.log("Actual world X:", actualWorldX);

  // Jump to it
  window.playheadX = actualWorldX;
  window.scrollToPlayheadVisual();

  // Verify alignment
  setTimeout(() => {
    const newCueRect = cueEl.getBoundingClientRect();
    const playheadEl = document.getElementById('playhead');
    const playheadRect = playheadEl.getBoundingClientRect();
    const diff = newCueRect.x - playheadRect.x;
    console.log("Alignment error (px):", diff);
    console.log("Aligned:", Math.abs(diff) < 5 ? "☐" : "X");
  }, 100);
}
```

// Usage: testRehearsalAlignment('A');

Common Issues

Symptom	Likely Cause	Check
Playhead lands behind marks, error increases with position	Wrong coordinate extraction method	Using <code>getBBBox()/getCTM()</code> instead of <code>getBoudingClientRect()</code>
Works on one device, broken on others	CSS affecting SVG sizing	Check SVG position is not absolute
Parent containers have 0 width	SVG removed from document flow	Check for position: absolute/fixed on SVG

Symptom	Likely Cause	Check
Score jumps/jitters during playback	Transform transition CSS	Remove any transition ON #scrollStage OR #score
scoreWidth doesnt match viewBox	Server caching old value	Send reset_project_state when loading new score

Interaction Layer Elements

Any element that can be **shared across clients** or **positioned on the score** must use world coordinates. This includes:

Elements That Need World Coordinates

Element	Properties	Notes
Markers	placement.x, labelY, fontSize	Drop markers, structural waypoints
Annotations	placement.x, placement.y, style.fontSize, extent	Text pins, triggers
Cues	x, width	Extracted from SVG
Rehearsal marks	x	Extracted from SVG

Conversion Pattern

When creating/editing interaction elements:

```
// CREATE: Convert click position to world coordinates
function getClickPlacement(evt, innerRect) {
  const localScale = window.localScale || 1;
  const screenX = evt.clientX - innerRect.left;
  const screenY = evt.clientY - innerRect.top;
  return {
    x: screenX / localScale, // Store world coords
    y: screenY / localScale
  };
}

// RENDER: Convert world coordinates to screen pixels
function positionElement(el, placement) {
  const localScale = window.localScale || 1;
  el.style.left = `${placement.x * localScale}px`;
  el.style.top = `${placement.y * localScale}px`;
}

// DRAG: Convert screen delta to world delta
function onDrag(dx, dy, baseWorldX, baseWorldY) {
  const localScale = window.localScale || 1;
  return {
    x: baseWorldX + (dx / localScale),
    y: baseWorldY + (dy / localScale)
  };
}

// FONT SIZE: Scale for consistent visual size across clients
function getScaledFontSize(baseFontSize) {
  const localScale = window.localScale || 1;
  return baseFontSize * localScale;
}
```

Why Font Size Needs Scaling

If a marker label is set to 24px on a large screen and synced to a small screen: - Without scaling: Label is 24px on both takes up more of the score on smaller screen - With scaling: Label is 24px localScale same proportion of score on all screens

The labels **edges** must align with the same score content on all clients.

Common Mistakes

Mistake	Symptom	Fix
Storing screen pixels directly	Elements appear in wrong position on other clients	Divide by <code>localScale</code> before storing
Not scaling font size	Text edges dont align across clients	Multiply by <code>localScale</code> at render
Forgetting to scale extent/width	Trigger duration bars wrong length	Apply same <code>worldscreen</code> conversion
Using <code>offsetWidth</code> directly	Width calculation wrong	Divide by <code>localScale</code> to get world width

Coordinate System Summary

WORLD SPACE (viewBox units)	SCREEN SPACE (pixels)
<code>scoreWidth = 40000</code>	<code>localRenderedWidth = 37539px</code>
<code>playheadX = 20000</code>	<code>screenX = 20000 × 0.938 = 18769px</code>
$\begin{aligned} \text{localScale} &= \text{localRenderedWidth} / \text{scoreWidth} \\ &= 37539 / 40000 \\ &= 0.938 \end{aligned}$	

World → Screen: `screenX = worldX × localScale`

Screen → World: `worldX = screenX / localScale`

Golden Rule: Always store and sync positions in **world coordinates**. Convert to screen coordinates only at render time.

Oscilla Developer Guide: Adding a New Cue Type

This guide walks through the complete process of adding a new cue type to Oscilla, using the `color()` / `colour()` animation cue as a case study.

Overview

Adding a new cue type requires modifications to four key files:

File	Purpose
<code>oscillaParser.js</code>	Token definition, grammar rules, AST extraction
<code>oscilla<CueName>.js</code>	The cue handler implementation
<code>oscillaCueDispatcher.js</code>	Playhead-triggered cue dispatch
<code>oscillaAnimation.js</code>	Load-time cue assignment (for animation cues)

The general flow is:

SVG element ID → Parser → AST → Dispatcher/Assignment → Handler

Step 1: Define the Token (Parser)

Tokens are the lexical building blocks. Add your cue keyword to `oscillaParser.js`.

Location

Near the top of the file, after other keyword tokens (around line 80-90).

Pattern

```
const Color = createToken({
  name: "Color",
  pattern: /\b(color|colour)\b/, // supports both spellings
  longer_alt: Identifier
});
```

Key Points

- Use `\b` word boundaries to prevent partial matches
- `longer_alt: Identifier` ensures the token doesn't consume longer identifiers
- For aliases (like `color/colour`), use regex alternation: `(color|colour)`

Add to Token Array

Find the `allTokens` array and add your token:

```
export const allTokens = [
  Cue, Fade, Page, Stopwatch, Video, Text, Pause, Stop,
  Audio, AudioPool, AudioImpulse, Synth, Button, Nav,
  Rotate, Scale, ScaleXY, O2P, Color, Ui, // ← Color added here
  // ...
];
```

Important: Token order matters. More specific tokens should come before `Identifier`.

Step 2: Add Parser Rule (Parser)

Parser rules define the grammar structure for your cue.

Location

Inside the `CueParser` class constructor, after similar cue rules (around line 860-870).

Pattern for Animation Cues

Animation cues (like `rotate`, `scale`, `color`) use a shared parameter list:

```
$.RULE("cueColorTop", () => {
  $.CONSUME(Color);
  $.SUBRULE($.animGenericParamList);
});
```

The `animGenericParamList` rule handles: - Parentheses `()` - Key-value pairs `dur:2, loop:0` - Arrays `vals:[1, 2, 3]` - Patterns `vals:Pseq([a, b], 3)` - Nested functions `prestate:fadein(500)`

Add to cueTop Alternatives

Find the `cueTop` rule and add your new rule as an alternative:

```
$.RULE("cueTop", () => {
  // ...
  $.OR([
    // ... existing alternatives ...
    { ALT: () => $.SUBRULE($.cueRotateTop) },
    { ALT: () => $.SUBRULE($.cueScaleTop) },
    { ALT: () => $.SUBRULE($.cueO2PTop) },
    { ALT: () => $.SUBRULE($.cueColorTop) }, // ← Add here
    { ALT: () => $.SUBRULE($.cueOscTop) },
    // ...
  ]);
});
```

Step 3: Add AST Extraction (Parser)

The CST (Concrete Syntax Tree) must be converted to an AST (Abstract Syntax Tree).

Location

In the `cstToAst()` function, near the end (around line 2480-2490).

Pattern

```
const colorNode = cst.children?.cueColorTop?.[0] ||
  (cst.name === "cueColorTop" ? cst : null);

if (colorNode) {
  return { type: "cueColor", args: extractAnimKvArgs(colorNode) };
}
```

AST Structure

The resulting AST looks like:

```
{
  type: "cueColor",
  args: [
    { key: "vals", value: ["#f00", "#0f0", "#00f"] },
    { key: "dur", value: 2 },
    { key: "mode", value: "alt" }
  ]
}
```

Step 4: Create the Handler Module

Create a new file `oscillaAnimationColor.js` (or `oscilla<CueName>.js`).

Module Structure

```
// oscillaAnimationColor.js

// =====
// IMPORTS
// =====
import { registerAnimation } from "../oscillaAnimation.js";
import { scheduleCueStart } from "../oscillaCueDispatcher.js";
import { createHitLabel } from "../oscillaHitLabels.js";
import {
  applyPrestateBeforeStart,
  applyPrestateOnStart,
  installOscToggleHandler,
  armGhostClickable,
  needsArming,
  needsFadeIn,
  triggerFadeIn,
  isOscEnabled
} from "../oscillaAnimationShared.js";
import { sendOSCMessages, createOscOverlay } from "../oscillaOSC.js";

// =====
// INTERNAL UTILITIES
// =====

// Pattern generator (reusable across animation types)
function makePatternGenerator(pattern) {
  // ... implementation
}

// Cue-specific utilities
function parseColorToHSL(colorStr) {
  // ... implementation
}

// =====
// ANIMATION ENGINES
// =====

// Different modes may need different engines
function handleColorHueCycle(el, cfg) {
  // Continuous hue cycling implementation
}

function handleColorSequence(el, cfg) {
  // Discrete color sequence implementation
}
```

```

}

// =====
// MAIN CUE HANDLER (exported)
// =====
export function handleColorCue(el, astArgs, options = {}) {
  // ... implementation
}

export default { handleColorCue };

```

Handler Function Anatomy

```

export function handleColorCue(el, astArgs, options = {}) {
  if (!el) return;

  const { fromCueTrigger = false } = options;

  // =====
  // 1. CHECK FOR RE-TRIGGER (ghostClickable, fadein)
  // =====
  const existingCfg = el._oscillaCfg;

  if (fromCueTrigger && existingCfg && existingCfg._ghostClickable) {
    if (needsArming(existingCfg)) {
      armGhostClickable(el, existingCfg);
    }
    return;
  }

  // =====
  // 2. PARSE COMMON PARAMETERS
  // =====
  let trig = "auto";
  let uid = el.id || ("color_" + Math.random().toString(36).slice(2));
  let cfgStartDelay = 0;
  let prestate = "show";

  for (const a of astArgs) {
    const key = a.key || a.type;
    const val = a.value;

    if (key === "trig") trig = String(val).toLowerCase();
    if (key === "uid") uid = String(val).trim();
    if (key === "tdelay") cfgStartDelay = Number(val) || 0;
    if (key === "prestate" && val !== null) prestate = val;
  }

  const shouldStartNow = fromCueTrigger || trig === "auto" || trig === "playhead";

  // =====
  // 3. BUILD BASE CONFIG
  // =====
  const baseCfg = {
    uid,
    trig,
    start: cfgStartDelay,
    prestate,
    astArgs,
    fromCueTrigger,
    kind: "color",
    _anim: null
  };

  // =====
  // 4. DETERMINE MODE & DISPATCH
  // =====
  const valsArg = astArgs.find(a => a.key === "vals" || a.type === "vals");

```

```

// Check for hue cycling mode
if (valsArg?.value === "hue" || valsArg?.value?.name === "hue") {
  const cfg = { ...baseCfg, mode: "hue-cycle" };
  setupAndStart(el, cfg, handleColorHueCycle, shouldStartNow);
  return;
}

// Check for pattern sequence
if (valsArg?.value?.type === "pattern") {
  const cfg = { ...baseCfg, pattern: valsArg.value, mode: "sequence-pattern" };
  setupAndStart(el, cfg, handleColorSequence, shouldStartNow);
  return;
}

// Array sequence
if (Array.isArray(valsArg?.value)) {
  const cfg = { ...baseCfg, values: valsArg.value, mode: "sequence" };
  setupAndStart(el, cfg, handleColorSequence, shouldStartNow);
  return;
}
}

// Helper to avoid repetition
function setupAndStart(el, cfg, engineFn, shouldStartNow) {
  el._oscillaCfg = cfg;

  installOscToggleHandler(el, cfg);
  applyPrestateBeforeStart(el, cfg);

  const rawStart = () => engineFn(el, cfg);
  const start = wrapStart(cfg, rawStart);

  registerAnimation(el, `color-${cfg.mode}`, cfg, start);

  createHitLabel(el, "color", cfg.uid, {
    anchorMode: "center",
    color: "magenta",
    sizeMode: "follow"
  });

  if (shouldStartNow) start();
}

```

Start Function Wrapper

The wrapStart pattern handles delayed starts and prestate transitions:

```

function wrapStart(cfg, rawStartFn) {
  cfg._start = rawStartFn;
  cfg._applyPrestateOnStart = () => applyPrestateOnStart(el, cfg);

  return () => {
    if (cfg.start > 0) {
      scheduleCueStart(
        cfg,
        el,
        () => {
          if (cfg._ghostClickable && cfg._startBlocked) {
            applyPrestateOnStart(el, cfg);
            return;
          }
          applyPrestateOnStart(el, cfg);
          rawStartFn();
        },
        cfg.uid
      );
    } else {
      if (cfg._ghostClickable && cfg._startBlocked) {
        applyPrestateOnStart(el, cfg);
      }
    }
  };
}

```

```

        return;
    }
    applyPrestateOnStart(el, cfg);
    rawStartFn();
}
};
};
}

```

Step 5: Register in Dispatcher

Add your handler to `oscillaCueDispatcher.js` for playhead-triggered execution.

Add Import

```

import { handleRotateCue } from "./oscillaAnimationRotate.js";
import { handleScaleCue } from "./oscillaAnimationScale.js";
import { handleO2PCue } from "./oscillaAnimationO2p.js";
import { handleColorCue } from "./oscillaAnimationColor.js"; // ← Add

```

Add Case in Switch

Find the cue dispatch switch statement (around line 290-400):

```

case "cueO2P": {
    handleO2PCue(cueElement, ast.args, { fromCueTrigger: true });
    triggerNestedCues(ast, cueElement);
    return;
}

case "cueColor": { // ← Add this case
    handleColorCue(cueElement, ast.args, { fromCueTrigger: true });
    triggerNestedCues(ast, cueElement);
    return;
}

case "cueOsc": {
    // ...
}

```

Key Points: - Pass `{ fromCueTrigger: true }` to indicate playhead activation - Call `triggerNestedCues()` if your cue can contain nested cues

Step 6: Register in Animation Assignment

For animation cues that should activate on page load (auto-start), add to `oscillaAnimation.js`.

Add Import

```

import { handleScaleCue } from "./oscillaAnimationScale.js";
import { handleRotateCue } from "./oscillaAnimationRotate.js";
import { handleO2PCue } from "./oscillaAnimationO2p.js";
import { handleColorCue } from "./oscillaAnimationColor.js"; // ← Add

```

Add Case in animationAssign

```

switch (ast.type) {
    case "cueScale":
        handleScaleCue(ast, el);
        break;

    case "cueRotate":
        handleRotateCue(el, ast.args);
        break;

    case "cueO2P":
        handleO2PCue(el, ast.args);
        break;

    case "cueColor": // ← Add this case
        handleColorCue(el, ast.args);
        break;

