PD-LUA SIGNALS AND GRAPHICS

Albert Gräf
Computer Music Dept.
Johannes Gutenberg University (JGU)
Mainz, Germany
aggraef@gmail.com

Timothy Schoen
@timothyschoen
Utrecht, Netherlands
timschoen123@gmail.com

Benjamin Wesch
Tangible Music Lab
Kunstuniversität Linz, Austria
benjamin.wesch@kunstuni-linz.at

ABSTRACT

Pd-Lua is a programming extension for Miller Puckette's Pd which lets you develop Pd objects in the Lua scripting language. Pd-Lua was originally designed for control processing only. The paper describes the new facilities for signal and graphics processing which have recently been added, so that you can now program pretty much any kind of Pd object in Lua.

1. INTRODUCTION

Pd-Lua lets you develop Pd externals in the Lua scripting language. It was originally written by Claude Heiland-Allen and has since been maintained by a number of people in the Pd community [3]. Pd-Lua also ships with a large collection of instructive examples which you'll find helpful when exploring its possibilities. Lua, from PUC Rio, is open-source (under the MIT license), mature, and supported by a large developer community [11]. It is a small programming language, but very capable and easy to learn. The Lua interpreter is efficient and light-weight, and has been designed to be easily embeddable in other environments. This makes it ideal as a Pd programming extension. For the same reasons, it has also been very popular in game development.

Pd-Lua was originally designed for control processing. The paper reports on some recent work in the 0.12 version to add signal and graphics processing. This effort was spearheaded by Timothy Schoen who designed the API and implemented the (vanilla) Pd and plugdata back-ends, while Benjamin Wesch added the multi-channel support, and Albert Gräf worked on the Purr Data integration. The new release is fully backward-compatible with Pd-Lua 0.11 (and earlier). While the signal and graphics support has been the main focus of the 0.12 release, there are also various other bugfixes and improvements, such as advanced live-coding support [7]. Moreover, Benjamin Wesch reworked our GitHub CI framework so that new releases are now published immediately on Pd's package manager Deken [6].

Pd-Lua works inside any reasonably modern Pd flavor. This encompasses vanilla Pd [14], of course, but also Purr Data [5] which includes an up-to-date version of Pd-Lua for Lua 5.4 and has it enabled by default, so you should be ready to go immediately; no need to install anything else. The same is true for plugdata (version 0.9.1 or later), Timothy Schoen's Pd flavor which can also run as a plug-in inside a DAW [15]. With vanilla Pd, you can install the pdlua package from Deken. You can also compile Pd-Lua from source, using the Github repository [3]. Compilation instructions are in the README, and you'll also find some Mac and Windows binaries there. In either case, after installing Pd-Lua from source or Deken, you also have to add `pdlua` to Pd's startup libraries.

A general introduction to Pd-Lua is beyond the scope of this paper, instead we refer the reader to the pd-lua tutorial which contains many step-by-step instructions and examples [8]. In the following, we'll first discuss how to write a signal processing (a.k.a. dsp) object in Pd-Lua, and then go on to show the implementation of a simple GUI object using the graphics API. The examples presented in this paper are also available for your perusal in the tutorial/examples folder in the sources.

2. SIGNALS

Enabling signal processing in a Pd-Lua object involves three ingredients:

1. **Adding signal inlets and outlets:** As in previous Pd-Lua versions, this is done by setting the `inlets` and `outlets` member variables in the `initialize` method. But instead of setting each variable to just a number, you specify a *signature*, which is a table indicating the signal and control in- and outlets with the special `SIGNAL` and `DATA` values. The number of in- and outlets is then given by the size of these tables. Thus, e.g., you'd use `self.inlets = { SIGNAL, SIGNAL, DATA }` if you need two signal and one control data inlet, in that order. Note that a number as the value of `inlets` or `outlets` corresponds to a signature with just `DATA` values in it.

2. **Adding a dsp method:** This step is optional. The `dsp` method gets invoked whenever signal processing is turned on in Pd, passing two parameters: `samplerate` and `blocksize`. The former tells you about the sample rate (number of audio samples per second) Pd runs at, which will be useful if your object needs to translate time and frequency values from physical units (i.e., seconds, milliseconds, and Hz) to sample-based time and frequency values, so usually you want to store the given value in a member variable of your object. The latter specifies the block size, i.e., the number of samples Pd expects to be processed during each call of the `perform` method (see below). You only need to store that number if your object doesn't have any signal inlets, so that you know how many samples need to be generated. Otherwise the block size can also be inferred from the size of the `in` tables passed to the `perform` method. Adding the `dsp` method is optional. You only have to define it if the signal and control data processing in your object requires the `samplerate` and `blocksize` values, or if you need to be notified when dsp processing gets turned on or the signal processing chain changes for some other reason.

