Generative Art, SVG, and Celestine

GO TO: Github: redcodefinal/raw_crystal_2020_talk
DOCS: https://docs.celestine.dev

Who Am I?

I'm Ian.

- **Web Developer @ sol.vin**
  Writing fun web apps and what-not
- **Generative Artist**
  Crafting pretty pictures and stuff
- **Security Dabbler**
  Making poorly written Chinese netcam software suffer
- **Celestine Author**
  Shamelessly plugged throughout this whole talk

Who is this talk for?

- People who want to make art
- Web Developers
- Plotters, charters, and visualizers
- Dopamine addicts

What should you know?

- Basic to complete knowledge of Crystal/Ruby
- Basic Kemal/Sinatra usage
- Very Basic HTML/SVG (Celestine does all the heavy lifting don't worry!)

SVG

Scalable Vector Graphics (SVG) is an Extensible Markup Language (XML)-based vector image format for two-dimensional graphics.

- (https://en.wikipedia.org/wiki/Scalable_Vector_Graphics)

```
<svg viewBox="0 0 200 200" width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <defs></defs>
  <path stroke="#5D737E" fill="#C0FDFB" stroke-width="3" transform="rotate(40 100 100) "
    d="M100,10c-50,100 -50,100 0,180c50,-100 50,-100 0,-180" /><text x="87" y="110" fill="#64B6AC"
    style="font-size:40px">C</text>
</svg>
```

What that means for us?

SVG can make pretty graphics, and do it with an HTML-like structure.

DID YOU KNOW?

SVG CAN....

- ➔ Make complicated animations?
- ➔ Use incredibly strong filters?
- ➔ Use CSS and JS in the file format itself?

Celestine

An SVG library for Crystal.

- Domain Specific Language for SVG
- All drawing can be done through Celestine.draw
- Ctx can draw simple elements to the canvas.
- Can be used with IO for on-demand server-side creation of assets
- Easy to use filters, masking, transforming and animation
    - Animation of:
        - Simple attributes
        - Motion along a path
        - Simple transform ops

```
require "kemal"
require "celestine"

get "/" do |env|
  Celestine.draw do |ctx|
    ctx.view_box = {x: 0, y: 0, w: 100, h: 100}
    ctx.rectangle do |rect|
      rect.x = 10
      rect.y = 20
      rect.width = rect.height = 30

      # Use CSS color names
      rect.fill = "red"
      # Use Hex Colors
      rect.stroke = "#000000"

      rect.stroke_width = 10
      rect.stroke_width_units = "px"

      # Always return
      rect
    end
  end
end

Kemal.config.port = ARGV.size == 0 ? 3000 :
ARGV[0].to_i
Kemal.run
```

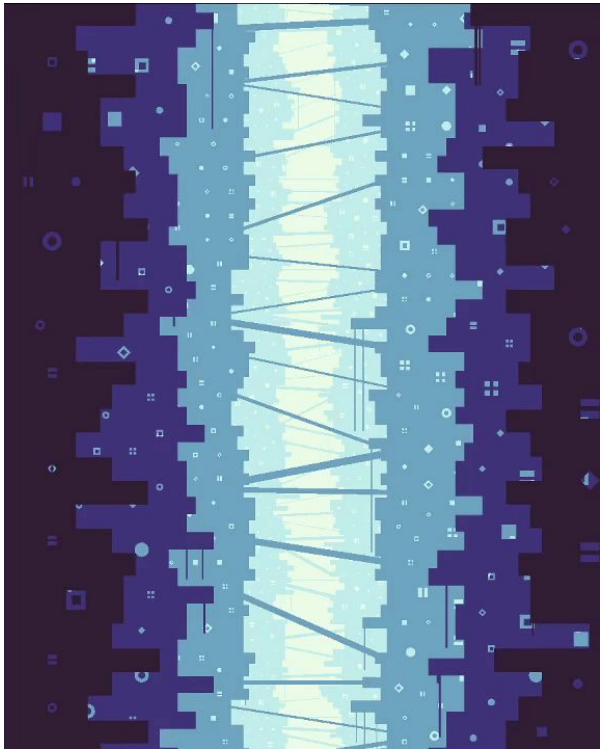
What is Generative Art

Generative art refers to art that in whole or in part has been created with the use of an autonomous system.

