

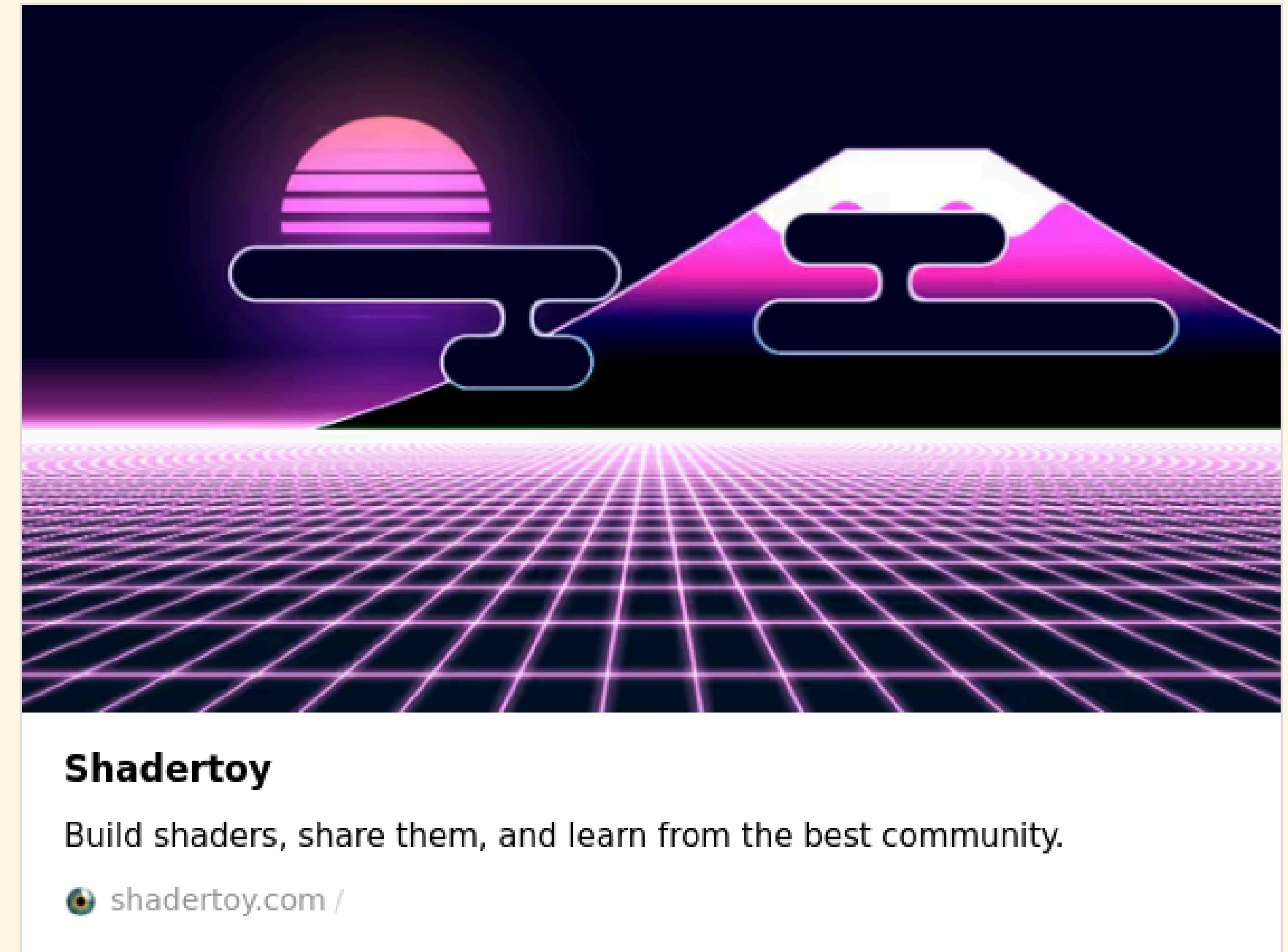


GLSL

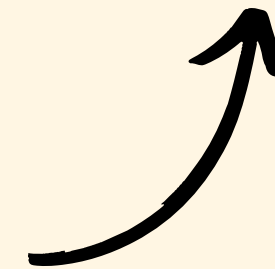
OpenGL Shader Language

WHAT IS A SHADER?

A shader is code that will calculate positions and colors to determine the pixel color on a screen. In short, it is code to render an image.



Click me to learn more!



LETS MAKE A SHADER!

