**SHADERS**

**\* What are shaders?**

- Shaders are the sole reason for why we are able to see what is rendered

- The Material that we have been using are actually shaders in-built with three.js

- It is one of the main components of WebGL

- They are programs written in GLSL & sent to the GPU. They position the vertices of the geometry on the renderer and colorize each visible pixel of the geometry

Note - Pixel is not the accurate term when it comes to the points of the renderer as each point on the render does not necessarily match each pixel on the screen. Hence, going forward, we will use the term fragment instead

**\* We send a lot of data to the shader.** For example -

Vertex coordinates

Mesh transformations

Information about the camera

Colors

Textures

Lights

Fog, etc.

\* The GPU processes all of this data following the shader instructions

**\* There are two types of shaders** -

1. Vertex Shader

2. Fragment Shader

**\* VERTEX SHADER** -

- It positions each vertex of the geometry

- We create the vertex shader and send the shader to the GPU with data like vertex coordinates, mesh transformations, camera instructions, etc.

- The GPU follows these instructions and positions the vertices on the render

- The same vertex shader will be used for each vertex

- Some data for each vertex would be the same, e.g., camera position. Such data are called '**uniform**'

- But the vertex coordinates are going to change for every vertex. This data that changes are called '**attributes**'

**\* Once the vertices are placed by the vertex shader, the GPU knows what pixels of the geometry are visible and can proceed to the fragment shader**

**\* FRAGMENT SHADER** -

- It colors each visible pixel of the geometry

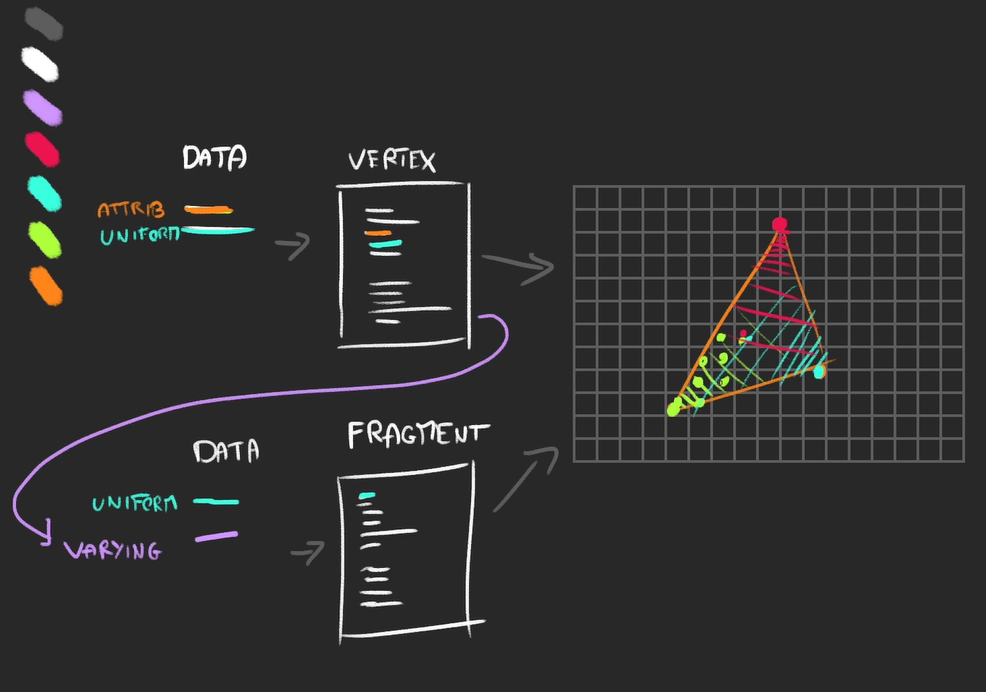
- The same fragment shader will be used for every visible fragment of the geometry

- We create the fragment shader and send the shader to the GPU with data like color

- The GPU then follows instructions and colors the fragments accordingly

- We can send values from the vertex shader to the fragment shader

- These are called ‘Varying’s and the color is interpolated between the vertices



**\* WHY TO USE CUSTOM SHADERS?**

1. *Three.js materials are limited*

- It will be very difficult to apply transformations that would change the appearance of the geometries using the three.js materials

- Example: Consider a plane moving like a wave. It would be very difficult to make something like this with the in-built material using transformations. Also, if we do that, it could have a big impact on the performance

2. *Our shaders can be simple and performant*

- Although three.js shaders are very optimized, it might be an overkill to use them for basic renders with all the lighting and complex logic for shadowing, etc.

