WebGL & Three.js Tutorial

What is WebGL?

WebGL (**Web G**raphics **L**anguage) is a low-level 3D graphics API based that is designed to run in the web browser. WebGL is based on OpenGL ES 2.0.

How It Works in the Browser

WebGL exposes itself through the HTML 5 `<canvas>` element. JavaScript is then used to setup and control the WebGL drawing on the canvas.

1. Raw WebGL

The first step in this tutorial will be to look at how to setup WebGL rendering in the browser with a simple introduction to calling WebGL draw functions

This section uses **01_WebGL_StartingPoint**

Take a look at the three included files and see what's going on:

1.1. index.html

```
<!DOCTYPE HTML>
<html>
<head>
    <!-- here we include out main css and javascript files-->
    <script type="text/javascript" src="js/main.js"></script>
    <link href="css/main.css" rel="stylesheet" type="text/css"/>
</head>

<body onload="onPageLoaded()">

    <!--just a nice title for the page-->
    <h1>WebGL StartingPoint</h1>

    <!--
    the canvas element only shows interanl html when it's not supported
    the width and height properties represent the renderable area of the canvas
    **note that this is different than the css width and height of the element
      on the page
    -->
    <canvas id="webgl-canvas" width="960" height="540">
        Sorry, the canvas element is not supported by your browser.
    </canvas>
</body>
</html>
```

1.2. main.css

```
html, body{
    width:100%;
    height:100%;
    border:none;
}

h1{
    width:100%;
    text-align:center;
    font:30px sans-serif;
}

/*
the css width and height of the
canvas should match the elements
width and height properties for best
results

here we are centering the canvas
and giving it a background color that we
will recognize
*/
#webgl-canvas{
    margin-left:calc(50% - 480px);
    width:960px;
    height:540px;
    background-color:#F0F;
   }
```

1.3. main.js

```
//we create global variables which point to
//the canvas DOM element and the webGL
//drawing context for the canvas
var canvas,
    gl;

//called when the page is done loading
function onPageLoaded()
{
    //we store the canvas DOM element in our variable
    canvas = document.getElementById("webgl-canvas");

    //this function gets the drawing context for the
    //canvas element, it will return an undefined
    //object if it fails
    gl = canvas.getContext('webgl');

    //a simple check to make sure the context
    //was successfully initialized
    if(!gl)
    {
        alert("webgl not supported");
    }

    //requestAnimationFrame lets the browser
    //know that we have drawing to do
    //we pass it a callback function
    window.requestAnimationFrame(update);
}

function update()
{
    //this allows us to keep the loop going
    window.requestAnimationFrame(update);

    // update code here //


    //////////////////////

    render();
}

function render()
{
    // render code here //


    //////////////////////
}
```

2. Putting Color on the Screen

```
// render code here //

//as with OpenGL, setting values is usually a
//function call in WebGL
//clear color sets the color that is used to
//clear the frame
gl.clearColor(1.0, 0.0, 0.0, 1.0);

//clear is the function we call to clear the frame
//is takes in a bit mask that describes which buffers to clear
gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT );

//////////////////////
```

If we open index.html, you should see that the canvas is now red from our clear call.

3. Beyond the Color

This is as far as we will go in raw WebGL code. We have learned about the rendering pipeline and WebGL requires that you are directly involved with each one.

If you are interested in writing raw WebGL, some of the steps to follow are:

- ⇒ SETUP
  - Use gl.enable to turn on necessary features for your project (depth testing, lighting etc)
  - Write and compile shader programs (vertex & fragment) to use for rendering
  - Setup geometry data buffers and push them to the graphics card
  - Setup model and view matrices
- ⇒ RENDER
  - Set shader progams
  - Set shader attributes (if necessary)
  - Set vertex buffer
  - Set other buffers for rendering (normal, colors etc)
  - Render buffers

The process is a long one, but provides extremely low level access for extremely efficient and customizable graphics processes.

4. Libraries

To help save us time and energy, many JavaScript libraries exist which wrap WebGL and allow a higher level, more efficient workflow for web graphics.

