Code of Art

RubyConf: November, 2009

Jeff Casimir / Jumpstart Lab

processing

Processing is an open source programming language specialized for images, animation, and interaction. It was created at the MIT Media Lab and can best be understood as a specialized Java environment, running code through the JVM, and is cross-platform. It was designed to be easy to understand and appeal to non-programmers.

ruby-processing

Ruby-Processing is a gem created by Jeremy Ashkenas that allows us to interact with Processing from a ruby environment. It is, at once, both a wrapper to the library and the execution environment to run the code utilizing JRuby. Install it with

```
gem install ruby-processing
```

but why ruby?

I'd argue that Ruby-Processing is better than Processing itself. In my opinion, Ruby is an easier language to learn than Java. Even though Processing hides many of Java's complexities, they're still there under the hood. Once you get beyond simple programs, you'll need to know Java. When learning with ruby-processing, the complexities of your program are just Ruby!

Having access to Java libraries and the JVM is incredibly powerful, but thanks to ruby-processing we leverage them from Ruby. There are great libraries for doing 3D, working with sound and video. Plus, we can take advantage of all the libraries and frameworks available for Ruby like Hpricot, ActiveRecord, and FasterCSV.

what processing isn't

There's no "Make Beauty" button. You'll still need to have an inspiration, a goal, to create something great. Most programmers wouldn't classify themselves as artists, in the traditional sense, but with Processing we can practice through emulation, then create great pieces on our own.

core methods & functions

Events & Control

`setup`
Called once when the sketch is started

`draw`
Called over and over and over until the sketch is closed

`mouse_clicked`
Called when a mouse click (press & release) occurs, preempts the draw method.

`mouse_dragged`
Called when the mouse is moved while the button depressed

`mouse_pressed`
`mouse_released`
Called when the mouse button is pressed or released

`key_pressed`
Called when a key is pressed, string **`key`** holds the key that was pressed

Canvas Setup

`smooth`
Turn on anti-aliasing for smooth curves

`background`
Set the window background color

`quit`
Halt execution

Shape Properties

`fill(color_set)`
Set the object fill color

`stroke(color_set)`
Set the object stroke

2D Shapes

`ellipse(x,y,w,h)`
Create an ellipse or circle

`line(x1,y1,x2,y2)`
Create a line or stroke from point **`(X1,Y1)`** to **`(X2,Y2)`**

`quad(x1,y1,x2,y2,`
`x3,y3,x4,y4)`
Create a quadrilateral with corners at the four specified points

`rect(x,y,w,h)`
Create a rectangle anchored at point **`(x,y)`** with width **`(w)`** and height **`(h)`**

`triangle(x1, y1, x2, y2,`
`x3, y3)`
Create a triangle from the three points

`ellipse_mode(CENTER|CORNER)`
`rect_mode(CENTER|CORNER)`
Anchor shape of this type on the **`CENTER`** or the top left **`CORNER`**

Color Set

`0-255`
Gray

`0-255, 0-255`
Gray with Alpha

`0-255, 0-255, 0-255`
RGB Color

`0-255, 0-255, 0-255, 0-255`
RGB Color with Alpha

Create a New Window

```
Sketch.new(
  :width => 1024,
  :height => 768,
  :title => "MySketch",
  :full_screen => false)
```

execution

The ruby-processing library contains everything you need to execute processing code, thanks to an embedded JRuby interpreter. There are three distinct execution modes:

`rp5 run my_sketch.rb`

Load the sketch into the ruby-processing interpreter and run it normally.

`rp5 watch my_sketch.rb`

Run the sketch and, whenever the source file is changed, reinitialize the window.

`rp5 live my_sketch.rb`

Run the sketch and open an IRB terminal that can interact with the live process, including changing variables, redefining methods, etc while live-updating the sketch.

let's code!

Enough talk, let's learn by coding. We'll go through four iterations to create an interactive abstract drawing that interacts with our mouse.

Iteration 0: Up and Running

First, let's confirm that we can get ruby-processing to run, execute code, and render a window. Here's a very barebones structure to get you started.

Create the file to the right and name it **`my_sketch.rb`**

Open a terminal window, change to the file's directory, and start up the interpreter with this command:

`rp5 watch my_sketch.rb`

What have we done? We defined an object named **`Sketch`** which extended **`Processing::App`**. We then defined the two essential methods, **`setup`** to perform our one-time initialization, and **`draw`** to draw the frames of our animation.

Yes, I said animation. Yours isn't moving? Right, it's drawing the same circle in the same place over and over.

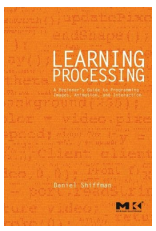
In the **`setup`** method we set a black background, turned on anti-aliasing, then set both ellipses and rectangles to the **`CENTER`** anchoring mode.

Then in the draw method we set the fill to a blue color, define the stroke as white, set a width of ten pixels, then draw an ellipse with it's center anchored at (400,300) and a height and width of 100.

Finally, the last line just creates an instance of our class like a normal Ruby object. Just creating the instance initializes the display window, runs the setup method, then begins looping the draw method.

```
class Sketch < Processing::App

  def setup
    background 0
    smooth

    ellipse_mode CENTER
    rect_mode CENTER

  end

  def draw
    fill 128,128,255
    stroke 255
    stroke_width 10
    ellipse 400, 300, 100, 100
  end
end

Sketch.new(:width => 800, :height => 600,
  :title => "MySketch", :full_screen => false)
```

Resources to continue learning about Processing



Learning Processing

Daniel Shiffman

Processing.org

GitHub.com/ jashkenas

Iteration 1: Animation

Animation is a simple matter of changing variables inside our **draw** method.

