Game Graphics & Real-time Rendering

CMPM 163, W2018

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

Lucas Ferreira (TA)
lferreira@ucsc.edu

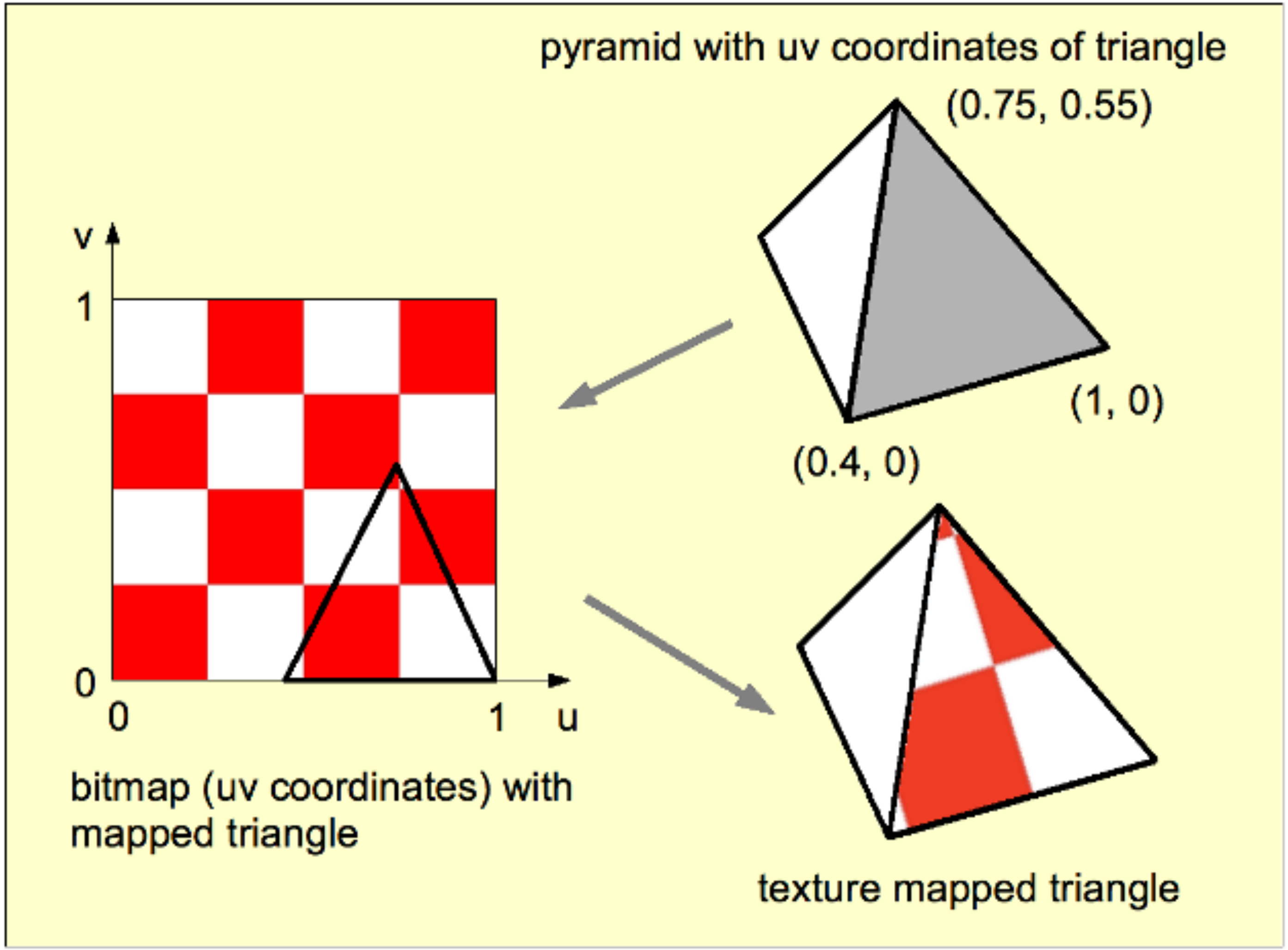
creativecoding.soe.ucsc.edu/courses/cmpm163
github.com/CreativeCodingLab