- (https://en.wikipedia.org/wiki/Generative_art)

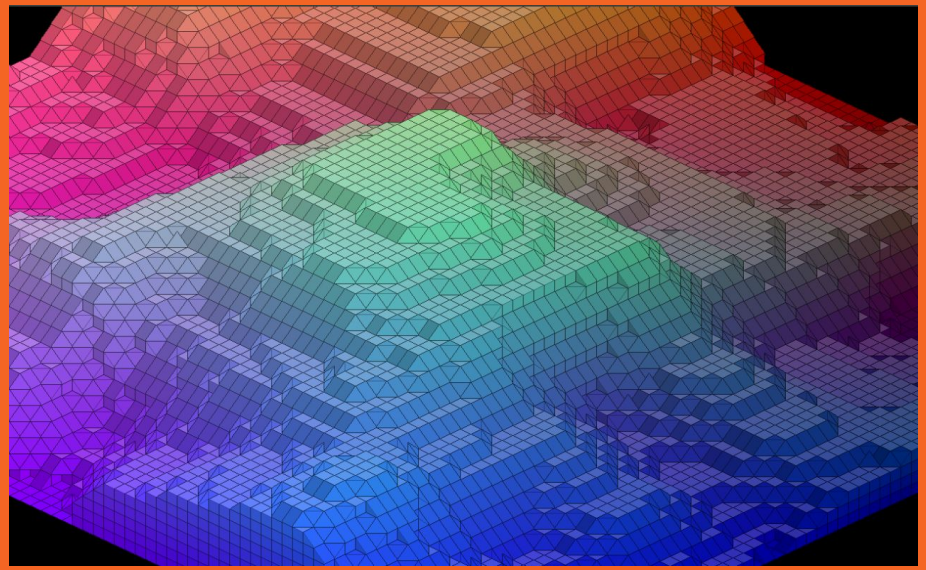
What that means for us?

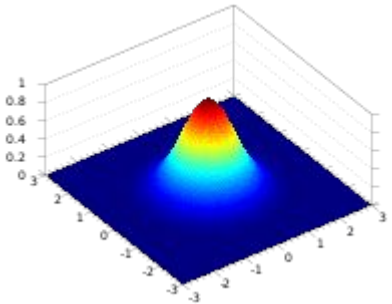
Programs can make art.

