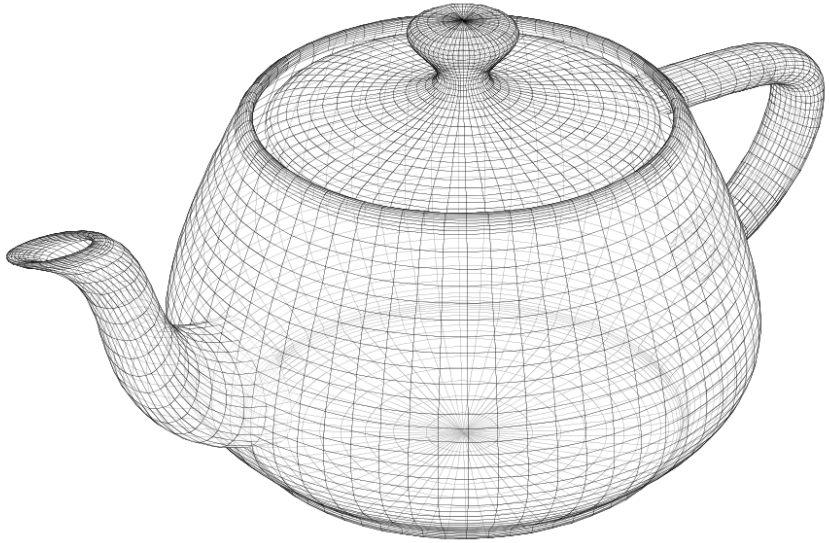
Introduction to WebGL



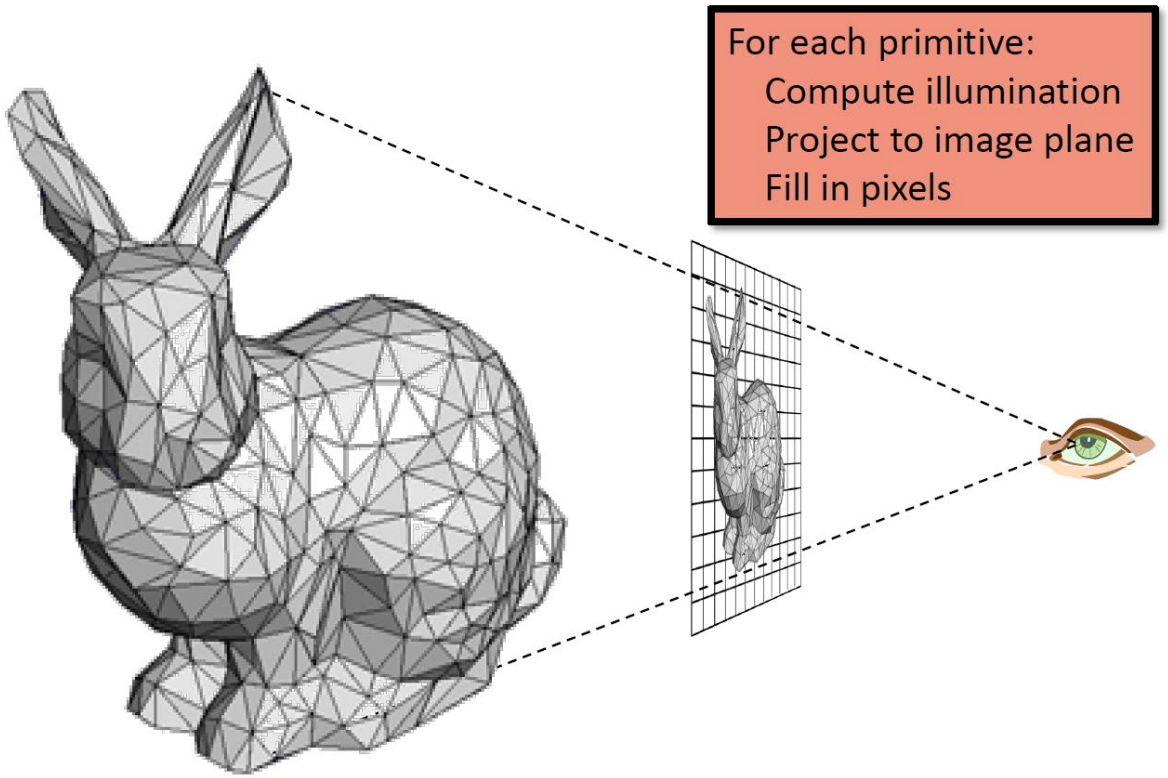
CS 418: Interactive Computer Graphics

Professor Eric Shaffer

Rasterization

A rasterization engine transforms geometric data into an image

- A *raster image* is one formed by a grid of pixels



ILLINOIS

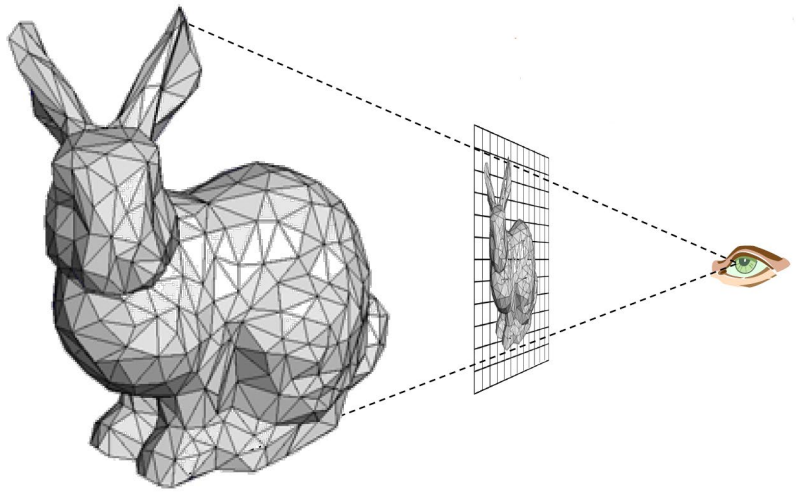
Geometric Data

Camera location (eyepoint)

View plane

Pixel resolution

Models (surface meshes)



```
// View parameters
