Spring 5-2013

Water Simulation on WebGL and Three.js

Kerim J. Pereira
University of Southern Mississippi

The University of Southern Mississippi

Water Simulation on WebGL and Three.js

by

Kerim Pereira

A Thesis
Submitted to the Honors College
of The University of Southern Mississippi
in Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science
in the Department of Computer Science

March 2013

Beddhu Murali, Associate Professor, Advisor
Department of Computer Science

Chaoyang (Joe) Zhang, Chair,
Department of Computer Science

David R. Davies, Dean
Honors College

Abstract

Technology is constantly moving forward. Computers are getting better and better every single day. Processors, memory, hard drives, and video cards are getting more powerful and more accessible to the users. With all of this hardware progressing, it is also logical that users want software to evolve as fast as the hardware.

Since this new hardware is available to the user, the easiest way to make graphics even more accessible to everyone is through the web browser. This move to the browser simplifies the life of the end user so that he does not have to install any additional software. Web browser graphics is a field that has been growing since the launch of social media that allowed its users to play 2D games for free.

As a result of the increase in demand for better graphics on the web, major internet browser companies have decided to start implementing WebGL. WebGL is just a 3D drawing context that can be implemented on the web after the addition of the "<canvas>" tag within HTML5.

This thesis will create a water simulation using WebGL and Three.js. The simulation will depend on a graphics component called a shader that produces moving water and implements a water texture. The final simulation should look realistic and be performance-oriented so it can be used in large scenes.

Table of Contents

Table of Figures

Chapter 1 - Introduction

Section 1.1: Advances in Computer Graphics

Technology is constantly moving forward. Computers are getting better and better every single day. Processors, memory, hard drives, and video cards are getting more powerful and more accessible to the users. With all this hardware progressing, it is also logical that users want software to evolve as fast as the hardware.

Computer graphics is a field that is progressing at a rapid rate. Users always want bigger and better graphics that look more and more realistic. Also, they expect their experience with these graphics to be as smooth as possible. In order for users to experience smooth renderings, the developer must either create software that adapts to the capabilities of widely varying computers, smart phones, tablets, and game consoles.

Section 1.2: Moving Graphics to the Browser

With all of this new hardware available to the user, the easiest way to make graphics more accessible to everyone is through the web browsers. This move to the browser simplifies the life of the end user that does not have to install any additional software. Web browser graphics is a field that has been growing since the launch of social media that allowed its users to play 2D games for free.

As a result of the increase in demand for better graphics on the web, major internet browser companies have decided to start implementing WebGL. WebGL is just a 3D drawing context that can be implemented on the web after the addition of the "<canvas>" tag within HTML5. With this new technology we are now able to access the

video card directly without having to install a third party plug-in. This allows programmers to create simulations or video games that can utilize all the power from the video card and its processing unit. The following example is a small pool simulation that was created with WebGL by Evan Wallace (2011).

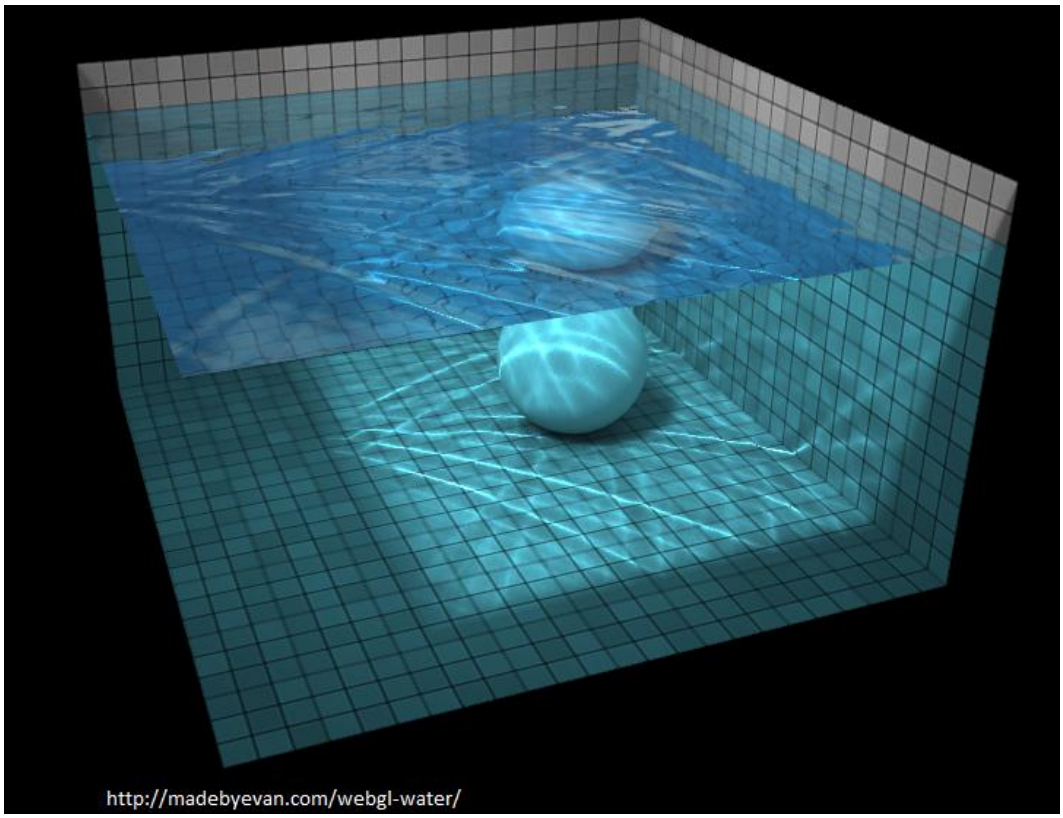

Figure 1.1 - WebGL Water Example (Wallace 2011)

Section 1.3: Challenges of Internet Browsers

Although browser companies are improving at a fast pace, there are still several blocks on the way. A programmer cannot just free their mind and render any scenes that they want. The developer needs to take into consideration the limitations of the browsers in comparison to the actual hardware. The browser has a specific amount of cache memory that the user cannot exceed, or after a certain amount of memory RAM used the application will crash.

Another limitation is that not every web browser acts in the same way. The manufacturers can decide what they want to implement and what they do not want to implement. For example, WebGL has been adopted by the majority of browsers, but some, like Internet Explorer, still refuse to implement WebGL as a standard.