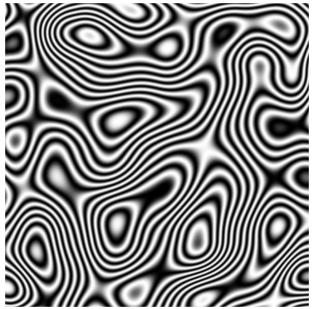


Generative Art Techniques

Or how to generate pretty numbers







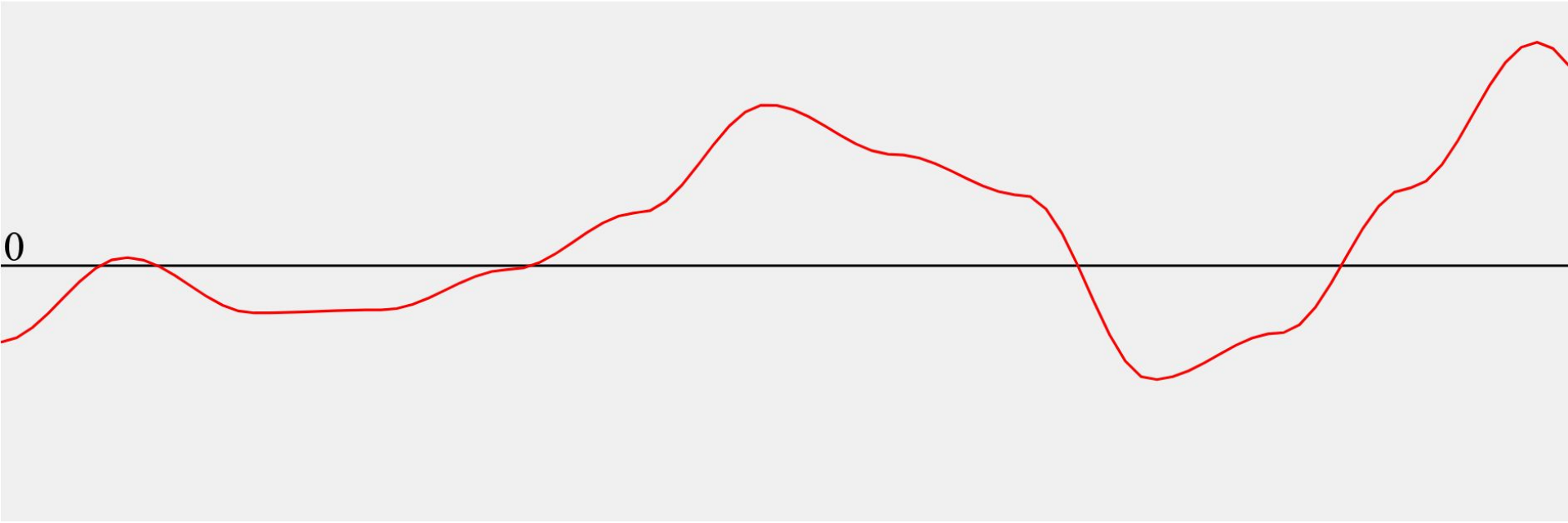
Random Numbers

1. PRNG (Pseudo-random)
    a. Generates a random number with an even distribution
    b. Can be seeded to “re-roll” the same random values
    c. Random.new and rand()
2. Gaussian/Normal Distribution
    a. Generates random numbers but distributes them closer to the “center” mark of zero.
3. Perlin Noise
    a. Takes an (x), (x, y), (x, y, z) coordinate and returns a value from Float32::Max to Float32::Min.
    b. Attempts to normally distribute, at least from my understanding of the rubygem I ported.

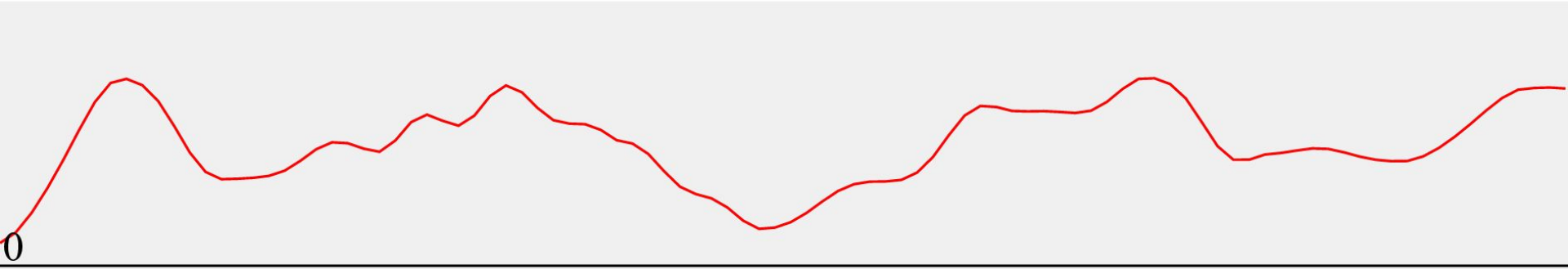
Using Perlin Noise

```
require "perlin_noise"

pn = PerlinNoise.new(rand(1, Int32::MAX))
pn.noise(x: 0, y: 0, z: 0)
pn.normalize_noise(x: 0, y: 0, z: 0) # Restricts the noise by inverting the values between 0 and 1.0
pn.int(x: 0, y: 0, z: 0, min: 0, max: 100) # Gets an int via perlin height
pn.prng_int(x: 0, y: 0, z: 0, min: 0, max: 100) # PRNG using perlin noise as a seed
pn.item(x: 0, y: 0, z: 0, items: ["red", "orange", "yellow"]) # Pulls an item out based on perlin height
pn.prng_item(x: 0, y: 0, z: 0, items: ["red", "orange", "yellow"]) # Pulls an item out based on prng_int

# Use seeds to prevent “tied” values
pn.prng_int(x: 0, y: 0, z: 0, min: 0, max: 100, a_seed: 1.2)
pn.prng_item(x: 0, y: 0, z: 0, items: ["red", "orange", "yellow"], a_seed: 1.1)
```

```
pn.noise(x: 0, y: 0, z: 0)
```



```
pn.normalize_noise(x: 0, y: 0, z: 0)
pn.int(x: 0, y: 0, z: 0, min: 0, max: 100)
pn.item(x: 0, y: 0, z: 0, items: ["red", "orange", "yellow"])
```