var eyePt = glMatrix.vec3.fromValues(0.0,0.0,40.0);
var viewDir = glMatrix.vec3.fromValues(0.0,0.0,-1.0);
```

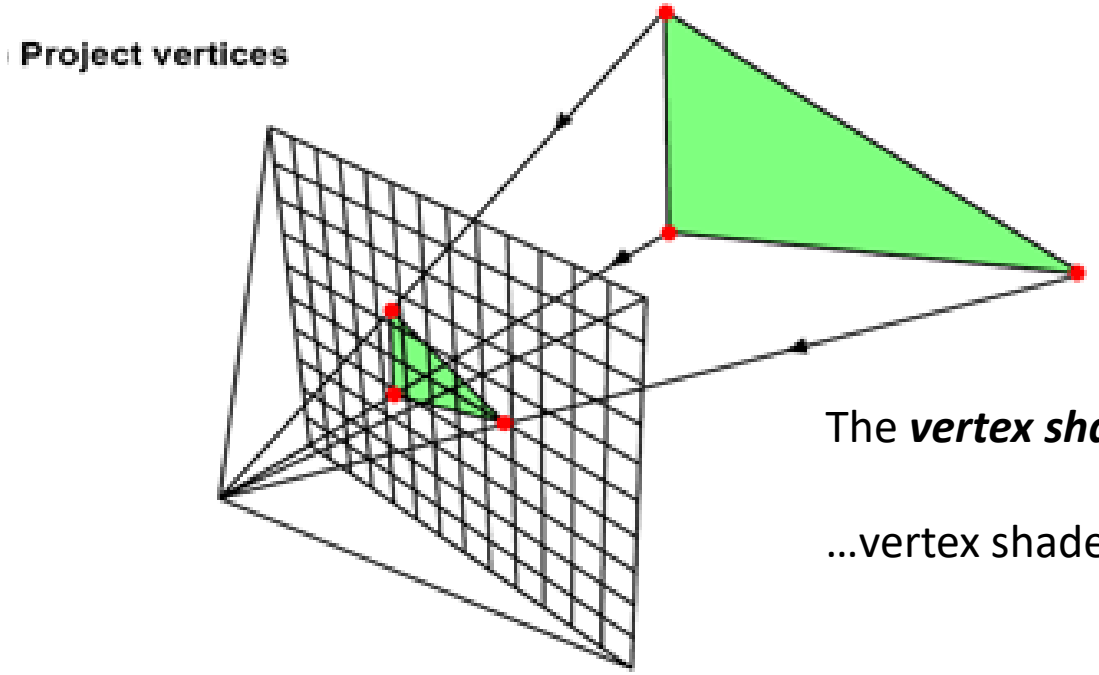
```
// We'll use perspective projection
glMatrix.mat4.perspective(pMatrix,degToRad(90),
                          gl.viewportWidth / gl.viewportHeight,
                          0.1, 200.0);
```

```
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
```

```
var triangleVertices = [
        0.0,  0.5,  0.0,
       -0.5, -0.5,  0.0,
        0.5, -0.5,  0.0
];
```

