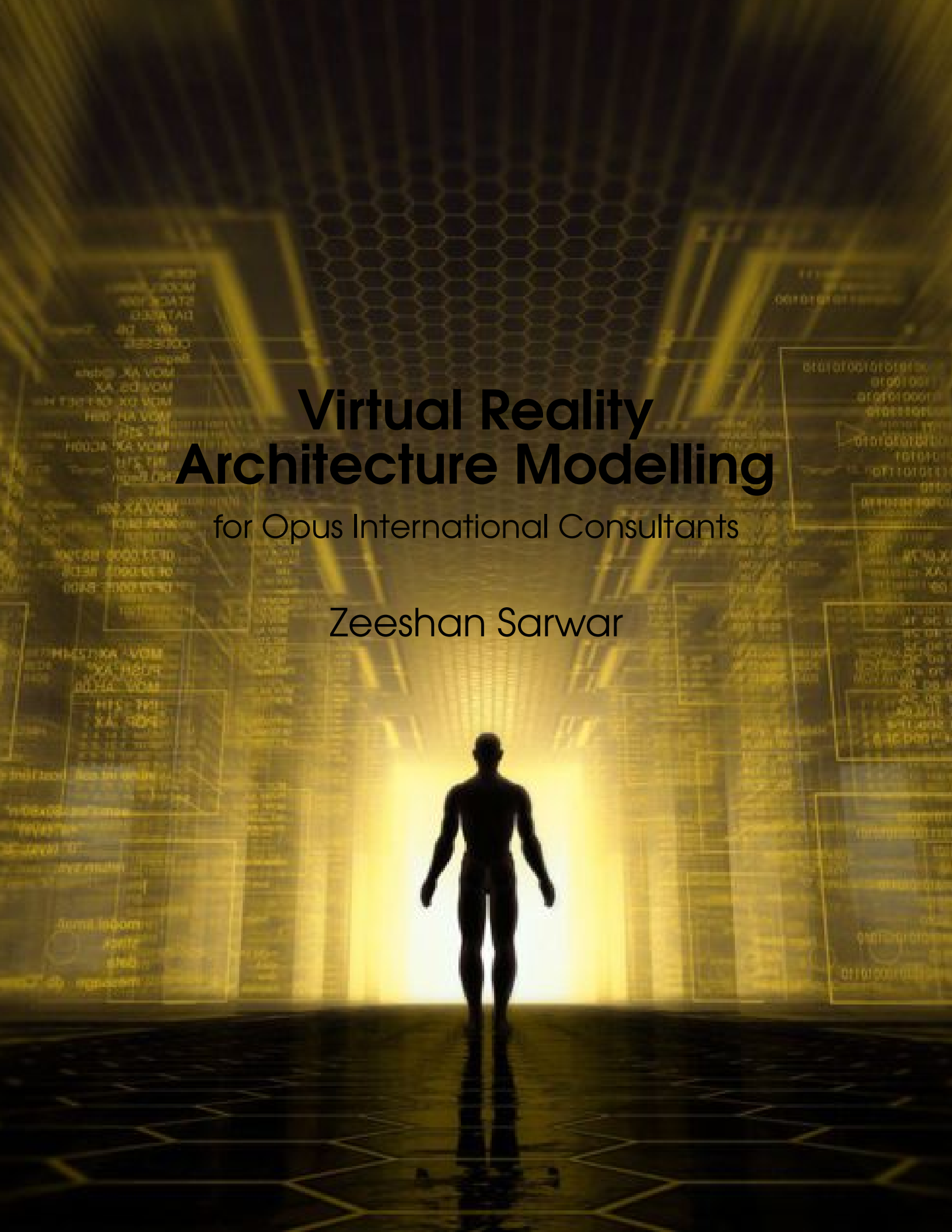


Virtual Reality Architecture Modelling

for Opus International Consultants

Zeeshan Sarwar

Abstract

Virtual Reality is an emerging technology which replicates a real environment or an imaginary setting. VR has shown to have great potential in delivering immersive experiences for its users.

Opus International Consultants plans to develop a web platform which will allow for VR walkthroughs of 3D architectural models. Opus wants to use utilize cutting edge Virtual Reality (VR) technologies as an immersive medium through which they can showcase the 3D models to their clients. The 3D content will be hosted onto a server which that will enable their clients to view architecture models on a cheap VR system such as the Google Cardboard. In this project we propose a novel WebVR architecture through which Opus will host 3D VR walkthroughs or their clients.