    // ...
}

```

Common Parameters

Most animation cues support these shared parameters:

Parameter	Type	Description
uid	string	Unique identifier for the animation
trig	auto playhead	Trigger mode
tdelay	number	Delay before start (seconds)
prestate	show hide ghost fadein(ms) ghostClickable(...)	Initial visibility state
dur	number pattern	Duration per cycle
loop	number	Repeat count (0 = infinite)
osc	0 1 2	OSC output mode
oscaddr	string	Custom OSC address

Pattern Support

Oscilla patterns work across all animation cues:

```
// Sequential
vals:Pseq([a, b, c], 3)    // Play a,b,c three times

// Random (with repeats)
vals:Prand([a, b, c], inf) // Random selection, may repeat

// Random (no repeats)
vals:Pxrnd([a, b, c], inf) // Random, never same twice in a row

// Shuffled
vals:Pshuf([a, b, c], 2)   // Shuffle, play through twice
```

Pattern Generator Template

```
function makePatternGenerator(pattern) {
  if (!pattern?.values || !Array.isArray(pattern.values)) {
    return { next: () => null };
  }

  const values = pattern.values.slice();
  let repeats = pattern.repeats ?? Infinity;
  let index = 0;
  let cycleCount = 0;

  switch (pattern.name) {
    case "Pseq":
      return {
        next() {
          const v = values[index++];
          if (index >= values.length) {
            index = 0;
            if (++cycleCount >= repeats) return null;
          }
          return v;
        }
      };
    // ... other patterns
  }
}
```

OSC Integration

To add OSC output to your cue:

1. Check if OSC is Enabled

```
import { isOscEnabled } from "../oscillaAnimationShared.js";

if (isOscEnabled(cfg, oscMode)) {
  // Send OSC
}
```

2. Send OSC Message

```
import { sendOSCMessage } from "../oscillaOSC.js";

function sendOSCColor(cfg, hsl) {
  const payload = {
    type: "osc_color",
    uid: cfg.uid,
    h: hsl.h,
    s: hsl.s,
    l: hsl.l,
    timestamp: Date.now()
  };

  if (cfg.oscAddr) payload.addr = cfg.oscAddr;

  sendOSCMessage(payload);
}
```

3. Add Visual Overlay (Optional)

```
import { createOscOverlay } from "../oscillaOSC.js";

if (isOscEnabled(cfg, oscMode)) {
  cfg._overlay = createOscOverlay({
    anchorEl: el,
    label: cfg.oscAddr || cfg.uid,
    anchorMode: "center",
    mode: "auto"
  });
  cfg._overlay.update("initial value");
}
```

Hit Labels

Hit labels provide visual indicators and click targets:

```
import { createHitLabel } from "../oscillaHitLabels.js";

createHitLabel(el, "color", cfg.uid, {
  anchorMode: "center",      // center | pathStart | pathEnd
  color: "magenta",          // CSS color
  sizeMode: "follow"         // follow | fixed | rotate40
});
```

Testing Your Cue

1. Parser Test

```
import { parseCueToAST } from "../oscillaParser.js";

const ast = parseCueToAST("color(vals:[#f00,#0f0],dur:2,mode:alt)");
console.log(JSON.stringify(ast, null, 2));
```

Expected output:

```
{
  "type": "cueColor",
  "args": [
    { "key": "vals", "value": ["#f00", "#0f0"] },
    { "key": "dur", "value": 2 },
    { "key": "mode", "value": "alt" }
  ]
}
```

2. SVG Test

Create an SVG element with your cue ID:

```
<rect id="color(vals:[#f00,#0f0,#00f],dur:2)" width="100" height="100" fill="#f00"/>
```

3. Console Debug

```
// Check animation registry
console.log(window.oscillaAnimRegistry);

// Check running animations
console.log([...window.runningAnimations.entries()]);
```

Checklist

- Token added to `oscillaParser.js`
 - Token added to `allTokens` array
 - Parser rule created (`cue<Name>Top`)
 - Rule added to `cueTop` alternatives
 - AST extraction added to `cstToAst()`
 - Handler module created (`oscilla<Name>.js`)
 - Handler imported in `oscillaCueDispatcher.js`
 - Case added to dispatcher switch
 - Handler imported in `oscillaAnimation.js` (if animation cue)
 - Case added to `animationAssign` switch (if animation cue)
 - Hit labels configured
 - OSC output implemented (if applicable)
 - Documentation written
-

File Locations Summary

```
public/js/
├─ oscillaParser.js           # Token + grammar + AST
├─ oscillaCueDispatcher.js    # Playhead dispatch
├─ oscillaAnimation.js        # Load-time assignment
├─ oscillaAnimationShared.js  # Shared utilities
├─ oscillaAnimationColor.js   # Your new handler
├─ oscillaHitLabels.js        # Hit label system
└─ oscillaOSC.js              # OSC messaging
```

Example: Complete `color()` Implementation

DSL Syntax

```
color(vals:[#f00,#0f0,#00f], dur:2)
color(vals:Pseq([red,yellow,green],3), dur:0.5)
color(vals:hue, dur:10, loop:0)
color(vals:hue(120,240), dur:4, osc:1)
colour(vals:[#f80,#08f], mode:alt, dur:1.2)
```

Files Modified

1. `oscillaParser.js` - Token, rule, AST
2. `oscillaAnimationColor.js` - New file
3. `oscillaCueDispatcher.js` - Import + case
4. `oscillaAnimation.js` - Import + case

Total Lines Added

- Parser: ~15 lines
- Handler: ~600 lines

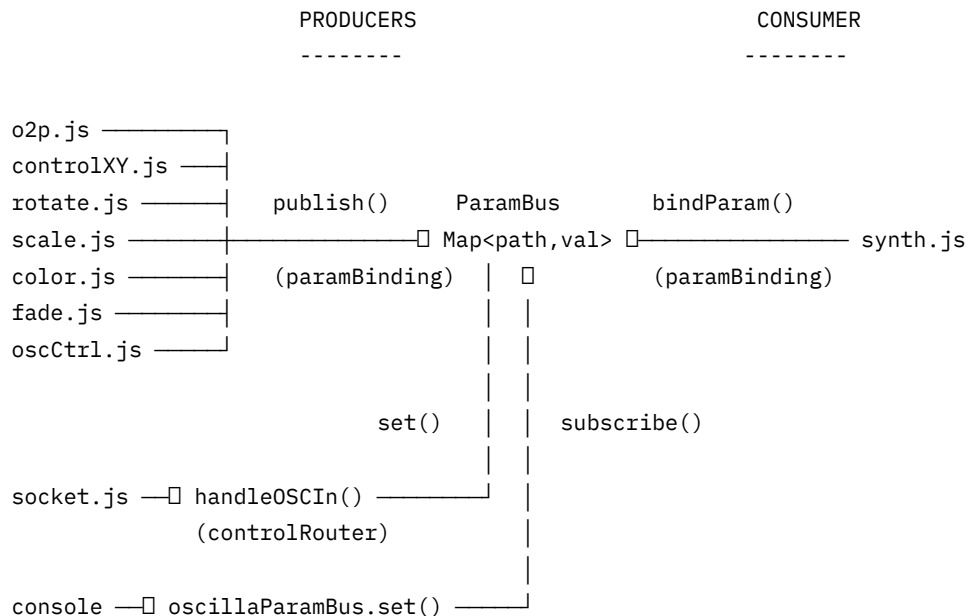
- Dispatcher: ~6 lines
- Animation: ~5 lines

Document version: 1.0 Last updated: January 2026 Case study: color() / colour() animation cue

Control Plane & Cross-Cue Modulation Developer Guide

This document explains how Oscillas internal signal routing works, how the pieces connect, and how to extend it. It is written for developers contributing to the codebase, not for end-user score authoring (see `control-and-modulation.md` for the composer-facing documentation).

Architecture at a Glance



Every animation and control cue publishes its current values into the `ParamBus`. Any cue that accepts signal bindings subscribes to the `ParamBus` via `bindParam()`. The `ParamBus` is the single meeting point producers and consumers never reference each other directly.

File Map

All control-plane code lives under `js/control/`. Cue handlers that publish signals live under `js/cues/`. The parser extension that detects signal references lives under `js/parser/`.

control/

<code>paramBus.js</code>	Signal store with pub/sub (the core)
<code>paramBinding.js</code>	<code>publish()</code> and <code>bindParam()</code> helpers
<code>controlRouter.js</code>	OSC-in routing, explicit modulation API
<code>controlIntegration.js</code>	(dead code -- never imported, kept as reference)
<code>controlShared.js</code>	Shared persistence for launcher/preset state
<code>easing.js</code>	Easing functions for tween interpolation
<code>rotationMath.js</code>	Rotation math utilities

parser/

<code>parserSignalRef.js</code>	Detects <code>"uid.channel[min,max]"</code> in DSL values
---------------------------------	---

cues/

<code>o2p.js</code>	Publishes: t, x, y, angle
<code>controlXY.js</code>	Publishes: x, y, p (per handle)
<code>rotate.js</code>	Publishes: angle, rad, norm
<code>scale.js</code>	Publishes: sx, sy, uniform

color.js	Publishes: hNorm, sNorm, lNorm
fade.js	Publishes: opacity
oscCtrl.js	Publishes: t, v
synth.js	Consumes via bindParam()

system/

oscillaOSCClient.js	OSC send/receive, routes incoming to controlRouter
oscillaTargets.js	Target registry (for future OSC-in direct control)
socket.js	WebSocket handler, calls handleOSCIn on messages

The Signal Pipeline in Detail

There are three independent data paths. They all converge on the ParamBus but serve different purposes.

Path 1: Internal publish/subscribe (the main one)

This is how a fader controls a synth.

Producer side (in a cue handlers animation loop):

```
import { publish } from '../control/paramBinding.js';
```

```
// Inside the tick/update callback:
```

```
publish("o2p", cfg.uid, {
  t: pathT,      // 0-1 position along path
  x: normX,      // 0-1 horizontal in bbox
  y: normY,      // 0-1 vertical in bbox
  angle: angle   // degrees, tangent direction
});
```

publish() does two things for each channel:

1. Writes a **typed path**: o2p:myFader.t
2. Writes a **source-agnostic path**: myFader.t

Both are stored in the same ParamBus Map, both notify subscribers. The typed path exists for debugging and the explicit modulation API. The agnostic path is what bindParam() subscribes to.

Publishing is throttled per source+uid at ~60fps to avoid flooding the bus. The throttle interval is PUBLISH_THROTTLE_MS (default 16ms) in paramBinding.js.

Consumer side (in synth.js or any cue that accepts bindings):

```
import { bindParam } from '../control/paramBinding.js';
```

```
const freqBinding = bindParam(
  params.freq, // may be a number OR a signalRef
  (hz) => osc.frequency.value = hz, // callback on each update
  { min: 20, max: 20000, default: 440 }
);
```

```
// Later, on cleanup:
```

```
freqBinding.unbind();
```

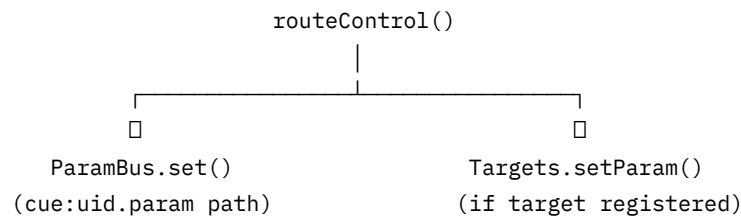
If params.freq is a plain number (e.g.440), bindParam returns immediately with { value: 440, unbind: noop, isBinding: false }.

If params.freq is a signalRef object (e.g. { type: "signalRef", source: "fSlider", channel: "t", range: [200, 800] }), bindParam subscribes to the agnostic path fSlider.t and calls the update callback whenever the value changes, mapping the raw 0-1 input through mapRange(raw, 0, 1, 200, 800).

Path 2: OSC-in (external control)

External software (SuperCollider, Max, Pd) sends OSC to the Oscilla server, which forwards it over WebSocket to all clients.

```
External OSC —□ server.js —□ WebSocket —□ socket.js
                                     |
                                     handleOSCIn()
                                     (controlRouter.js)
                                     |
```



The OSC address format is:

```
/oscilla/set <uid> <param> <value>
```

or the path-based form:

```
/oscilla/<uid>/<param> <value>
```

socket.js (line ~381) dispatches incoming messages to `handleOSCIn()` in `controlRouter.js`, which calls `routeControl()`. The router writes to the `ParamBus` at `cue:uid.param` and also forwards to any registered target.

Note: the router does NOT send OSC back out. OSC output is handled independently by each cue handlers own `sendOSC()` calls. This prevents feedback loops between incoming and outgoing OSC.

Path 3: Explicit modulation API (console / programmatic)

The `controlRouter` exposes `addModulation()` for wiring arbitrary signal-to-target connections at runtime, with optional scaling, offset, clamping, and smoothing.