3. **Adding a perform method:** This method is where the actual signal processing happens. It receives blocks of signal data from the inlets through its arguments, where each block is represented as a Lua table containing floating point sample values. The method then needs to return a tuple of similar Lua tables with the blocks of signal data for each outlet. Note that the number of *arguments* of the method matches the number of signal *inlets*, while the number of *return values* corresponds to the number of signal *outlets*. The `perform` method is *not* optional; if your object outputs any signal data, the method needs to be implemented, otherwise you'll get a "perform: function should return a table" or similar error in the Pd console as soon as you turn on dsp processing.

In addition to the `dsp` and `perform` methods, your object may contain any number of methods doing the usual control data processing on the `DATA` inlets. It is also possible to receive control data on the `SIGNAL` inlets; however, you won't be able to receive `float` messages, because they will be interpreted as constant signals which get passed as blocks of signal data to the `perform` method instead.

2.1. Example 1: Mixing signals

Let us take a look at a few simple examples illustrating the kind of processing the `perform` method might do. For starters, let's mix two signals (stereo input) down to a single (mono) output by computing the average of corresponding samples (see Fig. 1).

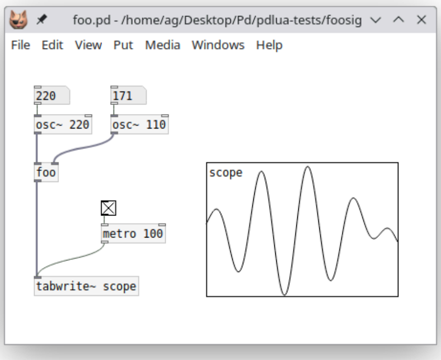
foo.pd - /home/ag/Desktop/Pd/pdlua-tests/foosig
File Edit View Put Media Windows Help
220
171
osc~ 220
osc~ 110
foo
metro 100
tabwrite~ scope
scope

Figure 1: Mixing signals.

We need two signal inlets and one signal outlet, so our `initialize` method looks like this:

```
local foo = pd.class("foo")

function foo:initialize(sel, atoms)
   self.inlets = {SIGNAL,SIGNAL}
   self.outlets = {SIGNAL}
   return true
end
```

And here's the `perform` method (in this simple example we don't need `foo:dsp()`):

```
function foo:perform(in1, in2)
   for i = 1, #in1 do
      in1[i] = (in1[i]+in2[i])/2
   end
   return in1
end
```

Note that here we replaced the signal data in the `in1` table with the result, so we simply return the modified `in1` signal; no need to create a third `out` table. (This is safe because it won't actually modify any signal data outside the Lua method.)

2.2. Example 2: Analyzing a signal

A dsp object can also have no signal outlets at all if you just want to process the incoming signal data in some way and output the result through a normal control outlet. E.g., here's one (rather simplistic) way to compute the rms (root mean square) envelope of a signal as control data (see Fig. 2):

```
function foo:initialize(sel, atoms)
   self.inlets = {SIGNAL}
   self.outlets = {DATA}
   return true
end
```

```
function foo:perform(in1)
   local rms = 0
   for i = 1, #in1 do
      rms = rms + in1[i]*in1[i]
   end
   rms = math.sqrt(rms/#in1)
   self:outlet(1, "float", {rms})
end
```

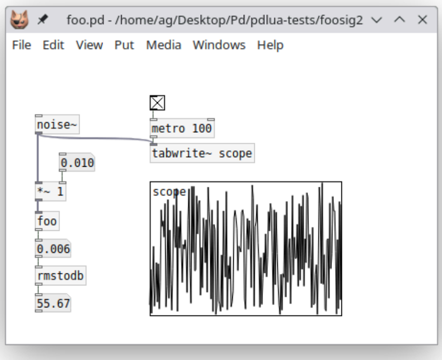
foo.pd - /home/ag/Desktop/Pd/pdlua-tests/foosig2
File Edit View Put Media Windows Help
noise~
metro 100
tabwrite~ scope
0.010
*~ 1
foo
0.006
rmstodb
55.67
scope

Figure 2: Analyzing a signal.