The project is part of my full year BTech451 degree at the University of Auckland. The project was initialized in March 2016 and completed in November 2016. This is a yearlong project divided into two distinct sections. The first section was primarily focused on the research of Virtual Reality (VR) - this includes: VR systems, VR applications, VR use cases and WebVR. Additionally, this part was also devoted to identifying the optimal methodology to implement a web based virtual reality system for Opus. The second section is devoted to developing and implementing a WebVR solution based on the research of the first section.

The final solution must be user-friendly for Opus and their clients. This means that the Opus VR team must be able to utilize the program/script without any difficulty and pre-requisite knowledge. Opus's clients should be able to visit the website and view/walkthrough the models intuitively. Taking these factors into account, the solution is primarily implemented in Javascript (and HTML). The WebVR modelling architecture is developed using the ThreeJS graphics library which abstracts much of the complex WebGL code while still maintaining performance. Such an architecture allows users/clients to perform VR walkthroughs on their smartphones using the Google Cardboard without having to download any plugins.

We discuss the results of the proposed WebVR architecture by posing two important research questions which address the viability and success of the solution. According to basic testing, the WebVR solution can allow clients to successfully perform VR walkthroughs (on mobile devices) in small 3D models which have a low polygon-count. The quality of the walkthrough experience deteriorates when using larger 3D models/structures due to their inherently high polygon-count. Results are significantly better on PC (compared to mobile devices) due to superior hardware and thus the architecture is very viable for PC (non-VR) walkthroughs.

BTECH 451, UNIVERSITY OF AUCKLAND

This project was undertaken under the supervision of Dr Aniket Mahanti from the University of Auckland and Sam White from Opus International Consultants between March and November of 2016 . *Second release, November 2016*

Contents

1 Introduction 6

1.1 Goals 6

1.2 Research Questions 7

1.3 Contributions 7

1.3.1 Novelty 8

1.3.2 Advancing state of art 8

1.3.3 For Opus 8

1.4 The Company 9

1.4.1 The VR Team 9

1.4.2 Opus's interests in VR 9

2 Virtual Reality 11

2.1 Brief History of Virtual Reality 12

2.2 How Does Virtual Reality work 14

2.3 VR devices 15

2.4 Opus VR device(s) 18

2.5 VR Applications 19

3 Literature Overview 21

3.1 Virtual Reality 21

3.2 WebGL 22
3.3 ThreeJS 22

4 Opus and WebVR ... 24
4.1 WebVR 24
4.2 Opus Objectives 24
4.2.1 The Current Procedure ... 25
4.2.2 The Desired Procedure ... 26
4.3 My Objectives 26
4.3.1 Motivation ... 27

5 Methodology ... 29
5.1 Web Technologies 29
5.1.1 HTML ... 29
5.1.2 canvas ... 29
5.1.3 CSS ... 30
5.2 javascript 31
5.2.1 WebGL ... 31
5.3 Prototyping 34
5.3.1 Workflow ... 34
5.3.2 3D graphics - basics ... 35
5.4 Google SketchUp 37
5.5 Unity 38
5.5.1 Unity WebGL compatibility Issue ... 41
5.6 ThreeJS 42
5.7 WebVR solution implementation (ThreeJS) 43
5.7.1 1 - Setting up the a basic WebGL scene ... 43
5.7.2 2 - Load a model (collada file) into the scene ... 45
5.7.3 3 - First person camera controls ... 47
5.7.4 4 - VR smartphone compatibility ... 61
5.7.5 5 - VR controls ... 64
5.7.6 6 - GUI controls (online scene modification capabilities) ... 65
5.7.7 EXTRA: Performance statistics ... 73

6 Limitations ... 75

7 Discussion ... 81

8 Potential Improvements 83

9 Conclusion 87

9.1 Significance for Opus 88

10 Future Work 89

1. Introduction

3D virtual reality walkthroughs allow designers and technicians the ability to gain valuable new perspectives of their design that may not be otherwise apparent. Via the use of VR, company staff and clients can explore and experience their design in full scale by being fully immersed in the virtual 3D environment.