Section 1.4: Research Statement

This thesis will create a water simulation using WebGL and Three.js. The simulation will depend on a shader that renders moving water and implements a water texture. The final simulation should look realistic and be performance-oriented so it can be used in large scenes.

Chapter 2 - Literature Review

Section 2.1: Rendering Pipeline

According to Akenine-Moller et al (2008) The rendering pipeline is the core component of computer graphics. The function of the pipeline is to render a two-dimensional image when given a virtual camera, three-dimensional objects, light sources, shading equations, and more.

The pipeline has been evolving over many years. In the beginning, the hardware rendering pipeline was fixed; this means that the developers did not have any means to make their code interact with the pipeline. With the evolution of software, the manufacturers of the hardware decided to make their pipeline more programmable. Programmable hardware is important because it lets developers implement algorithms and coding methods that have a significant effect on the performance of the software.

There are various steps in the rendering pipeline and they can be represented in various ways, so in order to represent it better we will divide the rendering pipeline into three conceptual stages: the application stage, the geometry stage, and the rasterizer stage. These conceptual stages are pipelines by themselves, which mean that inside of each stage there could be other sub-stages.

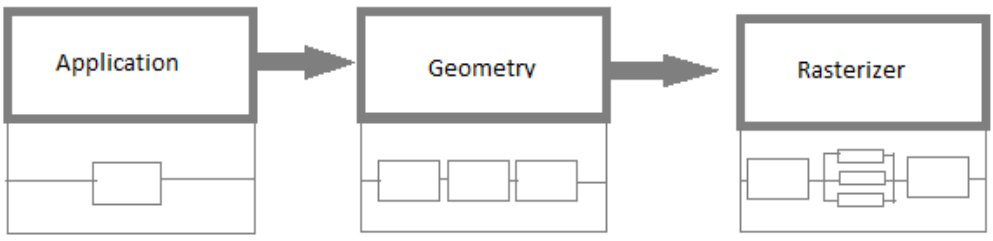

Figure 2.1 - Abstraction of the Rendering Pipeline

Figure 2.1 is a good illustration of what the conceptual stages are and how inside each one of them there is an internal pipeline, like in the geometry stage, or it could be parallelized, like the rasterizer stage.

Section 2.1.1: The Application Stage

The application stage is where it all begins. Here, the developer has full control over what happens since everything is being executed on the CPU.

This is the stage where the developer allows the user to provide all the inputs to the program and based on this input generates the geometry to be rendered, material properties and virtual cameras. This data is then passed on to the geometry stage. Accepting rendering primitives like points, triangles, lines and transferring it to the next stage is the purpose of this stage.

Section 2.1.2: The Geometry Stage

The geometry stage, as discussed by Akenine-Moller et al (2008), is responsible for the majority of the per-vertex operations. This stage is divided into the following functional stages: model and view transform, vertex shading, projection, clipping, and screen mapping. These new stages are shown in Figure 2.2

Figure 2.2 – The Geometry Stage (Akenine-Moller et al., 2008)

Section 2.1.3: Model and View Transform

During the model and view transform step, the input that is coming from the application stage has to be transformed into the supported coordinate system. This process grabs the vertices and normals of an object and locates them in the world space. The world space is a unique entity, so after this process every object is in the same world space. Only models that can be seen by the virtual camera that is also set during the application stage can be rendered to the screen. The camera is set in the world space with a position and a direction, where the camera is going to aim.

Models and camera use the view transform to relocate the objects and the camera. The purpose of the view transform is to reposition the camera to the center of the coordinate system. An example of this reorganization can be seen in Figure 2.3

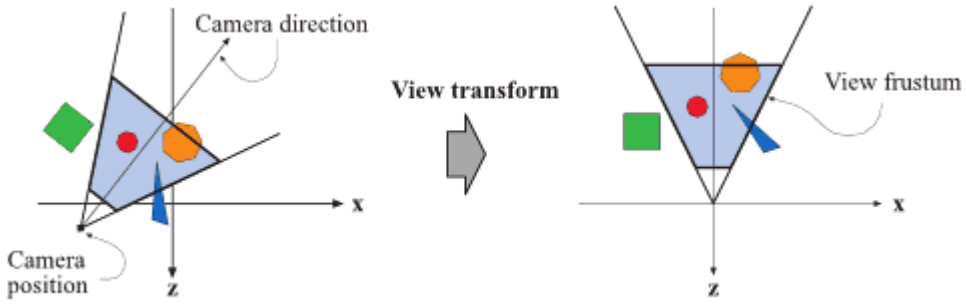

Figure 2.3 - View Transform Reorganization (Akenine-Moller et al., 2008)

Section 2.1.4: Vertex Shading

The vertex shading stage is one of the most programmable parts of the rendering pipeline. In order to create realistic graphics, the developer needs to give an appearance to the objects. During this phase, the rendering pipeline will be focused on the vertices of the objects. Each vertex in an object stores different data like: position, normal, color, vectors, texture coordinates, etc.

Vertex shaders can be used to do traditional vertex-based operations such as transforming the position by a matrix, computing light equations to generate color, and generating texture coordinates. The output of vertex shaders, stored in the "varying" variables, are passed to the rasterizer stage to be used as input for the fragment shaders.

Section 2.1.5: Projection

The next stage in the rendering pipeline is the projection. The main point of the projection stage is to convert 3D models into 2D projections with respect to the camera. Here, the view volume is transformed into the canonical view volume, which is a unit cube with extreme points (-1,-1,-1) and (1,1,1). Figure 2.4 shows an example of a unit cube. There are two main types of projections: orthographic and perspective.

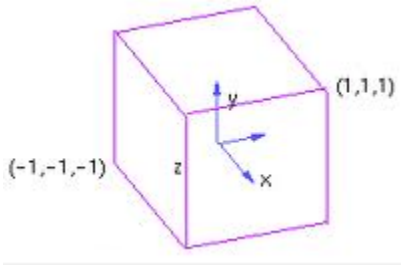

Figure 2.4 - Unit Cube

According to Akenine-Moller et al (2008) the orthographic projection is just a form of parallel projection, which means that that all projection lines are orthogonal to the projection plane. The orthographic projection is just a combination of object translations and scaling.

In a perspective projection, just like the name implies, it depends on the perspective. Objects that are further away from the camera are rendered smaller, and this gives the feeling of being in a three-dimensional space.

