CANVAS OBJECT

Overview

Canvas is an object that lets you draw custom widgets with full multitouch support. The object is adapted from the HTML5 Canvas element, which means that you can do cool stuff like animation, shadows, transparency, etc. The Canvas object opens the door to a completely new Lemur experience, where the appearance and behavior of objects can be designed to precisely match your vision.

Canvas operates within the standard Lemur framework, introducing nearly 50 new scripting functions. We've created a large amount of factory templates as well as tutorial projects to get you started.

Initialisation

In On Demand mode, the Canvas is drawn one single time when project is loaded. At that time, variables may not be initialized yet, so they default to zero. If your draw() script relies custom variables, you might encounter unexpected results (or simply a black Canvas). Using local variables inside your init script instead of external variables ensures that the initial draw on load will be successful.

MIDI/OSC Mapping

You can output data from a Canvas object in several ways. You can assign values to an variable. Then, the variable can be mapped to a MIDI message, using the standard MAPPING window in Lemur Editor. Feedback will also work, as long as your Canvas drawing scripts take the variable into account when you draw your shapes. You can also use noteout() or ctlou() functions in your Canvas scripts.

To send OSC messages from Canvas, you must use oscout(). If you want to receive OSC feedback, create an On OSC script that listens to the same OSC address you specified with oscout().

Variables

There are no built-in variables in the Canvas object. All MIDI/OSC mapping must be done within scripts or with custom variables.

Properties

There are no built-in properties in the Canvas object besides Name and dimensions (X,Y,W,H).

Behavior

Anti-Aliasing (Off, On)

Anti-aliasing produces better results at the cost of a lower framerate. Disabled by default.

Touch (No Touch, Mono-touch, Multi-touch)
Canvas can report single or multi-touch using On Cursor scripts.

Redraw (Always, On Demand)
Always: Canvas is drawn on every frame. This is the default refresh mode, useful when you are animating shapes.

On Demand: Canvas is only drawn when a canvas_refresh() function is explicitly called. This is useful to save memory when you don't need any animation, for example a simple static shape.

Attributes

aaLevel	0 or 1	anti-aliasing off or on
name	text	get object name
rect	{X,Y,W,H}	object's position and dimensions
redrawMode	0 or 1	Always or On Demand redraw mode
touchSupport	0, 1 or 2	No Touch, Mono-touch or Multi-touch support

On Cursor Scripts

Lemur 5.0 introduces three new script execution modes for handling touch points (cursors) in Canvas scripts. These script execution modes will only be available if you create your script inside the Canvas object.

On cursor down

Script is executed as soon as any new cursor begins.

On cursor move

Script is executed as any existing cursor moves.

On cursor up

Script is executed as any cursor is released.

In all three cases, several variables are available inside the script: cursor, hit, x, y. If the Canvas Touch attribute is set to Mono-touch or Multi-touch, the cursor variable identifies which cursor is handled by this script instance. The x and y variables return the raw position of the cursor. See the **Parser Reference** below for more information on hit regions.

On Redraw Scripts

On Redraw scripts are executed when a canvas_refresh() command is called from another script. See the **Parser Reference** below for more information on Canvas refresh.

CANVAS PARSER REFERENCE

Canvas Clearing

Canvas clearing overwrite all or a portion of the pixels with transparent black (0,0,0,0). This also deletes any existing Hit Regions in the cleared area.

IMPORTANT: If no clearing function is called at the start of a frame, the Canvas retains its previous pixels, and any drawing function that is called will write on top of the old pixels. One will generally call a clearing function at the start of a drawing script, unless an "additive" effect is desired.

canvas_clear(c)	clear the whole Canvas to transparent black
canvas_clearRect(c, x, y, width, height)	clear a rectangular portion of the Canvas to transparent black, at origin (*x*, *y*) with size (*width*, *height*)

Canvas State

The Canvas State contains various options that affect drawing, namely:

- current transformation (combination of translation/rotation/scaling operations)
- current stroking style
- current filling style
- current shadow style
- current font size
- current text alignment options

Saving and restoring States

The whole Canvas state can be saved, modified, and restored to a previous saved configuration multiple times. Saving the Canvas State writes it on top of a "stack". Restoring the Canvas State reads the state that is on top of the "stack".