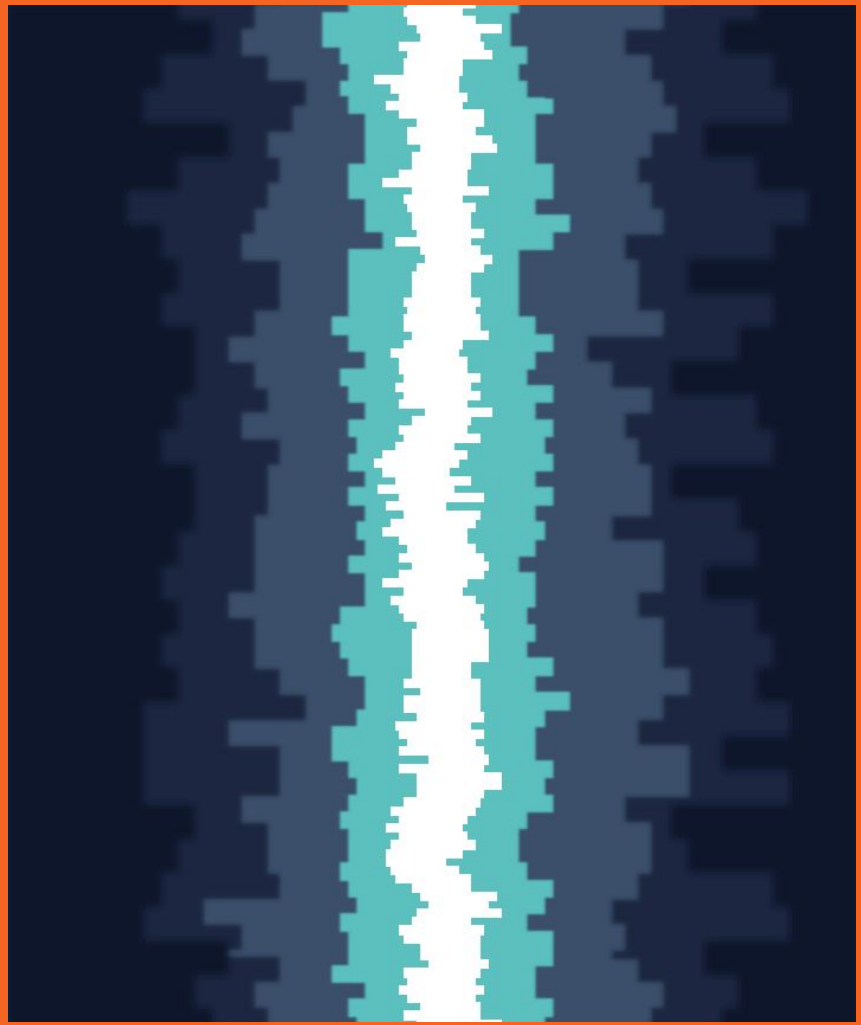
```
pn.prng_int(x: 0, y: 0, z: 0, min: 0, max: 100)
pn.prng_item(x: 0, y: 0, z: 0, items: ["r", "o", "y"])
pn.prng_int(x: 0, y: 0, z: 0, min: 0, max: 100, a_seed: 1.2)
```

Takes the value located at normalize_noise(x, y, z), and uses it as a seed to roll a number on a PRNG.

Can take a “secondary seed” which allows better iterative grabbing of values. This prevents random value “tying” bugs.