This offers a number of opportunities including:

- Ability to visualize the completed construction of infrastructure including complex details which are much more difficult to visualize on a 2D screen
- Better understanding of the design by users, this will allow them to improve their current design by making much more informed decisions due to being able to visualize the final product (in 3D) before devoting resources to construction

1.1 Goals

The final goal of this project is to develop a graphical web based solution that can assist Opus in hosting 3D models in VR format which can be viewed using a VR headset - such as Google Cardboard. This will allow staff and clients to be able to view the VR content from anywhere in the world (assuming they have a VR headset Google cardboard, Samsung VR, Oculus rift) and/or a compatible smartphone. Recent advancements of browser based graphics APIS (such as WebGL) can render complex 3D graphics on webpages without the use of any plugins. Thus, for this project, the solution will be developed in WebGL so that users can visit the webpage without needing to download any proprietary software (plugins). The final product will require clients to only possess a smartphone (that runs Chrome) and a cheap Google Cardboard headset to view and walkthrough the 3D models. WebGL is a javascript API, thus the solution will be a implemented in Javascript using an external graphics library - ThreeJS. ThreeJS is another API which works on top of WebGL by abstracting away much of the complex WebGL code (shaders and objects) while still

maintaining performance. Using this library will significantly decrease development time while also increasing code readability for potential future developers (who may wish to continue/modify this code). The initial solution will be a website that will allow users/clients to walkthrough 3D models in first person via mouse and keyboard (on PC); we will then extend this platform to support VR (compatibility with smartphones and Google Cardboard).

1.2 Research Questions

WebVR is basically VR on WebGL; released in March 2011, WebGL is a recently new web graphics API which is a javascript port of the original C/C++ OpenGL library. Still in its infancy, there will be issues in this technology which can hinder or constrain the development of the *perfect* WebVR solution. To measure the effectiveness and/or viability of the proposed WebVR solution, the following research questions arise:

- **Will the initial WebGL solution be successful in allowing users/clients to perform pluginless 3D walkthroughs of models on the PC platform?**
- **Will the extension of the WebGL solution to VR (WebVR) on smartphones (and Google Cardboard) be viable for users to perform 3D VR walkthroughs?**

The first research question will allow us to comment on the effectiveness on the implementation of the first basic WebGL solution. We need a bare-bones pluginless solution which is effective in providing virtual walkthroughs for clients from the freedom of their homes (on their PC's). A successful WebGL PC based solution should be able to provide walkthroughs of 3D models in atleast 60FPS without requiring users to download any plugins. The controls should be intuitive and feel like a *First person shooter game*. Then we must extend the initial PC based implementation to support VR on smartphones. A viable WebVR solution should allow clients to perform 3D virtual walkthroughs on their smartphones when wearing the Google Cardboard headset. The WebVR scene should preferably run at 60 FPS in (and in first person), but however due to the infancy of the technology, this may not be possible depending on the complexity of the loaded model. Thus viability will be the main concern, a viable WebVR solution is one that can allow clients to VR walkthroughs in atleast 30 FPS.

1.3 Contributions

Through taking this project with Opus International Consultants, I have managed to develop a Web based platform which can allow users to walkthrough models and/or entities within a ThreeJS scene. Additionally, I have successfully extended the capabilities of this web based platform to integrate VR. Such novel VR capabilities allows users to walkthrough 3D architectural models in first person via smartphone and a cheap Google Cardboard headset. The use of VR greatly enhances user experience (walkthrough) within the scene, it delivers a more immersive experience which gives the user a better insight on the scale of objects/models and entities within the scene.

Additionally, I have also implemented GUI controls which allow users to load in custom models and adjust (fine-tune) the parameters of these models and other entities within the

scene (more details in section 4.6). The integration of these real-time GUI controls allows for extensive control over the environment, this can be used to produce highly customized and self-tailored walkthrough experiences for users (especially in VR).

1.3.1 Novelty

The novelty of the solution lies in the fact that I have successfully integrated first person controls, custom model loading (and GUI controls), physics and also virtual reality in one project (ThreeJS based). An ability to perform VR walkthroughs on the Web just by having a smartphone and a cheap Google Cardboard is a very powerful tool for conveying visual information to users. Additionally, there are no current solutions which allow users to perform web based VR walkthroughs (most likely due to the infancy of the technologies). This research can be utilized by future researchers to advance the current state of technology (WebGL, WebVR or the ThreeJS library). Additionally, my implementation of this WebVR solution can be used as a stepping-stone by future developers in their implementation of an enhanced and more immersive web based platform.

1.3.2 Advancing state of art

My solution also advances the current state of art since Im incorporating multiple technologies within this solution. In my solution, Im integrating model loaders, open source web based GUI controls, first person mouse controls (and keyboard controls), physics and collision detection (via ray casting), device orientation and VR controls. No example or solution currently exists on the ThreeJS examples page (or on Google) which seamlessly integrates all these technologies/implementations. By potentially uploading a stripped-down version of my solution on the ThreeJS example page, or open sourcing my project; this will advance the current state of the ThreeJS library and could potentially stir great interest on the capabilities of this graphics library (which will allow the library to progress).

1.3.3 For Opus

For the purposes of Opus, this platform can allow their clients to view and walkthrough 3D models from the freedom of their homes (provided that they have a smartphone and the cheap Google Cardboard headset for VR mode). Thus allowing clients to make well informed decisions on their architectural models.

For example, in the case of an architect (client), they can upload their CAD file (which represents the structure of interest) on the web (or send it to Opus VR team) and then intuitively walk through that model on PC in first person. To get a better sense of scale, they could walk through that model in VR (using their smartphone) too. From the walkthrough experience, the architect can better fine tune the parameters of the actual building/structure of interest, thus allowing them to make a well-informed decision whether to proceed to construction or not.

1.4 The Company

Opus International Consultants is a leading multi-disciplinary infrastructure with local reach and global connections. Opus operates in 5 major markets - Australia, Canada, New Zealand, the United Kingdom and the United States of America.[1] Opus has over 3,000 engineers, designers, planners, researchers, advisers and work with more than 12,000 clients.[1]

Opus offer fully integrated asset development and management services at all phases of the lifecycle including concept development, planning, detailed design, procurement, construction, commissioning, operation, maintenance, rehabilitation and upgrading. In New Zealand, Opus operates from a network of 40 offices and employs over 1,700 staff which provide services on leading infrastructure projects for both the public and private sectors.[1]

1.4.1 The VR Team

Opus is a very large company so it assigns teams/groups that perform niche roles to cater for clients. A specific VR team was assigned the task to produce VR solution(s) to for clients interested in modelling 3D architecture and design. I was very privileged to work with the Opus VR team since VR is an emerging technology which has great potential. The VR team consists of Sam, Andrew, James, Kodie and me (Zeeshan). Kodie has no direct involvement in VR but is rather leading the website development team that will be working alongside the VR team to host the VR solutions for their clients.

1.4.2 Opus's interests in VR

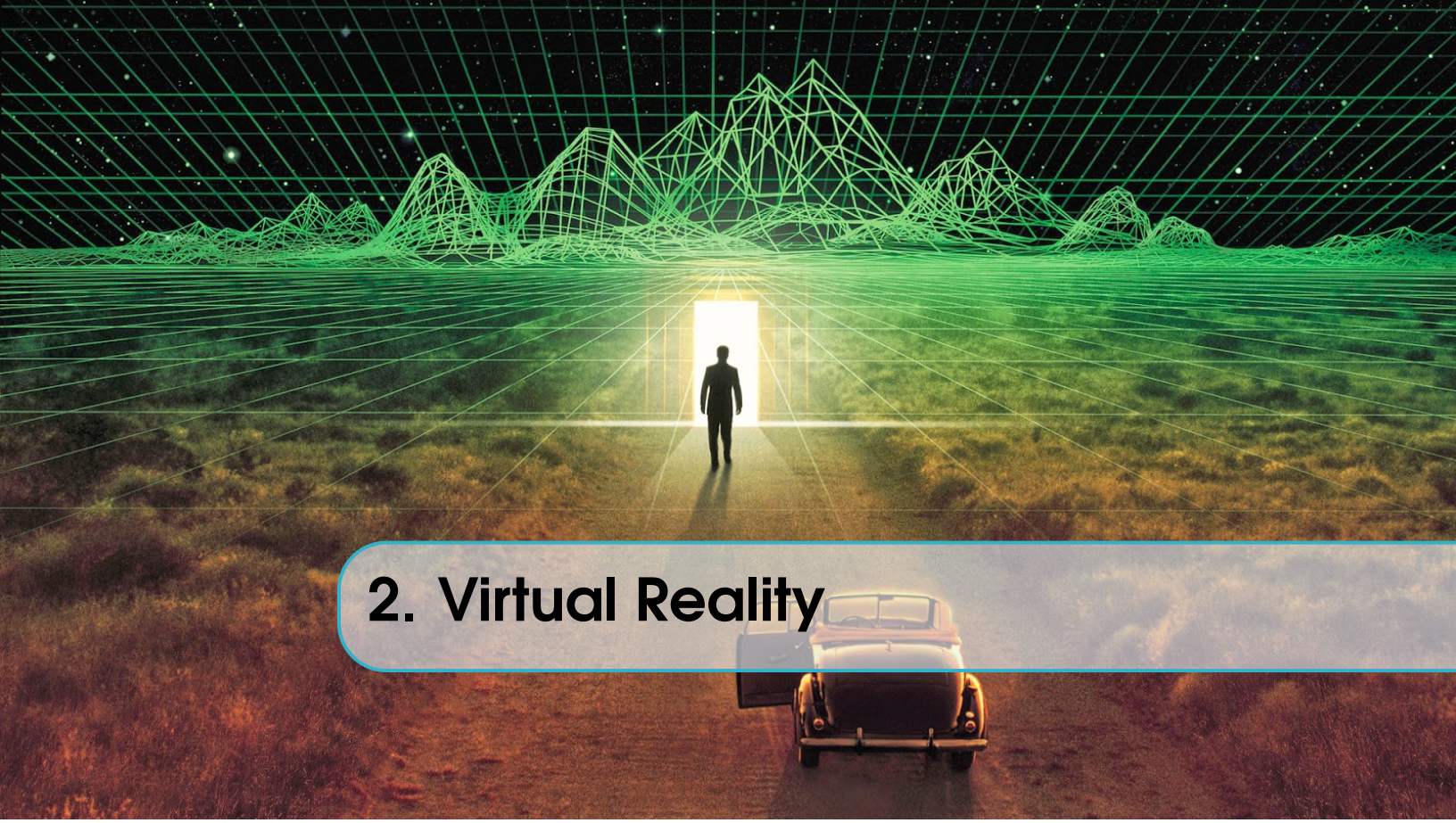
Opus International consultants are interested in VR because they believe it is important to utilise and leverage new technologies to gain and maintain competitive advantages in the industries in which the company operates. In addition to that, Opus-VR is a low cost, low risk project to explore the use of virtual reality technologies throughout all departments. The Opus VR team aims to achieve:

- Improved design quality and efficiency
- Better client experience and reputation
- Promotion of Opus

To understand Opus's interest in VR technologies we must first understand what is Virtual Reality.

The first part of this project (this report) will have general information on Virtual Reality, this is because Opus instructed me to conduct research on VR before delving into producing code/solution. Opus wants me to identify the best possible VR solution to cater for clients. To find such a '*perfect*' solution it is essential that I perform research on VR (as a general topic) because it is possible that the solution may not lie in WebVR at all (due to technological limitations) and we may have to resort to some other form of VR solution (Android/IOS app). My research and findings on VR will be presented in the following report; in addition to that I will also prototype a WebVR solution on which I can build upon in the next semester.

As part of my research, it is my duty to first introduce what Virtual Reality is before getting into the details of Opus and VR.



2. Virtual Reality

Virtual reality is the term used to describe three-dimensional, computer generated environment which can be explored and interacted by a user (person). It is a form of immersive multimedia in which physical presence is simulated in virtual environments. This is generally achieved through the use of a head mounted display in which a stereoscopic image is rendered, giving the illusion of depth. Motion tracking enhances the simulation, allowing the user to look or move around the virtual environment by turning their head or moving their body.

Virtual reality is generally associated with video gaming, but is currently being used for a wide range of applications including engineering and architecture. Significant developments have taken place over the past few years into virtual reality technologies; developments in a number of fields including display technology, computing power and motion tracking have significantly enhanced user experience. The technological improvements in VR has significantly reduced the price of virtual reality devices while also increasing their performance.

When using a VR headset, a user becomes part of this virtual world or is immersed within this environment and whilst there, they are able to manipulate objects or perform a series of actions. Actions such as head movements trigger a response in the headset which causes the scene to move/rotate relative to your head - using gyroscopes and accelerometers. Virtual Reality is primarily experienced through two of the five senses - sight and sound.

2.1 Brief History of Virtual Reality

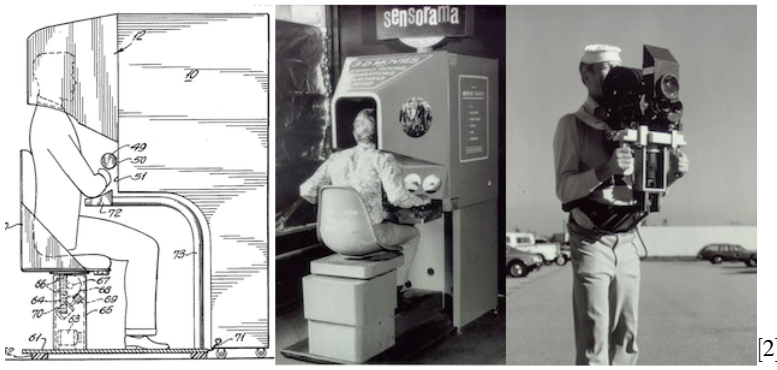
- 1938 - Stereoscopic photos and viewers: Initially VR originated from stereoscopic images. A pair of stereoscopic images are images which are angled in such a way to provide an illusion of depth - more on this later.

[2]

- 1920: - Flight simulator: The first flight simulator produced by Edwin Link was designed to train novice pilots.

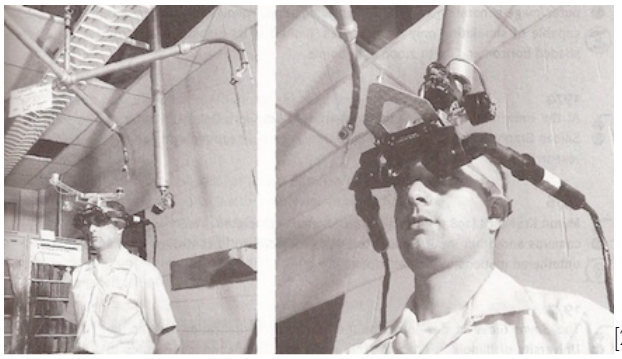

Left: Edward Link, Right: The Link Trainer [2]

- 1957 - Sensorama: The Sensorama produced by Morton Heilig was the first interactive theatre experience with stereoscopic images, oscillating fans, audio output (speakers) and also devices which emitted smell.



[2]

- 1968 - Head Mounted Displays: In 1968 Ivan Sutherland introduced one of the first head mounted displays which attached to a computer. They enabled the user to see virtual world - in wireframe format. However had many problems such as being too heavy thus required suspension.

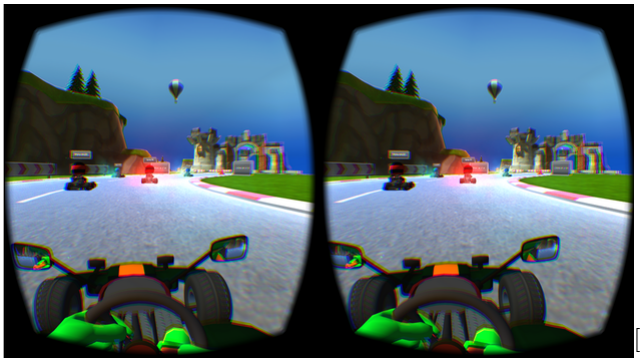

[2]

- 1970 - Interactive Map: The first interactive map was created by researchers at MIT. This enabled people to virtually walk through the town of Aspen.
- 1990's - Introduction of VR gear: This is when the term *Virtual Reality* was adopted and picked up by the media. VR was overhyped by Jaron Lanier and Tim Zimmerman to market the devices; however VR did not live up to expectations and people started losing interest. This is mainly because the technology was not ready yet - unable to provide full immersion.[2]
- 2012 to Present - A Kickstarter project introduced the Oculus Rift headset to the masses in 2012; this headset offered to fulfil previous promises which the previous headsets could not; and thus virtual reality was resurrected.[3] The term Virtual Reality is often repackaged as '*Virtual Environments*' since VR was associated with dissatisfaction. Due to substantial improvements in hardware and computational power Virtual Reality has gained consumer confidence and seems like the next frontier.

2.2 How Does Virtual Reality work

From the list above, we have seen how VR has changed over the years, with each iteration generally providing more immersion than the previous. But how does Virtual Reality actually work?

- A Split screen video feed is sent from the computer via HDMI into the headset. The video feed can also be sent through mobile display - for Google Cardboard and/or Gear VR. An example of the video feed is shown in the image below.


[4]

- The two slightly different views of the video feed are independently rendered. In a video feed, each frame acts as a stereoscopic image delivered to each eye.
- The lens in the headset enlarges each of the video feeds sent to each eye
- The video feeds reshape the overall frame delivered to each eye - by angling the two images. The stereo image pairs (frame-by-frame) are different to ordinary images since a stereo image (rectified) contains an element of depth. This effect mimics how we view the world which causes the immersion. An example of this is shown in the diagram below:

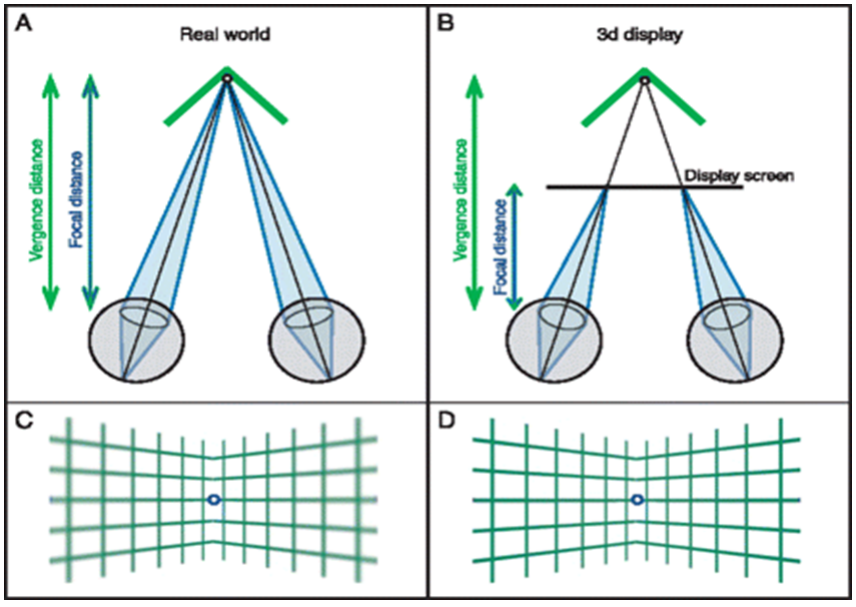


Fig-1: VR view vs reality view

As shown above, in ideal conditions Box-C should appear as Box-D; the eyes should not be able to distinguish between a virtual image and the world.
- The headset/phone renders the environment based on the desired field of view. It must be noted that a greater field of view requires more computational performance since a greater number of pixels being rendered; 180 degree FOV is recommended
- Movement of the head sends signals to the gyroscopes of the headset/phone which causes the scene to rotate in the opposite direction (of the user head rotation) - this mimics how we see the world
- To note - For the virtual environment to be convincing, a minimum of 60 fps is required to avoid stuttering, otherwise users may start to feel sick

Now we know how VR actually works, so what devices offer the best VR experience? The amount of immersion in VR is dependent on the VR device one uses. Generally the more expensive options offer better immersion with rich and detailed virtual environments while the other cheaper ones are usually used to introduce communities to VR (such as Google cardboard).

2.3 VR devices

There are many Virtual Reality devices in the market. Each VR device may offer a unique VR immersion experience, some more than others. There are four major players in the market: Google Cardboard, Samsung Gear VR, Oculus Rift and the HTC Vive.

Google Cardboard

Cardboard offers a *beginners virtual reality* experience. It has its own pros and cons; with the main pro being that its **cheap**, in fact the Cardboard is the cheapest VR option there is. At a price tag of $5, It's basically a cardboard dock with two lenses that holds a smartphone (for the display).[5] The docked smartphone must possess a gyroscope to function (synchronize with head movements). This makes the Cardboard extremely simple to use - since Cardboard is just a phone holder, thus potentially every modern cellphone is compatible with it. For VR content, the user requires access to the Google PlayStore (or AppStore). But now there are websites which are developed specifically for VR; thus a user can test out VR regardless of the phone they possess - via a WebVR website. This is also what Opus wishes to utilize; developing a web solution to showcase models in VR to their clients regardless of what smartphone they possess.