Section 2.1.6: Clipping

During the clipping stage, only projections that are in the view frustum are passed to the next stage. Objects that have part of them in the view and part outside the view will be clipped and only the parts that are inside of the unit cube will remain.

The clipping stage is not programmable for the developer and is limited to how the hardware manufactures implemented this stage.

Section 2.1.7: Screen Mapping

During the screen mapping stage, the models that were not clipped out are converted into screen coordinates. A screen coordinate is the position of the object within the drawing context. Since the drawing context can only be represented in 2D, the screen coordinates are also represented in 2D with an x-coordinate and a y-coordinate. The new screen coordinate will be passed to the rasterizer stage along with the z-coordinate that indicates depth.

Section 2.1.8: The Rasterizer Stage

After an object passes though the geometry stage, the next stop is the rasterizer stage. The purpose of this stage is to compute and set the color for each pixel that is part of the object, this process is called scan conversion or rasterization. Just like the previous stages, the rasterizer stage can be divided into the following functional stages: triangle setup, triangle traversal, pixel shading, and merging. A graphical representation of the rasterizer stage is shown in Figure 2.5

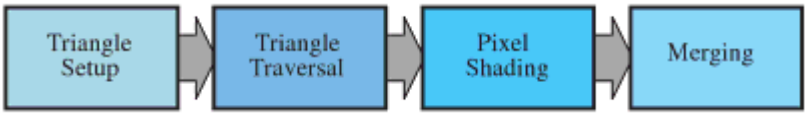

Figure 2.5 – The Rasterizer Stage (Akenine-Moller et al., 2008)

Section 2.1.9: Triangle Setup

The first step in the rasterizer stage is the triangle setup. The triangle setup is a fixed operation, which its main purpose is to compute the triangle's surface.

Section 2.1.10: Triangle Traversal

Triangle traversal is the process where a line is drawn on a raster screen between two points; this process is also called scan conversion. With this process, each property of a fragment is set. One example of these properties is the depth.

Section 2.1.11: Pixel Shading

Just like the vertex shading stage, pixel shading, also called fragment shading, is a highly programmable step during the rendering pipeline. At this point the developer can change the appearance of an object one pixel at a time. The pixel shader is extremely important because here is where texturing is applied to the objects.

Section 2.1.12: Merging

The merging stage is the last step of the rendering pipeline. This stage depends on several buffers to store information about the individual pixels. The common buffers that

we can find during the merging stage are: the color buffer, the depth buffer, and the frame buffer.

The color buffer is where the color information that comes from the fragment shader is stored. The color buffer also contains an alpha channel, which is used to store the opacity value for each pixel.

The depth buffer, also called z-buffer, is where the visibility issue is solved. The depth buffer will draw only objects that are closer to the camera; this is done with a simple comparison of the z value, with the pixel with the greater z value being drawn.

The frame buffer is just a name for all the frames on the system. In addition to the color and depth buffer, there may be an accumulation buffer. The accumulation buffer is used to create other effects like depth of field, motion blue, antialiasing, etc.

Section 2.2: OpenGL ES

OpenGL ES is an application programming interface (API) for drawing advanced 3D graphics that is targeted at embedded devices like video game consoles, cell phones, tablets, handheld devices, etc. OpenGL ES was created by the Khronos group, which is a non-profit member-funded industry consortium that focuses on the creation of open standard, royalty free, APIs for a wide variety of devices.

According to Munshi et al (2008) OpenGL ES is subset of well-defined profiles of OpenGL. It provides a low level API between applications and hardware or software graphics engines. This API addresses some of the problems with embedded devices like: very limited processing capabilities, sensitivity to power consumption, low memory bandwidth, and lack of floating-point hardware.

Section 2.3: WebGL

As Parisi (2012) describes, WebGL is a cross-platform open-source API that is used to create 3D graphics in a Web Browser. WebGL is based on OpenGL ES 2.0, it is OpenGL ES 2.0 on JavaScript, and so it uses OpenGL's shading language often abbreviated as GLSL.

WebGL runs in the HTML5 <canvas> element and has full integration with all the Document Object Model (DOM) interfaces. WebGL is a cross-platform API since it can run in many different devices and is for the creation of dynamic web applications. (Parisi, 2012)

Just like OpenGL, WebGL is a low-level language that requires a lot of code in order to have something created and working. You have to load, compile, and link the shaders, set up the variables that are going to be passed to the shaders, and perform complex matrix math to animate objects. At the same time, it gives freedom to the programmer to experiment and develop complex graphics.

Now every major browser supports WebGL, except Internet Explorer, since HTML5 is the new standard in the industry. Firefox 4 or higher, any Google Chrome because it updates itself (version 10 or higher), and Safari OS X 10.7 are some of the examples of current browsers that support this technology. Also, now browsers on tablets and smartphones are starting to implement WebGL as a standard.

Section 2.4: Three.js

As said in the last section, WebGL is a low-level API that even to draw a simple triangle to the screen takes a lot of work. Three.js is a framework that simplifies the

creation of 3D scenes. With this framework, it is not necessary to explicitly compile and apply shaders with WebGL commands. All of that is done for you in the background. Three.js is an open-source JavaScript library that can be downloaded by anyone from https://github.com/mrdoob/three.js/

As discussed by Danchilla (2012), Three.js provides several different draw modes and can fall back to the 2D rendering context if WebGL is not supported. On this project, Three.js is the library that is going to be used to create the scene and the water simulation.

Section 2.5: Shaders

As mentioned in the rendering pipeline section, shading is the operation of determining the effect of light on a material. Shaders are responsible for the materials and appearance of an object in a scene.

Shaders are so important than they even have their own programming language. This programming language is called OpenGL Shading Language (GLSL). GLSL is a C-like language that has its own types that will be showed in the tables below.

GLSL types	Description
vec2, vec3, vec4, ivec2, ivec3, vec4, bvec2, bvec3, bvec4	Vector of size 1×2, 1x3, or 1x4; and of type float, integer, or bool, respectively
mat2, mat3, mat4	Floating point matrix of size 2x2, 3x3, or 4x4
sampler2D, samplerCube	Handles to 2D or cube mapped textures

Figure 2.6 – GLSL Types and Descriptions (Danchilla, 2012).

Qualifier	Description
const	Constant throughout the program. Read only.
uniform	Constant value across an entire primitive.
attribute	VS per vertex information from our WebGL application.
varying	VS write, FS read.

