A Prehistoric Scene of Mars in WebGL

Final Report for CS39440 Major Project

Author: Samuel Snowball (sds10@aber.ac.uk)
Supervisor: Dr Helen Miles (hem23@aber.ac.uk)

07th May 2017

Version 1.5 (Final)

This report is submitted as partial fulfilment of a BSc degree in Computer Science (G400)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name: Samuel Snowball

Date: 19/04/2017

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Samuel Snowball

Date: 19/04/2017

Acknowledgements

I am grateful to the Khronos group for designing WebGL and also to Google and Mozilla for keeping their browsers up to date. Mozilla also has a great tutorial series on the basics of WebGL, including various pages of documentation.

I would like to thank my supervisor Dr Helen Miles for her continuous support throughout.

Last but not least I would like to thank Greg Tavares for his tutorial series on WebGL fundamentals and also his m4.js matrix math library.

Abstract

A prehistoric scene of Mars created in WebGL (Web Graphics Library), GLSL (OpenGL Shading Language) and JavaScript. The scene allows users to roam around and explore Mars as it existed in its Noachian period, around 4 billion years ago.

The motivation for the project was to learn how computer graphics work. How can we display things that look like 3D, on a 2D screen? Most of the heavy lifting in computer graphics is done for us by the use of game engines and libraries. However these hide the details of how the graphics actually get displayed. They also do not give us much flexibility or control over the rendering and efficiency of the scene. There have been few projects developed from scratch in WebGL. Most of the graphical scenes on the web use libraries to take care of the graphics for them.

Most of the scenes features, such as instanced rendering and rendering to textures, have few online tutorials. Therefore this project is a useful learning tool as it demonstrates these complex techniques.

Contents

1. Background, Analysis & Process

1.1. Background

The original aim of this project was to create an interactive Mars mission control game, where the user would roam around as a rover, completing various tasks. The project changed direction after the mid project demonstration. It became clear the game aspects of the project were not working. Therefore, these aspects were removed and the project just focused on the graphics instead.

I wanted to create the game in 3D as this type graphics had always fascinated me. How can we display 3D objects on a 2D screen? I had some experience in 3D graphics with game engines and libraries, but I never found enjoyment in using them. These tools covered up the details of how the scene was actually being displayed. This was the main thing I wanted to learn from building the project. I wanted to dive deeper than just dragging objects into a premade 3D scene, adding textures and then hitting play.

I was interested in low level graphics before the project came out and gained some experience in OpenGL (Open Graphics Library) over Christmas. Regardless of if I got this project, developing low level graphics was a skill I wanted to have. Luckily I got assigned this project and with my head start in graphics programming, I was ready to start.

You can check if your browser can run the project by viewing the user manual here [12]. An overview of what has been added to the project can be seen by viewing the sprint additions file [13].

The project has been hosted on GitHub since its creation and the repository is available here [11]. Having a public repository allows anyone to come and see the progress of the project, this is especially useful for my supervisor. GitHub also stores numerous backups of the project at different stages in development.

1.1.1. Related Projects

NASA had developed a Mars rover game. Unfortunately, there was not anything that could learnt from this as it had obviously built in a rush, being 2D and lacking game playability. The project did not have available source code and was not specifically developed in WebGL.

As WebGL is a new technology, there have not been very many games or projects created in it, at least not from scratch. Most web games or scenes use the three.js library to handle the graphics for them. Therefore, there are few learning resources for developing WebGL from scratch. However, as WebGL is based off OpenGL ES 2.0, most OpenGL projects are useful learning resources. The main project that inspired this one, was a first person game in OpenGL [36]. The great thing about the project mentioned above is that it is open source and well commented.

1.1.2. Language Research and Comparisons

There was a variety of options to develop the scene. Game engines, libraries and raw graphics libraries where all available options. However, these engines and libraries cover up the details of how the graphics actually get displayed.

Using game engines like Unity or Unreal would have created a very realistic looking scene. They were designed to make creating games as fast and as painless as possible. Most engines will handle: user input, sound, rendering, memory management and many more features for you. Most often, engines will have sub-engines handling complex physics and collision detection. Having all of these pre-built features allows you to simply focus on building the game, rather than the engine behind it.

Engines are powerful, if you know how to use them. They require lots of time to be able to use them effectively. It might take too long to complete the project after learning how to use an engine. They also require lots of memory, have long installation times and are very computationally expensive. As this was a relatively small project, rather than an AAA sized game, the overhead of using an engine was not worth it. With a game engine there is also less flexibility, we cannot fix bugs or render more efficiently (unless it is open source).

Game engines have their place, and you would rarely write a game in the industry without one. However for learning purposes, developing this project at a lower level was the best way to go.

Mid-weight graphics libraries exist giving us a lower level option than game engines. Three.js is an example of this. It is a 3D JavaScript library for rendering graphics within a browser, using WebGL at its back end. Libraries like three.js are much lighter weight than full game engines, but they still cover up the details of the graphics actually get displayed.

A lower level option was Direct3D. This is an API built to render graphics on Windows machines. If we wanted to create a game or display graphics with Direct3D on multiple platforms, it would not be possible. Still, most games are written in Direct3D due to its solid Microsoft framework and developer toolsets. It can also make use of the other DirectX libraries such as DirectSound. However, there are more flexible alternatives than Direct3D and it offered no benefits to the project.

A flexible alternative is OpenGL, which is a C API for rendering graphics on desktops. Unlike Direct3D, OpenGL can run on various different operating systems such as: Windows, MacOSX, Linux and UNIX. This means OpenGL applications are highly portable. This is the biggest reason why OpenGL is the most widely used and supported choice for graphical applications. OpenGL has been around for 20 years, having many stable releases, good tutorials and proper documentation.

It is recommend to use C, C++ or Java for the base code of an OpenGL application. These are the most widely used languages paired with OpenGL. Most of the tutorials and documentation is aimed at these languages. However OpenGL was not the last option to develop the project. There was an alternative to OpenGL called WebGL. This is a JavaScript API for rendering graphics within a browser. It is based off an older version of OpenGL for embedded systems. This means the base code of the project would be in JavaScript, rather than the base code being in C/C++/Java using OpenGL.

The graphics aspect alone for the project will be difficult. Therefore having a good knowledge of the base language is critical to the project's success. If the base code was poor, then the graphics would suffer as well. Since JavaScript is a beginner friendly language, the WebGL graphics would be easier to write on top of it.

All graphics APIs use some form of shader language. WebGL uses GLSL (OpenGL Shading Language).

Hence the final languages I decided to use where: WebGL, GLSL and JavaScript. However, I had next to no experience with GLSL and no experience with WebGL. Learning these new languages whilst going along was challenging, however my desire to learn outweighed the problems I faced.

1.1.3. WebGL

WebGL is based off OpenGL ES 2.0 (an old version of OpenGL for Embedded Systems). This means it lacks many features of the modern graphics libraries such as OpenGL and Direct3D. These other graphics APIs are much better documented and have more learning resources, as they have existed longer. WebGL was designed to run within browsers, rather than these others libraries which run on desktops.

Programming in WebGL requires an understanding of what is actually going on at the low level to get graphics to display on the screen. Learning WebGL is useful due to its flexibility running on various different operating systems, rather than an alternative like Direct3D, which only runs on Windows.

1.1.4. WebGL Requirements

To run WebGL scenes, up to date hardware and graphics drivers are required. As WebGL runs within a browser, an update to date version of the browser is also required. But still, having an update to date browser might not be enough. As WebGL itself is just a specification (not an actual implementation) it is up to the browser companies to implement WebGL themselves. To see if your browser can run the project, view the user manual [12].

Modern OpenGL can use a GLSL version of up to 4.40. However the more up to date GLSL version used, the more is required from the hardware. This means if a program is using GLSL version 4.40, it will not be supported on as many devices as an older version. As WebGL uses GLSL ES version 1.00, it is supported on the vast majority of hardware. WebGL was essentially built to run anywhere. Its 1.0 version will run on mobile, tablets, IOS, Android, Chrome and Firefox. However with this project, it is limited to the most up to date version of Chrome and Firefox, due to the extension libraries used. These extension libraries are necessary for the more advanced features of the scene, such as instanced rendering.

1.1.5. Benefits of WebGL

Although WebGL is low level, it has many advantages over alternative approaches. It has no heavy engine which means scene loading times are much faster. It is also less computationally expensive as it only requires a browser, rather than a full game engine or library. WebGL requires no plugins, installation or download, making it much easier to access for the end user.

As WebGL is essentially developing graphics from scratch (or as low level as you would ever want to go), you have to understand what you are doing to use it effectively. With game engines, you do not have to understand how anything works to create some objects in a 3D scene. This means WebGL is a great learning tool as you have to learn the most important aspects of graphics first. For example, the engine components of

a 3D scene all need to be build first, before worrying about any other less important features like lighting or texture effects.

WebGL is also extremely fast allowing us to create graphically complex scenes. It also gives us the power to display the graphics how we want. This is done through the use of customizable shaders. By using WebGL, we have other technologies readily available. HTML APIs and other web technologies can easily be integrated into WebGL applications.

WebGL (or graphics in general) has many uses such as: medical imaging, image processing, games, simulations, augmented reality and data visualization [35].

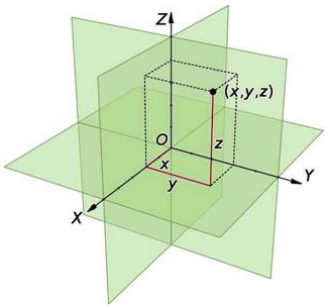
1.1.6. Benefits of working within a browser

WebGL runs within a browser. So when working with WebGL, you get all of the benefits of working within a browser. HTML5 has very easy APIs to tap into such as audio, local storage and text rendering.

If the project was not switched from a game to a graphical scene, then easy access to networking would have been useful for multiplayer. Libraries like socket.io are extremely easy to get two clients communicating with a server exchanging data. Because of libraries such as socket.io, it is easy to interoperate with web applications due to the ease of networking. You can also get resources at runtime from URLs easier, instead of having large amounts of memory pre allocated for textures, video and libraries.

1.1.7. Cartesian Space

A vertex is just a point in a space. We usually think of vertices in Cartesian space. This space is just a set of axes which cross at an origin. Points in computer graphics are usually represented in 2D or 3D. The vertices can be connected to form meshes, also known as objects. The image on the left [37] shows a rectangular mesh, created by connecting several points together. The vertices are defined in the Cartesian space described above.



1.1.8. Vectors

A vector has a length and a direction. The key difference between a vector and a vertex is that the vector does not have a position. A vector from [0, 0, 0] to [1, 1, 1] is exactly the same as the vector from [5, 5, 5] to [6, 6, 6]. The same vector can be used anywhere so long as the length and direction of it does not change. This vector can be used as a displacement for vertices, for example moving the vertices of a cube. Vectors and vertices are also represented in the same [x, y, z] format.

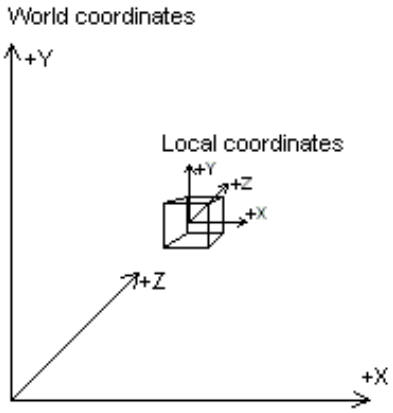
We can use various operations on vectors. The dot product is an operation to calculate how much two vectors are pointing in the same direction. This is especially useful for things like lighting calculations. We can also perform another operation on two vectors called the cross product. This is used to find the normals of surfaces, also useful for lighting.

1.1.9. Model to World

Imagine we have a 3D cube. All of its vertices are defined with respective to the cubes origin at (0x, 0y, 0z). This is called the objects coordinate system, also known as the model coordinate system. For every new object, its vertices are defined relative to its origin.

Now you have that cube, imagine you want to position it in the game/scene world. To do this, translations need to be applied to the vertices of the cube. These translations can be formed in 4x4 matrices. Matrices are used as they can combine several operations into one matrix, rather than having to apply operations to the vertices individually. For example a singular matrix can contain: where the point should be positioned, how it should be rotated and how it should be scaled.

The world coordinate system contains all of our objects. Just imagine two coordinate systems within one another. The image to the right [38] shows just that. This positioning, or translation of the cube is done via a model matrix. Rather than thinking of it as a model matrix, it is easier to think of it as the model to world matrix. As this model to world matrix moves our model, into the world.



The below image [39] shows a 4D vertex that we want to move into world space. Imagine this is one of the vertices in the cube to the right. We have to convert our original 3D cube vertex to a 4D vertex (a homogenous coordinate). This conversion must be done to be able to multiply it by the 4x4 matrix. Having a 4x4 matrix allows us to fit all the transforms (positioning, scaling, and rotating) into one matrix. The vertex output on the right is the vertices translated position in the world.

Translation matrix

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ z + dz \\ 1 \end{bmatrix}$$

Not only can we use matrices to move the vertices of an object, but we can also use matrices to scale them as well. The below image [39] shows how we can scale an object by using a scaling matrix. This takes in (Sx, Sy, Sz) which contains the scale values for each (x, y, z) point of the object. For example if we wanted to scale an object by 2 in all x, y and z directions, the scaling vector would be (2, 2, 2).

$$H_S(\vec{S}) = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ Where: S is a 3D Scaling Vector

Lastly, rotations can be applied to vertices by the use of a rotation matrix. The rotation matrix math is not mentioned here as it quite lengthy and has not been manually implemented. Instead, all of the matrix math is taken care for us by the m4.js library [1]. Using a matrix library like m4 is extremely useful, as we no longer have to write these matrix operations from scratch.

By performing these translations, the cube is now in the world's coordinate system. This is where all objects in the scene are placed. This model to world translation is done in our vertex shader, defined later on. The objects have their own coordinate system, starting from their origin, but we position them into the world's coordinate system. This is so all objects are now defined with the same origin (0x, 0y, 0z). If all objects are within the same coordinate system (the worlds) then it is easy to move all of them at the same time. This is useful for moving the entire world, giving the illusion that the user is moving within the world.

A benefit of matrices is instead of multiplying them all together, then lastly with the vertex (translationMatrix * scaleMatrix * rotationMatrix * vertex). We can instead just multiply all the matrices together once and apply that single matrix with each vertex of the model. This saves multiplying the matrices together for each vertex we want to translate. As previously mentioned, the final matrix containing the translation/scale/rotation values is usually called model to world matrix. This is also known as the full transforms matrix.

When multiplying matrices together we need to be careful, as multiplying them in different orders yields vastly different results. The usual way is: scale, rotate then translate, which in code looks like:

```
matrix4 fullTransforms = translate * rotate * scale (read operations from
right to left).
```

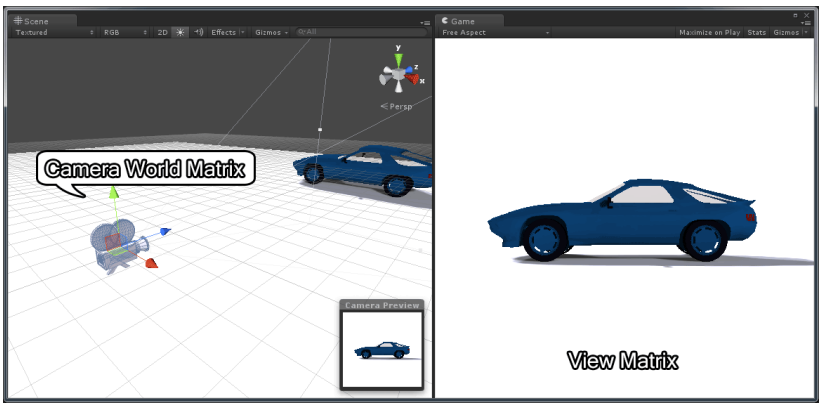
This is the usual case as we would want to rotate and scale the object about its origin, rather than about the point after we have done the translation.

1.1.10. World to View

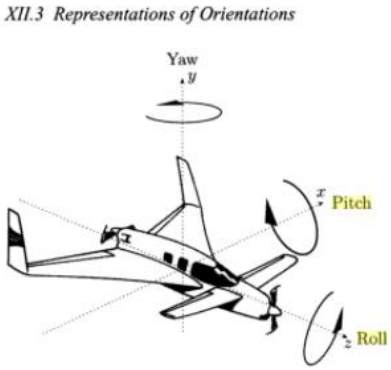
A camera is an object in our world like any other, it has a position and orientation. This is represented by a camera matrix. It is essentially the same thing as a model matrix for a standard object. This camera matrix holds the cameras position and orientation. The camera can be moved by updating the cameras position in this matrix. When the scene is displayed on the screen, it appears that the scene is rendered from the cameras position. However this is not the case. A view matrix is created which is the inverse of the camera matrix. This makes the camera at the center of the world and

moves everything with respect to it, as this is actually what happens in graphics. Like with the model matrix, this world to view transformation is done within the vertex shader.

The image below [52] shows the camera in the world position (left). The view matrix is then applied and the camera becomes the origin of the scene (right). The scene is finally rendered with the camera at the center of the world.

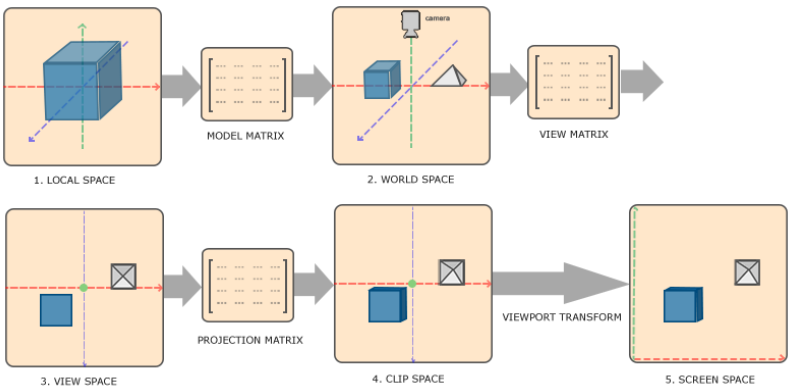


The camera class has been implemented with the standard yaw and pitch values. Knowledge of this was gained from [50] and [51]. These values represent the rotations of the camera around its axes. As seen in the image on the right [50], the yaw value represents the cameras rotation around the Y axis, useful for looking left and right. The pitch value represents the cameras rotation around the X axis, useful for looking up and down. The roll value has not been used as rotating around the Z axis is not needed.



