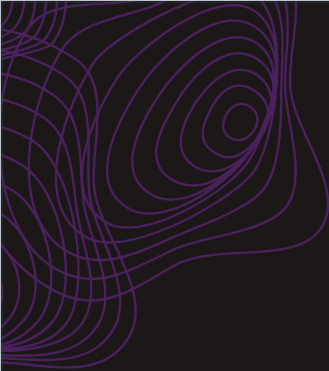




Intro to Shader Programming

An introductory exploration of shader
concepts and examples.



What we will be using today

shadertoy.com

and

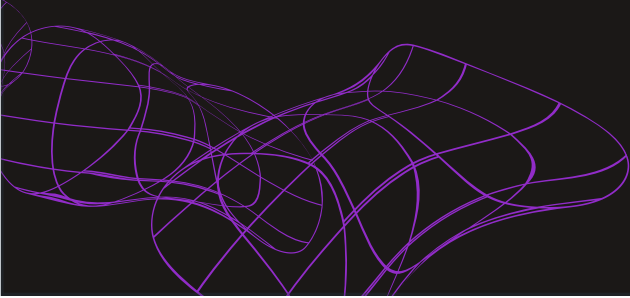
github.com/GraphicsProgrammingKnights/Workshops

Code for today is in "Fall 26" folder.



What is a shader?

A shader is a GPU program
that runs on GPU hardware.



Basically, a shader is a GPU program that runs on GPU hardware.

Why is that different than the regular programs that we write in our favorite languages?

Unlike traditional code written for the CPU, shaders have special constraints that make programming them much different as we will see soon

What types of shaders are there?

Vertex Shaders

- Operates per-vertex on predefined geometry data (i.e. your 3D models).

Fragment Shaders

- Operates per-pixel or 'fragment' in the framebuffer.
- Output of the Vertex shader is input of Fragment shader (sort of).

Compute Shaders

- Used to perform any kind of parallel compute of the GPU.

There are more!

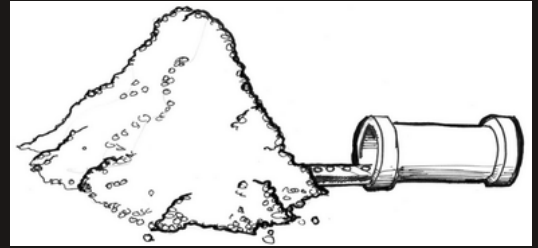
- Geometry shaders
- Tessellation shaders
- Mesh shaders
- Ray-tracing shaders

When most people think of shaders, they are usually referring to either Vertex shaders or Fragment/Pixel shaders. But, you can write a shader to solve many kinds of problems as we will discuss soon.

Why shaders?

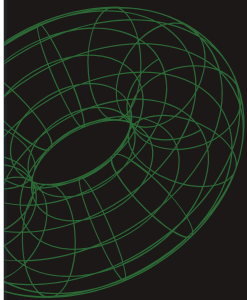
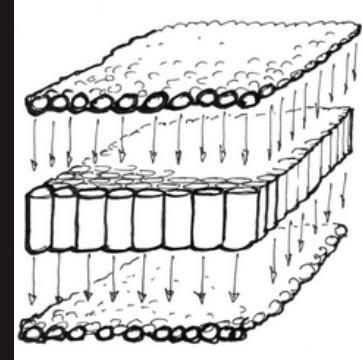
Why write a shader when traditional code gives us the same outputs?

CPU



VS.

GPU



<https://thebookofshaders.com/01/>

I will explain the benefit of shaders (or GPU code), and why it makes sense to use them for problems like the ones on the previous slide.

This analogy comes from a great resource called "The Book of Shaders", and I recommend you guys check it out later.

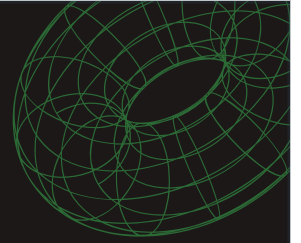
Imagine that your CPU is a big industrial pipe and the tasks that the CPU handles are the things that pass through the pipe. In this analogy, a single pipe is what we call a 'thread'.

Well, this large pipe (the CPU hardware) is great for making complex products one at a time, but what if we want to make a bunch simple products all at once?

This is where the GPU comes in! GPU hardware is specifically designed to be great at performing relatively simple tasks in parallel.