Figure 2.7 – GLSL Qualifiers (Danchilla, 2012).

Figure 2.7 shows the GLSL qualifiers. These qualifiers are used in different ways throughout the shaders. Attributes and uniforms are input values for the shaders and varying types are going to be output values because they can change during the lifetime of the shader program. Varying variables are changed throughout the vertex shader and are the output. After the vertex shader outputs the varying varible the fragment shader uses it as an input.

Variable	Type	Description	Used In	Input/Output
gl_Position	vec4	Vertex position	VS	output
gl_PointSize	float	Point size	VS	output
gl_FragCoord	vec4	Fragment position within the frame buffer	FS	input
gl_FrontFacing	bool	Whether the fragment is part of a front or back facing primitive	FS	input
gl_PointCoord	vec2	Fragment position within a point	FS	input
gl_FragColor	vec4	Final fragment color	FS	output
gl_FragData[n]	vec4	Fragment color for a color attachment, n	FS	output

Figure 2.8 – GLSL Built-in Variables (Danchilla, 2012).

Figure 2.8 shows the built-in variables that GLSL gives the developer. They are already implemented and they can be used without any prior declaration.

There is plenty more built-in functionality that will not be discussed in this chapter. GLSL has plenty of built-in functions like mathematical functions, texture functions, geometric functions, etc. more information can be found about these

functionalities in the following link from the OpenGL website:

http://www.opengl.org/sdk/docs/manglsl/

As mentioned before in Chapter 2, there are two types of shaders: vertex and fragment. Figure 2.9 will show an example of a simple vertex and fragment shader.

```
1  <script id="shader-vs" type="x-shader/x-vertex">
2      attribute vec3 aVertexPosition;
3
4      void main(void) {
5          gl_Position = aVertexPosition;
6      }
7  </script>
8
9  <script id="shader-fs" type="x-shader/x-fragment">
10     void main(void) {
11         gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
12     }
13 </script>
```

Figure 2.9 – Example of Vertex and Fragment Shader

In the above example, we have a very simple vertex and fragment shader within the "<script>" tag. This is one of the ways that we can use shaders in WebGL and Three.js. In line 1, we declare our script tag with an id of "shader-vs" and a type that says that is a vertex shader. Line 4-5 is the main body of any shader since it is required to have a main function and in this case we just assign our vertex position to the GL variable "gl_Position". The fragment shader starts in line 9 with the same declaration of the script tag, but this time we make the distinction of a fragment shader, then we declare the main function and assign a vector of four dimensions with the value of red and store it in the GL variable "gl_FragColor".

Chapter 3 - Methodology

Section 3.1: Initializing Three.js

In order to initialize Three.js, first it is necessary to get a copy of the source Three.js code. The source code can be downloaded from https://github.com/mrdoob/three.js/ . Next, we want to create the folder for our project and create the index.html. Figure 3.1 shows the code necessary to create the html file.

```
1  <!DOCTYPE_HTML>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width,
6          user-scalable=no, minimum-scale=1.0, maximum-scale=1.0">
7          <style>
8              body {
9                  font-family: Monospace;
10                 background-color: #fff;
11                 margin: 0px;
12             }
13         </style>
14         <script src="../three.js/build/three.js"></script>
15         <script src="js/Detector.js"></script>
16         <script src="js/object.js"></script>
17         <script src="js/terrain.js"></script>
18         <script src="js/TrackballControls.js"></script>
19         <script src="js/stats.js"></script>
20         <script src="js/app.js"></script>
21     </head>
22     <script>
23         if (!Detector.webgl)
24         Detector.addGetWebGLMessage();
25     </script>
26
27
28     <body onload="init()">
29     </body>
30 </html>
```

Figure 3.1 – index.html

Figure 3.1 is a simple HTML page with basic Cascading Style Sheets (CSS) code. Next, it is necessary to load every library that is needed to create the scene. At this point the most important one is in line 14, where Three.js is actually loaded and now can be used in our webpage. During initializing, it is a good idea to check if the browser uses

WebGL, which is done in lines 22-25. Finally, in line 28 there is a call to the function "init()" that comes from the js/app.js file. In this function a WebGL Renderer will be created whose purpose is to get the drawing context and then render the scene using WebGL. Figure 3.2 shows how the renderer is set up using init().

```
21  function init()
22  {
23      setupRenderer();
24      setupScene();
25      container = document.createElement( 'div' );
26      document.body.appendChild( container );
27      container.appendChild( renderer.domElement );
28      setupStats();
29      setupLoader();
30      setupCamera();
31      setupControls();
32      setupLights();
33      setupSkybox();
34      setupTerrain(1024, 1024, 32, 32, "images/dirt2.jpg");
35      renderer.autoClear = false;
36      animate();
37  }
38
39  function setupRenderer()
40  {
41      renderer = new THREE.WebGLRenderer();
42      renderer.setSize(CANVAS_WIDTH, CANVAS_HEIGHT);
43      renderer.setClearColor( scene.fog.color );
44      document.body.appendChild(renderer.domElement);
45  }
46
47  function animate()
48  {
49      requestAnimationFrame( animate );
50      render();
51      controls.update();
52  }
53
54  function render()
55  {
56      stats.update();
57      uniforms.time.value += .001;
58      uniforms.time2.value += 0.004;
59      uniforms.eyePos.value = camera.position;
60
61      camera.lookAt(scene.position);
62      renderer.render(skyboxScene, skyboxCamera);
63      renderer.render(scene, camera);
64  }
```

Figure 3.2 – Setting up the WebGL Renderer

Figure 3.2 shows how the WebGL Renderer is being created for the project. Everything starts from the init() function in app.js. From here, setupRenderer() is called which creates a new renderer with the size of the canvas and the renderer is attached to the body tag of the html file using JavaScript. At this point the renderer is set but it has to be refreshed several times a second, and that is the functionality of the render() function.

