Game Graphics & Real-time Rendering

CMPM 163, W2018

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Lucas Ferreira (TA)
lferreira@ucsc.edu

creativecoding.soe.ucsc.edu/courses/cmpm163
github.com/CreativeCodingLab

Class information

Class website:

https://creativecoding.soe.ucsc.edu/courses/cmpm163

Slack is the main form of class communication:

https://ucsc-creativecoding.slack.com/messages/cmpm163

https://ucsc-creativecoding.slack.com/messages/cmpm163-lab

Lucas Ferraria is our TA:

He will run the lab sessions starting next week

For Thursday

Play around with Three.js:

- Make sure you can get the Three.js library downloaded + working on your laptop or a lab computer
- Get the code example from this page to work: https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene
- Look through the example code at: https://threejs.org/examples/
- Some examples:
- https://threejs.org/examples/#webgl_materials_shaders_fresnel
- https://threejs.org/examples/#webgl_shader_lava
- https://threejs.org/examples/#webgl_shaders_ocean
- https://threejs.org/examples/#webgl_shader2

Setting up Three.js

Pretty straightforward to setup!

Copy the three.js (or three.min.js) library to the same folder as your program, or a subfolder that lives in the same directory as your program. (I have it in a folder called js/)

Start a webserver (not needed for simple examples, but is useful when you are loading in models, textures):

```
python -m SimpleHTTPServer 8888
```

Then in your browser, go to:

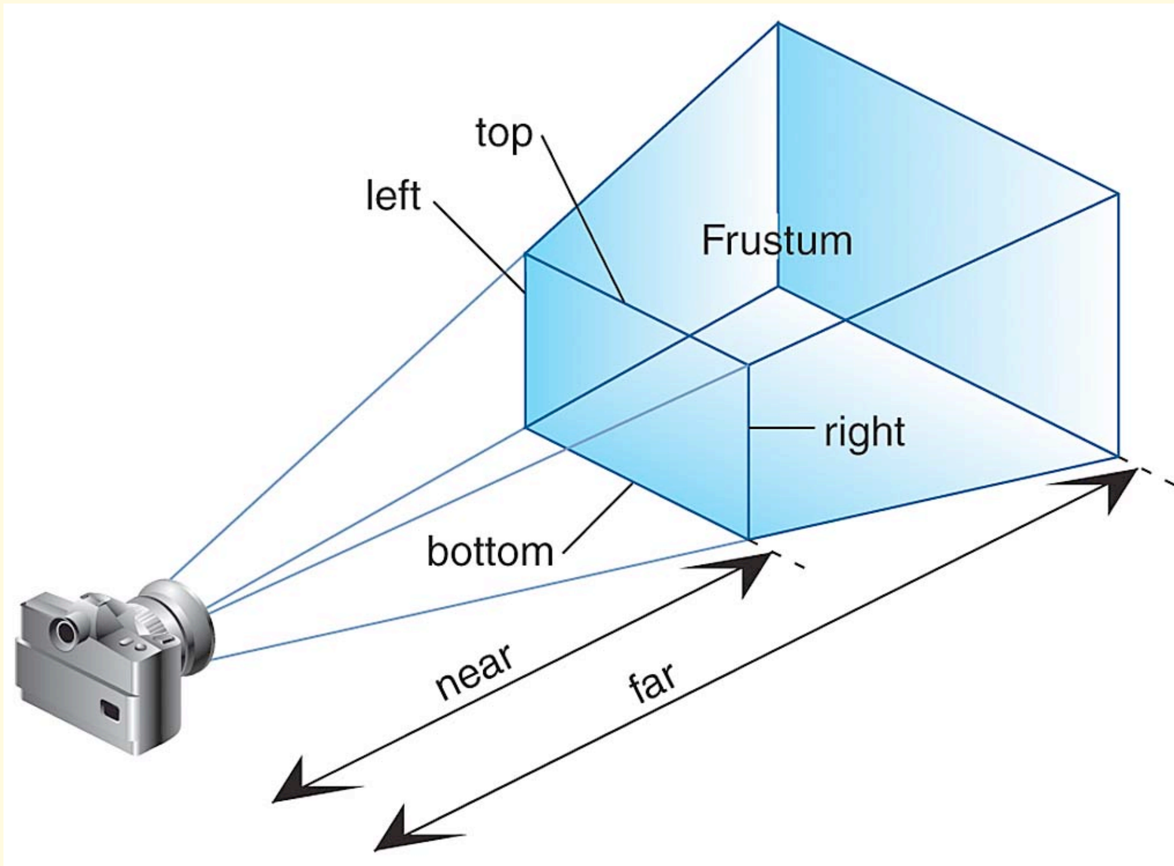
http://localhost:8888

3D Scene

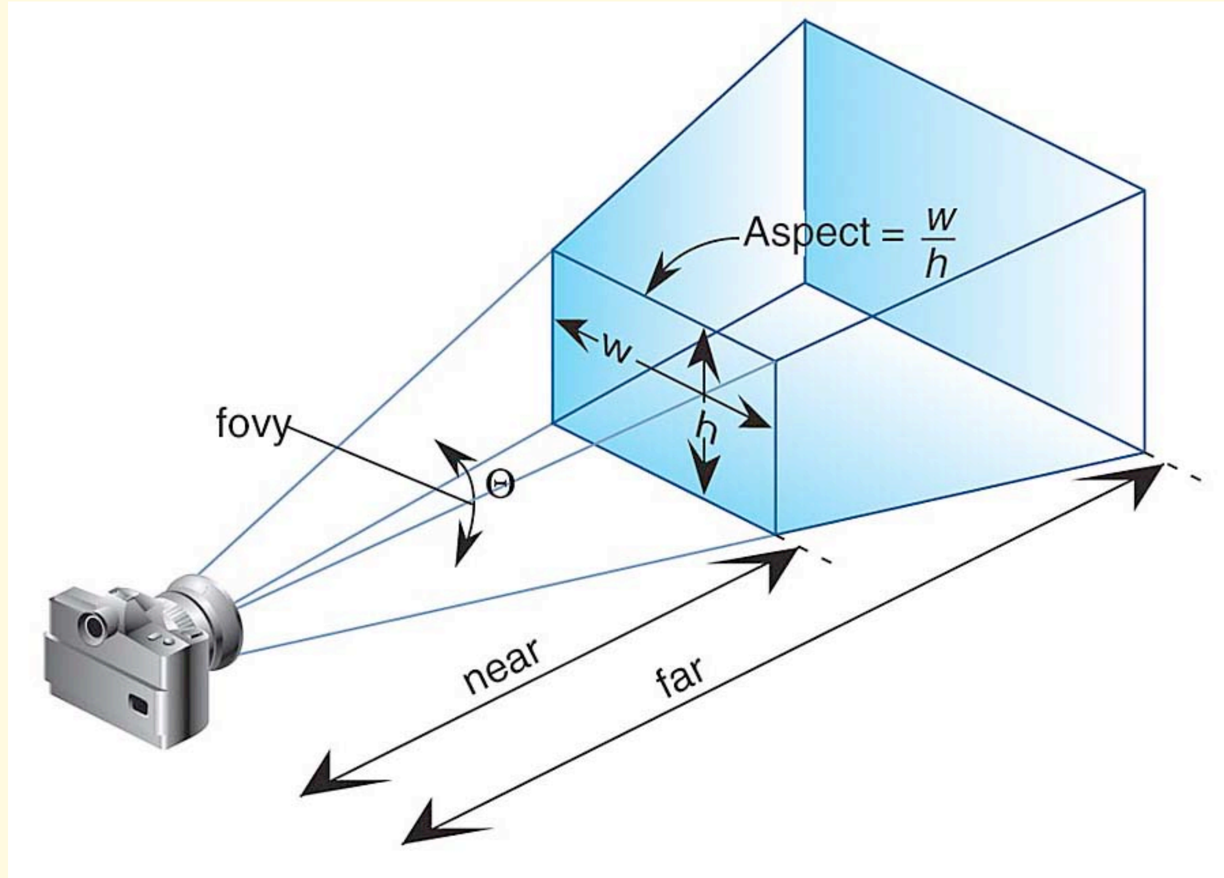
In most graphics framework, a scene consists of:

- **Cameras:** A camera characterizes the extent of the 3D space that will be projected onto the 2D image plane
- **Lights:** Different positions, orientations, colors, and types of light are used to illuminate the scene
- **Geometry:** Objects made out of triangles populate the scene, and are lit by the different light source
- **Materials:** The objects in the scene can be characterized by different material properties
- **Textures:** The objects can also have different textures, or images, placed on them

Perspective camera



Perspective camera



Rendering pipeline

The rendering pipeline describes the series of operations that transform your programmatically defined 3D scene into an actual 2D image.

This is done using shaders.

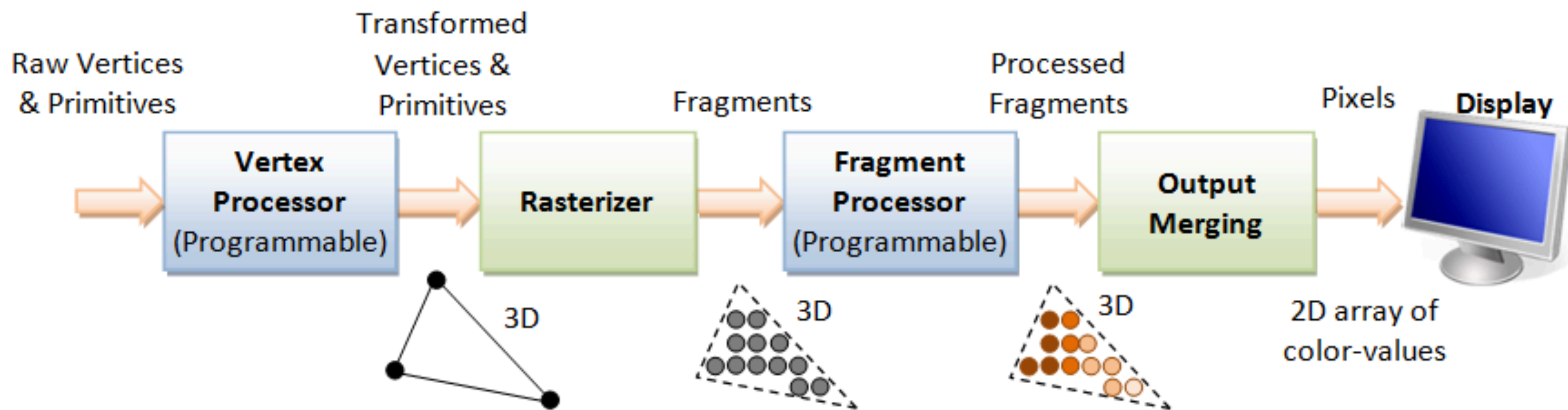
- **Vertex shader:** The vertex shader is responsible for turning 3D geometry into "normalized device coordinates" (2D values between -1 and +1, plus a depth value)
- **Fragment/pixel shader:** The fragment shader is responsible for coloring in each of those pixels, and then outputting it on the 2D screen

Each GLSL shader lives in its own file. At the start of an application, these files are compiled into a "program" and copied over to the GPU, along with texture data.

When the application is running, usually at 60fps, during *each frame*, the following step occur:

- 1. The shader program is activated on the GPU ("bound")
- 2. Any texture data you will use is also bound (ie, made available to the shader)
- 3. 3D points describing your geometry are passed in to the GPU, and input into your vertex shader, one triangle at a time
- 4. The vertex shader projects the triangle into a simpler coordinate system and outputs it as "fragment," or pixel data, which is input into the fragment shader
- 5. The fragment shader decides what color to make each pixel, and then draws it on the screen

Perspective camera



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Rendering pipeline

Let's look at a high level description of this process, using **Three.js**. (Most graphics/game frameworks conceptualize this process similarly.)

This is done using shaders.