ILLINOIS

Projection



The ***vertex shader code*** applies geometric transformations to vertex positions

…vertex shaders can do other things as well

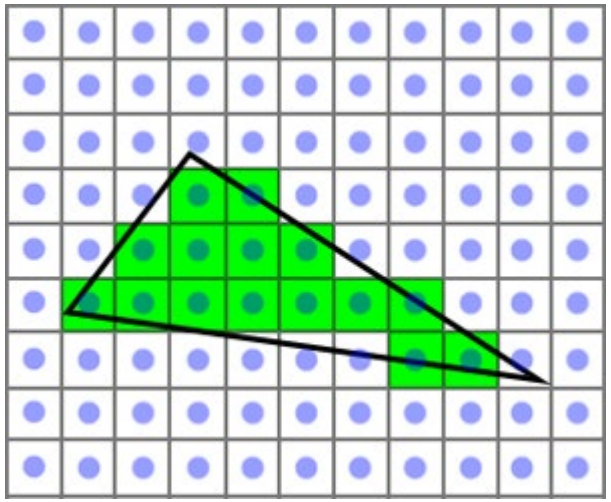
The happens in the vertex shader code and the WebGL library

…on the GPU

ILLINOIS

Rasterization

Which pixel locations are covered by a triangle?

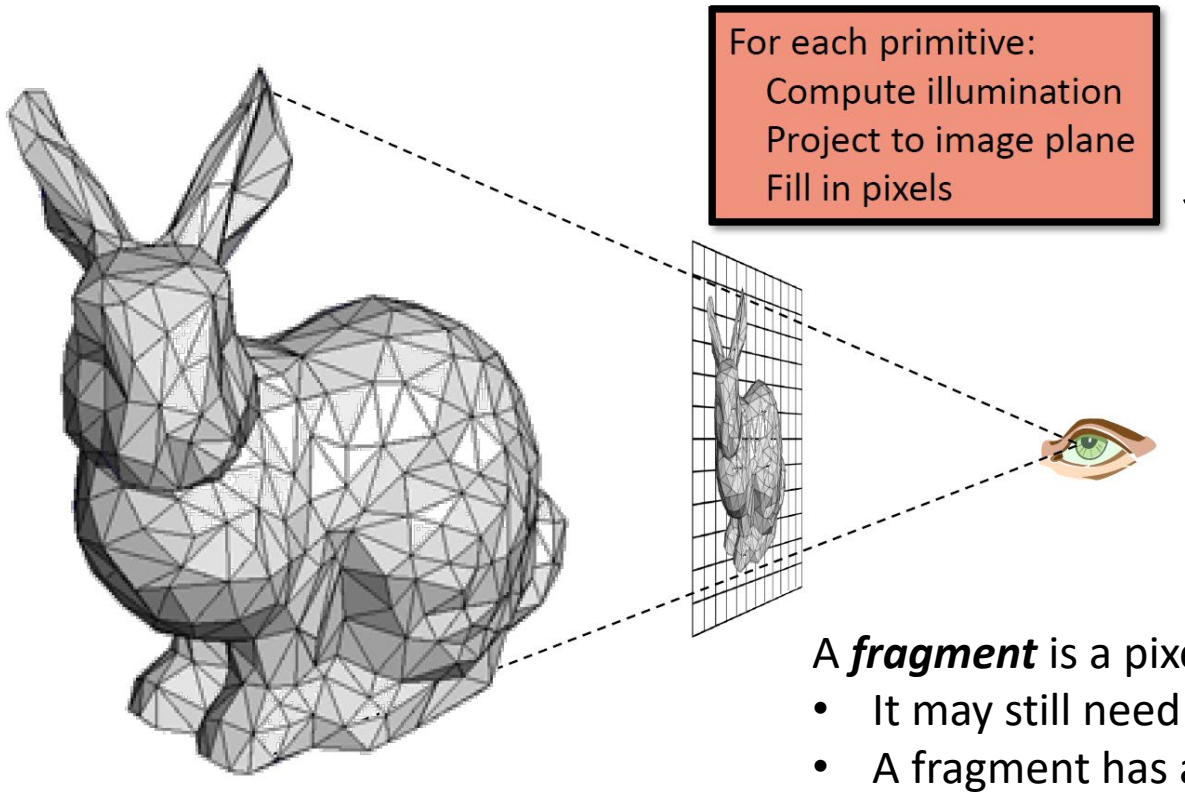


The happens in the WebGL library (…on the GPU)