Section 3.2: Creating a Scene

Now it is time to create the scene where the water simulation is going to take place. The first thing that is going to be implemented in the scene is the camera and a simple object (a sphere). Figure 3.3 will show how to set the camera, controls, lights, add a sphere, and add a skybox.

```
34  function setupSkybox()
35  {
36      skyboxCamera = new THREE.PerspectiveCamera( 50, window.innerWidth / window.innerHeight,
37                                                  1, 100000 );
38      skyboxScene = new THREE.Scene();
39      texturesUrl = "images/skyboxes/moon/";
40      urls = [ texturesUrl + "px.jpg", texturesUrl + "nx.jpg", texturesUrl + "py.jpg",
41              texturesUrl + "ny.jpg", texturesUrl + "pz.jpg", texturesUrl + "nz.jpg" ];
42
43      var textureCube = THREE.ImageUtils.loadTextureCube( urls, new THREE.CubeRefractionMapping() );
44
45      var shader = THREE.ShaderLib[ "cube" ];
46      shader.uniforms[ "tCube" ].value = textureCube;
47
48      var material = new THREE.ShaderMaterial( {
49
50                                  fragmentShader: shader.fragmentShader,
51                                  vertexShader: shader.vertexShader,
52                                  uniforms: shader.uniforms,
53                                  side: THREE.BackSide
54
55                              } );
56
57      mesh = new THREE.Mesh( new THREE.CubeGeometry( 100000, 100000, 100000 ), material );
58      skyboxScene.add( mesh );
59  }
60
61  function setupScene()
62  {
68      scene = new THREE.Scene();
69      scene.fog = new THREE.Fog( 0xcce0ff, 500, 10000 );
70      var ballGeo = new THREE.SphereGeometry( 5, 20, 20 );
71      var ballMaterial = new THREE.MeshPhongMaterial( { color: 0xffffff } );
72      sphere = new THREE.Mesh( ballGeo, ballMaterial );
73      scene.add( sphere );
74  }
75  function setupLights()
76  {
77      var light = new THREE.DirectionalLight( 0xdfebff, 1.75 );
78      light.position.set( 50, 200, 100 );
79      light.position.multiplyScalar( 1.3 );
80      scene.add(light);
83  }
```

Figure 3.3 – Setting up the Scene

Figure 3.3 shows the functions setupSkyBox(), setupScene() and setupLights(). All of these functions are being called in init(). The skybox is a method of creating background to make a scene look bigger. This is created by adding a cube to the scene and adding a texture to the inside of the cube. In order to create a skybox in Three.js, it is necessary to create a camera for the skybox. This is implemented this way because the skybox needs its own renderer. In line 36, the skybox camera is created as a perspective camera, where PerspectiveCamera is a type of object already implemented in Three.js. In line 37, the new scene for the skybox is created. Next, several images are loaded. These images are going to be attached to the inside part of the cube. After loading the texture, it is time to use shaders to make the skybox look like a background, so it is convenient that Three.js already has those shaders created for us (for the skybox and other simple shaders). Then the material is created and the geometry is created. The last step is to join them together as a mesh and add it to the scene.

The setupScene function simply creates a new scene with the new THREE.Scene() function, then some fog is added just for looks. Next, an object is added just for reference and shows a simple scene.

The setupLights function uses the built-in functionality of Three.js to create a directional light. In lines 78-80 the lights are set in the desired position and added to the scene. Figure 3.4 shows the result from the code in Figure 3.3

Figure 3.4 – Result from Figure 3.3

Section 3.3: Adding a Plane

In order to create the water simulation, a plane is needed. The plane is a simple object in Three.js that later will have the water shader applied to it. The code to add a plane is not large. Figure 3.5 shows how to create a plane in Three.js

```
123  function addPlane(max_row, max_col, segment_row, segment_col, texture)
124  {
125      var xm = new THREE.MeshPhongMaterial( { color: 0xffffff } );
126      var geometry = new THREE.PlaneGeometry( max_row, max_col, segment_row, segment_col );
127      var mesh = new THREE.Mesh( geometry, xm );
128      mesh.doublesided = true;
129      mesh.rotation.x = -1.570796;
130      scene.add(mesh);
131  }
```

Figure 3.5 – Code for Adding a Plane

The code in the Figure above is pretty much the same as that for creating the ball in the previous section. Just create the geometry, material, join them in the mesh, and add it to the scene.

Section 3.4: Creating the Vertex Shader

Now that the plane is set, it is time to actually start creating the simulation. Since what the project is looking for is to create a water simulation, it is necessary to know how water behaves. Water moves in waves; these waves move just like the sine function. Figure 3.6 shows the sine graph.

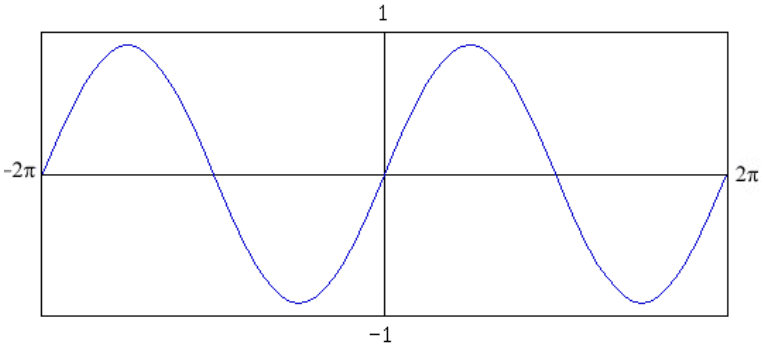

Figure 3.6 – Sine Graph

Knowing that waves behave in the same way as the sine or cosine function tells us that in order to make the waves move, the sine function will be needed. Since the vertex shader is the one that will give the effect of movement, the sine function will be applied here. Now, it is also important to make the shader as easy to configure as possible; the solution for this is to create uniforms that are easy to change the value. Figure 3.7 shows how easy uniforms are to configure and change the appearance of the shader.

```
uniforms.time = { type: "f", value: 0.1 };
uniforms.envMap = { type: "t", value: 1, texture: textureCube};
uniforms.texture2 = { type: "t", value: THREE.ImageUtils.loadTexture("images/water1.jpg") };
uniforms.texture2.value.wrapS = uniforms.texture2.value.wrapT = THREE.Repeat;
uniforms.texture2.value.repeat.set(25, 25);
uniforms.eyePos = { type: "v3", value: new THREE.Vector3(300, 50, 4) };
uniforms.waterHeight = { type: "f", value: 0.1 };
uniforms.amplitude = { type: "fv1", value: [0.5, 0.25, 0.17, 0.125, 0.1, 0.083, 0.714, 0.063] };
uniforms.wavelength = { type: "fv1", value: [25.133, 12.566, 8.378, 6.283, 5.027, 4.189, 3.590, 3.142] };
uniforms.speed = { type: "fv1", value: [1.2, 2.0, 2.8, 3.6, 4.4, 5.2, 6.0, 6.8] };
var i = 0;
var angle = [];
for(i = 0; i < 8; i++) {
    var a = Math.random() * (2.0942) + (-1.0471);
    angle[i] = new THREE.Vector2(Math.cos(a), Math.sin(a));
}
uniforms.direction = { type: "v2v", value: angle };
```

