*Tutorial based on https://www.youtube.com/watch?v=f4s1h2YETNY&t=222s*

The first and most important thing to understand about writing shaders is that the code runs on the graphics card **per pixel**, and the code is extremely localized, meaning that the code is limited to the knowledge of the information of one pixel.

When it comes to Shadertoy shaders, the code is written in the glsl language, which is very similar to C, and the main function signature looks like this:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
```

The "`in vec2 fragCoord`" variable means that in terms of per pixel information to start with all we have is a fragment coordinate value, meaning a `Vec2` value from (0, ResolutionWidth) and (0, ResolutionHeight) denoting the pixel's position on the screen. But since this is different per pixel, this is all you need to create a huge range of effects. And then we output a vec4 fragColor value, which is a Vec4 of floats from 0 to 1, which encodes the final RGBA value sent to the screen for that pixel.

The second thing to understand about shader code is that a lot of the code you write has somewhat the opposite effect of sequential code : to move a shape to the left, you usually add a value, to move to the right, you subtract, to make a shape bigger you divide a value.

Before we begin in earnest, let's go over a mini example to express this point:

As a mini example before we start in full : Let's say we want to make a gradient from black to red horizontally across the screen. The starter shader code looks like this:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
      // Normalized pixel coordinates (from 0 to 1)
      vec2 uv = fragCoord/iResolution.xy;

      // Time varying pixel color
      vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

      // Output to screen
      fragColor = vec4(col,1.0);
}
```

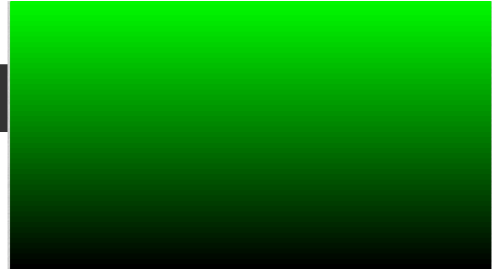
Note that the code doesn't explicitly return anything. The final pixel output is whatever `fragColor` is ultimately set to. Since uv l is fragCoord divided by resolution, the uv variable should be a vec2 of variables from 0 to 1, giving us positional coordinates that do not rely on the resolution of the screen (ie clip space). Delete the `col` line and replace the fragColor line with:

```
fragColor = vec4(uv.x, 0.0, 0.0, 1.0);
```

then press the play button in the bottom left of the code to compile.

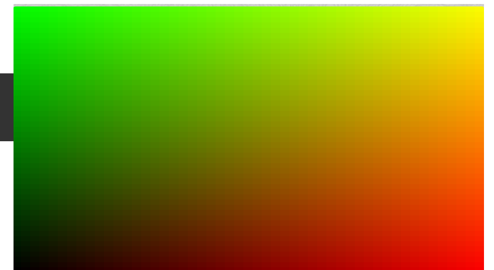
Note that you have to add the decimal parts to constant numbers (so 1.0 instead of 1, 0.0 instead of 0, etc), or else glsl will read the value as an int and will not compile. Now let's say we want to create a vertical gradient from black to green to black across the screen. Set

```
fragColor = vec4(0.0, uv.y, 0.0, 1.0);
```



Now let's combine the two with:

```
fragColor = vec4(uv.x, uv.y, 0.0, 1.0);
```



Intuitively, this output should make sense. On the bottom right the x coordinate is 0 and the right is 1, getting us a `fragColor` value of (1, 0, 0), or red, the top left has an x coordinate of 0 and y coordinate of 1, getting us (0, 1, 0), or green, and on the top right the x and y values are 1, getting us (1, 1, 0), or yellow, and every pixel in between is some mix of red and green.
*(Maybe take break here and make sure everyone is up to speed)*

Now a lot of times to have our shader be centered in the middle of the screen it is helpful to have our uv coordinates range from -1 to 1 instead of 0 to 1. To change to this range we simple set uv to

```
vec2 uv = (fragCoord/iResolution.xy * 2.0) - 1.0;
```