Last class

- Went over GLSL data types and syntax
- Discussed how to pass data from the CPU to the GPU
- Discussed how to pass data between the vertex shader and the fragment shader
- Looked more closely at GLSL syntax https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf
- Introduced GLSL textures
- Announced Homework #1

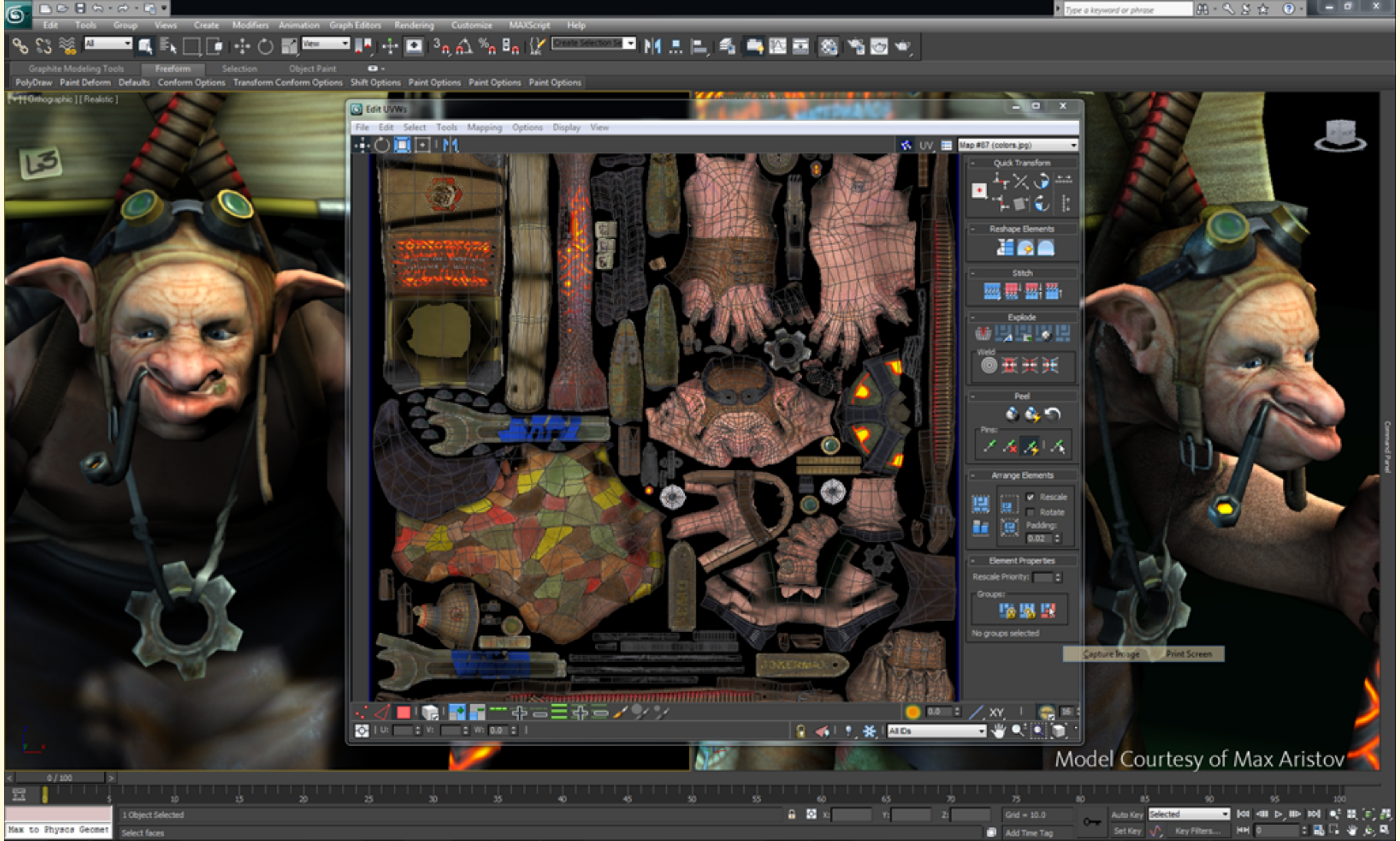
This class

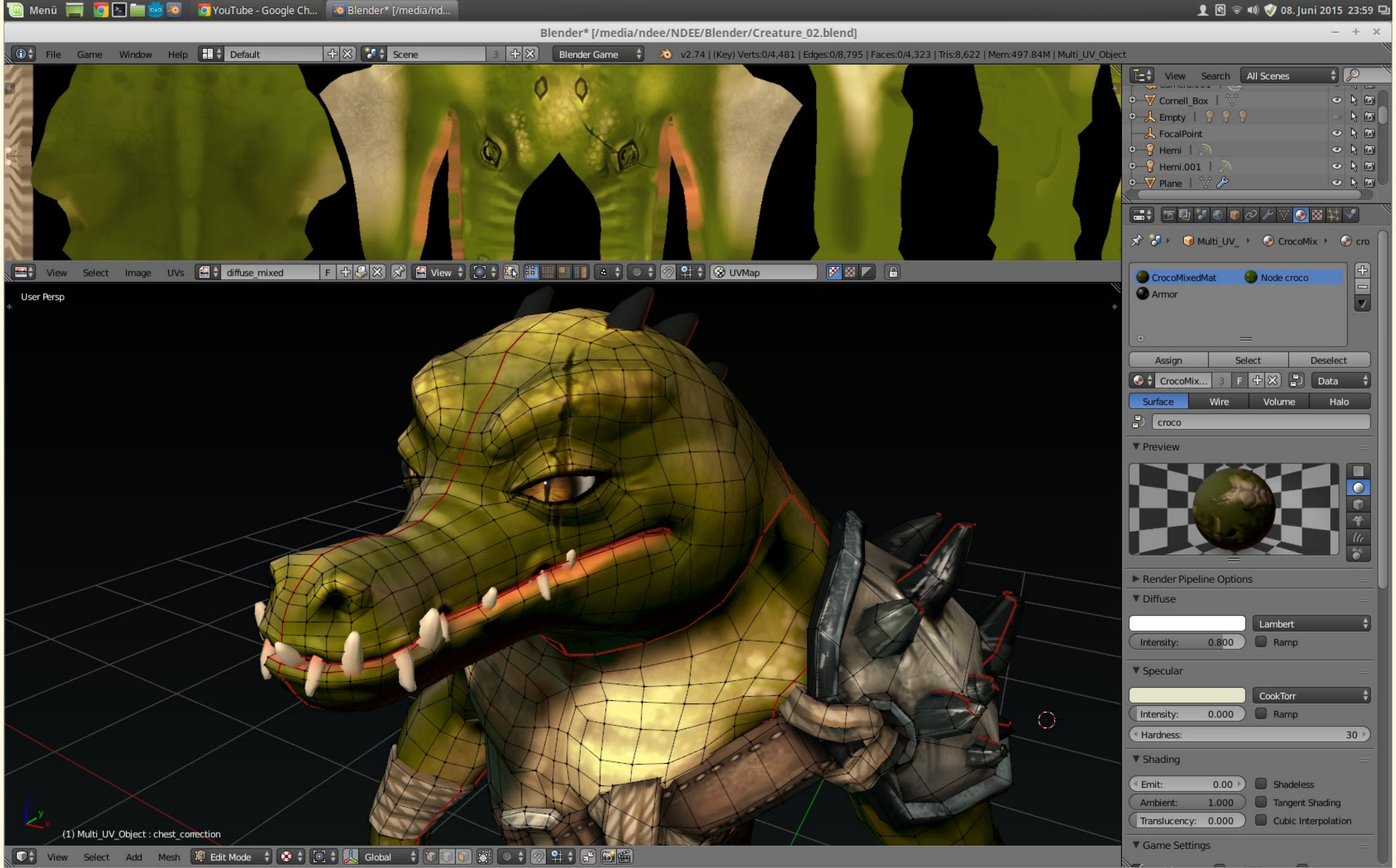
- Exploring textures
- How to write an image processing shader
- How to use Frame Buffer Objects to render to an off-screen texture (using Three.js' WebGLRenderTarget)
- How to swap textures using a "pingpong" strategy to perform computation using data stored in textures
- Using textures as arrays of data
- https://creativecoding.soe.ucsc.edu/courses/cmpm163/code/week2_codeExamples.zip