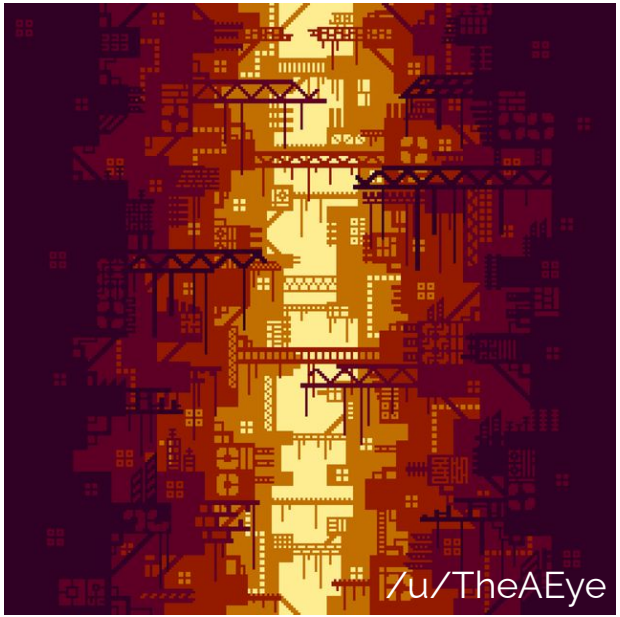
“The Process”

Mineshift-Simple



The setup

- So I'm browsing on Reddit when I come across this picture.
- Something about it really called to me. I like the colors, the way they achieve "depth", and the layout.
- I can easily visually identify the steps I need to take. (construction, masking, etc)

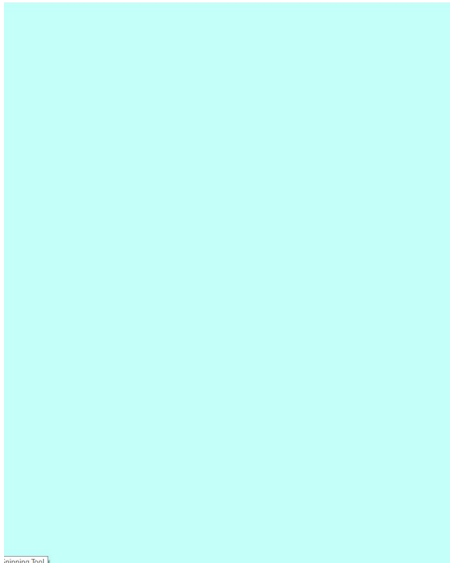


Step 0 - Basic Setup

```
COLORS = [
    {bg: "#e2f1af", levels: ["#e3d888", "#84714f", "#5a3a31", "#31231e"]},
    {bg: "#ddf3b5", levels: ["#83c923", "#74a31d", "#577a15", "#39510e"]}
]
```

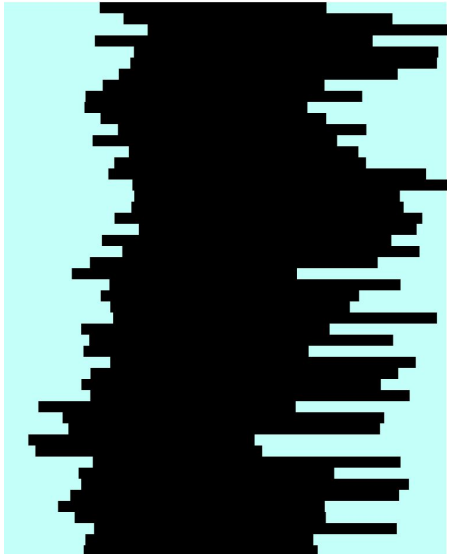
```
module Seeds
  COLORS = 1.1_f32
  CENTER_RECT_DEVIATION = 1.2_f32
  CENTER_RECT_PERLIN_DEVIATION = 1.3_f32
  CENTER_RECT_WIDTH = 1.4_f32
end
```