Figure 3.7 – Uniform Declaration

The next step is actually writing the vertex shader. The method that is going to be used is to implement the shader as a string inside the apps.js file. Figure 3.8 shows the fragment shader that is based from the work of Concord (2010). The shader starts with a call to the main() function and from there waveHeight() and waveNormal() are called. The waveHeight function returns a float value to create the z value for the position. The waveNormal function will return a vector3 with the normalized value of the each vertex, and stores it in the worldNormal variable.

```
194     "float wave(int i, float x, float y) {",
195         "float frequency = 2.0*pi/wavelength[i];",
196         "float phase = speed[i] * frequency;",
197         "float theta = dot(direction[i], vec2(x, y));",
198         "return amplitude[i] * sin(theta * frequency + time * phase);",
199     "}",
200         "float waveHeight(float x, float y) {",
201         "float height = 0.0;",
202         "for (int i = 0; i < numWaves; ++i)",
203         "height += 10.0*wave(i, x, y);",
204         "return height;",
205     "}",
206
207     "float dWavedx(int i, float x, float y) {",
208         "float frequency = 2.0*pi/wavelength[i];",
209         "float phase = speed[i] * frequency;",
210         "float theta = dot(direction[i], vec2(x, y));",
211         "float A = amplitude[i] * direction[i].x * frequency;",
212         "return A * cos(theta * frequency + time * phase);",
213     "}",
214
215     "float dWavedy(int i, float x, float y) {",
216         "float frequency = 2.0*pi/wavelength[i];",
217         "float phase = speed[i] * frequency;",
218         "float theta = dot(direction[i], vec2(x, y));",
219         "float A = amplitude[i] * direction[i].y * frequency;",
220         "return A * cos(theta * frequency + time * phase);",
221         "}",
222     "vec3 waveNormal(float x, float y) {",
223         "float dx = 0.0;",
224         "float dy = 0.0;",
225         "for (int i = 0; i < numWaves; ++i) {",
226         "dx += dWavedx(i, x, y);",
227         "dy += dWavedy(i, x, y);",
228         "}",
229         "vec3 n = vec3(-dx, -dy, 1.0);",
230         "return normalize(n);",
231     "}",
232
233     "void main() {",
234         "vUv = vec2( 3.0, 1.0 ) * uv;",
235         "vec4 pos = vec4(position, 1.0);",
236         "pos.z = waterHeight * waveHeight(pos.x, pos.y);",
237         "vPosition = pos.xyz / pos.w;",
238         "worldNormal = waveNormal(pos.x, pos.y);",
239         "eyeNormal = normalMatrix * worldNormal;",
240         "vec4 mvPosition = modelViewMatrix * pos;",
241         "gl_Position = projectionMatrix * mvPosition;",
242     "}"
243     ].join("\n");
```

Figure 3.8 – Vertex Shader

Section 3.5: Creating the Fragment Shader

After the vertex shader has calculated its output, it passes all needed values to the fragment shader. The fragment shader is where the texture is going to be applied. The main goal after having the shader moving like actual water is to make it look realistic. In order to make the water texture that has been loaded look better, the built-in function “noise” will be used to give a sense of randomness to the water. Then the color of the texture will be passed to gl_FragColor. Figure 3.9 shows the complete fragment shader.

```
245     var fragmentShaders = [
246     "varying vec2 vUv;",
247     "uniform sampler2D texture2;",
248     "uniform float time2;",
249
250     "void main() {",
251             "vec2 position = -1.0 + 2.0 * vUv;",
252             "vec4 noise = texture2D( texture2, vUv );",
253             "vec2 T = vUv + vec2( -2.5, 10.0 ) * time2 * 0.01;",
254
255             "T.x -= noise.y * 0.2;",
256             "T.y += noise.z * 0.2;",
257
258             "vec4 color = texture2D( texture2, T * 1.5 );",
259             "gl_FragColor = color;",
260             "}"
261     ].join("\n");
```

Figure 3.9 – Fragment Shader

Section 3.6: Applying the Shader to the Plane

After both vertex and fragment shader it is time to apply them to the material of the plane. Three.js makes this extremely simple. Figure 3.10 shows the four lines of code that are needed for the material and then just create the mesh with the new material as part of its properties, and add it to the scene.

```
264     var xm = new THREE.ShaderMaterial({ uniforms: uniforms,
265                 vertexShader: vertexShaders,
266                 fragmentShader: fragmentShaders
267     });
268
269     var geometry = new THREE.PlaneGeometry( max_row, max_col, segment_row, segment_col );
270     var mesh = new THREE.Mesh( geometry, xm );
271     mesh.doublesided = true;
272     mesh.rotation.x = -1.570796;
273
274     scene.add(mesh);
275 }
```

Figure 3.10 – Applying Shaders to the Material

Figure 3.11 shows the plane with the water shader applied to it.

Figure 3.11 – Plane with Shader

Section 3.7: Adding the Terrain

In order to prove that the shader can be used in large scenes, a terrain will be created. This terrain function is exported from a Three.js example that is available when Three.js is downloaded. Figure 3.11 shows the function that loads the terrain.

```
277 function setupTerrain(maxRow, maxColumn, segmentColumn, segmentRow, texture) {
278     data = Terrain.d;
279     camera.position.y = data[ worldHalfWidth + worldHalfDepth * worldWidth ] + 5;
280     var terrainGeometry = new THREE.PlaneGeometry( 1024, 1024, worldWidth - 1, worldDepth - 1 );
281     for ( var i = 0, l = terrainGeometry.vertices.length; i < l; i ++ ) {
282         terrainGeometry.vertices[ i ].z = data[ i ] * 5;
283     }
284     var xm = new THREE.MeshLambertMaterial( { map: THREE.ImageUtils.loadTexture( texture, THREE.UVMapping() )});
285     xm.map.wrapS = THREE.RepeatWrapping;
286     xm.map.wrapT = THREE.RepeatWrapping;
287     xm.map.needsUpdate = true;
288     xm.map.repeat.set(64,64);
289     terrainMesh = new THREE.Mesh( terrainGeometry, xm );
290     terrainMesh.rotation.x = -1.570796;
291     terrainMesh.position.y = - 355;
292     console.log(terrainMesh.position);
293     scene.add( terrainMesh );
294 }
```

