Game Architecture

2/23/16: WebGL
3D Graphics Drivers are literally the worst thing ever.
“Direct-3D IM is a horribly broken API. It inflicts great pain and suffering on the programmers using it, without returning any significant advantages. I don't think there is ANY market segment that D3D is appropriate for, OpenGL seems to work just fine for everything from quake to softimage. There is no good technical reason for the existence of D3D.”

–John Carmack, 1/6/97
problem?
<table>
<thead>
<tr>
<th>GLSL Version</th>
<th>OpenGL</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1/92</td>
</tr>
<tr>
<td>1.1</td>
<td>1.1</td>
<td>1/97</td>
</tr>
<tr>
<td>1.2</td>
<td>1.2</td>
<td>3/98</td>
</tr>
<tr>
<td>1.3</td>
<td>1.3</td>
<td>8/01</td>
</tr>
<tr>
<td>1.4</td>
<td>1.4</td>
<td>7/02</td>
</tr>
<tr>
<td>1.5</td>
<td>1.5</td>
<td>7/03</td>
</tr>
<tr>
<td>1.10.59</td>
<td>2</td>
<td>9/04</td>
</tr>
<tr>
<td>1.20.8</td>
<td>2.1</td>
<td>7/06</td>
</tr>
<tr>
<td>1.30.10</td>
<td>3</td>
<td>7/08</td>
</tr>
<tr>
<td>1.40.08</td>
<td>3.1</td>
<td>3/09</td>
</tr>
<tr>
<td>1.50.11</td>
<td>3.2</td>
<td>8/09</td>
</tr>
<tr>
<td>3.30.6</td>
<td>3.3</td>
<td>3/10</td>
</tr>
<tr>
<td>4.00.9</td>
<td>4</td>
<td>3/10</td>
</tr>
<tr>
<td>4.10.6</td>
<td>4.1</td>
<td>7/10</td>
</tr>
<tr>
<td>4.20.6</td>
<td>4.2</td>
<td>8/11</td>
</tr>
<tr>
<td>4.30.6</td>
<td>4.3</td>
<td>8/12</td>
</tr>
<tr>
<td>4.40</td>
<td>4.4</td>
<td>7/13</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>8/14</td>
</tr>
</tbody>
</table>
And So...

- No more Immediate Mode!
- Modern OpenGL programs *must* contain a vertex shader with a `main()` function and a fragment shader with a `main()` function
- Also, as you’ve seen, you now must supply your own matrices, so those homeworks haven’t been for naught…
DATA

Uniform Variables
- time,
- modelMatrix,
- viewMatrix,
- projectionMatrix,
- lights,
- textures...

Vertices
-1.000000, -0.899410, 0.899410, //position
-1.000000, 0.000000, 0.000000, //normal
0.451000, 0.545000, 0.635, //color
0.899410, -1.000000, 0.899410, //position
0.000000, -1.000000, 0.000000, //normal
0.451000, 0.545000, 0.635 //color
...
...

Indices
[18,22,1,1,22,3,5,11,2,2,11,8,7,13,0...]

CODE

Vertex Shader
- input: attribute vars,
- input: uniform vars,
- local vars,
- output: gl_Position,
- output: varying vars (seriously)

Fragment Shader
- input: varying vars,
- output: gl_FragColor

GPU
Shaders

- Software that runs on the GPU
- Vertex shaders modify vertex attributes (position, normal vector, UV coords, etc.)
- Fragment shaders modify the color of the final pixel (taking in values such as time, lights, etc)
- Geometry and compute shaders out of scope for WebGL
Modern Graphics Programming in Three Easy (?) Steps

- Send the geometry/mesh data to the GPU
- Send the shader code to the GPU
- Specify how to load the data into the shader
Getting the Data to the GPU

• Vertex Buffer Object (VBO)

• All buffer objects are regions of memory controlled by OpenGL

• Allows the implementation to arbitrate CPU and GPU access to the buffer

• Allows the implementation to place the buffer in the fastest memory for a particular operation
Getting the Data to the GPU

- Generate buffer(s)
- Bind a buffer to be active
- Send the data
Getting the Shader Code to the GPU

- Shader object is created (allocated)
- Shader source is specified
- Shader is compiled into a shader object
- Shader object(s) are attached to a program object (that must first be allocated)
- Program object is linked and then made active
That is...

- Shaders must be loaded with a source string (shaderSource), compiled (compileShader), and attached to a program (attachShader) which must be linked (linkProgram) and then used (useProgram)
Loading the Data Into the Shader

- **Vertex Array Object (VAO)**
- Describes how the vertex attributes are stored in a VBO
- **Not** the actual object storing the vertex data, but the **descriptor** of the vertex data
Loading the Data Into the Shader

• Despite the use of the word “pointer” in `glVertexAttribAttribPointer`, we’re not using pointers

• The values used are offsets into buffer object currently bound to the `GL_ARRAY_BUFFER` target
Example Vertex

Stride = 9 * Float32Array.BYTES_PER_ELEMENT

colorOffset = 6 * Float32Array.BYTES_PER_ELEMENT

positionOffset = 0

normalOffset = 3 * Float32Array.BYTES_PER_ELEMENT

colorOffset = 6 * Float32Array.BYTES_PER_ELEMENT

-0.899410, -1.000000, 0.899410, 0.000000, -1.000000, 0.000000, 0.451000, 0.545000, 0.635

position normal vector color
Pseudocode

Setup (once)
• get canvas
• get GL context from canvas and configure
• create, bind, and send vertex array
• create, bind, and send index array
• compile and link vertex and fragment shaders together into shader programs
• get references to all vertex attributes
• get references to all uniform variables

Draw (every frame)
• select shader
• bind appropriate vertex and index buffers
• describe correspondence between values in vertex array and vertex attributes in shader program
• set uniform variables, including mesh and view matrices
• draw call
Homework 5, due 3/1
Run Local Webserver

```bash
#!/bin/sh
python -m SimpleHTTPServer &
then open a browser to http://127.0.0.1:8000
```

Load File

```javascript
function loadContainer() {
    var request = new XMLHttpRequest();
    request.open("GET", "Teapot.json");
    request.onreadystatechange = function () {
        if (request.readyState == 4) {
            writeThisFunctionToProcess(JSON.parse(request.responseText));
        }
    }
    request.send();
}
```
you may hardcode the path for now