2.3. Example 3: Generating a signal

Conversely, we can also have an object which converts control inputs into signal data, such as this little oscillator object which produces a sine wave (see Fig. 3):

```
function foo:initialize(sel, atoms)
   self.inlets = {DATA}
   self.outlets = {SIGNAL}
   self.phase = 0
   self.freq = 220
   self.amp = 0.5
   return true
end

-- message to set frequency...
function foo:in_1_freq(atoms)
   self.freq = atoms[1]
end

-- ... and amplitude.
function foo:in_1_amp(atoms)
   self.amp = atoms[1]
end

function foo:perform()
   local freq = self.freq
   local amp = self.amp

   -- calculate the angular frequency
   local angular_freq = 2 * math.pi * freq
        / self.samplerate

   local out = {} -- result table
   for i = 1, self.blocksize do
      out[i] = amp * math.sin(self.phase)
      self.phase = self.phase +
          angular_freq
      if self.phase >= 2 * math.pi then
```

```
            self.phase = self.phase - 2 *
                math.pi
        end
    end

    return out
end
```

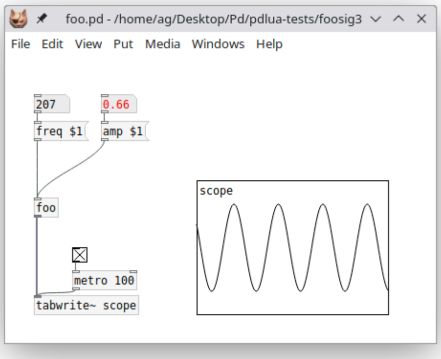
foo.pd - /home/ag/Desktop/Pd/pdlua-tests/foosig3
File Edit View Put Media Windows Help
207
0.66
freq $1
amp $1
foo
metro 100
tabwrite~ scope
scope

Figure 3: Generating a signal.

2.4. Multi-channel support

The examples above all employ Pd's standard single-channel audio signals. But Pd-Lua also has support for Pd's multi-channel signals [4]. This requires Pd 0.54 or later (all signals are single-channel in earlier Pd versions). In Pd-Lua 0.12.20 and later, the `dsp` method receives a *third* `nchannels` argument. This is a Lua table which tells you about the number of channels in the signal data for each inlet. This will be 1 for normal Pd signals, but if `nchannels[i] > 1`, then your `perform` method should be prepared to process that many channels worth of sample data (i.e., the `i`th table argument of `perform` will actually contain `nchannels[i] * blocksize` samples – a block of `blocksize` samples for the first channel, followed by another block of `blocksize` samples for the second channel, etc.).

For the *output* signals, you can set up the desired number of channels per outlet in the `dsp` method, using the `signal_setmultiout` method which is also new in Pd-Lua 0.12.20. In that case your Lua table with the output samples should contain the `blocksize` samples for channel 1 followed by the `blocksize` samples for channel 2, etc., using the same layout as the tables for multi-channel input signals. We won't go into this here any further, but you can find a simple example for your perusal in the examples/multichannel folder in the Pd-Lua source.

3. GRAPHICS

Pd-Lua's new graphics API provides you with a way to equip an object with a static or animated graphical display inside its object box on the Pd canvas. Typical examples would be various kinds of wave displays, or custom GUI objects consisting of text and simple geometric shapes. To these ends, you can adjust the size of the object box to any width and height you specify. Inside the box rectangle you can then draw text and basic geometric shapes such as lines, rectangles, circles, and arbitrary paths, through stroke and fill operations using any rgb color.

In order to enable graphics in a Pd-Lua object, you have to provide a `paint` method. This receives a graphics context `g` as its argument, which lets you set the current color, and draw text and the various different geometric shapes using that color. In addition, you can provide methods to be called in response to mouse down, up, move, and drag actions on the object, which is useful to equip your custom GUI objects with mouse-based interaction.

Last but not least, the `set_args` method lets you store internal object state in the object's creation arguments, while `get_args` lets you retrieve those arguments. This is useful if you need to keep track of persistent state when storing an object on disk (via saving the patch) or when duplicating or copying objects. Also, their companion `canvas_realizedollar` method allows you to expand symbols containing "$" patch arguments like `$0`, `$1`, etc. These three are often combined, but they can also be used separately, and they work just as well with ordinary Pd-Lua objects which don't utilize the graphics API.