Some libraries to look out for are: Three.js (link), PhiloGL (link), GLGE (link), J3D (link), among others.

5. Three.js

This tutorial will go into more detail and provide information using three.js.

This section uses **03_Three_StartingPoint**

5.1. Basic Setup

There are only a couple of objects that we need to setup in order to get three.js running. Don't worry about what they are all for right now, it will be explained later.

```
//TODO declare global variables
var renderer, scene, camera, objects;
```

We are going to actually define these variables only after the page loads. Three.js supports a few types of rendering including 2D methods. We are concerned with WebGL so we are going to create an instance of THREE.WebGLRenderer. **NOTE** that the three.js library is accessed through the keyword **THREE** (in all caps).

```
//TODO initialize three.js renderer
renderer = new THREE.WebGLRenderer();
```

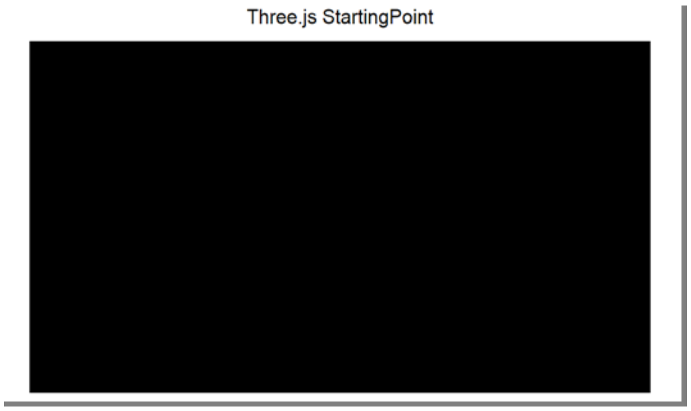
The renderer creates it's own canvas element, but doesn't add it to the html page. All we need to do is attach the created canvas to our DOM tree under the wrapper div that we have in the html already.

```
//attach the renderers output canvas to
//the DOM tree under our wrapper div
var canvasWrapper = document.getElementById('canvas-wrapper');
canvasWrapper.appendChild( renderer.domElement );
```

The last little thing to do here is set the size of the renderer to match the wrapper.

```
//set the size to match the wrapper
renderer.setSize( 960, 540 );
```

Now, if we open the webpage our wrapper should appear black because of the new canvas. To ensure that it's working, check the debug console (F12). There should be a message that reads 'THREE.WebGLRenderer 66'. (here 66 is simply the revision number of the three.js library that we are using).



5.2. The Scene and Camera

The other important object that we need for rendering is a three.js scene. The scene organizes all of the objects and lights that we want to render together. Creating it just takes one function call.

```
//TODO setup three.js scene
scene = new THREE.Scene();
```

The last item that's necessary to render our scene is a camera to view it through. To create the camera, we need to define 4 parameters: the field of view, aspect ratio, near clip plane and far clip plane.

```
camera = new THREE.PerspectiveCamera(75,        //horizontal field of view angle
                                     960 / 540, //aspect ratio
                                     0.1,       //near clip plane
                                     1000);     //far clip plane
```

Three.js can also create an orthographic camera using THREE.OrthographicCamera, but we won't discuss that here.

Now we are ready to render the scene! It's as easy as adding a render call to our game loop.

```
// render code here //
renderer.render(scene, camera);
```

As you can see, one of the great things about three.js is that we can create multiple scenes and cameras and switch between them very easily. Open index.html to make sure everything is working alright.

5.3. Adding Geometry

Now this is all great and working but it’s still just a black screen, so let’s add something to our scene.