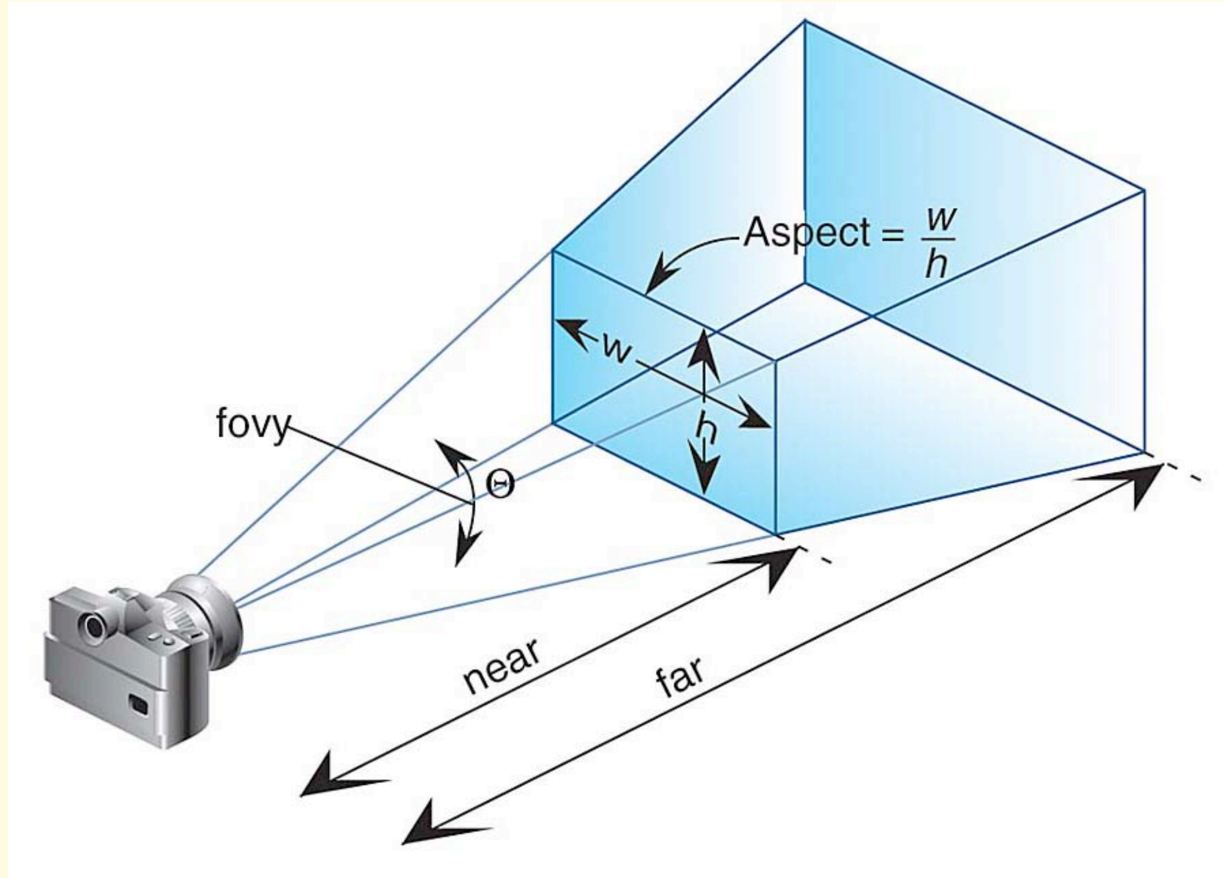
- **Vertex shader:** The vertex shader is responsible for turning 3D geometry into 2D pixels
- **Fragment/pixel shader:** The fragment shader is responsible for coloring in each of those pixels

```
//define the (currently empty) scene, which keeps track of all the elements used in the rendering process
var scene = new THREE.Scene();

//define the camera that looks onto this scene
var camera = new THREE.PerspectiveCamera(
75, window.innerWidth/window.innerHeight, 0.1, 1000 );

//define the renderer that is used to visualize this geometry
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

Perspective camera



```
//create geometry, made out of 12 triangles
var geometry = new THREE.BoxGeometry( 1, 1, 1 );
//define a material that an be used to describe how the geometry absorbs, reflects, and/or emits light
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
//define a mesh, which assigns a material to the geometry
var cube = new THREE.Mesh( geometry, material ); scene.add( cube );