We use a custom GUI object, a simple kind of dial, as a running example to illustrate most of these elements in the following subsections. To keep things simple, we will not discuss the graphics API in much detail here, so you may want to check the graphics subpatch in the main pdlua-help patch, which contains a detailed listing of all available methods for reference.

3.1. A basic dial object

Let's begin with a basic clock-like dial: just a circular *face* and a border around it, on which we draw a *center point* and the *hand* (a line) starting at the center point and pointing in any direction which indicates the current *phase* angle. See Fig. 4.

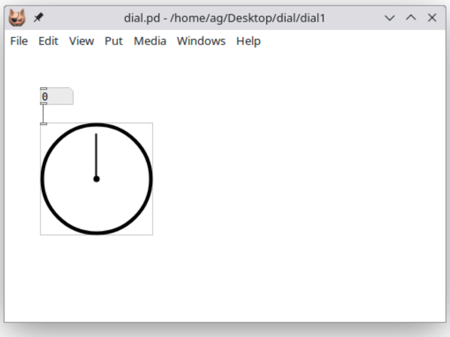
dial.pd - /home/ag/Desktop/dial/dial1
File Edit View Put Media Windows Help
0

Figure 4: Basic dial.

Following the clock paradigm, we assume that a zero phase angle means pointing upwards (towards the 12 o'clock position), while +1 or -1 indicates the 6 o'clock position, pointing downwards. Phase angles less than -1 or greater than +1 wrap around. Positive phase differences denote clockwise, negative differences counter-clockwise rotation. And since we'd like to change the phase angle displayed on the dial, we add an inlet taking float values.

Here's the code implementing the object initialization and the float inlet:

```
local dial = pd.class("dial")

function dial:initialize(sel, atoms)
    self.inlets = 1
    self.outlets = 0
    self.phase = 0
    self:set_size(127, 127)
    return true
end

function dial:in_1_float(x)
    self.phase = x
```

```
   self:repaint()
end
```

The `self:set_size()` call in the `initialize` method sets the pixel size of the object rectangle on the canvas (in this case it's a square with a width and height of 127 pixels). Also note the call to `self:repaint()` in the float handler for the inlet, which will redraw the graphical representation after updating the phase value.

We mention in passing that `self:repaint()` always redraws everything, in this example the face, border, center point, and hand, even though only the hand will change with the phase angle. As of Pd-Lua 0.12.19, it is also possible to partition your graphics into *layers*, each with their own paint method, so that you only need to repaint layers that actually changed, which will improve rendering performance for complex drawings.

We still have to add the `dial:paint()` method to do all the actual drawing:

```
function dial:paint(g)
   local width, height = self:get_size()
   local x, y = self:tip()

   -- object border, fill with bg color
   g:set_color(0)
   g:fill_all()

   -- dial face
   g:fill_ellipse(2, 2, width - 4, height
       - 4)
   g:set_color(1)
   -- dial border
   g:stroke_ellipse(2, 2, width - 4,
       height - 4, 4)
   -- center point
   g:fill_ellipse(width/2 - 3.5, height/2
       - 3.5, 7, 7)
   -- dial hand
   g:draw_line(x, y, width/2, height/2, 2)
end
```

The existence of the `paint` method tells Pd-Lua that this is a graphical object. As mentioned before, this method receives a graphics context `g` as argument. The graphics context is an internal data structure keeping track of the graphics state of the object, which is used to invoke all the graphics operations. The `set_color` method of the graphics context is used to set the color for all drawing operations; in the case of `fill` operations it fills the graphical element with the color, while in the case of `stroke` operations it draws its border. There's just one color value, so we need to set the desired fill color in case of `fill`, and the desired stroke color in case of `stroke` operations. The color values 0 and 1 we use in this example are predefined, and indicate the default background color (usually white) and default foreground color (usually black), respectively.

It is possible to choose other colors by calling `g:set_color(r, g, b)` with rgb color values instead, with each r, g, b value ranging from 0 to 255 (i.e., a byte value). For instance, the color "teal" would be specified as 0, 128, 128, the color "orange" as 255, 165, 0, "black" as 0, 0, 0, "white" as 255, 255, 255, etc. It's also possible to add a fourth *alpha* a.k.a. opacity value a, which is a floating point number in the range 0-1, where 0 means fully transparent, 1 fully opaque, and any value in between will blend in whatever is behind the graphical element to varying degrees. As of Pd-Lua 0.12.7, alpha values are fully supported in both plugdata and Purr Data. In vanilla Pd they are simply ignored at present, so all graphical objects will be opaque no matter what alpha value you specify.