In three.js rendered objects are called **Mesh Objects**. In order to create a mesh object we require **Geometry** and a **Material.**

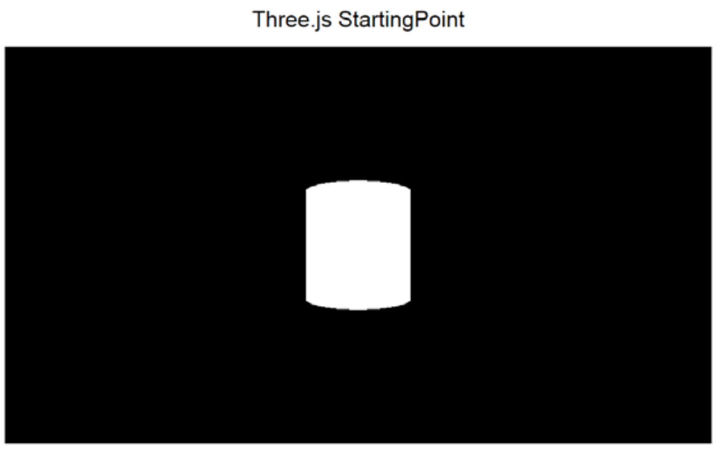
For geometry, there are many built in primitives that Three.js can create for us. Let’s make a cylinder.

```
//create a cylinder
//parameters are:
//top radius, bottom radius, height, radius segments, height segments, open ended
var cylinderGeometry = new THREE.CylinderGeometry(1.0, 1.0, 2.0, 20, 1, false);
```

For the material, three.js also has a whole set of default materials/shaders that we can use.

```
//now we make a material
var basicMaterial = new THREE.MeshBasicMaterial();
```

Now if we open the html page you can see a cylinder! Notice that MeshBasicMaterial has no shading on it, but a lambert material wouldn’t have shown up without any lights in our scene. So let’s add a light!



5.4. Adding a Light

Three.js supports point lights, directional lights and spot lights. For simplicity, let's make a directional light to represent a sun. The light will always shine towards 0, 0, 0 so we move it up and over to get a nice shadow.

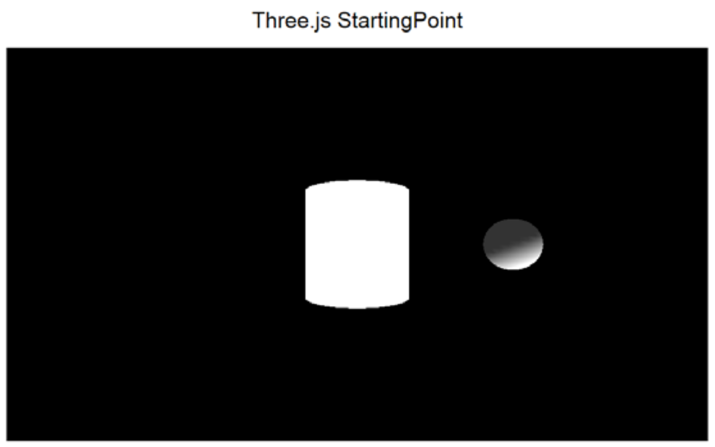
```
//create a light and add it to the scene
//parameters: light color, light intensity
objects.directionalLight = new THREE.DirectionalLight( 0xFFFFFF, 1.0 );
//move it to get an angle
objects.directionalLight.position.set(2, -5, 1);
scene.add( objects.directionalLight );
```

In order to see the shading, we will need to add an object that uses a lambert material, so let's make a nice sphere. We'll move it off to the side of the

```
//create a sphere lambert material for shading
//params: radius, width segments, height segments
var sphereGeometry = new THREE.SphereGeometry(0.5, 20, 20);
var lambertMaterial = new THREE.MeshLambertMaterial();
objects.sphere = new THREE.Mesh( sphereGeometry, lambertMaterial );
objects.sphere.position.set(3, 0, -5);
scene.add( objects.sphere );
```

This creates a shaded sphere but the shadow area is completely black. An ambient light will help with that.

```
//adding an ambient light will help with the harsh shadows
objects.ambientLight = new THREE.AmbientLight( 0x333333 );
scene.add( objects.ambientLight );
```



5.5. Models and Textures

Another key aspect of three.js that is loading external models and apply textures. It's when you get into this kind of workflow that you start to appreciate the simplicity of three.js.