Figure 3.12 – Adding the Terrain

Section 3.8: Adding Stats.js

The last step in the project is to add a way to record the frames per second (FPS) that it takes the scene the run. Three.js brings a file called stats.js that can be implemented on top of our canvas and will show the FPS. In order to make the stats.js work, it is necessary to create a new container and append it to the body of the html file. After that is done it is as simple as call the new Stats() function and position it wherever it is desired. Figure 3.11 shows the full implementation of stats.js .

```
41  function setupStats()
42  {
43      container = document.createElement('div');
44      document.body.appendChild(container);
45      container.appendChild(renderer.domElement);
46      stats = new Stats();
47      stats.domElement.style.position = 'absolute';
48      stats.domElement.style.top = '0px';
49      stats.domElement.style.zIndex = 100;
50      container.appendChild( stats.domElement );
51
52  }
```

Figure 3.13 – Setting up Stats.js

Chapter 4 - Results

Section 4.1: Final Scene

After all the programming and over three hundred lines of code, there is a final scene that can be viewed in Figure 4.1. This scene has a skybox, a sphere in the middle for reference, the terrain, and the water shader. Also, the scene is highly programmable and changing its parameters is very simple. The scene can be found at the following url: http://bigcat.cs.usm.edu/~kpereira/thesis/

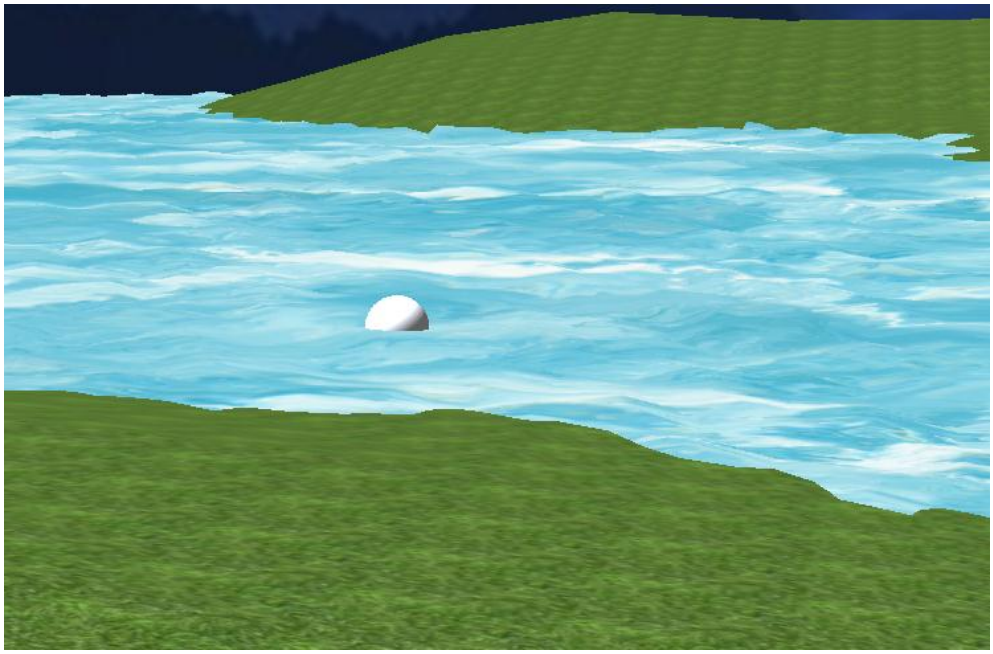

Figure 4.1 – Final Scene

Section 4.2: Performance

The performance of the final scene is very stable. The scene has been tested on 3 different computers with different graphics cards.

Framerate Performance:

- Nvidia GTX 580: 60 FPS
- Nvidia GTX 465: 60-55 FPS
- Intel HD Graphics 4000: 60-45 FPS

Figure 4.2 – GTX 465 Performance 60FPS

Memory Performance:

After testing the final scene with the task manager that Google Chrome has built-in, leaving the scene running for hours produced little to no change in memory. This means that the final scene does not have any memory leaks and it performs really well. In Figure 4.3, the amount of Desktop memory and GPU memory that the scene is using is shown.

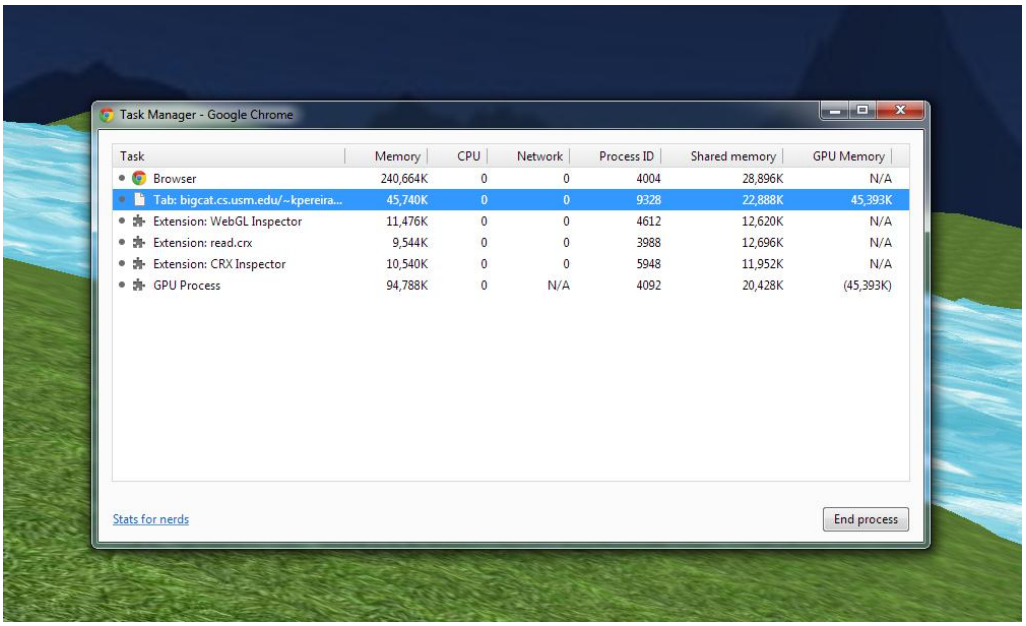

Figure 4.3 – Google Chrome's Task Manager

Section 4.3: Easy Configuration

The final scene is highly programmable and can change appearance with simple changes. The texture in the shader, skybox, and terrain can be changed in less than a minute. The way the shader behaves also can be changed with no effort. The following Figures will show different configurations for the scene.