Let's start by making an
account on ShaderToy.
<https://www.shadertoy.com/>



Shadertoy

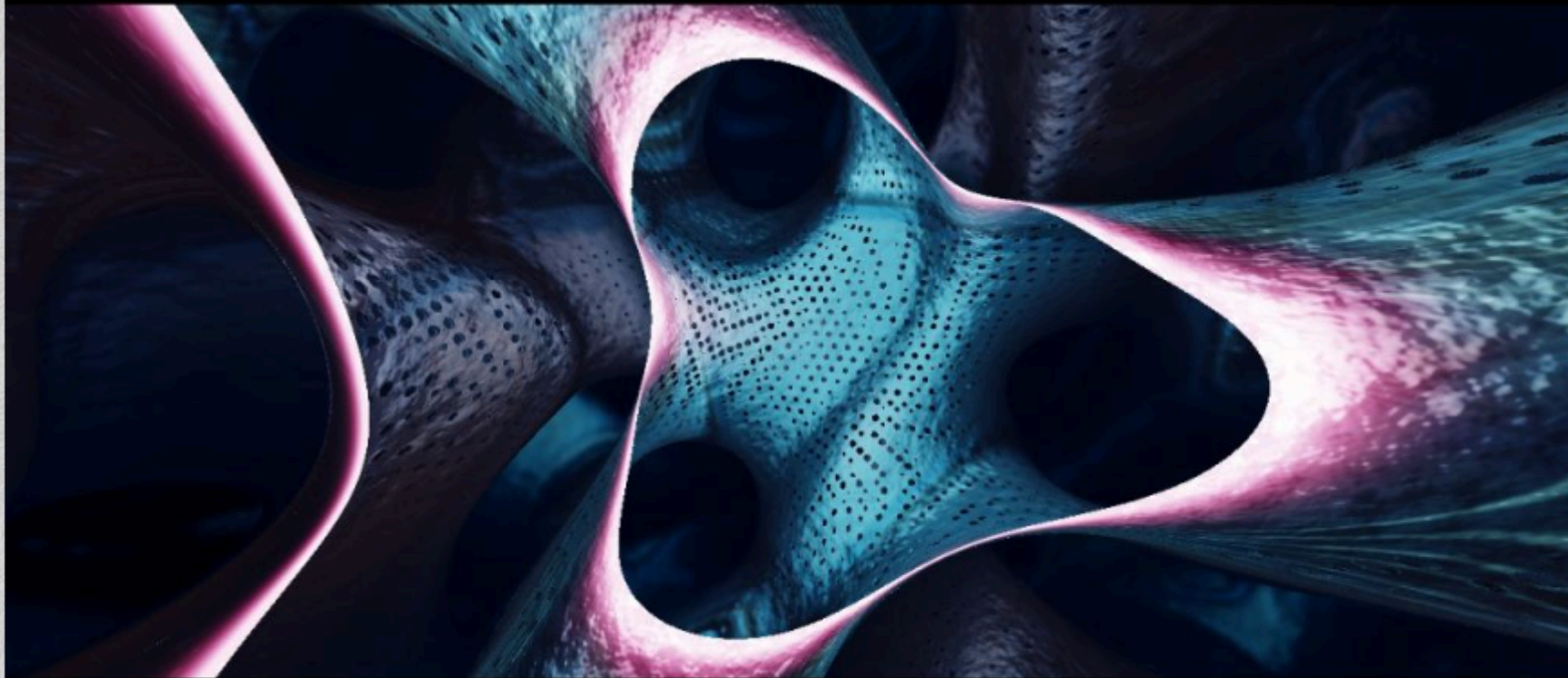
Browse

New

Sign In

Shader of the Week

Warning



Alien Tunnel by Iz

👁 4345 ❤ 58

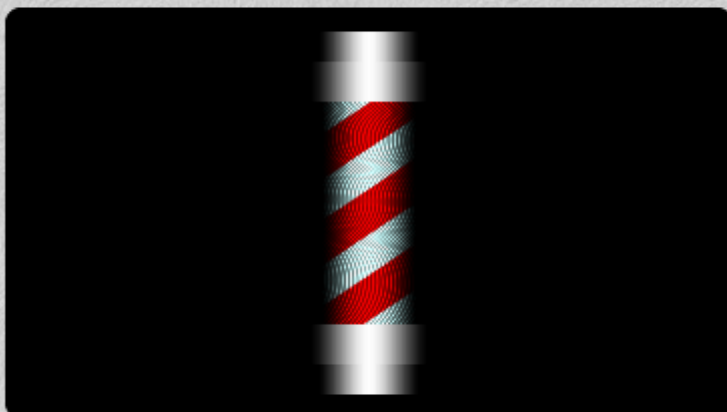
Build and Share your best shaders with the world and get Inspired