pyramid with uv coordinates of triangle
(0.75, 0.55)
(1, 0)
(0.4, 0)
v
1
0
0
1
u
bitmap (uv coordinates) with mapped triangle
texture mapped triangle

UV mapping

- Three.js will generate UV coordinates (texture coordinates) for its basic Geometry objects. Not always what you want though…
- If you use BufferGeometry, you have to specify the UV coordinates yourself
- All modeling software (such as Blender) will create UV coordinates for you, which can be loaded into Three.js (as shown by Lucas in the Lab session)
- Most modeling software helps you to create "texture atlases" for complex characters


Edit
Tools
Group
Views
Create
Modifiers
Animation
Graph Editors
Rendering
Customize
MAXScript
Help
Graphite Modeling Tools
Freeform
Selection
Object Paint
PolyDraw
Paint Deform
Defaults
Conform Options
Transform Conform Options
Shift Options
Paint Options
Edit UVWs
File
Edit
Select
Tools
Mapping
Options
Display
View
Map #87 (colors.jpg)
Quick Transform
Reshape Elements
Stitch
Explode
Weld
Peel
Pins:
Arrange Elements
Rescale
Rotate
Padding:
0.02
Element Properties
Rescale Priority:
Groups:
No groups selected
Capture Image
Print Screen
All IDs
0 / 100
1 Object Selected
Select faces
Grid = 10.0
Auto Key
Selected
Set Key
Key Filters...
Add Time Tag
Model Courtesy of Max Aristov



Blender* [/media/ndee/NDEE/Blender/Creature_02.blend]

v2.74 | (Key) Verts:0/4,481 | Edges:0/8,795 | Faces:0/4,323 | Tris:8,622 | Mem:497.84M | Multi_UV_Object

(1) Multi_UV_Object : chest_correction

Diffuse: Lambert, Intensity: 0.800, Ramp

Specular: CookTorr, Intensity: 0.000, Ramp, Hardness: 30

Shading: Emit: 0.00, Ambient: 1.000, Translucency: 0.000, Shadeless, Tangent Shading, Cubic Interpolation

Using textures in GLSL

In Three.js, use the **TextureLoader** helper method and the “**t**” data type to link it to your shader program

```
var myTexture = new THREE.TextureLoader().load( ’myImage.jpg' );
var uniforms = {  tex: { type: "t", value: myTexture } };
var material = new THREE.RawShaderMaterial( {
    uniforms: uniforms,
    vertexShader: vs,
    fragmentShader: fs
} );
```

Using textures in GLSL

The vertex shader needs to set up a **varying** so that the UV coordinates are available in the fragment shader.

```
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
attribute vec3 position;
attribute vec2 texCoords;
varying vec2 UV;

void main() {
    UV = texCoords;
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Using textures in GLSL

The fragement shader can then read that varying in order to sample from the texture.