Now, looking from a rendering point of view, your 1920x1080 screen has 2,073,600 pixels that we need to calculate! We don't want to have to wait for each pixel to be calculated one after the other while using the CPU, so we use the GPU instead to operate on all pixels at the same time (or more correctly, in an order independent fashion).

Limitations of shader programs



At least for Vertex / Fragment shaders.

1. Every thread must be independent from every other one.
2. A thread has no 'memory' of its state at a previous moment.

Basically, threads are **BLIND** and **MEMORYLESS**!

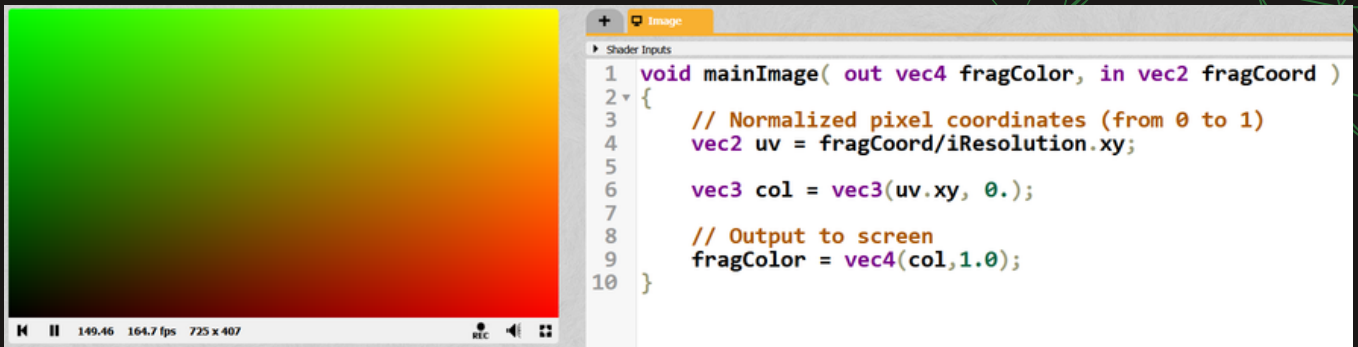
<https://thebookofshaders.com/01/>

In order to run in parallel every thread (or pipe in our analogy) has to be independent from every other thread so that they don't have to wait for one another. This means that threads cannot read out of or write into other threads as they run.

There are workarounds for this. For example, fragment shader output can be written to a texture that is passed back in at the start of the next draw call for those threads to read.

Also, the GPU keeps its threads constantly busy. As soon as a thread is done, it immediately gets new data to process. This makes knowing what a thread was doing in a previous moment impossible.

Looking at a simple example



NOW:

Go to shadertoy.com/new and look at the starter code.

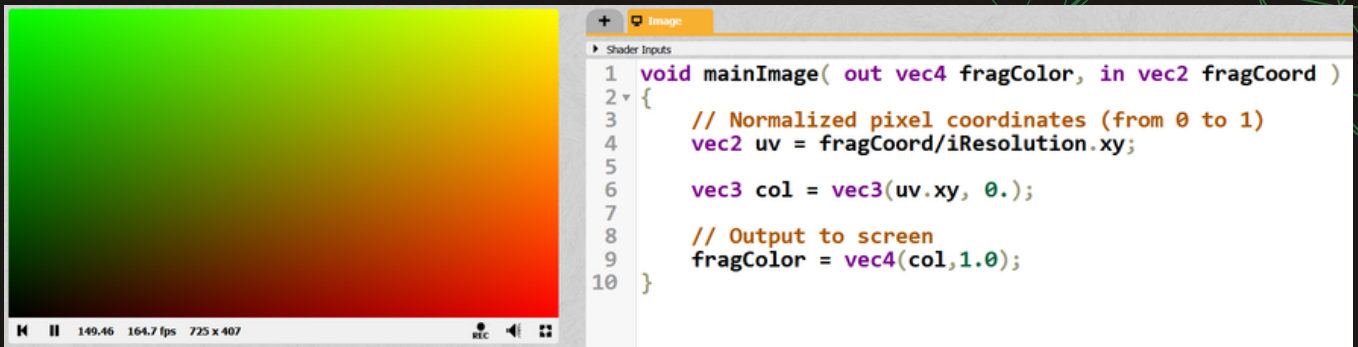
“mental model”:

```
for each pixel (x, y) on screen:
    color = mainImage(x, y, time)
```

Talk about IO (in and out qualifiers assign variables for input and output for our shader program).

Predefined global variables (in ShaderToy, iResolution and things under 'Shader Inputs').