Step one is to load the texture, so that it is available for the models material. The THREE.ImageUtils allows us to load images as textures really easily. We can pass it the image URL and it will return a **Texture** object for us to use in a material. We then create another lambert material, and define our texture as the **map.**

```
//let's load the external model
//we load the texture first
var cubeTexture = THREE.ImageUtils.loadTexture( "assets/box_color.png" );
//we create a lambert material and add the texture as a map
var cubeMaterial = new THREE.MeshLambertMaterial( {map : cubeTexture} );
```

Now that we have a material, we can load the model. I have included one that I exported from Maya using the three.js exporter. The exporter creates JSON objects that play well with JavaScript and the web in general.

To load it in, we use an instance of the THREE.JSONLoader. The loader has a function called 'load' which takes the URL to our object, and then a callback function which gets called when the loading is done. It's inside the callback function that we are going to create the mesh, move it, and add it to the scene.

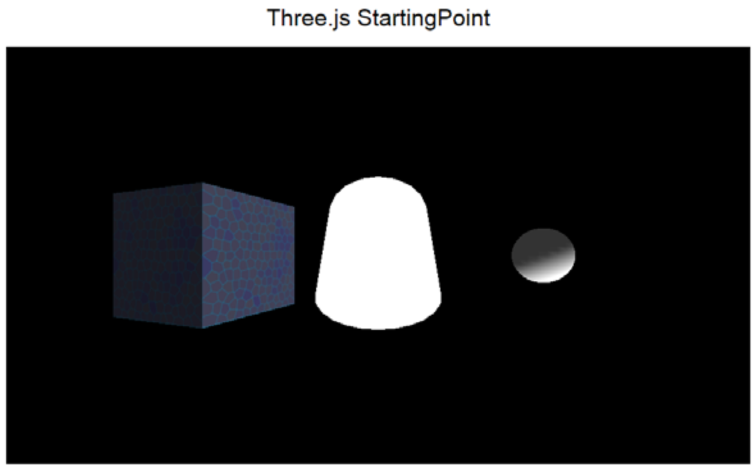
```
//then we load the cude geometry
//we create a json loader and then pass it the url to our json object
//we also define the callback function for when the geometry
//is done loading
var jsonLoader = new THREE.JSONLoader();
jsonLoader.load( "assets/cube.js", function( geometry ){
    objects.cube = new THREE.Mesh( geometry, cubeMaterial );
    objects.cube.position.set(-3, 0, -5);
    scene.add( objects.cube );
});
```

Just for fun, in our update function let's rotate our objects a little.

```
// update code here //

//rotate our objects around a little for fun
objects.cylinder.rotation.x += 0.02;
objects.sphere.rotation.y += 0.03;
//because this one loads async, we do a simple check
if(objects.cube)
    objects.cube.rotation.y += 0.01;
```

And there we have it, some wonderful rotating geometry with lighting and textures!



6. User Input

The last part of this tutorial will be using three.js to create a first person control scheme. Although many of the concept in this section aren't directly related to three.js or WebGL it is important to understand how they fit together to create a final product.

6.1. Keyboard Input

The first step in having a game is allowing user input. The keyboard is the simplest form of input so let's start there. In order to capture keyboard events, we add an event listener to the global *window* object.

```
//TODO initialize input events

//this function takes a string denoting the event type to listen to
//and a callback function for the event
window.addEventListener(“keydown”, onKeyDown);
```

This creates the listener, but we'll get errors right now because we haven't created the *onKeyDown* function. Let's add it to the bottom of our main.js file.

```
//event handler for keyboard input
//it needs to take in the event object passed by the browser
//the event object contains info about the key event
function onKeyDown(event)
{
}
```

The event object will allow us to check which key is pressed. The only issue is that Internet Explorer doesn't pass the event object. In order to get around this we add a simple check at the beginning of our function, and if event is undefined, we can get it from window.Event.

```
//get the event if this is internet explorer
if( !event ) event = window.Event;
```