```
uniform sampler2D tex; //special type used to access texture data
varying vec2 UV; //must match a varying defined in the vertex shader

void main() {
    vec4 c = texture2D(tex, UV); //special method to sample from texture
    gl_FragColor = vec4(c);
}
```

Querying neighbors

- Conceptually, all fragments are processed simultaneously. Thus, when the fragment shader is in the midst of processing one pixel, it can not directly get information about any of the other fragments.

- However, in the fragment shader, if we are passing in a texture, then we can easily query the neighboring pixels in the texture (also called “texels”), as long as we know the width and height of the texture.

- We can pass this info in as uniforms. (The resolution of the texture usually doesn’t match the resolution of the display…)

Querying neighbors

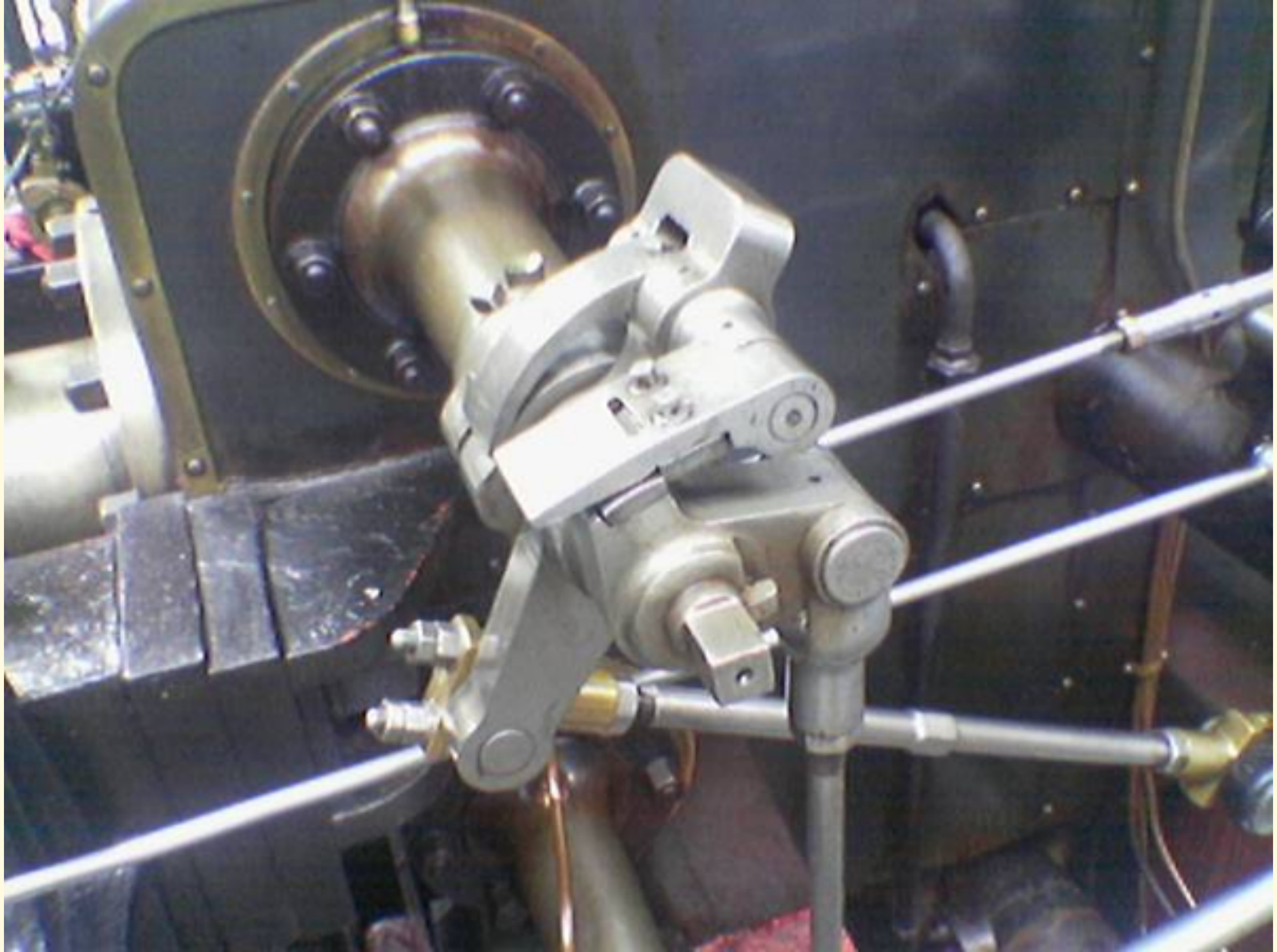
```
uniform sampler2D tex; uniform float rx; uniform float ry;
varying vec2 UV;
void main() {
    vec4 c = texture2D(tex, UV);
    vec2 texelSize = vec2( 1.0/rx, 1.0/ry );
    vec2 left = UV + texel * vec2( -1.0, 0.0 ); //get texcoord for
texel to the left
    vec4 pixelValueOfTexelToTheLeft = texture2D( tex, left );
