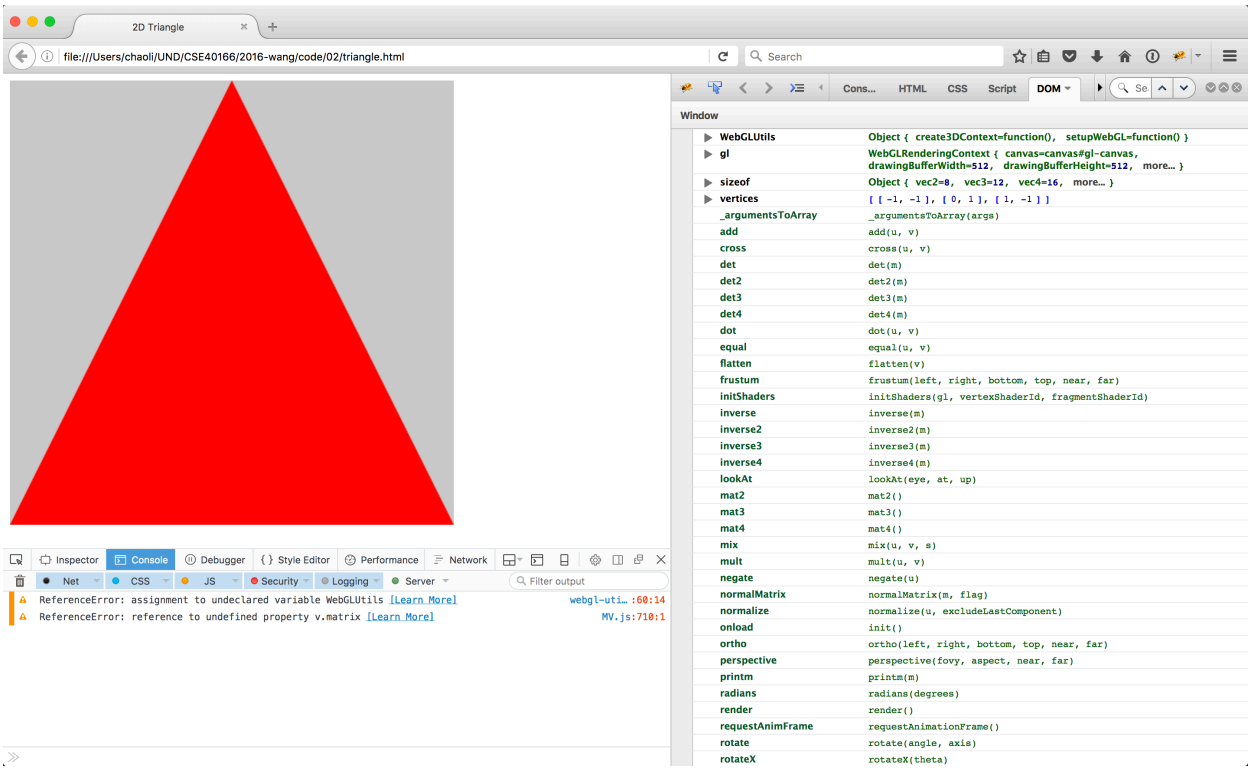
WebGL Programming Notes
Chaoli Wang

1. Browser Debugging Environments

- All major browsers (Chrome, Firefox, IE, Safari) have limited built-in features to support code debugging. Their capabilities vary. To the best of my knowledge, Firefox + Firebug is one of the best choices. **We ask you to stick to Firefox + Firebug for this class.**
  In Firefox, when you run a WebGL program, you can go to the menu: "Tools-> Web Developer -> Web Console" to examine any warning or errors reported.
  In addition, Firebug is a Firefox add-on for debugging HTML / JavaScript / WebGL programs. You need to install Firefox first, then open the Firefox, go to the menu "Tools->Add-ons", type in "firebug" in the search box, and follow the instruction to install Firebug.
  A screenshot of the Firefox + Firebug environment when running the `triangle.html` example is shown below. The **document object model** (DOM) content is displayed. DOM is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.



2. Programming Notes for JavaScript

- JavaScript is case sensitive! For example, `var id` is not the same as `var ID` or `var iD`. They are three different variables!
- JavaScript allows you to define a variable w/o using the `var` keyword, in that case, the variable will be global in scope. However, **we strongly recommend you to add `"use strict";` to the beginning of your JavaScript code to enforce explicit variable definition.** This would make it easier to write "secure" JavaScript code.