```
// In browser console:
window.oscillaRouter.mod(
  "o2p:slider.t",      // source signal (typed path)
  "drone.freq",        // target uid.param
  { scale: 1800, offset: 200, min: 200, max: 2000 }
);
```

This subscribes to the `ParamBus` at the source path and calls `routeControl()` for the target on every update. It is independent of the DSL binding mechanism and is primarily useful for live performance patching or scripted compositions.

`listModulations()`, `removeModulation()`, and `clearModulations()` manage the active modulation set.

How the Parser Creates Signal References

When the parser encounters a synth cue, it extracts parameter values with `signalRefAware: true` (parser.js line ~1297). For each parameter value that is a string, it calls `maybeConvertToSignalRef()` from `parserSignalRef.js`.

The function checks: 1. Is the parameter name in the `BINDABLE_PARAMS` set? (freq, amp, pan, cutoff, q, etc.) 2. Does the string match the pattern `identifier.identifier[num,num]`?

If both are true, it returns a `signalRef` object instead of the raw string:

DSL input: freq:fader1.t[200,800]

Parser output:

```
{
  type: "signalRef",
  source: "fader1",
  channel: "t",
  range: [200, 800]
}
```

This object flows through the AST into `synth.js`, where `bindParam()` recognises it via `isSignalRef()` and sets up the subscription.

Dual-Path Addressing

Every `publish()` call writes two `ParamBus` entries:

Path	Example	Purpose
Typed	<code>o2p:fader1.t</code>	Debugging, explicit modulation
Source-agnostic	<code>fader1.t</code>	What <code>bindParam</code> subscribes to

The agnostic path exists because the composer should not need to know whether a control source is implemented as `o2p`, `controlXY`, `rotate`, or anything else. They just write `freq:fader1.t[200,800]` and it works.

The typed path survives for: - `window.oscillaParamBus.snapshot("o2p:")` to list all `o2p` signals - `window.oscillaRouter.mod("rotate:spin.norm", ...)` for explicit wiring - Debug logging which shows the source type

Both paths hold the same value at all times.

Published Channels by Source

Each cue type publishes a specific set of channels. All values are numeric. Most are normalised to 0-1 unless noted.

o2p (path-following animation)

Channel	Range	Description
<code>t</code>	0-1	Position along path
<code>x</code>	0-1	Normalised X within path bbox
<code>y</code>	0-1	Normalised Y within path bbox
<code>angle</code>	0-360 deg	Tangent direction at current pos

Published in: `o2p.js` lines ~555, ~684, ~1202, ~1398.

controlXY (touch/drag pad)

Channel	Range	Description
<code>x</code>	0-1	Horizontal position (left=0)
<code>y</code>	0-1	Vertical position (bottom=0)
<code>p</code>	0-1	Rotation handle (if <code>hmode</code> present)
<code>{handle}.x</code>	0-1	Per-handle X (multi-handle pads)
<code>{handle}.y</code>	0-1	Per-handle Y
<code>{handle}.p</code>	0-1	Per-handle rotation

Published in: `controlXY.js` line ~481.

rotate (CSS/transform rotation)

Channel	Range	Description
<code>angle</code>	0-360 deg	Current angle
<code>rad</code>	0-2pi	Angle in radians
<code>norm</code>	0-1	Normalised (angle / 360)

Published in: `rotate.js` lines ~465, ~583.

scale (CSS/transform scaling)

Channel	Range	Description
<code>sx</code>	varies	Scale factor on X

Channel	Range	Description
sy	varies	Scale factor on Y
uniform	varies	Average of sx and sy

Published in: `scale.js` line ~618.

Note: scale values are raw factors (e.g. 0.5 to 2.0), not normalised to 0-1. Binding these directly with a `[min,max]` range requires awareness that `mapRange` assumes 0-1 input. A normalised channel could be added in future.

color (HSL color animation)

Channel	Range	Description
hNorm	0-1	Hue / 360
sNorm	0-1	Saturation / 100
lNorm	0-1	Lightness / 100

Published in: `color.js` lines ~426, ~610.

fade (opacity animation)

Channel	Range	Description
opacity	0-1	Current opacity

Published in: `fade.js` lines ~173, ~177, ~256.

oscCtrl (playhead-driven control lane)

Channel	Range	Description
t	0-1	Playhead position within lane
v	0-1	Normalised Y value from path

Published in: `oscCtrl.js` line ~175.

Bindable Parameters (Consumer Side)

Only synth cues currently accept signal references through the DSL. The `BINDABLE_PARAMS` set in `parserSignalRef.js` defines which parameter names the parser will check:

`freq`, `frequency`, `amp`, `amplitude`, `gain`, `pan`, `cutoff`, `filterFreq`,
`q`, `resonance`, `detune`, `pitch`, `playbackRate`, `rate`, `delayTime`, `delay`,
`feedback`, `fb`, `mix`, `wet`, `dry`, `reverbMix`, `reverbTime`, `speed`, `dur`, `duration`

`synth.js` wraps each of these with `bindParam()`, so if the parser delivers a `signalRef`, the subscription activates. If it delivers a plain number, `bindParam` returns a static value and no subscription is created.

How to Add a New Publisher

If you are writing a new cue type that produces values (e.g. a new animation or sensor input), follow this pattern:

1. Import publish

```
import { publish } from '../control/paramBinding.js';
```

2. Call publish in your animation loop

```
// Inside your tick/update/rAF callback:
publish("myNewType", cfg.uid, {
  value1: normalisedValue, // 0-1 preferred
  value2: anotherValue
});
```

Use 0-1 normalised values where possible. The `[min,max]` range syntax in the DSL assumes 0-1 input.

3. Done

No registration, no setup, no teardown. The ParamBus handles everything. When your cue stops and stops calling `publish()`, the last value remains in the bus (stale but harmless). Subscribers simply stop receiving updates.

Document the channel names and their ranges so composers know what to bind to.

How to Add a New Consumer

If you are writing a cue type that should accept signal-driven parameters (like synth does):

1. Import `bindParam` and `isSignalRef`

```
import { bindParam, isSignalRef } from '../control/paramBinding.js';
```

2. Wrap parameter initialisation

```
const unbinders = [];

const speedBinding = bindParam(
  params.speed, // from parser, may be signalRef
  (val) => { cfg.animSpeed = val; }, // live update callback
  { min: 0.1, max: 10, default: 1 }
);
unbinders.push(speedBinding.unbind);

// Use speedBinding.value as the initial value
```

3. Clean up on stop

```
function stop() {
  unbinders.forEach(fn => fn());
}
```

4. Enable signal refs in the parser

In `parser.js`, find the AST extraction for your cue type and pass `{ signalRefAware: true }`:

```
return {
  type: "cueMyType",
  args: extractGenericArgs(getGenericParams(node), { signalRefAware: true })
};
```

Currently only synth cues have this enabled (line ~1297). Extending it to other cue types is a matter of adding the flag.

OSC-In Target Registration (Not Yet Active)

The `controlRouter` has a second routing path: when `routeControl()` is called (e.g. from OSC-in), it also tries `Targets.get(uid)` and calls `setParam()` on the registered target. This would allow external OSC to directly set parameters on running cue instances.

However, **no cue handler currently registers as a target**. The `controlIntegration.js` file contains `registerO2Ptarget()`, `registerRotateTarget()` etc., but nothing imports it.

To activate this path in future: 1. Import the register function in the cue handler 2. Call it when the cue starts, passing a `setParam` implementation 3. Call `unregister()` when the cue stops

This is a separate feature from the publish/subscribe mechanism and is not required for cross-cue modulation to work.

Debugging

Enable ParamBus logging

```
window.oscillaParamBus.setDebugMode(true);
```

Every `set()` call that changes a value will be logged with the path and value.

Inspect current signals

```
window.oscillaParamBus.snapshot() // all signals
window.oscillaParamBus.snapshot("o2p:") // only o2p typed paths
window.oscillaParamBus.get("fader1.t") // specific agnostic path
```

Watch a signal

```
const unsub = window.oscillaParamBus.subscribe("fader1.t", (val, path) => {
  console.log(`${path} = ${val}`);
});

// Later: unsub()
```

List active modulations

```
window.oscillaRouter.listModulations()
```

Simulate a signal (no SVG needed)

```
let t = 0;
const sim = setInterval(() => {
  t = (t + 0.01) % 1;
  window.oscillaParamBus.set("testFader.t", t);
}, 30);
// clearInterval(sim) to stop
```

Check if a binding is active

In synth.js, bindParam() logs to console when a subscription is created: [bindParam] Bound to fader1.t -> range [200, 800]. If you don't see this log, the parser didn't produce a signalRef check that the parameter name is in BINDABLE_PARAMS and that the value string matches the uid.channel or uid.channel[min,max] pattern.

Known Limitations

Scale and rotate raw ranges. The scale channels publish raw factors (0.5, 1.0, 2.0 etc.), not 0-1 normalised values. The mapRange function in bindParam assumes 0-1 input. Binding amp:scaler.sx[0,0.3] when sx ranges from 0.5 to 2.0 will produce unexpected mappings. Use rotate:norm (which is 0-1) for rotation bindings. For scale, a normalised channel should be added.

Only synth accepts signal refs in DSL. The parser's signalRefAware flag is only enabled for synth cues. Other cue types (rotate, scale, fade, audio) will silently treat speed:fader.t as a literal string. Extending this requires the flag in the parser plus bindParam() calls in the cue handler.

No input range specification. The [min,max] syntax specifies the output range only. Input is assumed 0-1. There is no syntax for freq:spinner.angle[0,360][200,2000] (input 0-360, output 200-2000). For non-normalised sources, use the norm channel where available or add normalisation in the publisher.

Stale signals. When a cue stops, its last published values remain in the ParamBus. This is harmless for subscribers (they just stop receiving updates) but can be confusing when inspecting state via snapshot(). A future cleanup hook could clear signals on cue stop.

controlIntegration.js is dead code. It is not imported anywhere. It was written as a convenience layer over the router and targets but the cue handlers use publish() and bindParam() directly instead. It can be deleted or kept as reference for the target registration pattern.

No feedback prevention. The system does not detect or prevent circular signal routing. If cue A modulates cue B and cue B modulates cue A, both will update continuously. This is by design feedback is an intentional compositional tool but accidental loops can cause CPU spikes.

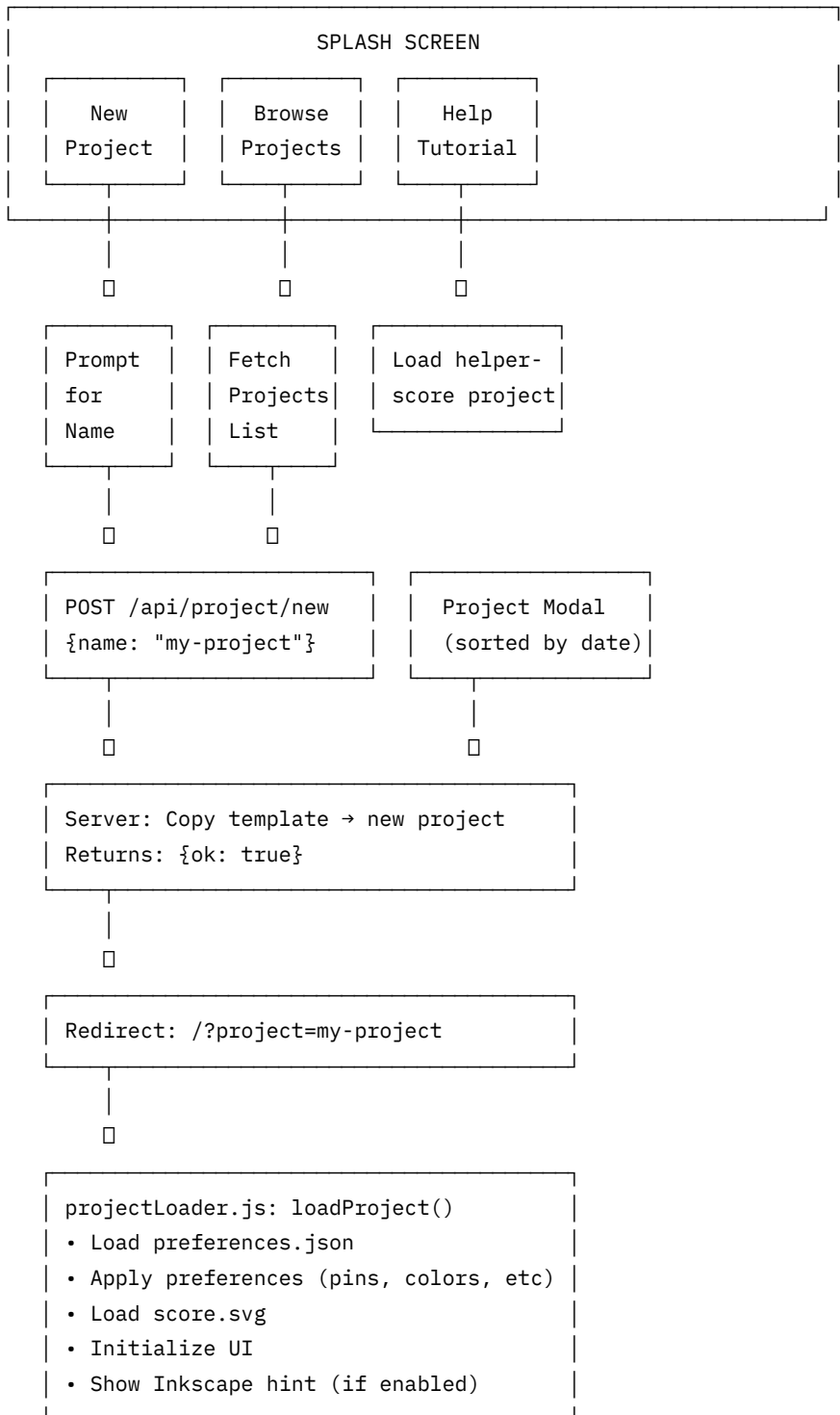
New Project Flow - Developer Documentation**Overview**

This document explains how Oscillas new project creation flow works, from the splash screen through project initialization. It covers the interaction between client-side UI, server API, and the project loading system.

Table of Contents

1. [Architecture Overview](#)
2. [Splash Screen System](#)
3. [New Project Creation Flow](#)
4. [Project Browser Flow](#)
5. [Preferences System](#)
6. [Pin State Management](#)
7. [File Locations](#)
8. [API Endpoints](#)

Architecture Overview



Splash Screen System

Location

- **HTML:** public/index.html (lines 22-59)
- **CSS:** public/css/oscillaSplash.css
- **JavaScript:** public/js/projectLoader.js (wireSplashActions)

Visual Design

Modal Behavior: - Dark semi-transparent backdrop: `rgba(40, 40, 40, 0.95)` - Blur effect: `backdrop-filter: blur(6px)` - Blocks all interaction with background (`pointer-events: all`) - White container with prominent shadow for visual separation

Three Action Buttons:

```

<!-- Order: New → Browse → Help -->
<button id="new-project-btn">
  <svg><!-- Plus icon --></svg>
  <span>New</span>
</button>

<button id="browse-projects-btn">
  <svg><!-- Folder icon --></svg>
  <span>Browse</span>
</button>

<button id="open-tutorial-btn">
  <svg><!-- Help icon --></svg>
  <span>Help</span>
</button>

```

When Splash Appears

The splash screen appears when: 1. User visits Oscilla without a `?project=` URL parameter 2. No project has been loaded yet in the session

Code location: `projectLoader.js` (bottom of file)

```

const projectFromURL = urlParams.get("project");
if (projectFromURL) {
  loadProject(projectFromURL);
} else {
  window.showSplashScreen();
}

```

Hiding the Splash

The splash is hidden when a project loads:

```

// In loadProject() function
showLoader(projectName); // This hides splash via z-index layering

// Later, after load completes:
hideLoader();
setSplashVisibility(false);

```

New Project Creation Flow

Step-by-Step Process

1. User Clicks “New” Button

File: `projectLoader.js` `wireSplashActions()`

```

newProjectBtn.onclick = async () => {
  console.log("[Splash] Creating new project");

  // Step 1: Prompt for name
  const name = prompt("New project name:");
  if (!name) return;

  try {
    // Step 2: Call server API
    const res = await fetch("/api/project/new", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ name })
    });

    const data = await res.json();
    if (!data.ok) {
      alert(data.error);
      return;
    }
  }
}

```

```

// Step 3: Set hint for post-load guidance
sessionStorage.setItem("oscilla.showInkscapeHint", name);

// Step 4: Navigate to new project
window.location.href = `/?project=${encodeURIComponent(name)}`;