ILLINOIS

Shading

Shading is the process of computing the color of objects in a scene



You write the code to shade

Fragment shader code finalizes fragment color

A ***fragment*** is a pixel…but not the final pixel at a location

- It may still need to be composited…among other things
- A fragment has a screen space location (e.g. (0,0) lower left corner)
- It has a color…maybe alpha…and a depth value

ILLINOIS

The WebGL Rasterization Engine

- WebGL advantages
  - runs in browser
  - naturally cross-platform
  - don’t need to obtain/build other libraries
  - gives you “windowing” for free
  - easy to publish/share your stuff

- Disadvantages
  - Depends on how you feel about JavaScript
  - Performance can be tricky
  - Pretty low-level

Programming Language for CS 418

- HTML
- JavaScript
- WebGL 2.0
- GLSL ES 3.0 version of the GLSL shading language (**runs on GPU**)

- Chrome as default browser
- Chrome DevTools to debug code
- Some WebGL examples: https://www.chromeexperiments.com/webgl

Firefox is fine too…and probably Edge.
We test in Chrome…

Coding Resources

- We will provide example code

- Mozilla reference/tutorials are very good https://illinois-cs418.github.io/resources

Coding References

HTML

- General HTML information: Mozilla HTML Docs
- How to create input elements in HTML and access input in JS: Mozilla HTML input element documentation

JavaScript

- JavaScript tutorials and reference: Mozilla JavaScript Docs

WebGL

- WebGL tutorials and API reference: Mozilla WebGL Docs

glMatrix JS Math Library

- Use the most recent version of the library glMatrix Library Download
- API Reference: glMatrix Library Docs

- You are expected to use them when completing assignments

WebGL Application Structure

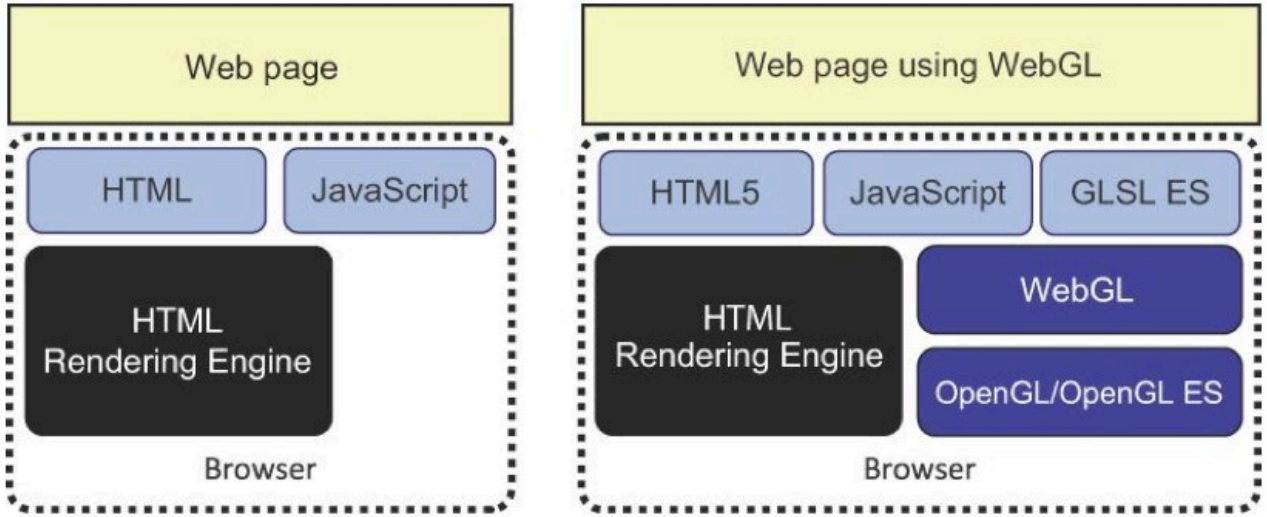


Figure from *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL* by Matsuda and Lea

Your application will generally just have HTML and JavaScript files

WebGL and GLSL

- WebGL requires you provide shader programs
- GLSL OpenGL Shading Language
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- WebGL functions compile, link and get information to shaders