1.1.11. View to Projection

The projection matrix takes the world coordinates and squashes them to clip space coordinates. These clip space coordinates range from -1 to +1 in all x, y and z axes. The below image [40] shows a slightly minified version of how a cube appears on screen. The cube is taken from its own coordinate system, placed into the worlds, transformed into view space and is then projected into clip space. Finally, the viewport transformation is performed to display it on the screen.



The viewport transformation is what converts our 3D world into 2D screen space. After all, how is it possible to display a 3D point on a 2D screen? The answer is, it is impossible. We need to pass the screen, or frame buffer, 2D coordinates to display. The conversion from 3D to 2D is done via the Z coordinates. These Z coordinates are used to sort the X and Y pixels to see what is visible. After this sorting has been done, the Z coordinates are simply thrown away and we are left with a 2D image that can be displayed on the screen.

1.1.12. Vertex Buffer Objects

Vertex buffer objects, or VBOs for short, are places in memory on the GPU. We need to put all of our data that we want WebGL to use in these objects. For example, all of our vertices and colours need to go in these VBOs before we can render the data on the screen.

This is the layout of a VBO, it just contains data.

MyVBO = | 0.0, 1.0, 2.0, 0.5, 0.0, 1.0 |

There is more detail on VBOs, with an example, in the programmable graphics pipeline section.

1.1.13. Shaders and Programs

Shaders are programs executed on the GPU, written in GLSL. This is a C style language which allows us to specify how our data should be manipulated on the graphics card. These shaders are needed to process our data, to eventually get it displayed it on the screen. The main types of shaders are vertex and fragment shaders. The vertex shader processes all of our vertices and the fragment shader processes the fragments/pixels in between those vertices.

The shaders used in this project are located in separate JavaScript files. Once the shader has been written, similar to a C program, it has to be compiled and linked.

The compilation of the shader happens in the file in which the shader content is defined. For example in WaterFragmentShader.js file, we call:

```
gl.compileShader(waterFragmentShader);
```

After the vertex and fragment shaders have been compiled and checked for errors, we can attach them to a program. A program is the combination of the shaders, this program has to then be linked. We finally call gl.useProgram() with our program as a parameter. This says we want to use it on the graphics card to process our data. In this project we have multiple programs and shaders, for rendering different things. For example this project includes code to render a skybox, this is done in a separate skybox shader for readability. So when we want to render the skybox, we swap the current program with a call to gl.useProgram(skyboxProgram). We then render the skybox and then switch back to our default program, to render the rest of the scene.

The below code shows the creation of the program used to render the water. It attaches the waterVertexShader and waterFragmentShader variables. These contain the actual content of the shaders. It is then linked and afterwards, we can use it.

```
var waterProgram = gl.createProgram();
gl.attachShader(waterProgram, waterVertexShader);
gl.attachShader(waterProgram, waterFragmentShader);
gl.linkProgram(waterProgram);
console.log("waterProgram status: " + gl.getProgramInfoLog(waterProgram));
gl.useProgram(waterProgram);
```

Now we have created a program with some shaders, we need to send data to them. This can be done by getting the location of the variable in the shader we want to send data to. We get the location of the variable in the shader and access it through JavaScript, passing it the data we want.

Sending data to shaders is done within the: MainProgram, WaterProgram and SkyboxProgram classes. Each class has its own version of the updateAttributesAndUniforms function. These functions pull the global matrices (model, view, projection) and other values and pass them into the shaders. All of this happens before the draw call is made.

In this example, we want to be able to send the lightPosition and lightColour into the shader. As previously mentioned, to send data to a shader, we need a variable located in a shader to send it to.

Let's first define variables in a shader to send data to. The uniform/attribute types are explained later on. The below code is in a shader, written in GLSL.

```
'uniform vec3 lightPosition;',
'attribute vec3 lightColour;',
```

Now we have some shader variables, to be able to send data to them, we need to get their location. This can be done with calls to:

```
var location = gl.getUniformLocation(program, 'lightPosition);
// or
var location = gl.getAttributeLocation(program, 'lightColour);
```

We now have the location of variable in the shader that we want to send data to. It is finally time to send the data to the shader. Depending on the variable type used in the shader, we call different functions. To send a uniform variable to a shader, in our JavaScript updateAttributesAndUniforms function we call:

```
gl.uniform3fv(location, lightColour);
```

This passes the data in the JavaScript variable lightColour, to the location variable (which references the actual shader variable) we got earlier. The data has now been sent to the shader! All of the shader location variables are defined in the: MainProgram, WaterProgram and SkyboxProgram classes.

This is a reasonably simple process but seems lengthy when explained step by step. It has to be done for every variable we want WebGL, or our shaders, to use.

Like with all C programs, there needs to be a main function defined in each shader. Finally, there are some global variables that a vertex and fragment shader must output. A vertex shader must output the final position of the vertex, this is done by

setting the gl_Position variable. The fragment shader must output the gl_FragColour variable, which is the final colour of the fragment/pixel.

1.1.14. GLSL Types

As GLSL is an entire language, it has its own set of functions, keywords, types and global variables. To reiterate, WebGL uses GLSL ES version 1.00. This means most of the examples and tutorials online are not relevant, as they use a more up to date version of GLSL.

The main types used in this project are attributes and uniforms. Attributes are data that changes over the course of a draw call, for example a vertex position. We would not want to use the same vertex over an entire draw call. Instead, we would want to change the vertex that is being processed within the draw call. This allows us to draw multiple vertices with a single draw call. On the other hand, we can have uniform variables. These are the same over an entire draw call and are mainly used for matrices.

Another variable type used in GLSL is a sampler2D. This is a variable that we use to sample a texture for a given point. We have to provide GLSL a texture to sample and also the texture coordinates to sample it at. This returns us a colour value from the texture, allowing us to texture our objects.

If we want the fragment shader to process something, it must first be passed to the vertex shader (unless it is a uniform variable). The varying keyword is used for passing data from the vertex shader, to the fragment shader.

The meaning of this varying keyword depends on what shader it is used in. If the varying keyword is used in a vertex shader, then it means the variable gets passed to the fragment shader. If the varying is in the fragment shader, then it means that variable gets taken in from the vertex shader. For example, a vertex shader will sometimes take in a colour value, but it will not actually use it. Remember the vertex shader only processes vertices. Instead, the vertex shader will just pass it along to the fragment shader as that is where the fragment/pixel colour is calculated.

1.1.15. Fixed Function vs Programmable Graphics Pipeline

WebGL uses something called the programmable graphics pipeline, as opposed to the old fixed function pipeline. This newer pipeline was implemented into later versions of OpenGL. Hence the new pipeline is in WebGL, as it was based off OpenGL ES 2.0. This pipeline is just the series of steps needed to take some data and display it on the screen.

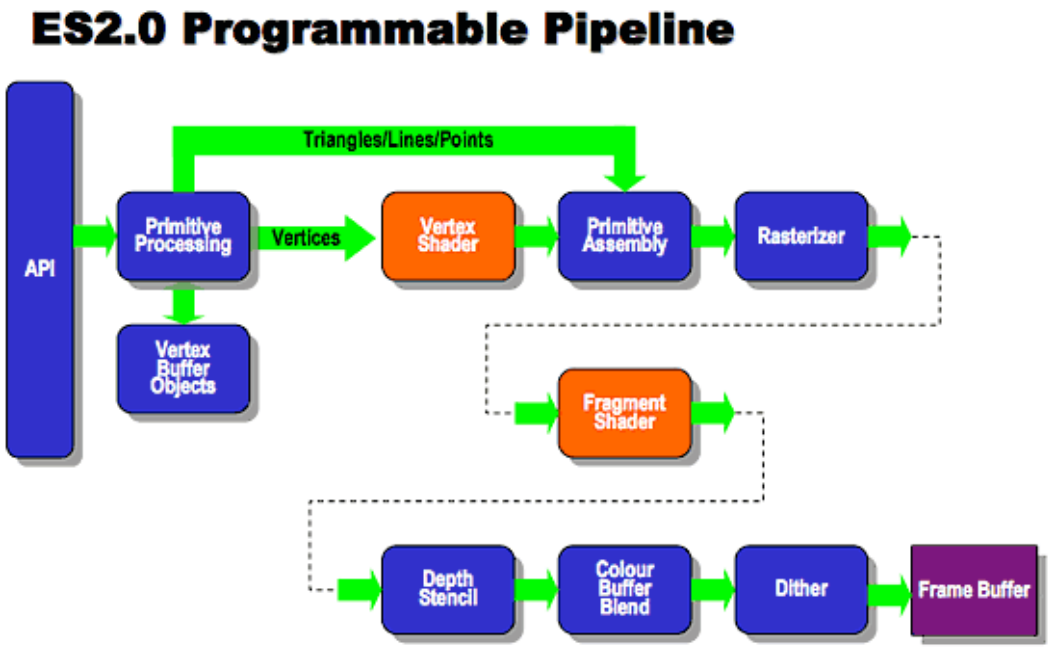
In the old fixed function pipeline it was very one size fits all. There was no functionality for doing any custom calculations (lighting/fog), only what was built in. We could opt in for the features that we wanted. For example, the pipeline could calculate fog, or we could opt out have it not calculate fog. However, there was no functionality allowing us to customize the operations.

But now, using the new programmable pipeline, we can use our own programs (containing our shaders). These programs and shaders replace parts of the graphics pipeline. We now have the ability to manipulate the data in whatever way we want.

These shaders give us much more power over what gets rendered. However a disadvantage of these shaders is that they add complexity to application. Previously, we could not touch the shaders as they were fixed. But if we use customizable shaders, we have to manage some stages of the graphics pipeline ourselves.

For simple applications which are not going to do any fancy graphics, then an older version of OpenGL (pre 2.0) would better suited. These older versions of OpenGL use the fixed function pipeline without any customizable shaders. But with using WebGL, we do not have the option of a fixed function pipeline – so we must define our own shaders.

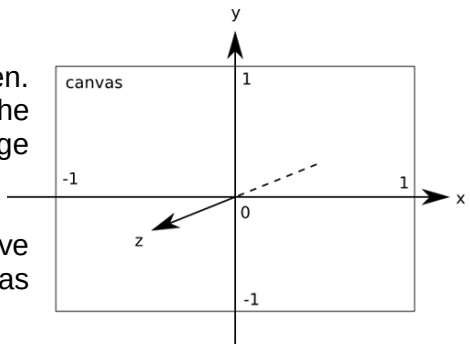
1.1.16. Programmable Graphics Pipeline



Before the details of the above pipeline are explained [41], it is important to understand that there is not just one of these pipelines which everything goes through. There are usually a few thousand of these running in parallel on the graphics card. Each of them processing a vertex/pixel at a time. This is why GPUs are used for graphics, due to them being able to process a large amount of data in parallel.

The scene data (vertices, colours) are created by us on the CPU. The data then gets passed to WebGL (the API component on the left) through the calls made to its API. WebGL then takes care of passing the data through the graphics pipeline, all on the GPU. As WebGL uses the programmable pipeline, we have to replace the vertex and fragment shaders (highlighted in orange in the above diagram).

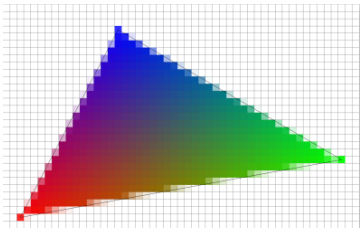
1.1.16.1. Stage 1 – Data input and binding



In this example, we will render a triangle onto the screen. The triangles vertices are defined in clip space. The WebGL clip space system is shown in the right image [42]. It goes from -1 to +1 in all of the x, y and z axes.

Take the triangle in below image [43] and the above coordinate system. The triangle vertices would be as follows:

```
var vertices = [
    (-0.3x, +0.8y, +0.0z), // top middle vertex
    ( -0.9x, -0.9y, +0.0z),// bottom left vertex
    (+0.8x, -0.3y, +0.0z)  // bottom right vertex
];
```



The colours for each vertex also need to be defined.

```
var colours = [
    (0 red, 0 green, 255 blue), // top middle
    (255 red, 0 green, 0 blue), // bottom left
    (0 red, 255 green, 0 blue) // bottom right
];
```

For WebGL to draw something, you have to provide that data and then tell WebGL how to connect the data and then finally how to draw it. Take the triangle vertices in the above example. This data needs to be put into GPU memory. This is done by first creating a VBO (discussed in the vertex buffer objects section), binding that buffer and then finally putting our data in the buffer.

```
// a piece of GPU memory, in which we will store our data
var vertexBuffer = gl.createBuffer();
// tell WebGL we want to use the buffer we just made
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
// put the vertex data into the buffer, which gets sent onto the GPU
gl.bufferData(gl.ARRAY_BUFFER, vertices);
```

After sending data onto the GPU, we have to describe the data we just sent. This is done with a call to vertexAttribPointer.

```
/*
This call describes the data that we just sent to the GPU

Parameters:
The shader location variable that we are sending it to (position location,
colour location)
The size of the data per primitive (3 because we are drawing triangles)
Data type of each component in the array, FLOAT
Whether to normalize the data or not
Stride, the gap between the defined data. This is used when defining
vertices/colours in the same array, but we have not. So it is set to 0.
Offset to the data in the array, again, set to 0.
*/
gl.vertexAttribPointer(shader_location, 3, gl.FLOAT, false, 0, 0);
```

The above 4 calls are then repeated for the colour data as well.

This process of uploading data to the GPU is slow, so it should be minimized in render loops. At render time, we need to select the buffer on GPU which we created earlier, as it contains our data we want to render. We can do this with a call with a call to:

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer)
```

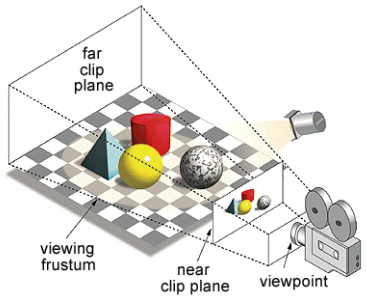
Do not worry about the first parameter. But the second is the buffer we created earlier containing our vertex data. We also have to describe the data again and repeat the process for the colour buffer. Now, we can finally issue the draw call.

```
/*
Parameters:
Drawing mode, what are we drawing?
The start element
How many vertices to draw
*/
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

After we have issued the draw call, the vertex shader gets executed.

1.1.16.2. Stage 2 – Vertex Shader

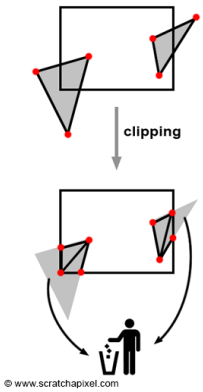
The vertex shader will get run for each vertex (3 vertices in this triangle example). We can also input 4x4 matrices (model to world, world to view, view to projection) into the shader. These matrices specify where the vertices should be positioned in the world.



After these matrices have been applied to the vertex, the shader outputs the vertices coordinates in clip space. Clip space ranges from -1 to +1 in the x, y, and z axes. Once the clip space coordinates of the vertices have been calculated, then any vertices outside of the viewing frustum will be clipped or culled. The viewing frustum is shown in the image on the right [44]. This contains everything the camera should see. The projection matrix converts everything in the viewing frustum to clip space coordinates. When defining a projection matrix, we can specify how much of the scene to process. In the below example, we want to view from 0.1 units in front of the camera, to 512 units away from it. We also specify the field of view and aspect ratio.

```
//Projection variables
var fovInRadians = Math.PI * 0.3;
var aspectRatio = window.innerWidth / window.innerHeight;
var zNear = 0.1;
var zFar = 512;

//Projection matrix turns world coordinates to clipspace
var projectionMatrix =  m4.perspective(
    fovInRadians,
    aspectRatio,
    zNear,
    zFar
);
```

If any of the objects are calculated to be completely outside of the viewing frustum, then they get culled. This means the entire object is discarded as the user cannot see it. It is important to get rid of any unnecessary vertices as early as possible, if the user cannot see something – why waste time processing it? If the shape is partly in view, then the vertices that are out of view get clipped and recalculated. Then, only the visible parts of the object remains. The image to the right [45] shows clipping taking place.



1.1.16.3. Stage 3 – Primitive Assembly

Depending on the draw call that was issued, WebGL will form different primitives from the given vertices.