//position the camera so that it looks toward the origin of the scene
camera.position.z = 5;
```

```
//define the animation loop
var animate = function () {
    //sync up the timing of this loop with screen refresh (e.g., 60fps)
    requestAnimationFrame( animate );

    //make some changes to the position of your mesh
    cube.rotation.x += 0.1; cube.rotation.y += 0.1;

    //pass your data (the geometry in the scene + the camera definition)
into the WebGL renderer, ie, into the Three.js default shaders
    renderer.render(scene, camera);
};

//start the animation loop
animate();
```

Three.js shaders

Three.js provides some default shaders, which are defined by the different THREE.Material classes.

This line

```
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
```

creates a simple vertex shader and fragment shader, which is bound when the geometry it's attached to (via the Mesh object) is rendered.

Three.js shaders

The most basic vertex shader is called a "pass-through shader"

This vertex shader does only this:

It takes each triangle used to define your geometry and projects it into 2D, then passes this 2D data to the fragment shader.

The most basic fragment shader simply gives each pixel a color value.

When we used { color: 0x00ff00 } as an argument when we instantiated our "Basic Material," it was used to create a fragment shader that colors ("shades") every pixel green.

Three.js shaders

We can also define more complex shaders.

A more interesting shader is called a "Phong shader," named after Bui Tuong Phong, a computer scientist who studied at University of Utah (where much of computer graphics was invented). It approximates ambient, diffuse, and specular lighting.

Three.js has this built in to one of its materials, MeshPhongMaterial. But we can create it ourselves using shaders.

As before, it takes each triangle used to define your geometry and projects it into 2D, then passes this 2D data to the fragment shader.

But each triangle is colored depending on the interaction between: the camera orientation, the surface normal, and the light position

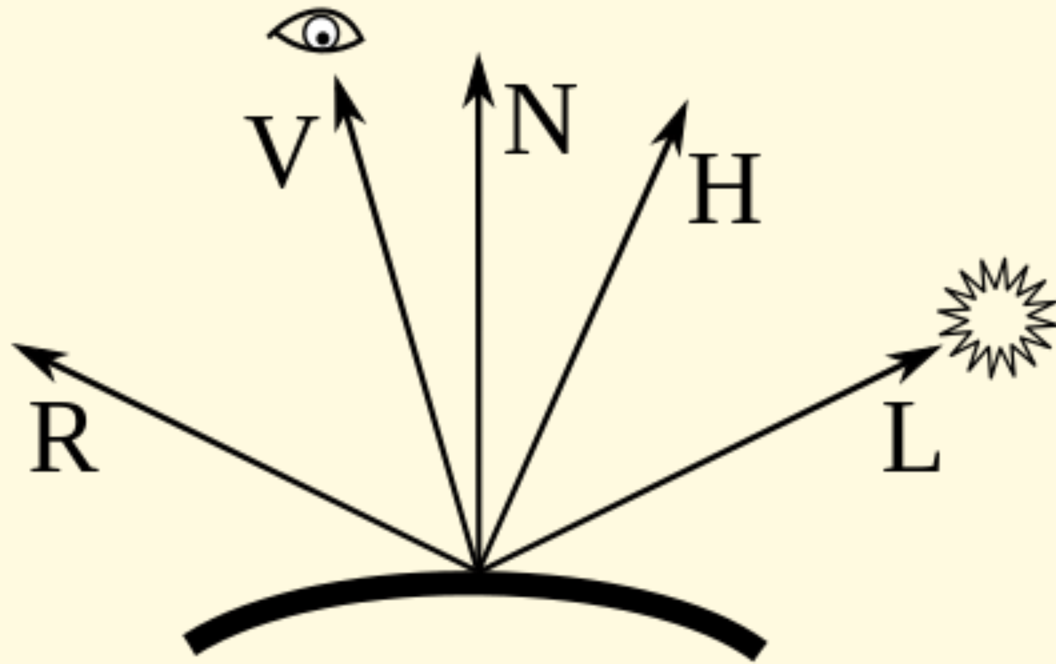
Phong shading

As before, this shader it takes each triangle used to define your geometry and projects it into 2D, then passes this 2D data to the fragment shader.

But each triangle is colored *depending on the interaction between*: the camera orientation, the surface normal, and the light position.

- The camera position and orientation is defined by your renderer
- The surface normal is created automatically by Three.js for its default shapes, and any modeling software will also create these for you
- The light position is specified by you and explicitly passed into the shader program

Blinn-Phong lighting