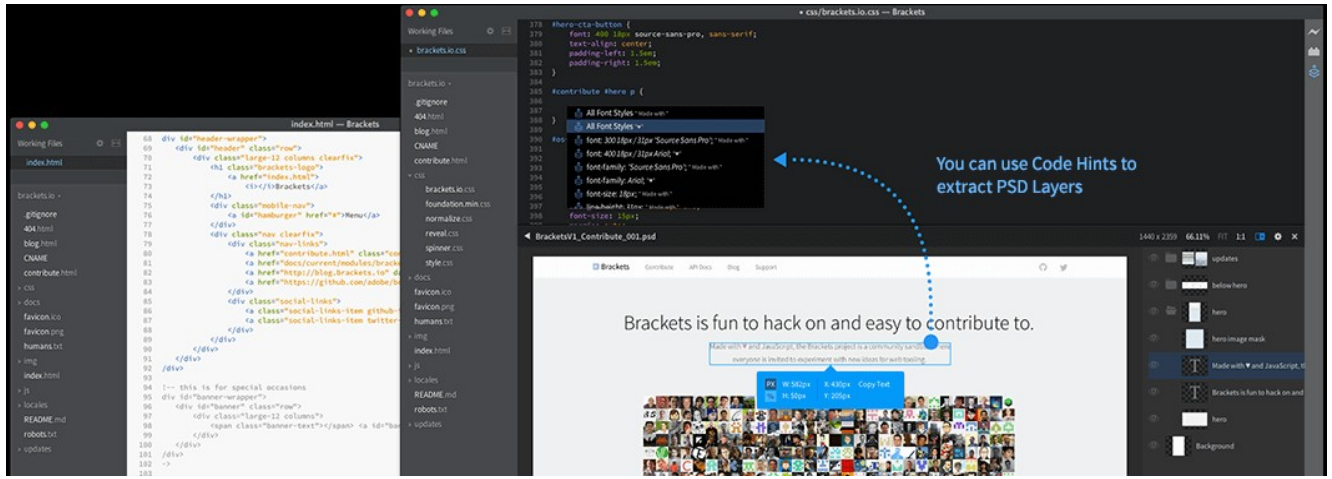
Shaders

- Shader source code will be in the HTML file or a JS file…usually
- Vertex Shaders generally move vertices around
  - Projection, animation, etc.
  - Assign a value to the built-in variable gl_Position
- Fragment Shaders generally determine a fragment color
  - Assign a value to an ***out*** variable

You Need a Text Editor

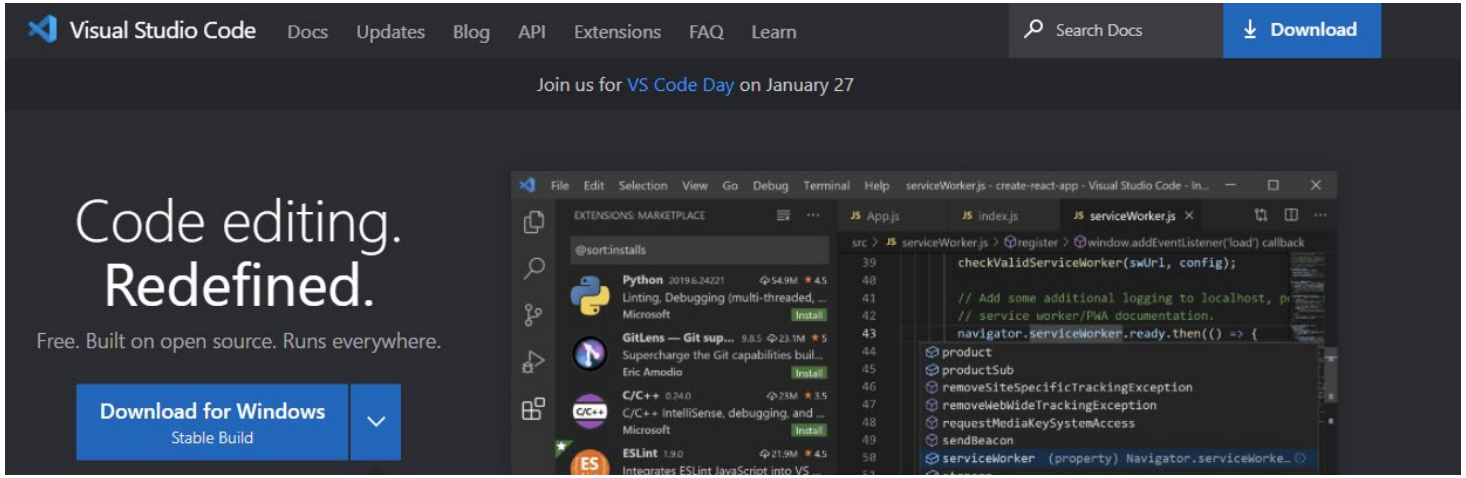
Brackets is a good choice…but whatever works for you is fine

http://brackets.io/



VS Code is good as well

https://code.visualstudio.com/



Time to Write Some HTML

A few notes

- We will keep everything in a single HTML file for this example
- …for larger programs we will separate the HTML and JavaScript