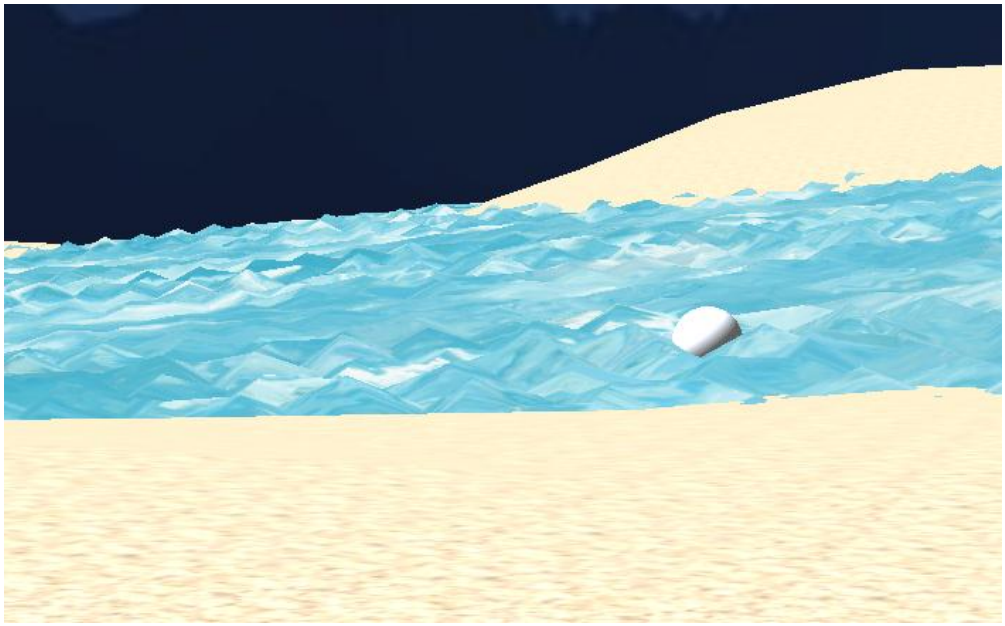

Figure 4.4 – Different Texture and Higher Waves

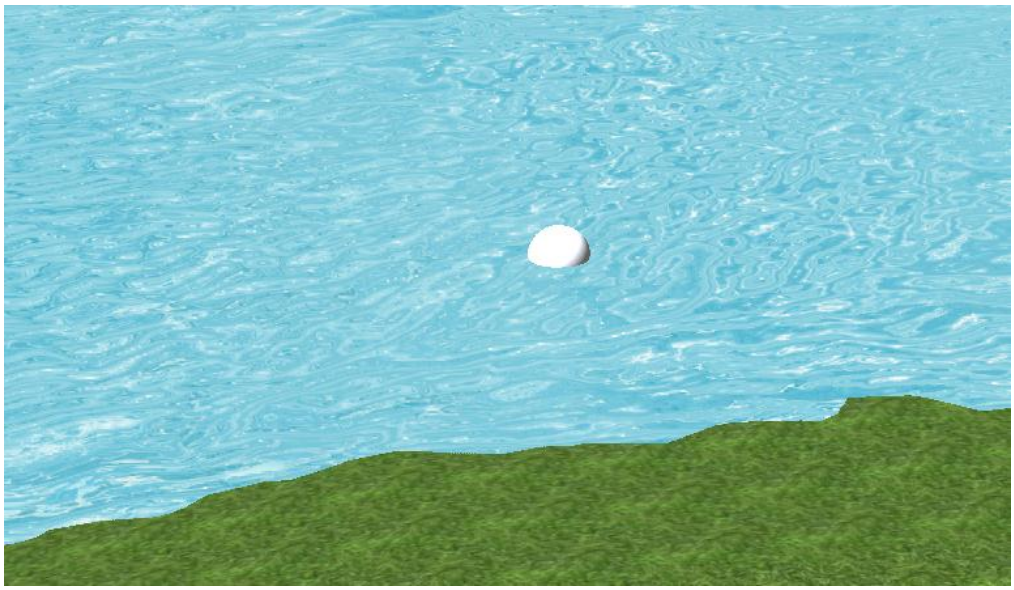

Figure 4.5 – Higher Value For Noise

Figure 4.6 – Dirt Texture in the Water Shader

Chapter 5: Analysis and Evaluation

Section 5.1: Conclusion

The water simulation looks realistic and has very good performance on a large space with a terrain in comparison to other water simulations in WebGL like the example showed in Figure 1-1. The project includes all the advantages that Three.js provides and it is highly programmable. The thesis accomplished all its objectives and even a little more.

Section 5.2: Future Work

There is more work to do with the water simulation. For future work, the simulation can have collision detection with objects. For example, when the waves move up there is clipping between the terrain and the water, and if collision detection is implemented this will not happen. Also, the simulation can be more interactive; this means that objects in the water like the sphere would move and that any interaction with the water will cause a ripple effect.

There are always more improvements to work on in the world of computer graphics, and this water simulation is not the exception.

References

Akenine-Moller, Tomas, Haines and Eric, Hoffman, Naty. "Real-Time Rendering, Third Edition", CRC Press, pp 11-24, July 2008

Conrod,Jay *Water simulation in GLSL* 02 July 2010. 20 Feb, 2013
< http://www.jayconrod.com/posts/34/water-simulation-in-glsl> Web

Danchilla, Brian. "Beginning WebGL for HTML5", Apress, pp 42-49, Aug 2012

Munshi, Aaftab, Ginsburg and Dan, Shreiner Dave. "OpenGL ES 2.0 Programming Guide", Addison-Wesley Pearson Education, pp 1-5. 77-90, Aug 2008

Parisi, Tony. "WebGL Up and Running", O'Reilly, pp 2-7, Aug 2012

Wallace, Evans *WebGL Water*. 30 Sept 2011. 10 Oct, 2011
<http://madebyevan.com/webgl-water/> Web