 [Donate](#)

[Become a patreon](#)

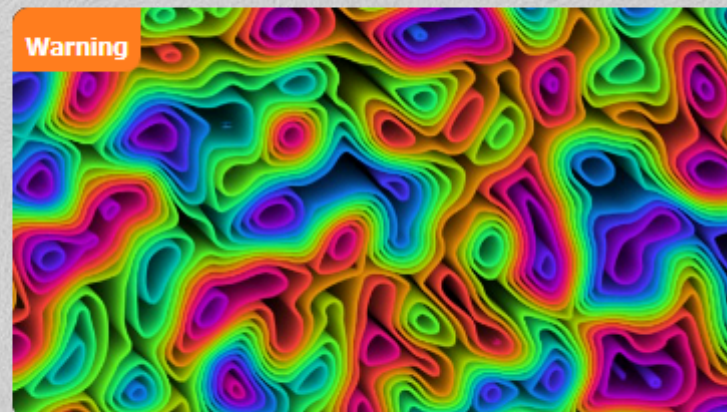
Latest contributions: "flowery thing" by [jonasfrey](#) 50 minutes ago, "jul202024" by [brendanluu](#) 1 hour ago, "Cheap 2d Rain" by [Elsio](#) 2 hours ago, "Plain Walker" by [SnoopethDuckDuck](#) 2 hours ago, "Belousov-Zhabotinsky CA" by [draradech](#) 2 hours ago

Featured Shaders



Barber by [okro](#)

👁 13823 ❤ 22



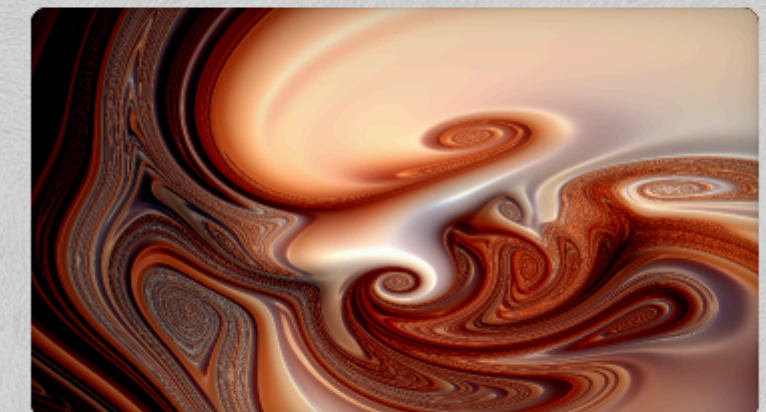
Warning
isovalues 3 by [FabriceNeyret2](#)

👁 28938 ❤ 256



Raymarched Hexagonal Truchet by [Shane](#)



👁 28897 ❤ 253






Iterations - inversion 2 by [iq](#)

👁 26008 ❤ 109



85.23164.9 fps800 x 450



GLSL!

Orpheus

Heidi ate my dinner

private

Submit

Create a new shader, give it a name, tag, and description! (Be creative...)

THIS IS WHAT YOU WILL SEE!

When dealing with shaders, we think in pixels. In this case, every pixel is calculated as a different color!



```
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.xy*vec3(0,2,4));

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

LET'S START FROM THE BEGINNING

Delete everything inside void mainImage!

There are a couple parameters that we can see, vec4 is a vector with 4 different components, and vec2 is a vector with 2 different components. We can see that there is an input of fragCoord, and an output of fragColor. In this case, we are stating that the position of the fragment is the input, and we are going to write code to show the output of that fragment or pixel.



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

INITIALIZE THE COORDINATE PLANE

Here, we are creating a new vector, with 2 components named uv. As we know, XYZ are axes of 3D coordinate systems, but in our case we will be dealing with 2D, with U being horizontal (X) and V being vertical (Y).

Since everyone has different screen resolutions, we are going to initialize the vector with a coordinate plane.

```
fragCoord/iResolution.xy;
```

This states that the bottom left corner will be (0,0), and the top right corner will be (1,1).

Copy this code into mainImage!