3. *We can add custom post-processing*

- We can modify the render once we have built it

- Example: Nico driving after a drink. We need to add the transformation for the camera / render to move as waves

**\* GLSL -**

- This shader language is called GLSL which is short for OpenGL Shading Language

- It is close to C language

- Since it is executed on the GPU, there is no console and hence we cannot use logging to debug (does not make sense anyway as there are millions of calculations that go on simultaneously which can make it difficult to read logs)

- Indentation is not important

- Semicolon is essential unlike JavaScript

- It is a typed language. We must specify a variable's type and we cannot assign any other input type to that variable. We cannot create a float variable and assign an integer to it. The assignment should have a decimal point

- We cannot mix float with integer. If we have an operation where a float variable is multiplied with an integer, it cannot work

- Although, we can typecast it on the fly like this:

float a = 1.0;

int b = 2;

float c = a \* float(b);

**\* Variable Types in GLSL -**

- Float

- Floats are decimals

- Can be positive or negative

- We must always provide the decimal point

- We can do mathematical operations like +, -, \* & /

- Integer

- Like floats without the decimal

- Can be positive or negative

- Can do mathematical operations

- Are not used much

- Boolean

- This can take true / false just like other languages

- Are not used much

- Vector2

- Used to store 2 coordinates (x & y)

- A little like three.js Vector2

- vec2 foo = vec2(1.0, 2.0);

- An empty vec2() will lead to an error

- Although, we can provide one value for both coordinates like vec2 foo = vec2(0.0);

- We can access x and y using the properties and change them

- foo.x = 3.0;

- Multiplying the vex2 will multiply both the coordinates by that value. ie.

vec2 foo = vec2(1.0, 2.0);

foo \*= 2.0;

will give us a vec2 with x = 2.0 and y with 4.0

- Vector3

- Similar to vec2 but with a z coordinate

- A little like three.js Vector3

- Convenient for 3d coordinates

- We can alias x, y and z with r, g and b if we are working with colors

- Can be partially created from a vec2. eg.

vec2 foo = vec2(1.0, 2.0);

vec3 bar = vec3(foo, 3.0);

- Can be used in part to create a vec2 with a process called swizzle. eg.

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

vec2 bar = foo.xy;

- Vector4

- Similar to Vector3 but with a w

- A little like three.js Vector4

- ‘a’ can be used as an alias for w (a for alpha)

- There are other types as well like mat2, mat3, mat4 or sampler2D

**\* Functions in GLSL -**

- It is just like functions in other programming languages. eg.

float loremIpsum() {

float a = 1.0;

float b = 2.0;

return a + b;

}

and then call it from the main function or other function within the .glsl file

- void is supported

- If we provide parameters, we need to type the parameters

- There are many built-in classic methods that can be used like sin, cos, max, min, pow, exp, mod, clamp, etc.

- Also, there are many practical functions available like cross, dot, mix, step, smoothstep, length, distance, reflect, refract, normalize, etc.

**\* Documentation for GLSL -**

1. Shaderific

2. Kronos Group Registry

3. Book of Shaders Glossary

**\* HOW TO CREATE A CUSTOM SHADER? -**

- We can use either **ShaderMaterial** or **RawShaderMaterial**

- ShaderMaterial will have some code automatically added to the shader code

- The RawShaderMaterial will have nothing up-front

\* ***RawShaderMaterial*** -

- After creating two .glsl files, one for vertex and other for fragment and importing them into our script file, we can see that it works just like the in-built materials

- The RawShaderMaterial constructor still allows some properties like wireframe, side, transparent, flatShading, etc.

- Properties like map, alphaMap, opacity, color, etc do not work as we would need to write features for them within our shaders ourselves

\* ***ShaderMaterial*** -

- We have some variables already created for us when we are using ShaderMaterial

- We need not define the following uniforms, attributes and precisions in both the shaders

- uniform mat4 projectionMatrix

- uniform mat4 viewMatrix

- uniform mat4 modelMatrix

- attribute vec3 position

- attribute vec2 uv

- precision mediump float

**\* UNDERSTANDING THE VERTEX SHADER -**

**\* The main()**

- This is called automatically and returns void

- Shader will not work without the main()

**\* gl\_Position variable**

- This is a predefined variable

- We need to assign it

- It will contain the position of the vertex on the screen

- Its type is vec4

\* But why is it a vec4?