} catch (err) {
  console.error("[Splash] Failed to create project:", err);
  alert("Failed to create project. Please try again.");
}
};

```

2. Server Creates Project

File: server.js /api/project/new endpoint

The server: 1. Receives project name 2. Validates name (no slashes, dots, etc.) 3. Checks if project already exists 4. Copies template directory to new project 5. Returns success/error response

Template Location:

```

public/scores/template/
├─ score.svg           # Blank SVG template
├─ preferences.json    # Default preferences
├─ audio/              # Empty audio directory
├─ pages/              # Empty pages directory
└─ videos/            # Empty videos directory

```

3. Browser Redirects

After successful creation, the browser navigates to:

`/?project=my-new-project`

This triggers the project loading flow.

4. Project Loads

File: projectLoader.js loadProject(projectName)

```

async function loadProject(projectName, options = {}) {
  // 1. Show loader (hides splash)
  showLoader(projectName);

  // 2. Set paths
  window.projectBase = `scores/${projectName}/`;
  window.currentProject = projectName;

  // 3. Load preferences
  const prefs = await loadPreferences(window.projectBase);
  applyPreferences(prefs);

  // 4. Load score.svg
  const svgUrl = `${window.projectBase}score.svg`;
  // ... load and initialize SVG

  // 5. Hide loader
  hideLoader();

  // 6. Show Inkscape hint if new project
  const hintProject = sessionStorage.getItem("oscilla.showInkscapeHint");
  if (hintProject === projectName) {
    sessionStorage.removeItem("oscilla.showInkscapeHint");
    showInkscapeHint(projectName);
  }
}

```

5. Inkscape Hint Appears

File: projectScoreSetup.js showInkscapeHint()

A modal dialog appears explaining: - How to edit the score in Inkscape - Where the score.svg file is located - Next steps for the user

The user can dismiss this dialog, and theres a checkbox to “Dont show again” which stores the preference in localStorage.

Project Browser Flow

When User Clicks “Browse”

1. Fetch Projects from Server

Client: projectLoader.js browseBtn.onclick

```
browseBtn.onclick = async () => {
  const projects = await fetchProjects();
  openProjectModal(projects);
};
```

API Call:

```
async function fetchProjects() {
  const res = await fetch("/api/projects");
  if (!res.ok) throw new Error("Failed to fetch projects");
  return res.json();
}
```

2. Server Returns Project List with Metadata

Server: server.js /api/projects endpoint

Returns JSON array:

```
[
  {
    "name": "my-project",
    "modified": "2025-02-02T10:30:00.000Z"
  },
  {
    "name": "another-project",
    "modified": "2025-02-01T15:20:00.000Z"
  }
]
```

Server Implementation:

```
app.get("/api/projects", (req, res) => {
  const scoresDir = path.join(WRITE_DIR, "public", "scores");

  fs.readdir(scoresDir, { withFileTypes: true }, (err, entries) => {
    if (err) return res.status(500).json([]);

    const projects = entries
      .filter(e => e.isDirectory())
      .filter(e => !e.name.startsWith("."))
      .map(e => {
        const projectPath = path.join(scoresDir, e.name);
        const stats = fs.statSync(projectPath);
        return {
          name: e.name,
          modified: stats.mtime.toISOString()
        };
      });

    res.json(projects);
  });
});
```

3. Display Modal with Sorted List

Client: projectLoader.js openProjectModal()

```
async function openProjectModal(projects) {
  const modal = document.getElementById("project-modal");
  const list = document.getElementById("project-list");

  // Normalize data
  projects = projects.map(p =>
    typeof p === 'string' ? { name: p } : p
  );
```

```

// Sort by date (newest first)
projects.sort((a, b) => {
  if (a.modified && b.modified) {
    return new Date(b.modified) - new Date(a.modified);
  }
  return (a.name || '').localeCompare(b.name || '');
});

// Create list items
projects.forEach(project => {
  list.appendChild(makeProjectListItem(project));
});

modal.classList.remove("hidden");
}

```

4. Format Timestamps

Client: projectLoader.js makeProjectListItem()

Timestamps are formatted as: - “today” - modified today - “yesterday” - modified yesterday - “3d ago” - modified 3 days ago - “Jan 15” - modified earlier (with year if different)

```

function makeProjectListItem(project) {
  const item = document.createElement("div");
  item.className = "project-item";

  // Format timestamp
  if (project.modified) {
    const date = new Date(project.modified);
    const now = new Date();
    const diffDays = Math.floor((now - date) / (1000 * 60 * 60 * 24));

    if (diffDays === 0) meta.textContent = "today";
    else if (diffDays === 1) meta.textContent = "yesterday";
    else if (diffDays < 7) meta.textContent = `${diffDays}d ago`;
    else meta.textContent = date.toLocaleDateString(/* ... */);
  }

  // Click handler
  item.onclick = () => {
    window.loadProject(project.name, { resetOnLoad: true });
    closeProjectModal();
  };

  return item;
}

```

Preferences System

File Structure

Each project has a preferences.json file:

```

public/scores/my-project/
├── preferences.json

```

Default Preferences

Location: projectLoader.js loadPreferences()

```

{
  "darkMode": false,
  "defaultPlaybackSpeed": 1.0,
  "defaultViewMode": "scroll",
  "pinControls": true,
  "pinTopbar": true
}

```

Loading Preferences

When a project loads:

```

async function loadPreferences(basePath) {
  const prefsPath = `${basePath}preferences.json`;
  try {
    const res = await fetch(prefsPath);
    if (!res.ok) throw new Error("No preferences.json found");
    return await res.json();
  } catch (err) {
    // Return defaults
    return {
      darkMode: false,
      defaultPlaybackSpeed: 1.0,
      defaultViewMode: "scroll",
      pinControls: true,
      pinTopbar: true,
    };
  }
}

```

Applying Preferences

Location: projectLoader.js applyPreferences(prefs)

```

window.applyPreferences = function applyPreferences(prefs) {
  // 1. Dark mode
  applyDarkMode?.(!prefs.darkMode);

  // 2. Duration
  if (prefs.duration_minutes > 0) {
    window.duration = prefs.duration_minutes * 60 * 1000;
  }

  // 3. Pin states (NEW)
  if (prefs.pinControls !== undefined) {
    window.oscillaControlsPinned = prefs.pinControls;
  } else {
    window.oscillaControlsPinned = true; // Default pinned
  }

  if (prefs.pinTopbar !== undefined) {
    window.oscillaTopbarPinned = prefs.pinTopbar;
  } else {
    window.oscillaTopbarPinned = true; // Default pinned
  }

  // 4. Visual preferences
  const playhead = document.getElementById("playhead");
  if (playhead && prefs.playheadColor) {
    playhead.style.backgroundColor = prefs.playheadColor;
  }
  // ... etc
};

```

User-Facing Preferences Dialog

Location: oscillaPreferences.js openPreferencesDialog()

Users can access via: **Menu Preferences**

The dialog includes: - Project metadata (title, author, description) - Visual preferences (colors, dark mode) - **Pin Controls** checkbox (default: true) - **Pin Top Bar** checkbox (default: true) - Playback settings - Touch/interaction settings

When saved, preferences are POSTd to:

/save-preferences/:projectName

Server writes to preferences.json in the project directory.

Pin State Management

How Pinning Works

Pin State Variables:

```
// Set from preferences (default: true)
window.oscillaControlsPinned = true;
window.oscillaTopbarPinned = true;

// Used by transport system
window.controlsPinned = window.oscillaControlsPinned ?? true;
window.topbarPinned = window.oscillaTopbarPinned ?? true;
```

Initialization Flow

1. Preferences Load First

File: projectLoader.js loadProject()
// Load preferences early in project load
 const prefs = await loadPreferences(window.projectBase);
 applyPreferences(prefs); *// Sets window.oscillaControlsPinned/Topbar*

2. Transport Reads Preference

File: oscillaTransport.js (top-level)
// Read from preference, default to true
 window.controlsPinned = window.oscillaControlsPinned ?? true;
 window.topbarPinned = window.oscillaTopbarPinned ?? true;

3. UI Initializes

File: oscillaTransport.js initializeControlsPin()
 export function initializeControlsPin() {
 const pinButton = document.getElementById("pin-controls");

// Set initial visual state based on preference
 pinButton.classList.toggle("active", window.controlsPinned);

// Wire toggle handler
 pinButton.addEventListener("click", () => {
 window.controlsPinned = !window.controlsPinned;
 pinButton.classList.toggle("active", window.controlsPinned);

 if (window.controlsPinned) {
 controls?.classList.remove('dismissed');
 } else {
 window.hideControlsLater();
 }
 });
 }

Visual Feedback

CSS:

```
/* Pin button shows "active" state when pinned */
.gui-button.active {
  background: rgba(255, 255, 255, 0.2);
  border-color: rgba(255, 255, 255, 0.4);
}
```

When pinned: - Button has “active” class - Controls/topbar stay visible - Auto-hide is disabled

When unpinned: - Button loses “active” class - Controls/topbar auto-hide after inactivity - User must move mouse to reveal

File Locations

Client-Side Files

```
public/
├─ index.html           # Main HTML (splash screen structure)
├─ css/
│   └─ oscillaSplash.css # Splash screen styles
└─ js/
    ├─ projectLoader.js   # Project loading, splash wiring
    ├─ oscillaTransport.js # Transport controls, pin logic
    ├─ oscillaPreferences.js # Preferences dialog
    └─ projectScoreSetup.js # Score setup, Inkscape hint
```

Server-Side Files

```
server.js           # Main server (API endpoints)
serverUtils.js      # Project creation utilities
```

Project Template

```
public/scores/template/
├─ score.svg         # Blank SVG template
├─ preferences.json  # Default preferences with pins=true
├─ audio/
├─ pages/
└─ videos/
```

User Projects

```
public/scores/my-project/
├─ score.svg         # User's score
├─ preferences.json  # User's preferences
├─ audio/            # User's audio files
├─ pages/            # User's page overlays
└─ videos/          # User's videos
```

API Endpoints

GET /api/projects

Returns: List of projects with metadata

Response:

```
[
  {
    "name": "my-project",
    "modified": "2025-02-02T10:30:00.000Z"
  }
]
```

Server Implementation: - Reads public/scores/ directory - Filters out hidden directories (starting with .) - Uses fs.statSync() to get modification time - Returns sorted array of project objects

POST /api/project/new

Purpose: Create new project from template

Request Body:

```
{
  "name": "my-new-project"
}
```

Response:

```
{
  "ok": true
}
```

Or on error:

```
{
  "ok": false,
  "error": "Project already exists"
}
```

Server Logic: 1. Validate project name 2. Check if project exists 3. Copy template directory to scores/my-new-project/ 4. Return success/failure

Implementation: serverUtils.js createNewProject(name)

POST /save-preferences/:project

Purpose: Save project preferences

Request Body:

```
{
  "darkMode": true,
  "pinControls": true,
  "pinTopbar": false,
  "playheadColor": "#ff0000"
}
```

Response:

```
{
  "ok": true
}
```

Server Logic: 1. Validate project exists 2. Write JSON to scores/:project/preferences.json 3. Return success/failure

Development Tips**Testing New Project Flow**

1. **Clear localStorage/sessionStorage** to test first-run experience:

```
localStorage.clear();
sessionStorage.clear();
```
2. **Test without URL params** to see splash screen:
<http://localhost:8001/>
3. **Test with URL params** to bypass splash:
<http://localhost:8001/?project=test-project>

Debugging Pin State

Check console for these messages:

```
[Prefs] applyPreferences() called: {...}
[UI] Controls pin initialized.
[UI] Topbar pin initialized.
[UI] Controls pinned – will stay visible.
```

Check window variables in console:

```
console.log('oscillaControlsPinned:', window.oscillaControlsPinned);
console.log('controlsPinned:', window.controlsPinned);
console.log('topbarPinned:', window.topbarPinned);
```

Modifying Default Preferences

To change defaults for all new projects:

1. Edit template/preferences.json
2. Edit projectLoader.js loadPreferences() defaults object
3. Edit oscillaPreferences.js field definitions defaults

Common Issues

Issue: Splash blocks clicks after hiding - **Cause:** pointer-events CSS still blocking - **Fix:** Ensure splash has .hidden class OR display: none

Issue: Pins dont persist after reload - **Cause:** Preferences not being saved - **Fix:** Check /save-preferences endpoint is working

Issue: Project modal shows wrong dates - **Cause:** Server not returning modified field - **Fix:** Check /api/projects endpoint implementation

Summary

The new project flow creates a seamless experience:

1. **Splash screen** appears as modal blocker (must choose action)
2. **“New” button** prompts for name creates project redirects
3. **Server** copies template returns success
4. **Client** loads project applies preferences (pins defaulted to true)
5. **Inkscape hint** shows for new projects (dismissible, rememberable)

6. **User** starts working immediately with pinned controls

The system is designed to be: - **Modal**: Forces deliberate choice (no accidental clicks) - **Fast**: Minimal steps from idea to working - **Configurable**: Preferences system allows customization - **Developer-friendly**: Clear separation of concerns, documented APIs

Last Updated: 2025-02-02
Oscilla Version: 0.4.2

Oscilla OSC System Developer Guide

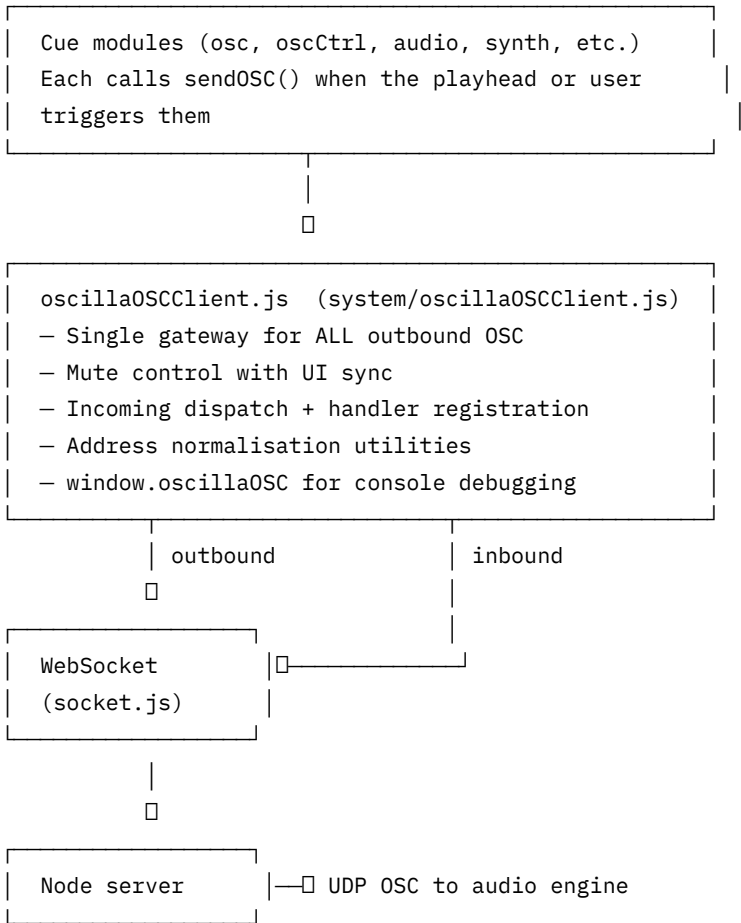
What is OSC in Oscilla?

OSC (Open Sound Control) is how Oscilla talks to external audio engines SuperCollider, Pure Data, Max/MSP, etc. The browser client doesnt produce sound directly. Instead, cues in the SVG score generate OSC messages that travel:

SVG cue → oscillaOSCClient → WebSocket → Node server → UDP OSC → audio engine

Incoming control messages (e.g.from a hardware controller routed through the server) travel the reverse path.

Architecture Overview



Key Files

File	Location	Role
oscillaOSCClient.js	system/	Central gateway all OSC in/out passes through here

File	Location	Role
osc.js	cues/	Discrete osc(...) cue handler + createOscOverlay() for visual HUD
oscCtrl.js	cues/	Continuous control lanes follows SVG path shape as playhead moves
socket.js	system/	WebSocket transport routes incoming osc_in / osc_control to dispatchOSC()
controlRouter.js	control/	Routes control values to animation targets + ParamBus
paramBus.js	control/	Global signal store with pub/sub the nervous system for cross-cue modulation

Cue modules that send OSC

These all import sendOSC from oscillaOSCClient.js:

- **audio.js** audio pool cues (osc_audio_pool messages)
- **synth.js** synth parameter snapshots
- **color.js** colour-as-control data
- **o2p.js** object-to-parameter traversal values
- **rotate.js** rotation angle output
- **scale.js** scale factor output
- **controlXY.js** / **controlXYPresets.js** multitouch XY pad values
- **metro.js** metronome beat/BPM (has its own 50ms throttle)

Sending OSC

All outbound OSC goes through one function:

```
import { sendOSC } from "../system/oscillaOSCClient.js";

sendOSC({
  type: "osc_value",           // message type (server uses this for routing)
  addr: "voice/pitch",        // OSC address (without leading /oscilla/)
  args: [0.5, 0.8],           // values
  timestamp: Date.now()
});
```

sendOSC() handles mute checking, UI preview updates, and WebSocket dispatch. You never need to touch window.socket directly for OSC.

Theres also a convenience form:

```
import { send } from "../system/oscillaOSCClient.js";

send("voice/pitch", 0.5, 0.8);
// Equivalent to sendOSC with type "osc_value"
```

Receiving OSC

Incoming OSC arrives via WebSocket as osc_in or osc_control messages. socket.js hands them to oscillaOSCClient.dispatchOSC(), which does three things:

1. **Fires registered handlers** any module can listen for specific addresses
2. **Stores in ParamBus** at path osc:<normalised-address> for modulation use
3. **Routes through controlRouter** for /oscilla/set and /oscilla/<uid>/<param> patterns

Registering a handler

```
import { onAddress } from "../system/oscillaOSCClient.js";

// Exact match
const unsub = onAddress("fader1", (args, address) => {
  console.log("fader1 value:", args[0]);
});

// Wildcard (prefix match)
const unsub2 = onAddress("mixer/*", (args, address) => {
  console.log(`${address}:`, args);
});

// Clean up when done
unsub();
```

OSC Mute

Global mute prevents all outbound OSC without stopping cue evaluation or overlays. The mute button (#osc-mute-btn) is wired in UIBindings.js and delegates to:

```
import { toggleMuted, setMuted, isMuted } from "../system/oscillaOSCClient.js";

toggleMuted(); // flip state
setMuted(true); // explicit
isMuted(); // query
```

window.oscMuted stays synced for backward compatibility. When muted, the UI preview box shows [muted] prefix but still displays what *would* have been sent.

ParamBus Integration

Every signal in Oscilla flows through ParamBus (control/paramBus.js). Signal paths follow the pattern:

<source>:<id>.<channel>

Examples: - o2p:sliderA.t traversal position of an O2P animation - rotate:orb1.angle rotation angle in degrees - osc:fader1 external OSC input value

Incoming OSC is automatically stored at osc:<address>. You can subscribe to any signal for cross-cue modulation:

```
import * as ParamBus from "../control/paramBus.js";

ParamBus.subscribe("osc:fader1", (value, path, meta) => {
  // React to external fader changes
});
```

Control Router

controlRouter.js sits between signals and targets. It provides:

- **routeControl(uid, param, value)** update a target + ParamBus
- **publishSignal(source, id, channel, value)** emit from an animation (rate-limited)
- **addModulation(signalPath, targetUid, targetParam, options)** wire signaltarget with scale/offset/smoothing/clamp

The router does NOT send OSC itself thats intentional to prevent feedback loops.

Visual Overlays

osc.js exports createOscOverlay() which provides the HUD display attached to cue elements. This is purely visual and completely independent of the sending path. Every cue module imports it separately:

```
import { createOscOverlay } from "./osc.js";

const overlay = createOscOverlay({
  anchorEl: svgElement,
  label: "/voice/pitch",
  mode: "auto"
});

overlay.update("val:0.500");
overlay.position();
overlay.destroy();
```

Message Types

Common type values the server expects:

type	Source	Purpose
osc_value	osc.js, controlXY.js	Generic addressed value
osc_control	oscCtrl.js	Continuous control lane output
osc_audio_pool	audio.js	Audio cue trigger with parameters
oscilla/metro	metro.js	Metronome beat events

Debugging

Open the browser console and use `window.oscillaOSC`:

```

window.oscillaOSC.setDebugMode(true); // log all in/out
window.oscillaOSC.isMuted();          // check mute state
window.oscillaOSC.getLastMessage();   // inspect last sent payload

```

For the full signal state:

```

window.oscillaParamBus.snapshot("osc:"); // all incoming OSC values
window.oscillaParamBus.snapshot();       // everything
window.oscillaRouter.listModulations();   // active signal→target wires

```

drag() Developer Reference

Technical documentation for the drag pre-processor system. Covers architecture, initialization pipeline, data flow, socket protocol, persistence, and API surface.

Architecture Overview

Drag is implemented as a **pre-processor** it runs between `propagate()` and `animationAssign()` in the SVG initialization pipeline, stripping `drag(...)` tokens from element IDs before the Chevrotain parser ever sees them. This means zero changes are needed to the parser grammar, cue handlers, or animation system.

SVG Load Pipeline:

```

settleDomForPropagate()    DOM ready
propagate(svgElement)     expand group IDs to children
cleanupDrag()             remove handlers from previous SVG
preProcessDrag(svgElement) strip drag(), attach pointer handlers
    ↳ requestDragPositions() ask server for saved positions
— mode branch —
animationAssign(svgElement) parser sees clean IDs
assignCues(svgElement)     playhead cues work normally

```

File Location

```

parser/
  preProcessPropagate.js  existing pre-processor
  preProcessReuse.js      existing pre-processor
  preProcessDrag.js       ← this module
  parser.js               Chevrotain grammar

```

Module Exports

```

// Core lifecycle
export function preProcessDrag(svgRoot) // main entry, scans + strips + attaches
export function cleanupDrag()           // remove all handlers (before SVG swap)
export function resetDragPositions()     // zero all positions + clear storage + broadcast

```

```
// Socket message handlers (called from socket.js)
export function handleDragSync(data)           // drag_position from peer
export function handleDragPositionsResponse(data) // full state from server on connect
export function handleDragPositionsReset(data)   // reset broadcast from peer

// Persistence
export function requestDragPositions()          // ask server for saved positions
export function clearDragPositions()            // clear localStorage only
```

All exports are also available on window for console access.

Transform Layering

Drag applies `translate()` on the **outer** `<g>` element. Animations operate on the **inner** `.oscilla-anim` wrapper created by `ensureAnimWrapper()`. No transform conflicts.

```
<g id="scale(...)" transform="translate(dx, dy)">    ← DRAG
  <g class="oscilla-anim" transform="scale(1.5)">    ← ANIMATION
    <circle ... />
  </g>
</g>
```

The `applyTranslate()` helper preserves any existing non-translate transforms on the outer group (e.g. transforms authored in Inkscape).

ID Splitting

Compound IDs are split using the same regex as `cueDispatcher.splitCueId()`:

```
id.split(/\)\s*(?=[a-zA-Z_][a-zA-Z0-9_-]*\s*\()/)
```

This splits `rotate(dir:1, dur:2) drag(1, osc:1)` into: `- rotate(dir:1, dur:2) - drag(1, osc:1)`

The drag expression is extracted, the remaining expressions are re-joined with spaces and written back to `el.id`.

Pointer Handling

Uses the Pointer Events API with `setPointerCapture()` for reliable cross-device drag (mouse, touch, pen). Coordinates are converted from screen space to SVG space via `getScreenCTM().inverse()` so drag works correctly regardless of zoom, pan, or `viewBox` state.

Event flow: 1. `pointerdown` capture, record start position in SVG coords 2. `pointermove` update cumulative translate, apply transform 3. `pointerup` finalize, save to `localStorage`, broadcast, send OSC 4. `pointercancel` revert to position at drag start

Interaction filtering: - Only primary button (left click / single touch) - Ignores clicks on `[data-cue-button]` and `.oscilla-hit-label` so cue buttons and hit labels within draggable groups still work - `touchstart` / `touchmove` listeners prevent scroll while dragging

Data Flow

DRAG EVENT

```
|
|→ applyTranslate(el, dx, dy)      visual update
|→ savePosition(elId, x, y)       localStorage
|→ broadcastDragPosition(elId, x, y) socket → server stores → peers
|→ sendOSC(payload)              if osc:1 enabled (throttled ~60fps)
```

SOCKET CONNECT (late joiner)

```
|
| socket open → send drag_positions_request
```

```

server → drag_positions_response with all saved positions
client → handleDragPositionsResponse() → apply to registered elements

```

SOCKET MESSAGE (real-time peer sync)

```

|
peer drags → server broadcasts drag_position
client → handleDragSync() → applyTranslate + savePosition

```

RESET

```

|
resetDragPositions() → zero all entries in _dragRegistry
↳ clearDragPositions()                clear localStorage
↳ sendSocketMessage("drag_positions_reset") tell server
    server → deletes in-memory + JSON file
    server → broadcasts drag_positions_reset to all peers
    peers → handleDragPositionsReset() → zero + clear local

```

Socket Protocol

Three message types, following the annotation pattern:

drag_position (client server peers)

Sent on every drag move (throttled to ~60fps).

```

{
  "type": "drag_position",
  "project": "myProject",
  "elementId": "rotate(dir:1, dur:30, uid:abc)",
  "x": 142.5,
  "y": -38.2
}

```

Server stores in memory + writes to JSON (debounced 500ms). Broadcasts to all other clients via broadcastToOthers().

drag_positions_request / drag_positions_response

Client requests on socket connect. Server replies with all saved positions.

```

// request
{ "type": "drag_positions_request", "project": "myProject" }

// response (server → requesting client only)
{
  "type": "drag_positions_response",
  "project": "myProject",
  "positions": {
    "rotate(dir:1, dur:30, uid:abc)": { "x": 142.5, "y": -38.2 },
    "scale(min:1, max:2)": { "x": 0, "y": 55.8 }
  }
}

```

drag_positions_reset (client server peers)

Clears all positions for a project.

```

{ "type": "drag_positions_reset", "project": "myProject" }

```

Server Persistence

Positions are stored in-memory and written to disk:

scores/myProject/drag-positions.json

```

{
  "rotate(dir:1, dur:30, uid:abc)": {
    "x": 142.5,
    "y": -38.2
  }
}

```

- **In-memory:** dragPositionsByProject[project][elementId] = { x, y }
- **File writes:** debounced at 500ms via saveDragPositions(project)
- **File reads:** lazy-loaded on first drag_positions_request per project
- **Reset:** deletes both in-memory entry and JSON file

Persistence Key

The persistence key for each draggable element is the **cleaned ID** after drag(...) is stripped. For example:

Original ID	Persistence Key
rotate(dir:1, dur:30, uid:abc) drag(1)	rotate(dir:1, dur:30, uid:abc)
scale(min:1, max:2) drag(1, osc:1)	scale(min:1, max:2)
drag(1) (standalone)	auto-generated: drag_0_a3f2

Standalone drag(1) groups without other cues get a generated key based on sibling index + random suffix. This is stable within a session but **not** across SVG edits. For reliable persistence on standalone draggables, combine with a uid: drag(1) scale(min:1, max:1, uid:myId).

OSC Output

When osc:1 is enabled, drag sends messages through `oscillaOSCClient.sendOSC()`:

```
{
  "type": "osc_drag",
  "uid": "rotate(dir:1, dur:30, uid:abc)",
  "addr": "drag/rotate(dir:1, dur:30, uid:abc)",
  "x": 142.50,
  "y": -38.20,
  "timestamp": 1738000000000
}
```

With `osc:/custom/addr`, the `addr` field uses the custom address instead. Messages are throttled at ~60fps during drag, with a guaranteed final send on `pointerup`.

Global Registry

All draggable elements are tracked in `window._dragRegistry` (a Map):

```
window._dragRegistry.get("rotate(dir:1, dur:30, uid:abc)")
// → {
//   el: SVGGELEMENT,
//   cfg: { enabled: true, osc: false, oscAddr: null },
//   dragging: false,
//   dx: 142.5,
//   dy: -38.2,
//   ...
// }
```

Useful for debugging or programmatic position control.

Console API

```
// Reset all positions to origin (local + server + all peers)
window.resetDragPositions()

// Re-request positions from server
window.requestDragPositions()

// Clean up all handlers (before manual SVG swap)
window.cleanupDrag()
```

```
// Inspect registry
window._dragRegistry

// Check specific element
window._dragRegistry.get("myElementId")

// Programmatic move (not synced — for testing)
const entry = window._dragRegistry.get("myElementId")
entry.dx = 100; entry.dy = 200;
// then call: applyTranslate not exported, use entry directly
```

Known Interactions

Hit Labels

Rotate, scale, and o2p create HTML overlay hit labels (o2pTouchOverlays.js) that sit above the SVG layer. These intercept pointer events before the SVG drag handlers can receive them. Currently, elements with both a hit label and drag(1) will prioritize the hit label (click to play/pause). Drag works on elements without hit labels, or on the parts of a group not covered by the hit label overlay.

Touch Seek

oscillaTransport.js initializes touch-drag seeking on the score container. Drag elements use e.stopPropagation() on pointerdown to prevent the score from seeking when dragging an element.

propagate()

Propagate expands IDs to children before preProcessDrag runs, so each child gets its own drag handler with an independent persistence key and position.

Integration Points

SVGInit.js lifecycle hook

```
import { preProcessDrag, cleanupDrag } from "../parser/preProcessDrag.js";
```