Step 1 - Choose Color Scheme



```
perlin = PerlinNoise.new(seed)
colors = perlin.prng_item(0, COLORS, Seeds::COLORS)
Celestine.draw do |ctx|
  ctx.view_box = {x: 0, y: 0, w: WIDTH, h: HEIGHT}

  ctx.rectangle do |r|
    r.x = 0
    r.y = 0
    r.width = WIDTH
    r.height = HEIGHT
    r.fill = colors[:bg]
    r
  end
end
```

Step 2 - Make Chasm Mask



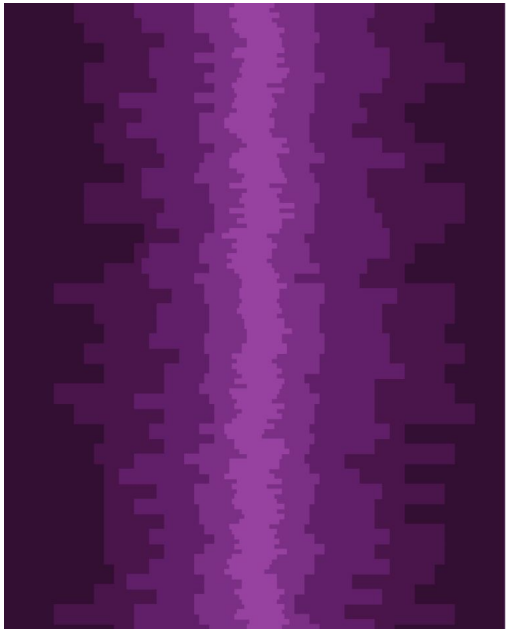
- This is the negative space between the two different sides.
- Make a bunch of random width rectangles in the middle x axis. The height will all be the same.
- Apply the perlin noise function to each rectangle from the center on the x-axis. (wave)
- Apply x-axis deviation (jitter)

Step 3 - Mask


```
ctx.rectangle do |r|
  r.x = 0
  r.y = 0
  r.width = WIDTH
  r.height = HEIGHT
  r.fill = colors[:levels][0]
  r.set_mask mask
  r
end
```

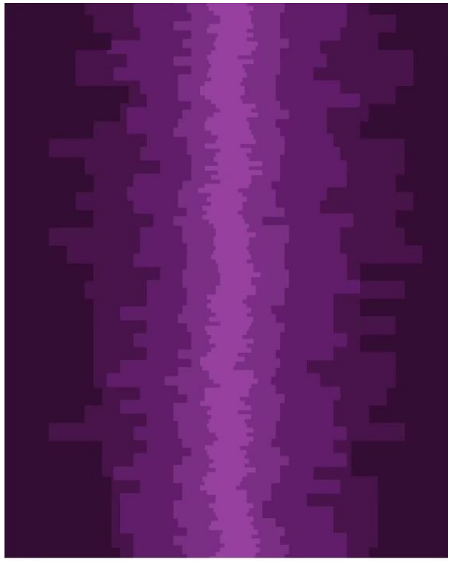
```
ctx.mask do |mask|
  mask.id = "center-rects"

  mask.rectangle do |r|
    r.x = 0
    r.y = 0
    r.width = WIDTH
    r.height = HEIGHT
    r.fill = "white"
    r
  end

  center_rects.each do |c_rect|
    mask.rectangle do |rect|
      rect.x = c_rect[:x]
      rect.y = c_rect[:y]
      rect.width = c_rect[:w]
      rect.height = c_rect[:h]
      rect.fill = "black"
      rect
    end
  end
  mask
end
```

Step 4 - Clean up chasm generation



- Improve the chasm generator to be cleaner, better centered, and more visually appealing
- Resize the base chasm rectangle height and width to match the depth of the level.
- Ensure the chasm itself gets smaller so we can see the other layers.
- Layer the results on top of each other, making a unique mask per layer.

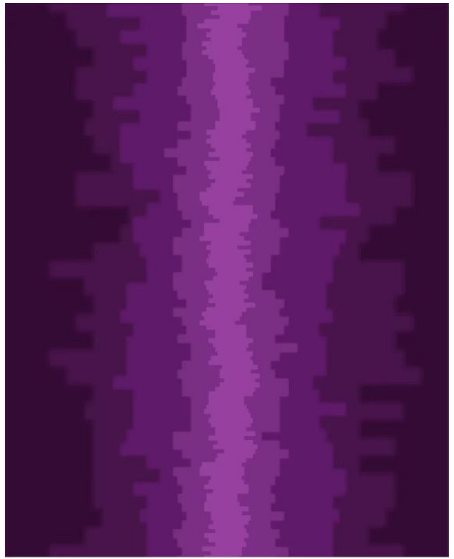
Step 5 - Animate!