Using WebGL entails writing a bunch of startup code

- Complexity comes from the flexibility of the API

https://illinois-cs418.github.io/Examples/WebGL2/HelloTriangle/HelloTriangle.html

The HTML

We create an HTML page

We create an HTML5 **<canvas>** 500 x 500 pixels which we will draw into.

We give it an id so we can refer to it in the javascript that we will write.

onload specifies an event handler
It will call a JS function named **startup()** when a page loads

```
<title>Hello Triangle</title>

<script type="text/javascript">
```

...some JS code

```
</script>

</head>

<body onload="startup();">
  <canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>
</html>
```

Adding JavaScript

```
/**
 * Entrypoint into the renderer.
 * Runs some initialization code, and then draws a still image to the canvas.
 */
function startup() {
  canvas = document.getElementById("myGLCanvas");
  gl = createGLContext(canvas);
  setupShaders();
  setupBuffers();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  draw();
}
```

gl.clearColor is a WebGL function that sets the initial color of the pixels in the raster

getElementByID is a function that gets us a reference to the canvas created in the HTML document

…the other functions are ones we will write

Getting a WebGL Context

```
/**
 * Creates a WebGL 2.0 context.
 * @param canvas The HTML5 canvas to attach the context to.
 * @return The WebGL 2.0 context.
 */
function createGLContext(canvas) {
  var context = null;
  context = canvas.getContext("webgl2");
  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create WebGL context!");
  }
  return context;
}
```

We need to make sure the browser supports WebGL 2.0

If we get a context, we set the viewport dimensions of the context to match the size of the canvas.

You can choose to use less than the full canvas.

Creating Vertex Shader

```
// Source code for a simple vertex shader. Does no transformations.
var vertexShaderSource =
  `#version 300 es
   in vec3 aVertexPosition;
   void main() {
     gl_Position = vec4(aVertexPosition, 1.0);
   }`;
```

We'll talk more about shaders later but for now you should know:

We need to create a vertex shader program written in GLSL

We will use a JavaScript string to hold the source code for the vertex shader. **We'll see a better way to do this later.**

The shader must assign a value to gl_Position

Our shader basically just takes the position of an incoming vertex and assigns that position to gl_Position.

Creating Fragment Shader

```
// Source code for a simple fragment shader that colors everything white.
var fragmentShaderSource =
  `#version 300 es
   precision mediump float;
   out vec4 fragColor;
   void main() {
     fragColor = vec4(1.0, 1.0, 1.0, 1.0);
   }`;
```

Like the vertex shader program, the fragment shader code is written in GLSL and held in a string.

You can think of fragments as being almost pixels
…they are produced by the WebGL rasterizer and have a screen space position and some other data related to them.

Our shader simply assigns each fragment the same color.

Again, we'll talk more about what the shaders do later…

Compiling the Shaders

```
/**
 * Creates a WebGLShader object representing GLSL code for a single shader
 * stage.
 * @param type The shader stage; either gl.VERTEX_SHADER or gl.FRAGMENT_SHADER.
 * @param shaderSource The source code for the shader.
 * @return The WebGLShader object created.
 */
function loadShader(type, shaderSource) {
  var shader = gl.createShader(type);
  gl.shaderSource(shader, shaderSource);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Error compiling shader" + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
  }
  return shader;
}
```

We have a function that compiles the shader and checks if there were compilation errors.
If there was an error, a JavaScript alert is issued and the shader object deleted.
Otherwise the compiled shader is returned.

Important:

- You can create multiple shader programs
- You can switch which one you use while drawing a single frame…just use the **useProgram** function in WebGL
- Each shader program needs a vertex shader and fragment shader

Linking the Shaders

```
// Create shader objects and compile each of those shaders.
var vertexShader = loadShader(gl.VERTEX_SHADER, vertexShaderSource);
var fragmentShader = loadShader(gl.FRAGMENT_SHADER, fragmentShaderSource);

// Link the shaders together into a program.
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
  alert("Failed to link shader program");
}

// We only use one shader program for this example, so we can just bind
// it as the current program here.
gl.useProgram(shaderProgram);
```

We create a program object and attach the compiled shaders and link.
At this point, we have a complete shader program that WebGL can use.

Creating a Buffer Object

```
function setupBuffers() {
  // Define a triangle in clip coordinates.
  var triangleVertices = [
         0.0,  0.5,  0.0,
        -0.5, -0.5,  0.0,
         0.5, -0.5,  0.0
  ];

  // Create an attribute buffer with position data, and bind it to the vertex array object.
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

  // Send the vertex position data to the buffer.
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW);
  vertex Buffer.numberOfItems = 3;

  //Unbind the buffer for safety
  gl.bindBuffer(gl.ARRAY_BUFFER, null) ;
```

Buffers are used to ship data down to the GPU

A buffer feeds data to attribute variables in vertex shaders

Each element of the buffer has the attribute data for a specific vertex

(e.g. the position of vertex 0 is given by the first 3 coordinates in the buffer)

Attributes

```
  // Create the vertex array object, which holds the state information
  // needed to send a set of attributes to the shader program.
  // Attributes contain data that applies to a specific vertex.
  vertexArrayObject = gl.createVertexArray();
  gl.bindVertexArray(vertexArrayObject);

  // Query the index of the vertex position attribute in the list of attributes
  // maintained by the GPU.
  vertexPositionLoc = gl.getAttribLocation(shaderProgram, "aVertexPosition");

  // Bind the buffer with the data for the attribute
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

  // Tell the attribute how to get data out of positionBuffer (ARRAY_BUFFER)
  var size = 3;          // 3 components per iteration
  var type = gl.FLOAT;   // the data is 32bit floats
  var normalize = false; // don't normalize the data
  var stride = 0;        // 0 = move forward size * sizeof(type) each iteration to get the next position
  var offset = 0;        // start at the beginning of the buffer

  // Binds the buffer that we just made to the vertex position attribute.
  gl.vertexAttribPointer(vertexPositionLoc, size, type, normalize, stride, offset);

  // We must enable each individual attribute we are using.
  gl.enableVertexAttribArray(vertexPositionLoc);

  // Unbind the last bound VAO
  gl.bindVertexArray(null);

}
```

attributes are user-defined variables that contain data specific to a vertex. Like a position in XYZ space.

The ***attributes*** used in the vertex shader are bound to an index (basically a number given to a slot).
Our code needs to know the index associated with the attributes we use in the shader so that our draw function can feed the data correctly.

VAOs

```
  // Create the vertex array object, which holds the state information
  // needed to send a set of attributes to the shader program.
  // Attributes contain data that applies to a specific vertex.
  vertexArrayObject = gl.createVertexArray();
  gl.bindVertexArray(vertexArrayObject);

  // Query the index of the vertex position attribute in the list of attributes
  // maintained by the GPU.
  vertexPositionLoc = gl.getAttribLocation(shaderProgram, "aVertexPosition");

  // Bind the buffer with the data for the attribute
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

  // Tell the attribute how to get data out of positionBuffer (ARRAY_BUFFER)
  var size = 3;          // 3 components per iteration
  var type = gl.FLOAT;   // the data is 32bit floats
  var normalize = false; // don't normalize the data
  var stride = 0;        // 0 = move forward size * sizeof(type) each iteration to get the next position
  var offset = 0;        // start at the beginning of the buffer

  // Binds the buffer that we just made to the vertex position attribute.
  gl.vertexAttribPointer(vertexPositionLoc, size, type, normalize, stride, offset);

  // We must enable each individual attribute we are using.
  gl.enableVertexAttribArray(vertexPositionLoc);

  // Unbind the last bound VAO
  gl.bindVertexArray(null);

}
```

A Vertex Array Object records the state necessary to send a set data to a set of attributes in the shader program

It’s like it records the gl.bindBuffer, gl.vertexAttribPointer, and the gl.enableVertexAttribArray function calls and can replay them

```
/**
 * Renders the image on the screen.
 */
function draw() {
  // Transform the clip coordinates so the render fills the canvas dimensions.
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

  // Clear the screen.
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Make the attribute state for this model current
  gl.bindVertexArray(vertexArrayObject);

  // Render triangles.
  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);

  // Unbind the vertex array object.
  gl.bindVertexArray(null);
}
```