```
// In initializeSVG():
cleanupDrag();
preProcessDrag(svgElement);
```

socket.js message routing

```
import {
  handleDragSync,
  handleDragPositionsResponse,
  handleDragPositionsReset,
} from "../parser/preProcessDrag.js";

// In handleSocketMessage switch:
case "drag_position":      handleDragSync(data); break;
case "drag_positions_response": handleDragPositionsResponse(data); break;
case "drag_positions_reset": handleDragPositionsReset(data); break;
```

```
// In socket open handler:
socket.send(JSON.stringify({ type: "drag_positions_request", project }));
```

server.js persistence

```
const dragPositionsByProject = {};
// + ensureDragPositionsLoaded(), saveDragPositions()
// + 3 switch cases: drag_position, drag_positions_request, drag_positions_reset
```

oscillaOSCClient.js OSC output (existing, no changes needed)

```
import { sendOSC } from "../system/oscillaOSCClient.js";
```

Marker Navigation Architecture

How user-placed markers integrate into the transport navigation system alongside SVG rehearsal marks.

Resolution Order

nav(scroll@name) and jumpToRehearsalMark(name) use a **fallback chain**:

1. Check `window.rehearsalMarks[name]` → SVG-authored rehearsal mark
2. Check `findMarkerByName(name)` → user-placed marker (interaction layer)
3. Log error, do nothing

This is intentional. Rehearsal marks are authored into the score and should take priority. Markers are runtime additions that fill gaps. If a rehearsal mark and a marker share a name, the rehearsal mark wins.

`nav(mark@name)` bypasses step 1 it only searches user markers via `findMarkerByName()`. Use this when you explicitly want to target a marker and not risk hitting a rehearsal mark with the same name.

Where this lives

Step	File	Function
Fallback chain	<code>transportNav.js</code>	<code>jumpToRehearsalMark(mark)</code>
Marker-only lookup	<code>nav.js</code>	<code>action === "mark" case</code>
Marker search	<code>markers.js</code>	<code>findMarkerByName(name)</code>

Unified Nav Points

Arrow keys and FF/RW buttons navigate a **merged list** of rehearsal marks and user markers, sorted by world X position.

How the list is built

`getUnifiedNavPoints()` in `transportNav.js`:

1. Collect rehearsal marks from `window.sortedMarks + window.rehearsalMarks`
→ `{ name, x, source: "rehearsal" }`
2. Collect user markers from `getSortedMarkerNavPoints()`
→ `{ name, x, id, source: "marker" }`
3. Concatenate and sort by x ascending

Rebuild strategy: per-action, not cached

The unified list is rebuilt on **every** arrow key press or FF/RW button click. This is deliberate:

- Markers can be created, deleted, or dragged at any time
- Caching would require invalidation hooks in marker CRUD, drag handlers, and annotation sync
- Merging ~2050 items and sorting is microseconds
- The cost of a stale cache (jumping to a deleted marker, skipping a new one) is worse than the cost of rebuilding

Do not add caching here without a very good reason.

Position-based, not index-based

Next/prev navigation finds the next point **relative to the current playhead position**, not by incrementing a stored index:

```
// "Next" = first nav point with x > playheadX + 1
// "Prev" = last nav point with x < playheadX - 1
```

The 1 tolerance prevents getting stuck when the playhead is exactly on a nav point.

This matters because the playhead can move in many ways manual seeking, cue jumps, timer onComplete actions, server sync and a stored index would go stale after any of them. Position-based lookup is always correct.

Jump Mechanics

Two code paths exist for jumping, depending on the nav point source:

Rehearsal marks name-based

```
jumpToRehearsalMark(name)
  → looks up name in window.rehearsalMarks
  → sets playheadX = entry.x
  → scrollToPlayheadVisual()
  → notifies server (type: "jump")
```

Markers position-based

```
jumpToWorldX(worldX)
  → sets playheadX = worldX directly
  → scrollToPlayheadVisual()
  → notifies server (type: "jump")
```

Arrow key / FF/RW navigation routes through `jumpToNavPoint(point)` which checks `point.source` and calls the appropriate path.

Markers use `jumpToWorldX` instead of `jumpToRehearsalMark` because multiple markers can share the same name (e.g. default “m”). Name-based lookup would always resolve to the leftmost one. Position-based jumping lands on the correct marker regardless of name collisions.

Marker Coordinate Space

Markers store position as `placement.x` in **world coordinates** (SVG `viewBox` units). This is the same coordinate space as `window.playheadX`, `window.scoreWidth`, and rehearsal mark positions.

Markers are positioned correctly because they're created from click events using the standard world-coordinate conversion:

```
// At creation time (markers.js):
placement.x = screenX / localScale // screen pixels → world units

// At jump time (transportNav.js):
window.playheadX = marker.placement.x // world units → playheadX (same space)
scrollToPlayheadVisual() // playheadX → screen transform
```

No coordinate conversion is needed when jumping to a marker `placement.x` and `playheadX` are the same unit.

See `dev-sync-architecture.md` for the full coordinate system documentation.

Name Collisions

Multiple markers with the same name

`findMarkerByName(name)` returns the **leftmost** marker (lowest `placement.x`) when multiple markers share a name. A console warning is logged:

```
[marker] Multiple markers named "intro" — using leftmost (x=4200)
```

This only affects `nav(scroll@name)` and `nav(mark@name)`. Arrow key navigation uses position-based jumping, so all markers are reachable regardless of name.

Marker name matches rehearsal mark name

The rehearsal mark wins in `nav(scroll@name)`. Use `nav(mark@name)` to force marker-only resolution.

Timer onComplete Integration

Countdown timer slots can trigger marker jumps via their `onComplete` field.

Data flow

1. User sets `onComplete: "nav(scroll@bridge)"` in timer editor (`timers.js`)
2. Cue data syncs to server: `{ name, seconds, onComplete }` (socket broadcast)
3. Server runs countdown, cue slot reaches zero (`server.js advanceCountdown`)
4. Server broadcasts: `{ type: "countdown_cue_complete", onComplete: "nav(scroll@bridge)" }`
5. All clients receive message (`socket.js`)

- 6. Each client calls: `handleCueTrigger("nav(scroll@bridge)", false, true)`
- 7. Cue dispatcher parses expression → nav handler → `jumpToRehearsalMark("bridge")`
- 8. Fallback chain: rehearsal mark "bridge" or user marker "bridge"

Why onComplete fires on all clients

The server broadcasts `countdown_cue_complete` to every client, not just the one that started the timer. This is correct all performers should jump together. The timer is server-owned; the completion action should be too.

Single cues vs sequences

- **Sequence cue:** `onComplete` fires when that slot finishes, before advancing to the next slot
- **Single cue:** `onComplete` fires when the standalone countdown reaches zero, before the timer stops

Both paths through `advanceCountdown()` in `server.js` check for and broadcast `onComplete`.

Rehearsal Popup

`openRehearsalPopup()` in `rehearsalUI.js` builds a combined view:

- 1. Existing rehearsal mark buttons (from SVG, unchanged)
- 2. A “Markers” section with buttons for each named user marker

Marker buttons are built from `getSortedMarkerNavPoints()` and use `jumpToRehearsalMark(name)` on click which follows the standard fallback chain.

The marker section is rebuilt each time the popup opens (no caching), so newly created or deleted markers are always reflected.

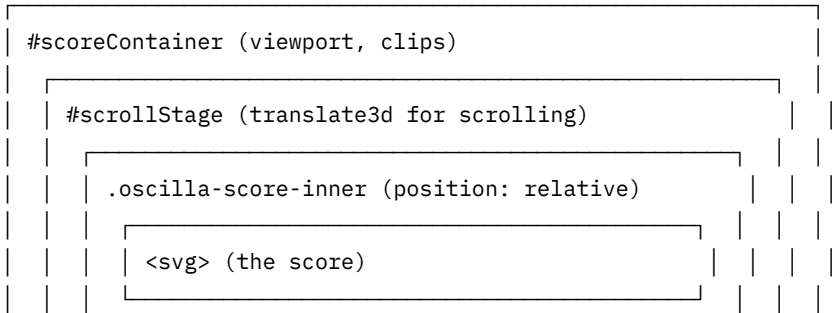
File Map

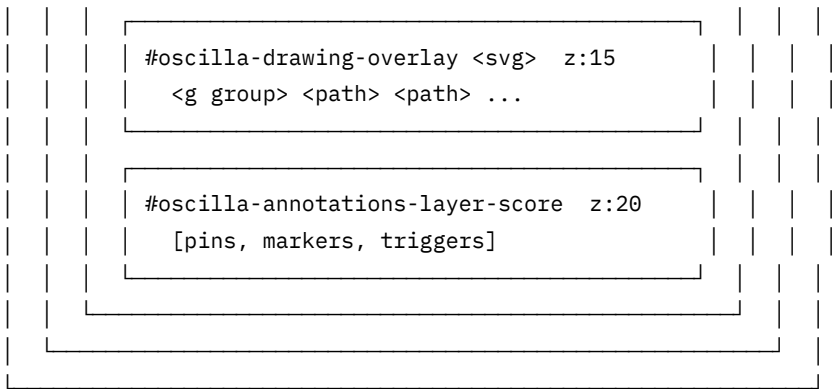
File	What it does for marker nav
<code>markers.js</code>	<code>findMarkerByName()</code> , <code>getSortedMarkerNavPoints()</code> marker lookup and listing
<code>transportNav.js</code>	<code>getUnifiedNavPoints()</code> , <code>jumpToNavPoint()</code> , <code>jumpToWorldX()</code> unified nav, position-based jumping
<code>nav.js</code>	<code>mark</code> and <code>markPaused</code> action handlers explicit marker-only cue targets
<code>rehearsalUI.js</code>	Marker section in rehearsal popup
<code>timers.js</code>	<code>onComplete</code> input field in countdown editor UI
<code>server.js</code>	Stores <code>onComplete</code> , broadcasts <code>countdown_cue_complete</code> on slot completion
<code>socket.js</code>	Handles <code>countdown_cue_complete</code> , dispatches to <code>handleCueTrigger</code>

Freehand Annotation Layer Developer Guide

This document covers the internal architecture, design decisions, and integration points of the freehand annotation system.

Architecture Overview





The drawing overlay is an SVG element parented inside `.oscilla-score-inner`. It sits between the score SVG (z-index implicit) and the annotation layer (z-index 20). Because it is a child of `scrollStage`, it moves with the score automatically when `scrollToPlayheadVisual()` applies its `translate3d` transform. No scroll synchronisation code is needed.

Files Involved

File	Role
js/interaction/drawing.js	Freehand module: overlay, pointer capture, stroke rendering, grouping, selection, toolbar
js/interaction/interactionSurface.js	Orchestrator: imports drawing, calls <code>renderStrokes()</code> in <code>renderAll()</code>
js/interaction/shared.js	CRUD, persistence, WebSocket relay (unchanged)
js/system/uiUtils.js	<code>makeDraggable</code> utility (used for toolbar)
css/oscillaDrawing.css	Styles for overlay, toolbar, selection, hover feedback
index.html	Draw toggle button in topbar

Key Design Decision: SVG over Canvas

The central decision is using an SVG `<svg>` overlay with `<path>` elements rather than an HTML5 `<canvas>`.

Why SVG

Score length. Oscilla scores can be very wide tens of thousands of world units. A `<canvas>` sized to cover the full score at device pixel ratio would exceed browser limits. Chrome caps canvas backing stores at 16384x16384 pixels; Safari at 32768 on the long axis. A 20000-unit score at `localScale 2.0` and `devicePixelRatio 2.0` would need an 80000px-wide canvas, which silently fails or produces blank output. An SVG element has no such pixel limit the browser rasterises only the visible portion.

Tiling a canvas across the score width would solve the size problem but introduces significant complexity: managing multiple canvas elements, splitting strokes across tile boundaries, redrawing on scroll. Not worth it for an annotation layer.

Stroke-level interaction. With SVG, each stroke is a DOM element. Hit-testing for the eraser and the group selection system is free `elementFromPoint()` identifies the path under the pointer. On a canvas, stroke selection requires either maintaining a separate spatial index or redrawing strokes offscreen with unique colours for picking. The DOM approach is simpler and aligns with how the structured annotation system already works.

Persistence. SVG path `d` attributes are compact strings. Stroke data is an array of `{x, y, p}` points stored as JSON in the existing annotation data model. Canvas would require either storing point arrays anyway (and re-rendering on load) or storing rasterised PNGs (lossy, large, no individual stroke manipulation).

Scaling. SVG paths re-render at any resolution. If `localScale` changes (window resize, different device), the paths remain sharp. Canvas content becomes blurry when scaled and needs explicit re-rendering.

Coordinate system. By setting a `viewBox` on the overlay SVG that matches the score SVG, all coordinates are natively in world units. No `localScale` multiplication or division is needed at draw time or render time. The browsers SVG viewport-to-`viewBox` mapping handles the conversion automatically.

Why not Canvas

Performance at high stroke count. SVG DOM elements have overhead. Hundreds of complex paths could slow down rendering. For rehearsal markup this is unlikely to be a problem, but a dense generative drawing session might hit limits. The mitigation is point thinning during capture (the `SIMPLIFY_TOLERANCE` constant) and potential future path simplification.

Pixel-level erasing. Canvas supports `globalCompositeOperation: 'destination-out'` for natural eraser strokes. SVG erasing works at the stroke level only you remove entire paths, not parts of them. For score markup this is actually preferable (you usually want to remove a whole marking, not part of one), but it differs from what users might expect from a painting application.

Summary

For a score annotation layer with unpredictable score dimensions, per-stroke persistence, group selection, and network sharing, SVG is the more robust choice. Canvas would be more appropriate for a dense, high-frequency drawing surface with pixel-level blending a different use case.

Coordinate System

World coordinates

All stroke points are stored in world units the same coordinate space as the SVG scores `viewBox`. The drawing overlays own `viewBox` matches the scores:

```
svgOverlay.setAttribute("viewBox", `${vb.x} ${vb.y} ${vb.width} ${vb.height}`);
```

This means `worldCoordsFromEvent()` converts screen pointer positions to world units using the SVGs own CTM (Current Transformation Matrix):

```
const pt = svgOverlay.createSVGPoint();
pt.x = e.clientX;
pt.y = e.clientY;
const worldPt = pt.matrixTransform(svgOverlay.getScreenCTM().inverse());
```

No `localScale` arithmetic is needed anywhere in the drawing module. This is in contrast to the structured annotation system, which stores world coordinates but must multiply by `localScale` when positioning HTML elements.

Why this works

The structured annotation layer uses HTML `<div>` elements positioned with CSS `left/top` in pixels. These need `localScale` conversion because CSS pixel positioning operates in screen space. The freehand layer uses SVG `<path>` elements inside a `<svg>` with a `viewBox`. The browser maps `viewBox` coordinates to rendered pixels automatically that is what `viewBox` is for.

Scroll Integration

`scrollToPlayheadVisual()` in `oscillaTransport.js` scrolls the score by applying:

```
stage.style.transform = `translate3d(${translateX}px, 0, 0)`;
```

where `stage` is `#scrollStage`, the parent of `.oscilla-score-inner`. Because the drawing overlay is a child of `.oscilla-score-inner`, it inherits the transform. No additional scroll handling is needed.

Notably, `scrollToPlayheadVisual` also sets `container.scrollLeft = 0` Oscilla does not use native scrolling. This means there are no scroll events to listen to, no scroll offsets to account for, and no race conditions between scroll position and overlay position.

Session Grouping

When `setDrawMode(true)` is called, a new `groupId` is generated:

```
currentGroupId = "grp_" + ulidLike();
```

Every stroke created during that session gets this `groupId` stored in `stroke.groupId`. When draw mode is toggled off and on, a new `groupId` is generated.

The `groupId` is stored inside the `stroke` sub-object of the annotation item, not at the top level. This keeps the annotation schema clean the grouping is a drawing-specific concern.

Strokes without a `groupId` (e.g. from older versions) continue to render and function normally; they are simply not selectable as a group.

Pointer Event Flow

`pointerdown` (on SVG overlay)

```
→ route by mode:
  draw   → worldCoordsFromEvent(e) → start activeStroke
  erase   → elementFromPoint → deleteAnnotation
  select → hitTestGroup → selectGroup or begin drag
```

`pointermove`

```
→ draw:   append point, update live <path>
→ select: translate selected paths via SVG transform attribute
```

`pointerup`

```
→ draw:   discard if < 2 points, else addAnnotation(item) with groupId
→ select: commit drag delta to all stroke points via updateAnnotation
```

Pointer capture (`setPointerCapture`) ensures that once drawing or dragging starts, all subsequent move/up events go to the overlay regardless of whether the pointer drifts outside it.

The `touchstart` listener with `preventDefault()` suppresses browser gestures (pan, zoom) while in any active mode. This is the same pattern used by `controlXY.js` for its pad interaction.

Data Model Integration

Strokes are stored as items in the existing `state.items` array with `kind: "stroke"`. This reuses the entire annotation infrastructure:

Concern	How it works
Persistence	<code>addAnnotation()</code> calls <code>saveLocal()</code> writes to <code>localStorage</code>
Network sharing	<code>addAnnotation()</code> calls <code>wsSend("annotation_add", ...)</code> if <code>scope === "shared"</code>
Group move	<code>updateAnnotation(id, { stroke: {..., points: movedPoints} })</code> for each stroke
Scope toggle	<code>updateAnnotation(id, { scope: newScope })</code> for each stroke in selection
Deletion	<code>deleteAnnotation(id)</code> removes from array, saves, broadcasts
Import/export	Strokes are included in <code>exportAnnotationsJSON()</code> / <code>importAnnotationsJSON()</code>
Mode filtering	<code>shouldRenderItem()</code> checks <code>anchor.mode</code> against current scroll/page context
Project scoping	Items are keyed by project name in <code>localStorage</code>

No changes to `shared.js` are required. The drawing module is a pure consumer of the existing CRUD API.

Rendering separation

Although strokes live in `state.items` alongside annotations and markers, they are rendered separately.

In `renderAll()`:

```
for (const item of state.items) {
  if (item.kind === "stroke") continue; // skip - handled by drawing module
  // ... render pins and markers as before
}
renderStrokes(); // drawing module renders its own items
```

This separation exists because strokes render to a different DOM target (the SVG overlay) than annotations (the HTML annotation layer). The drawing modules `renderStrokes()` iterates `state.items`, filters for `kind === "stroke"`, and creates `<path>` elements in the SVG overlay.

Select and Drag

State model

Selection state is a `Set<string>` of `groupIds`:

```
let selectedGroupIds = new Set();
```

Normal click replaces the set with one group. Shift-click toggles a group in/out.

Hit testing

Reuses the same `elementFromPoint` approach as the eraser:

```
function hitTestGroup(e) {
  const target = document.elementFromPoint(e.clientX, e.clientY);
  if (!target || target.tagName !== "path" || !target.dataset.strokeId) return null;
  const item = state.items.find(i => i.id === target.dataset.strokeId);
  return item?.stroke?.groupId || null;
}
```

Each rendered path carries both `data-stroke-id` and `data-group-id` attributes. The hit test finds the stroke, then looks up its `groupId`.

Live drag

During drag, SVG `transform="translate(dx, dy)"` is applied to all paths in selected groups. This is a visual-only transform the actual point data is not modified until drag completes on `pointerup`.

Commit

On `pointerup`, the drag delta is applied to every point in every stroke of every selected group:

```
for (const gid of selectedGroupIds) {
  for (const item of getGroupStrokes(gid)) {
    const movedPoints = item.stroke.points.map(pt => ({
      x: pt.x + dx, y: pt.y + dy, p: pt.p,
    }));
    updateAnnotation(item.id, { stroke: { ...item.stroke, points: movedPoints } });
  }
}
```

Each `updateAnnotation` call saves locally and broadcasts if shared. `triggerRender` is called, which calls `renderAll()`, which calls `renderStrokes()`, which re-creates all paths from the updated data.

Bounding box

`getGroupBounds()` accepts a `Set` of `groupIds` and computes a combined bounding box across all selected groups. The selection rectangle is a single `<rect>` element with dashed stroke and a marching-ants CSS animation. It carries `_baseBounds` for efficient offset during drag.

Network Sync

Outbound (drawing client)

Strokes use the existing annotation WebSocket messages:

- `annotation_add` new stroke (on `pointerup` if `scope === "shared"`)
- `annotation_update` moved stroke points or scope change
- `annotation_delete` erased stroke

These are sent by `addAnnotation`, `updateAnnotation`, and `deleteAnnotation` in `shared.js`. No drawing-specific WebSocket messages exist.

Inbound (receiving client)

Socket messages arrive via `annotationsHandleSocketMessage` in `interactionSurface.js`:

- `annotation_add` pushes the item to `state.items` and calls `renderAll()`
- `annotation_update` replaces the item and calls `renderAll()`
- `annotation_delete` removes the item and calls `renderAll()`

Since `renderAll()` calls `renderStrokes()`, incoming shared strokes are rendered immediately.

Initial sync

On connect, `socketPoll` sends `annotation_list_request` to the server. The server replies with `annotation_list_response` containing all shared items for the project. `loadSharedAnnotations` merges them into `state.items`. Shared strokes from other clients appear on the new client without any drawing-specific code.

Share toggle

`shareDrawingEnabled` (default `true`, matching markers) controls the scope of new strokes. The toolbar share button toggles it. Existing strokes can have their scope changed via `toggleSelectedScope()`, which calls `updateAnnotation(id, { scope: newScope })` for each stroke in the selection.

Eraser Implementation

The eraser uses `document.elementFromPoint()` for hit-testing:

```
function handleEraserTap(e) {
  const target = document.elementFromPoint(e.clientX, e.clientY);
  if (target?.tagName === "path" && target.dataset.strokeId) {
    deleteAnnotation(target.dataset.strokeId);
  }
}
```

Each rendered `<path>` carries `data-stroke-id` matching its annotation item ID. In erase mode, paths have pointer-events: `stroke` (hit-test on the stroke outline, not the bounding box) and hover listeners that highlight on rollover.

This is significantly simpler than canvas-based erasing, where you would need to either re-render each stroke in a unique colour to an offscreen canvas, or maintain a spatial index of stroke bounding boxes.

Draggable Toolbar

The toolbar uses `makeDraggable` from `js/system/uiUtils.js`, imported as a shared utility (the same function used by the annotation editor panel and the controlXY preset panel).

The toolbar is initially positioned with CSS centering (`left: 50%; transform: translateX(-50%); bottom: 60px`). Since `makeDraggable` uses `offsetLeft` for offset calculation, the centering transform must be converted to absolute `left/top` before dragging works correctly. This conversion happens in `updateToolbarState()` via `requestAnimationFrame` on first show:

```
if (shouldShow && wasHidden && drawToolbar.style.transform !== "none") {
  requestAnimationFrame(() => {
    const rect = drawToolbar.getBoundingClientRect();
    drawToolbar.style.left = `${rect.left}px`;
    drawToolbar.style.top = `${rect.top}px`;
    drawToolbar.style.bottom = "auto";
    drawToolbar.style.transform = "none";
  });
}
```

After this one-time conversion, `makeDraggable` handles subsequent drag positioning via `left/top`. The `makeDraggable` function itself already converts `right/bottom` to `left/top` on first drag (via its own `getBoundingClientRect` logic), but it does not handle `transform`, which is why the explicit conversion is needed.

vector-effect: non-scaling-stroke

The CSS rule:

```
#oscilla-drawing-overlay path {
  vector-effect: non-scaling-stroke;
}
```

This SVG property makes stroke width independent of the viewBox-to-viewport transform. A 3-unit stroke looks the same visual thickness whether the score is displayed on a phone or a 4K monitor. Without it, stroke width scales with the viewBox mapping.

For score markup, non-scaling stroke mimics ink on paper. For strokes that are meant to be part of the scores visual language, scaling strokes might be more appropriate. The choice is a single CSS rule, easily toggled or made per-stroke.

Touch Disambiguation

When any mode is active:

```
svgOverlay.style.pointerEvents = "all";
```

This captures all pointer input over the score area. Score dragging is blocked because the overlay sits above the score in the stacking order and stops event propagation.

The toggle is explicit: active mode = overlay captures input, no active mode = normal interaction. No implicit palm rejection or gesture classification is attempted.

Point Thinning

During capture, consecutive points closer than SIMPLIFY_TOLERANCE (0.8 world units) are discarded:

```
const dist = Math.sqrt(dx * dx + dy * dy);
if (dist < SIMPLIFY_TOLERANCE) return;
```

This reduces data size and rendering complexity without visibly affecting stroke quality. The quadratic bezier smoothing in `buildPathD()` interpolates between captured points, so moderate thinning produces smooth results.

Path Smoothing

`buildPathD()` applies quadratic bezier smoothing:

```
for (let i = 1; i < points.length - 1; i++) {
  const cp = points[i];
  const next = points[i + 1];
  const mx = (cp.x + next.x) / 2;
  const my = (cp.y + next.y) / 2;
  d += ` Q ${cp.x} ${cp.y} ${mx} ${my} `;
}
```

Each captured point becomes a control point for a quadratic bezier segment. The on-curve points are midpoints between consecutive captured points. This produces smooth C1-continuous curves that pass near (but not exactly through) each captured point.

This is a standard technique used by most freehand drawing implementations (Paper.js, Excalidraw, tldraw). It is fast (no iterative fitting), produces compact path data, and looks natural.

Pressure Data

The `PointerEvent.pressure` property is captured and stored in each point:

```
{ x: world.x, y: world.y, p: e.pressure || 0.5 }
```

The `p` value is stored but not yet used for rendering. Implementing variable-width strokes would mean changing `buildPathD()` to generate a filled shape (two offset outlines) rather than a single stroked path. The data model and rendering pipeline would not need modification.

Server Changes

None. The drawing module uses the existing annotation WebSocket messages. The server (`server.js`) already stores shared annotations by project in `annotationsByProject` and broadcasts add/update/delete to other clients.

One pre-existing bug was fixed in `interactionSurface.js`: `socketPoll` was sending `"annotation_request"` but the server only handles `"annotation_list_request"`. This type mismatch meant late-joining clients never received existing shared items. The fix applies to all annotation types, not just freehand strokes.

What is not implemented (yet)

Feature	Notes
Page mode	Strokes only anchor to scroll mode currently. Page mode would need <code>anchor.mode: "page"</code> with <code>pageId</code> , same pattern as page-mode annotations.
Pressure-variable width	Data is captured. Rendering change is localised to <code>buildPathD()</code> .
Per-stroke colour/width change	Would require a property editor panel when a stroke is selected.
Drawing layers	Named groups of strokes with independent visibility. Would be a <code>layer</code> field on the stroke data, plus UI for layer management.
Path simplification	Ramer-Douglas-Peucker on completion. Reduces storage for long/complex strokes.
Undo history	Current undo is last-stroke-by-author. A proper undo stack with redo would need a separate data structure outside the annotation CRUD flow.

Developer Guide: Live Console

Technical reference for `oscillaLive.js` and `livecode.css`.

Architecture

The live console is deliberately thin. It creates a UI panel and routes user input through the **existing dispatch pipeline** it does not import or reimplement any cue handlers.

```
User types DSL string
|
v
parseCueToAST(line)      -- validate syntax
|
v
handleCueTrigger(line,   -- dispatch through standard pipeline
  isRemote=false,
  force=true,            -- bypass dedupe (always execute)
  cueElement=targetEl|null)
|
v
cueDispatcher switch(ast.type) --> handler
```

Every cue type the system supports works automatically, including types added after this module was written.

Dependencies

The module depends entirely on window globals set up by existing modules. It has **zero cross-module imports**:

Global	Source	Used for
window.handleCueTrigger	cueDispatcher.js	Dispatch DSL strings
window.parseCueToAST	parser.js	Validate before dispatch
window.oscillaAnimRegistry	animation.js	Target lookup by uid
window.runningAnimations	animation.js (Map)	Check running state
window.oscillaParamBus	paramBus.js	Signal monitor
window.oscillaRouter	controlRouter.js	(future) mod patching

This means:

- No changes to any handler when adding new cue types
- No circular dependency risk
- Module can be loaded or removed without affecting any other module

Files

File	Location	Purpose
oscillaLive.js	js/system/	Module: panel creation, execution, picker, signal monitor
livecode.css	css/	All styling for the panel and picker highlights

Integration

CSS

Add to styles.css imports:

```
@import url("livecode.css");
```

JS

In app.js:

```
import { initLiveConsole } from "../system/oscillaLive.js";
```

Call inside DOMContentLoaded:

```
initLiveConsole();
```

This creates the >_ toggle button inside #topbar-actions, before the .view-tools cluster. The panel is built lazily on first open.

HTML

No changes required. The button and panel are created dynamically.

Exports

```
export function initLiveConsole() // Setup: creates topbar button
export function destroyLiveConsole() // Teardown: removes button + panel
```

Execution Model

Element-targeted cues

Cue types rotate, scale, scaleXY, o2p, color, colour, fade require a DOM element. The console checks this with a regex before dispatch:

```
const ELEMENT_CUES =
  /^(rotate|scale|scaleXY|o2p|color|colour|fade)\s*\(/i;
```

If the user has not picked a target, execution is blocked with an error message.

Elementless cues

Everything else (synth, audio, speed, nav, osc, stop, pause, text, video, etc.) dispatches with `cueElement = null`.

Force flag

All dispatches pass `force=true` to `handleCueTrigger`, which bypasses the dedupe guard (`window.triggeredCues Set`). This is essential livecoded cues must always execute even if the same DSL string was triggered before.

Re-application on same element

When a new animation cue is dispatched to an element that already has a running animation, the handler itself manages teardown. For example, `handleRotateContinuous` calls `e1._oscillaRotateAnim.pause?()` before starting a new anime instance. The console does not need to stop anything explicitly.

Target Resolution

The picker and target input resolve elements through three paths, in order:

1. **Animation registry** `oscillaAnimRegistry[uid].e1` (exact uid match)
2. **DOM** `document.getElementById(id)` (exact id match)
3. **Partial registry match** first key in registry containing the input string

When an element is picked, its SVG id attribute (which contains the DSL expression) is pre-filled into the editor so the user can modify and re-execute.

Element Picker

Pick mode attaches three listeners on the document in capture phase:

- `click` selects the element, exits pick mode
- `mouseover` adds `.livecode-highlight` outline
- `mouseout` removes highlight

`findMeaningfulElement()` walks up from the clicked element looking for the nearest ancestor with `data-anim-uid` or a non-livecode id. This avoids selecting leaf `<tspan>` or `<path>` nodes when the user means the parent group.

Signal Monitor

Subscribes to all ParamBus changes via wildcard:

```
oscillaParamBus.subscribe("*", (value, path) => { ... })
```

Values accumulate in a snapshot object. A `setInterval` at 200ms (5 fps) renders the snapshot to DOM. This avoids DOM thrash from high-frequency signal updates (animations publish at ~60fps).

The filter input narrows the display by case-insensitive substring match on the signal path.

Keyboard Isolation

The panel stops propagation of `keydown`, `keyup`, and `keypress` events at the panel boundary. This prevents transport keybindings (arrow keys, spacebar, etc.) from firing while the user types in the editor or input fields.

