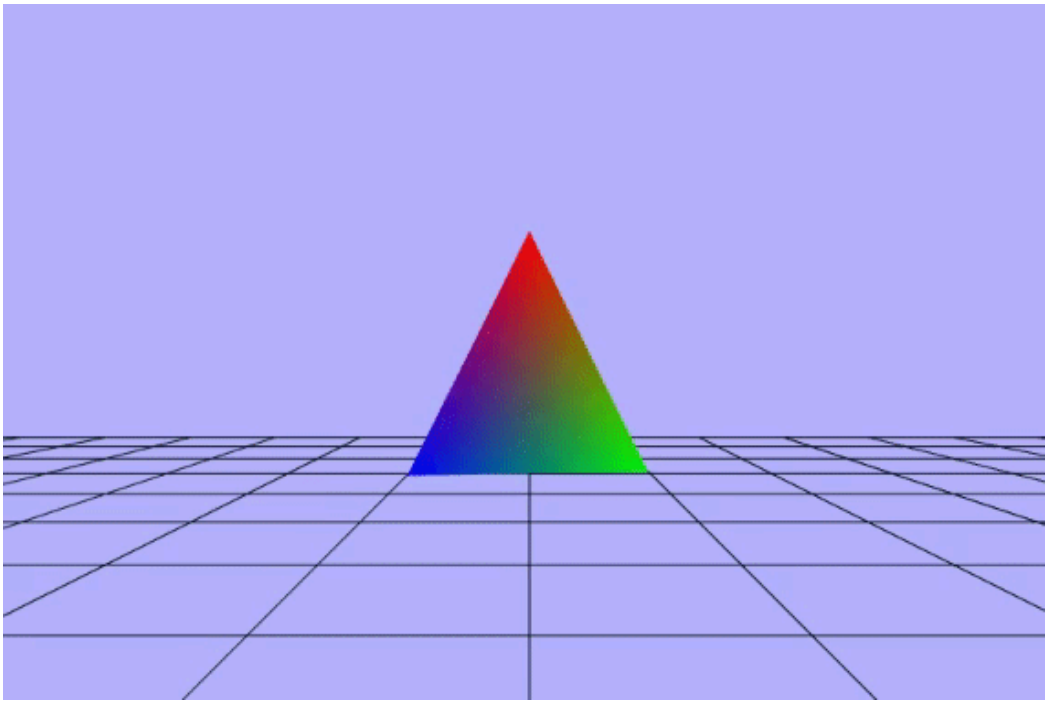
Intro to WebGL Guide I

Overview

In this lab you will be producing a “hello world” of sorts with respect to real-time computer graphics: the spinning triangle. Behold!



By now, you probably understand that there is a lot more that goes into producing a simple image than would appear. Unfortunately, even with this expectation, the reality often exceeds it..

In the past when new game console hardware arrived for developers to start utilizing, a somewhat joking measurement was that of how long it took them to figure out how to render the first triangle (“time to first triangle”). It was a hurdle that had to be jumped because there tends to be a great deal of scaffolding surrounding the basics. Here, most of it is done for you as an example from which you can build off of in order to add **color** and **animation**. Despite this, you will still have to review and understand a significant amount of content.

Associated Reading

Before proceeding, it is assumed that you have read through and are **familiar** with the material presented in the rasterization pipeline lectures and chapter 2 of Real-time Rendering (online or kindle -- click “look inside”). You may also find the early chapters of the WebGL Programming Guide helpful as well as this walkthrough from Indigo Code.

About this Guide

Note, anything that is **underlined in blue is also a link**. These links will make it easy for you to access documentation to WebGL and Javascript functions without having to find them in your book. Here is an example: **gl.clear** ← clickme

About the WebGL API

WebGL can be understood as a state machine. If you modify any of its attributes ("state"), that modification is maintained until you modify that attribute again. Let's look at a concrete example.

Before every frame we render, we clear out the results of the previous one by calling the *clear* function on the gl context (known as the variable "***gl***" in our code).

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

It takes in values that specify which of our frame buffers to clear. In this case, it wipes the color buffer and the depth buffer "clean". However, clearing the color buffer also involves knowing what color to clear it to. That information is not passed into the function. It is inferred based on what the current internal value (the current state) is for clear color. We set this persistent state by using other GL functions. In this case we would use:

```
gl.clearColor(0.707, 0.707, 1, 1.0);
```

This function does not do any clearing. Rather, it stores the color that should be used when the call to gl.clear is made for the color buffer. If we called *gl.clear* after *gl.clearColor* as seen above, the color buffer would be a bluish-purple for every pixel. When calling a WebGL function, always be mindful of what the implied hidden state information it might use.

This may seem like a potentially confusing way to do things. However, there are some good reasons for it. One of them is that specifying all of the values for each function could lead to a cumbersome number of arguments (e.g. calling a function where you had to pass 10 different variables would hardly be an improvement). Another reason is that having special functions for state changes better reflects how the hardware actually works. It makes changing the underlying state an explicit action rather than a byproduct. This in turn enables the GPU to run more optimally (given judicious use).

Structure & Execution Order

All code for this assignment will be done inside of a single html file. See the course website for the files. However, this file will also make use of the ***matrix4.js*** and ***vector3.js*** code that you have previously developed.

The structure of the code is laid out as follows (2 shaders and 7 functions):

<script> *...vertex shader code...* ***</script>***

<script> *...fragment shader code...* ***</script>***

<script>

...Declaration of global variables…

function ***initializeAndStartRendering()***

function ***initGL()***

function ***createShaders()***

function ***createCompiledShader()***

function ***createTriangleGeometry()***

function ***createLineGridGeometry()***

function ***updateAndRender()***

</script>

Many of these functions are incomplete and we will be filling in the missing content as we go. The application starts via the **onload** callback (fired when the page finishes loading) which in turn calls the ***initializeAndStartRendering()*** function. Look at the ***<body>*** HTML tag below.

```
<body onload="initializeAndStartRendering();">
    <canvas id="webgl-canvas" style="border: none;" width="768" height="512"></canvas>
</body>
```

This function is done for you and looks like this:

```
// --------------------------------------------------------------------------
function initializeAndStartRendering() {
        initGL();

        // createShaders();
        createTriangleGeometry();
        // createLineGridGeometry();

        if (gl) {
            updateAndRender();
        }
}
```

initGL() is called first and will be executed just once to perform initialization. Our basic triangle geometry is also already created in ***createTriangleGeometry()***.

updateAndRender() will be called every frame to update our scene and render it. This first call to it kicks off the application loop. It will repeatedly call itself after that to keep generating frames. The remaining setup functions are commented out until we get a chance to finish implementing them.