```
vec2 uv = fragCoord/iResolution.xy;
```



```
void mainImage( out vec4 fragColor, in vec2 fragCoord )  
{  
    vec2 uv = fragCoord/iResolution.xy;  
}
```

DISPLAYING COLORS

fragColor is a vec4 vector, meaning it consists of 4 components. These components are (R,G,B,a)! In our case, we will not worry about the last value. Try changing these values and run the code. What do you see?

Copy this code into mainImage!

```
fragColor = vec4( 0.0, 0.0, 0.0, 1.0 );
```



```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

    fragColor = vec4( 0.0, 0.0, 0.0, 1.0 );
}
```


DISPLAYING COLORS

Lets try to create a gradient!

How do you think we could increment the color as it goes higher?

Hint: as uv.x gets higher, RGB gets higher

The solution to this is to simply place uv.x or uv.y in the fragColor vector!

As you can see, the higher the x value, the higher the red value is!

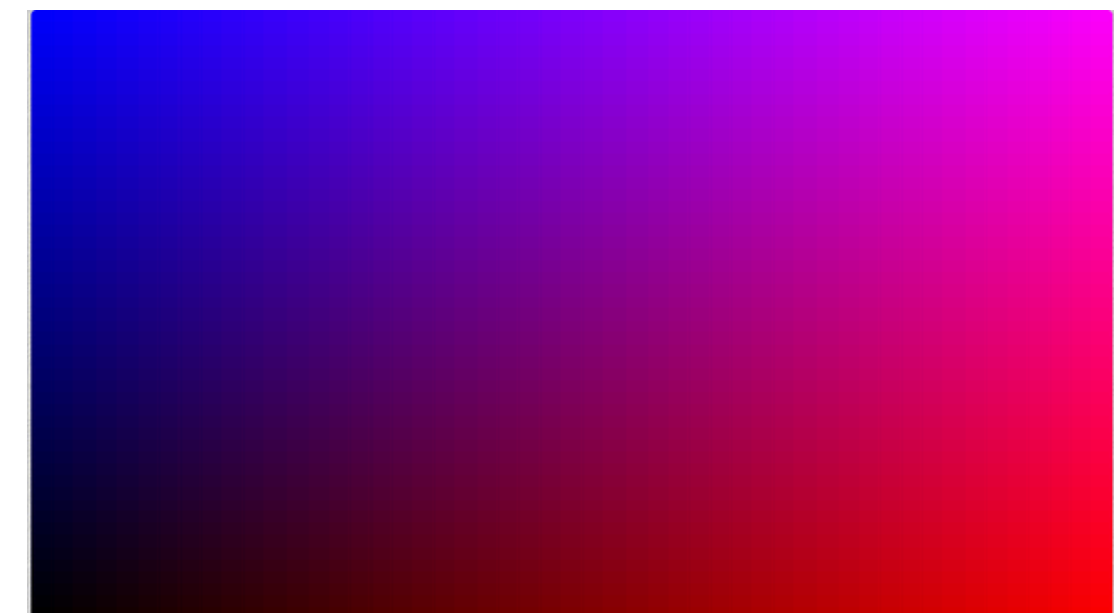
We can do multiple gradients upon multiple axes as such:

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

It is visible that there is a blue gradient going up, and a red gradient going right.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

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



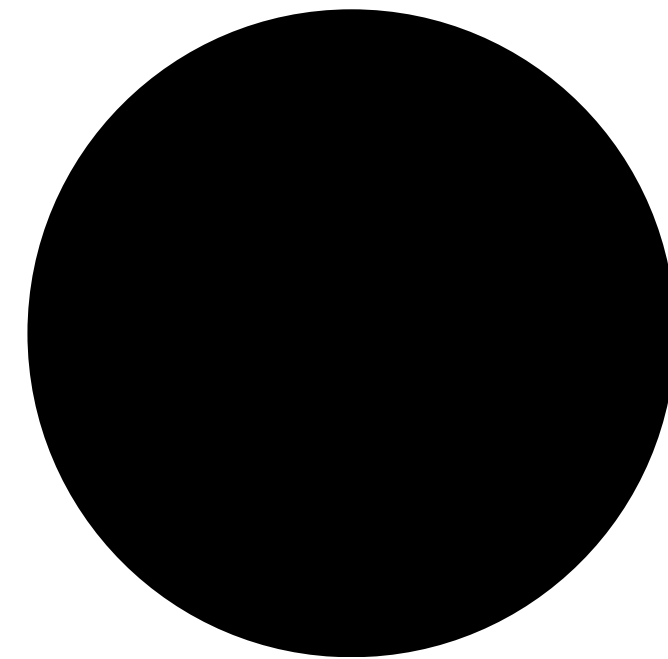
SHAPES

Lets display a circle...

If we think about it, if vector uv is at a random point, we should be able to make a circle around the center with some if statements.

Try doing it on your own! The solution is on the next slide.

Hint: the `length()` function takes any coordinate and returns the length away from (0,0).



Make me!



SHAPES