For this triangle example, we have issued the draw call:

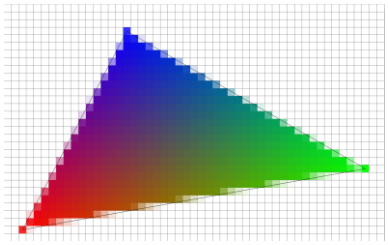
```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

This means we want WebGL to draw triangles between the given points. For every 3 points we give WebGL, it will construct a triangle between them. There are also other drawing modes such as GL.POINTS, GL_LINES and the GL.TRIANGLE_STRIP.

Now we have some colourless shapes, they need to be filled with colour.

1.1.16.4. Stage 4 – Fragment Shader

As you can see with the colours defined in stage 1, we have only defined 3 colours for the entire triangle. If we were to give colour information for each pixel, this would require a huge amount of data. So instead we just define colours for each vertex. WebGL then interpolates (blends) between the colours of the vertices. You can see this by looking at the points in the middle of the triangle in the below image. The pixel colours are blended between all of the vertices colours. This is only noticeable when a triangle is this big. Usually, objects are made of thousands of triangles so the user will not notice this interpolation.



The fragment shader is part of rasterization (or colouring in) stage of the pipeline. It gets run for every fragment between the defined vertices. A fragment just means a potential pixel. It then outputs an (R,G,B,A) colour value. The fragment has not been tested to see if it is visible, so it might not get rendered. Pixels are things that appear on the screen, a fragment might not.

1.1.16.5. Stage 5 – Fragment testing and the depth buffer

What if we calculate a fragments colour value, but then calculate another fragment that should be in front it. The fragment that is closer to the camera needs to get rendered on top of the fragment that is behind it. There needs to be some sort of test to see which fragments are closer than others. This is where the depth buffer comes in handy.

The depth buffer stores all of the depth information for the entire scene, it is just a huge 2D array of values. Objects in the scene are given depth (or Z) values. This information is used to determine what fragments should turn into pixels and eventually get rendered. The current fragments depth is checked against the value stored in the depth

buffer. If the current fragment has a lower depth value than the one it is being checked against, then its value replaces the one in the depth buffer. When we clear the entire screen, we also clear the depth buffer to all -1 values. This allows the depth of the scene to be recalculated for the next frame.

Now the fragment shader has run for every fragment between the given vertices. All of the fragments that pass the depth test are stored into the colour buffer, which is a huge 2D array of pixels. The fragments turn into pixels here as they are no longer potential pixels, but actual pixels. Their depth has been tested and they are definitely going to be rendered this frame. All fragments that fail the depth test are discarded.

1.1.16.6. Stage 6 – Updating the frame buffer

The frame buffer is the final 2D image that is displayed on the screen. It consists of the depth and colour buffers.

Once all objects in the scene have been added to the colour buffer (or discarded), it can be swapped with the current frame buffer to show the newly updated frame. This technique is called double buffering. The pixels do not render onto the screen directly. This would result in parts of the scene being drawn before others. It is better to wait for the entire scene to be rendered to an off screen buffer, before swapping it with the current frame buffer to update the display.

The vertices and data in the scene goes through this graphics pipeline, ideally upwards of 30 times per second, for a result of 30 frames per second.

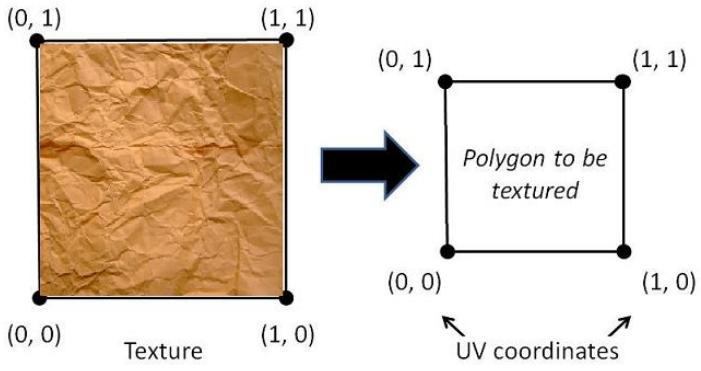
1.1.17. Texture Mapping

Knowledge of texture mapping has been gained from [46], [47] and [48].

Defining the colours of vertices like we did in the previous example is good to get things to display on the screen – but not practical in applications. Let's say we had a model containing 5000 vertices, we would not want to go and define a colour for all of them. Instead, we can generate texture coordinates for the model and use a simple image to wrap over it. Texture coordinates, sometimes referred to as UV coordinates, are 2D coordinates which are assigned to vertices.

Like with colour information, we do not give each pixel a different UV coordinate. This would mean creating a very large amount of data. Instead, we just assign the vertices these texture coordinates, then let our fragment shader handle filling everything in. It does this by using a sampler2D discussed in the GLSL types section.

The left image [49] shows a texture that we want to map onto a square. The 4 vertices of the square need to be assigned UV coordinates. These coordinates tell WebGL where to place the texture on the object.

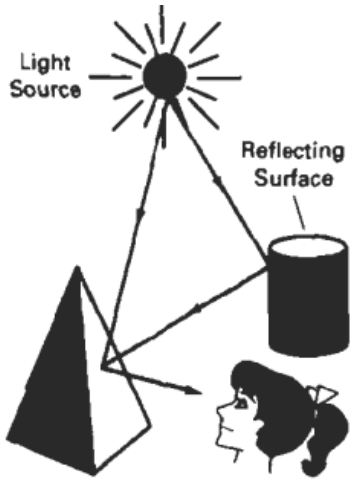


The bottom left vertex of the square would get assigned the texture coordinate of (0, 0). This would map the bottom left corner of the texture, to the bottom left vertex on the square. This process is repeated for the 3 other vertices on the square. Now that we have UV coordinates in place, we can get rid of all the colour information. We now use textures rather than specifically defined colours.

Like with all data, the objects UV coordinates need to be created, put into GPU memory, then bound at render time. The texture we want to use also needs to be bound. When the draw call is issued, the sampler2D in the shader will fill in the fragments/pixels in the square with the correct texture values.

1.1.18. Lighting

Lighting in the real world is extremely complex. Light can reflect off most objects and in turn, those objects become light sources as well. This is known as ambient light, or global illumination. If we tried to calculate lighting vectors as they exist in the real world, the computation time for our scene would be so slow – we might only get 1 frame per second. For this reason, we have to make approximations.



Light sources in this scene are referred to if they give off light directly – rather than through reflection. The image on the right shows how lighting would be calculated in the real world. The point light [55] would illuminate all objects in the scene and in turn those objects become light sources. Therefore the light entering the user's vision is through a combination of these light sources.

Lighting is calculated from normals and light positions. A normal is just a vector pointing away from the objects surface. If the incoming light direction is facing the surfaces normal, then that surface will be fully lit. The less direct the light hits the surface, the less lit that surface will be. Therefore the brightness of a fragment depends on the angle between the surface normal and the light direction vector.

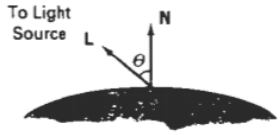


Figure 14-9
Angle of incidence θ between the unit light-source direction vector **L** and the unit surface normal **N**.

In the image on the right [55] the angle that light hits the surface is around 45 degrees. Because of this, the surface will be mildly lit, but not completely lit or unlit.

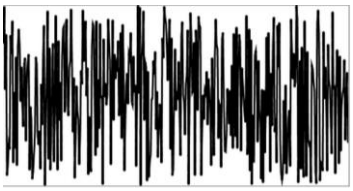
1.1.19. Noise

There should obviously be terrain for the user to roam around on. So there needed to be some sort of terrain generation. This can be done using noise, this is simply a way to generate numbers.

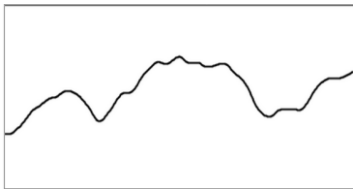
There were two main options for generating the terrain. The first was using existing mars height maps and the second was using perlin noise. The problem with using existing height maps is that there is no technical skill involved. It is as simple as loading an OBJ into the scene. An OBJ file is simply a text file defining the vertices

and other data of an object. This OBJ model would never change and therefore, the terrain would not change either.

Perlin noise allows for the terrain generation to be randomized. This is an algorithm created by Ken Perlin [67] which generates smooth random numbers. These smooth random numbers can be used to set the heights of vertices, which leads to natural looking terrain.



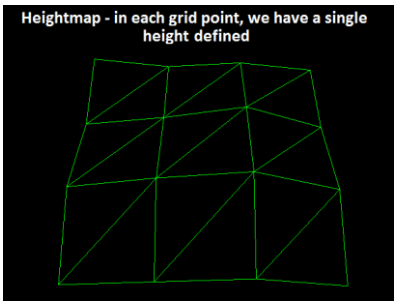
Imagine you used a random() function and plotted the returned numbers on a graph. The output would the similar to the graph on the right [53]. Every number chosen, or every point plotted, has no relation to the number before it, or the number after it.

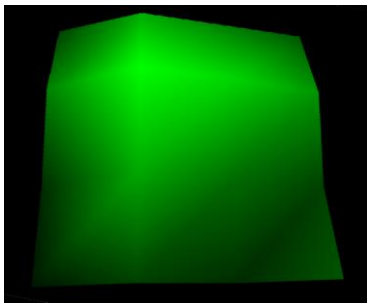


However a perlin function generates smooth random numbers. This means the random number chosen is based on what came before it. Therefore, the value picked can never be very far away from the previous value. Plotting perlin noise values on a graph would look something like the image on the right.

The above examples are graphical representations of 1D perlin noise. The perlin function just returns smooth random numbers, it is up to us as to what we use them for.

Imagine a 2D grid rotated into the screen, representing the terrain of the world. With a 2D perlin noise function, the height value for the cell is chosen, based on the 8 cells surrounding it.





This gives the terrain a smooth natural mesh (a connected set of vertices). Once a texture is placed over the top, we end up with terrain like in the image on the right [54].

1.2. Analysis

All WebGL allows you to do is draw points, line segments and triangles on the screen. Apart from that it handles nothing else for you. There is no premade 3D scene where you can drag and drop objects with nice textures and lighting. It is very lightweight and close to the hardware.

A basic WebGL scene has no: physics, event handling, sound, helpful error messages, loading/saving or fast rendering efficiency. However these are not disadvantages. Instead, these are rewarding tasks that can be implemented and by implementing them, I will learn how they work. This completes one my personal goals of with project which was learning how 3D graphics works.

Features and techniques used will be researched as and when they are needed. Heavy background reading was done before the project was even assigned. This was to learn about all of the information mentioned in the background reading section. This head start on the project was crucial to create a realistic looking scene.

Researching about specific features at the beginning of the project would not be very useful, as the information might not be used right away. Other features might be needed first. The techniques learnt early on might never get used as the requirements could change. The things learnt in the background reading section where mandatory, like how to display things on the screen. There was no initial research done on the advanced features of WebGL.

Working software early is also a priority. It is better to get something less efficient on the screen rather than attempting a complex technique and failing completely. For this reason, the engine components of the scene needed to be built first, followed by the graphics later on.

A specification [14] had to be created with brief details of what the project should contain.

This is a minified list of the initial tasks, when the project was still a game:

Some general information about the project and its game mechanics:

- Focus on first person, perhaps adding in third person later on. A first person view will feel more interactive
- User will interact via keyboard and mouse
- There needs to be cross browser testing

Tasks (in loose priority order, highest priority first):

- Build the 3D environment, allowing a camera to move around in the world.
- Build terrain using perlin noise.
- Allow user to move around the terrain, a camera class is needed.
- Add collision with terrain, rather than flying over it.
- Give the terrain a texture, procedurally generated perhaps.
- Add in rocks. A procedural rock generation library could be used to generate rocks. However I would rather try implement rocks myself. A sphere geometry and using perlin noise on the vertices could give a bumpy, rock-like geometry.
- Start work on GUI's, with a possible minimap.

- Add in various missions for the player to complete.
- Add water.

Project deliverables

- 3D Scene.
- Terrain generation.
- User navigating terrain.
- User collision with terrain.
- Textured terrain/added in rocks.
- Some GUI's and the user receiving and completing missions.
- Then after this, probably random extra features like clouds and audio.

The main focus of these lists was to ensure the main mechanics, or engine, was finished early. If a 3D scene proved too difficult to implement, then it would have been easy to adapt the project to a 2D scene instead.

1.2.1. Alternative Approaches

There was an alternative approach which involved heavy up-front design and planning of all the components in the project. This would cover everything from a class diagram to UI designs and what techniques I was going to use to implement the various features. It would also include how the project was going to be documented and tested.

The features list, or task priority list might have looked like this:

Tasks:

- Research all techniques that are going to be used, only taking the most efficient versions and methods that will appear in the end product.
- Complete a full project specification, outlining all tasks that need to be completed. These tasks are given priorities and start/estimated completion times.
- Build full design diagrams such as the class and flow diagrams.
- Create documentation on the techniques, methods and designs used.
- Start implementing the code.
- Test the code as the project deadline approaches.

With this approach, implementation would start at around sprint 4-5 (at least 1 month into the project). I would have no experience with practical programming of WebGL and would therefore struggle with the advanced techniques chosen. These techniques might have been the most efficient to use, however if they could not be implemented, the whole design would have been a waste of time.

This heavy upfront design is a dangerous approach for the project and therefore, it was not used. It is discussed further in the Process section (1.3).

1.2.2. Security

Since the project began, the security aspects were irrelevant. There is no data collection or storing of personal information. This means all of the data protection principles are out of the question. In the early stages of the project the game loaded

data from HTML5 local storage, from within the user's browser. If the user somehow changed this local storage data, then they would be able to break the scene by loading in positions that were off the map. However if a user breaks the scene, it is just an annoyance for them.

There are no database calls that executed meaning things like SQL injection are non-existent. The scene is usually run on the user's computer, which would contain all of the source files. This means the user can go in and break what they wish – but again, this is not a security concern.

If this project continued as a game, perhaps with multiplayer capabilities, then security would be an issue. If this were the case then the source code would not be available to the regular players. It would not be ideal for players to generate infinite gold or develop hacks to beat other players online. Dealing with this would require heavy server-side checking, ensuring players are in the correct location and that player had reasonable stats (health, damage, speed).

1.3. Process

There were a few software development methodologies that could have been used for the project. The standard waterfall model, SCRUM, extreme programming (XP) and feature driven development (FDD) where all available options. Completing the Agile Methodologies module in the first semester definitely helped with the development of the project. Using an agile approach took little research, as the techniques were already familiar.

Since WebGL was the chosen language for the project, an agile approach was best. The biggest reason is that being agile allows you to focus on delivering working software early. If a waterfall style approach was used, then as mentioned in the analysis section, the code would be implemented from sprint 4-5 onwards. If there were any difficulties with implementing the code, then there would be no easy way to go and change the project design or deliverables.

If there were no unexpected errors or problems when developing with WebGL, then using the waterfall model would have yielded a much better result. More upfront planning and design would have meant the best techniques could have been used from the first sprint. Realistically, learning a new language and encountering no problems is not going to happen. If the requirements and languages of the project were well known at the start, then a waterfall style approach would have been suitable.

Since extreme programming principles mainly rely on a team (code inspection, collective code ownership, pair programming) it was not possible to use. SCRUM is a flexible framework and therefore it made the most sense to use for the project. SCRUM allows us to pick and choose which agile processes we will use. For example, we have the option to use TDD, rather than it being enforced by the methodology.

1.3.1. SCRUM

A task list, or story list, was created at the start of the project [14], as described in the analysis section. A story is just a feature that could eventually get added to the

project. So through the development of the project, many stories have been added and removed from the story list (the product backlog, described later on).

Like all agile methodologies, SCRUM uses fixed time boxes for development. These are called sprints. In this project the sprints are 1 week long. This amount of time was chosen as the project supervisor was available for weekly meetings. This allowed the project to receive feedback each week on the previously completed sprint.

SCRUM has 3 main pillars (or aspects). These are: transparency, inspection and adaptation. All projects developed with SCRUM must follow some of the processes and methods within these pillars.

As part of the inspection pillar, SCRUM uses artifacts. The main artifacts are the product and sprint backlogs. These are simply files which contain information about the project. The product backlog [15] contains all of the tasks that need to be added and have already been added to the project. At the end of a sprint, a retrospective is performed. This is essentially looking back over the weeks work and highlighting any problems, then thinking of how to solve them for the future.

A benefit of using SCRUM is that if the project struggled early on with the chosen languages or design, then it would be easy to change. The language could have been swapped to use a mid-weight library such as three.js. Or, the design of the project could have changed to create it in 2D. If the language choice hindered the project with a waterfall approach, it would require scrapping the entire first month of work (planning and documentation), then restarting the project with the new language/library choice.

The adaptability of SCRUM proved useful when changing the project from a game to a graphical scene. The classes were simply refactored and some documentation and tests were removed.

At the start of each sprint, the stories with the highest priority are taken from the product backlog and moved into the sprint backlog. Then at the start of each day, a 15 minute plan is done, laying out a foundation for how the code will be implemented. This is adapted from the standard 15 minute standup – as there are no people to standup and discuss with.

A common agile practice is having an onsite customer. This was achieved by having a meeting with my supervisor most weeks. The project was demoed to her allowing her to suggest various improvements. This was useful and ensured the project was staying on the right track.

Test Driven Development, or TDD for short, is a technique used for writing code. It is commonly used in agile methodologies such as FDD and XP. It involves writing the tests first, then writing the code afterwards. This is useful in projects where the requirements and techniques are well known. However for this project, this was not the case. The tests would have been very difficult to write as the techniques and tasks that were not well understood.

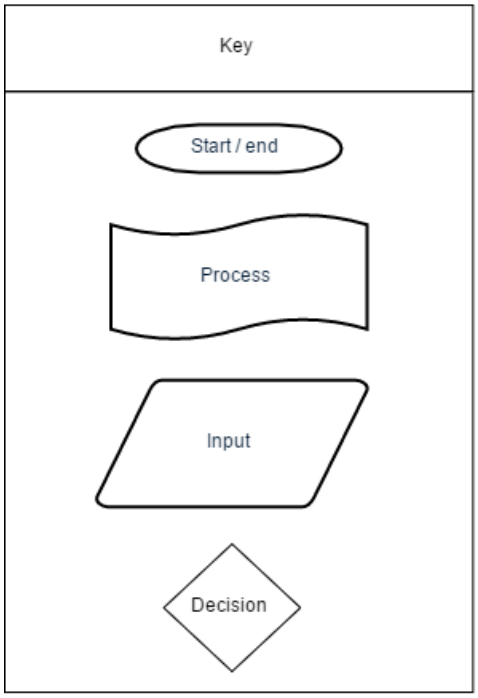
2. Design

This project was developed on Windows 10 using notepad++ as the editor. As WebGL is a new technology no dedicated IDE's have been developed to help program with it. The only thing available is the browser developer console, we can view this by pressing F12 whilst in Chrome/Firefox. This developer console contains a useful debugger and also features to view: resource loading times, code execution times and memory usage. These features where used throughout the development of the project.

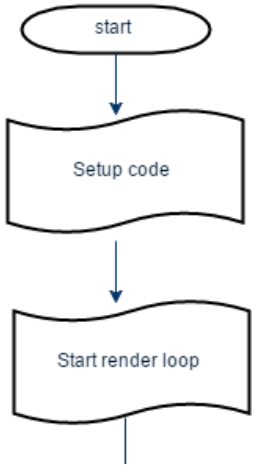
The design of the project evolved throughout. Agile development does not recommend a large up front design, hence why this project did not have one. Tasks were taken from the product backlog then they would be: researched, designed, built, tested and finally documented. There was 15 minutes of dedicated design per day, going over the features that were going to implemented – but really there was design all throughout the day.

2.1. Overall Architecture

The easiest way to understand the scene is to look at the flow diagram of a simplified render pass [16]. First, we define a key (right image) for the different components of the diagram.

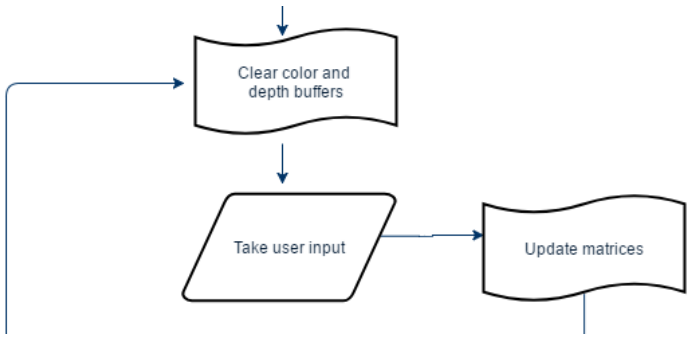


The below image shows the start of the program. The scene begins by creating all of the objects and vertex data. This is done within the MarsScene.js class. After the scene data has been setup (terrain, rocks, water) then the render cycle is started.



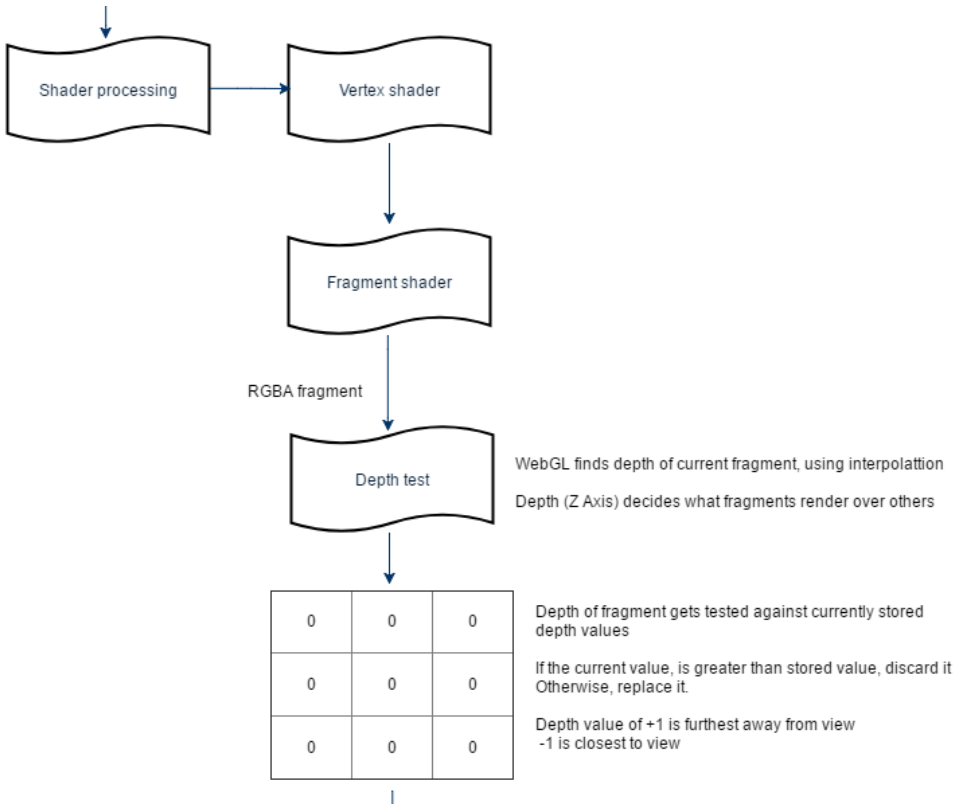
As shown in the image below, at the start of the render loop the entire screen is cleared, ready for drawing the next frame. The program then checks if the user is currently pressing down a key (W for example). The global matrices (defined in program.js) are then updated appropriately. If the user was trying to move forward, then the cameras X and Z position gets updated by the direction in which they are

facing, multiplied by the camera speed. The cameras Y position is not updated here, instead the user can move vertically by pressing R or F (for rise and fall).



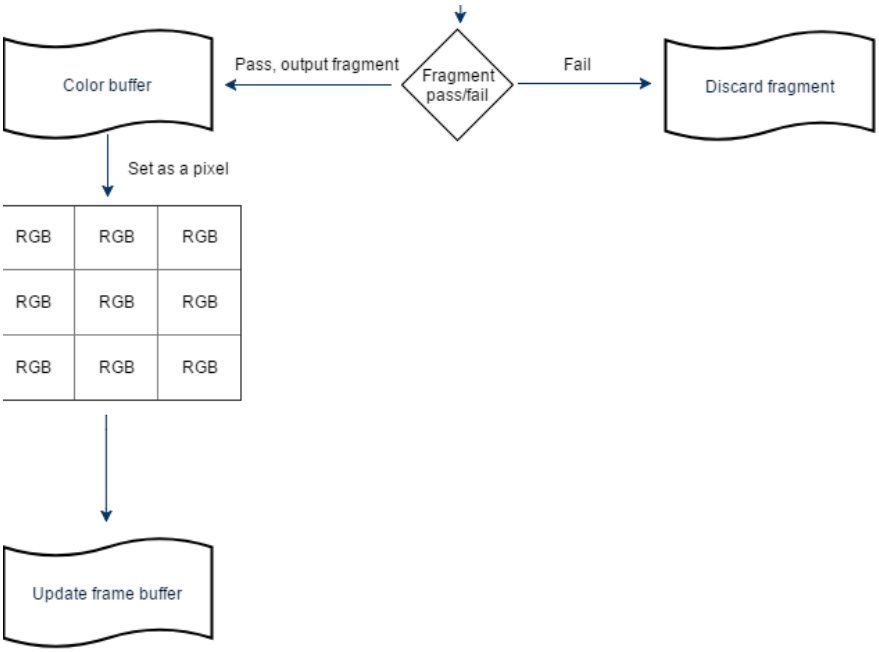
After the matrices have been updated for the current frame, they are passed into the vertex shader. This is done with a call to a version of updateAttributesAndUniforms. The shaders can then process the objects in the scene with the correct matrix values.

The vertex shader runs on every vertex and outputs the final position of it, by setting the global gl_position variable. The information that is not used by the vertex shader is passed into the fragment shader. This in turn performs the depth test.



The fragments that cannot be seen are discarded. This is shown with the decision (diamond) box in the below diagram.

If the current fragment passes the depth test, then it is written into the colour buffer. If the fragment fails, it is simply discarded as we no longer want to process it.



Once all objects in view of the camera have been drawn to the colour buffer, it is swapped with the frame buffer to update the display. The render cycle is then started all over again.

2.2. Detailed Design

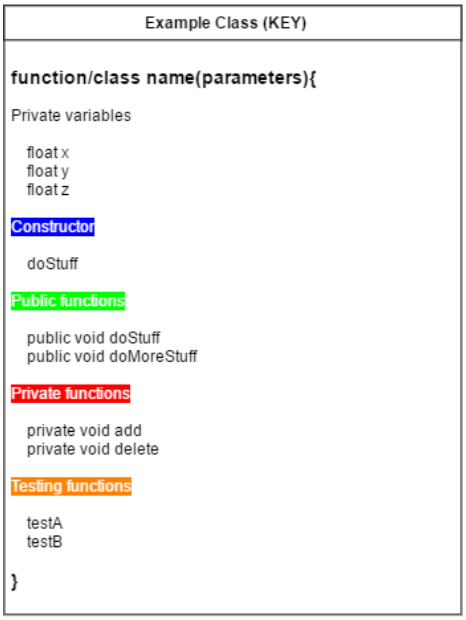
Since this project was developed with SCRUM, the design was constantly evolving. New classes were simply added on, with each new class containing setup and render functions. An object of the new class was then created in the MarsScene.js file. Its render function was then added into the main render loop.

In the folder documentation/class diagram, you will find lots of XML files. These files contain the class diagram of the scene at different stages of development. The only important versions are the minified and final ones, which you can find the png images files for. The XML files can be opened with draw.io [66]. However this is not necessary. The more brackets a filename has, file(1)(1)(1), the later it was created.

2.2.1. Even More Detail

The full class diagram was too big to include here, so instead it is in the appendices located here [18].

The class diagram described below [17] is a minified version of the actual diagram. There have been some small modifications, mainly just removing the variables and methods. However each class, minus the shaders and index, has setup and render functions, even if not shown.

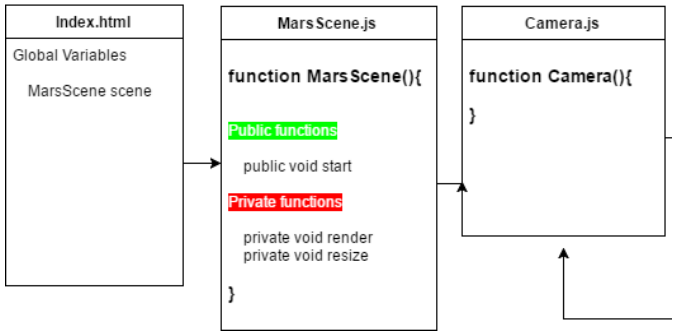


The key of my class diagram, as shown on the right, is defined in the top right of the main class diagram image.

Instead of the regular var JavaScript keyword being used everywhere, it has been replaced with the actual descriptive data type. Using the var keyword in the initial class diagrams added nothing to them. Having the actual type is much easier to read.

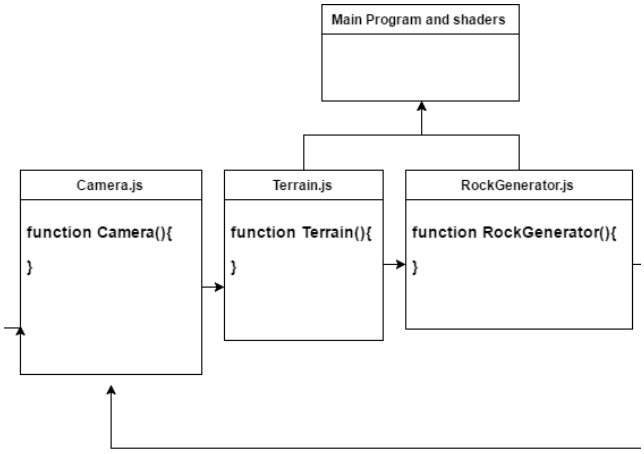
An important part of the class diagram is the switching of the shaders when rendering different parts of the scene.

The scene has an index file which creates a MarsScene object. This MarsScene file is where everything starts. It is where all of the scene objects are created and it contains the main setup/render functions.

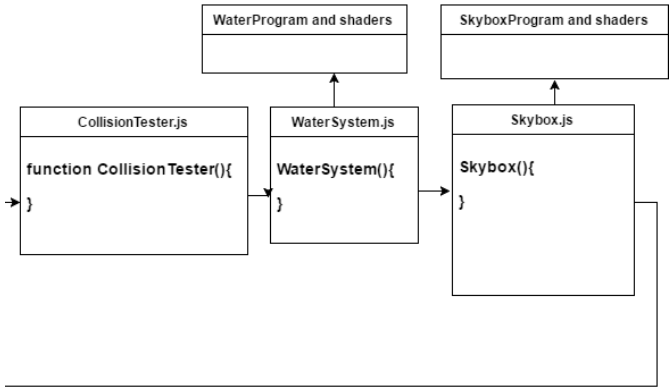


The arrow leading from the MarsScene.js file into the Camera.js file represents the render loop starting. The arrow coming back into the camera class is when the render call finishes, which starts the cycle all over again.

Now, the render pass continues. The camera gets updated and the user's quadrant is calculated. The user's quadrant is needed to render the rocks and the terrain around the user. After the user's quadrant has been calculated, the terrain and rocks can be rendered. Both of these files use the Main program containing the main vertex and fragment shaders. This is opposed to the skybox or water shaders.



After the rocks have rendered, the collision of the scene is tested. This is to stop the user moving through the floor and map boundaries. Next, the water gets rendered. Notice how we swap the currently used program, or shaders, over to the water ones.



Like with the water, when rendering the skybox we have to swap the program to render it. After the skybox is rendered, we swap back to the default MainProgram and shaders. This is the end of a render pass. It then loops from the skybox, back to the camera class for the next frame.

Before the project direction changed from a game to a graphical scene, various user interfaces where designed. The user interface design document is located in the appendices here [19].

3. Implementation

This project started out as a game and continued this way until around sprint 5 when the mid project demonstrations were held. In the mid project demonstration, it became clear that the game did not have enough, or any, playability aspects to it. In the projects defence, it was extremely difficult to create a game where you could sit down and play for hours, whilst programming the graphics in a low level library. There simply was not enough time to do both.

There were 2 different things that were being worked on: the game code written in JavaScript and the graphics code in WebGL. Since the graphics were what I was interested in, writing thousands of JavaScript lines to create a game was not ideal. The game features also took so long to write and because of this, the graphics of the game suffered. After the demo the game elements were stripped out and the project just focused on the graphics.

As mentioned in the sprint additions document [13], the background reading and implementation sections actually started happening around 1 month before the project was even assigned. This gave the project a good head start going into the first few sprints. After all, 2 new languages had to be learnt to complete this project so getting a head start was crucial. This document gives an estimate of how much time was spent on each feature, each sprint.

The full product backlog containing everything that has been added to the project can be found here [15]. However, the sprint additions document as mentioned above is much easier to read.

3.1. Setup

To be able to interact with the WebGL API, we first need a HTML canvas element. Then, we need to create a WebGL context to allow us to draw on it. This is done in the GLSetup.js file.

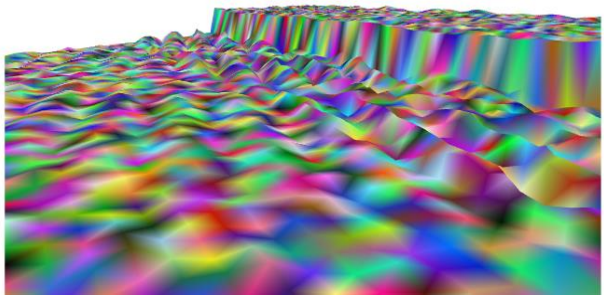
The scene starts in the index.html file by loading in the various project files. This creates a MarsScene object. In the MarsScene file, the scene objects are created and in their constructors, the data for each is made. After this, the render loop is started.

3.2. Terrain

To create the initial terrain, a 2D height map (an array containing some random values) was used. This height map was used to assign the Y (or height) position of 3D vertices. These vertices where connected together and this created the terrain. To create the 3D vertices, we can use a simple double for loop.

The initial double for loops went from 0 to 2048 in the X and Z. This created a flat grid of vertices spanning outwards into the scene. A value from the 2D height map, created earlier, was assigned to the height position of the current vertex. So a vertex looked like: x, heightmapvalue, z.

This created very bumpy terrain, as the height values used where just random. There had to be an alternative approach. As mentioned in the background reading section, the perlin noise algorithm can be used to create natural looking terrain. To summarize, this algorithm generates smooth random numbers, which can then be assigned to the Y position of the vertices. This perlin library [2] was used to generate the noise values and the result can be seen in the below image [43].



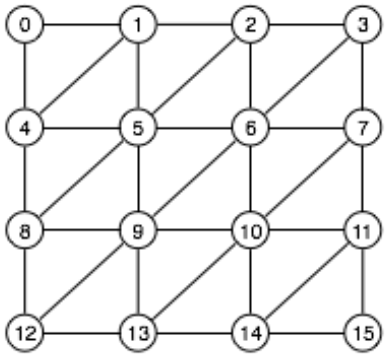
The terrain here is much more natural because of the smooth random numbers generated by the noise algorithm. The terrain looks like some smooth rolling hills.

Later on in the project, a texture was mapped over the terrain. The texturing algorithm used is described here [21].

You will notice that there is a cliff in the distance. This was created by adding +5 to the height values after a certain Z position in the for loops was reached.

However, if you remember from the background reading section, all the above information has described is what points we have sent to WebGL. But WebGL also needs to know how to connect those points. This is where the GL_TRIANGLE_STRIP comes in handy.

The GL_TRIANGLE_STRIP will connect triangles between points, given a set of indices to use. The image to the right shows the points being connected into triangles [56]. The code to generate the indices has been taken from here [9]. This code simply generates the indices for the vertices created above. The indices are then stored in a buffer and sent onto the GPU as usual. Then at render time, that index buffer (also known as the element buffer) is bound and used.



The original terrain draw call looked this like:

```
/*
Parameters:
    The drawing mode we want to use
    Number of indices
    Type
    The indices buffer (or element buffer)
*/
gl.drawElements(
    gl.TRIANGLE_STRIP,
    terrainVertices.length / 3 * 2,
    gl.UNSIGNED_INT,
    elementsBuffer
);
```

You will notice in the above draw call that terrainVertices length is divided by 3. The draw call requires the total number of indices. As the terrainVertices array contains the individual x, y, z values, it is divided by 3 to give the total number of vertices.

When generating a set of indices, there will always be twice as many indices as there are vertices. This is why for the number of indices parameter in the above code, the number of terrain vertices is multiplied by 2.

The colours for each vertex are also defined in an array, however at this point they were just assigned random red, green, blue values. As usual, they are put into a buffer, sent to the GPU and bound at render time.

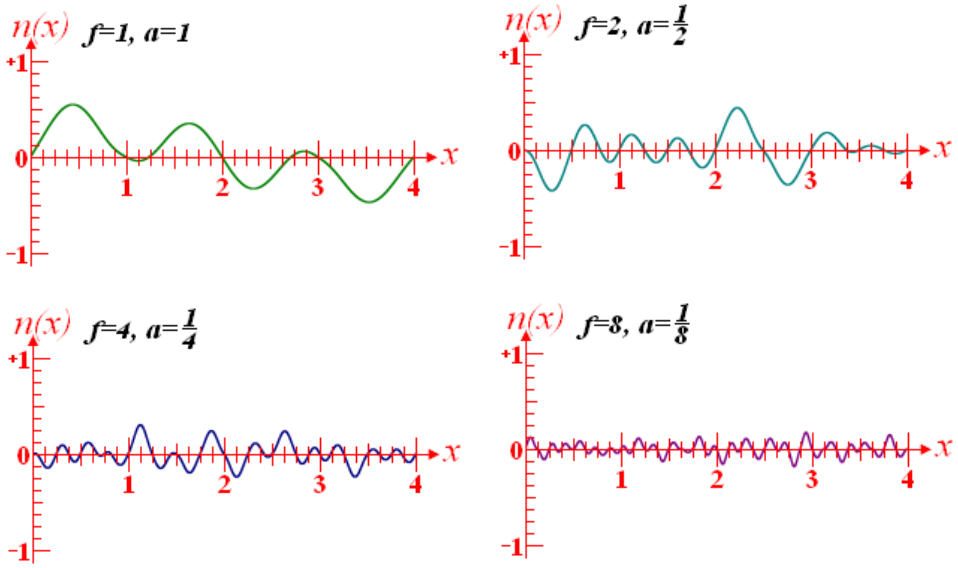
3.3. Terrain rendering efficiency

This initial terrain was good, especially since it was done before the start of the project. However later on in the project, the terrain was rewritten. A stacked noise technique was used to create the height map, instead of using single noise values for each height like before. Stacked noise is essentially taking multiple octaves of perlin noise and adding them together. This generates numbers which can give much more detail to the terrain.

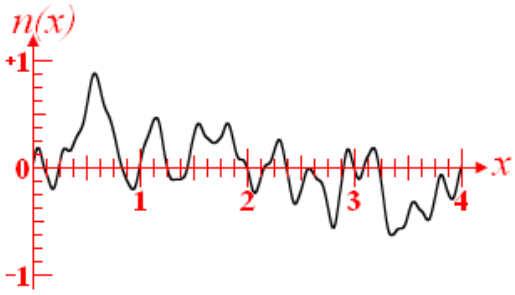
A single perlin noise octave is a group of noise values chosen over time. For every new perlin noise octave, we half the maximum possible value and double the frequency in which we pick values. The maximum value is the amplitude and the frequency at which a value is chosen, is just called the frequency. The code for this is in the terrain class. This code will generate numOctaves amount of noise values, adding them all together gives us the final noise value for a vertex height.