```
//define the position of the point lights
var ambient = new THREE.Vector3(0.1,0.1,0.1);

var light1_pos = new THREE.Vector3(0.0,10.0,0.0); //above
var light1_diffuse = new THREE.Vector3(1.0,0.0,0.0); //red
var light1_specular = new THREE.Vector3(1.0,1.0,1.0);

var light2_pos = new THREE.Vector3(-10.0,0.0,0.0); //left
var light2_diffuse = new THREE.Vector3(0.0,0.0,1.0); //blue
var light2_specular = new THREE.Vector3(1.0,1.0,1.0);
```

```
// define the geometry
var geometry1 = new THREE.SphereGeometry( 1, 64, 64 );
var geometry2 = new THREE.BoxGeometry( 1, 1, 1 );
var geometry3 = new THREE.TorusKnotGeometry( 1, 0.1, 100,
16 );

// define the materials – here we are defining our bridge to
the shader programs
material = new THREE.RawShaderMaterial( {
    uniforms: uniforms,
    vertexShader: vs,
    fragmentShader: fs,
} );
```

uniform: data that is static over the life of the binding – It will be the same for all geometry passed in the shader. Three.js defines the camera data for you automatically if you use their Camera objects and link it to the scene.

attribute: data that is defined per vertex for each geometry that you pass in (i.e., position, normal, texture coordinates). Three.js defines most of this for you automatically.

varying: this is how you link data from the vertex shader to the fragment shader.

```
var uniforms = {
    ambient: { type: "v3", value: ambient },
    light1_pos: { type: "v3", value: light1_pos },
    light1_diffuse: { type: "v3", value: light1_diffuse },
    light1_specular: { type: "v3", value: light1_specular },
    light2_pos: { type: "v3", value: light2_pos },
    light2_diffuse: { type: "v3", value: light2_diffuse },
    light2_specular: { type: "v3", value: light2_specular },
};
```

uniform: data that is static over the life of the binding – It will be the same for all geometry passed in the shader. Three.js defines the camera data for you automatically if you use their Camera objects and link it to the scene. Uniform variable are available to both the vertex and the fragment shader.

attribute: data that is defined per vertex for each geometry that you pass in (i.e., position, normal, texture coordinates). Three.js defines most of this for you automatically. Attribute data is only available in the vertex shader.

varying: this is how you link data from the vertex shader to the fragment shader.

```
var uniforms =  {
    ambient: { type: "v3", value: ambient },
    light1_pos: { type: "v3", value: light1_pos },
    light1_diffuse: { type: "v3", value: light1_diffuse },
    light1_specular:  { type: "v3", value: light1_specular },

    light2_pos: { type: "v3", value: light2_pos },
    light2_diffuse: { type: "v3", value: light2_diffuse },
    light2_specular:  { type: "v3", value: light2_specular },
};
```

A GLSL shader looks a lot like a C program, which some differences and limitations.

In its “main()” function, the vertex shader **must** define a variable called **gl_Position**.

In its “main()” function, the fragment shaderr must define a variable called **gl_FragColor**.

Let’s look at our shader programs…

Get code to run on your laptop

Customize the code (without breaking it!)

- Change the color of the lights
- Try out different shapes (go to https://threejs.org/docs/ and then scroll down to “Geometries”)
- Change the gl_FragColor output and see what happens
- Play with the rotation speeds of the objects or make the lights move

Questions?

- Homework package #1 will be handed out on Tuesday next week

- Lab sessions will start next week (led by Lucas)

Game Graphics & Real-time Rendering

CMPM 163, W2018

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Lucas Ferreira (TA)
lferreira@ucsc.edu

creativecoding.soe.ucsc.edu/courses/cmpm163
github.com/CreativeCodingLab