//get color of that texel
    gl_FragColor = mix(c, left, 0.5); //create a simple blur effect
by averaging the current pixel with its neighbor to the left
```

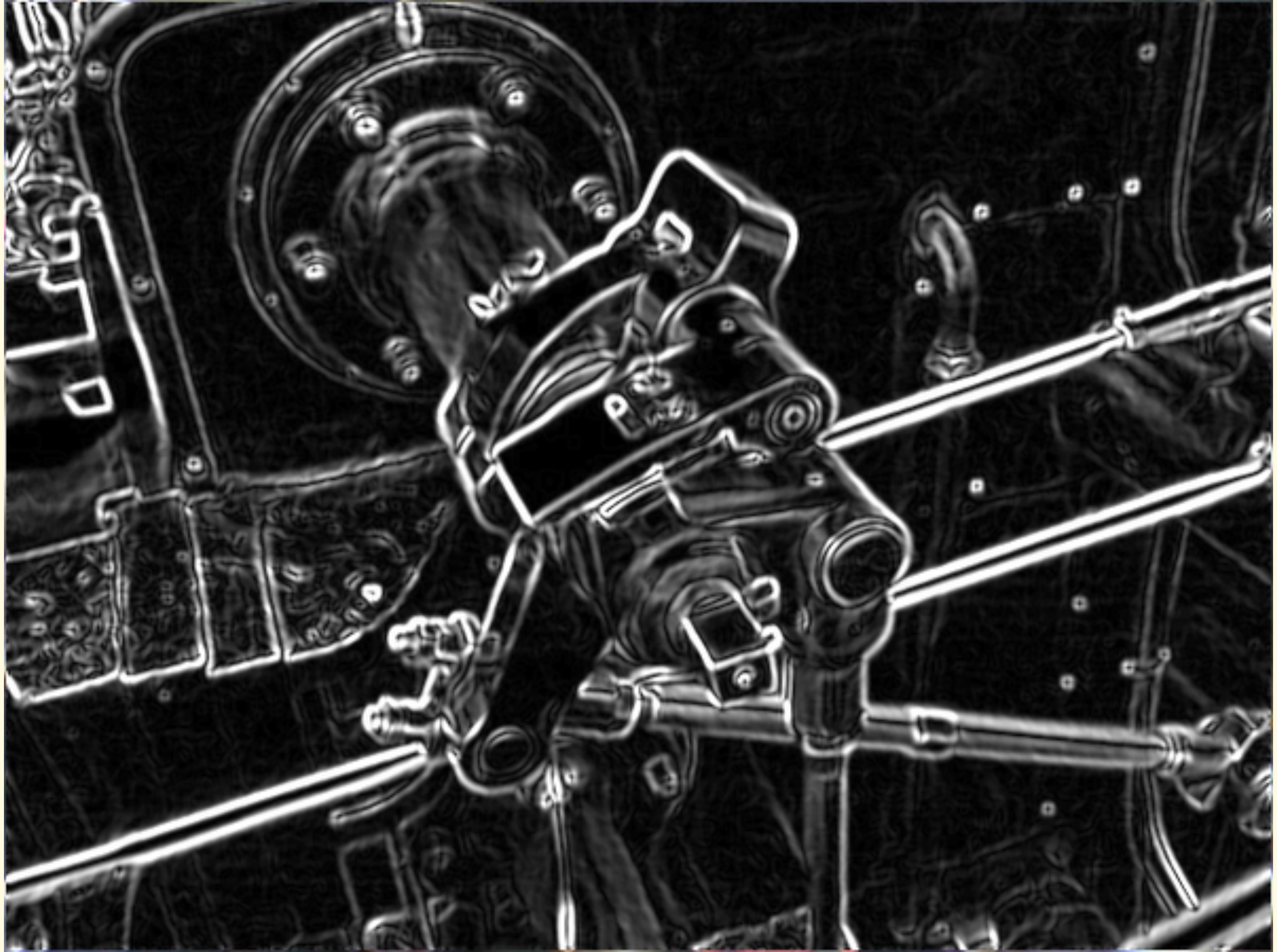
Image processing – edge detection

- You can also do more sophisticated querying and processing of texture data within the fragment shader. For example, we can define an image processing kernel that looks at each pixel's neighbors to determine if the current pixel is an edge.
- That is, we can check if there is a discontinuity between the color of the current pixel and its neighbors in the x or y direction, and have the fragment shader output that information to the screen.
- (See code example "w2_edge.html" where we blend an input image and its edges to create a sketch-like output of a photo.)

  https://en.wikipedia.org/wiki/Sobel_operator

  https://en.wikipedia.org/wiki/Edge_detection





Render to texture

- A very powerful idea used to create a wide range of visual effects is to render the output of a shader program to an off-screen buffer (rather than to the display) so that you can apply multiple rendering passes to your scene (or parts of your scene).