In order to display a shape, we will take the length of the current pixel (vec2 uv) and see if it is bigger than 1. Here, I have created a vec3 vector that is initialized as col, to initialize all of them as 0.0. When the length of uv exceeds 1, the vector will turn into 1.0. We can put col as the RGB values of the output, as when they are all 1, it will display white. Run this, anything strange happen?

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

    vec3 col = vec3(0.0);

    if(length(uv) > 1.0)
    {
        col = vec3(1.0);
    }

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

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

    vec3 col = vec3(0.0);

    if(length(uv) > 1.0)
    {
        col = vec3(1.0);
    }

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

SHAPES

You should see something like this on the output. This isn't a circle!

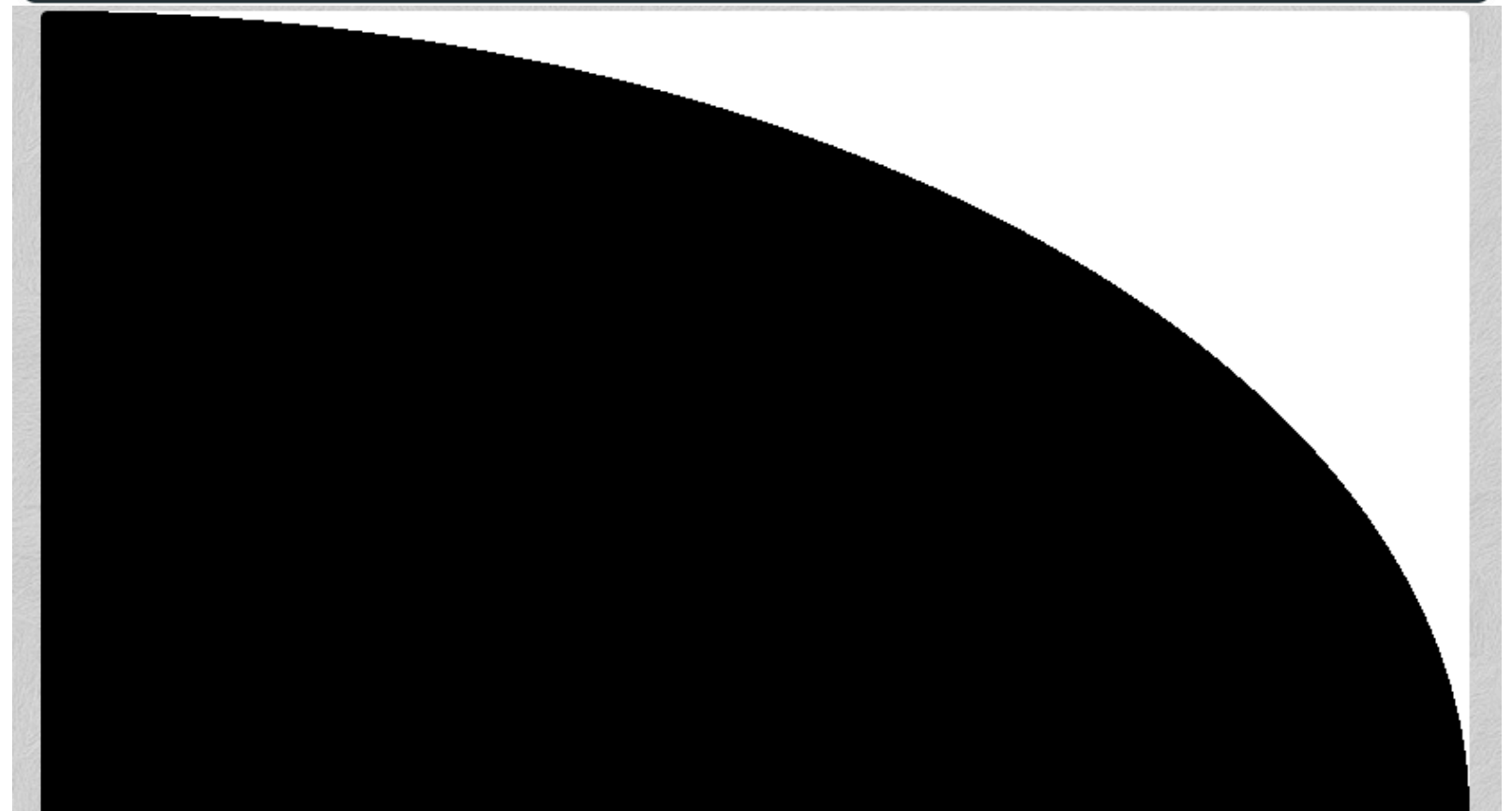
Recall that we initialized the coordinate plane as (0,0) to be in the bottom left, and (1,1) in the top right. In order to change this, let's shift the image 0.5 pixels right, and up.

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

    vec3 col = vec3(0.0);

    if(length(uv) > 1.0)
    {
        col = vec3(1.0);
    }

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



SHAPES

We will do this by changing uv.x to -0.5 and uv.y to -0.5. This will center the image. We will also change the radius to 0.5 instead of 1.0 since the radius is 0.5 when the origin is at the center.

Still doesn't look quite right...

Recall that we initialized each of them to be 1 unit tall and wide, so the resolution is ignored. To fix this, we will add this line of code:

```
uv.x *= iResolution.x / iResolution.y;
```

All we are doing is multiplying the X value of the coordinate by the aspect ratio of the screen!

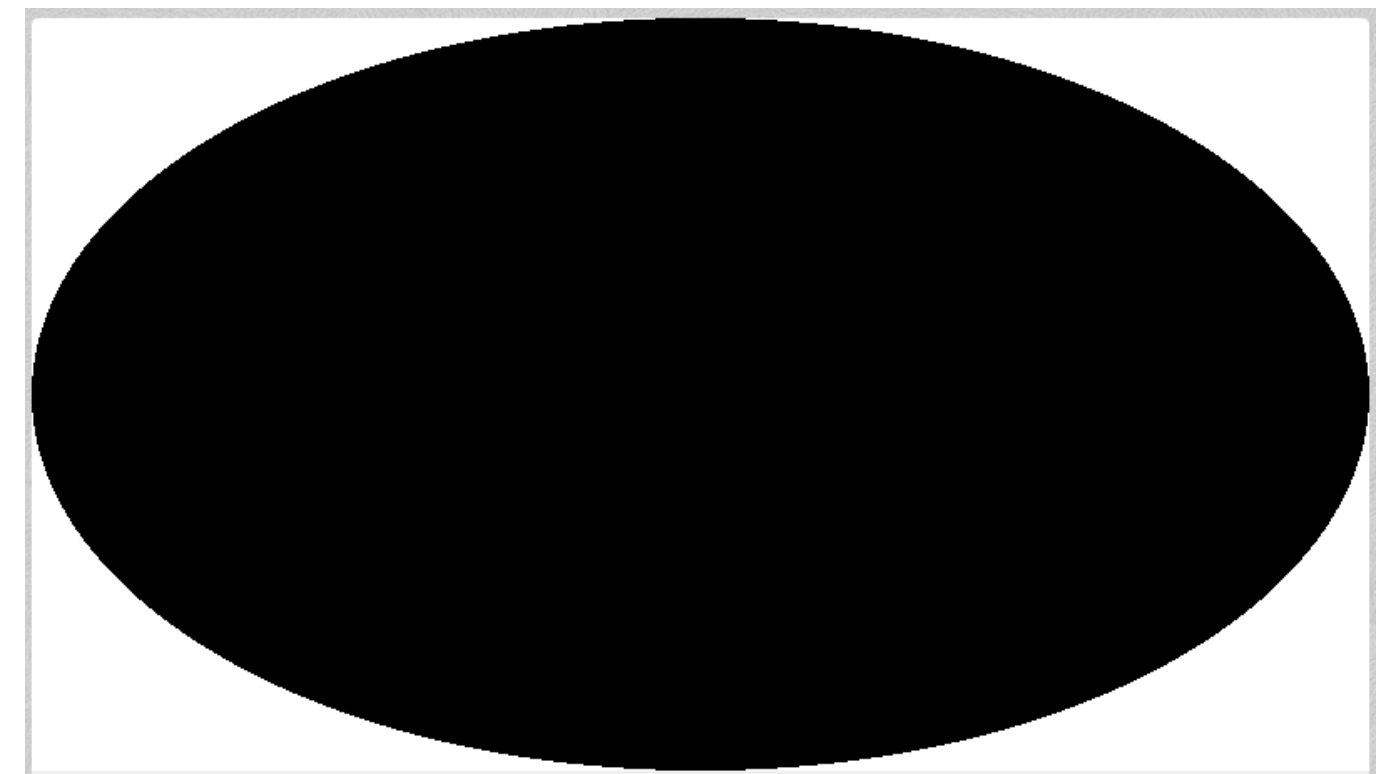
```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