Looking at a simple example



- C-like language.
- Single 'main' function that returns a color at the end.
- Built in *variables* (fragColor, fragCoord, iResolution, etc.), *types* (vec2,3,4, etc.), and *functions* (mainImage()).
- Access vectors using
 - .xyzw
 - .rgba
- These accessors can be used in any order, aka, Swizzling. Example: 'uv.yz'

Talk about IO (in and out qualifiers assign variables for input and output for our shader program).

Predefined global variables (in ShaderToy, iResolution and things under 'Shader Inputs').

Useful GLSL functions

length(x)

$$\sqrt{x[0]^2 + x[1]^2 + \dots}$$

step(edge, x)

if $x < \text{edge}$, return 0.0
else, return 1.0

mix(x, y, a)

Linear interpolation between x and y , where a is the interpolation factor.

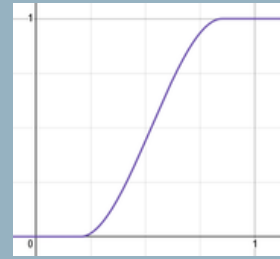
fract(x)

Returns the fractional part of x .
This is calculated as
 $x - \text{floor}(x)$.

More useful GLSL functions

`smoothstep(edge0, edge1, x)`

- Similar to `step()`, but with a lower and upper bound.
- "Smooth" interpolation of x between $0-1$ when $edge0 < x < edge1$
- Otherwise, returns 0.0 or 1.0 respectively.



`sin(angle) & cos(angle)`

Returns the sin/cos of a given angle in **radians**.

smoothstep on Desmos:

<https://www.desmos.com/calculator/t5nfk17ihx>

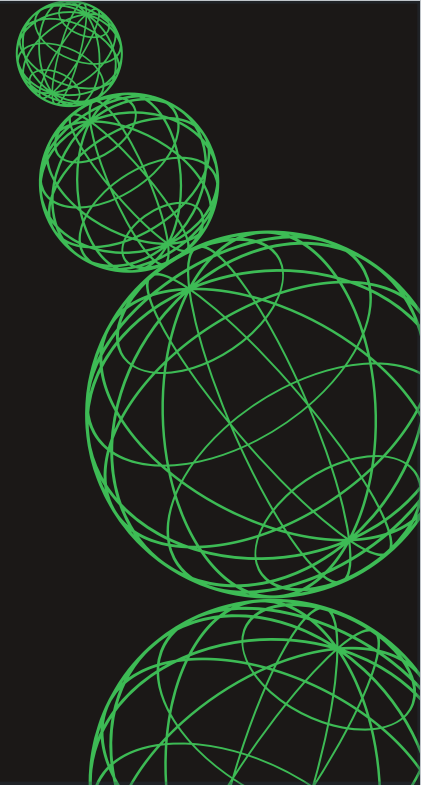
Shaping Functions

To create cool visuals, often times you need to be able to 'shape' your inputs (uv coords, textures, noise, etc.) to get a desired effect.

Shaping functions are special combinations and tweaks applied to common math that helps blend, attenuate, clip, or otherwise **shape** values. Mastering these is the key to making really cool shaders.

If you want to dive into cool shaping functions, Inigo Quilez has a great article.

iquilezles.org/articles/functions/

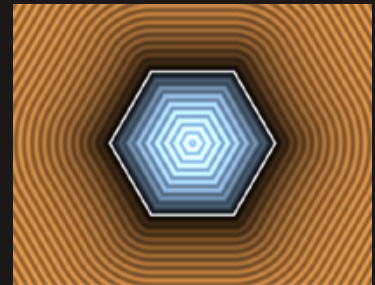
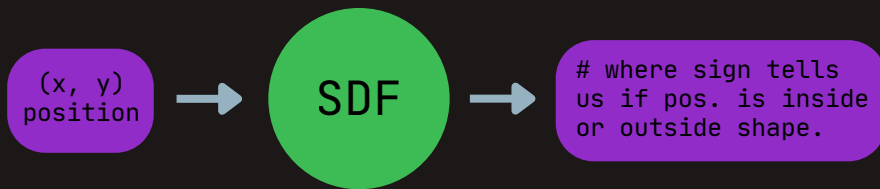


Let's try a
simple shader
together!



<https://www.shadertoy.com/view/Wc2yzy>

Signed Distance Fields



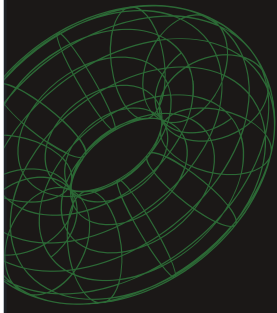
iquilezles.org/articles/distfunctions2d/

Signed Distance Fields (SDFs) can be used to represent shapes.