3. Programming Notes for WebGL

- For this class, we utilize the utility JavaScript libraries provided by Angel (the author of the text book). Similar to how we include header files in C/C++, we always include the following JS files under the "`Common`" directory:

```
<script type="text/javascript"
      src="../Common/webgl-utils.js"></script>
<script type="text/javascript"
      src="../Common/initShaders.js"></script>
<script type="text/javascript"
      src="../Common/MV.js"></script>
```

  In the above example, we assume the "`Common`" directory and your main program directory (containing html/js files) are under the same parent directory.
- Keep in mind that `gl.bindBuffer` will make the specified buffer the current buffer, until a different buffer is bound. Therefore, all subsequent calls including `gl.bufferData` and `gl.vertexAttribPointer` will work with the current buffer. For instance, the following piece of code is correct:

```
var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER,flatten(vertices),gl.STATIC_DRAW );
var a_vPositionLoc = gl.getAttribLocation( program, "a_vPosition" );
gl.vertexAttribPointer( a_vPositionLoc, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( a_vPositionLoc );

var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER,flatten(colors),gl.STATIC_DRAW );
var a_vColorLoc = gl.getAttribLocation( program, "a_vColor" );
gl.vertexAttribPointer( a_vColorLoc, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( a_vColorLoc );
```

  The following one, however, is not correct (`gl.bindBuffer` are called twice back to back, so the vertex data are bound to the color buffer instead of vertex buffer!):

```
var vBuffer = gl.createBuffer();
var cBuffer = gl.createBuffer();

gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
```

```
gl.bufferData( gl.ARRAY_BUFFER,flatten(vertices),gl.STATIC_DRAW );
var a_vPositionLoc = gl.getAttribLocation( program, "a_vPosition" );
gl.vertexAttribPointer( a_vPositionLoc, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( a_vPositionLoc );

gl.bufferData( gl.ARRAY_BUFFER,flatten(colors),gl.STATIC_DRAW );
var a_vColorLoc = gl.getAttribLocation( program, "a_vColor" );
gl.vertexAttribPointer( a_vColorLoc, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( a_vColorLoc );
```

- Pay attention to `gl.drawArrays(mode, first, count)`. The 2nd parameter specifies the starting index in the array of vector points. The 3rd parameter specifies the number of indices to be rendered, not the ending index in the array of vector points!
- If your WebGL program needs to load in external files from your local disk, not from a URL, the browser by default will not permit this due to security concern. You can get around this by following the instructions given here: `https://github.com/mrdoob/three.js/wiki/How-to-run-things-locally`

  We recommend that you to do with following with the FireFox browser:
  1. Go to `about:config`
  2. Find `security.fileuri.strict_origin_policy` parameter
  3. Set it to `false`

- Notice the different settings for Firefox and Safari in the function "`function loadFileAJAX(name)`" in the "`Common/initShader2.js`" file.
- Notice the different ways of calling `event` for Safari and Firefox, see example code in "`code/04/gasket5.js`".

```
  // works for Safari
    document.getElementById("slider").onchange = function() {
        numTimesToSubdivide = event.srcElement.value;
    };

  // works for Firefox
    document.getElementById("slider").onchange = function(event)
{
        numTimesToSubdivide = event.target.value;
    };
```

4. Programming Notes for GLSL

- Make sure that you understand the differences among the three different types of qualifiers: `attribute`, `uniform`, and `varying`, and use them appropriately.
- Make sure that a shader quantifier and its corresponding application variable are appropriately associated. For an `attribute` quantifier, use `gl.getAttribLocation()`, for an `uniform` quantifier, use `gl.getUniformLocation()`. For a varying quantifier that is passed from a vertex shader to a fragment shader, there is no additional code needed in the main application.

- For an `attribute` or `uniform` quantifier, the application variable name and the shader quantifier name do not need to be the same. We suggestion you to use for example, `a_vPositionLoc` in the application and `a_vPosition` in the shader. However, for a `varying` quantifier, the name defined in the vertex shader must be the exactly same as the name defined in the fragment shader. We suggestion you to add "`v_`" to the beginning of each varying quantifier.
- We ask you to strictly follow this naming convention to reduce possible programming errors:

  In vertex shader:

```
attribute vec4 a_vPosition; // vertex position
attribute vec3 a_vColor; // vertex color
attribute vec3 a_vNormal; // vertex normal
attribute vec2 a_vTexCoord; // vertex texture coordinate
uniform mat4 u_modelViewMatrix; // modelview matrix
uniform mat4 u_projMatrix; // projection matrix
uniform mat3 u_normalMatrix; // normal matrix
uniform float u_shininess; // material shininess
varying vec2 v_fColor; // vertex color to fragment color
varying vec3 v_fTexCoord; // vertex texcoord to fragment texcoord
```

  In fragment shader:

```
uniform sampler2D u_texSampler; // texture image
varying vec2 v_fColor; // vertex color to fragment color
varying vec3 v_fTexCoord; // vertex texcoord to fragment texcoord
```

- Carefully check that a shader quantifier defined in the application is exactly the same shader quantifier that you use in the shader.
- If you define a shader quantifier in the application and do not use it in the shader, errors may occur! Be particularly careful with this since this goes against with what we do in JavaScript or C/C++.