Drawing the Scene

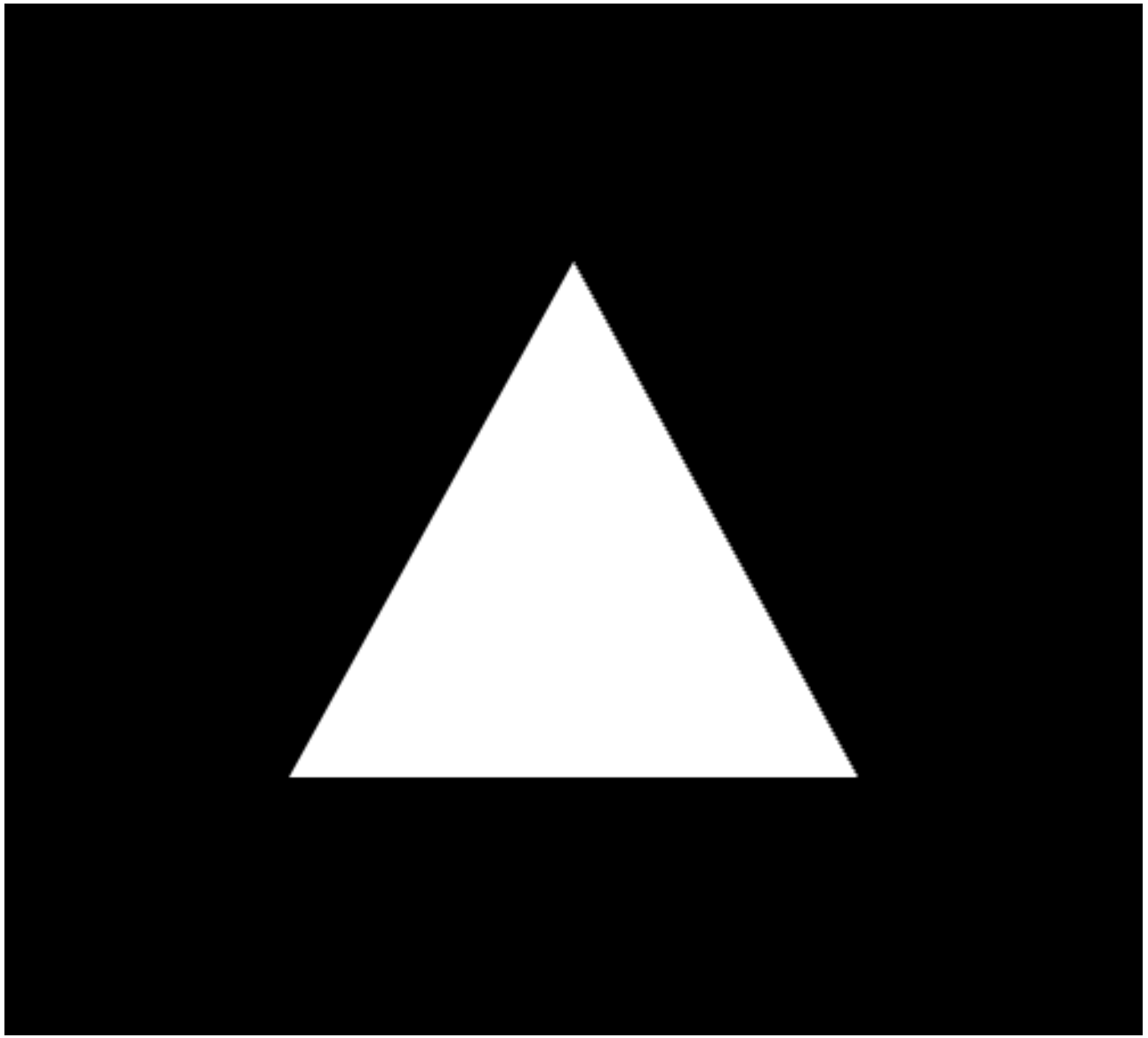
gl.viewport method tells WebGL how to convert from *clipspace* (with coordinates in [-1, 1]) into pixel coordinates.

gl.clear initializes the color buffer to the color set with ***gl.clearColor***.

We then use ***gl.bindVertexArray*** to set up connection between buffers and attributes

And then ***gl.drawArrays*** send data from buffers bound to shader attributes will sent to the GPU to be processed.

Result…



Sure…not super impressive

But consider that we could have drawn 1 million triangles with essentially the same code…..

Things to try….can you…

Change the triangle color?

Add a color attribute?

Change the background color?

Draw multiple triangles?

Figure out what the coordinate system of the canvas is?
Where is the origin?
What are the coordinates of the bottom left corner?
What are the coordinates of the top right corner?