Basic GL Setup & Initialization

Todo #1 Get access to the WebGL context

WebGL works with HTML5 by utilizing the **canvas**. Take a look at this code which can be found at the **bottom** of the file:

```
<body onload="initializeAndStartRendering();">
    <canvas id="webgl-canvas" style="border: none;" width="768" height="512"></canvas>
</body>
```

Inside of ***initGL()*** you will need to access this html element and get a reference to its WebGL interface. **Use the correct ID.**

```
// ----------------------------------------------------------------------------
function initGL(canvas) {
    // todo get a reference to the canvas object
    var canvas = document.getElementById(/* todo - choose the correct id*/);

    try {
        gl = canvas.getContext("webgl");
    } catch (e) {}

    if (!gl) {
        alert("Could not initialise WebGL, sorry :-(");
    }
}
```

If you put in the correct ID then the canvas should be able to return the WebGL context we will need to interact with the API and get the image below. Otherwise, you will get an alert.

Basic Update & Render

To start, we'd like to make sure we have a working foundation before attempting to send geometry down the pipeline for rendering. Here's how ***updateAndRender()*** (around line 235) looks to start with:

```
// --------------------------------------------------------------------
function updateAndRender() {
    requestAnimationFrame(updateAndRender);

    // todo #2 - specify what portion of the canvas we want to draw
    // (this should be the full width and height of the canvas html element)
    gl.viewport(0, 0, /* width of viewport equal to canvas width*/,
                /*height of viewport equal to canvas height*/);

    // this is a new frame so let's clear out whatever happened last frame
    gl.clearColor(0.707, 0.707, 1, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
}
```

First, notice the function called ***requestAnimationFrame*****.** This is the browser's preferred way to call a function repeatedly for rendering. By calling it, we are letting the browser know that it should in turn call our ***updateAndRender()*** function as soon as it is ready to have us render again.

Todo #2 Set Viewport Dimensions

The idea of a window on the screen that we draw to has some flexibility. It is not assumed that we have the entire screen to work with. The portion of the screen that is available for us gives us the freedom to use it however we want. Maybe we want to only render on the left half… or some corner. In any case, it's our choice. Here, we just want to grab the whole thing that is available to us. In the updateAndRender*()* function, **use the whole canvas width and height for your viewport in the call to gl.viewport.**

```
// --------------------------------------------------------------------
function updateAndRender() {
    requestAnimationFrame(updateAndRender);

    // specify what portion of the canvas we want to draw to (all of it, full width and height)
    gl.viewport(0, 0, /* width of viewport equal to canvas width*/, /*height of
                viewport equal to canvas height*/);

    // this is a new frame so let's clear out whatever happened last frame
    gl.clearColor(0.707, 0.707, 1, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    // render stuff ...
}
```

The last 2 “clear” functions are there to wipe out anything that was previously drawn so we can create a new image to display. Recall that internally, we have multiple screen buffers to control what eventually appears on the screen: the color buffer (the one you normally think of) and the depth buffer which stores how far away from the camera the geometry at each pixel is. We need to make sure each one of them is cleared of its current contents before reuse.

Shaders (how to render)

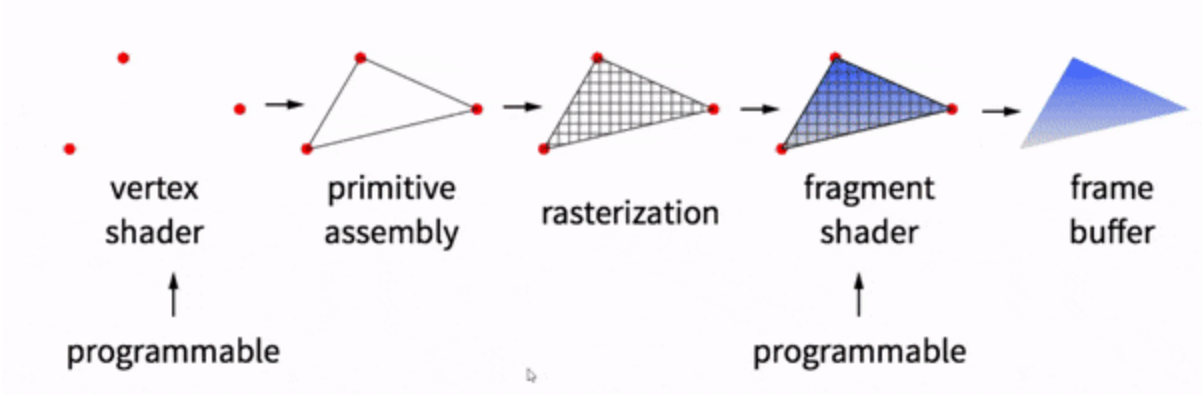
Shaders are code that run on the GPU during specific pipeline stages. They give us the power to customize the pipeline rather than have a more rigid or “fixed” way of processing and displaying our geometry. Providing shader code to WebGL is required and you will not be able to execute the pipeline without it.

Shaders in WebGL use their own C-like programming language called GLSL (GL Shader Language) that is syntactically similar to languages you already know. In the places where we interact with this code, instructions will be given on how to proceed.

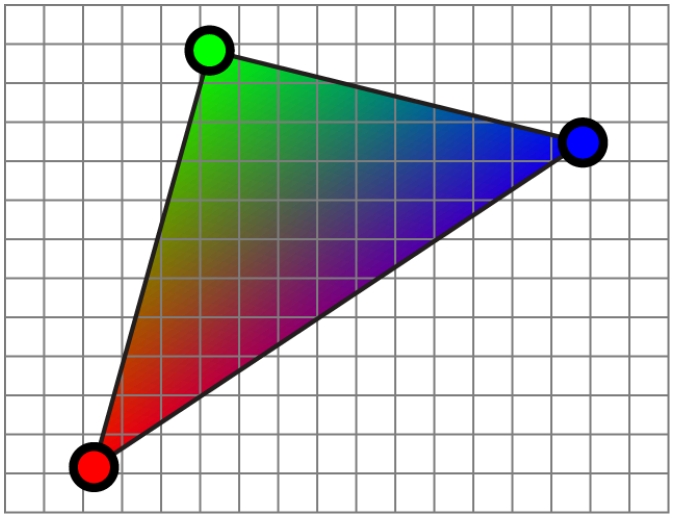
The first shader we encounter, the vertex shader, is ultimately responsible for transforming each vertex from its local space into clip space. It may also compute and pass on other data like color (as we will see later). Note that this shader only gets information about a **single** vertex with no knowledge of what it is a part of (the triangle). To sum it up, the vertex shader is a stage of the pipeline which receives information about a vertex, transforms it, and forwards its results to the next stage (which will start assembling each vertex into the triangles they are composed of).