```
    function stackNoise(x, y, numOctaves){
        var v = 0;
        var amplitude = 1;
        var frequency = 1;
        var noiseTotal = 0;

        for(var i=0; i<numOctaves; i++){
            // https://github.com/josephg/noisejs
            v += noise.perlin2(x * amplitude, y * amplitude, x * amplitude)
* frequency;
            noiseTotal += frequency;
            amplitude *= 0.5;
            frequency *= 2.0;
        }

        return v / noiseTotal;
    }
```

An easier way of looking at it can be seen below (read graphs left to right). As explained above, in each new octave we half the amplitude and double the frequency.



Take the above graphs, and imagine them added together. After all of the noise octaves have been added together [62], we get a much more natural looking graph (as seen in the right image). This is a similar result used for the terrain code.



The image on the left is the terrain with singular perlin noise height values. The image on the right is the terrain with stacked perlin noise height values. You can see much the terrain has much greater detail in the image on the right [22].

It soon came apparent that the FPS on the scene was terrible. The FPS was averaging 20-30 frames. How? The terrain size was only 2048.

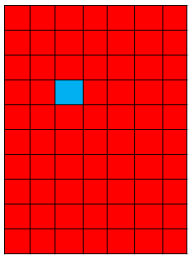
Well actually the terrain size was 2048 by 2048, meaning it had over 4 million vertices. You may be thinking, but surely WebGL would just clip off the vertices that are not visible and only a few thousand vertices will get rendered? And the answer is you are right. However, the actual rendering of the vertices is not the issue. The issue is that the vertex shader has to process 4 million vertices per frame, then cull 99% of them as they are not visible.

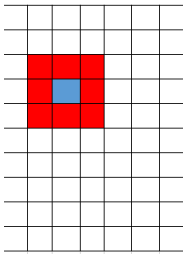
Vertices that are far away from the user will definitely not get rendered, yet the vertex shader was still being asked to process them anyway (as the vertex shader was being passed the entire terrain data every frame). To improve this, there needs to be a way to cut down on the number of vertices sent to the vertex shader in the first place. This would mean the vertex shader would process much fewer vertices, leading to an increase in FPS.

The solution involved splitting the huge terrain section into smaller sub-sections, or quadrants. Then, the user's position would be calculated to see where they were in the world. From this, the surrounding 3x3 terrain quadrants of the user would get passed to the vertex shader to be processed.

This is a visual representation of the clipping that is being performed on the CPU, before the vertices are sent to the vertex shader. The below grids represent a bird's eye view of the terrain. The cells represent the terrain quadrants. The quadrants in red represent the quadrants that are processed. The blue quadrant is the quadrant in which the user is in, that quadrants vertices also get processed.

Before the CPU clipping, all quadrants vertices where processed (image on the left). After the CPU clipping was implemented, only 3x3 quadrants where processed (image on right.)





This saved a total of 4,046,848 vertices being processed every frame. Previously, when rendering the 2048x2048 section, 4,194,304 vertices where being processed. But now, rendering only a 3x3 section of the terrain means only 147,456 vertices are being processed (128 rows * 128 columns * 9 sub sections).

The new terrain is generated in quadrants, rather than all at once like before. There is a buildAllTerrainData function in the terrain class that does this. This function loops over the number of quadrants we should have, creating the data for each of them. The data created for each quadrant includes its: vertices, indices, UVs and normals. After a quadrants data is created, the data is stored in the appropriate vertex buffer objects.

After implementing this, a new texture was used to map over the terrain, this is the currently used terrain texture. It is a simple sand texture, however it looks very bumpy. This, along with the stacked perlin noise gives even more detail to the terrain surface.

3.4. Terrain VAOs

Up until now, we have used vertex buffer objects (VBOs) to store our data. Then at render time, we select these VBOs before issuing the draw call. However, as the terrain and scene has gotten more complex, more and more data has been associated with rendering. A render call now requires binding: vertices, indices, texture coordinates, normals and the actual texture being used. The old terrain rendering code can be found here [7].

With using vertex array objects (VAOs), we can save the number of binding calls at render time.

The example usage of a VAO at setup time, in pseudo code, is as follows:

```
/*
Create VAO
Bind VAO
Create quadrants data
Put quadrants data in VBOs (which automatically get bound to the VAO)
Unbind VAO
*/
```

Now at render time, we no longer have to bind all VBOs individually, we can just bind one VAO, which contains the VBOs. The new terrain rendering code can be found here [8]. This not only makes the code cleaner by having less buffer binds, but it also more efficient as there as less calls made to the WebGL API.

As mentioned in the background reading section, a VBO just contains data.

The below image shows the layout of a VAO. The attribute lists are just slots in the VAO. These slots store our VBOs containing our actual data. We can include whatever buffers we want here, they do not have to be the specified: position, texture coordinate, normals or colours as defined below.

Attribute Lists		
0	(VBO) Vertex position buffer	→ 0.0, 1.0, 0.5
1	(VBO) Vertex UV coordinate buffer	→ 0.0, 1.0, 0.5
2	(VBO) Vertex normal buffer	→ 0.0, 1.0, 0.5
3	(VBO) Vertex colours buffer	→ 0.0, 0.0, 1.0

3.5. Camera

The initial multicolored terrain was actually implemented before the camera class. It would be difficult to know if a fly around camera was working, if there was nothing moving on the screen.

The important parts of the camera class are the updateCamera and assignCameraQuadrant functions. The updateCamera function checks if the user is holding down a key, if they are, the camera position is updated in the appropriate X and Z directions. Only the X and Z camera position values are updated here as otherwise, the camera would be able to fly up in the Y. Instead, going up and down was implemented by the use of the R (rise) and F (fall) keys. The updateCamera function then recreates the camera and view matrices with the new values.

At the end of the updateCamera function, a call to its updateAttributesAndUniforms function is made. This function loads in the matrices and global variables into the shaders ready for rendering.

The assignCameraQuadrant function was an addition made when the terrain was rewritten with its new rendering efficiency. The quadrant that the user is currently in needs to be worked out so we can render the 3x3 quadrant cells around the user. The function loops over the positions of each quadrant checking if the user is inside of it. When it eventually finds the users quadrant, it is set in the cameras quadrant variable.

3.6. Fog

The techniques used for the fog generation where learnt from [63]. The fog image used below is from [57].

Fog was a very simple addition to the scene. The further the objects distance is away from the camera, the more that objects texture gets blended with the sky colour. So if an object is close to the camera, its texture is not blended with the sky colour at all. This is simply done by calculating the vector from the current vertex to the camera position. If we were looking at the scene from above, this is how the objects textures would get blended.

The image obviously is not from the scene, but the technique is the same.

The visibility decreases exponentially with the vertices distance away from the camera. If the visibility of a vertex is 1, then it is fully visible and not blended with the sky colour. If it is 0, then the vertex is completed blended with the sky colour and is not visible. The gradient determines how quickly the visibility decreases with distance. These visibility, gradient, density and distance variables are all calculated and defined in the vertex shader.

The position of the vertex relative to the camera is calculated by multiplying the viewMatrix, by the worldPosition of the vertex. Now to get the distance, we just take

the length of the vector. To get the length of the vector, we can just use the GLSL length function to do it for us.

Now the distance from the current vertex to the camera has been calculated, we can use it to set the vertex visibility value. We use an exponent function here to achieve a realistic effect.

This visibility value then gets passed from the vertex shader to the fragment shader. Finally, the fragments colour is set to the GLSL function mix, of the visibility and the skyColour, determining how foggy that fragment is.

3.7. Rocks

To generate the rocks, an existing library could have been used. There is a procedural rock generator library for WebGL. However simply loading in rocks from a library is not very technical.

The first attempt at the rock generation was done by taking code to generate a sphere and indenting its vertices. So now instead of a sphere geometry, it became slightly bumpier and rock like.

However after implementing this, it became apparent that the FPS of the scene was dropping. Each of these spherical rocks had 2000 vertices.

The next attempt simply involved importing rock OBJs. This cut down the vertex count to around 300 per rock, but this was still way too many. The scene needed hundreds of rocks within a small area and having 300 vertices per rock was just not good enough.

To achieve the finished rocks, a technique called instanced rendering was used. This technique was learnt from Jamie Kings OpenGL instancing tutorial [58]. This allows an object to be rendered multiple times. The GPU is very efficient when drawing lots of objects at once, rather than only drawing objects one at a time.

In this example, the GPU has to wait for each object to be sent to it through individual draw calls before displaying them. Imagine we had some rocks in an array and we looped over all of them, drawing them one by one (not syntactically correct code).

```
for(var i=0; i<numRocks; i++){
    gl.draw(rocks[i]);
}
```

Each rock would have to be sent through the graphics pipeline individually, eventually ending up in the colour buffer. After all rocks had been drawn to the colour buffer, one by one, the frame buffer would be updated.

A much more efficient method is to pass the GPU all of our data at once, then just issue a single draw call. This single instanced draw call tells the GPU to render the object multiple times. We no longer have to keep issuing separate draw calls for every new rock we want to render.

This is done by passing the GPU a set of rock vertices and a set of translation matrices to apply to instances of the rock. We cannot just pass one set of rock vertices and have

the GPU magically draw the rocks in different places. So we need to build full transforms matrices to send to the GPU as well. The GPU will apply these matrices to the one set of rock vertices and then draw the rock vertices in their translated positions.

We need to send these full transforms matrices into the vertex shader. However there is no way to send a matrix4 attribute (varying per instance) into the shader. We can only send in a uniform matrix4 (the same over the entire draw call). We cannot use a uniform matrix4 because we need the translations to be different for each rock. So instead, we have to send each column of the matrix individually and then build the matrix in the vertex shader. We are allowed to send in a vec4 attribute, but not a matrix4 attribute. A matrix4 is simply composed of 4 vec4s.

In the RockGenerator class, the generateMatricesForTransformRow functions create each column of the full transforms matrix. So at setup time, we create thousands of these full transforms columns and send them onto the GPU. At render time, we have to bind these full transforms columns and finally issue the instanced draw call.

The rock rendering code actually uses 9 draw calls as if you remember, we render 9 terrain quadrants. Each quadrant has its own set of rocks. This was just a fast easy way of doing it and this could definitely be cut down to 1 draw call in the future. Anyway, 9 instanced draw calls are much better than 5000 regular draw calls.

After the draw call has been issued, the vertex shader constructs the matrix from the 4 columns we passed in. The 4 columns get passed into the shader variables defined below. The row/column variable names are confusing here, GLSL actually uses a different row/column matrix order than used in the JavaScript code.

```
// Instancing
'uniform bool useInstancing;',
'attribute vec4 instanceMatrixRow0;',
'attribute vec4 instanceMatrixRow1;',
'attribute vec4 instanceMatrixRow2;',
'attribute vec4 instanceMatrixRow3;',
```

We then construct the final matrix4 in the shader, from these 4 attribute vec4s we passed in.

```
'fullInstanceTransform = mat4(
    instanceMatrixRow0,
    instanceMatrixRow1,
    instanceMatrixRow2,
    instanceMatrixRow3
);',
```

This final matrix is then applied to the world position of the rock instance.

```
'vec4 worldPostion = model * vec4(position, 1.0) * fullInstanceTransform;',
```

The vertex shader applies all of the matrix columns we sent to it, to as many rocks we told WebGL to draw.

The below image [31] shows the rock instancing processing/rendering 200,000 rocks at 20 FPS. The reason why it can render so many rocks is firstly due to the instancing described above, but also due to the very low vertex count of the rock model used.

3.8. Skybox

Unlike the fog which only required a minor addition to the main shaders, the shader code needed for the skybox was quite different. For this reason, a new vertex and fragment shader were written. These are created and linked in the SkyboxProgram class. Before rendering the skybox, the skybox program is selected as the program to use. We then render the skybox and switch back to the MainProgram afterwards.

A skybox, as its name suggests, is just a box. The box has textured sides and is placed on the outside of the world. The skybox always stays at the edges of the world, so the user can never reach it.

To texture this skybox cube, we would just map images to each side of the cube individually. However there is a better approach and that is by using a CubeMap texture. This singular texture is composed of 6 other textures, representing all the sides of the cube. Now instead of binding one side of the cubes textures, then rendering that side and repeating 5 times. We can instead just bind the cube map texture and render all the sides once, with one draw call. This saves 5 draw calls and texture swaps per frame.

The skybox vertex shader simply sets the position of the skybox in the world. The fragment shader samples the 2 cube map textures (day and night) and blends between them depending on the time of the in-game day.

The skybox.js file has an updateDay function. This rotates the skybox slowly and because of this, the suns position also needs to be updated. Otherwise the specular highlights on the water would be in a different direction than the sun.

This function also modifies a time variable and different things happen depending on the value of it. For example, if it is between 6pm and midnight, then the skybox starts blending to the night texture. This happens by updating the blendFactor variable used in the shader. The blendFactor says how much to mix the day and night skybox textures.

Not only does this function cause the skybox texture to change, but it also fades the specular highlights on the water. It does not make sense for the water to have specular highlights when there is no sun. The fog colour and map boundaries also change based on the time of day.

3.9. Old Water

Water was added to the scene in sprint 1, however it was just a quick spike solution with no research of how to do it properly. A very similar method to the terrain generation was used to create it. A grid of vertices was made, however instead of setting the heights of the vertices once, they were changed every frame. The result was flowing water. However, the movement created by this was awkward. The vertices where relatively spread out so you could see the different rows and columns in the water. This did not yield a realistic effect as you can see on the original water below:

The worst thing about this water was that every single vertex was being updated on the CPU. The CPU is much worse than the GPU for manipulating vertices as it is not very parallel. Therefore the CPU takes longer to manipulate the vertices, resulting in fewer frames per second.

3.10. New Water

At the start of sprint 6 the main features of the scene where done (terrain, rocks and the skybox), so the scene was ready for some realistic water. Knowledge was gained of how to implement this water from ThinMatrixs water tutorials [65]. However his project was implemented in Java and OpenGL. These tutorials obviously had to be ported to WebGL and JavaScript, which was not an easy task. This was mainly due to the syntax changes and the out of date GLSL version used by WebGL.

The water in the scene is simply a flat square. Everything that is happening is done via texture effects. This instantly makes it much more efficient than the previous water as no vertex data is being changed. Like with the skybox, the water has its own set of shaders which are selected at render time. All of the texture effects seen on the water are done within the water shaders.

The water texture is made by mixing the reflection and refraction textures of the scene, then mapping them onto the water's surface. To create these textures, we need to use frame buffer objects, or FBOs.

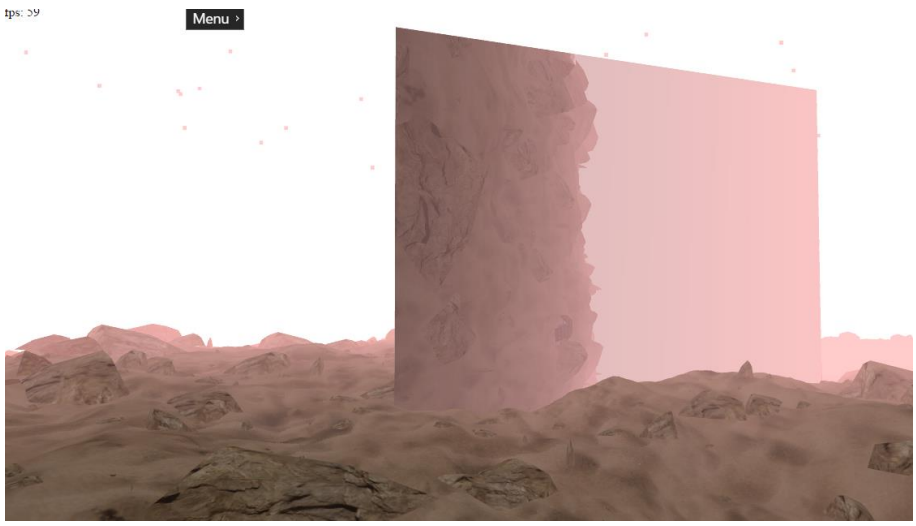
You may remember the frame buffer. This is the 2D array of pixels displayed on the screen. To create the reflection/refraction textures we need to render the scene to off screen textures, or frame buffers. A texture is simply a 2D array of pixels (a frame buffer!). So, we create a FBO, bind it as the one we want to render to, render the scene as usual (which renders it to the off screen FBO, or texture). Then, to render

the rest of the scene as usual, we have to switch back to the default frame buffer and call the usual render functions. A texture and a frame buffer are essentially the same thing. The WaterFrameBuffers class creates these off screen reflection and refraction frame buffers that we render to.

This means to create the water effect, we need to render the scene 3 times per frame. We first render the scene to the reflection texture, then once more to the refraction texture and then finally to the screen. An important thing to note about the water is that it is not transparent. Everything below the water actually gets rendered onto the texture above.

3.10.1. Rendering to a texture

To get the water effect working, we first have to render the scene to a texture. The texture then gets rendered onto something within our scene. The below image [24] shows my scene being rendered to a texture. That texture is then being rendered onto a quad in the actual scene, slightly confusing. Ignore that fact that the texture is sideways, it does not matter for now.



This is the code to render the scene to the refraction texture, or refraction frame buffer. We first bind the refraction FBO that we made earlier. We then set the clip plane which will be discussed further on. After that, we clear the refraction texture as we want it to constantly update, not just be a snapshot of the waters refraction at one point. We set the size of the viewport, which says how much detail the texture will have. Now we can finally render our scene to this off screen water frame buffer object.