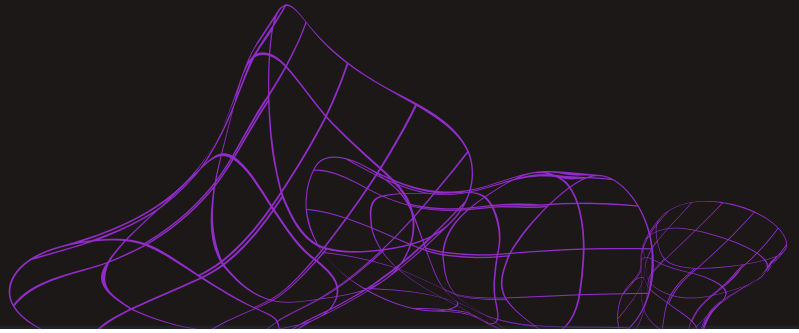
They take in a position in space as input, and return a value representing if that position is inside of the shape or not.

It is called signed because the distance is positive outside the shape, negative inside the shape, and zero on the edge.

They are extremely useful for shaders as you can define pretty much any shape you want given some position on your screen.



Let's try another
shader together!

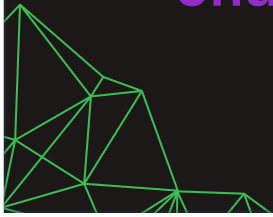


<https://www.shadertoy.com/view/3fSyz3>

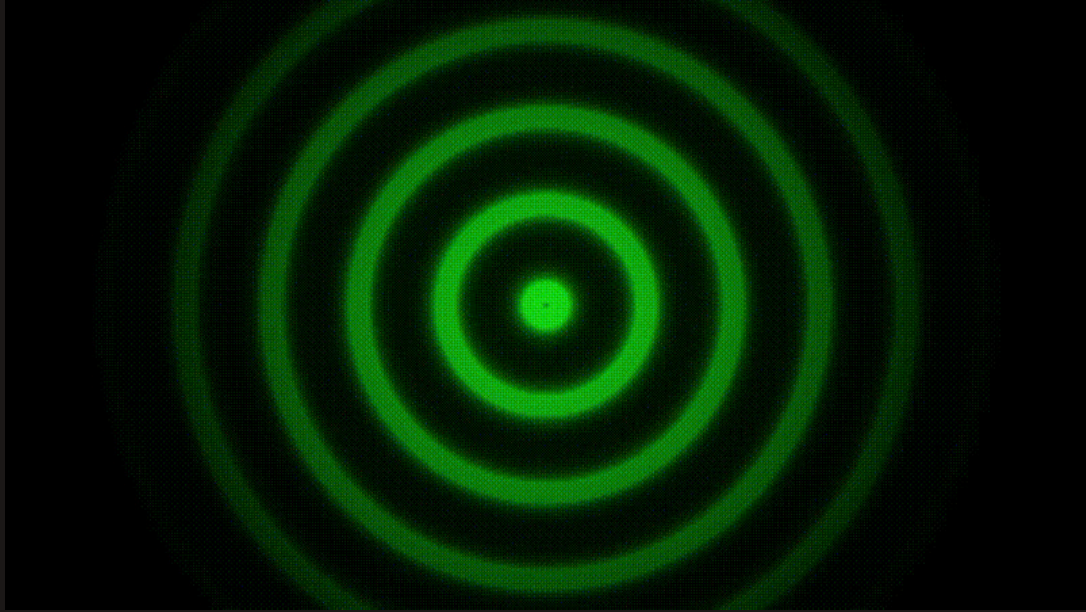
Now you try!

1. Go to shadertoy.com/new if not already there.
2. Mess with the starter code and notice how things change.

Challenge on the next slide...



Challenge



Links for Further Exploration

General Learning for Shaders:

- [An introduction to Shader Art Coding - YouTube](#)
- [thebookofshaders.com](#)
- [iquilezles.org/articles](#)
- [shaderacademy.com](#) (new personal fav)

For my Unity Devs:

- [catlikecoding.com](#)
- [Freya Holmér Shader Tutorials - YouTube](#)

Follow our Socials!

- Stay up to date on our events by following our Instagram!
- Rewatch GPK meetings at your own pace when published on YouTube!





Thank you!