Let's now take a closer look at the drawing operations themselves. We start out by filling the entire object rectangle, which is our drawing surface, with the default background color 0, using `g:fill_all()`. This operation is special in that it not only fills the entire object rectangle, but also creates a standard border rectangle around it. If you skip this, you'll get an object without border, which may be useful at times.

We then go on to fill a circle with the background color, the dial's face. The graphics API has no operation to draw a circle, so we just draw an ellipse instead. The coordinates given to `g:fill_ellipse()` are the coordinates of the rectangle surrounding the ellipse. In this case the width and height values are what we specified with `self:set_size(127, 127)` in the `initialize` method, so they are identical, and thus our ellipse is in fact a circle. Also note that we make the ellipse a little smaller and put it at a small offset from the upper left corner, so the actual width and height are reduced by 4 and the shape is centered in the object rectangle (or square, in this case).

Note that we could have skipped drawing the face entirely at this point, since it just draws a white circle on white background. But we could make the face a different color later, so it's good to include it anyway.

After the face we draw its border, drawing the same ellipse again, but this time in the default foreground color and with a stroke width of 4. We then go on to draw the remaining parts, a small disk in the center which mimics the shaft on which the single hand of the dial is mounted, and the hand itself, which is just a simple line pointing in a certain direction.

Which direction? The line representing the hand goes from the center point width/2, height/2 to the point given by the x, y coordinates. Both width, height and x, y are calculated and assigned to local variables at the beginning of the `paint` method:

```
   local width, height = self:get_size()
   local x, y = self:tip()
```

The `get_size()` call employs a built-in method which returns the current dimensions of the object rectangle; this is part of the graphics API. We could have used the constant 127 from the `initialize` method there, but we could change the size of the object rectangle later, so it's better not to hard-code the size in the `paint` method.

The `tip()` method we have to define ourselves. It is supposed to calculate the coordinates of the tip of the hand. We have factored this out into its own routine right away, so that we can reuse it later when we add the mouse actions. Here it is:

```
function dial:tip()
   local width, height = self:get_size()
   local x, y = math.sin(self.phase*math.
       pi), -math.cos(self.phase*math.pi)
   x, y = (x/2*0.8+0.5)*width, (y
       /2*0.8+0.5)*height
   return x, y
end
```

This just converts the position of the tip from polar coordinates (1, phase) to rectangular coordinates (x, y) and then translates and scales the normalized coordinates to pixel coordinates in the object rectangle which has its center at (width/2, height/2). We also put the tip at a normalized radius of 0.8 so that it is well within the face of the dial. Moreover, the formula computing the x, y pair accounts for the fact that the y coordinates of the object rectangle are upside-down (0 is at the top), and that we want the center-up (a.k.a. 12 o'clock) position to correspond to a zero phase angle. Hence the sin and cos terms have been swapped and the cos term adorned with a minus sign compared to the standard polar - rectangular conversion formula.

So now that we hopefully understand all the bits and pieces, here's the Lua code of the object in its entirety again:

```
local dial = pd.class("dial")

function dial:initialize(sel, atoms)
   self.inlets = 1
   self.outlets = 0
   self.phase = 0
   self:set_size(127, 127)
   return true
end

function dial:in_1_float(x)
   self.phase = x
   self:repaint()
end

-- calculate the x, y position of the tip
-- of the hand
function dial:tip()
   local width, height = self:get_size()
   local x, y = math.sin(self.phase*math.
       pi), -math.cos(self.phase*math.pi)
   x, y = (x/2*0.8+0.5)*width, (y
       /2*0.8+0.5)*height
   return x, y
end

function dial:paint(g)
   local width, height = self:get_size()
   local x, y = self:tip()

   -- object border, fill with bg color
   g:set_color(0)
   g:fill_all()

   -- dial face
   g:fill_ellipse(2, 2, width - 4, height
       - 4)
   g:set_color(1)
   -- dial border
   g:stroke_ellipse(2, 2, width - 4,
       height - 4, 4)
   -- center point
   g:fill_ellipse(width/2 - 3.5, height/2
       - 3.5, 7, 7)
   -- dial hand
   g:draw_line(x, y, width/2, height/2, 2)
end
```

3.2. Adding an outlet

We can already send phase values into our dial object, but there's no way to get them out again. So let's add an outlet which lets us do that. Now that the grunt work is already done, this is rather straightforward. First we need to add the outlet in the `initialize` method:

```
   self.outlets = 1
```

And then we just add a message handler for `bang` which outputs the value on the outlet:

```
function dial:in_1_bang()
   self:outlet(1, "float", {self.phase})
end
```

Fig. 5 shows how our patch looks like now.