The event object has a *keyCode* property that defines the key that was pressed. I usually include an object in my projects which allow me to select key values by name rather than number. I've included a file in this project called **keyCodes.js** which does exactly that, take a look if you want. Let's make a switch statement and define some simple camera movement through the arrow keys.

```
switch(event.keyCode)
{
    case KeyCodes.up: //forwards
        camera.position.z -= 1;
        break;
    case KeyCodes.down: //backwards
        camera.position.z += 1;
        break;
    case KeyCodes.left: //left
        camera.position.x -= 1;
        break;
    case KeyCodes.right: //right
        camera.position.x += 1;
        break;
}
```

You'll notice that these controls are jumpy and holding them down has a delay. To fix this we will create another event handler for the "keyup" event. We can then create some Boolean flags and keep track of which keys are down. Then, instead of moving the camera in the event handler, we add code to our update function which moves our camera a little bit every frame if the appropriate key is down.

First the variables to use for flags:

```
//for input
var keyUp = false,
    keyDown = false,
    keyLeft = false,
    keyRight = false;
```

Then we add the other event listener:

```
window.addEventListener("keyup", onKeyUp);
```

Change the old event handler and add a new one (the key down handler should look like this one except it will set the flags to true:

```
function onKeyUp(event)
{
    //get the event if this is internet explorer
    if(!event) event = window.Event;

    switch(event.keyCode)
    {
        case KeyCodes.up: //forwards
            keyUp = false;
            break;
        case KeyCodes.down: //backwards
            keyDown = false;
            break;
        case KeyCodes.left: //left
            keyLeft = false;
            break;
       case KeyCodes.right: //right
            keyRight = false;
            break;
    }
}
```

Last, we add the movement to our update function:

```
function onKeyUp(event)
{
    //get the event if this is internet explorer
    if(!event) event = window.Event;

    switch(event.keyCode)
    {
        case KeyCodes.up: //forwards
            keyUp = false;
            break;
        case KeyCodes.down: //backwards
            keyDown = false;
            break;
        case KeyCodes.left: //left
            keyLeft = false;
            break;
       case KeyCodes.right: //right
            keyRight = false;
            break;
    }
}
```

6.2. Mouse Movement

Adding first person controls to the camera is a little more complicated, but let's work through it. The first issue with the web browser is that the mouse can go out of the window, which interferes with using the mouse to look around. Thankfully, newer browsers allow us to lock the mouse pointer to the browser and continuously capture its movement. The first step will therefore be to write some code which enables the **pointer lock**. Chrome doesn't allow us to capture the mouse unless it's after some user input, so we'll add a *mouseodown* event handler with a simple listener that captures the mouse pointer. This can go with our other listeners in the *onPageLoaded* function:

```
//get pointer lock when the canvas is clicked on
var canvasWrapper = document.getElementById('canvas-wrapper');
canvasWrapper.addEventListener(“mousedown”, function(event){
    canvasWrapper.requestPointerLock();
});
```

When you click on our canvas now, you chrome will ask you to allow the pointer lock, but we still aren't responding to the mouse movement. We will need to add another listener for the *mousemove* event, and a hander to go with it.

```
window.addEventListener("mousemove", onMouseMove);
```

```
function onMouseMove(event)
{
    if(!event) event = window.Event;
}
```

Usually, the *mousemove* event returns the position of the mouse in the browser window, but with no pointer to track, this doesn't work after the pointer lock is activated. Instead, we look for **movementX** and **movementY** properties on the event variable. When the pointer lock is activated, the document keeps track of which element called the lock. We can use this variable to see if the pointer is currently locked.

```
//we only want to continue if the pointer lock is activated
if( null == document.pointerLockElement) == 'undefined') return;
```

Instead of mapping the **movementX** and **movementY** into the cameras rotation values, which can get messy, I have found that it's better to calculate a point around the player and have the camera look at that point (via the cameras built-in **lookAt** function).