```
    gl.bindFramebuffer(gl.FRAMEBUFFER,
waterFramebuffers.get.refractionFrameBuffer);
    // Want to render everything under the water
    clipPlane = [0, -1, 0, -WATER_HEIGHT];
    gl.clearColor(0.8, 0.8, 0.8, 0.7);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT); // clear texture
    gl.viewport(0, 0, 512, 512);
        // render the scene to the currently bound FBO (or texture)
        terrain.render();
```

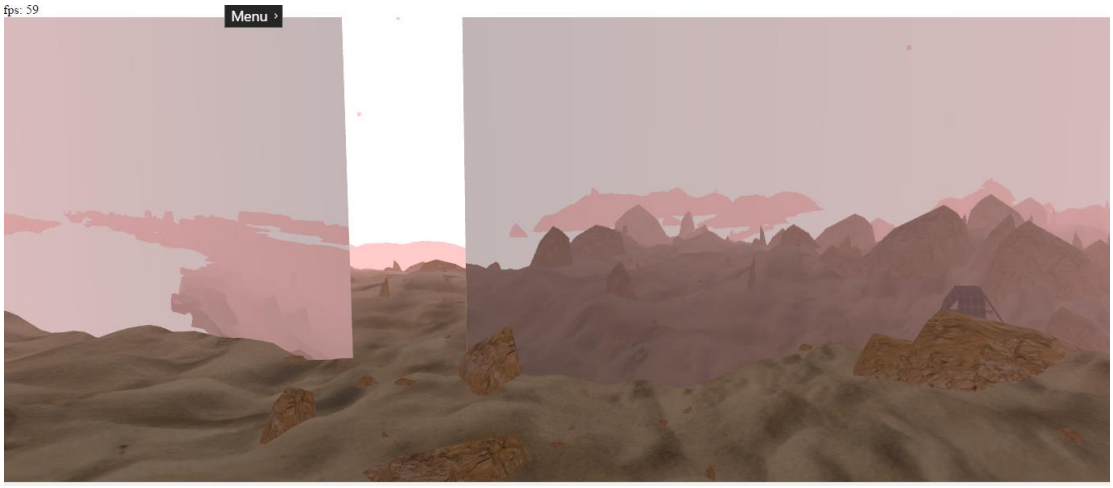
```
        rockGenerator.renderInstancedRocks();
        skybox.render(viewMatrix, projectionMatrix);
    // Unbinds frame buffer, reset viewport
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
    gl.viewport(0, 0, window.innerWidth, window.innerHeight);
```

Later on in the render pass, we render this texture onto an object in the scene. To avoid an infinite paradox, we do not want to render the water quad within the render passes to the reflection and refraction textures. This is why the water render function is not being called in the above code, but only when we render the scene normally.

If we rendered the scene to the water textures as usual, then it would not line up with the world and it would not look realistic. Instead, we need to render different parts of the scene at different angles. This creates the reflection and refraction texture effects.

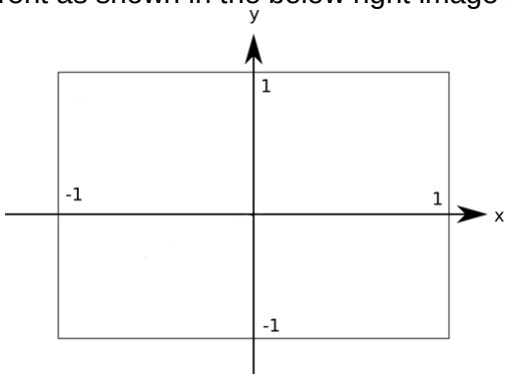
The vec4 clipPlane is used to cut off everything above or below a certain position in the scene. In the case of a refraction texture, everything above the water's surface should be clipped off. This is because a refraction should show everything below the water's surface. You can see this happening in left quad in the image below. Everything above 0 (the waters height) is being clipped off.

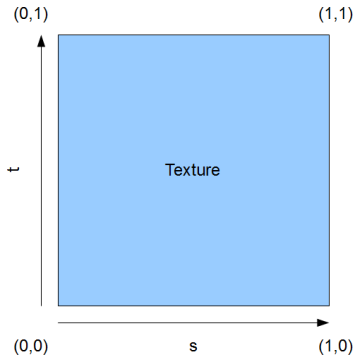
The code to render the reflection texture is slightly different. We need to render the scene as if we were below the water's surface. To do this, we move the camera down by its distance above the water, invert its pitch to look up, render the scene to the texture and then finally revert everything back. The reflection texture is rendered to the quad on the right. Only things above a height of 0 (the water's height) get rendered onto the texture.



To ensure the reflection and refraction textures line up properly on the water's surface, we need to use projective texture mapping. In the right quad in the above image [25], the reflection texture is actually the wrong way around. The reflection needs to be inversed, as after all, it is a reflection. We simply need to inverse the Y coordinates in the shader. To find out what point on the reflection and refraction textures we should use for the current water fragment, we can simply use the screen space coordinates of the water fragment. All we have to do is sample the reflection and refraction textures at this exact screen space coordinate. Again, this will only work if the reflection texture is flipped upside down like it is meant to be.

In the vertex shader we have the clip space (-1 to +1) position of the vertices in the x, y and z axes. To find the screen space coordinate of a fragment in the shader, we just need to do the perspective divide of these coordinates This is just each of the coordinates divided by its w component. We need to output the clip space coordinates of the vertex from the vertex shader, to the fragment shader. The conversion from clip space to normalize device coordinates (NDC) happens here. We now have any point on the water's surface in the -1 to +1 NDC coordinate system shown in the below left image [59]. However the texture coordinate system is different as shown in the below right image [60].





The conversion between these two coordinate systems is simple, we just divide by 2 then add 0.5. This is done in the water fragment shader on line 36.

```
'vec2 ndc = (clipSpace.xy / clipSpace.w) / 2.0 + 0.5;',
```

We can now use these NDCs to sample the reflection and refraction textures.

Now that the refraction and reflection texture coordinates are calculated properly, we can mix them together and map them onto the water quad. This is done with the GLSL mix function in the water fragment shader.

3.10.2. Water ripples

The water ripples, or distortion, was added by using a DuDv map, also known as a displacement map. A DuDv map is a texture containing lots of green and red colour values. All we have to do is distort the waters texture coordinates (as we calculated them above) by adding these red and green values from the DuDv map. We just need to map the DuDv texture onto the water quad like any other texture, then sample it for each point.

However these red and green values in the DuDv map are all colours. This means the values are always greater than 0, as we cannot have a negative colour value. If we used these as they were, then the distortion on the water would always go in the same direction. To get a realistic effect, the fragments texture coordinates need to be distorted in both the positive and negative directions. To do this, we simply map the colour values from -1 to +1. This can be done by multiplying the colour values by 2, then subtracting 1. This conversion is done the fragment shader:

```
‘texture2D(dudvMapSampler, distortedTexCoords).rg * 2.0 - 1.0);’,
```

If we sampled this DuDv map every frame, then we would get the same distortion for each fragment, after all – the DuDv map is not changing, nor are the waters texture coordinates. So to make the water look likes it is moving we need some sort of offset value to add to the texture coordinates.

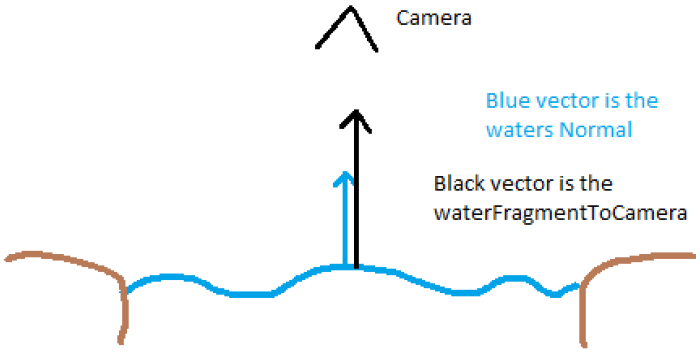
This is done by the moveFactor variable, located in the WaterSystem class. The waterPrograms updateWaterAttributesAndUniforms function updates this variable via the getters. It is incremented by a modified Date.now() call. Afterwards, it gets passed into the shaders with the rest of the variables and this gives the illusion that the water is moving. This offset is then added to the distortion which is then applied to the texture coordinates.

Since we are incrementing the waters texture coordinates, some of the coordinates go below 0 and some go above 1. This causes the texture to wrap around to the opposite side causing a distortion on the edges. The GLSL function clamp is used to ensure the values are kept between 0 and 1. This happens on the lines 49, 52 and 53 of the water fragment shader.

3.10.3. Fresnel effect

The Fresnel effect has been added to the water's surface. This changes the waters reflectivity based on the cameras viewing angle. If the water is viewed from above, then its surface should not be very reflective. However if viewed from eye level, the water should be very reflective.

We have the cameras position and the waters normal. If we calculate the vector from the water fragment to the camera, then the closer the water's normal and the waterVectorToCamera are, the less reflective the water should be. In the example below, the water normal vector and the waterFragmentToCamera vector are pointing in the same direction. Therefore the water's surface should not be reflective. If the camera was near the brown lines (ground), then the water's surface would be very reflective. This is because the waters normal and the waterFragmentToCamera vector would be far apart.

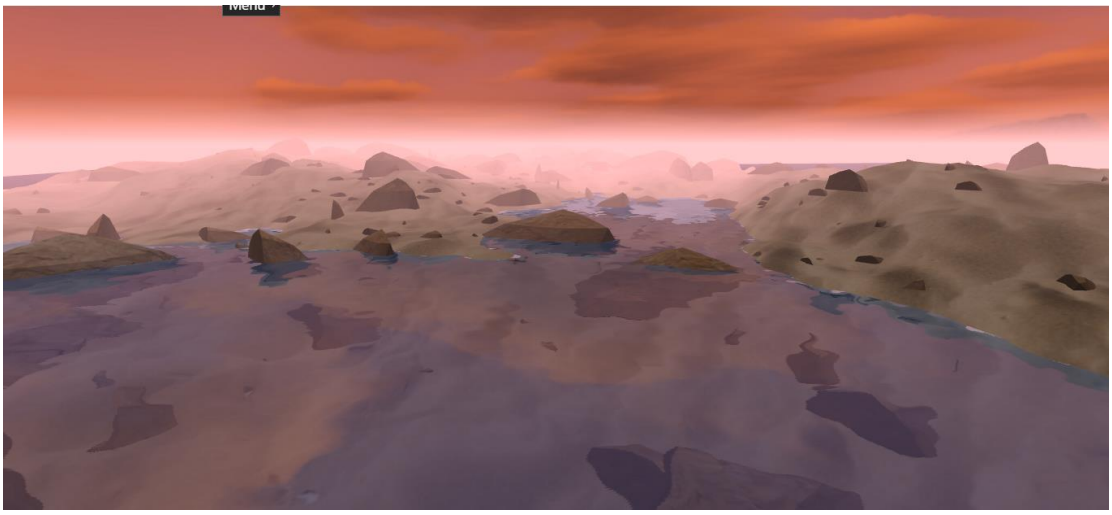


This calculation can be done with the dot product of two vectors. If the dot product of the vectors returns 1, then they are pointing in the same direction. If it returns 0, then they are in opposite directions. This value is used to set the fragments reflectivity in the shader.

```
// First normalize toCameraVector, as dot product requires it to be unit
    'vec3 viewVector = normalize(toCameraVector);',
    'float refractiveFactor = dot(viewVector, vec3(0.0, 1.0, 0.0));',
```

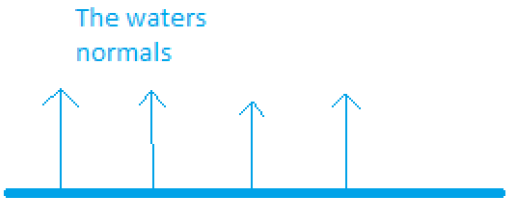
We then mix the reflective and refractive water textures with this refractiveFactor value, to give the fragment colour.

```
'gl_FragColor = mix(reflectColour, refractColour, refractiveFactor);',
```

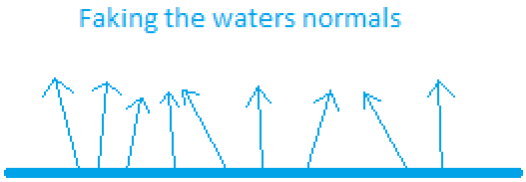


3.10.4. Faking the normals

To calculate the lighting for the water, we could use the current normals. However this would not yield a realistic result. This is because every effect applied to the water is within the texture effects. The waters vertices are never actually being changed and therefore, the waters normals are not changing either. The below image shows the actual waters normals.

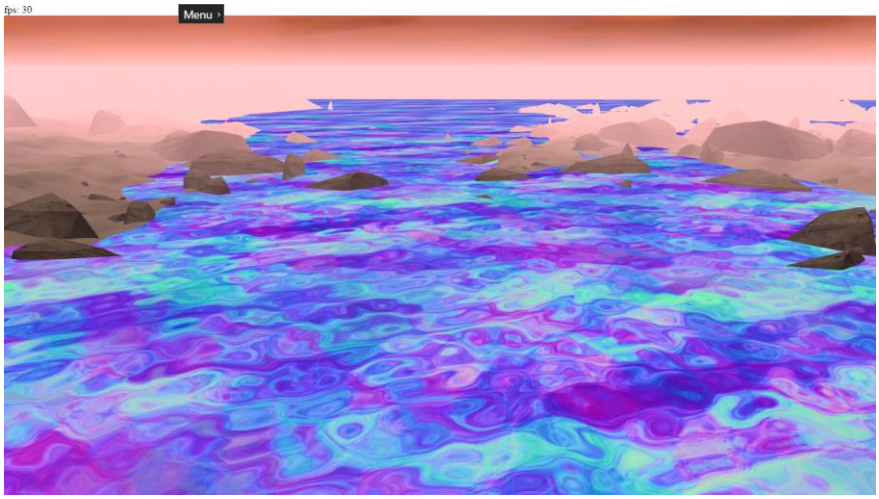


So to calculate realistic lighting on the water, we need to fake the normals and change them over time. The below image shows us faking the normals on the water.



We can use a normal map to generate these normals on the water's surface. This is a similar idea to using a DuDv map to indicate the distortion on the water's surface. We can sample the normal map texture in the fragment shader to return an R, G, B colour value for a given fragment. We can then convert that colour value into a normal vector for the same fragment. The blue colour of the fragment represents the up axis and should therefore correspond to the Y component of the normal vector, hence why it is swapped around in the shader.

Since we are sampling colour values from the texture, we run into a similar problem as we did with the distortion. A colour value can never be negative, meaning our normal would always be pointing in a positive direction. Having all of the normals as positive would not be very realistic. So to fix this, we just need to map the normal values from -1 to +1. This calculation is not done of the Y component of the normal as we always want the normal to point upwards. If we overlay the normal map onto the water's surface, then we can see how the normal for a fragment would get calculated.



First, the normal map gets sampled for a given fragment, returning the colour at that point on the texture (R, G, B value). That colour then gets used to calculate the normal as previously mentioned. The below code shows all of the above happening. The Coords variable is the normalMapCoordinates, the variable name was just shortened.