The next shader we must concern ourselves with, the fragment shader, is ultimately responsible for determining the color of a particular pixel. To do this, it typically uses information that had previously been computed at each triangle vertex that surrounds the pixel. Because the pixel could lie anywhere inside the triangle, it will receive **data** as a **weighted blend** from what was passed on from the surrounding vertices.

Both of these shaders need to be written such that they are compatible with each other. What the vertex shader **outputs** will generally be interpolated (weighted blend) for each pixel inside the triangle and received as **input** to the fragment shader. See the following images for a better idea of how the data flows.



If the only thing a vertex passed on for interpolation was color, you might get something like the image below. The vertex shader transformed the position from local space to clip space and passed along a color (red, green, or blue). The triangle was assembled, the color was interpolated based on its position in the triangle, and the fragment shader applied that color to the pixel.



Todo #3 Acquire Shader Text (shader source code)

The initial version of the shaders has been written for you. We will focus more on the details throughout the rest of the class. Your next job will be to access the raw shader source code text and create a compiled and linked version of that code into a combined "program". This compilation and linking happens at runtime via code you write using the WebGL API.

First, **uncomment** the call to ***createShaders()*** found in ***initializeAndStartRendering()*** so that the changes we make next are actually executed. Next, take a look at the ***createShaders()*** function. The following lines will grab the shader source code text for each shader. As before, **you will need to fill in the correct id to fetch the data**. Note that ***document.getElementById*** is looking for an HTML "element" (<SomeElement>...</SomeElement>).

```
function createShaders() {
  // Get the objects representing individual shaders
  var vertexShaderText = document.getElementById(/* todo - choose correct id*/).textContent;

  // uncommented when directed
  // var vertexShader = createCompiledShader(gl, vertexShaderText, gl.VERTEX_SHADER);

  var fragmentShaderText = document.getElementById(/* todo - choose the correct id*/).textContent;

  // uncommented when directed
  // var fragmentShader = createCompiledShader(gl, fragmentShaderText, gl.FRAGMENT_SHADER);
```

After choosing the correct id, verify your code (by setting a breakpoint or outputting via console.log()) and verify that ***vertexShaderText*** and ***fragmentShaderText*** contain the code for each of the shaders. Make sure there are no errors in the console.

Todo #4 Complete the function to compile the shader code

First uncomment the calls to ***createCompiledShade(...)*** inside of ***createShaders()*** because we will finish implementing it now. Next, uncomment the lines shown below that start with "gl".

```
function createCompiledShader(shaderText, shaderType) {
    var shader = gl.createShader(shaderType);
    // uncomment these lines when ready
    // gl.shaderSource(/* todo arg1*/, /*todo arg2*/);
    // gl.compileShader(/*todo arg1*/);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        console.log(gl.getShaderInfoLog(shader));
        return null;
    }
    return shader;
}
```

Note that this function is mostly written for you. It is meant to take in the raw text of a particular shader's code along with a special value that specifies what type of shader it is ("shaderType" which should be either gl.VERTEX_SHADER or gl.FRAMENT_SHADER), compile it, and return the compiled result.

Your task is to understand what the code is doing and **fill in the missing function arguments**. Consult Mozilla's API documentation via all of the links in this document, your textbook, or google. There is nothing to verify the correctness but do make sure no errors are present in the console.

Todo #5 Link Shader Program

Both of your shaders are meant to work together at different parts of the pipeline when rendering a **single** object. For instance, when we render a triangle, first the vertex shader will execute for all of its vertices and later, the fragment shader will run for pixels that lie within that triangle on the screen. In order to specify that they are part of a complete package where one is meant to follow the other, we have to bundle them in a "program".

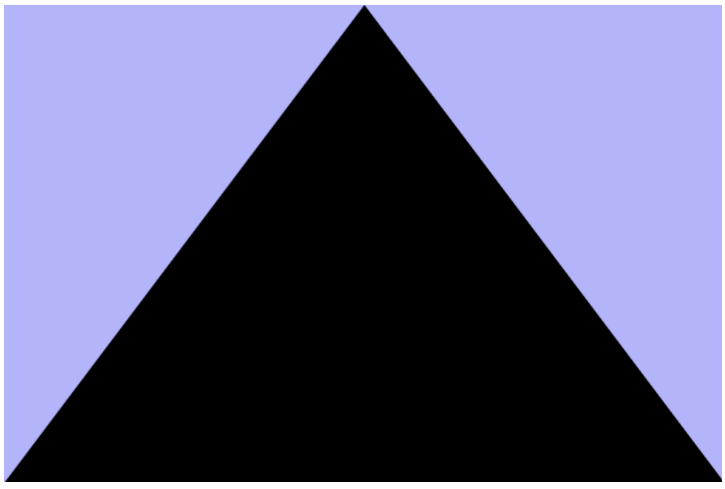
First, **remove the "return" statement** that is in the middle of ***createShaders***() in order to allow code execution to continue. Then as before, use all available resources to fill the correct function arguments.

```
// Create an empty gl "program" which will be composed of shaders
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, /* todo - arg2*/);
gl.attachShader(shaderProgram, /* todo - arg2*/);

// Tell gl it's ready to go, link it
gl.linkProgram(/*todo arg1*/);
if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert("Could not initialise shaders");
}
```

Run it once again and verify that no errors have occurred.

This is all part of the scaffolding around our program to get it up and running. Taking a small amount of time to focus on it will help give you a feel for what it's like to interact with the WebGL API. Now that the shaders are set up, **uncomment** ***renderTriangle(0)*** in ***updateAndRender()*** and you should see the image below.