We can boil the look direction down to:

- a point on a 2D circle which surrounds the camera on the X, Z plane
- a Y value for the height

I like this model because it lends well to the x and y movement of the mouse. The x movement can be used to adjust an angle value which defines the position of the look direction around the 2D circle, and the y movement can be used to adjust the up and down position. If this seems confusing, try following along with the code. First we declare a *lookVector* variable at the top of the main.js file, and then define it in our *onPageloaded* function. Three.js has a whole library of functions for vector math so we'll make a variable of type THREE.Vector3. We are then going to add our own *angle* value to it.

```
//to be used for cameras look direction
var lookVector;
```

```
//set the look vector as a new THREE.Vector3
lookVector = new THREE.Vector3();
lookVector.angle = 0;
```

Now comes the math. In our *onMouseMove* function, we use the movementX value to adjust the angle of our look vector. We can then use sin and cos functions to extract the X and Z components of the new direction.

```
//adjust the angle of the look vector and
//calculate the x and z components
//angles are in radians, so we multiply by a
//small number so that it doesn't move to fast
lookVector.angle += event.movementX * 0.002;
lookVector.x = Math.cos(lookVector.angle);
lookVector.z = Math.sin(lookVector.angle);
```

From there, we need the y value of our vector, so we use the **movementY** value to do this.

```
//now we adjust the y value of the vector directly with
//the mouse y movement. Clamp the values so that you
//don't get stuck with a huge number
lookVector.y -=   event.movementY * 0.02;
if(lookVector.y > 3) lookVector.y = 3;
else if(lookVector.y < -3) lookVector.y = -3;
```

We now have to look vector relative to the camera, but we'll need it in the world coordinate system for the *lookAt* function, so let's just add it to the cameras position. Then we can call *camera.lookAt*.

```
//add this to the camera position to get
//the global position
var pointAt = new THREE.Vector3();
//this function sets pointAt to the sum of the given vectors
pointAt.addVectors(camera.position, lookVector);

//point the camera at this spot
camera.lookAt(pointAt);
```

Run the browser and try moving the mouse around. You may find that your shapes disappear at first, but it's because starting our look vector at an angle of 0 means it's pointed away.

6.3. Fixing the movement

The last step in creating first person controls is to fix the movement we created with keyboard events. Now that the camera is rotating, left doesn't just mean move down the x-axis. To fix this, we can reuse our look vector, and extract the x and z components like we did in the last step, but use them for movement instead. Here's the new movement logic in the update function.

```
//move the camera with the arrow keys
var moveX = 0,
    moveZ = 0;
if(keyUp) {
    //forward towards look vector
    moveX += Math.cos(lookVector.angle);
    moveZ += Math.sin(lookVector.angle);
}
if(keyDown) {
    //the opposite of look vector
    moveX += Math.cos(lookVector.angle + Math.PI);
    moveZ += Math.sin(lookVector.angle + Math.PI);
}
if(keyLeft) {
    //90 degrees counter-clockwise of look vector
    moveX += Math.cos(lookVector.angle - 0.5 * Math.PI);
    moveZ += Math.sin(lookVector.angle - 0.5 * Math.PI);
}
if(keyRight) {
    //90 degrees clockwise of look vector
    moveX += Math.cos(lookVector.angle + 0.5 * Math.PI);
    moveZ += Math.sin(lookVector.angle + 0.5 * Math.PI);
}
//multiply by a small number to slow it down
camera.position.x += moveX * 0.1;
camera.position.z += moveZ * 0.1;
```

7. The Game

By adding a simple collider script to the project, and spicing up the update code that we have, you can easily turn this scene into a game. Take a look at the game example in folder **05_Three_GameExample**. This game is not very fancy but shows these elements can easily come together to create useful project.

8. Further Reading

That's all folks, but if you want to keep going with this, look into the three.js documentation. For a starter, look up how to change the color of you MeshBasicMaterial to something other than white. This is a good introduction to material properties/parameters as well as customized graphics. Three.js even supports custom shaders and post-processing… the power of WebGL is there, you just need to keep learning how to use it! ☺

Any questions or comments are welcome!
ryanbottriell@live.ca