# Progressive Renderer & Noise Machine

10 Points

## Assignment Description

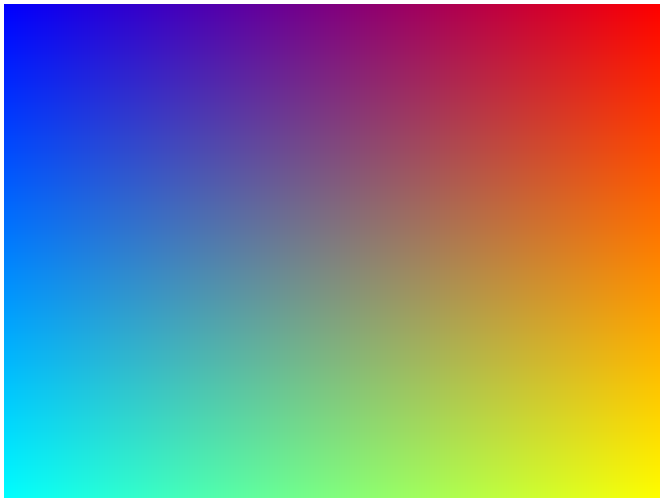
In this assignment you will familiarize yourself with the **`ProgressiveRenderer`** class, how to use it, and create your own **`NoisePattern`** class. Download the initial code base HERE.

## 1. RainbowRenderer Class

### 2 points

Examine the **`renderDemo.py`** file. Within there is a **`RandomRenderer`** class which extends the **`ProgressiveRenderer`**. There is also testing code at the bottom. Run the code to become familiar with what to expect. The **`RandomRenderer`** will fill in a random color for each chunk size rendered.

Following the **`RandomRenderer`** design, create a **`RainbowRenderer`** class in the same file. The color it will create is based on the x and y coordinates as a percentage of the total width and height. Set the red component to the horizontal percentage, the green component to the vertical percentage, and the blue component to 100% minus the horizontal percentage. You should see something that looks like the following image.

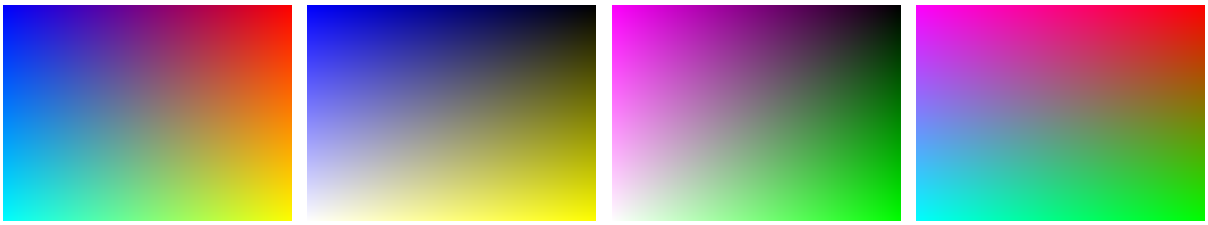


Override the **`handleOtherInput()`** method so that pressing the 1, 2, or 3 keys will advance each of the red, green, or blue components between the following visualization options. If the

key is pressed, wrap back around to the horizontal percentage. Keep the default behavior as described above.

horizontal percentage -> vertical percentage ->
100% - horizontal percentage -> 100% - vertical percentage

Below is a series of images of what it will look like if you press 1 repeatedly to change the R component to cycle through the behaviors as listed above. Make sure to call **`type(self).restart()`** after you change visualizations to restart the progressive renderer from the start.



# 2. NoisePatterns Class

## 6 points

Examine the **`noise.py`** module given to you in **`modules.utils`**. Inside is the **`NoiseMachine`** class, which we detailed during our lectures. Additionally, inside **`noise.py`** is a new class called **`NoisePatterns`**. In future projects you will use this **`NoisePatterns`** class in multiple locations, so the initial design of this class is a singleton design pattern. To visualize the instance methods, you will need to program Part 3, **`NoiseRenderer`**, in tandem to this part.

## Properties and Behaviors Provided in Code Base

### Singleton Pattern

A singleton design pattern is a way to set up a class such that only one instance is ever created from it. A class variable named **`_instance`**, initially set to **`None`**, is used as the universal reference to a single instantiation of the class. All code outside of the singleton class will use a class method named **`getInstance()`** which will return the class variable **`_instance`**. If **`_instance`** is **`None`** when **`getInstance()`** is called, **`_instance`** is set to a new **`NoisePatterns`** object. Outside of the class, do not use **`NoisePatterns()`** to gain access, but instead use **`NoisePatterns.getInstance()`**:

```
noisePat = NoisePatterns.getInstance()
noisePat.clouds(x, y)
```