    uv.x = uv.x - 0.5;
    uv.y = uv.y - 0.5;

    vec3 col = vec3(0.0);

    if(length(uv) > 0.5)
    {
        col = vec3(1.0);
    }

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



SHAPES

Finally! Something that resembles a circle.

We can simplify the code that modifies the aspect ratio and coordinates:

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

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord/iResolution.xy;

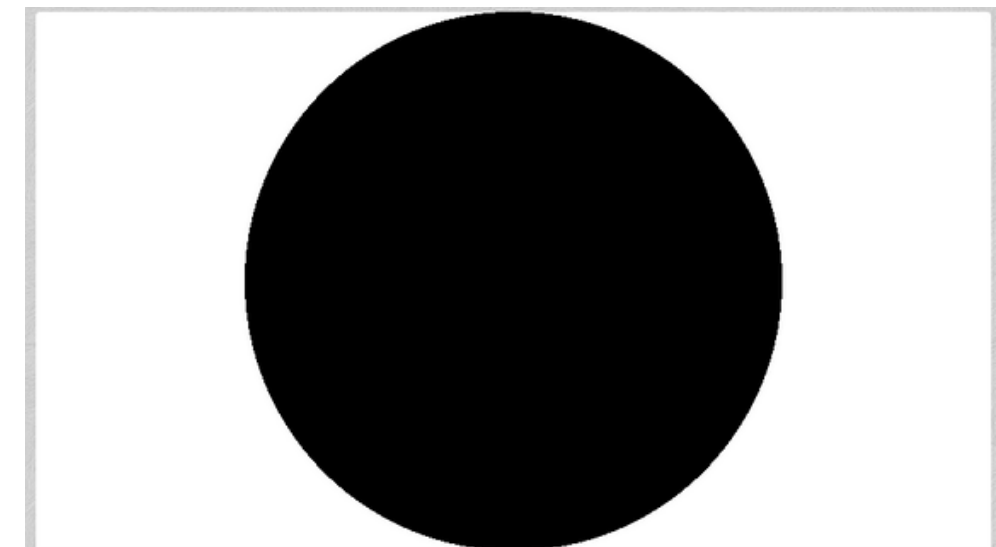
    uv.x = uv.x - 0.5;
    uv.y = uv.y - 0.5;

    uv.x *= iResolution.x / iResolution.y;

    vec3 col = vec3(0.0);

    if(length(uv) > 0.5)
    {
        col = vec3(1.0);
    }

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



MOVEMENT AND TIME

With an introduction into shapes, lets try to create something that moves. There are many variables that can change the movement of a render, but we will focus on `iTime` here.

`iTime` is a value that will give a variable of the time in seconds.

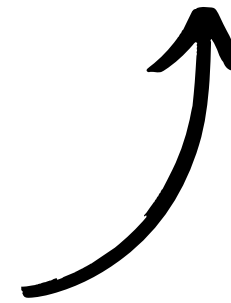
We can use this in conjunction with some of the trigonometric functions: `sin()`, `cos()`, and `tan()`.

Lets try! The challenge will be to make a circle go bigger and smaller. Answer on next slide!

Hint: Use a `sin()` function and put `iTime` inside of it!



Make me!



MOVEMENT AND TIME

Here is the answer!

We will first initialize the uv coordinate system like before, then create a floating decimal point variable called dist. This is the distance between the center and the pixel. Then, we will create another float variable named radius. This will be further explained in the next slide. After that, another float named circle is created. We will add a smoothstep here to create smoothing around the circle. You also have the option of replacing smoothstep with step to have no smoothing. You will also need to remove the middle variable. Lastly, we will get the negative of the circle, and lastly output the fragColor.