In your **setup** method, create the **@x**, **@x_start**, **@y**, and **@y_start** variables to contain your circle target point.

Create a **blue_circle** method that takes two parameters **x** and **y**, specifying the center for the circle. Refactor your **draw** method to use the **blue_circle** method.

Now, to animate we need to change values each time we run the **draw** method. First, draw the circle with the **blue_circle** method.

Then increase the value in **@x** by your **@delta**. If **@x** is getting large, we can go down to the next row by increasing **@y** by **@delta** and setting **@x** back to **@x_start**.

When **@y** gets above 400 pixels, call the **quit** method to halt the animation.

```
def setup
  background 0
  smooth

  ellipse_mode CENTER
  rect_mode CENTER

  @x = @x_start = 100
  @y = @y_start = 100

  @delta = 30
end

def blue_circle(x,y)
  fill 128,128,255
  stroke 255
  stroke_width 10
  ellipse x, y, 100, 100
end

def draw
  blue_circle(@x, @y)
  @x += @delta
  if @x > 300
    @x = @x_start
    @y = @y + @delta
  end

  if @y > 400
    stop
  end
end
```

Iteration 2: Mouse Controlled Action

Now lets start using the mouse. We're going to make it so when we depress the mouse button a circle is drawn in that spot.

Create a variable named **@draw_on** in your **setup** method and set it to **false**.

Change your **draw** method so it checks if **@draw_on** is **true**. If it is **true**, call the **blue_circle** method with the special variables **mouse_x** and **mouse_y** as the parameters. Then set **@draw_on** to **false**.

Add a method named **mouse_pressed** which takes no parameters and just sets **@draw_on** to **true**.

Try out your sketch. Whenever you depress your mouse button, the **mouse_pressed** method is being called, flipping on the **@draw_on** flag. When the next frame is rendered, the **draw** method sees that flag is **true** and outputs a circle, then removes the flag.

```
def setup
  background 0
  smooth

  ellipse_mode CENTER
  rect_mode CENTER

  @draw_on = false
end

def draw
  if @draw_on
    blue_circle(mouse_x,
mouse_y)
    @draw_on = false
  end
end

def mouse_pressed
  @draw_on = true
end

def blue_circle(x,y)
  fill 128,128,255
  stroke 255
  stroke_width 10
  ellipse x, y, 100, 100
end
```

Iteration 3: Injecting Variety

Cute, but I don't want to look at 300 copies of the exact same circle. Let's inject some variety.

In your **setup** method, create the array **@color_set** where each element is itself a three element array representing RGB colors. A suggested set is to the right.

Next, refactor your **blue_circle** method so it has a third parameter **d** that specifies the diameter of the circle and rename the method **my_circle**.

Change your **draw** method to create a variable named **d** that is equal to 100 plus a random number between 0 and 100. Then call **my_circle** with the mouse position parameters and **d** as the third parameter.

Create a method named **random_color** that takes no parameters and returns a randomly selected element from **@color_set**.

Change your **my_circle** method so it gets a color from **random_color**, then passes sends those values as the R, G, and B for the **fill**.

```
def setup
  background 0
  smooth
  ellipse_mode CENTER
  rect_mode CENTER

  @draw_on = false
  @color_set =
[[0,113,188],
[0,173,239],
[68,199,244],
[157,220,249],
[255,235,149]]
end

def random_color

@color_set[rand(@color_set
.size)]
end
```

```
def mouse_pressed
  @draw_on = true
end

def draw
  if @draw_on
    d = rand(100) + 100
    my_circle(mouse_x,
mouse_y, d)
    @draw_on = false
  end
end


def my_circle(x,y,d)
  c = random_color
  fill *c
  stroke 255
  stroke_width 10
  ellipse x, y, d, d
end
```

Iteration 4: Bringing Back Animation

In your **setup** method, keep everything that's there and add a declaration that creates **@d**, sets it equal to **@d_start**, and sets that equal to **10**.

Add a method named **mouse_released** that sets **@draw_on** to **false** and sets **@d** equal to **@d_start**. Add a **mouse_dragged** method that increases the variable **@d** by one.

Create a **random_transparency** method that returns a random number between **128** and **255**.

Change your draw method so it sends in **@d** as the last parameter to **my_circle** and remove the line setting **@draw_on** to **false**.

Change your **my_circle** method so both the **fill** and **stroke** instructions include a transparency value from **random_transparency**.

With that, your sketch is done. I hope you enjoyed experimenting with Processing!

```
def setup
  # Old lines not shown
  @d = @d_start = 10
end

def draw
  if @draw_on
    my_circle(mouse_x,
mouse_y, @d)
  end
end

def random_color
@color_set[
rand(@color_set.size)]
end

def random_transparency
  rand(128) + 128
end
```

```
def mouse_released
  @draw_on = false
  @d = @d_start
end

def mouse_dragged
  @d += 1
end

def my_circle(x,y,d)
  c = random_color
  fill c[0],c[1], c[2],
random_transparency
  stroke 255,
random_transparency
  stroke_width 10
  ellipse x, y, d, d
end
```