```
// Sample normal map, sampling at distortedTexCoords, as used for dudv
'vec4 Coords = texture2D(normalMapSampler, distortedTexCoords);',
// Extract the normal, from the normal map colour, does the conversion
'vec3 normal = vec3(Coords.r * 2.0 - 1.0, Coords.b, Coords.g * 2.0-1.0);',
// normalize to make unit vector
'normal = normalize(normal);',
```

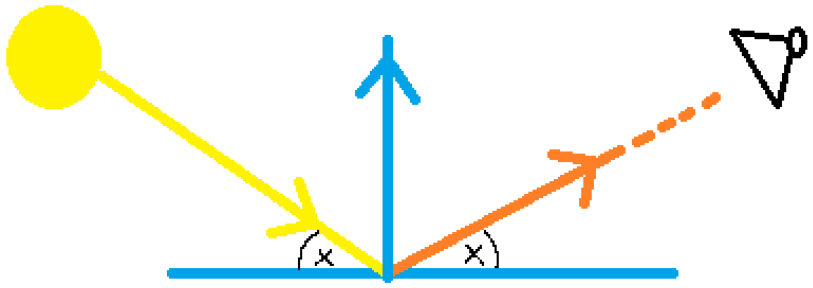
3.10.5. Specular highlights

Knowledge of specular lighting has been gained from here [61].

The vector from the sun to the water fragment is calculated, then a reflection vector is calculated as the light reflects off the water's surface. The reflected angle is exactly the same as the incoming angle. We also use the normal as calculated in the previous section. The brightness of the fragment depends on its reflectivity variable and also the position of the camera. If the reflected light vector is going directly into the camera then that fragment is lit very brightly.

The yellow vector is the vector from the sun to the water fragment. The orange vector is the reflected light vector, notice how it has the reversed angle to the incoming vector. The blue vector is the waters normal. Since the reflected light vector and the water fragment to camera vector (not shown here) are pointing in the same direction, the reflected light is going into the camera. This means the waters fragment would

have a bright specular highlight. If the water fragment to camera vector was facing away from the reflected vector, no light would be going into the camera. Therefore the fragment would not have any specular highlights.



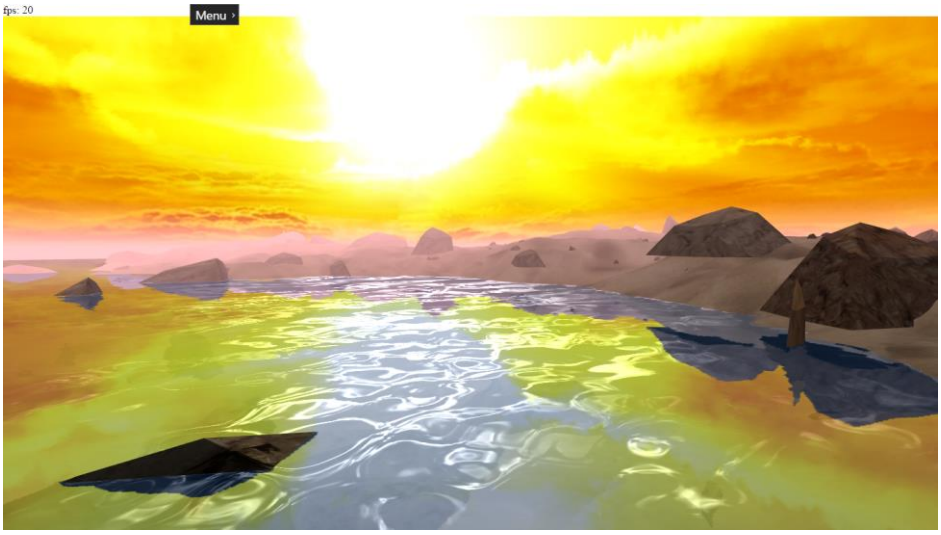
The view vector in the below code is the water fragment to camera vector (not shown in above diagram).

```
'vec3 reflectedLight = reflect(normalize(fromLightToWaterVec), normal);',
'float specular = max(dot(reflectedLight, viewVector), 0.0);',
'specular = pow(specular, shineDamper);',
'vec3 specularHighlights = lightColour * specular * reflectivity;',
```

Finally, we add a blue tint to the waters fragment to make it look more realistic. This is the vec4(0.0, 0.3, 0.7, 1.0) in the below line. We also add on the lighting value that we just calculated.

```
'gl_FragColor = mix(gl_FragColor, vec4(0.0, 0.3, 0.7, 1.0), 0.2) +
vec4(specularHighlights, 0.0);',
```

The below image shows the finished water.



4. Documentation

Along with the other documentation, full pages of docs have been generated. The YUIDoc tool [68] was used to generate documentation from the code comments. This tool generates JavaDoc style HTML pages from formatted JavaScript comments. An example yuidoc comment is as follows:

```
/**
Explanation of what the function does and why it is needed

@method functionName
@private
@param paramName {Bool} if true, do something
*/
function functionName(paramName){

}
```

This in turn will be added to the HTML generated by the yuidoc tool.

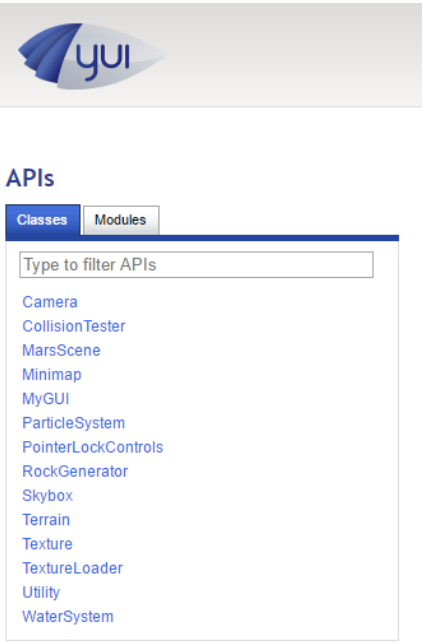
YUIDoc comments are declared with /** your content here */. Rather than a regular comment /* content */. Notice the extra * needed for YUIDoc to recognize the comment. The main YUIDoc variables that have been used are:

- @method
- @public/private
- @param

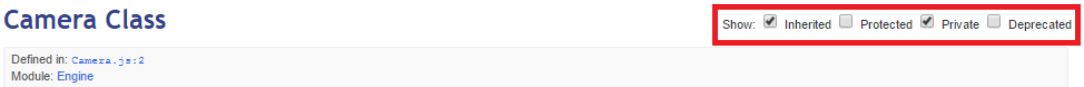
All files have accompanying yuidoc. To get an overview of the classes and details of the functions, go to:

Major Project/documentation/docs/out/index.html

You can then click through the documentation pages in the APIs section on the left of the page. The image on the right shows the API panel you can navigate.

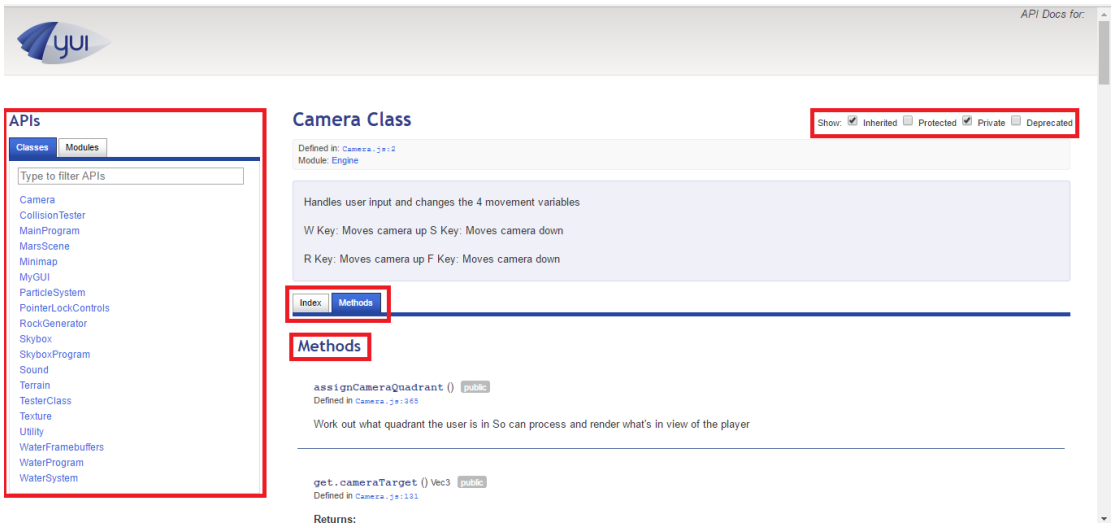


To see private methods as well, click the "Private" checkbox in the top right options panel, as shown in the below image.



The "index" tab is not working properly, when you click a function it does not re-direct properly. So just go on the "Methods" panel to view them. These two panels can be seen below, outlined in red rectangles just right of the API panel. Notice how the private methods in the top right are selected to show as well.

Example, docs of the Camera class:



5. Testing

5.1. Overall Approach to Testing

Most of the testing was done via JavaScript unit tests and through the use of the Chrome developer console. There was also testing on separate operating systems, revealing the project is most stable on Firefox.

The options for unit testing the project can be seen here [27]. The overall approach was to use a Tester class containing generic testing functions. Specific testing functions are in their respective classes. This minimizes test code duplication by having all generic functions in one Tester class. However not everything can be put into this generic testing class, so the testing code is split between the implementation file and the generic testing file. All unit tests are at the bottom of their respective files.

5.2. Testing the Game

Up until sprint 5, the project was still a game. Testing the game features were relatively simple. Acceptance tests where written in a test table to see if the user could complete various tasks. This type of testing was perfectly valid for the project as after all, a user would be playing it.

After sprint 5, the game aspects where stripped out and now most of the test table tests where irrelevant. The tests that were still valid have been kept, but it lost over half of its content.

The old test table can be found at [28]. After the game aspects where removed, the test table was updated to show this. The new test table can be found here [29].

5.3. Things that cannot be tested

Testing rendering output is infeasible (testing what actually appears on the screen).

One way to actually test the rendering output is by writing an implementation of the WebGL API our self. This would show all the calls made to the graphics hardware and from that, we could test if they are correct. This is obviously not feasible for this project.

Testing rendering output is essentially testing the WebGL implementation itself, rather than the application. This is the same as writing a C function, and testing that the compiler converts it into the correct assembly language calls. This is the compilers job not ours, likewise the rendered graphics are WebGLs responsibility. Testing the rendering output is simply out of the scope of the project.

5.4. Things that can be tested

- The data we give to WebGL/shaders to get displayed on the screen.
- Checking buffers were created properly
- Changing the UI does what it should, for example changing water strength
- User colliding with terrain/map boundaries
- Vertex creation, ensuring that the correct amount of vertices are made
- Cross browser testing
- Cross operating system testing
- The users browser, check it can run at least WebGL 1.0
- WebGL console errors
- Memory usage and see how scene properties affect this
- FPS tests and see how scene properties affect this

5.5. Automated Testing

The unit tests run every time someone starts the scene. We can enable/disable testing of the scene via changing the useTests variable defined in the MarsScene.js file. However the tests that run are not expensive and therefore this is not necessary.

5.6. Unit Tests

As discussed in the options for unit testing section [27], the unit tests are in the regular JavaScript files. However they are split into functions and are kept at the bottom of the file to ensure the code is readable.

The below code snippet is an example taken from my TesterClass. This class holds all generic testing functions that are used throughout the project.

```
/**
Tests if location is a WebGLUniformLocation

@method test_isWebGLUniformLocation
@public
@param name {string} the name of the attribute to test, so we can print an
error
@param location {buffer} the location value to test
```

```
*/
this.test_isWebGLUniformLocation = function(name, location){
    if(!location instanceof WebGLUniformLocation){
        // error message
    }
}
```

However not all testing functions are generic and therefore, we cannot place all testing functions within this TesterClass. Some class-specific testing functions have to be written and they are placed in their respective classes. The below code snippet is a testing function found in the Terrain class. It makes sure the terrainVAOs array was filled with WebGLVertexArrayObjectOES (VAOs) properly. VAOs are only used in this terrain class, therefore it makes sense that the testing function is also kept within with class.