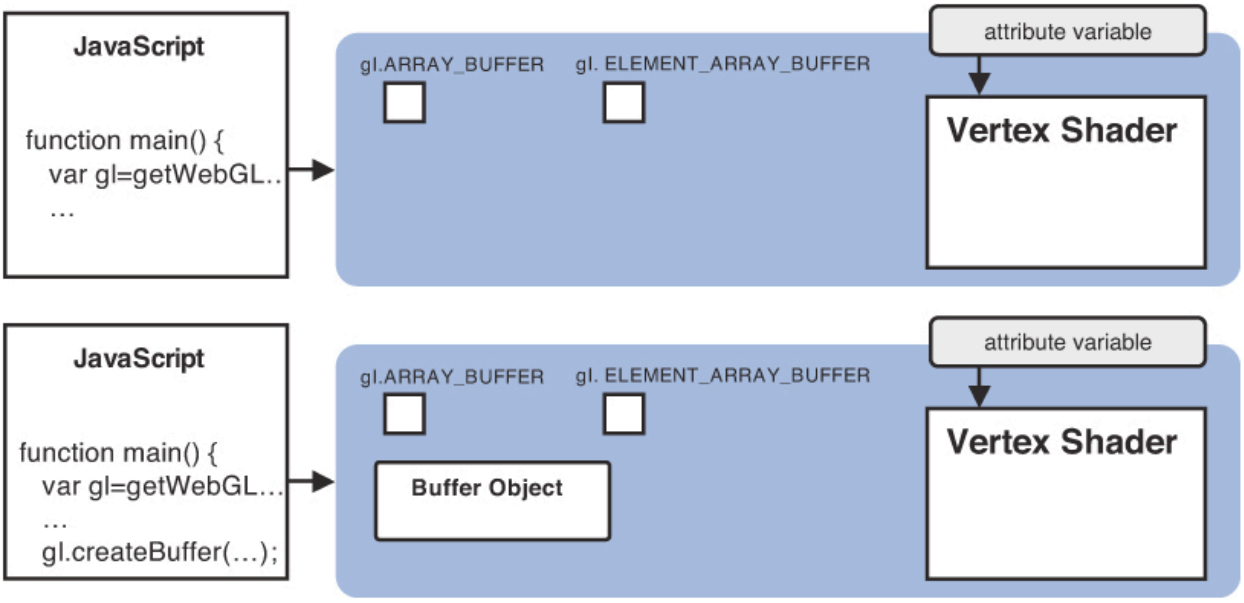
Geometry & Buffers (what to render)

Part of what allows graphics processing to be fast is specialized hardware. The GPU (graphics processing unit) has its own set of local memory that is completely separate from the RAM you are already familiar with on the CPU side (clearly visible on your motherboard). Graphics APIs like WebGL provide a way for you to utilize that memory on the GPU side (inside the graphics card) so that operations can be carried out as fast as possible. To do this, note the following steps.

Pay close attention to the images which depict the current state inside WebGL. See chapter 3 of the WebGL Programming Guide for more detail. Below is an overview of the steps you will take along with visuals of how the WebGL state is affected.

1. Creating a buffer (allocate memory on video card)

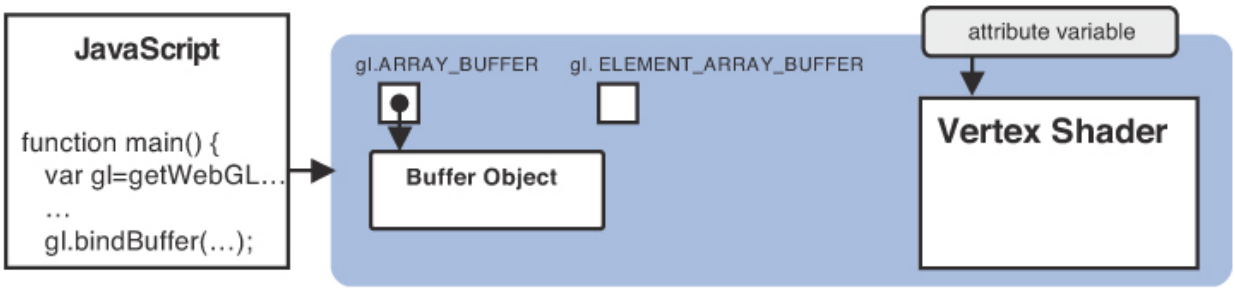
Note that the image is in 2 parts: the top half is a representation of the starting state and the bottom half is a representation after calling ***gl.createBuffer()***.



var vertexBuffer = gl.createBuffer();

2. Binding the buffer (making this buffer the active one).

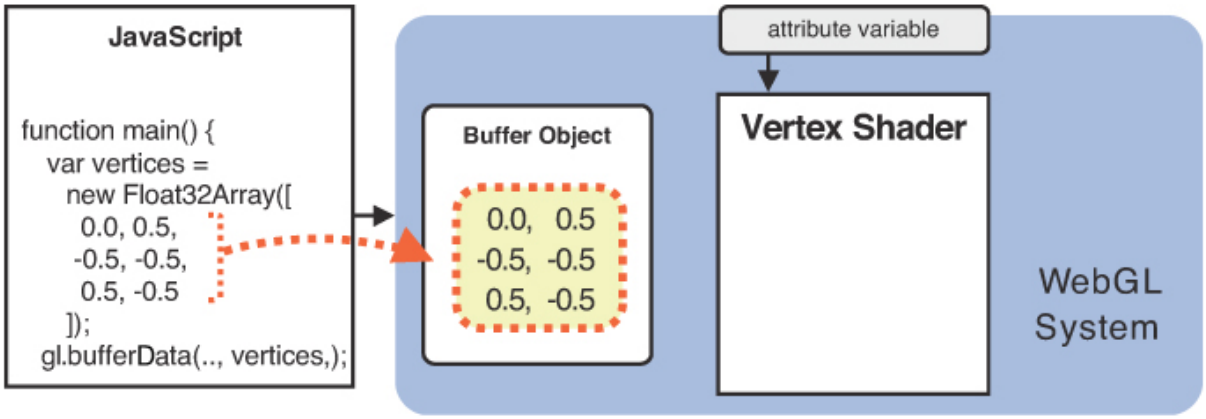
WebGL commands do different things based on what "state" the API is in. Binding a buffer let's future commands implicitly know what buffer they should use without passing the buffer in as an argument to it.



gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

3. Buffering the data

This command will take the data that you have available on the CPU side and move it into the currently bound buffer on the GPU side.