3.3. Mouse actions

Our dial now has all the basic ingredients, but it still lacks one important piece: Interacting with the graphical representation itself using the mouse. The graphics API makes this reasonably easy since it provides us with four callback methods that we can implement. Each of these gets invoked with the current mouse coordinates relative to the object rectangle:

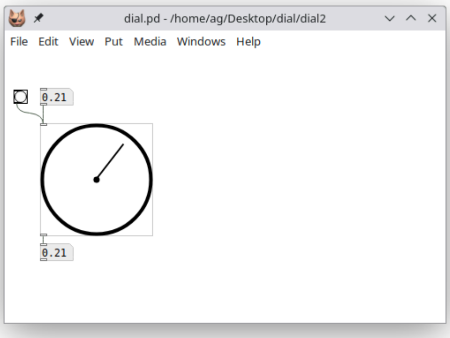
dial.pd - /home/ag/Desktop/dial/dial2
File Edit View Put Media Windows Help
0.21
0.21

Figure 5: Adding an outlet.

- `mouse_down(x, y)`: called when the mouse is clicked
- `mouse_up(x, y)`: called when the mouse button is released
- `mouse_move(x, y)`: called when the mouse changes position while the mouse button is *not* pressed
- `mouse_drag(x, y)`: called when the mouse changes position while the mouse button *is* pressed

Here we only need the `mouse_down` and `mouse_drag` methods which let us keep track of mouse drags in the object rectangle in order to update the phase value and recalculate the tip of the hand. Here's the Lua code. Note that the `mouse_down` callback is used to initialize the `tip_x` and `tip_y` member variables, which we keep track of during the drag operation, so that we can detect in `mouse_drag` when it's time to output the phase value and repaint the object:

```
function dial:mouse_down(x, y)
   self.tip_x, self.tip_y = self:tip()
end

function dial:mouse_drag(x, y)
   local width, height = self:get_size()
   local x1, y1 = x/width-0.5, y/height
       -0.5
   -- calculate the normalized phase,
   -- shifted by 0.5, since we want zero
   -- to be the center up position
   local phase = math.atan(y1, x1)/math.pi
       + 0.5
   -- renormalize if we get an angle > 1,
   -- to account for the phase shift
   if phase > 1 then
      phase = phase - 2
   end

   self.phase = phase

   local tip_x, tip_y = self:tip();

   if tip_x ~= self.tip_x or tip_y ~= self
       .tip_y then
      self.tip_x = tip_x
      self.tip_y = tip_y
      self:in_1_bang()
      self:repaint()
   end
end
```

3.4. More dial action: clocks and speedometers

Now that our dial object is basically finished, let's do something interesting with it. The most obvious thing is to just turn it into a clock (albeit one with just a seconds hand) counting off the seconds. For that we just need to add a metro object which increments the phase angle and sends the value to the dial each second. See Fig. 6.

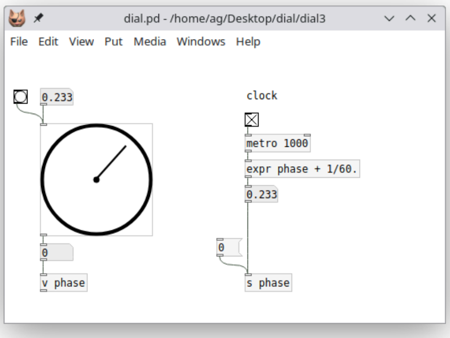
dial.pd - /home/ag/Desktop/dial/dial3
File Edit View Put Media Windows Help
0.233
clock
metro 1000
expr phase + 1/60.
0.233
0
0
v phase
s phase

Figure 6: A clock.

Pd lets us store the phase angle in a named variable (the `v phase` object) which can be recalled in an `expr` object doing the necessary calculations. The `expr` object sends the computed value to the `phase` receiver, which updates both the variable and the upper numbox, and the numbox then updates the dial. Note that we also set the variable whenever the dial outputs a new value, so you can also drag around the hand to determine the starting phase. And we added a `0` message to reset the hand to the 12 o'clock home position when needed.