Function	Description	Notes
canvas_save(c)	save the current Canvas (*c*) state, by pushing it to the top of the State stack	after the state is saved, it remains the current state
canvas_restore(c)	restore the current Canvas (*c*) state, by popping it from the State stack	the previous state is discarded

The Canvas State and stack of saved States is always cleared and reset to default at the beginning of a frame. The following functions are used to modify the Canvas State.

Editing Canvas State: Transformations

Each time a transformation function is called, it is combined to the current transformation by way of matrix multiplication. The total Canvas transformation is then used when performing drawing functions: fills and strokes.

IMPORTANT: transformations are actually performed in reverse order to how they were called. For instance, calling the following functions in this order:
- canvas_scale
- canvas_rotate
before filling a path, will first rotate the path, then scale the result, and finally draw it.

Function	Description
canvas_translate(c, x, y)	apply translation transform on Canvas *c* by amount (*x*, *y*)
canvas_scale(c, x, y)	apply scale transform on Canvas *c* from origin (0,0), with factor *x* and *y*
canvas_rotate(c, radians)	apply rotate transform on Canvas *c* from center (0,0), by angle in radians
canvas_resetTransform(c)	reset transformation of Canvas *c* to default ("identity")

Editing Canvas State: Styles

Fill and stroke styles are set separately. A style can be either a Color, or a Gradient Object.
A Color can be represented several ways :

style = 0.8
Single value: grayscale

style = {0.8, 1.0}
Array of 2 items: alpha, grayscale

style = {1.0, 0.4, 0.4}
Array of 3 items: red, green, blue

style = {1.0, 1.0, 0.0, 0.0}
Array of 4 items: alpha, red, green, blue
Gradient objects are described in a later section.

Function	Description
canvas_setFillStyle(c, style)	change fill style of Canvas *c* to *style*, where *style* is either a Color or Gradient Object
canvas_setStrokeStyle(c, style)	change stroke style of Canvas *c* to *style*, where *style* is either a Color or Gradient Object

Editing Canvas State: Line Styles

Several options exist to customize how lines and curves are drawn when stroking a path.

Function	Description	Notes
canvas_setLineWidth(c, width)	set width of stroke lines in pixels	this is affected by the Canvas transform
canvas_setLineCap(c, cap)	set cap style of lines: butt (0), round (1), square (2)	see Canvas HTML spec for descriptions of Cap styles. One can use special constant variables instead of numerical values: lineCapButt, lineCapRound, lineCapSquare

Editing Canvas State: Text Style

Function	Description	Notes
canvas_setFontSize(c, size)	set font size used for text drawing	
canvas_setTextAlign(c, align)	set horizontal alignment used for text drawing: 0/textAlignLeft, 1/textAlignCenter, 2/textAlignRight	special constant variables can be used instead of numerical values
canvas_setTextBaseline(c, baseline)	set baseline vertical position for text drawing: 0/textBaselineBottom, 1/textBaselineMiddle, 2/textBaseLineTop	special constant variables can be used instead of numerical values

Editing Canvas State: Shadow Style

Shadows can be placed underneath any shape that is drawn when filling or stroking a Path. The following functions can be used to control Shadows behaviour.

Function	Description	Notes
canvas_setShadowColor(c, color)	set the Shadow color of Canvas *c* to *color*	*color* is initially transparent black, i.e no shadows are displayed
canvas_setShadowBlur(c, level)	set the Shadow blur level of Canvas *c* to *level* from 0 to 100	*level* is 2 times the standard deviation of the Gaussian blur, in pixels
canvas_setShadowOffset(c, x, y)	set the Shadow 2D offset of Canvas *c* to (*x*, *y*)	initially set to (0,0)

Gradient Objects

The stroke or fill style of a Canvas can be set to a Gradient Object instead of a plain color. Two functions exist to create Gradient Objects: one for linear gradients, the other for radial gradients. Once a Gradient Object has been created, colors are added to it with another function, they are called Color Stops. A Color Stop comprises:

- a Color (see how to represent Colors above)
- an offset in the Gradient, between 0 and 1, respectively start and end of the Gradient

A Gradient Object can be kept in a project variable for later re-use (assignment as a fill/stroke style, adding color stops, etc). Also note that a Gradient Object is not tied to a specific Canvas object, it can be used for several canvasses.