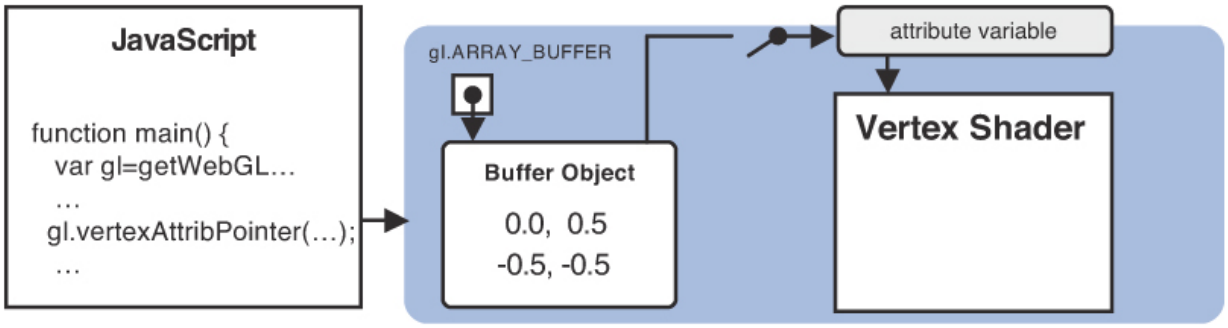
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

4. Assign the buffer contents to a shader vertex variable (“attribute”).

A buffer is just a chunk of memory. It doesn’t specify what the data in the chunk means. The data that we generally put in these buffers can come in different shapes and sizes. For example, if the buffer was full of vector information we would need to specify if each vector is 2 numbers, 3 numbers, or 4 numbers. This gl.***vertexAttribPointer()*** function allows us to tell WebGL how to map the data that is nested inside the chunk.

The buffer also needs to be associated with a variable in our vertex shader in order to access it on the GPU side. But first we need to get a reference to the shader variable. You can see this is done for you if you look toward the end of the ***createShaders()*** function.

```
shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVertexPosition");
```

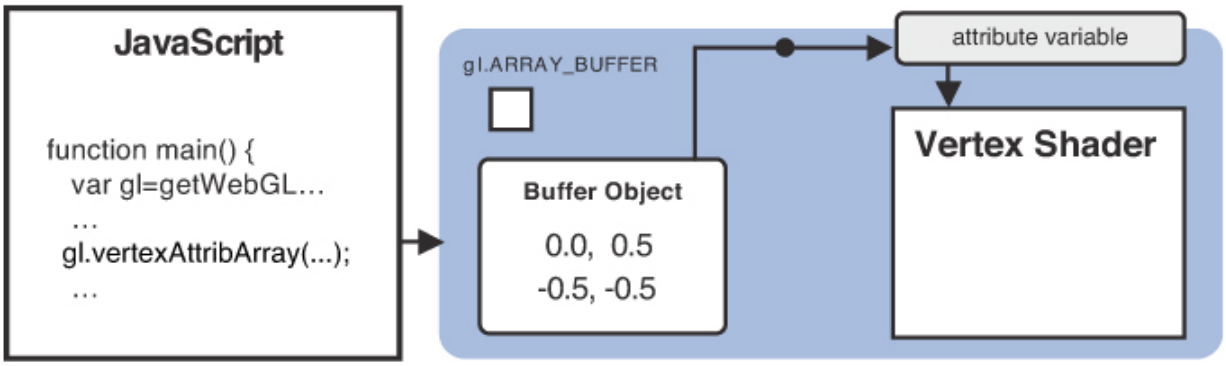


gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute, 2, gl.FLOAT, false, 0, 0);

In this example, the buffer only contains 2d positions and is associated with a variable of matching type in the vertex shader where it is declared as ***attribute vec2 aVertexPosition;*** (see in the vertex shader code at the top of the file).

5. Enable (turn on) the vertex attribute that you just assigned.

   Notice the illustration of this as a switch ("off" in the image above, "on" in the image below).



gl.enableVertexAttribArray(***shaderProgram.vertexPositionAttribute***);

It may be useful to consult your textbook or other resources to get a better feel for the aforementioned steps. We will start putting it to use below.

Creating and Filling the Position Buffer

First to make it easier to keep track of everything needed to render our triangle we have created a custom Javascript object called ***triangleGeometry*** that will contain related data. It is at the top of the file and looks like this:

```
// The core data associated with the triangle that we will need to render it
var triangleGeometry = {
    worldMatrix: null,   // the matrix that transforms this object from local space to world space
    shaderProgram: null, // the shader program of "how" to render this object (vertex & fragment)
    positionBuffer: null, // the buffer that holds all of the position data
    colorBuffer: null,    // the buffer that holds all of the color data
    bufferItemCount: null, // how many vertices the buffer contains (bookkeeping)
};
```

Next, take a look at the function ***createTriangleGeometry()***. Inside this function we see following sequence:

1. WebGL: creates a new buffer
2. WebGL: binds the buffer (make it the current "thing" in the WebGL associated state)
3. Javascript: create an array of positions (3 floats each)
4. WebGL: send our array of vertices to the GPU (bufferData)

This first part of the code is done for you (as shown below). Later you will write the code yourself to send color information to WebGL in a similar fashion.

```
// create the position information for this object and send it to the GPU
var newBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, newBuffer);
var vertices = [
    0.0,  1.0,  0.0,      // (vertex 1: x, y, z)
    -1.0, -1.0,  0.0,     // (vertex 2: x, y, z)
    1.0, -1.0,  0.0,      // (vertex 3: x, y, z)
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
```

However, before you move on to color, it is important to keep track of a few more things so that we have what we need later to command WebGL to render. Remember the ***triangleGeometry*** Javascript object we made right before this (top of the file)? Note how the values are set inside of ***createTriangleGeometry()***.

1) First, we will create a matrix to be used with this object so that we can transform its vertices independently of any other object (will transform vertices from local space to world space when applied).

```
// allocate a matrix that we will reuse for this object
triangleGeometry.worldMatrix = new Matrix4();
```

2) Next, we store which shader program we intend to use with this object.

```
// specify the shaders which carries the instructions for "how" to render
triangleGeometry.shaderProgram = shaderProgram;
```

3) We should also remember the buffer that we just created and keep a reference.

```
triangleGeometry.positionBuffer = newBuffer;
```

4) Finally, we need to store how many vertices are in our buffer.

```
triangleGeometry.bufferItemCount = 3;
```

Mapping Vertices & Matrices to the Vertex Shader

Getting References to Vertex Shader Attributes

In order to actually process the data in our vertex buffer, we will need to make sure that it becomes available in the vertex shader through the appropriate variables defined there.