```
panelEl.addEventListener("keydown", (e) => e.stopPropagation());
panelEl.addEventListener("keyup", (e) => e.stopPropagation());
panelEl.addEventListener("keypress", (e) => e.stopPropagation());
```

Z-Index

The panel sits at z-index: 38000, between the top bar (35000) and controlXY panels (40000). This ensures it is above the score and transport but below modal controlXY surfaces.

Future Extensions

WebSocket/OSC input

handleCueTrigger is already on window. A single line in the socket message handler could route incoming DSL strings:

```
// in socket.js message handler:
case "livecode":
  window.handleCueTrigger(msg.dsl, true, true, null);
  break;
```

This would allow live coding from an external editor over the network.

Modulation patching

The console could accept mod() expressions to create live modulation routes via oscillatorRouter.mod(). The signal monitor already shows the values that would be routed.

History / recall

Editor content could persist to localStorage per-project, giving a recall buffer of recent expressions.

Autocompletion

The animation registry and ParamBus key list provide all the data needed for uid and signal path auto-completion in the editor.

Related

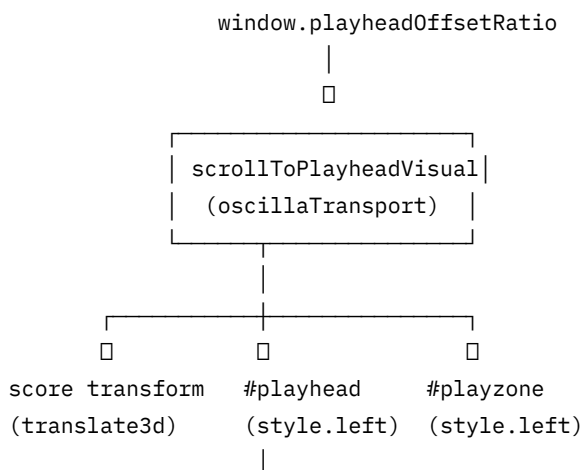
- [Live Console \(user guide\)](#)
- [Cue Dispatcher](#)
- [Control & Modulation](#)
- [Adding a New Cue](#)

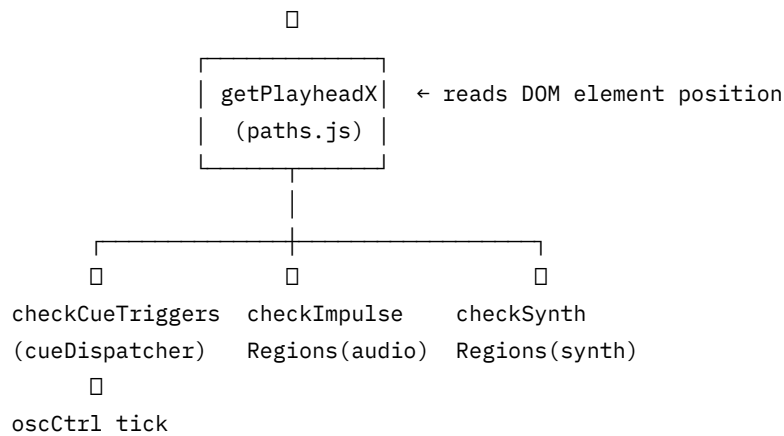
Playhead Offset Developer Guide

This document covers the architecture of the adjustable playhead position system for developer onboarding and maintenance. For musical context on why performers adjust the playhead, see playhead.md.

Architecture Overview

The playhead offset is a single ratio value that controls where on screen the playhead line appears. All downstream systems (cue collision, audio regions, synth regions, OSC control lanes) read the playheads actual DOM position, so they follow automatically.





Key insight: `getPlayheadX()` reads the `#playhead` elements `getBoundingClientRect().left` relative to the score container. It does NOT hardcode screen centre. This means moving the DOM element is sufficient no collision code changes are needed.

Files Involved

File	Role
<code>js/system/playheadOffset.js</code>	Drag handle, lock toggle, persistence, public API
<code>css/playheadOffset.css</code>	Handle and lock icon styling
<code>js/transport/oscillaTransport.js</code>	<code>scrollToPlayheadVisual()</code> reads offset ratio
<code>js/system/paths.js</code>	<code>ensureWindowPlayheadX()</code> reads offset ratio
<code>js/system/paths.js</code>	<code>getPlayheadX()</code> reads DOM position (unchanged)
<code>js/cues/cueDispatcher.js</code>	<code>checkCueTriggers()</code> calls <code>getPlayheadX()</code> (unchanged)
<code>js/cues/audio.js</code>	<code>checkImpulseRegions()</code> calls <code>getPlayheadX()</code> (unchanged)
<code>js/cues/synth.js</code>	<code>checkSynthRegions()</code> calls <code>getPlayheadX()</code> (unchanged)
<code>js/cues/oscCtrl.js</code>	<code>tick()</code> calls <code>getPlayheadX()</code> (unchanged)
<code>js/system/oscillaPreferences.js</code>	Per-project offset slider in Appearance section

Data Flow

The Offset Variable

```
// Single source of truth - default 0.5 (centre)
window.playheadOffsetRatio = 0.5; // range: 0.10 - 0.90
```

Set by: - `playheadOffset.js` on init (from `localStorage`) - Drag interaction (writes to `localStorage`) - Preferences dialog (writes to server, calls `setPlayheadOffset()`)

Read by: - `scrollToPlayheadVisual()` in `oscillaTransport.js` - `ensureWindowPlayheadX()` in `paths.js`

Persistence

Two persistence paths serve different use cases:

Storage	Scope	Written by	Read on
<code>localStorage["oscilla_playheadOffsetRatio"]</code>	Per-device	Drag handle	Page load (init)
<code>preferences.json</code>	Per-project	Preferences dialog	Project load
<code>playheadOffset</code>			

`localStorage` takes priority. The rationale is that different performers on different screen sizes may want different offsets for the same score.

Score Positioning Logic

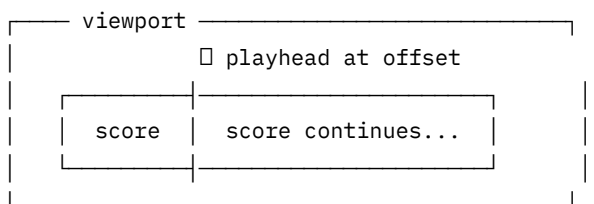
`scrollToPlayheadVisual()` is the core rendering function called every animation frame. The offset change is a single substitution:

```
// BEFORE (hardcoded centre):
const halfViewport = viewportWidth / 2;
let translateX = halfViewport - worldPx;
let playheadScreenX = halfViewport;

// AFTER (configurable offset):
const offsetRatio = window.playheadOffsetRatio ?? 0.5;
const targetScreenX = viewportWidth * offsetRatio;
let translateX = targetScreenX - worldPx;
let playheadScreenX = targetScreenX;
```

The edge clamping logic is unchanged in structure it just uses `targetScreenX` instead of `halfViewport` as the reference point.

Unclamped State (Main Score Body)



```
translateX = (viewportWidth * offsetRatio) - worldPx
playheadEl.style.left = `${offsetRatio * 100}%`
```

Left Clamp (Near Score Start)

When `translateX > 0`, the scores left edge would go past the screens left edge. Instead, the score stays flush left and the playhead moves from screen-left toward the offset position:

```
playheadScreenX = worldPx
translateX = 0
playheadEl.style.left = `${worldPx}px`
```

Right Clamp (Near Score End)

When `translateX < maxShiftLeft`, the scores right edge would go past the screens right edge. The score stays flush right and the playhead moves from the offset toward screen-right:

```
playheadScreenX = viewportWidth - (localRenderedWidth - worldPx)
translateX = maxShiftLeft
playheadEl.style.left = `${playheadScreenX}px`
```

Collision Pipeline (Unchanged)

The cue collision chain requires no modifications:

```
// paths.js - reads actual DOM position
export function getPlayheadX() {
  const playhead = document.getElementById("playhead");
  const scoreContainer = window.scoreContainer;
  if (!playhead || !scoreContainer) return null;

  const containerRect = scoreContainer.getBoundingClientRect();
  const playheadRect = playhead.getBoundingClientRect();
  return playheadRect.left - containerRect.left;
}
```

All consumers call `getPlayheadX()` and compare against cue element rects in the same coordinate space:

- **cueDispatcher.js** (`checkCueTriggers`): edge-crossing detection, OSC re-entrant triggering, repeat region logic

- **audio.js** (checkImpulseRegions): impulse region enter/exit
- **synth.js** (checkSynthRegions): synth region enter/exit
- **oscCtrl.js** (tick): continuous control lane Y-value sampling

Because all of these read the DOM elements actual left, they track the playhead regardless of its screen offset.

Drag Handle UI

The handle is injected into #playhead by playheadOffset.js at init time. Structure:

```
<div id="playhead">
  <div id="repeat-count-box" class="hidden">1</div>
  <!-- injected by playheadOffset.js: -->
  <div id="playhead-offset-handle" class="locked">
    <div class="playhead-grip-dots">
      <span></span><span></span><span></span><span></span></div>
    </div>
    <button id="playhead-lock-btn" class="playhead-offset-lock">
      <svg class="lock-icon-locked">...</svg>
      <svg class="lock-icon-unlocked">...</svg>
    </button>
  </div>
</div>
```

The handle is invisible by default (opacity: 0) and appears on hover via CSS. A ::before pseudo-element on #playhead extends the hover target to 30px wide around the 1px line.

pointer-events: auto on the handle overrides the pointer-events: none on #playhead, so the handle is interactive while the playhead line itself doesn't intercept score clicks.

Lock States

- **Locked** (default): grip shows cursor: default, drag events are ignored. Lock icon shows closed padlock.
- **Unlocked**: grip shows cursor: grab, horizontal drag updates window.playheadOffsetRatio in real time. Lock icon shows open padlock with accent colour.

Drag Behaviour

Horizontal only. On mousemove/touchmove:

```
const newRatio = clamp(clientX / viewportWidth, 0.10, 0.90);
window.playheadOffsetRatio = newRatio;
scrollToPlayheadVisual(); // live score repositioning
applyPlayzonePosition(); // playzone follows
```

On mouseup/touchend, the ratio is persisted to localStorage.

Playzone Tracking

The playzone centres itself on the playhead offset:

```
function applyPlayzonePosition() {
  const ratio = window.playheadOffsetRatio ?? 0.5;
  const halfWidth = 40 / 2; // PLAYZONE_WIDTH_PERCENT / 2
  const leftPercent = (ratio * 100) - halfWidth;
  playzone.style.left = `${leftPercent}%`;
}
```

This is called on drag, on init, and on window resize. The playzone width (40%) is a constant matching the CSS definition.

Public API

Exposed on window for cross-module access:

```
window.playheadOffsetRatio // current ratio (0.10-0.90)
window.initPlayheadOffset() // call once after DOM ready
window.setPlayheadOffset(r) // set ratio, persist, update visual
window.getPlayheadOffset() // read current ratio
```

```

window.resetPlayheadOffset() // reset to 0.5 (centre)
window.applyPlayzonePosition() // reposition playzone to match offset

```

Integration Points

app.js (Init)

```
import { initPlayheadOffset } from './system/playheadOffset.js';
```

```

// In DOMContentLoaded, after initAnimationLoop:
initPlayheadOffset();

```

oscillaPreferences.js (Per-Project Setting)

Field definition in Appearance section:

```

{ key: "playheadOffset", label: "Playhead Position %",
  type: "range", default: 50, min: 10, max: 90, step: 1 }

```

Live application:

```

if (prefs.playheadOffset !== null) {
  const ratio = Number(prefs.playheadOffset) / 100;
  window.setPlayheadOffset?.(ratio);
}

```

styles.css (CSS Import)

```
@import url("playheadOffset.css");
```

Testing Checklist

- Default position at 50% with no localStorage entry
 - Unlock, drag left to ~30% playhead and playzone reposition
 - Cues trigger at the playhead line, not at screen centre
 - Lock prevents drag
 - Offset persists in localStorage across reload
 - Edge clamping works correctly at score start and end
 - Each networked client can have an independent offset
 - oscCtrl continuous lanes track the playhead correctly
 - audio.js impulse regions fire at playhead intersection
 - Synth regions activate/deactivate at playhead
 - Rewind / forward / seek still work
 - Touch drag works on tablet
 - Preferences slider updates the position live
 - Resizing the browser window preserves the ratio
-

Common Issues

Symptom	Cause	Fix
Cues fire at old centre position	getPlayheadX not reading DOM	Verify #playhead element exists and has correct style.left
Playzone doesn't follow	applyPlayzonePosition not called	Ensure it runs on drag, init, and resize
Offset resets on project load	Preferences overwriting localStorage	Check priority: localStorage should win
Handle doesn't appear	CSS not loaded	Verify @import url("playheadOffset.css") in styles.css

Symptom	Cause	Fix
Handle intercepts score clicks	pointer-events conflict	Only the handle div should have pointer-events: auto
Drag feels sluggish	Too many repaints	scrollToPlayheadVisual uses translate3d (GPU-accelerated)