- Because the coordinates are in clip space

- It’s like positioning things in a box

- The z is for the depth (to know which part is in front of the other) and the w is responsible for the perspective (homogenous coordinates)

**\* The position attribute**

- We have used this attribute while we made our own buffer geometry

- This is different for each vertex

- Contains x, y and z coordinates

**\* The uniform matrices**

- There are three matrices, viz. projection, view and model

- They are uniform as they do not change across the geometry

- Each matrix will do a part of the transformation

- To apply a matrix, we multiply it

- The matrix is a mat4 as we need to have the same size as the vec4

**\* The Model Matrix**

- They apply transformations relative to the Mesh (position, scale and rotation)

- When we change these from the three.js code, it internally converts it to this mat4 matrix which contains information on the transformation

**\* The View Matrix**

- This applies transformations relative to the camera (position, rotation, field of view, near, far)

**\* The Projection Matrix**

- This matrix transforms the coordinates into the clip space coordinates

**\* Learn more about matrices and coordinates using this link** [**https://learnopengl.com/Getting-started/Coordinate-Systems**](https://learnopengl.com/Getting-started/Coordinate-Systems)

**\* We can also combine the model and the view matrices into one and have a viewModelMatrix but that will reduce the control we have over our variables**

**\* UNDERSTANDING THE FRAGMENT SHADER -**

**\* The main()**

- It is required for the shader to work

- It is called automatically

- Returns void

**\* Precision**

\* Decides how precise is the float we are using

\* It can be really important in deciding how much we can zoom in / out of the scene (example)

\* It can have three values

- highp

- mediump (default)

- lowp

\* Providing the precision is mandatory

\* highp is very precise and hence can cause performance issues. Also, due to this, it might not work on many devices

\* lowp can lead to glitches over lack of precision. Like a object suddenly jumps from two steps ahead in the path (example)

\* Hence, mediump is usually preferred

\* When using the ShaderMaterial class of three.js, this would be automatically handled

**\* gl\_FragColor**

\* Just like gl\_Position, it is a predefined variable in GLSL

\* We need to assign a value to it

\* It will contain the color of the fragment

\* It is a vec4(r, g, b, a)

\* If we only change the alpha in the shader, it will not work. We need to change the transparent property to true when we are calling the shader from the constructor in the three.js logic

**\* PASSING AN ATTRIBUTE TO THE VERTEX SHADER -**

\* We have done this before when we created our own BufferGeometry while we added a BufferAttribute with the name 'position' and passed in a Float32Array and provided the itemSize to determine how many items in the array are to be bound to one of the vertices of the geometry

\* So, all we need it to provide an array that is going to be added to a new BufferAttribute and define a itemSize

\* This can be imported into the Vertex Shader and we need to add the keyword 'attribute' before adding the datatype of the attribute

\* We need to use the same name that is passed into the Shader as the attribute name from the three.js file

**\* PASSING THE ATTRIBUTE TO THE FRAGMENT SHADER AS A VARYING -**

\* We need to define a varying within our vertex shader and assign the value of our attribute to the varying

\* We can then add the same varying variable to the fragment shader

\* We can then use this varying in the functions within our fragment shader

**\* PASSING A UNIFORM TO THE SHADERS -**

\* We can pass the uniform as an object using the 'uniforms' property of the RawShaderMaterial class as:

uniforms: {

uFrequency: { value: 10 }

}

\* We can use this uniform in both, the vertex as well as the fragment shader

**\* USING TEXTURES IN SHADERS -**

- Import the texture in three.js

- Pass it as a uniform to the fragment shader

- In the fragment shader the type of this uniform would be 'sampler2D'

- Then, we need to map the colors of the textures to the UV coordinates and we can do that using the 'uv' attribute present in the vertex shader and passing it as a varying to the fragment shader

- In the fragement shader, we can then use the texture2D(...) method which takes in two params:

1. The sampler2D variable

2. The UV coordinates

**\* DEBUGGING -**

\* When there is an error in the shader code, three.js and javascript print the whole shader code in the error console

\* This log can help us find our error

\* It also logs a line where the actual error has occured which makes it easier to find it

\* Since there is no console to log, we can use the gl\_FragColor to test whether a particular variable holds the correct value or not

**\* GLSLIFY -**

\* With glslify, we can import and export glsl codes just like modules

\* Perfect to separate our code into multiple smaller and reusable parts

\* You can use the glslify-loader and add it to the rules if the webpack configuration

**\* REFERENCES -**

\* The Book of Shaders

\* ShaderToy

\* The Art of Code - YouTube Channel

\* Lewis Lepton - YouTube Channel