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = (fragCoord * 2.0 - iResolution.xy) / iResolution.y;

    float dist = length(uv);

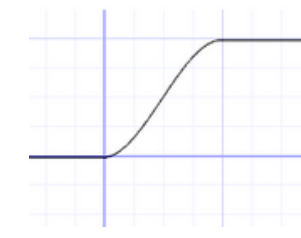
    float radius = 0.5 * (1.0 + sin(iTime));

    float circle = smoothstep(radius, radius + 0.05, dist);

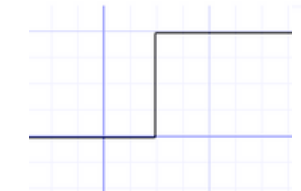
    circle = 1.0 - circle;

    fragColor = vec4(vec3(circle), 1.0);
}
```

A smoothstep looks like this!



While a step looks like this



We can use smoothsteps to round out edges by stretching the curve out.

MATHEMATIC EQUATIONS

As some of you may have learned, this line may look similar to a trigonometric function.

```
float radius = 0.5 * (1.0 + sin(iTime));
```

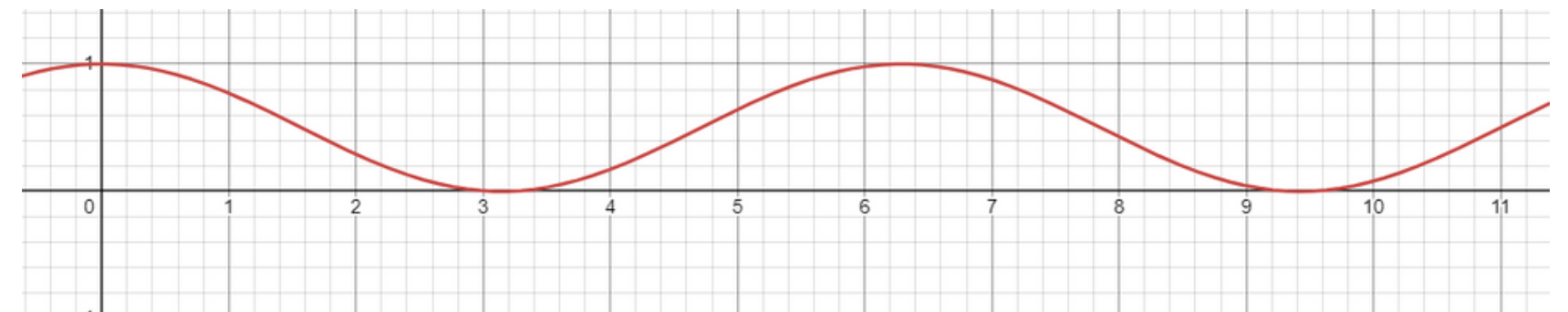
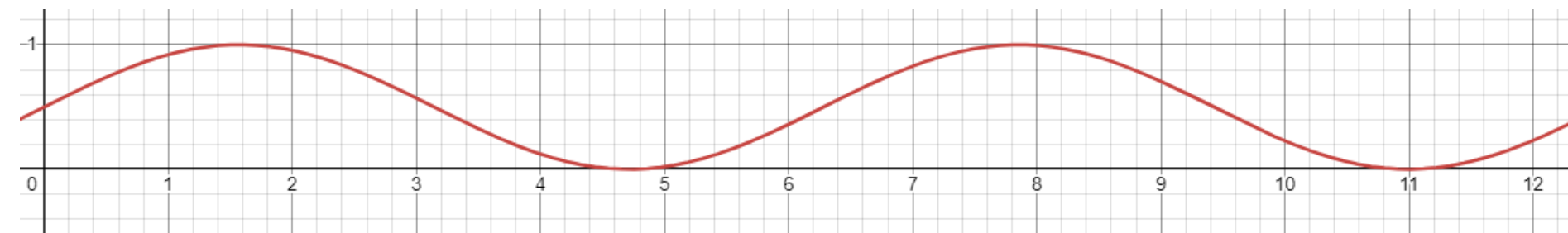
lets try to replace these with variables! To do so, we will replace “float radius” with $f(x)$ and $iTime$ as x ! Plugging this into desmos, we can see this:

I have cropped out the negative portions of this graph, as time will never be negative. Notice how the function starts at $(0,0.5)$. Since this controls our radius, it is possible to see how the circle starts with a radius of 0.5, goes to 1, then goes to 0 and repeats.

Lets try to modify this equation! If we replace the $\sin()$ function with a $\cos()$ function, a graph like this can be seen:

Guess what happens to the circle!

(These graphs were produced in Desmos! It is recommended to try out functions with a visualizer to see what values you can obtain)





NOW IT'S YOUR TURN!

Create a shader that includes all of
the ideas discussed, combining
everything you have learned!