Recall the simple vertex buffer we created storing vertex positions. **Elements of a vertex** are known in WebGL as "**attributes**". For now we only have a single buffer containing position attributes.

Take a look at the layout of the vertex shader.

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;

  uniform mat4 uWorldMatrix;
  uniform mat4 uViewMatrix;
  uniform mat4 uProjectionMatrix;

  varying vec3 vColor;

  void main(void) {
   ... shader implementation...
  }
</script>
```

The shader defines a variable prefixed with the "*attribute*" keyword of type vec3 (this is similar to Vector3 that we created, only this one is intrinsically built into the shader language GLSL).

Notice it matches with the type of data in our position vertex buffer (3 floats). Unfortunately, WebGL doesn't automatically know that and we must explicitly tell it to map our data to those variables.

Earlier we did the following:

1) Acquired the shader source code as text and compiled it
2) Created a shader program and attached our compiled shaders to it
3) Linked our shader program to get its final state (compatible set of vertex and fragment shaders)

That got our shaders ready to use but they're still not connected to the data in our vertex buffer in any way. We need to get references to the variables inside the shader in order to make that happen. We do this using the following (found in ***createShaders()***):

```
// Make sure the WebGL state knows this is the program we will now be working with
gl.useProgram(shaderProgram);

// Get a reference to the attribute "aVertexPosition" from the current program
shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVertexPosition");

// Turn on this attribute
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
```

The *getAttribLocation* will get us a reference to the variable on the shader program with the matching name (seen here as the string literal "aVertexPosition").

In the code above, we are using the lenient power of Javascript to create new variables on the shaderProgram object to store the reference. That object doesn't store these things by default, we simply created those variables on the existing ***shaderProgram*** object because it was convenient.

To further clarify, being an "***attribute***" means to be a part of a vertex. Another way to think about a vertex is as being all of the data that is specified along with a particular position. The vertices we specify for this assignment will eventually have 1 additional attribute: color. However, many extra pieces of data can also be grouped as part of a vertex. Typical examples are the normal vector (the direction away from the surface at the vertex position) and texture coordinates (which we will discuss later).

Getting References to Vertex Shader Uniforms

In addition to the attributes, it is also necessary to be able to render using additional data that stays constant (doesn't change with every vertex) and is not even a part of a vertex.

For instance, when we draw a triangle, we execute the vertex shader once for each vertex in our triangle (3 times). The vertex shader is primarily responsible for the geometric transformation; taking the position part of the vertex from its local object space to projection space (clip space). This requires the shader to know about the matrices (world, view, projection) used which **DO NOT change** per vertex but rather stay constant for every vertex as this object is being rendered. Each vertex position will be transformed by **the same matrices**.

Shader variables that stay constant for every run of the shader on a single object are called "***uniforms***". Contrast this with "***attribute***" variables which will be different for every run of the vertex shader because each run operates on a different vertex.

As before, we will get references to our uniforms in a fashion similar to attributes.

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;

  uniform mat4 uWorldMatrix;
  uniform mat4 uViewMatrix;
  uniform mat4 uProjectionMatrix;

  varying vec3 vColor;

  void main(void) {
    ... shader implementation...
  }
</script>
```

Much like we used *gl.getAttribLocation* before, this time we use *gl.getUniformLocation*.

We store references to these uniforms in the following places (at the bottom of ***createShaders()***):

```
shaderProgram.worldMatrixUniform
shaderProgram.viewMatrixUniform
shaderProgram.projectionMatrixUniform
```

Rendering (issuing commands)

Inside the ***renderTriangle()*** function we will be doing the following (next todo):

1) Specify where this object is in the world (set values on its world matrix)
2) Send the world, view, and projection matrices to the shader (via uniforms)
3) Bind the triangle's buffers (tell WebGL what we are going to be working with)
4) Tell WebGL how to map the position part of the vertex buffer to the shader
5) Submit the draw call

Todo #6 Transform the Triangle: Matrices

We will not be manipulating the camera (view) for this assignment so we can assume that it is at its default location: the origin. In WebGL the basis vectors for view space will be such that x is to the right, y is up, and z is pointing backwards or out of the screen. That means that -z is forward.

At this point we will use all of the information that we conveniently stored on the ***triangleGeometry*** Javascript object, starting with ***triangleGeometry.worldMatrix***. Perform the following 2 operations:

1) **Reset the matrix back to identity** (we don't want previous values to affect this render)
2) **Make it a translation in the -z direction of [0, 0, -7]** (7 units in front of the camera)

The **view** and **projection** matrices are the transforms that are associated with the camera and independent of any one object. However, for now we will set them here.

The view matrix (created at the top of the file) will remain unchanged but if you want to be extra careful, you could reset it to identity.

Unlike our default view (identity), the perspective projection matrix must be set. Use the ***makePerspective*** function that is a part of Matrix4. It takes 4 arguments consisting of vertical field of view, aspect ratio, near plane distance, and far plane distance.

Set the global variable *projectionMatrix* using the following values:

Vertical FOV: 45 degrees

Aspect Ratio: you will have to compute this based on the width & height of the canvas

Near plane distance: 0.1

Far plane distance: 1000

At this point we've got our matrices in the correct form, they will be sent to the shader using the code below.

```
// Send our matrices to the shader
gl.uniformMatrix4fv(shaderProgram.worldMatrixUniform, false,
                    triangleGeometry.worldMatrix.clone().transpose().elements);

gl.uniformMatrix4fv(shaderProgram.viewMatrixUniform, false,
                    viewMatrix.clone().transpose().elements);

gl.uniformMatrix4fv(shaderProgram.projectionMatrixUniform, false,
                    projectionMatrix.clone().transpose().elements);
```

The *uniformMatrix4fv* function will take the float array from our ***Matrix4*** object and send it to the corresponding **uniform** variable in our shader. There is one oddity to point out here. Notice the call to clone() and transpose(). For simplicity, we previously coded our matrix from a row first perspective like this:

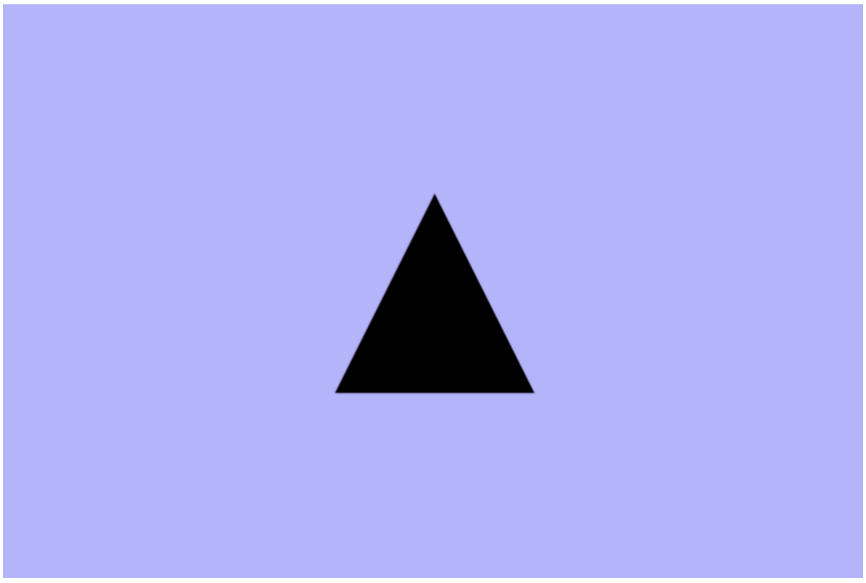
$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

For performance reasons, WebGL actually wants it in the following order:

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

This is the reason for the transpose. The clone is there to make sure that we don't change the original matrix via the transpose. It's not ideal but we are not worried about the performance cost here. It is fine for small learning applications like this.

Next we let WebGL know which vertex buffer we're working with by "binding" it again. That way, when we call the functions in the next todo below, it will apply to the correct object/buffer. Recall how we did the bind the first time? If you did this correctly, your triangle should now be correctly situated in the world and look similar to this:



Rendering the Triangle: Issuing the Draw Call

The function that actually issues the command to WebGL to render using the current state is as follows:

```
gl.drawArrays(gl.TRIANGLES, 0, triangleGeometry.bufferItemCount);
```

Enhancing the Triangle

Todo #7 Create and Fill the Color Buffer

Inside of the ***createTriangleGeometry()*** function, repeat the process that we saw for creating the position buffer; only this one will be used for color. It should consist of 3 colors (9 floats). Each color is composed of 3 color channels (RGB) ranging from 0 to 1. The first color should be red, the second green, and the third blue. Make sure to store a reference to this new buffer on the triangleGeometry object as shown below.

```
triangleGeometry.colorBuffer = newColorBuffer;
```

Nothing will have changed yet as we have yet to do anything with this buffer. **Open your console in Chrome and make sure that there are no errors.**

Todo #8 Add Vertex Color to the Vertex Shader

Currently the vertex shader has a single attribute: ***aVertexPosition***. Add a new attribute to the shader called ***aVertexColor***. You will also notice another type of variable in the shader marked as "**varying**". A "varying" value is one that is meant to be set in the vertex shader and interpolated across the triangle such that the fragment shader receives a weighted-blend based on how close it is to each of the surrounding vertices. Here we want to pass our vertex color attribute through as a varying so that it will affect pixel color. In summary, for this todo you will need to:

1) Declare vec3 attribute called ***aVertexColor***
2) Inside of the vertex shader, assign ***aVertexColor*** to ***vColor***

Todo #9 Acquire a reference to the color shader attribute

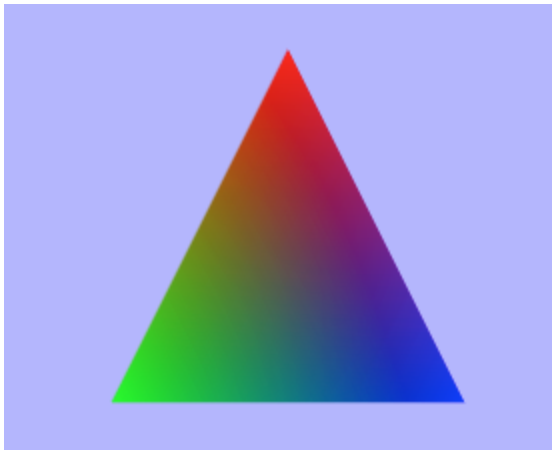
Inside of createShaders, do the following:

1) Use getAttribLocation to get a reference to aVertexColor
2) Store the reference at shaderProgram.vertexColorAttribute
3) Use enableVertexAttrib array to turn the attribute on

After this, we will have temporarily caused an error and no longer be able to see our triangle because we have essentially turned on the shader variable but not connected anything to it. In the console you will see something like: WebGL: INVALID_OPERATION: drawArrays: no buffer is bound to the enabled attribute. That message leads us right into the next todo.

Todo #10 Bind the color buffer to the attribute

Inside of renderTriangle, bind and set the attribute pointer for your color buffer. This should be very similar to what we did with the position buffer. If you did this correctly, the error should go away and we should get our color interpolated result.



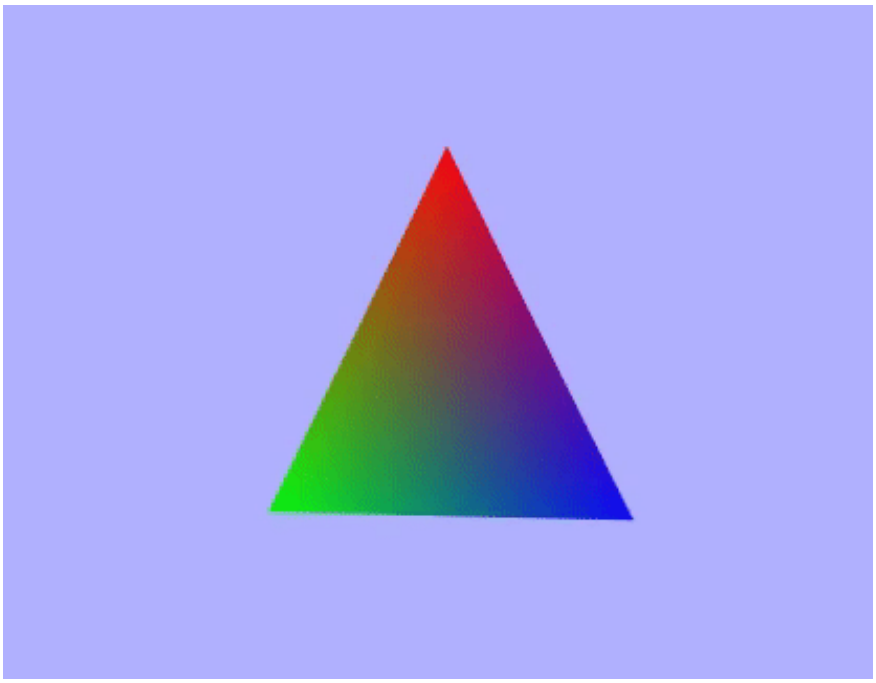
Rotating the Triangle

Todo #11 Rotation

For the moment, let's ignore the timing aspect of generating a rotation that animates smoothly over time and just get something working. At the beginning of the ***renderTriangle()*** function we set up our triangle's world matrix. First we cleared it to identity and then we translated it. We will also want our triangle to rotate about its origin and not [0, 0, -7] so make sure to apply your rotation before the translation. I recommend using the ***makeRotationY*** function on your **Matrix4** object. However, note that ***makeRotationY*** will overwrite everything in your matrix (not multiply anything).

