**A Prehistoric Scene of Mars in WebGL**

Final Report for CS39440 Major Project

*Author*: Samuel Snowball ([sds10@aber.ac.uk](mailto:sds10@aber.ac.uk))

*Supervisor*: Dr Helen Miles (hem23@aber.ac.uk)

30th April 2017

Version 1.1 (Draft)

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’d like to thank my supervisor Dr Helen Miles for assigning me the project and also for her continuous support throughout.

Last but not least I’d 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 project uses some features of WebGL (2.0) which are only available in Chrome and Firefox.

You can check if your browser can run the project here:

http://webglreport.com/?v=2

The original aim of this project was to create an interactive Mars mission control game, where the user would roam around Mars as a rover completing various tasks. The project direction changed after the mid project demonstration, It became clear I did not enjoy creating the game aspects of the project, so I decided to just focus on the graphics instead.

My motivation for the project was to learn how computer graphics work. At the start of this project 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.

Programming in WebGL will help me gain an understanding of what’s 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.

I haven’t been able to find a similar project built like this before. This is mainly due to it being in 3D and the low level languages used.

**Contents**

1. Background, Analysis & Process 3

1.1. Background 3

1.2. Analysis 3

1.3. Process 3

2. Design 3

2.1. Overall Architecture 3

2.2. Detailed Design 3

2.2.1. Even More Detail 3

2.3. User Interface Design 3

2.4. Other Relevant Sections 3

3. Implementation 3

4. Testing 3

4.1. Overall Approach to Testing 3

4.2. Automated Testing 3

4.2.1. Unit Tests 3

4.2.2. User Interface Testing 3

4.2.3. Stress Testing 3

4.2.4. Other Types of Testing 3

4.3. Integration Testing 3

4.4. User Testing 3

5. Critical Evaluation 3

6. Appendices 3

A. Third-Party Code and Libraries 3

B. Ethics Submission 3

C. Code Samples 3

Annotated Bibliography 3

# Background, Analysis & Process

## Background

The original project involved making a game which was something I had some experience in so I was immediately drawn to it. I had created some simple 2D games in the past, however I wanted this project to be in 3D. After the mid-project demonstration I switched the project from a game, to a graphical scene.

3D graphics had always fascinated me, how could you 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, which was the main thing I always wanted to learn. 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. I knew 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 confident to start.

## 1.1.1. Related Projects

NASA had developed a Mars rover game. Unfortunately there wasn’t anything I could learn from this as it had obviously built in a rush, being 2D and lacking game playability.

As WebGL is a very new technology, there haven’t been very many games or projects created in it, at least not from scratch. However as WebGL is based off OpenGL, most OpenGL projects are useful learning resources. The main project that inspired this one was a first person game in OpenGL. It was created by the youtube user OneGoldenCat. The great thing about this project is that it is open source and well commented.

//reference this https://www.youtube.com/watch?v=aviL3HX3UEc&t=267s

## 1.1.2. Language Research and Comparison

There was a variety of options to develop the scene. Game engines, libraries and raw graphics libraries where all available options. However, complex engines and libraries covered 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. Engines 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, you can’t fix bugs or render more efficiently (unless it’s 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 you 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. These libraries are much lighter weight than full game engines, but still cover up the details of the graphics get displayed.

A lower level option was Direct3D, which is an API built to render graphics on Windows machines. If you wanted to create a game or display graphics with this on multiple platforms, you can’t. Still, most games are written in Direct3D due to its solid Microsoft framework and developer toolsets. It can make use of the other DirectX libraries such as (DirectSound) as well. There are more flexible alternatives than Direct3D and it offered no benefits to the project.

Vulkan is another API for rendering graphics. It is new and much lower level than OpenGL, meaning it will run more efficiently on the hardware. It has a much cleaner API than OpenGL, which has had things thrown on top of it ever since its introduction in 1992. As Vulkan is very low level, it would be extremely difficult to learn with no real graphics knowledge. Because of this, it wasn’t suitable to use for the project.

A more 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.

Using OpenGL for graphics requires you to use a base language of C++. However there was an alternative to OpenGL called WebGL. This is a JavaScript API for rendering graphics within a browser and is based off an older version of OpenGL. This means the base code of the project would be in JavaScript, rather than the base code being in C++ using OpenGL.

The graphics aspect alone will be difficult for this project, 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. C++ is a complex language to use, handling your own memory and pointers are just a few of the main examples. JavaScript on the other hand is a beginner friendly language, meaning the WebGL graphics would be easier to write on top of it.

The standard graphics APIs like OpenGL and Direct3D would have been perfect if C++ was a forgiving language. However the benefits of these APIs did not matter as I couldn’t use C++ effectively.

All graphics APIs use some form of shader language. A shader is a program than runs on the GPU, processing data that you give to it. GLSL (OpenGL Shading Language) is used with OpenGL and is therefore used in WebGL.

Hence the final languages I decided to use where: WebGL, GLSL and JavaScript.

## 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 than WebGL, as they have existed longer. WebGL was designed to run within browsers, rather than other libraries which run on desktop.

## 1.1.3.1. 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 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 browser companies to implement WebGL themselves.

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, Firefox, Edge and Safari. All of this happens with no plugins or installation.

WebGL 2.0 was officially released in April 2017, therefore most browsers haven’t implemented the full specification fully. Internet Explorer, Edge and Safari are all examples of browsers that lack up to date versions of WebGL. For this reason, my project only runs in Chrome and Firefox.

## 1.1.3.2 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 all you need is a browser, rather than a full game engine or library. WebGL requires no installation or heavy loading times, making it much better 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’re doing to use it effectively. With game engines, you don’t 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 (lighting, texture effects).

WebGL (or graphics in general) has many uses such as: games, simulations, augmented reality and data visualization.

## 1.1.3.3. Benefits of working within a browser

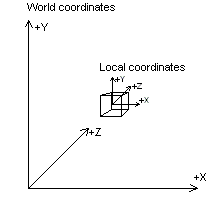
WebGL runs within a browser. So when working with WebGL, you get all of the benefits of working within a browser. HTML5 has very easy APIs to tap into such as audio, local storage, text rendering and methods to easily get user input.

If I didn’t switch the project direction 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’s easy to interoperate with web applications due to the ease of networking. You can also get resources at runtime from URLs instead of having large amounts of memory pre allocated for textures and video.

Web applications are also easier to test. Just test on one, or a few, browsers and that’s all you need. You no longer need to test on multiple different operating systems. If your application works fine on chrome in Windows, then it will work exactly the same in chrome on Linux. The operating systems don’t matter, it is all within the browser. Browsers are accessible through most devices, phones, tablets, laptops, essentially anything with an internet connection. More often than not they are lighter weight than desktop applications. All you have to do with a WebGL scene is just open the page, no installation or download needed.

## 1.1.4. Coordinate systems overview

We define a cube with some vertices, then use a 4x4 matrix to position it in our world. This positioning, or translation, is done via the model matrix. Rather than thinking of it as the model matrix, it is easier to think of it as the model to world matrix, as the matrix moves our model into the world. This world coordinate system contains all of our objects. Just imagine two coordinate systems within one another. The image to the right shows just that.



We then look at the world from the view of the camera. This is called the view matrix but again, it’s easier to think of it as the world to view matrix. As the entire world gets moved in front of the camera.

Finally, to display the world on the screen we need to convert our 3D virtual 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 3D to 2D conversion is done via a projection matrix.

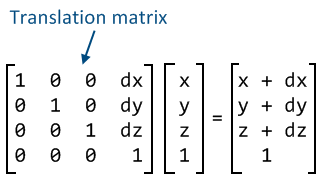
## 1.1.4. Coordinate systems in-depth

Imagine you 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 own 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 done by applying 4x4 matrices to the vertices of the cube. 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.

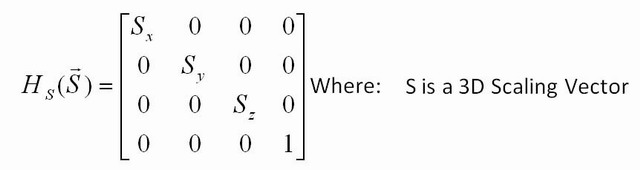
//not correct? … The original 3D points needs to be converted into 4D points before doing so. The forth column holds the values needed for translation.

The below image shows a 4D point that we want to move. To do this, it gets multiplied by a translation matrix. The output on the right is the points new position.



//credit this image

Not only can we use matrices to move the vertices of an object, but we can also use them to scale the vertices as well. The below image shows how you can scale an object by using a scaling matrix. This takes in (Sx, Sy, Sz) which is a 3D vector containing the scale values for each (x, y, z) point of the vertex. 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).



Lastly, rotation matrices can be applied to vertices

By performing these translations, the cube is now in the world’s coordinate system.

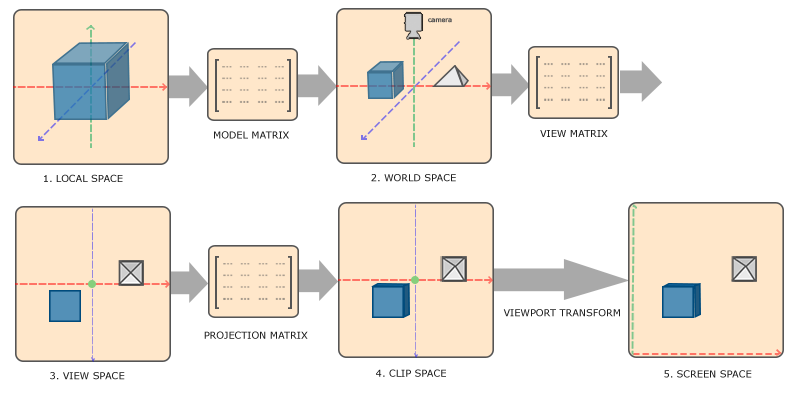
A benefit of matrices is instead of multiplying them all together then lastly with the vertex, like so: translationMatrix \* scaleMatrix \* rotationMatrix \* vertex. You can instead just multiply all the matrices together once, and apply that single matrix with each vertex of the model. This saves multiplying the matrices together for each point you want to translate. The final matrix containing the translation/scale/rotation values is usually called model matrix, or the full transforms matrix.

When multiplying matrices together you 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 from right to left).

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

The below image shows a cube, in its own coordinate system, that we want to move into world space. This is done by applying a model matrix containing the combination of the translation/scaling/rotation matrices as explained above. After this is done, the model is in the worlds coordinate system. This is where all objects in the scene are placed. 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 (worlds) then it’s easy to move all of them at the same time. This is useful for moving the world, looking like the user is moving within the world.



A camera matrix is then used to move everything in the world in front the camera. The camera is defined and has its own position and orientation in the world, just like a regular object. We want to render everything in front of where the camera is looking. However a step the below diagram misses out is creating the view matrix, by taking the inverse of the camera matrix. This moves the entire world in front of the camera, as this is actually what happens in graphics.

The projection matrix takes the world coordinates, and squishes them to clip space coordinates. These clip space coordinates range from -1 to +1 in all x, y, z spaces. A perspective division is done

View to projection (NDC) done with projection matrix. Multiply by projection matrix = Clip space, then perspective divide (divide by w) = NDC coordinates

//where does the 3d to 2d conversion happen

// https://learnopengl.com/#!Getting-started/Coordinate-Systems

## 1.1.4. Fixed function VS programmable graphics pipeline

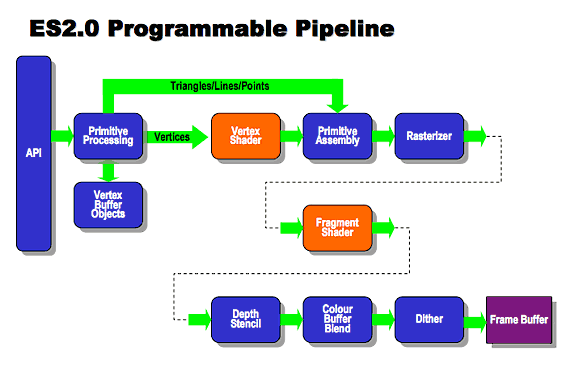
WebGL uses something called the programmable graphics pipeline, as opposed to the old fixed function pipeline. This newer pipeline was implemented into versions of OpenGL including and after 2.0. Hence the new pipeline is in WebGL, as it uses OpenGL ES 2.0. This pipeline is just the series of steps needed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

In the old fixed function pipeline it was very one size fits all and not customizable. There was no functionality for doing any custom calculations (lighting/fog), only what was built in. You could opt in for the features that you wanted, for example the pipeline should calculate fog, or you could opt out and not use them. However there was no functionality allowing you to customize the operations, only ones to enable or disable them.

But now, using the new programmable pipeline, it allows you to write programs called shaders which replace parts of the graphics pipeline. Shaders are programs which are written in GLSL. This is a C style language which allows you to specify how your data should be manipulated on the graphics card.

These shaders allow programmers to manipulate the data how they want, giving you much more power over what gets rendered. The disadvantage of these shaders is that they add complexity to application, as you now have to manage some stages of the graphics pipeline yourself. For simple applications which aren’t going to do any fancy graphics, then an older version of OpenGL (pre 2.0) would better suit your needs, as it uses the fixed function pipeline without customizable shaders.

## 1.1.4.1. Programmable pipeline

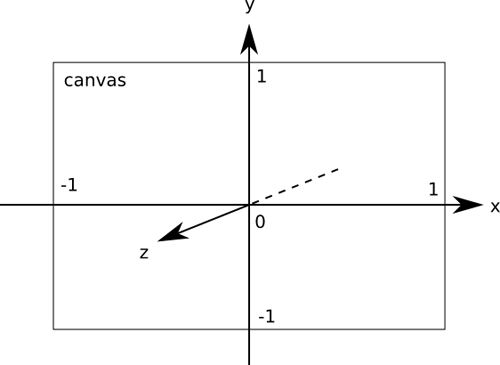


//credit image

Before I explain the details of the pipeline, 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.

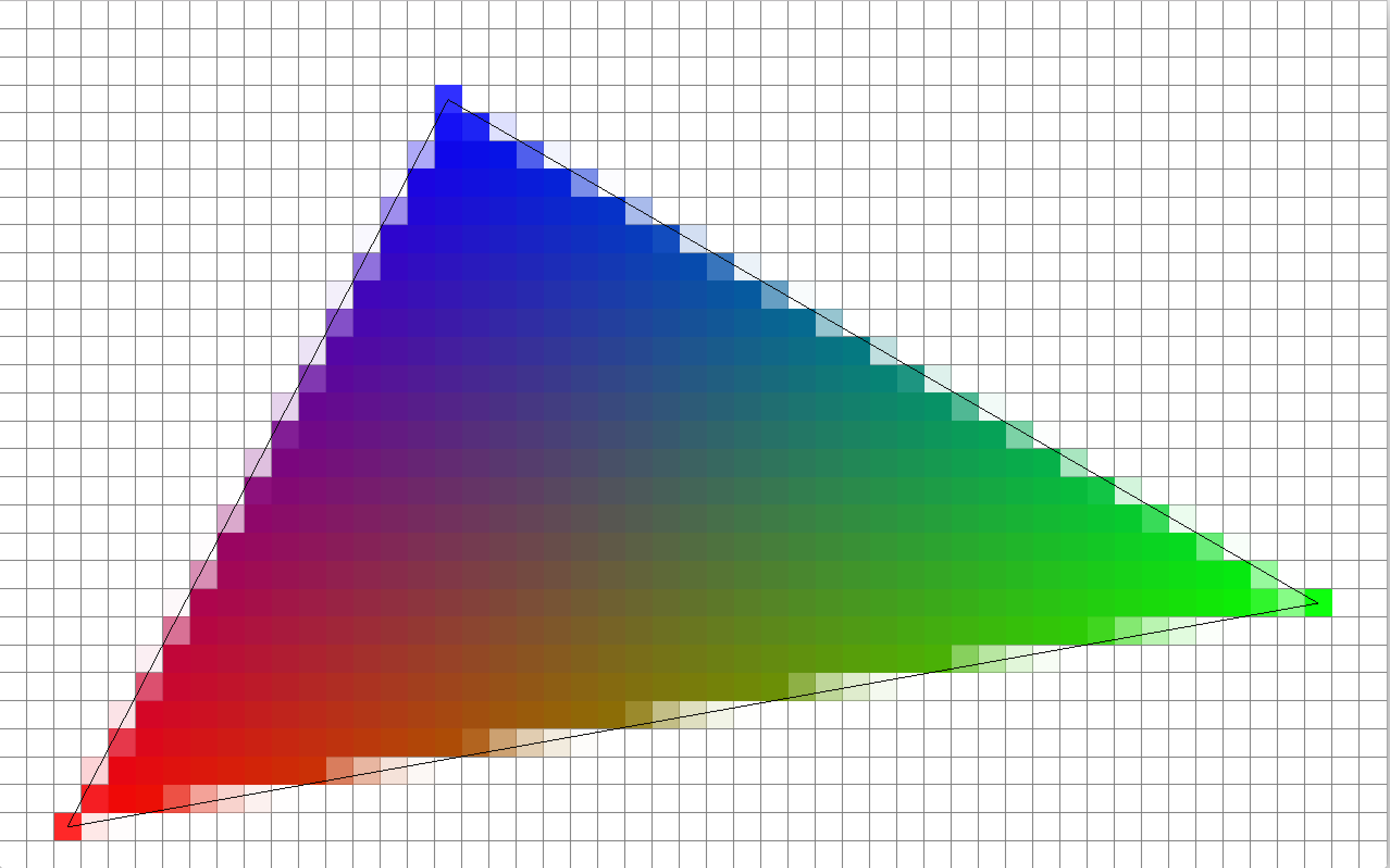
The scene data (vertices, colours) are created by the programmer on the CPU. The data then gets passed to WebGL (the API component on the left) through the calls to the its API. WebGL then takes care of passing the data through the graphics pipeline, all on the GPU. As WebGL uses the programmable pipeline,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

## 1.1.4.1. Stage 1 – Data input/data binding



In this example, the triangle vertices will be defined in normalized device coordinates (NDC). The WebGL NDC system is shown in the right image. It goes from -1 to +1 in all of the x, y and z axes.

Take the triangle in the image to the left and the above coordinate system as well. 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. These needs to be put into GPU memory and this is done by first creating a buffer, binding that buffer and finally putting our data in the buffer.

var vertexBuffer = gl.createBuffer(); // an array, in which we will store our data

gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer); // tell webgl we want to use the buffer we just made

gl.bufferData(gl.ARRAY\_BUFFER, vertices); // put the data into the buffer, which gets sent onto the GPU

/\*

This call describes the data that we just sent to the GPU

Parameters:

The shader location variable that we are sending it (position location, colour location)

The size of the data

Data type of each component in the array

Whether to normalize the data or not

Stride, the gap between vertex data. This is used when defining vertices/colours in the same array

Offset to the first vertex in the array

\*/

gl.vertexAttribPointer(shader\_location, 3, gl.FLOAT, false, 0, 0);

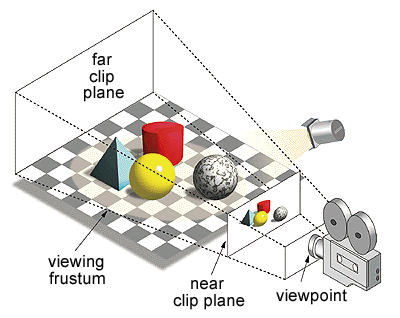
The process is then repeated for the colour data as well.

This process of uploading data to the GPU is slow, so should be minimized in render loops. The data then needs to be selected from the GPUs memory at render time with a call to gl.bindBuffer(gl.binding\_Target, buffer\_name) before issuing the draw call.

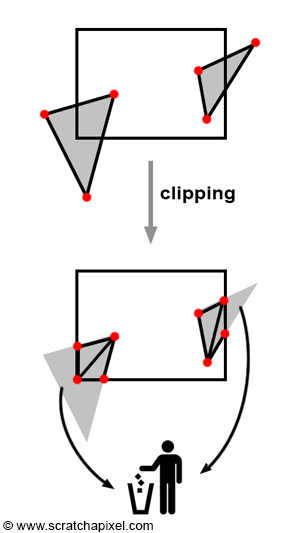
## 1.1.4.1. Stage 2 – Vertex shader

The vertex shader (orange component in above pipeline diagram) will get run for each vertex (3 vertices in this triangle example). You can also input 4x4 matrices into the shader to say where and how the vertices should be positioned in the world.

The vertex shader outputs the game/scene coordinates in clip space, ranging from -1 to +1 in the x, y, and z axes. Once the clip space coordinates of the vertices have been calculated, the shader will clip or cull any vertices that are fully or partially outside of the view of the frustum. The viewing frustum is shown in the image on the right.



If any of the shapes where outside of the viewing frustum, then they get culled. This means the entire object is discarded as the user can’t see it. It’s important to get rid of any unnecessary vertices as early as possible, if the user can’t see them – why waste time processing them through further stages of the pipeline? 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 objects remain. The image to the left shows the clipping taking place.



After the vertex shader has culled and clipped any out of range vertices, a step called perspective division happens which turns the 3D frustum (shown above) into 2D screen space. Essentially this squishes, or projects, the 3D scene into a 2D image so it can be displayed on screen (as shown above). Now we have some colourless shapes on the screen, they need to be filled with their colour.

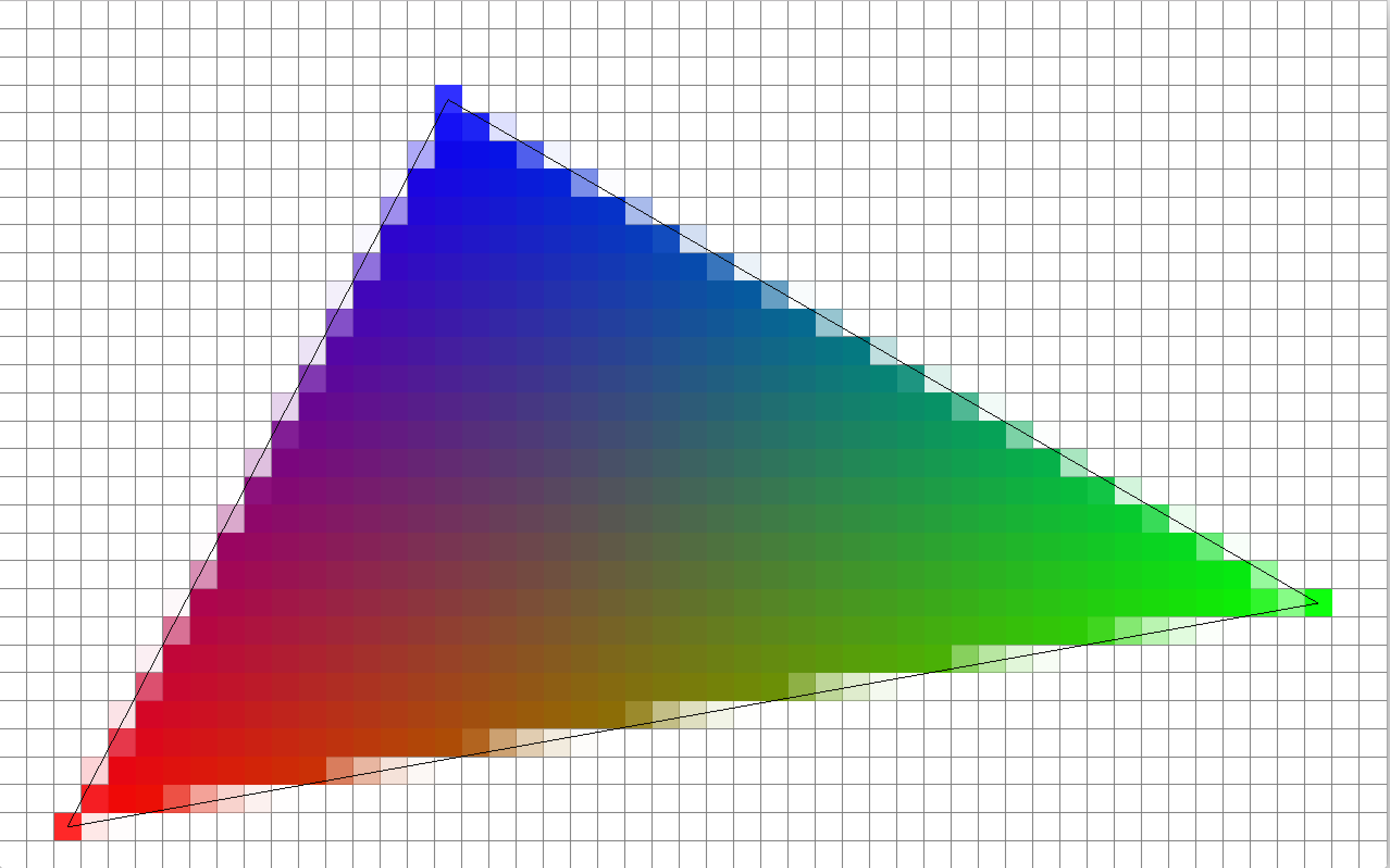
## 1.1.4.1. Stage 3 – Primitive assembly

Depending on the draw call that was issued, WebGL will form different primitives from the given vertices. For example with draw call: gl.drawArrays(GL.TRIANGLES) WebGL will construct a triangle between every 3 vertices given. There are also other drawing modes such as GL.POINTS and GL.TRIANGLE\_STRIP.

## 1.1.4.1. Stage 4 – Fragment shader

As you can see with the colours defined in stage 1, we only have 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, their colour is 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 wouldn’t notice this interpolation.

The programmers custom fragment shader, part of rasterization (or colouring in) stage of the pipeline, then gets run and sets the colour of the fragments between the defined vertices on the 2D image. The fragment shader runs for each fragment within the given object and outputs an (R,G,B,A) value. A fragment, just means a potential pixel. The fragment hasn’t been tested properly to see if it’s visible, so it might not get rendered. Pixels are things that appear on the screen, a fragment might not.



## 1.1.4.1. Stage 5 – Fragment testing and the depth buffer

What if we calculate the fragments/colour values for an object (take the blue pyramid in the above example) – but then calculate another objects colour that is in front it (take the yellow sphere in the above example). The object that is closer to the camera needs to get rendered on top of something that’s behind it. So in this case, the yellow sphere should partly render over the blue pyramid. There needs to be some sort of test to see which objects are closer than others. This is where the depth buffer comes in handy.

//depth values are probably wrong here

The depth buffer stores all of the depth information for the entire scene, it’s just a huge 2D array. For example the blue pyramids vertices would be stored with a depth of around -0.5. The yellow spheres vertices would be stored with a depth value of around +0.5, meaning they’re closer to the camera. This information is used to determine what the colour of the fragments should be. As the yellow sphere is closer, its fragments are chosen over the blue pyramids fragments.

Now the fragment shader has run for every fragment and has finished. All of these fragments are stored into the colour buffer, which is another huge 2D array of pixels. The fragments turn into pixels here as they’re no longer potential pixels, but actual pixels. Their depth has been tested and they’re definitely going to be rendered this frame.

## 1.1.4.1. Stage 6 – Updating the frame buffer

The frame buffer is the final 2D image that’s 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, it can be swapped with the current frame buffer to show the newly updated frame. This technique is called double buffering. The pixels don’t render onto the screen directly as it would result in parts of the scene being drawn before others. It’s better to wait for the entire scene to be rendered to an off screen buffer, before swapping it with the current frame buffer.

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

## 1.1.X. Learning resources

To start learning OpenGL I used:

<https://www.youtube.com/watch?v=6c1QYZAEP2M&list=PLRwVmtr-pp06qT6ckboaOhnm9FxmzHpbY>

<http://www.opengl-tutorial.org/>

https://learnopengl.com/

To start learning WebGL I used:

<https://webglfundamentals.org/>

## 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’s no premade 3D scene where you can drag and drop objects with nice textures and lighting, it is extremely lightweight and close to the hardware.

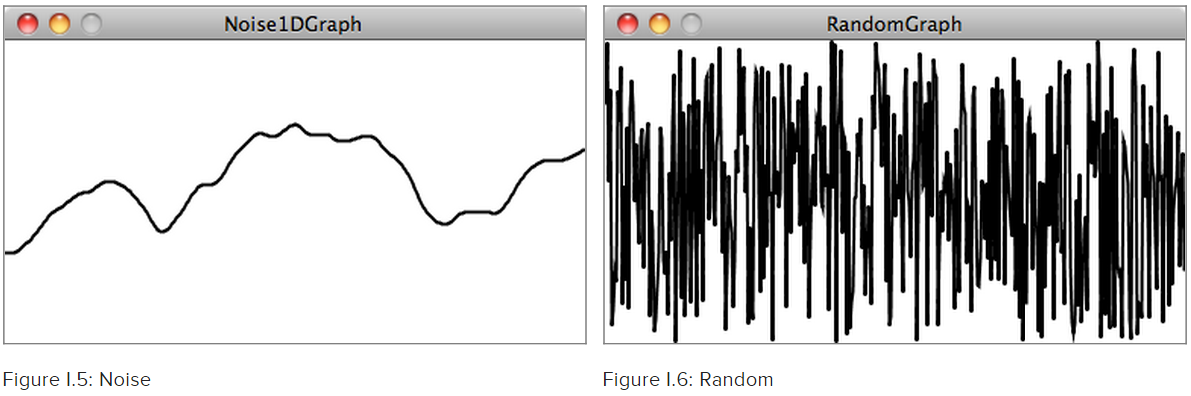
A basic WebGL scene has no: physics, event handling, sound, helpful error messages, loading/saving or fast rendering efficiency. However for me, these are not disadvantages. Instead, these are rewarding tasks that I can implement and by implementing them, I will learn how they work. This completes one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

Features and techniques used will be researched throughout the project. Researching lots at the beginning wouldn’t be very useful as I might not be able to apply the information straight away, as other features are needed first. Working software early is also a priority. For this reason the engine components need to be built first, followed by the graphics later on.

In terms of what the scene should have, there was no defined specification. Because of this, a specification had to be created with brief details of what the project should contain. The project essentially had infinite potential in terms of what could be added. There should obviously be terrain for the player to roam around on. So there needed to be some sort of terrain generation.

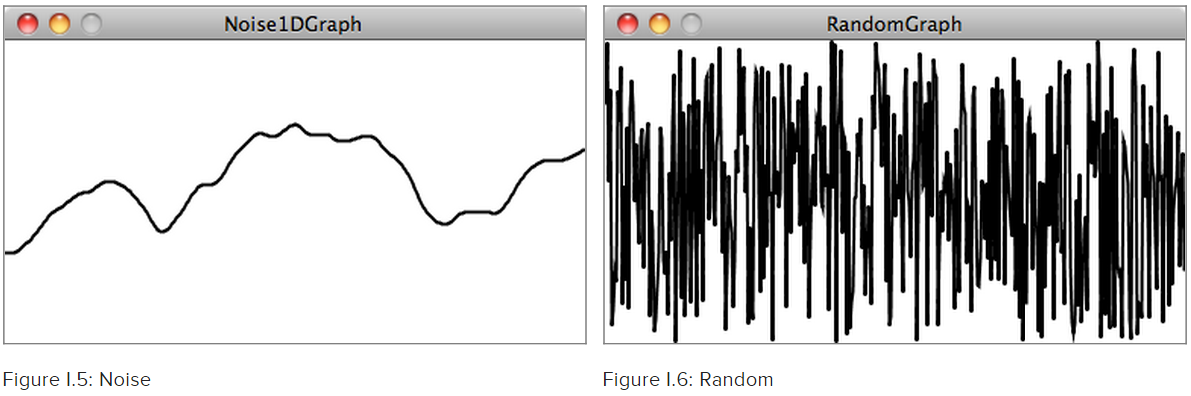
With the generating terrain, there were options of building it using existing height maps or, using perlin noise. The problem with using existing height maps is that there is no technical skill involved, it is literally loading an OBJ into the project. An OBJ file is simply a text file with the vertices and other data relating to an object pre-defined, this means it never changes.

However with using perlin noise, the terrain generation could be randomized. Perlin noise is an algorithm created by Ken Perlin reference which generates smooth random numbers. These smooth random numbers can be used to set the heights of vertices, which leads to the natural looking terrain.



Imagine you used a random() function, and plotted the result numbers on a graph. The output would the similar to this graph:

//http://stackoverflow.com/questions/25816351/java-2d-perlin-noise-for-random-terrain-generation

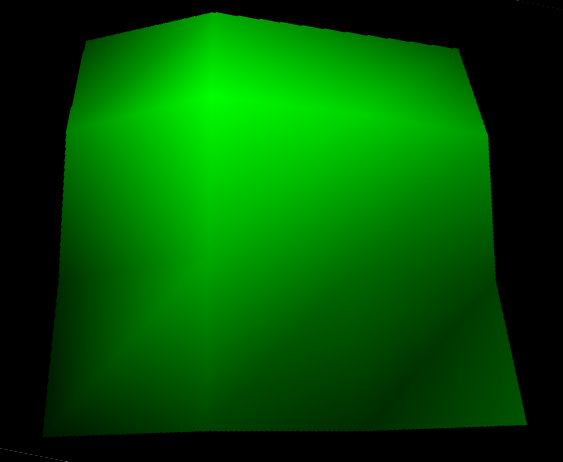
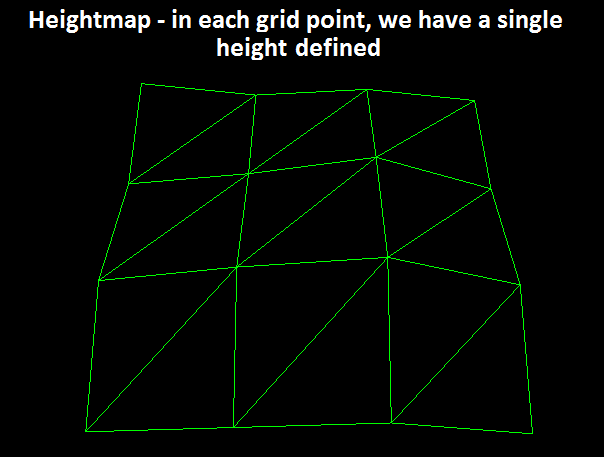


Every number chosen, or point plotted, has no relation to the number before it, or the number after it.

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 a 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’s up to us as to what they’re used 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, you end up with the image on the right.

//http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=8

This was the initial set of tasks, when the project was still a game:

Some general information about the project and 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
* I will need to test and cater for multiple browsers

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

* Build the 3D environment, allowing a camera to move around in the world. Translations and rotations should be applied to the world when the camera is moving.
* Build terrain using perlin noise. This terrain building process would probably be ongoing, as I add new areas and such throughout the project.
* Allow user to move around terrain, camera class needed.
* Add collision with terrain, rather than flying over it. Possibly connect terrain vertices with quad strip for more realistic collision, rather than existing triangle strip.
* Give terrain texture, procedurally generated perhaps.
* Add in rocks. I’ve found a procedural rock generation library I could use to generate awesome rocks. However I would rather try implement rocks myself. I could use a sphere geometry and use perlin noise on the vertices to give a bumpy geometry.
* Start work on GUI’s, with a possible (not necessary) minimap, and other GUI’s showing the rovers stats and location (perhaps just their coordinates). Commands from mission control would get sent here, the user would then complete them. Example: go to the X,Y,Z quadrant, find and sample X rock.
* Add in various missions for the player to complete, perhaps adding in levelling mechanics. For example once the player has completed x number of missions he can access new areas.
* Add audio.
* Add clouds.
* Add sunlight, shadows – I’m unsure how to do lighting in WebGL so would require a lot of research, perhaps taking up a whole sprint. Or it might not be feasible to do this at all, just an idea.
* Add initial loading scene, rover jetpacks into the landing site, terrain comes towards player. This wouldn’t take long.
* Add water, I imagine this would take a whole sprint in itself but it’s something that would be good to add, from a technical and graphical side. I would have to use various tutorials and resources to be able to do this.

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. The great thing about this project being a game, and later a graphical scene, is that it can be released with whatever it has at the time.

## 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 I had chosen. They might have been the best and most efficient techniques to use, however if I could not implement them, the design would have been a waste of time.

This is a dangerous approach for this project and 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. It also means a denial of service attack is useless. If a user is taken offline, then they will still be able to run the project on their localhost.

If this project continued as a game, perhaps with networking 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).

There is no place to execute a code injection attack, as the scene as no unchecked input boxes. The GUI that is being used is an open source library, used by thousands of people, this is safe from a code injection attack.

## Process

There were a few different 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. This was due to several reasons, the biggest reason being 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.

As with all software, writing documentation is essential. However documentation should only be written if it adds value to the project. It should not hinder the development process. An agile approach allows documentation to be written after the story is complete, meaning it is unlikely that the documentation will have to later be changed. With a waterfall style approach documentation is written early, even before the code is implemented. If there are any problems implementing the design and an alternative approach ends up being used, then the documentation also has to be rewritten wasting lots of time.

Pair programming is a major component of XPs development model. It involves a navigator and driver (inspector and programmer) sitting down helping each other write the code, on one machine. However this obviously requires 2 people and therefore was not suitable for this project.

Since XPs principles mainly rely on a team (code inspection, collective code ownership, pair programming) and since SCRUM is just a flexible framework, it made the most sense to use SCRUM for the project. SCRUM allows the developer to pick and choose which agile processes he will use. For example, you 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, as described in the analysis section. However these stories can be added throughout the project. 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 contains all of the tasks that need to be added, and have already been added, to the project. It also contains information about every sprint including what was implemented. 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, then the language choice could have easily been adapted to use a mid-weight library such as three.js. 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. Agile methodologies also value responding to change, this was especially important as when a new technique was learned, it was implemented into the project. For example the rocks where initially rendered with a draw call for each of them. Later on, instanced rendering was used to reduce the draw calls.

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 was done, laying out a foundation for how the code would be implemented. This is adapted from the standard 15 minute SCRUM standup – as there are no people to standup and discuss with.

At the end of a sprint, working, tested, documented software is meant to be released. However with this project, new techniques and methods were constantly being discovered, meaning it was a difficult thing to achieve. For example the rock generation was rewritten several times over, as new methods were learnt that improved the rendering efficiency. A release version is also published at the end of every sprint (minus one version, as I forgot) showing the sprints progress.

Due to this project being developed independently, most agile processes were altered, or removed completely.

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 for techniques and tasks that were not well understood.

On a side note, GitHub was used for version control. This provided various backups and restore points for the project. It also allowed my supervisor to easily come and see the progress of the project, without having to download the latest version.

# 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 use. The only thing you can use is the browser developer console, you 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 testing process.

The design of the project evolved throughout. Agile development does not recommend a large up front design, hence why this project didn’t have one. Tasks would be 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, on the features that were going to implemented – but really there was design all throughout the day.

//show how the design evolved over the project?

// show the method design, talk about final terrain rendering etc ???

## Overall Architecture



The easiest way to understand my scene is to look at the flow diagram. First, we define a key (left image) for the different components of the diagram.

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



At the start of the render loop, the entire screen is cleared, ready for drawing the new frame. The program then checks if the user is currently pressing down a key (W for forward). 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 is updated by the direction in which they’re facing, times by the camera speed. The cameras Y position isn’t updated here, instead the user can go up by pressing R (for rise).



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



The vertex shader gets run on every vertex, outputting the gl\_position variable.

The information that isn’t used by the vertex shader is passed into the fragment shader.

This in turn performs the depth test, discarding fragments that can’t be seen. This is shown with the decision (diamond) box in the below diagram.

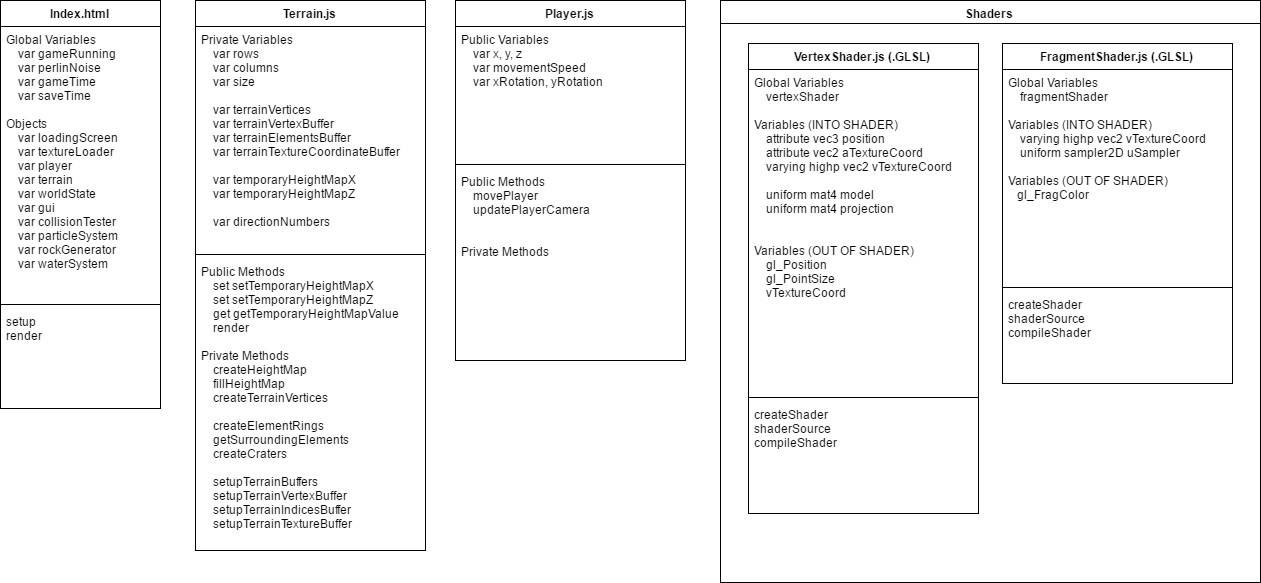
If the current fragment passes, then it is written into the colour buffer.

Once all objects in the scene have been drawn to the colour buffer, it is swapped with the frame buffer to update the display.

## Detailed Design

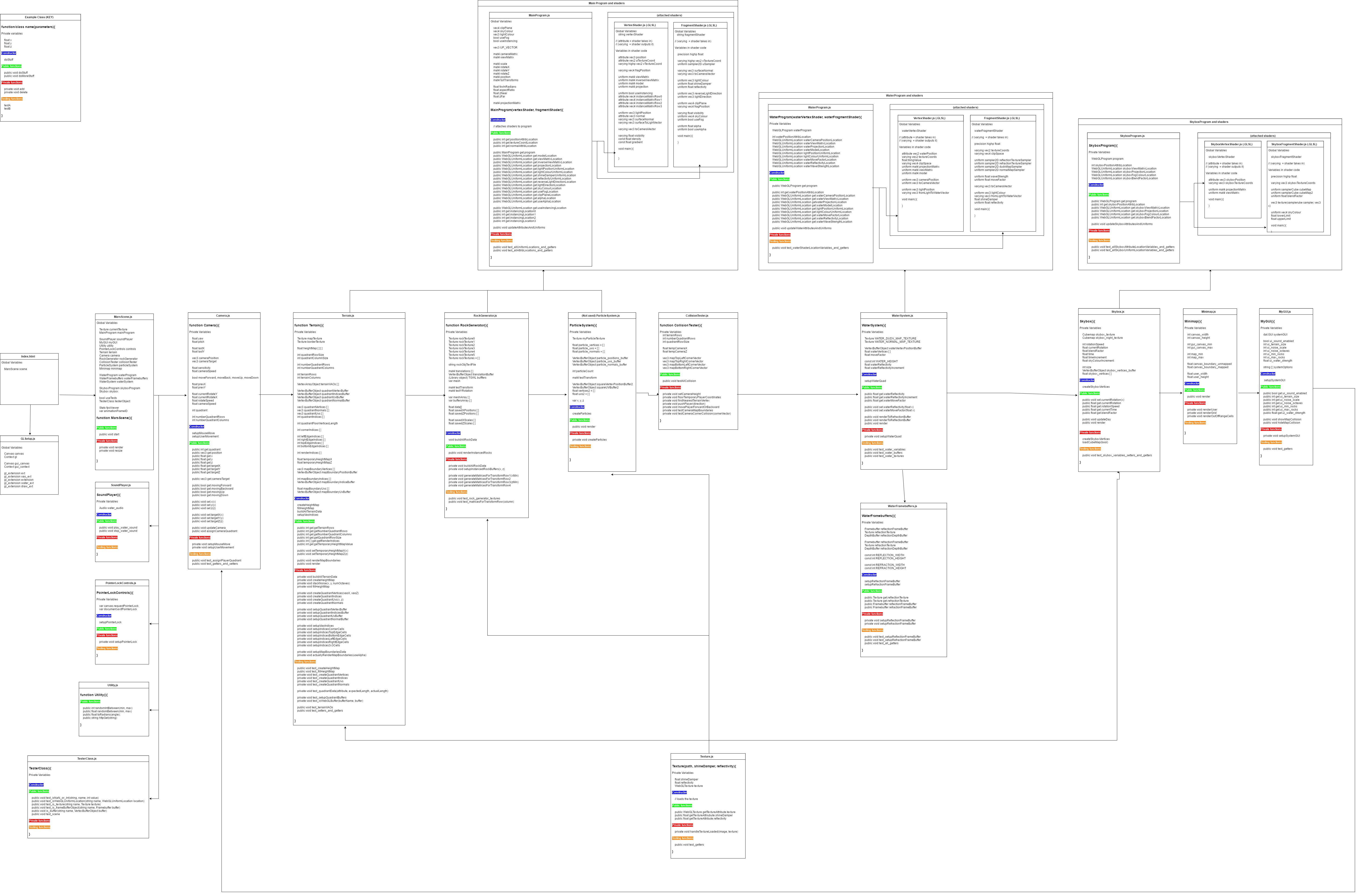
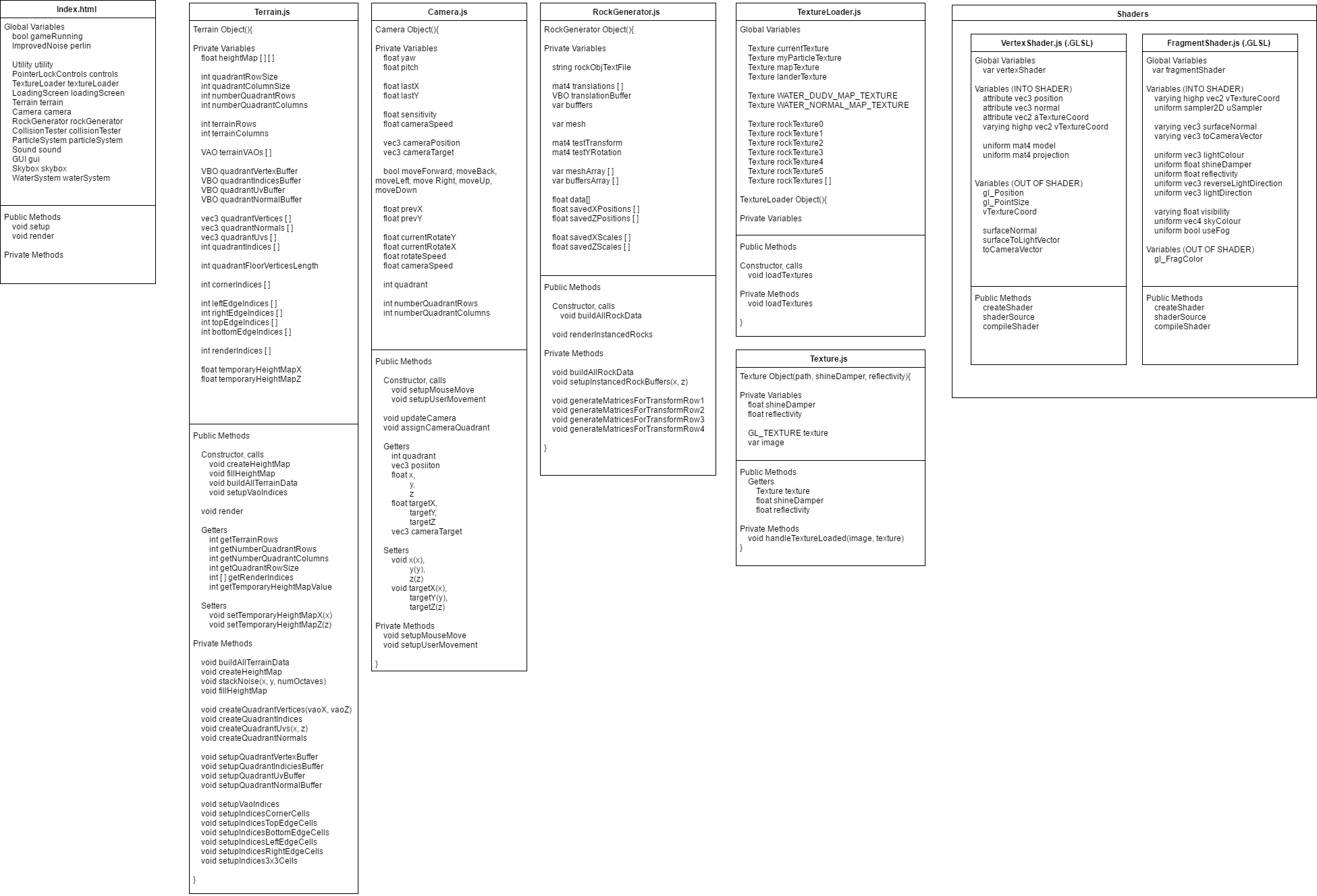
Since this project was developed with SCRUM, the design of the project was constantly evolving. A game/scene allows for a very adaptable design. New classes are simply added, containing with a setup and render method, then added these setup and render method calls into my main setup function and render loop.

In the documentation folder you will find lots of XML files. These files contain the class diagram of the scene at different stages of development. These XML files can be opened with draw.io. // reference draw.io here. In the initial stages of my project, the class diagram was this:



The index file contains the setup and render methods used to start the scene. As you can see the class diagram is not very descriptive just using the JavaScript var keyword to define variables. At the end of a SCRUM retrospective it became clear that the class diagram needed better description. This was added to later versions of the diagram.

The below image shows a later version of the project around sprint 5. This shows the adaptability of the design. The new classes are simply added on, not changing or effecting the older classes. RockGenerator, TextureLoader, Texture and Camera classes have all be added. Although the camera class is just the refactored player class.



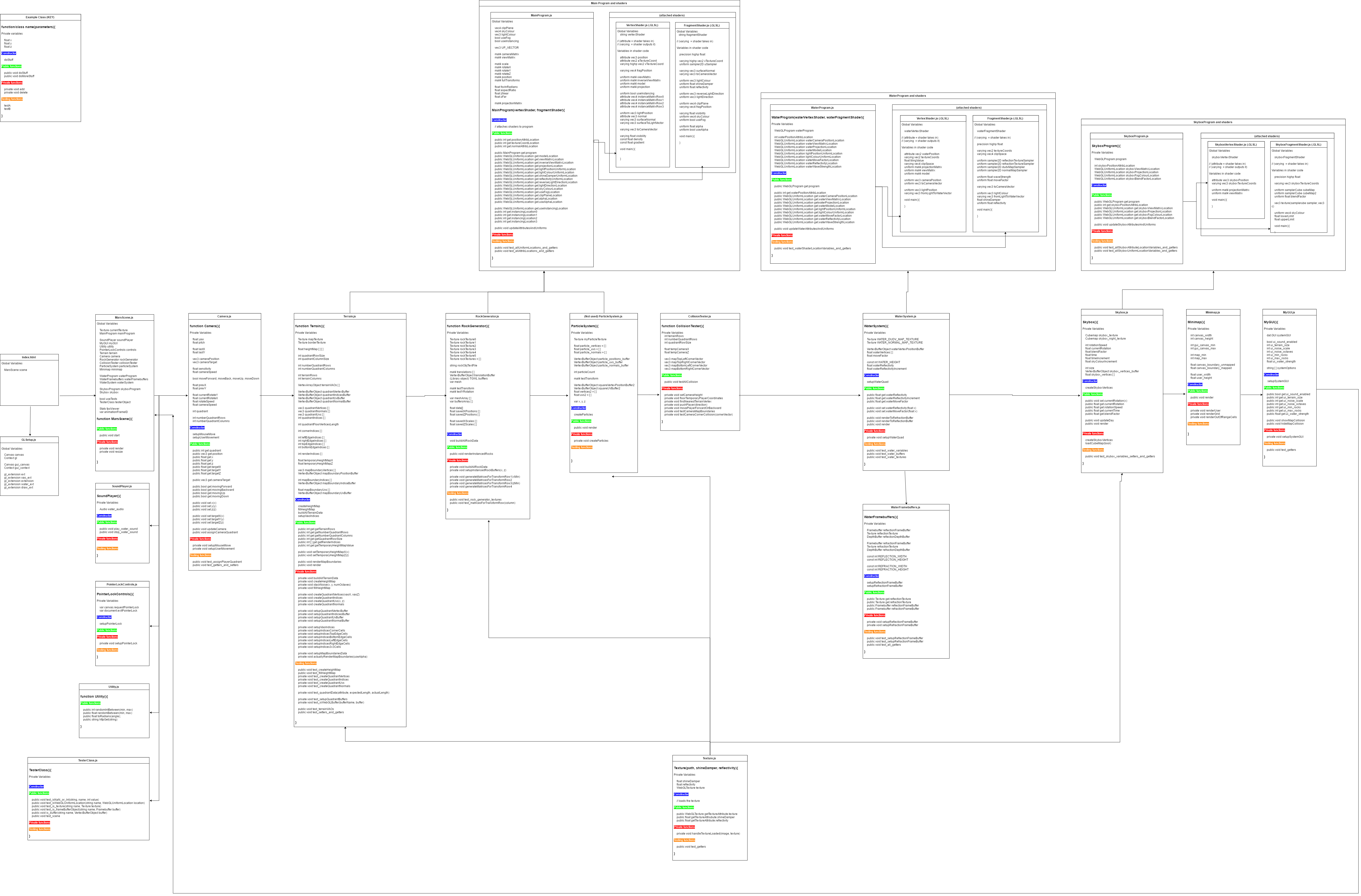
### Even More Detail

The key of my class diagram is defined in the top right of the image.

The important parts of each of my classes in the diagram are the setup and render methods.

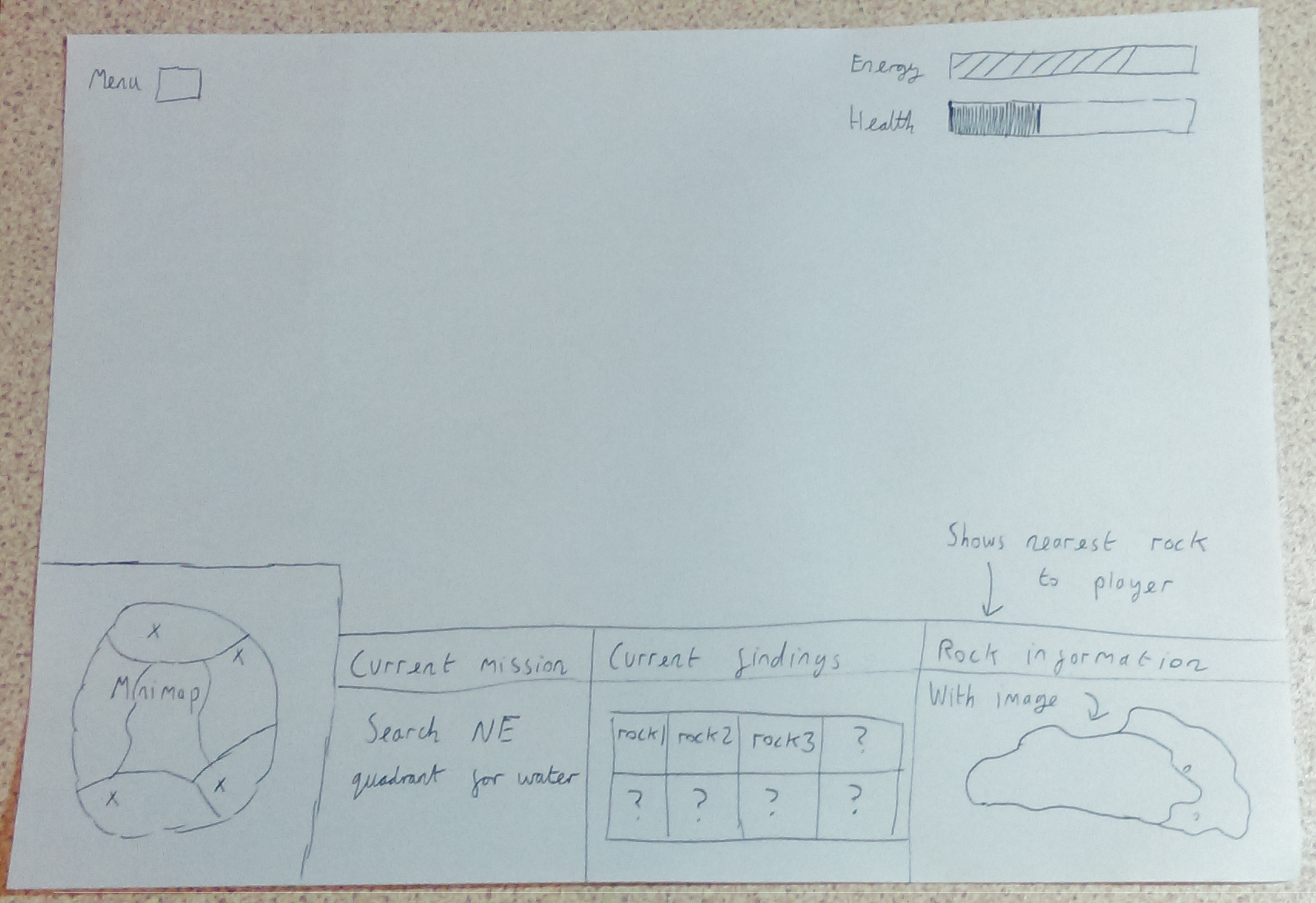
Instead of the regular JavaScript keyword var being used everywhere, it has been replaced with the actual descriptive data type. Using the var keyword in the initial class diagrams added nothing to it. Having the actual type is much easier to read.

Also the switching of the shaders used when rendering different parts of the scene.



## User Interface Design

Before the project direction changed from a game to a graphical scene various user interfaces where designed.



// SHOULD THE FULL IMAGE BE IN REPORT, OR JUST REFERENCE IT/THEM

## Other Relevant Sections

// terrain rendering design?

# Implementation

The implementation should look at any issues you encountered as you tried to implement your design. During the work, you might have found that elements of your design were unnecessary or overly complex; perhaps third party libraries were available that simplified some of the functions that you intended to implement. If things were easier in some areas, then how did you adapt your project to take account of your findings?

It is more likely that things were more complex than you first thought. In particular, were there any problems or difficulties that you found during implementation that you had to address? Did such problems simply delay you or were they more significant?

You can conclude this section by reviewing the end of the implementation stage against the planned requirements.

In the mid project demonstration, I realized I did not enjoy creating the game aspects of the project. There were 2 different things I was working 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 interesting. The games aspects also took so long to write and because of this, the graphics of the game suffered. Mid-way through the project I decided to strip out all the game aspects and just focus on the graphics.

// need to create a context to be able to talk to webgl, and get it tointeract with the hardware

// Show initial screenshot, in 1st week,

// say how didn’t know about camera matrix

Say about complete re-doing of the rock generation, as learnt something new – not ideal with scrum, but nothing can do about it

// talk about 2048x2048 grid, why so inefficnet? Realized shaders still have to process 4m vertices each frame just to discard them, needed to cut down number sent in the first place.

// talk about ticking off project deliverables, and how u kept adding new ones

// talk about stupid bug with the black lines on the terrain, say just removed normals

# Testing

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.

Provide information in the body of your report and the appendix to explain the testing that has been performed. How does this testing address the requirements and design for the project?

How comprehensive is the testing within the constraints of the project? Are you testing the normal working behaviour? Are you testing the exceptional behaviour, e.g. error conditions? Are you testing security issues if they are relevant for your project?

Have you tested your system on “real users”? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.

MAKE A THREE.JS SCENE WITH A BUNCH OF CUBES AND A FPS, COUNTER, COMPARE IT? COMPARE DATA BUILDING TIMES AS WELL!

How does this testing meet requirements and design of the project? Was valid in the test table..

## Overall Approach to Testing

All of the WebGL testing and monitoring tools are only for WebGL (v1.0). These cannot be used for this project as it uses various WebGL extension libraries.