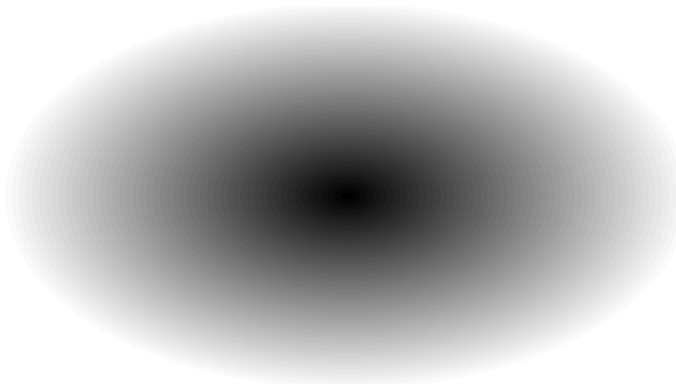
So now our uv coordinates are in the range -1 to 1 and uv coordinate 0, 0 is at the center of the screen.
Now let's begin writing the proper shader code.
To begin, we want the pixel's distance from the center of the screen as a variable . To do that, we can use glsl's `length()` function, and because the origin is at the center of the screen already, we do not have to modify our uv position at all. Before we set `fragColor`, write

```
float d = length(uv);
```

Now to view the effect of this, set fragColor to

```
fragColor = vec4(d, d, d, 1.0);
```

And we should see a sort of squashed radial gradient like so :



The reason it's squashed should make sense : u and v are both from 0 to 1 even though our x resolution > y resolution, meaning that for some change in u we go less across the screen horizontally than for the same change vertically with v. To fix this and get coordinates that have a more intuitive range, when setting uv let's change it to:

```
vec2 uv = (fragCoord/iResolution.y * 2.0) - (iResolution.xy/iResolution.y);
```

Getting us:

Our `uv.y` coordinate should be the same, but now our uv.x coordinates are in some range -a to a, where a > 1. It's fine that we don't quite know what a is, all that matters is that the origin of `uv` is at the center of the screen and the coordinates are normalized based on screen resolution.

Now let's subtract `d` by 0.5:

```
float d = length(uv) - 0.5;
```

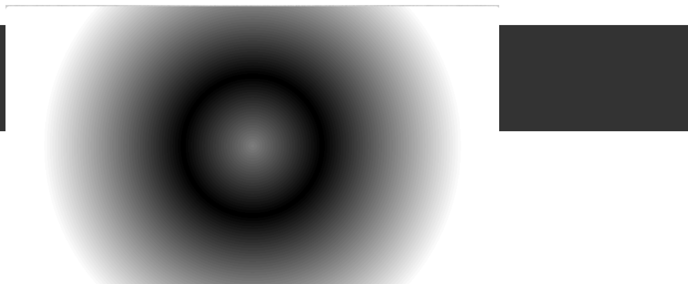
What d represents now is an signed distance field, or sdf, of a circle with radius 0.5. If you remember sdf's from lecture, a sdf is a function that takes in a position as input and returns a signed distance from that position to some shape. The distance is signed because it is positive if the position is outside the shape, and negative if it is inside.

***Check in**: the code should now look like this:)*

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
	// Normalized pixel coordinates (from 0 to 1)
	vec2 uv = (fragCoord/iResolution.y * 2.0) -
(iResolution.xy/iResolution.y);

	float d = length(uv) - 0.5;

	// Output to screen
	fragColor = vec4(d, d, d, 1.0);
}
```

So values outside the circle are positive and negative inside. With this being the case, let's take the absolute value of d:

```
float d = length(uv) - 0.5;
d = abs(d);
```



Now let's make a somewhat sharper gradient with the `smoothstep` function. This function compares a value x to some threshold values a and b. If x < a, then it returns 0, and if x > b, it returns 1. For a <= x <= b, the function linearly interpolates from 0 to 1.

Let's use step on d with a thresholds of 0.0 and 0.1,

```
float d = length(uv) - 0.5;
d = abs(d);
d = smoothstep(0.0, 0.1, d);
```

All put together this means all pixel distances from a circle of radius 0.5 greater than 0.1 are colored as white, and all distances from 0 to 0.1 interpolate from black to white.

Now to create a radial repetition of the rings, let's apply `sin` to create a value that oscillates between -1 and 1 based on distance, and let's multiply by some value like 8 to increase the frequency and then divide again by 8 to normalize the value:

```
float d = length(uv) - 0.5;
d = sin(d*8.0)/8.0;
d = abs(d);
d = smoothstep(0.0, 0.1, d);
```

Now let's take this shader to the next level and animate the shader, making it change based on time. Shadertoy gives us the `iTime` variable, which is a float that describes the number of seconds since starting to render the code. Let's add that to `d` inside `sin` to modify the phase of the sin wave based on time , creating an effect of the rings shrinking towards the center:

```
float d = length(uv) - 0.5;
d = sin(d*8.0 + iTime)/8.0;
d = abs(d);
d = smoothstep(0.0, 0.1, d);
```

Now let's start on getting some more appealing colors for this shader, starting by changing the interpolation of the distances to match more with neon lighting. First, get rid of the `smoothstep`:

```
float d = length(uv) - 0.5;
d = sin(d*8.0 + iTime)/8.0;
d = abs(d);
//d = smoothstep(0.0, 0.1, d); /* or delete */
```

And then make d the inverse of a small value like 0.02.