Function	Description	Notes
canvas_createLinearGradient(x1, y1, x2, y2)	create and return a linear Gradient Object that starts at position (*x1*, *y1*) and ends at position (*x2*, *y2*)	
canvas_createRadialGradient(x1, y1, r1, x2, y2, r2)	create a radial Gradient Object, that starts at position (*x1*, *y1*) with radius *r1*, and ends at position (*x2*, *y2*) with radius *r2*	see Canvas HTML spec for a description of how radial gradients are drawn
canvas_gradient_addColorStop(gradient, offset, color)	add Color Stop to Gradient Object *gradient*, with value *color* at offset *offset*	

Paths

A Canvas Path contains one or more Subpaths. A Subpath is list of connected 2D points. A Subpath can be closed, i.e the final point is connected to the first, or unclosed. A Path can be used to:

- perform drawing operations (fill, stroke)
- create hit regions
-

A Canvas object automatically comes with one Path, which is known as the Default Path. All Path editing functions will perform operations on the Canvas's Default Path, if their first argument is the Canvas object. When clearing the Canvas's content, which is typically done at each frame, the Canvas's Default Path is cleared of all its Subpaths.

Alternatively, one can create any number of Path objects, which are not necessarily tied to a Canvas object in particular. They can be stored in Parser variables and referenced later for various tasks. When passing a Path object as the first argument to a Path editing function, the function will perform its action on that Path.

One will generally use Path Objects instead of the Canvas's Default Path, when a Path's construction necessitates to call a lot Path editing functions. Those functions can be called just once, and the Path can be stored in a variable and used later. Using the Canvas's Default Path will generally mean "reconstructing" it every time it is needed (for instance before filling or stroking it at each frame).

IMPORTANT: a Path is distinct from the pixels displayed in a Canvas. Instead, Paths can be used to perform drawing operations that will *in turn* modify the pixels displayed in a Canvas.

Path creation

Function	Description
canvas_beginPath(canvas)	clear all existing Subpaths of *canvas*'s Default Path
canvas_createPath()	create and return a new Path Object

Path editing

All these functions can perform their action on either:

- a Canvas' Default Path, if a Canvas is passed as the first argument
- any Path Object, if a Path Object is passed as the first argument

Function	Description	Notes
canvas_moveTo(c, x, y)	add a Subpath to the Canvas's Default Path or Path Object, with the first point positioned at coordinates (x,y)	
canvas_lineTo(c, x, y)	edit last Subpath of Canvas's Default Path or Path Object, by adding a point at (x,y)	if no Subpath exists, this does the same as canvas_moveTo
canvas_bezierCurveTo(c, x1, y1, x2, y2, x3, y3)	edit last Subpath of Canvas's Default Path or Path Object, by adding points that form a Bezier Curve, from its last point (x,y) to (x3,y3), with control points (x1,y1) and (x2,y2)	if no Subpath exists, a Subpath is created beforehand with its first point at (x1, y1)
canvas_arcTo(c, x1, y1, x2, y2, r)	edit last Subpath of Canvas's Default Path or Path Object, by adding points that form an Arc of radius r, from its last point (x,y) to (x2,y2), with (x1,y1) as a guide point	
canvas_closePath(c)	close last Subpath of Canvas's Default Path or Path Object	
canvas_rect(c, x, y, width, height)	add a closed Subpath to the Canvas's Default Path or Path Object, containing 4 connected points that form a rectangle of origin (x,y) and size (width,height)	
canvas_arc(c, x, y, r, start, end, ccw)	create an Arc of center (x,y), radius r, from start to end in radians, couter-clockwise if ccw is 1, clockwise otherwise, and either connect it to the last point of the last Subpath, or in a new Subpath if none already exists	

Path drawing

A Canvas's Default Path or a Path Object can be used to perform drawing operations in a Canvas.