```
var bufferObject = new THREE.WebGLRenderTarget( w, h ); //creates the off-screen buffer, also called a Frame Buffer Object, or FBO
```

- We can then define an addition Scene object, as well as a camera that makes sense for that scene, and add geometry to this scene:

```
var bufferScene = new THREE.Scene();
var bufferCamera = new THREE.PerspectiveCamera( 60, w/h, 1, 1000 );
```

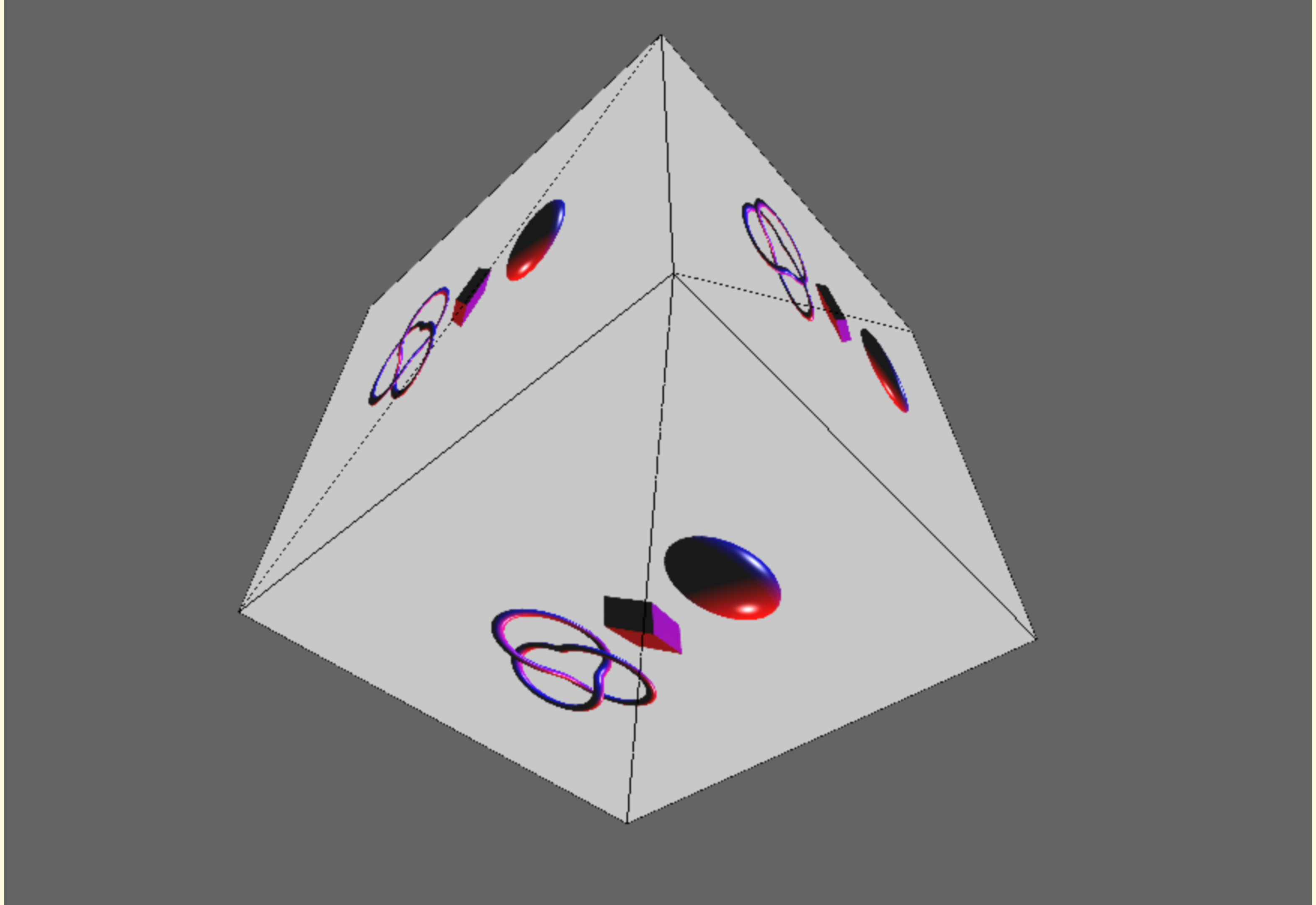
Render to texture

- Then, in our rendering loop, we can render all the objects in this scene, except that they are written to an off-screen texture, rather than displayed on the screen

renderer.render(bufferScene, bufferCamera, bufferObject); //to texture

- We can then use that off-screen texture for some other purpose. That is, we can use it like an ordinary texture. In the w2_renderToTexture.html code, I render our original week1 Phong shader demo to an off-screen texture, and then I use that texture to "decal" a cube, and then render *that* to the display.

renderer.render(scene, camera); //render to screen (also note wireframe)



“Ping pong” textures

- Building off of the render to texture example, we can use our shaders to perform general computation using our texture data.
- Within GLSL, textures are read-only, and we can’t write directly to our input textures, we can only “sample” them.
- Using the render to texture target, however, we can use the output of the shader itself to write to a texture.
- Then, if we can chain together multiple rendering passes, the output of one rendering pass becomes the input for the next pass.
- In the ping-pong technique, we create a kind of simple ring buffer, where after every frame we swap the off-screen buffer that is being read from with the off-screen buffer that is being written to.
- We can then use the off-screen buffer that was just written to, and display that to the screen.

“Ping pong” textures

- We can then use the off-screen buffer that was just written to, and display that to the screen.
- This provides with a way to perform iterative computation on the GPU, which is useful for simulating a range of effects, such as smoke, water, clouds, fire, etc. (which we’ll explore in the coming weeks).
- In the w2_gol_pingpong.html example, I show how we can use this technique to make a GPU version of Conway’s Game of Life: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- The Game of Life is a simple simulation that shows how even very simple rules can give rise to interesting, complex, emergent behaviors.

Game of Life

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*, or "populated" or "unpopulated". Every cell interacts with its eight *neighbours*, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Code is a bit complex, so let's walk through it together…

w2_gol_pingpong.html

Questions?

Can you put the output of the Game of Life onto a cube? (ie, combine the render to texture shader and the pingpong shader)

Look at new code examples for week2, you will build off of them for your Homework assignment

Next week:

- Noise functions
- Introduction to particle systems
- "Sky box" textures / environmental mapping
- reflective & refractive materials
- Vertex displacement shaders + height maps from textures

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