```
float d = length(uv) - 0.5;
d = sin(d*8.0 + iTime)/8.0;
d = abs(d);
d = 0.02 / d;
```

This creates a neon effect because the inverse function (1/x) has smaller and smaller falloff as x approaches infinity, creating glow as values are slow to approach 0, or black. We have a small number as the numerator so `d` stays relatively close to the 0 to 1 range.

***Check** in: your code should look close to this:*

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
	// Normalized pixel coordinates (from 0 to 1)
	vec2 uv = (fragCoord/iResolution.y * 2.0) - (iResolution.xy/iResolution.y);

	float d = length(uv) - 0.5;

	d = sin(d*8.0 + iTime)/8.0;
	d = abs(d);
	d = 0.02/d;

	// Output to screen
	fragColor = vec4(d, d, d, 1.0);
}
```

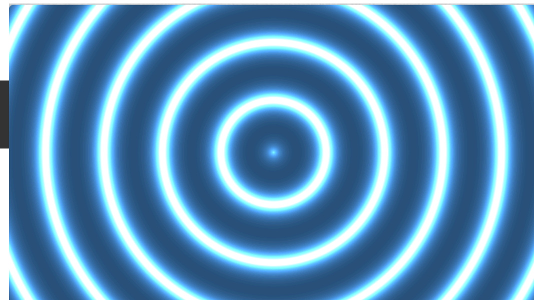
Now let us modify the hue of these neon rings. Let's create a new `vec3` called `col` right after we define d. As example, let's set it to red, and then multiply `col` by `d`:

```
float d = length(uv) - 0.5;

vec3 col = vec3(1.0, 0.0, 0.0);

d = sin(d*8.0 + iTime)/8.0;
```

```
d = abs(d);
d = 0.02/d;

col *= d;

fragColor = vec4(col, 1.0);
```

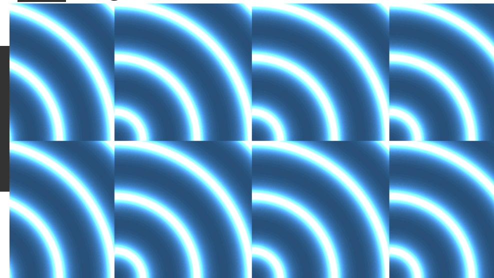
We can create more intense neon effects by setting col to `vec3` values greater than 1, like a very intense blue:

```
vec3 col = vec3(1.0, 2.0, 3.0);
```



Now let's introduce **spatial repetition,** meaning that we repeat this shader multiple times across the screen. To do this, let's use the `fract` function, which returns the fractional or decimal part of the input (ie 3.25 → 0.25). It's the same as `x % 1.0.` Using `fract` on `uv` we get:

```
vec2 uv = (fragCoord/iResolution.y * 2.0) -
(iResolution.xy/iResolution.y);
uv = fract(uv);
```

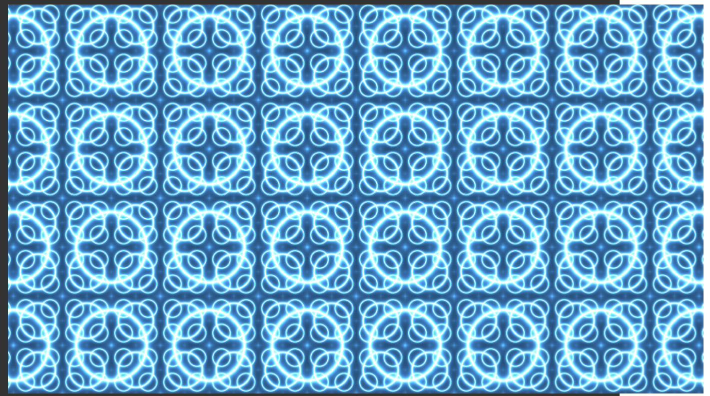


We get these very harsh cutoffs for the repetitions because `fract` is cutting off the `uv` to the 0 to 1 range, when instead they are in to (-1 to 1) and (-x to x) range. To fix this, we center the uv's like before:

```
vec2 uv = (fragCoord/iResolution.y * 2.0) -
(iResolution.xy/iResolution.y);
uv = fract(uv*2.0) - 0.5;
```