## Testing the game

Up until sprint 5, the project was still a game. Testing these game features was relatively simple. Acceptance tests where written in a test table to see if the user could complete various tasks. After sprint 5, the game aspects where stripped out and now most of the test table tests where irrelevant. From sprint 5 onwards, unit tests needed to be written rather than the previous acceptance tests. As the majority of the test table no longer valid, there needed to be some other form of testing,

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

## Options for unit testing the project

The idea here is to keep the testing and implementation code as separate as possible. Not having this would make the code very difficult to read.

**Option 1: Make everything public**

By making everything public, you can access the objects properties in separate ‘Tester’ classes.Then you can completely separate the testing and implementation code. However having everything public is very dangerous, the public variables might get changed accidently and debugging would be a nightmare.

**Option 2: Duplicate entire project and have ‘test’ and ‘release’ versions**

This is similar to option 1, however the test version of the project would be separated. This ‘test’ version of the project would be the exactly the same as the release version, however it would have all variables and functions as public instead. The release version of the project would have regular public/private functions. However this would mean remaking the test version of the project every time the actual version is changed.

**Option 3: Add a setter/getter for all private variables**

In terms of danger, this is one step better than making everything public, however it bloats the code even more. This would allow you to create objects in separate class and use the class’s setters/getters to test it.

**Option 4: Have implementation and testing code within the same class, split up into functions.**

In the terrain class there would be a VertexBufferObject test, seeing if a buffer was initialized properly. However in the RockGenerator class the same test was needed, therefore it would be duplicated.

**Option 5: Use a Tester class having generic testing functions, and also have specific testing functions 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 a generic testing function, so the testing code is split between the implementation file, and the generic testing file. This was the best option to use for the project and therefore it was chosen.

## Things that cannot be tested

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

This is essentially testing the WebGL implementation itself, rather than my application. This is the same as writing a C function, and testing that the compiler converts it into the correct assembly language calls. It is simply out of the scope of the project.

A way to actually test the rendering output is by writing an implementation of the WebGL API. This would show all the calls made to the graphics hardware are correct. This is obviously out the scope of the project.

## Things that can be tested

* The data I give to WebGL/my shaders to get displayed on the screen.
* Checking buffers were created properly
* Changing the UI does what it should, changing water strength for example
* User colliding with terrain/map boundaries
* Vertex creation, ensuring that the correct amount is made
* Cross browser, Firefox/Chrome testing
* The users browser, check it can run WebGL 2.0
* WebGL console errors
* Memory usage - See how rocks affect this
* FPS tests - see how rocks effect this
* Other language/library comparisons, three.js/engines

## Automated Testing

The automated tests run every time someone starts the scene. The user 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 isn’t necessary.

### Unit Tests

As discussed in the options for unit testing section, 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.

//give code example

### User Interface Testing

The user interface, created with the dat.gui library [] was tested via the test table.

### Stress Testing

// show high rock count, big terrain size,

// load times

// memory

// give screenshots of the project when loads of rocks etc, even change max limit to 10k, see what FPS you get,

// SHOW SCREENSHOTS OF THE 50K INSTANCED RENDERED ROCKS

### Other Types of Testing

// SHOW TERRAIN RENDERING CODE FROM MINIMAP WORKING

// explain minimap was a game aspect, removed it for scene, but useful to show terrain render code working

## Integration Testing

// rendering components in order, terrain to generate rock render indices

## User Testing

The scene was tested on a few users. This was to ensure the controls and usability of the project was natural. The feedback I received from users was that the fog needed to be weaker and the controls were way too sensitive. The project was updated to reflect this feedback.

# Critical Evaluation

Examiners expect to find in your dissertation a section addressing such questions as:

* Were the requirements correctly identified?
* Were the design decisions correct?
* Could a more suitable set of tools have been chosen?
* How well did the software meet the needs of those who were expecting to use it?
* How well were any other project aims achieved?
* If you were starting again, what would you do differently?

Other questions can be addressed as appropriate for a project.

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

In the latter stages of the module, we will discuss the evaluation. That will probably be around week 9, although that differs each year.

// had to keep going back and improving things, this isn’t what SCRUM is about. However I couldn’t do anything about this, didn’t forsee the FPS issues, only realized FPS suffered once I implemented it.

// look into some debugging ide, or use directX for useful tools

// horrible amounts of time starting at unhelpful webgl errors, offscreen erros in attriubte 0

// Midway through the project I realized Webgl not properly documented, few tutorials for advanced features, no tools to help debug

// evaluate project deliverables, say what and when they were achieved

//couldn’t forsee code not rendering efficiently, meant I had to change it terrain/rocks

//water edges

// Make sun a spotlight for nice specular!

//make minimap and collision boundaries not awful

# Appendices

The appendices are for additional content that is useful to support the discussion in the report. It is material that is not necessarily needed in the body of the report, but its inclusion in the appendices makes it easy to access.

For example, if you have developed a Design Specification document as part of a plan-driven approach for the project, then it would be appropriate to include that document as an appendix. In the body of your report you would highlight the most interesting aspects of the design, referring your reader to the full specification for further detail.

If you have taken an agile approach to developing the project, then you may be less likely to have developed a full requirements specification. Perhaps you use stories to keep track of the functionality and the ’future conversations’. It might not be relevant to include all of those in the body of your report. Instead, you might include those in an appendix.

There is a balance to be struck between what is relevant to include in the body of your report and whether additional supporting evidence is appropriate in the appendices. Speak to your supervisor or the module coordinator if you have questions about this.

* 1. Third-Party Code and Libraries

**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.

**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.

**Perlin library** – This contains the useful perlin functions to generate the terrain. The techniques come from Ken Perlin in his paper about perlin noise. This library was used without modification.

**Webgl-obj-loader library** – This library has been used to load in the singular OBJ rock used for my scene. This library is open source licensed under MIT and was used without modification.

**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.

**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.

**Stats.js library** – 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 useage. This library was used without modification.

* 1. Ethics Submission

This appendix includes a copy of the ethics submission for the project. After you have completed your Ethics submission, you will receive a PDF with a summary of the comments. That document should be embedded in this report, either as images, an embedded PDF or as copied text. The content should also include the Ethics Application Number that you receive.

* 1. Code Samples

This is an example appendix. Include as many appendices as you need. The appendices do not count towards the overall word count for the report.

For some projects, it might be relevant to include some code extracts in an appendix. You are not expected to put all of your code here - the correct place for all of your code is in the technical submission that is made in addition to the Final Report. However, if there are some notable aspects of the code that you discuss, including that in an appendix might be useful to make it easier for your readers to access.

As a general guide, if you are discussing short extracts of code then you are advised to include such code in the body of the report. If there is a longer extract that is relevant, then you might include it as shown in the following section.

Only include code in the appendix if that code is discussed and referred to in the body of the report.

Random Number Generator

The Bayes Durham Shuffle ensures that the pseudo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs.

// Some example code here…

From Terrain.js, the createQuadrantIndices function. Answer with 11 upvotes.

<http://stackoverflow.com/questions/5915753/generate-a-plane-with-triangle-strips>

Creates triangle strip indices from a set of vertices.

From MarsScene.js, the resize function. https://webglfundamentals.org/webgl/lessons/webgl-resizing-the-canvas.htm

THE TERRAIN INDICES FUNCITON, AND U ALSO USED IT SOMEWHERE ELSE!

The WATER TUTORIAL SETS, ?

# Annotated Bibliography

[]

[]

[]

[]

[]

[] Old test table

Major Project/documentation/Testing/Old test table.docx

[] New test table

Major Project/documentation/Testing/New test table.docx

[] 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](http://dx.doi.org/10.1145/566570.566636)  Key: Per02

Two deficiencies in the original Noise algorithm are corrected: second order interpolation discontinuity and unoptimal gradient computation. With these defects corrected, Noise both looks better and runs faster. The latter change also makes it easier to define a uniform mathematical reference standard.

[] JQuery – Available at:

<https://jquery.com/>

Source available at:

<https://github.com/jquery/jquery>

Accessed: 30th April 2017.

[] m4.js

(Remove space between webgl- and fundamentals in url):

Available at:

<https://github.com/greggman/webgl-> fundamentals/blob/master/webgl/resources/m4.js

Accessed: 30th April 2017.

[] perlin.js (ISC licensed) – Available at:

<https://github.com/josephg/noisejs>

Accessed: 30th April 2017.

[] Webgl-obj-loader (MIT licensed) – Available at:

https://github.com/frenchtoast747/webgl-obj-loader

Accessed: 30th April 2017.

[] TWGL.js library – Available at:

<https://github.com/greggman/twgl.js/>

Accessed: 30th April 2017.

[] Stats.js (MIT licensed) – Available at:

<https://github.com/mrdoob/stats.js/>

Accessed: 30th April 2017.

[] Dat.gui.js – Available at: <https://github.com/dataarts/dat.gui/blob/master/LICENSE>

Accessed: 30th April 2017.