Fig. 7 shows another little example, rather useless but fun, simulating a speedometer which just randomly moves the needle left and right.

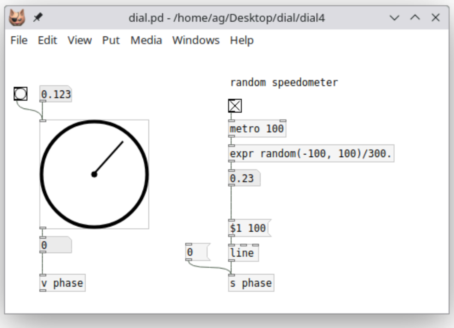
dial.pd - /home/ag/Desktop/dial/dial4
File Edit View Put Media Windows Help
0.123
random speedometer
metro 100
expr random(-100, 100)/300.
0.23
$1 100
line
0
0
v phase
s phase

Figure 7: A speedometer.

We're sure that you can imagine many more creative uses for this simple but surprisingly versatile little GUI object, which we did in just a few lines of fairly simple Lua code. An extended version of this object, which covers some more features of the graphics API that we didn't discuss here to keep things simple, can be found as dial.pd and dial.pd_lua in the pd-lua tutorial examples.

4. REAL-WORLD EXAMPLES

To complement the previous examples, let's examine two more complex applications that demonstrate what's possible with Pd-Lua's graphics capabilities in real-world scenarios.

4.1. Multi-channel signal inspector

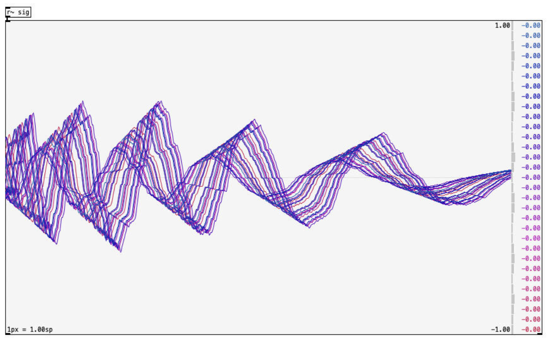

Figure 8: Screenshot of the [show~] object in Pure Data, showing a 31-channel signal of a spherical speaker array on a truncated icosahedron. The visualization reveals the delays applied for beam forming and vector based amplitude panning.

The [show~] object combines a multi-channel waveform view with a panel on the right displaying RMS levels and average values of each channel [17]. The object implements several interactive features:

- Mouse-based zooming through dragging to inspect signals in detail or on a larger scale
- Channel highlighting on hover to focus on specific signals

This combination of multi-channel signal processing and interactive graphics makes it a simple, but valuable tool for analysis and debugging.

4.2. DualSense controller visualization

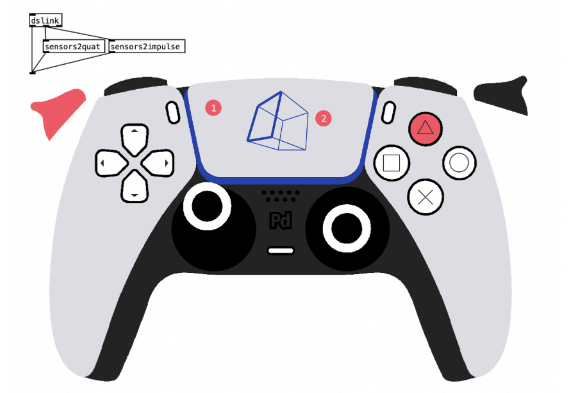

Figure 9: Screenshot of the [dsshow] object in Pure Data, displaying the controller state through input from a HID-based external.

This graphical object provides an intuitive way to monitor and demonstrate controller interaction in Pure Data, creating a detailed visual representation of a DualSense controller that shows real-time state of buttons, analog sticks, touchpad, and gyro data [16].

The implementation demonstrates how Pd-Lua's graphics API can handle complex, responsive GUI elements. With the upcoming SVG support (see Section 5), it will be possible to create this kind of visualization in a much more efficient way.

5. FURTHER DEVELOPMENT

While the current graphics API is highly versatile, certain features could not be implemented due to inherent limitations in vanilla Pd's graphics engine. These limitations include the ability to fill gradients, have rotation transforms, or have paths intersections handled with a specific fill rule (even-odd or nonzero).