Function	Description
canvas_stroke(c)	apply the current Canvas (*c*) transformation to the *c*'s Default Path and stroke the result with the *c*'s stroking style, drawing into *c*
canvas_strokePath(c, path)	apply the current Canvas (*c*) transformation to *path* and stroke the result with the current *c*'s stroking style, drawing into *c*
canvas_fill(c)	apply the current Canvas (*c*) transformation to the *c*'s Default Path and fill the result with the *c*'s filling style, drawing into *c*
canvas_fillPath(c, path)	apply the current Canvas (*c*) transformation to *path* and fill the result with the current *c*'s filling style, drawing into *c*

Path utility functions

Function	Description	Notes
canvas_isPointInPath(c, x, y)	test if point (*x*, *y*) is contained inside a Path, and return 1/0 accordingly: if *c* is a Canvas, the test is made against the Canvas Default Path, if *c* is a Path Object the test is made on that	Canvas current transform does not apply here

Text drawing

One can draw into the Canvas using text instead of path filling / stroking. Text drawing is affected by the font size and alignment options which are part of the Canvas State.

Function	Description	Notes
canvas_fillText(c, text, x, y, maxWidth)	fill *text* string using current fill style, font size and alignment, from position (*x*, *y*) affected by current transform, with *maxWidth*used as the maximum text width	*maxWidth* is ignored if 0 or negative

Fast drawing functions

Those functions draw rapidly into the Canvas without using any Paths.

Function	Description
canvas_fillRect(c, x, y, width, height)	fill a rectangle at origin (*x*, *y*) of size (*width*, *height*), using current transform and fill style
canvas_strokeRect(c, x, y, width, height)	stroke a rectangle at origin (*x*, *y*) of size (*width*, *height*), using current transform and stroke style

Controlling Canvas refresh rate

By default, the Canvas is set up to refresh at all frames. This means all Lemur scripts with trigger "On Redraw" will be called at each frame.

Canvas objects can optionally be set up to refresh on-demand only. In that case, Lemur scripts with trigger "On Redraw" will only be called if the *canvas_refresh(c)* function was called earlier during that frame.

Function	Description	Notes
canvas_refresh(c)	request all scripts set to trigger "On Redraw" for Canvas *c* to be called in that frame	this function does not perform anything if Canvas is set to draw "Always"

Hit Regions

Hit Regions are regions of pixels in a Canvas associated with an integer identifier. If a Canvas is touch-enabled, whenever a cursor enters a Hit Region, the corresponding Hit Region identifier will be sent as an argument to any Lemur Scripts attached to trigger "On Cursor Down / Move / Up".

Any number of Hit Regions can be added to a Canvas, with distinct identifiers. Adding a Hit Region with a identifier already attached to another Hit Region will destroy the old one and replace it with the new one. Hit Regions will also be destroyed when calling *canvas_clear* or *canvas_clearRect*.

Function	Description
canvas_addHitRegion(c, path, id)	add a Hit Region to Canvas *c*, with identifier *id*, replacing any known Hit Region with that identifier. if *path* is a non-null existing Path Object, it is used to specify the position and geometry of the Hit Region, otherwise the Canvas' Default Path is used for that

SEQUENCER OBJECTS

Overview

Lemur 5.0 introduces many new objects and functions to help you create step sequencers. There are three new objects: StepNote, StepSwitch and StepSlider. Each of these is suitable for a particular approach to sequencing. You can find the new objects in the Lemur Editor's object palette, or from the object drop-down menu of the In-App Editor. A new type of script execution, On Clock, lets you trigger events in sync to a musical subdivision. This type of script is available anywhere in your template, simply create a new script in Lemur Editor and select 'On Clock' as the execution type. Finally, nearly a dozen new scripting functions give you precise control, up to 8 independent clocks with independent bpm and transport states.

Timing and Framerates

Generally speaking, MIDI and OSC messages are tied to the Lemur app's framerate. Whereas a simple template might run at 60 fps, a very complex template might run only at 35 fps. MIDI and OSC messages are sent once per frame. For most parameter control situations, this is sufficient. In the case of sequencing, however, higher timing accuracy is necessary. To overcome this limitation, we've created a special internal system that can handle MIDI data with much higher timing accuracy. The sequencing objects and On Clock scripts both use this system to send messages exactly when they are scheduled, rather than waiting for the next frame.

Clock

There are eight clocks available in Lemur, one corresponding to each MIDI target. Each clock can operate as an internal master, or slave to an external signal. Sending MIDI clock from Lemur is not supported.

Clocks in slave mode are hard-coded to listen to a MIDI Target, for example clock 3 in slave mode will expect a MIDI clock coming in on MIDI Target 3. In the vast majority of cases, you'll typically use just one clock and therefore work with MIDI Target 0.

The **Slave Clocks to MIDI** option is available in the settings screen of the Lemur app.

Transport

Transport (start, stop, reset, bpm) is access through scripting functions. These functions are listed in the Lemur Editor, at the very bottom of the PROJECT window you can the Internal folder where all internal Lemur functions are listed. You can also find details in the parser reference, further down in this document.

Several templates bundled with Lemur 5.0 have a handy Container with objects and scripts already made to control the transport. "iPad – MonoSequencer" has a vertical transport Container and "iPad – Drum Sequencer" has a horizontal layout. You can copy these transport Containers

into your own templates, or create your own from scratch.

Transport is always clock-dependant. That means that any sequencer object (StepNote, StepSwitch, StepSlider) will stop, start, reset in sync with the clock. Sequencer objects do not independenty start or stop, they always follow the clock state.

On Clock scripts

Lemur 5.0 introduces a new way to trigger scripts. Execution can now be set to “On Clock”, you will be able to choose which Clock (0-8) and a musical subdivision. Any MIDI/OSC messages sent from this script will be in sync with the clock. Generally speaking, any Manual scripts called from the On Clock will also be processed in sync with the clock.

On Clock scripts are useful if you want to send messages at a specific musical time, or if you want to build your own sequencer templates from scratch.

Parser Reference

Following is a list of relevant scripting functions from the Clock folder. The ‘target’ in the arguments always refers to which of the eight clocks you’re manipulating.

clock_getbeats(target)	returns a float with the current beat position
clock_getbpm(target)	returns the current tempo in beats per minutes (bpm)
clock_isrunning(target)	returns the state of the clock (running/stopped)
clock_pause(target)	stop the clock without resetting the beat position
clock_reset(target)	reset the beat position without interrupting playback
clock_setbpm(target, bpm)	set the tempo (bpm)
clock_setoffset(target, ms)	adjust the phase (offset) of the clock in milliseconds
clock_start(start)	start the clock
clock_stop(target)	stop the clock

Custom MIDI

Custom MIDI functions not benefit from the sequencer system, regardless of where you create them. For tight sequencing, use the built-in variables and properties of sequencer objects (StepNote, StepSwitch, StepSlider), or use On Clock scripts.

StepSwitch

Overview

StepSwitch is based on the Switches object. It is best suited as a drum/trigger sequencer. It is an array of two-state buttons. There are a couple key differences explained below that distinguish StepSwitch from Switches.

Layout

Unlike Switches, where the layout of the object is defined in rows and columns, StepSwitch is defined in steps and rows. This means that you can very easily switch between a traditional x0x-style step sequencer, to a more modern 4x4 grid. It's as easy as setting the number of rows from 1 to 4.

It also means that you can work with odd number of rows/steps without worrying about the layout alignment. It's much easier to just try this it out and see for yourself rather than trying to explain this. For example, set a StepSwitches object to 15 steps and 3 rows and see what happens.

Variables

x

A list of the on-off values of the switches in the object. The list goes from top-left to bottom-right.

step

As the clock is running, the switches light up one after the other. The step variable tells you precisely which step is active right now.

out

The out variable sends out the value of the current step. Map this variable to use StepSwitch as a sequencer. You can map the out variable to an OSC/MIDI message in the Lemur Editor's standard MAPPING window.

For example, map the out variable of a StepSwitch to a Note On message. By default, pitch will be set to 60. You now have a single lane sequencer which will send a MIDI Note 60 on every active step. Assigning the x variable MultiSlider to the 'value' property of StepSwitch (i.e seting pitch = MultiSlider.x) will let you control the velocity for each step.

Properties

Name	Name of the object, also used as its default OSC address.
Label	If checked, the Object's name is displayed in the Interface.
Clock	Selects which clock this object follows.
Steps	Number of steps in the sequence and number of steps displayed. (1 to 64)
Rows	Number of rows into which the steps are divided. (1 to 32)
Division	Musical subdivision per step
value	Off steps zero, On steps are scaled to this value. (expression)
length	Length of each step, as a ratio of the Division. (expression)
legato	If enabled, out value will not go to zero before stepping to next switch. If legato is enabled on any given step, this will supersede the length setting for that step. (expression)
swing	Swing value for the entire sequence. Arrays are ignored, only the first value is set to be the global swing for the whole pattern. (expression)
Free Run	Disables phase reset for all parameters. (expression)

Color Off Color for the object's off state
Color On Color for the object's on state

Behaviour

Capture

If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, an Object will react to whatever cursor is present at any moment in its area.