To create an animation, the rotation should increase every frame. Every time your ***renderTriangle()*** function is called, increment the value that is passed to the function. I recommend doing this based on time by calculating the number of seconds that have elapsed since you started running the app (*secondsElapsedSinceStart*) and use the computed degrees value to create the rotation.

When you run, the **triangle should be rotating now**.



Todo Commit

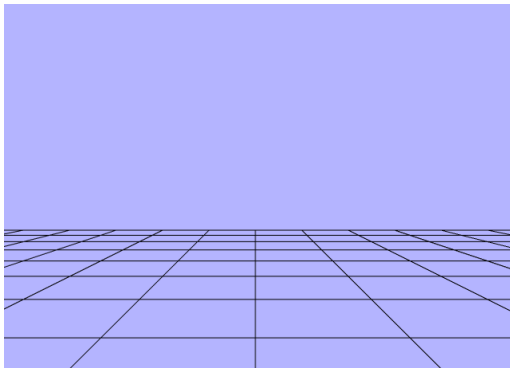
Don't forget the importance of saving your progress. Do this by committing changes to your GitHub repository and pushing those changes to Github. Commit with the message "B portion completed". 5 points will be deducted from the "A" portion for not having this.

Implementation “A”

Adding the Line Grid

A triangle by itself doesn’t do a good job of helping us judge space or see the effects of perspective. If we were to start moving the camera around at this point it would be easy to get lost once the triangle is no longer in sight (you’d only see blue even though your position and orientation would actually be changing).

To alleviate this, we will draw grid lines for the horizontal xz plane like so (as seen from a viewpoint roughly in the center of the grid):



Todo #12 Create and render a Line Grid (10 points)

Similar to how we created the triangle, we will create a grid made up of criss-crossing lines. There will be 21 going across and 21 going back and forward. Each line will be 20 units in length and they should cover a square area. Here is the part for you to fill out in *createLineGridBuffer*:

```
// specify the horizontal lines
for (var i = -10; i <= 10; ++i) {
    // add position for line start point (x value should be i)
    // add position for line end point (x value should be i)
}

// specify the lateral lines
for (var i = -10; i <= 10; ++i) {
    // add position for line start point (z value should be i)
    // add position for line end point (z value should be i)
}
```

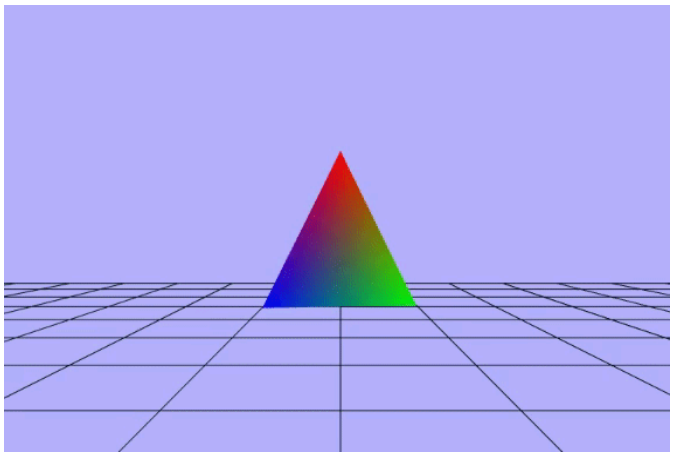
The rest follows the same sequence of operations as we did with the triangle earlier with the exception being that we’re creating **pairs** of vertices as lines rather than **triplets** as triangles.

Note that the line grid is meant to serve as our ground plane and lie underneath the triangle. Setting your y-coordinate to -1 will satisfy this as that is how far in the negative y direction it goes. Also, the line grid will not need to be moved with the world matrix. It is already created where we want it to go so you can just use the identity matrix for your world matrix.

Finally make sure the code you just wrote is running. You may need to **uncomment** *createLineGridBuffer()*.

Render the Line Grid

This will be very similar to rendering the triangle only now you must add the code. See the *renderLines()* function and fill it in as appropriate. You should get something like this:



You might notice something that doesn't quite match: lines drawn over your triangle instead of respecting depth as we described when going over the z-buffer algorithm. That's because the depth test (which uses the z-buffer) needs to be enabled. Oddly, it's off by default. Somewhere in your initialization code add the following:

```
gl.enable(gl.DEPTH_TEST);
```

Grading

Completing everything labeled as required for a "B" will net 85% of the available points. The remaining todos must be completed for the remaining 15%.

Bonus

Todo #1 Scrolling Colors (+5 points)

Truth be told, writing fragment shader code is really fun. This bonus is about the joy of playing with the code. Simple things like messing with color can be delightful.

Currently the fragment shader takes the color that was interpolated from the color at each vertex and displays it directly. Instead, we would like our colors to animate based on time. To do this you will have added one extra uniform to your fragment shader (no need to declare this in the vertex shader).

```
uniform float uTime;
```

Then you will have to do all of the legwork you originally did for the rest of your uniforms such that you get a reference to it in javascript and then update its value every frame.

Acquiring the reference will work in the same manner as before but pushing those values from Javascript to GLSL will be slightly different.

Before you used:

```
gl.uniformMatrix4fv(shaderProgram.worldMatrixUniform, false, /* a float array from a matrix */);
```

To send a single float you will use something like the following:

```
gl.uniform1f(shaderProgram.timeUniform, secondsElapsedSinceStart);
```

Now it's up to you to use that constantly updating value of time along with your color to produce some more interesting results.

Note that colors above 1.0 will be capped so you will see no animation. You will need a function that maps your time value to something in the 0 to 1 range.

Here are some possible functions available in GLSL to help with that:

- fract, sin, cos

Here is my result using fract: