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

Last week

- Gave an overview of the rendering pipeline
- Looked at two Three.js programs
- Introduced writing custom shaders in Three.js using the RawShaderMaterial object
- Looked at GLSL syntax
  https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf

This week

- Continue to explore writing custom shaders in Three.js using the RawShaderMaterial object
- Go over GLSL data types and syntax
- Discuss how to pass data from the CPU to the GPU
- Discuss how to pass data between the vertex shader and the fragment shader
- Introduce GLSL textures
- Intodoce offscreen buffers (Frame Buffer Objects)
- Introduce first homework packet
- https://creativecoding.soe.ucsc.edu/courses/cmpm163/code/week2_codeExamples.zip

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

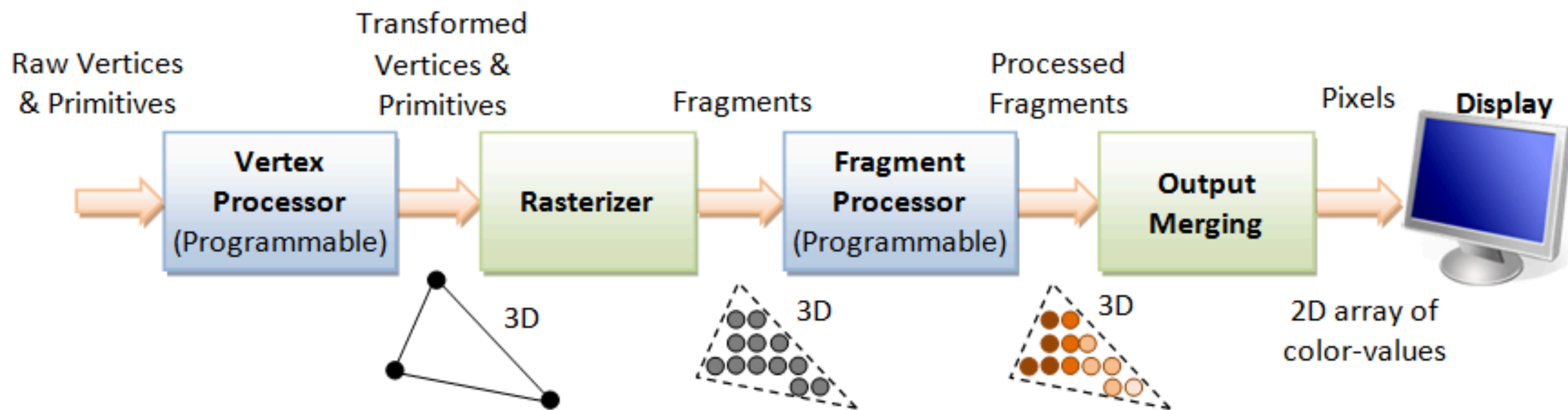
Rendering pipeline

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

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.

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.

uniforms:

To define a uniform in Three.js, you use the following syntax, var uniforms = {

```
    x_and_y: { type: ”2f", value: arrayWithTwoVals},
    light1_pos: { type: "v3", value: light1_pos },
    light1_diffuse: { type: "v3", value: light1_diffuse },
    light1_specular:  { type: "v3", value: light1_specular },
    timerVal: { type: ”f", value: time },
    someNumber: { type: ”i", value: myInteger},
    specialMatrix:  { type: "Matrix4fv", value: fancyMatrix},
};
```

It serves as a kind of a contract. You are telling the shader what to expect will be passed in from the CPU. Thus, your shader **needs** to have uniform variables defined so that it is ready to receive this data.

https://github.com/mrdoob/three.js/wiki/Uniforms-types

varying:

Varying variables provide an interface between the vertex and the fragment shader.

Vertex shaders compute values **per vertex** and fragment shaders compute values **per fragment**. If you define a varying variable in a vertex shader, its value will be **interpolated** over the primitive being rendered, and you can access the interpolated value in the fragment shader.

Download code:
https://creativecoding.soe.ucsc.edu/courses/cmpm163/code/week2_codeExamples.zip

The code example, “**w2_varying.html**”, shows how to use the varying keyword to pass color attribute data from the vertex shader to the fragment shader.

It also introduces Three.js’ BufferGeometry, which gives you more control over how you set up the geometry. Specifically, it lets you define the **attribute** data yourself.

We will create a triangle. We are required to define the position of all the points in the triangle. We also attach two colors to each point on the triangle.

```
//instantiate a new BufferGeometry
var geometry = new THREE.BufferGeometry();

//define the data for the geometry
var vertices = new Float32Array( [
    Math.cos(toRad(90.0)) * 2, Math.sin(toRad(90.0)) * 2,  0.0,
    Math.cos(toRad(210.0)) * 2, Math.sin(toRad(210.0)) * 2,  0.0,
    Math.cos(toRad(330.0)) * 2, Math.sin(toRad(330.0)) * 2, 0.0        ] );
var colors1 = new Float32Array( [1.0,0.0,0.0,  0.0,1.0,0.0,  0.0,0.0,1.0] );
var colors2 = new Float32Array( [0.0,1.0,0.0,  0.0,0.0,1.0,  1.0,0.0,0.0] );

//link the data as attributes available to the vertex shader
geometry.addAttribute( 'position', new THREE.BufferAttribute( vertices, 3 ) );
geometry.addAttribute( 'color1', new THREE.BufferAttribute( colors1, 3 ) );
geometry.addAttribute( 'color2', new THREE.BufferAttribute( colors2, 3 ) );
```

Questions?

- Look at other code examples for week2 (especially the texture example)

- Lab sessions take place tomorrow and Thursday (led by Lucas)

- Lab will cover how to load in more complex objects

- Make sure you understand the material we’ve covered so far

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