Paint

If this flag is active, you can "paint" on an array of switches by dragging your finger around. If paint is inactive, a touch only toggles the switch you hit first and dragging the finger around has no further effect.

Attributes

capture	0 or 1	capture off/on
clock	0 to 7	select which clock this object follows
color	{integer*,integer*}	color off state, color on state
div	integer	set musical subdivision for all steps (valid settings are 1,2,3,4,6,8,12,16,24,32,48,64)
free_run	0 or 1	enable or disabled free run mode (ignore parameter phase resets)
label	0 or 1	label off/on
name	text	get object name
paint	0 or 1	paint off/on
parameters_phase	int	get number of steps elapsed since clock start
rect	{X,Y,W,H}	object's position and dimensions
row	1 to 32	number of rows
steps	1 to 64	number of steps

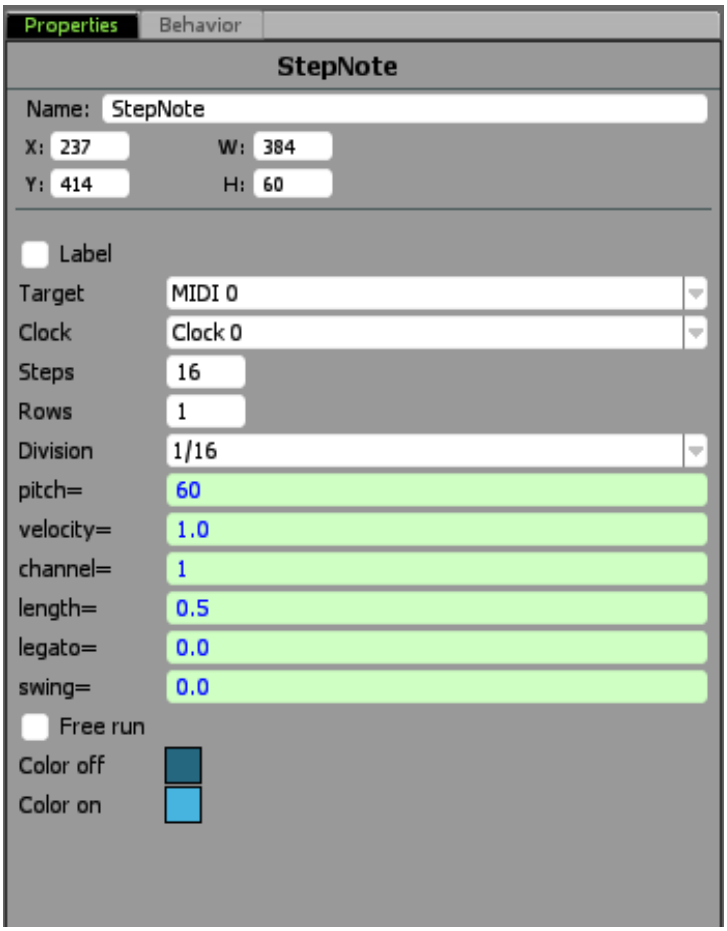
StepNote

Overview

StepNote appears identical to StepSwitch but it has very differences different. Whereas StepSwitch and StepSlider both use the standard Lemur MIDI/OSC mapping system (i.e. in the Lemur Editor MAPPING Window), StepNote handles MIDI mapping in a special way. StepNote handles MIDI mapping internally, as part of object properties. This opens up the door to a very fast and easy way to create custom sequencers.

When you load a StepNote object in the Lemur Editor, have a look at the OBJECTS window,

PROPERTIES tab. See screenshot below.



Unlike the standard MAPPING window, this lets you use an **expression** for all the aspects of MIDI note generation(pitch, velocity, channel, length, legato, switch). An expression can be a constant, an array, a function, or even a reference to another variable. For example you could set pitch=MultiSlider.x, and thus dynamically control the pitch of every step in the StepNote object with the sliders in the MultiSlider.

Variables

x
A list of the on-off values of the cells in the object. The list goes from top-left to bottom-right.

step
A single integer indicating the currently active step. As the clock is running, the cells light up one after the other. The step variable tells you precisely which step is active right now.