```
/**
@method test_terrainVAOs
@public
*/
this.test_terrainVAOs = function(){
    for(var i=0; i<numberQuadrantRows * numberQuadrantColumns; i++){
        if(!vao_ext.isVertexArrayOES(terrainVAOs[i])){
            // error message
        }
    }
}
```

These types of unit testing ensures that all of the scene data is created properly. As mentioned at the beginning of this section, we can only test the data that is passed to WebGL.

The terrain quadrant clipping has been tested and documented here [32].

5.7. User Interface Testing

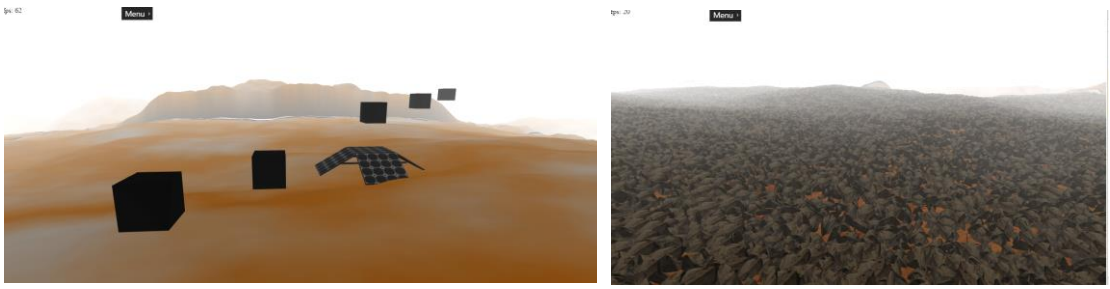
The user interface, created with the dat.gui library [5], was tested via the test table. The test table defines a series of acceptance tests for changing the interface values.

5.8. Stress Testing

The FPS of the scene heavily depends on the hardware. For example if the project is run in the Orchard on a Mac, then 60FPS will be reached. Even if the rock count is at the max (2048), then the scene will still reach 50FPS. On a standard IS machine, the scene averages 20 FPS – but this is still quite smooth.

The water rendering is one of the most expensive operations in the scene. This is because the scene has to get rendered an extra 2 times per frame (to render the reflection and refraction textures). Therefore whenever the rock amounts get changed, they are not just rendered once, but three times. Once to the scene, and once to the refraction and reflection textures. Changing the rock count from 100 to 200 per section actually means it goes from: 300 rocks (3 render calls of 100 rocks) to 600 rocks (3 renders of 200 rocks). So the increase is actually 300 rather than the expected 100.

The below image on the left shows the initial instanced rendering working. It has 5 cubes rendered from the same set of vertices, but with different matrices applied to them. After this was working, the actual rock vertices where introduced and the amount of rocks was changed to 200,000. This is shown in the image on the right. The scene held 20 FPS, which for that number of rocks is very good. Both of these images can be seen in higher resolution from [30] and [31].



As the water rendering was later introduced, rendering 200,000 rocks at 20 FPS is no longer possible due to the 3x render calls per frame. The scene would actually have to render 600,000 rocks per frame.

5.9. Memory

The terrain generation consumes the most memory because it has to create all of the vertices, normal, UVs and indices. The image on the right shows the total memory consumed by the terrain generation function, for a 4x4 map size.

Nothing else in the program even comes close to the memory taken up by the terrain. However, 1.9 million bytes is only 1.9 megabytes of data.

Ramping up the terrain size to a 12x12 section yields a 29 megabyte memory usage, for the fillHeightMap function alone.

Self Size (bytes)		Total Size (bytes)		Function
1 908 360	28.52 %	1 908 360	28.52 %	▶ fillHeightMap
1 046 240	15.64 %	1 046 240	15.64 %	▶ createHeightMap
423 588	6.33 %	423 588	6.33 %	▶ createQuadrantVertices
271 144	4.05 %	271 144	4.05 %	▶ createQuadrantUvs
269 520	4.03 %	649 140	9.70 %	▶ (anonymous)
203 428	3.04 %	203 428	3.04 %	▶ createQuadrantNormals
172 612	2.58 %	172 612	2.58 %	(V8 API)
131 328	1.96 %	131 328	1.96 %	▶ t
131 096	1.96 %	131 096	1.96 %	▶ createQuadrantIndices

Self Size (bytes)		Total Size (bytes)		Function
29 176 424	68.76 %	29 192 916	68.80 %	▶ fillHeightMap
8 925 096	21.03 %	8 925 096	21.03 %	▶ createHeightMap
891 344	2.10 %	891 344	2.10 %	▶ OBJ.Mesh
406 848	0.96 %	406 848	0.96 %	▶ createQuadrantVertices
294 868	0.69 %	294 868	0.69 %	(V8 API)
271 144	0.64 %	271 144	0.64 %	▶ createQuadrantUvs
203 428	0.48 %	203 428	0.48 %	▶ createQuadrantNormals
185 760	0.44 %	514 264	1.21 %	▶ (anonymous)
131 224	0.31 %	328 504	0.77 %	▶ (anonymous)
131 096	0.31 %	131 096	0.31 %	▶ createQuadrantIndices

It might not seem like the terrain has gotten much bigger, after all it only went from 4 to 12, right? Well actually the terrain is created by its row size * column size. This means it is not going from 4 to 12, but instead 4x4, to 12x12. This means an extra 128 sections are added onto the map. Each new section contains 16,384 vertices and other data such as normals, UV coordinates and indices.

6. Critical Evaluation

6.1. Initial Requirements

The requirements spec [14] developed at the start of the project was not very realistic. A good 3D rover game in WebGL, with no prior low level graphics knowledge, was not possible given 400 hours or even more. The playability aspects of the game where the hardest to create. There was not an obvious task the player could go and do, which would have been enjoyable for them, whilst remaining easy to program. If the project was not a Mars rover game, but instead a space shooter– then the game aspects would have been much easier to create. A simple high score, with shoot to kill aspects would have been great to play and simple to program. The initial requirements where quite simple, the game needed to be 3D and also have some terrain and rocks.

6.2. Design Decisions

Since I was interested in using a low-level language from the start, libraries and game engines were not suitable. Due to the lack of experience about the languages that were going to be used, using an agile approach to develop the project was best. SCRUM has allowed for early spike solutions and up front code, this proved crucial to the project's success.

6.3. Changing the projects direction

As previously mentioned, the project started out as a game and later transitioned into a graphical scene. The game aspects of the requirements where simply stripped out and few new requirements where added. This was because the graphics of the existing project needed to be massively improved, as shown in the below images:

So when the project changed direction, it already had some terrain, rocks and water. Now, no longer focusing on the game aspects meant the existing graphics could be improved. There were several reasons behind why the graphics in the game were not great. The first reason was that they were being created from scratch in WebGL, which takes ages, even if done right the first time – let alone trying to debug with no error messages. The second reason was that the graphics were only half of the problem, there was still an entire game to build. This included UIs, collision, level boundaries, building a minimap, user missions and adding XP. Nearly all of these features where removed after the mid-project demonstration.

6.4. Strengths and Weaknesses

This project has demonstrated complex computer graphics techniques such as: building a 3D environment, using instanced rendering and rendering to textures. This is especially valuable because the project has been built in WebGL, a new language with few online learning resources. If somebody wants to see, or learn, any of the above techniques, then this project will be a useful resource.

The project is easily accessible to users as it runs within a browser, this requires no plugins, installation or download.

The strengths of the project for me, are that I now have a solid low level graphics foundation. Learning WebGL will help me learn other graphics libraries in the future, such as Vulkan and Direct3D. It will also make using libraries even easier, as I now have a good understanding at the lower level.

A big weakness of the project is that it is not supported on as many devices as I would like. This is due to the extension libraries that have been used. There is no real fix to this as it is up to the browser companies to implement the extensions.

6.5. Problems Faced

Developing with WebGL is a nightmare, especially for a beginner. For example if we try to get the location of a variable in a shader and that variable does not exist, we will not get an error. Likewise if we try and pass data from the vertex shader to the fragment shader, we will not get an error if the variables are spelt differently.

The worst errors are “attempt to access out of range vertices in attribute 0”. This essentially means anything could be wrong with the data we are trying to draw. WebGL will say attribute 0, which is usually referring to the vertex positions. However the majority of the time the error will not be here and could instead be in any attribute. There is no easy way to solve this. The only way is to step back through every single setup and draw call, to eventually find a silly mistake, such as we told WebGL to draw one too many vertices.

When the terrain was changed to use sub sections, rather than one massive section, a lighting bug was introduced. The normals of the terrain where created by using the previous row of vertices. However since the terrain was no longer all connected, black lines appeared on the quadrant boundaries. This was solved by simply setting all the terrain normals to point straight upwards.

FPS issues set the project back, but there were only two real options. The first option was to try and use a really efficient technique early on and run the risk of it not working at all. Or, we can implement something not very efficient and build on it afterwards. Using WebGL was difficult enough already, let alone trying to use complex techniques like instanced rendering from the get go. This was why the initial versions of the rocks and terrain where implemented in less efficient ways.

Although the features where constantly changing, this is expected when learning a new language. The first projects or features created are not going to use the most efficient techniques as firstly, we are simply unaware of them and secondly they would be too difficult to start with.

6.6. Starting the project again

Looking back, there should been a clear game playability aspect going into it. When the game started to be written, it became apparent that creating it was just not feasible given the timeframe. I was so caught up in creating a 3D scene and engine, that the problem only arose when it was time to make the game.

If the project were being made again – then OpenGL would be the language of choice. The reason WebGL was chosen in the first place was due to the base language of JavaScript. But now, due to my experience with other languages, OpenGL would be a better fit. OpenGL is much better documented as it has existed for over 20 years. There are many tutorials on advanced techniques in OpenGL, the same cannot be said for WebGL.

An agile approach would still be best suited to the project if it were being made again. Agile approaches are best if you are not 100% sure on the technical work that needs to be done.

6.7. Future Work

There are still numerous improvements that can be made to the scene. The main problem is its engine. The viewing distance has been kept between 128 and 256 and everything outside of that is not rendered. However this simply is not enough. We can see the terrain coming in and out of view when moving between sections, which is not ideal. Earlier in the project the viewing distance was 512, the fog covered the objects appearing instantly in view of the user. However this viewing distance and terrain had to be rewritten to improve FPS. An idea to further improve the terrain efficiency can be found here [33].

Moving on to improvements for the water, right now the water is just done via texture effects. Instead of using a simple quad for the water, we could use a large grid of vertices, (the same as the terrain). The vertices, updated on the GPU, could be moved in the shape of a sine wave to give a fluid motion. This, combined with the texture effects would yield very realistic water. However as the scene currently is not holding a stable 60 FPS, various other efficiency improvements would need to be done first.

The VAOs used to reduce buffer binds for the terrain could have been used throughout the project. However there was not enough time to implement it.

I would also like to create tutorials on the advanced features of the scene. This would make it easier for people to come and learn these new WebGL techniques. It would hopefully allow people to avoid the errors I made when implementing them.

6.8. Evaluating the project deliverables

Even though the project changed halfway through, the deliverables remained similar. The main deliverables where:

- 3D Scene
- Terrain using perlin noise
- Camera class, allowing the user to move

- Add in rocks
- Add in water

All of these have been achieved and more. The 3D scene was implemented within the first month of the project (but it was started a month early, so it actually took 2 months). Nearly all of this time was spent researching 3D graphics. Only around 30 hours of that time was spent implementing the 3D scene and fixing errors.

Terrain generation was done using stacked perlin noise. This gives greater detail to the terrain over regular noise and the rocks were instanced rendered.

The deliverables added whilst the project was in progress were: adding fog, a skybox and water. All of these features have been implemented to a high standard, especially the rocks and water. Developing this project has achieved my personal goal of learning how low level computer graphics works.

7. Appendices

A. Libraries

[0] **JQuery library** – This is only used to display a message when the user is going off screen. It is open source and available from the JQuery website. The snippet of JQuery code can be found starting on line 199 of the MyGUI file. This library was used without modification.
Date accessed: 10th March 2017.
Available at:
https://jquery.com/
Source available at:
https://github.com/jquery/jquery

[1] **M4.js library** – This is a matrix library created by Greg Tavares. It handles all the matrix operations that are needed in my scene. This library was used without modification.
Date accessed: 1st February 2017.
(Remove space between webgl- and fundamentals in url):
Available at:
https://github.com/greggman/webgl-
fundamentals/blob/master/webgl/resources/m4.js

[2] **Perlin library** – (ISC licensed). This contains the useful perlin functions to generate the terrain. The techniques come from Ken Perlin in his paper about perlin noise [67]. This library was used without modification. The accessed date is late because a different library was being used before it. This library was used as it could be seeded.
Date accessed: 14th April 2017.
Available at:
https://github.com/josephg/noisejs

[3] **Webgl-obj-loader library** – (MIT licensed). This library has been used to load in the singular OBJ rock used for my scene. This library and was used without modification.
Date accessed: 30th April 2017.
Available at:
https://github.com/frenchtoast747/webgl-obj-loader

[4] **TWGL.js library** – This library was used in the RockGenerator and ParticleSystem classes. It simply made the buffer creation slightly easier. This library was used without modification.
Date accessed: 30th April 2017.
Available at:
https://github.com/greggman/twgl.js/

[5] **Dat.gui.js library** – This library was used for the interaction panel in the top right of the screen. It allows the user to interact and change the scene properties. It is licensed under the Apache version 2.0. This library was used without modification.
Date accessed: 30th April 2017.
Available at:
https://github.com/dataarts/dat.gui/blob/master/LICENSE

[6] **Stats.js library** – (MIT Licensed). This library is used for the top left component in the scene. It shows the FPS, and if you click on it, it also shows memory usage. This library was used without modification. Date accessed: 30th April 2017.
Available at:
https://github.com/mrdoob/stats.js/

B. Ethics Submission

Ethics Application Number: 6677

AU Status
Undergraduate or PG Taught
Your aber.ac.uk email address
sds10@aber.ac.uk
Full Name
Samuel David Snowball
Please enter the name of the person responsible for reviewing your assessment.
Reyer Zwiggelaar
Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
rrz@aber.ac.uk
Supervisor or Institute Director of Research Department
cs
Module code (Only enter if you have been asked to do so)
CS39440
Proposed Study Title
Mars Mission Control Game in WebGL
Proposed Start Date
31/01/2017
Proposed Completion Date
08/05/2017
Are you conducting a quantitative or qualitative research project?
Mixed Methods
Does your research require external ethical approval under the Health Research Authority?
No
Does your research involve animals?
No
Are you completing this form for your own research?
Yes
Does your research involve human participants?
No
Institute
IMPACS
Please provide a brief summary of your project (150 word max)
A game, allowing players to roam around on a game world like Mars. Playable in a browser.
Where appropriate, do you have consent for the publication, reproduction or use of any unpublished

material?
Not applicable
Will appropriate measures be put in place for the secure and confidential storage of data?
Yes
Does the research pose more than minimal and predictable risk to the researcher?
Not applicable
Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth
Office advise against travel to?
No
Please include any further relevant information for this section here:
If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check.
Tick to confirm that you will ensure you comply with this requirement should you identify that you
require one.
Yes
Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and
that you will inform your department should the proposal significantly change.
Yes
Please include any further relevant information for this section here:
N/A

C. Code Samples

[7] The old terrain rendering code – Can be found in code/old code snippets/old_terrain_code.js/

[8] The new terrain rendering code – Can be found in code/old code snippets/new_terrain_code.js/

D. Third-Party Code Samples

[9] GL_TRIANGLE_STRIP indices creation – Taken from user Jay Conrod on stack overflow. It is the answer with 11 up votes, and is just the second part of his code. It has been used in the createQuadrantIndices function in Terrain.js. Accessed: 2nd February 2017

The code is available from: https://goo.gl/9cwkgX

[10] A simple function to resize a canvas. Used in MarsScene.js. Available at: https://goo.gl/3Rm8v5

Annotated Bibliography

A. References

[11] The projects GitHub repository
Available at:
https://github.com/SamuelSnowball/Major-Project

[12] User manual – This file shows the compatibly of the project on various different browsers and operating systems. It also contains the build instructions.
Available at: Major Project/documentation/User manual.docx

[13] Sprint additions file – This is a minified version of the product backlog. It shows the major additions over the sprints.
Available at: Major Project/documentation/sprint additions.docx

[14] Requirements spec – The initial requirements document
Available at Major Project/Outline spec/sds10_OutlineProjectSpecification.pdf

[15] Full product backlog – This contains everything that was added to the project, including dates. To see a minified version, view the sprint additions file mentioned above.
Available at: Major Project/Product backlog.txt

[16] Flow diagram of a render pass
Available at: Major Project/documentation/Flow diagram/Flow Diagram.png

[17] Minified class diagram
Available at: Major Project/documentation/Class diagram/minified.png

[18] Full class diagram
Available at: Major Project/documentation/Class diagram/final.png

[19] User interface design document
Available at:
Major Project/documentation/User interface design/UI Desgins.pdf

[20] Initial multi-coloured terrain
Available at: Major Project/screenshots/terrain/terrain0.png

[21] Document for texturing the terrain
Available at: Major Project/documentation/Terrain Texturing.pdf

[22] Stacked noise terrain
Available at: Major Project/screenshots/terrain/stacked noise.png

[23] Old water image
Available at: Major Project/screenshots/water/0_original_water.png

[24] Rendering to a texture
Available at: Major Project/screenshots/water/1_render_to_texture.png

[25] Rendering to texture with clipping planes
Available at:
Major Project/screenshots/water/2_refraction_and_reflection_textures.png

[26] Water with horrible bugs
Available at: Project/screenshots/water/3_bugs_everywhere.png

[27] Options for unit testing JavaScript
Available at:
Major Project/documentation/Testing/Options for unit testing the project.pdf

[28] Old test table
Available at: Major Project/documentation/Testing/Old test table.pdf

[29] New test table
Available at: Major Project/documentation/Testing/New test table.pdf

[30] Initial cube instancing
Available at: Major Project/screenshots/rocks/initial_instancing.png

[31] 200,000 rocks instanced rendered
Available at: Major Project/screenshots/rocks/200k_rocks_20fps.png

[32] Testing terrain quadrant clipping
Available at:
Major Project/documentation/testing/Testing terrain quadrant clipping.pdf

[33] An idea to further improve the terrain rendering efficiency
Available at:
Major Project/documentation/Future terrain efficiency idea.pdf

B. External references

[34] To see if the browser can run WebGL (v1.0)
Date accessed: 3rd May 2017.
Available at:
http://webglreport.com

[35] Uses of computer graphics
Date accessed: 5th May 2017.
Available at:
http://www.explainthatstuff.com/computer-graphics.html

[36] OpenGL 3D game, created by YouTube user OneGoldenCat.
Date accessed: September 15th 2016.
Available at:
https://www.youtube.com/watch?v=aviL3HX3UEc&t=267s

[37] Cartesian coordinate system image
Date accessed: 1st May 2017.
Available at:
https://goo.gl/irYLwA

[38] Local coordinate system in the world coordinate system image
Date accessed: 1st May 2017
Available at:
https://goo.gl/tXeqqQ

[39] Translation matrix times vec4 image
Date accessed: 1st May 2017
Available at:
https://goo.gl/ckczxm

[40] Transforming between coordinate systems image
Date accessed: 1st May 2017
Available at:
https://learnopengl.com/#!Getting-started/Coordinate-Systems

[41] OpenGL ES 2.0 Graphics pipeline image
Date accessed: 1st May 2017
Available at:
https://www.khronos.org/opengles/2_X/

[42] Normalized device space image
Date accessed: 1st May 2017
Available at:
https://hacks.mozilla.org/2013/04/the-concepts-of-webgl/

[43] Multi-coloured triangle image
Date accessed: 1st May 2017
Available at:
http://web.eecs.umich.edu/~sugih/courses/eecs487/pa1.html

[44] Viewing frustum image
Date accessed: 1st May 2017
Available at:
http://encyclopedia2.thefreedictionary.com/Viewing+frustum+culling

[45] Vertex clipping image
Date accessed: 1st May 2017
Available at:
https://goo.gl/VBzsu6

[46] Texture knowledge, from Greg Taraves on Stack Overflow
Date accessed: 1st February 2017
Available at:
https://goo.gl/mT8Ck7

[47] Texture knowledge from the Mozilla tutorial. From various contributors.
Date accessed: 1st February 2017
Available at:
https://goo.gl/yB3Df6

[48] Texture knowledge from the Mozilla tutorial on GitHub.
From various contributors.

Date accessed: 1st February 2017.
Available at:
https://goo.gl/s3orxo

[49] Texture mapping image
Date accessed: 1st May 2017
Available at:
https://goo.gl/OFMg6n

[50] Samuel R Buss - Yaw, pitch and roll
Pages 296 – 297
Date accessed: 25th February 2017
Available at:
https://goo.gl/nQtkj8

[51] Steven M LaValle 2006 - Yaw, pitch and roll
Date accessed: 25th February 2017
Available at:
http://planning.cs.uiuc.edu/node102.html

[52] Camera to view image
Date accessed: 1st May 2017
Available at:
https://www.3dgep.com/wp-content/uploads/2011/07/View-Matrix.png

[53] Random noise plotted on a graph image
Date accessed: 1st May 2017
Available at:
https://goo.gl/6o8QIQ

[54] Height map image
Date accessed: 1st May 2017
Available at:
http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=8

[55] Hearn, D., & Baker, M. P. (1997). *Computer graphics, C version* (2nd ed.).
Pages 496 – 499.
Discusses lighting techniques such as point lights and diffuse/specular lighting.
Date accessed: 20th February 2017

[56] GL_TRIANGLE_STRIP image
Date accessed: 1st May 2017
Available at:
https://goo.gl/TWBVWX

[57] Fog image
Date accessed: 1st May 2017
Available at (taken as a snapshot at: 2:21):
https://www.youtube.com/watch?v=gslBNLeSPUc&t=141s

[58] OpenGL instancing tutorial, by Jamie King
Date accessed: 01st September 2016

Available at:
https://www.youtube.com/watch?v=GC8Pj7g_vc0

[59] Screen space image
This has been edited from the normalized device space image defined in one of the above references
Date accessed: 1st May 2017

[60] WebGL texture coordinate system image
Date accessed: 1st May 2017
Available at:
https://goo.gl/vrMTcV

[61] ThinMatrix (YouTube name) - Specular lighting tutorial
Date accessed: February 1st 2017
Available at:
https://www.youtube.com/watch?v=GZ_1xOm-3qU&t=409s

[62] Perlin noise octaves
Date accessed: 3rd May 2017
Available at:
http://libnoise.sourceforge.net/glossary/

[63] ThinMatrix (YouTube name) Fog tutorial, implemented in Java and OpenGL.
Date accessed: 1st February 2017
Available at: https://www.youtube.com/watch?v=gslBNLeSPUc&t=141s

[64] ThinMatrix (YouTube name) Skybox tutorial, implemented in Java and OpenGL.
Date accessed: 1st April 2017
Available at: https://www.youtube.com/watch?v=_Ix5oN8eC1E

[65] ThinMatrix (YouTube name) Water tutorial series, implemented in Java and OpenGL.
Date accessed: 4th April 2017
Available at: https://www.youtube.com/watch?v=HusvGeEDU_U&t=60s

[66] draw.io – A free online tool used to create the class and flow diagrams. It can be used to open the XML files found in the documentation/class diagram folder. These XML files show the project design at different stages. The more brackets a file has after its name, the later it was created. They are not very important though.
Date accessed: 1st February – 2nd May 2017
Available at: https://www.draw.io/

[67] Improving noise, by Ken Perlin
ACM Trans. Graph. In Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, Vol. 21, No. 3. (July 2002), pp. 681-682, doi:10.1145/566570.566636 Key: Per02

This was the noise used to generate the height map for the terrain.

Date accessed: 1st January – 1st April 2017

Available at:
https://goo.gl/ANVXN0

[68] YUIDoc – JavaScript Documentation Tool
Date accessed April 1st – May 07th 2017
Available at:
http://yui.github.io/yuidoc/

C. Scene resource references

[69] Inspiration image of Mars
Date accessed: April 03rd 2017
Available at:
Major Project/resources/inspiration/prehistoric_mars.jpg

[70] Skybox day and night textures:
Date accessed: March 20th 2017
Available at:
http://www.custommapmakers.org/skyboxes.php

[71] Rock OBJ file used and edited from (Public domain):
Date accessed: March 01st 2017.
Available at:
http://nobiax.deviantart.com/art/Free-LowPoly-Rocks-set01-587036357

[72] Some rock textures
Date accessed: March 01st 2017.
Available at:
https://www.textures.com/download/rocksarid0035/68071?&secure=login
https://www.textures.com/download/rocksarid0048/42217?&secure=login
https://www.textures.com/download/rocksarid0035/68071?&secure=login

[73] Terrain sand texture
Date accessed: March 27th 2017.
Available at:
https://www.textures.com/download/soilbeach0131/106132

[74] Water sound:
Date accessed: April 15th 2017.
Available at:
https://www.youtube.com/watch?v=rn6zP4P0his&t=526s

[75] Water textures, DuDv map, normal map
Date accessed: March 27th 2017.
Available at:
https://goo.gl/vJB3qO