### Instance Properties and Methods

These are set up in **`__init__()`**. A list of 5 **`NoiseMachines`**, each seeded differently, an integer representing the current **`NoiseMachine`** used in pattern creation, and an integer to scale the x and y coordinates. Any of your code which references the noise machines will reference via **`self.noiseID`**:

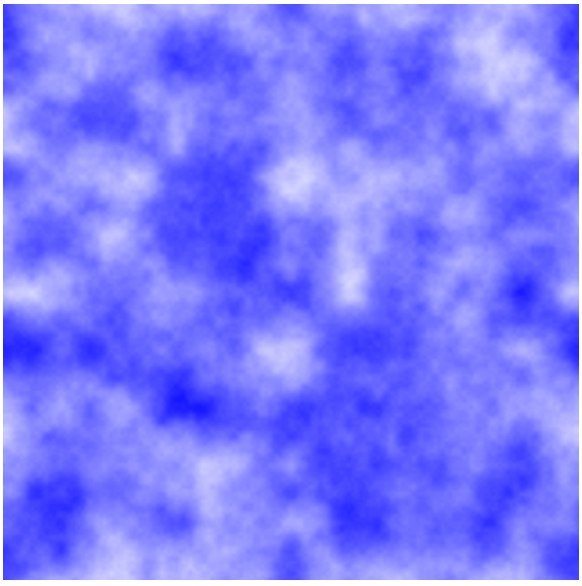
```
self.nms[self.noiseId].noise2d(...etc...)
```

Additionally, the methods **`self.next()`** and **`self.previous()`** are provided for you to cycle through which **`NoiseMachine`** to be used. You are also provided the **`self.clouds()`** method as an example to follow for the code you must create.

## Code to Create: Instance Methods

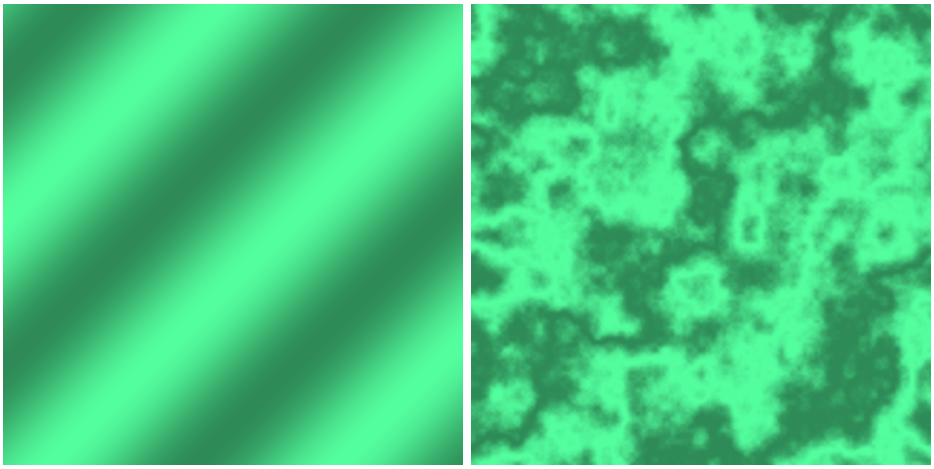
### cloudsTiled(self, x, y, xMod, yMod, c1, c2)

You are given the **`self.clouds()`** method already. Examine the code and create a new method, **`cloudsTiled()`**. At a given x and y, use the **`noise2dTiled()`** instead of the **`noise2d()`** method as a linear interpolation between colors **`c1`** and **`c2`**. Set the default colors for **`c1`** and **`c2`** to a blue and white color, respectively, see the **`definitions.py`** module. Return the interpolated color.



## marble(self, x, y, c1, c2, noiseStrength)

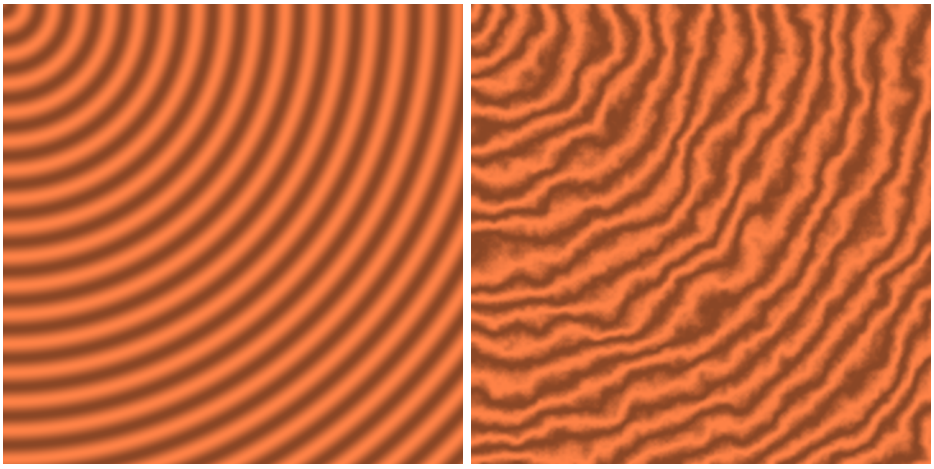
At a given **`x`** and **`y`**, obtain a **`noise2d()`** value. Then calculate the sine of **`x + y + noise * noiseStrength * scale`** to obtain a value between -1 and +1. Adjust the result to be a value between 0 and 1 and use the value to linearly interpolate between **`c1`** and **`c2`**, which is returned. Set the two color parameters to green and dark green by default, see the **`definitions.py`** module, and the **`noiseStrength`** to default 0.2.