- Use animate_motion to draw a simple path from 0,0 to 0, -screen height.
- Make an exact copy with use of each layer and paste it under the current frame.
- Change the duration to be longer the further back the layer is.
- Enjoy your parallax scrolling!

```
group.animate_motion do |anim|
  anim.duration = ((((3 - level)/3.0)*200) + 40)
  anim.duration_units = "s"
  anim.repeat_count = "indefinite"
  anim.mpath do |path|
    path.r_move(0, 0)
    path.r_line(0, -FRAME_HEIGHT - 1)
    path
  end
  anim
end

group.use(lvl_rect) do |use|
  use.y = (FRAME_HEIGHT) - 2
  use
end
```

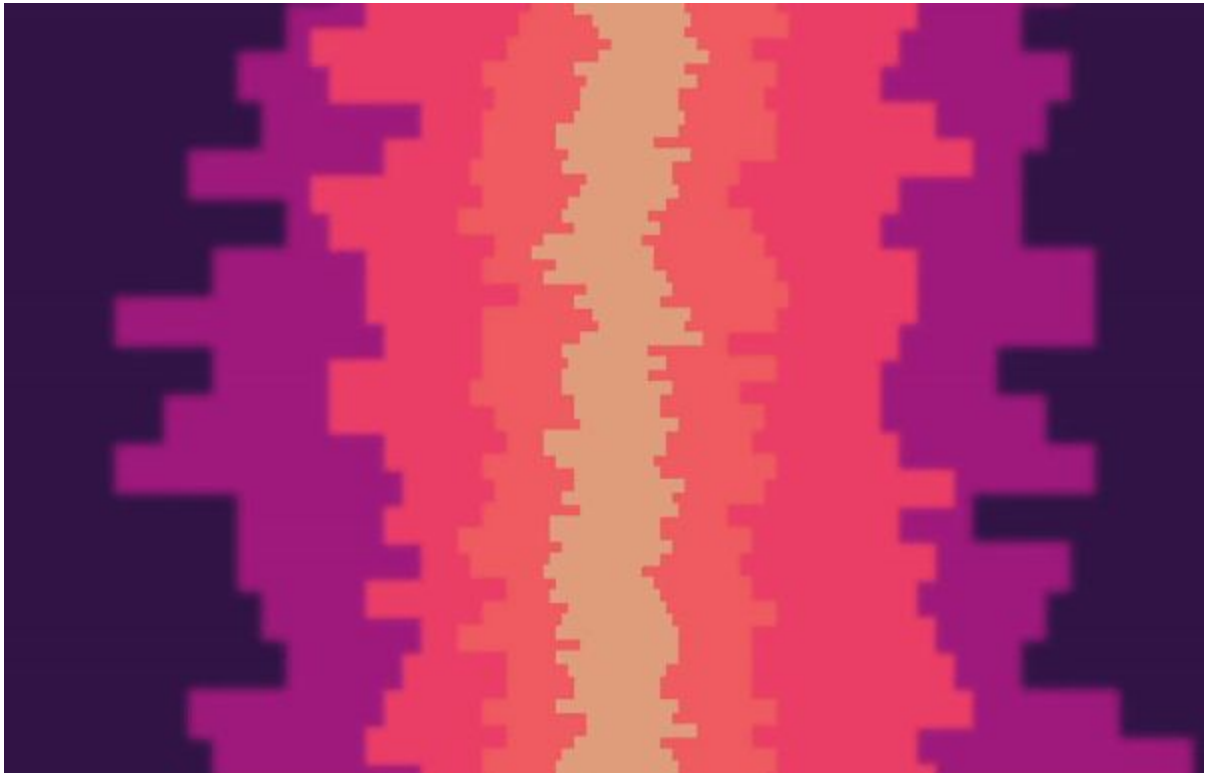
Step 6 - Filters

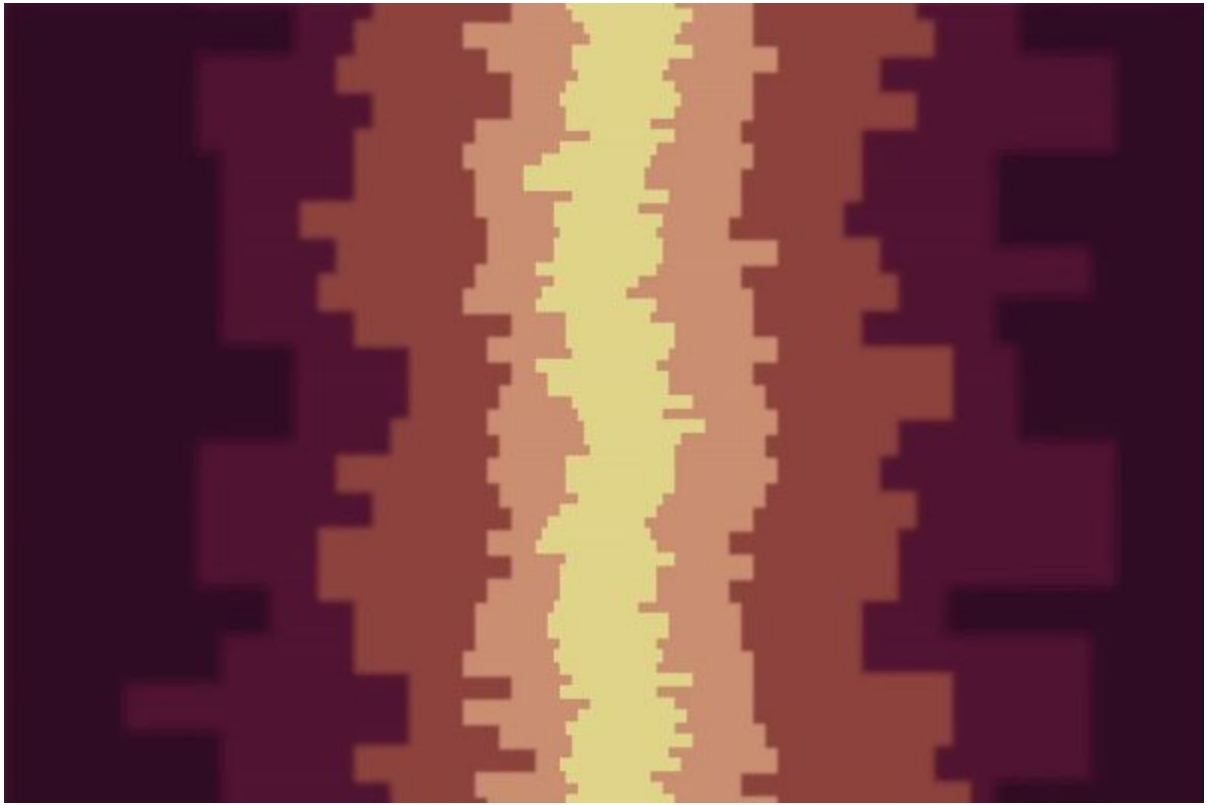


- I wanted to create a cool little DoF effect, to do this I need to use filters
- Define a filter for each layer, using blur
- Change the blur amount to be more blurry the closer it is.
- EZPZ DoF

```
MAX_LEVELS.times do |level|
  ctx.filter do |f|
    f.id = "filter-#{level}"
    f.blur do |b|
      b.input = Celestine::Filter::SOURCE_GRAPHIC
      b.standard_deviation = level
      b
    end
    f
  end
end
```

```
group.set_filter "filter-#{level}"
```





Links

Github: redcodefinal

Business or buy my art: ian@sol.vin

Email me about your thing: ian@0x42424242.in

Celestine: github.com/celestinecr/celestine

Docs: docs.celestine.dev

Thanks for watching!

Q & A