To address these limitations and make Pd-Lua truly a complete vector graphics renderer, a new function "draw_svg" is being added to the graphics API. This provides near-complete support for the Scalable Vector Graphics format, enabling aforementioned features such as gradients, path intersections and rotation transforms using Mikko Mononen's NanoSVG library [12]. The restrictions of vanilla Pd's rendering engine are overcome by rasterizing the SVG into an image. The API usage is as follows:

```
function example:paint(g)
   local svg = [[
      <svg width="24" height="24" fill="
          none" viewBox="0 0 24 24"><path
          d="M8 12a4 4 0 1 1 8 0 4 4 0 0
          1-8 0Z" fill="#212121"/></svg>
   ]]
   local x = 10, y = 10
   g:draw_svg(svg, x, y)
end
```

More ongoing work includes expanding the mouse interaction API with `mouse_enter(x, y)` and `mouse_exit(x, y)` callbacks to track whether the mouse is currently over the object. Additionally, an option to choose text justification when rendering text is also in development.

6. CONCLUSION

Pd externals are usually programmed using C, the same programming language that Pd itself is written in. But novices may find C difficult to learn, and the arcana of Pd's C interface may also be hard to master. Programming your externals in Lua makes this much easier and opens up many possibilities.

To help with that, the new capabilities of Pd-Lua 0.12 described in this paper vastly extend the scope of Pd-Lua applications, as you can now program pretty much any kind of Pd object in Lua, covering both signal and control processing, as well as custom GUI objects. Thus, next time you run into a Pd programming problem which cannot be solved easily with a Pd abstraction or an existing external, you may want to consider using Pd-Lua to implement the functionality that you need.

There are some alternatives to pd-lua worth considering if you'd like to program Pd objects with signal processing and/or graphics functionality. py4pd is a relatively new project which is based on the Python scripting language and offers some facilities to draw scores and perform sound analysis [13]. If your main focus is on audio signal processing, then you should look at Grame's Faust programming language [1]. This is a high-level functional programming language specifically tailored to sound processing which compiles to efficient native code. Employing Faust's just-in-time compiler back-end based on the LLVM compiler framework, there's support for running Faust programs in both Max and Pd [2, 9, 10].

7. ACKNOWLEGEMENTS

The authors would like to thank the anonymous reviewers for their suggestions.

8. REFERENCES

[1] Faust Programming Language. https://faust.grame.fr/.

[2] faustgen. https://github.com/grame-cncm/faust/tree/master-dev/embedded/faustgen.

[3] pd-lua. https://agraef.github.io/pd-lua/.

[4] Pd Manual - Chapter 2: Theory of operation. https://msp.ucsd.edu/Pd_documentation/resources/chapter2.htm#s2.5.6.

[5] purr-data. https://agraef.github.io/purr-data/.

[6] pure-data/deken, Feb. 2025. https://github.com/pure-data/deken.

[7] A. Gräf. pdx.lua - advanced live-coding support. https://agraef.github.io/pd-lua/tutorial/pd-lua-intro.html#pdxlua.

[8] A. Gräf. A Quick Introduction to Pd-Lua. https://agraef.github.io/pd-lua/tutorial/pd-lua-intro.html.

[9] A. Gräf. pd-faustgen2, Apr. 2025. https://github.com/agraef/pd-faustgen.

[10] P. Guillot. pd-faustgen: The FAUST compiler embedded in a Pd external. https://github.com/CICM/pd-faustgen.

[11] R. Ierusalimschy, L. H. d. Figueiredo, and W. Celes. The Implementation of Lua 5.0. *JUCS - Journal of Universal Computer Science*, 11(7):1159–1176, July 2005.

[12] M. Mononen. memononen/nanosvg, Mar. 2025. https://github.com/memononen/nanosvg.

[13] C. K. Neimog. py4pd: Bringing Python to PureData, May 2025. https://github.com/charlesneimog/py4pd.

[14] M. Puckette. Software by Miller Puckette. https://msp.ucsd.edu/software.html.

[15] T. Schoen. plugdata: A visual programming environment for audio experimentation, prototyping and education. https://plugdata.org/.

[16] B. Wesch. ben-wes/pd-dualsense, Mar. 2025. https://github.com/ben-wes/pd-dualsense.

[17] B. Wesch. ben-wes/pdlua-show_tilde, Mar. 2025. https://github.com/ben-wes/pdlua-show_tilde.