noiseStrength=0 noiseStrength=0.2

## wood(self, x, y, c1, c2, noiseStrength)

At a given x and y, obtain a **`noise2d()`** value. Calculate the radius value as the square-root of x squared plus y squared. Then calculate the sine of **`radius + noise * noiseStrength * scale`** to obtain a value between -1 and +1. Adjust the result to be a value between 0 and 1 and use the value to linearly interpolate between c1 and c2, which is returned. Set the two color parameters to light brown and dark brown by default, see the **`definitions.py`** module, and the noiseStrength to default 0.2.



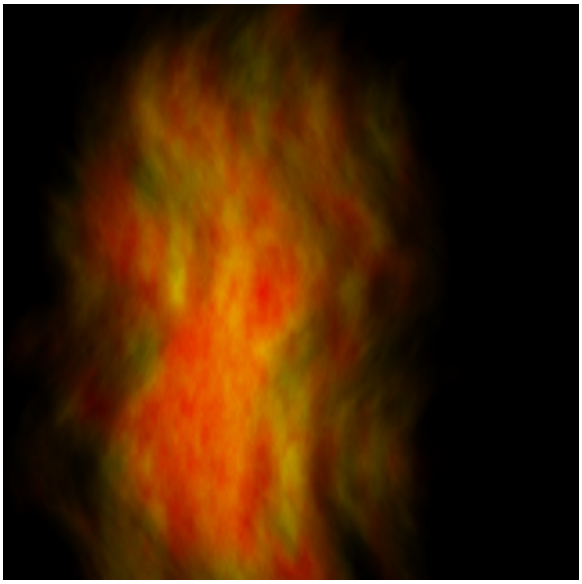
noiseStrength=0 noiseStrength=0.2

### fire(self, x, y, c1, c2, noiseStrength)

The fire noise has two parts: the fire shape+color and the wiggles. Divide the y-axis by 2 to stretch the shape into an oval instead of a circle.

To calculate the fire shape, get the color by obtaining a **`noise2d()`** value at **`x*2`** and **`y*2`** and using the noise to interpolate between c1 and c2 (red and yellow by default, see the **`definitions.py`** module). Calculate a **`radius`** about a midpoint by **`np.sqrt((x-xMiddle)**2 + (y-yMiddle)**2)/4`**.

For the wiggles, obtain a new **`noise2d()`** at **`x + sine(y * 2) * 0.5, y`**. Increase the **`radius`** value from before by **`(noise - 0.5) * noiseStrength`**. Then calculate a color multiplier **`s`** as **`1.0 - smerp(0.1, 1.0, radius)`**. Multiply the color from earlier by **`s`** and return.



# 3. NoiseRenderer Class

### 2 points

In the **`renderDemo.py`** file add a new class named **`NoiseRenderer`** which inherits from **`ProgressiveRenderer`**.

## Instance Properties

An integer representing the current noise pattern id and a list of **`NoisePatterns`** method references. The list will contain the methods themselves to be invoked later via the current id.

```
self.patterns[self.id](x, y)
```

Store the methods for **`clouds`**, **`tiledClouds`**, **`marble`**, **`wood`**, and **`fire`** in the list. Use lambda to add calls to **`marble`**, **`wood`**, and **`fire`** with **`noiseStrength`** set to 0.

## Instance Methods

### getColor(self, x, y, scale=64)

Overriding the **`ProgressiveRenderer`** behavior, divides x and y by the scale and then calls the current method of noise. While the methods in **`NoisePatterns`** return colors in 1.0 mode, the progressive renderer will multiply by 255 to satisfy PyGame.

### handleOtherInput(self, event)

Detect the keys **`q`** and **`w`** to either reduce or increase the current noise pattern id so you can cycle through many noise patterns. Use modulo to prevent index out of bounds errors and wrap back around to the start/end. Make sure to call **`type(self).restart()`** after you change patterns to restart the progressive renderer from the beginning.

Detect the keys **`e`** and **`r`** to call **`previous()`** or **`next()`** on the **`NoisePatterns`** instance. Make sure to call **`type(self).restart()`** to restart the progressive renderer from the beginning.