Now let's introduce **scale repetition** by enclosing the uv and d calculation code within a for loop, and initializing a `vec3 finalColor` and `float NUM_ITERATIONS`. We loop through the code `NUM_ITERATIONS` times, accumulating `finalColor` based on `d`, making sure to divide the code by `NUM_ITERATIONS` at the end to make sure the colors are more or less normalized.

```
// Normalized pixel coordinates (from 0 to 1)
vec2 uv = (fragCoord/iResolution.y * 2.0) - (iResolution.xy/iResolution.y);

vec3 finalColor = vec3(0.0, 0.0, 0.0);
float NUM_ITERATIONS = 3.0;
for(float i = 0.0; i < NUM_ITERATIONS; i ++) {
      uv = fract(uv*2.0) - 0.5;

      float d = length(uv) - 0.5;

      vec3 col = vec3(1.0, 2.0, 3.0);

      d = sin(d*8.0 + iTime)/8.0;
      d = abs(d);
      d = 0.02/d;

      col *= d;

      finalColor += col/NUM_ITERATIONS;
}

// Output to screen
fragColor = vec4(finalColor, 1.0);
```

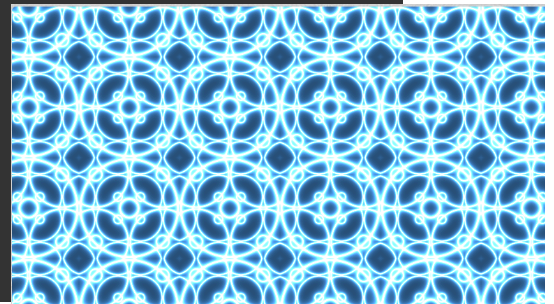


To get a bit more asymmetrical results let's multiply uv by 1.5 instead of 2.0 in the fract function:

```
uv = fract(uv*1.5) - 0.5;
```

***Check in*** *: Code should look pretty much like this:*

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
      // Normalized pixel coordinates (from 0 to 1)
      vec2 uv = (fragCoord/iResolution.y * 2.0) -
(iResolution.xy/iResolution.y);

      vec3 finalColor = vec3(0.0, 0.0, 0.0);
      float NUM_ITERATIONS = 3.0;
```



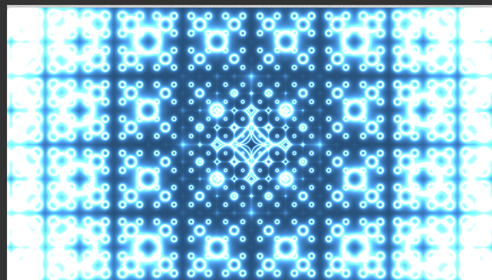
```
    for(float i = 0.0; i < NUM_ITERATIONS; i ++) {
        uv = fract(uv*1.5) - 0.5;

        float d = length(uv)- 0.5;

        vec3 col = vec3(1.0, 2.0, 3.0);

        d = sin(d*8.0 + iTime)/8.0;
        d = abs(d);
        d = 0.01/d;

        col *= d;

        finalColor += col/NUM_ITERATIONS;
    }

    // Output to screen
    fragColor = vec4(finalColor, 1.0);
}
```

To start to get a bit more large-scale, global detail, we can keep track of the original, non scaled uv's in a variable `vec2 uv0`

```
vec2 uv = (fragCoord/iResolution.y * 2.0) - (iResolution.xy/iResolution.y);
vec2 uv0 = uv;
```

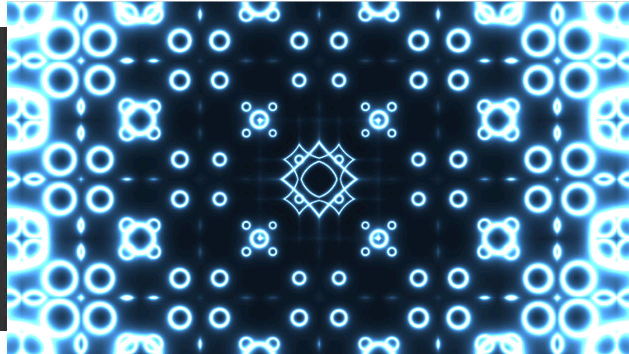
We can have the color intensity falloff from the center by scaling `d` with the function `exp(x),` which is like a smoother version of the inverse function, using `-length(uv0)` as input. We then modify the numerator of the neon inverse calculation to lower the intensity:

```
float d = length(uv)*exp(-length(uv0));

vec3 col = vec3(1.0, 2.0, 3.0);

d = sin(d*8.0 + iTime)/8.0;
d = abs(d);
d = 0.01/d;
```



We can also increase the contrast of the image using the pow function, which widens the gap between black and white areas of the shader.