Properties

Name	Name of the object, also used as its default OSC address.
Label	If checked, the object's name is displayed in the Interface.
Target	Select MIDI Target used by the object
Clock	Selects which clock this object follows.
Steps	Number of steps in the sequence and number of steps displayed. (1 to 64)
Rows	Number of rows into which the steps are divided. (1 to 32)
Division	Musical subdivision per step
pitch	Set pitch (expression)
velocity	Set velocity (expression)
channel	Set MIDI channel (expression)
length	Length of each step, as a ratio of the Division. (expression)
legato	If enabled, out value will not go to zero before stepping to next switch. If legato is enabled on any given step, this will supersede the length setting for that step. (expression)
swing	Swing value for the entire sequence. (expression)
Free Run	Disables phase reset for all parameters. (expression)
Color Off	Color for the object's off state
Color On	Color for the object's on state

Behaviour

Capture

If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, an Object will react to whatever cursor is present at any moment in its area.

Paint

If this flag is active, you can "paint" on an array of switches by dragging your finger around. If paint is inactive, a touch only toggles the switch you hit first and dragging the finger around has no further effect.

Attributes

capture	0 or 1	capture off/on
clock	0 to 7	select which clock this object follows
color	{integer*,integer*}	color off state, color on state
div	integer	set musical subdivision for all steps (valid settings are 1,2,3,4,6,8,12,16,24,32,48,64)
free_run	0 or 1	enable or disabled free run mode (ignore parameter phase resets)
label	0 or 1	label off/on
name	text	get object name
paint	0 or 1	paint off/on
parameters_phase	int	get number of steps elapsed since clock start

rect	{X,Y,W,H}	object's position and dimensions
row	1 to 32	number of rows
steps	1 to 64	number of steps
target	0 to 7	select MIDI Target used by the object

StepSlider

Overview

StepSlider is based on the MultiSlider object. It is best suited as a parameter/CC sequencer. It is an array of sliders that follows a clock and sends out data for each slider in time.

Variables

x

A list of the vertical positions of all the individual sliders.

step

A single integer indicating the index of the current step. As the clock is running, the sliders light up one after the other. The step variable tells you which step is active right now.

out

A single value indicating the value of the active slider. Map this variable to use StepSlider as a sequencer. You can map the out variable to an OSC/MIDI message in the Lemur Editor's standard MAPPING window.

For example, map the out variable of a StepSlider to a Control Change message. By default, controller will be set to 0. You now have a single lane sequencer which will send a CC0 message on every step, with the value corresping to the vertical position of the slider on that step.

Properties

Name	Name of the object, also used as its default OSC address.
Label	If checked, the object's name is displayed in the Interface.
Clock	Selects which clock this object follows.
Steps	Number of sliders in the sequence. (1 to 64)
Division	Musical subdivision per slider
ramp	Amount of slew between sliders. (expression)
Color	Set the color of the object

Behaviour

Grid

If checked, the range of values produced by the Sliders is quantized into [grid] steps. The maximum number of steps for the Fader is 33.

Capture

If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, the old school way from previous versions is restored, meaning an Object will react to whatever cursor is present at any moment in its area.

Attributes

capture	0 or 1	capture off/on
clock	0 to 7	select which clock this object follows
color	{integer*,integer*}	color off state, color on state
div	integer	set musical subdivision for all steps (valid settings are 1,2,3,4,6,8,12,16,24,32,48,64)
grid	0 or 1	grid off/on
grid_steps	1 to 33	number of grid steps
label	0 or 1	label off/on
name	text	get object name
rect	{X,Y,W,H}	object's position and dimensions
steps	1 to 64	number of steps

NEW IN LEMUR 5.0

String Concatenation

It is now possible to construct strings through concatenation. For example, the following produces the string 'Fader2':

```
decl i = 2;
decl name = "Fader" + i;
```

Referencing child objects

Function	Description
findchild(object,name)	object must point to a valid object, name is a string for the child object name. Returns a reference to the found child, or zero if unfound.

Example:

```
decl i = 2;
decl name = 'Fader' + i; // this produces the string "Fader2"
decl object = findchild(aContainer, name); // this obtains aContainer.Fader2
```