```
d = sin(d*8.0 + iTime)/8.0;
d = abs(d);
d = 0.01/d;

d = pow(d, 1.3);

col *= d;
```



As a final step we can add color palette variation by introducing the function palette, which takes in parameters to represent r,g,b signals as sign waves and samples the signals with a value t. This function is sourced from https://iquilezles.org/articles/palettes/

```
vec3 palette(float t) {
    vec3 a = /*?*/;
    vec3 b = /*?*/;
    vec3 c = /*?*/;
    vec3 d = /*?*/;
    return a + b*cos( 6.283185*(c*t+d));
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {...}
```

We can experiment with what parameters make good palettes with this website here, http://dev.thi.ng/gradients/

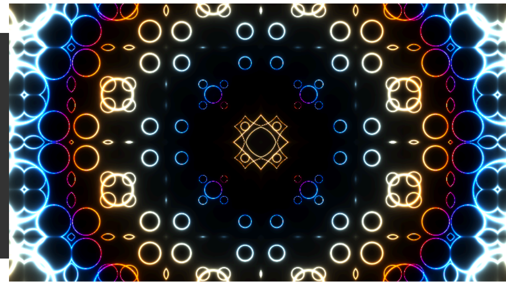
*If time could give students time to find their own palette and substitute values in for vec3 a, b, c, and d.*

For this example I'll use values from a palette on the original color article:

```
vec3 palette(float t) {
     vec3 a = vec3(0.5, 0.5, 0.5);
     vec3 b = vec3(0.5, 0.5, 0.5);
     vec3 c = vec3(1.0, 1.0, 1.0);
     vec3 d = vec3(0.0, 0.1, 0.2);
     return a + b*cos( 6.283185*(c*t+d));
}

void mainImage( out vec4 fragColor, in vec2 fragCoord ) {...}
```

Now let’s modify `col` to a color sampled from the palette based on the length of `uv0`.

```
float d = length(uv)*exp(-length(uv0));

vec3 col = palette(length(uv0));

d = sin(d*8.0 + iTime)/8.0;
```



Finally we can use **time based modification** and **scale repetition** on the palette sampling as well, taking into account the iterator `i` and time `iTime`, scale by some value to offset the timing from the shape scaling. We can also get rid of the division of the finalColor since our palette does not use super overblown colors.

```
uv = fract(uv*1.5) - 0.5;

float d = length(uv)*exp(-length(uv0));

vec3 col = palette(length(uv0) + 0.4*i + 0.4*iTime);

d = sin(d*8.0 + iTime)/8.0;
d = abs(d);
d = 0.01/d;

d = pow(d, 1.3);

col *= d;

finalColor += col;
```

*Final check in: your code should look something like this:*

```
vec3 palette(float t) {
	vec3 a = vec3(0.5, 0.5, 0.5);
	vec3 b = vec3(0.5, 0.5, 0.5);
	vec3 c = vec3(1.0, 1.0, 1.0);
	vec3 d = vec3(0.0, 0.1, 0.2);
	return a + b*cos( 6.283185*(c*t+d));
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
	// Normalized pixel coordinates (from 0 to 1)
```

```
      vec2 uv = (fragCoord/iResolution.y * 2.0) -
(iResolution.xy/iResolution.y);
      vec2 uv0 = uv;

      vec3 finalColor = vec3(0.0, 0.0, 0.0);
      float NUM_ITERATIONS = 3.0;
      for(float i = 0.0; i < NUM_ITERATIONS; i ++) {
           uv = fract(uv*1.5) - 0.5;

          float d = length(uv)*exp(-length(uv0));

          vec3 col = palette(length(uv0) + 0.4*i + 0.4*iTime);

          d = sin(d*8.0 + iTime)/8.0;
          d = abs(d);
          d = 0.01/d;

          d = pow(d, 1.3);

          col *= d;

          finalColor += col;
      }

      // Output to screen
      fragColor = vec4(finalColor, 1.0);
}
```