[8]

However Cardboard does come with its fair share of problems. Since each pixel is enlarged by the lenses of Cardboard, the experience is highly dependent on the smartphone display. The higher the smartphone resolution, the better the VR experience with cardboard. Also since Cardboard is just a dock, it is not specialized for any smartphone, but rather tries to cater to all smartphones. Due to this factor, there is no optimal positioning for any smartphone in the dock. Additionally, cardboard also lets in small amount of light which reduce can become a distraction and reduce the immersion factor. There is only one button for controls, this severely limits interactivity.[5] The phone must be removed from the cardboard each time you wish to launch a new VR application.[5] Due to all these factors Cardboard fails to provide the full-fledged feeling of VR immersion. With all these negatives however, Cardboard is still the most convenient way of getting VR to the masses; and I am privileged to be at the forefront of such a task.

Samsung Gear VR

The Gear VR on the other hand is Samsung's version of the Google Cardboard; it is specifically designed for Samsung's galaxy smartphones. At a price tag of $100, it is a much more expensive than the Google Cardboard; it's for those who have already invested in Samsung's ecosystem (devices).[5] The VR content is supplied primarily from Oculus store (which is now owned by Facebook), it combines software from Facebook's Oculus and downloaded apps created by a range of developers.[5] The screen resolution is the resolution of the inserted high end Galaxy smartphone - which is typically 1440x2560 pixels.

[9]

It has adjustable straps (unlike Google Cardboard) so the user does not have to hold the device while also having a scrolling wheel which allows a user to manually adjust the distance between the phone display and the lens. This allows users to view VR content for longer periods of time (compared to Google Cardboard). In addition to that, Gear VR also provides basic input such as directional pad, volume control and a dedicated back button. Users can also attach compatible game-pads (via Bluetooth) to further enhance their VR experience. This not only allows one to just view VR content, but also interact with the Virtual environment. The user does not need to take off the headset since the navigation such as switching between apps is done in VR mode. According to the VR team, they will be experimenting with this very soon and most possibly use it for prototyping on Samsung galaxy devices.

Oculus Rift

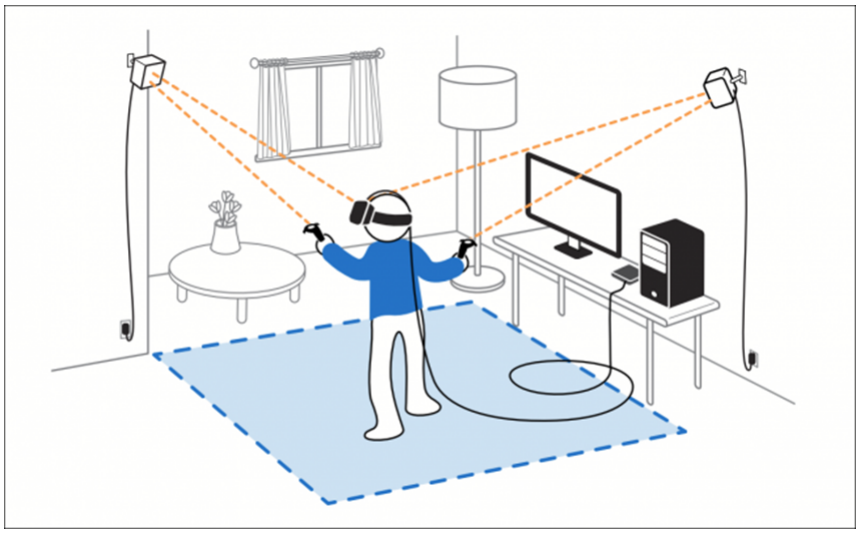
The Oculus Rift is the leading example of mainstream virtual reality technology. It features a low latency display and head tracking with support for a very wide range of applications which are run through a desktop or laptop computer. The Oculus rift is the original product which resurrected, re-introduced and to some extent re-hyped Virtual Reality. The Rift is a dedicated head mounted display (HMD) which connects to a PC via HDMI cable(s). It offers a high-end VR experience, though at a hefty price tag of $600.[6] The headset itself contains a gyroscope, accelerometer and magnetometer. It also offers head-tracking via an external sensor; this allows a user to lean in and out of virtual in the virtual environment - thus offering a greater field of depth than the Gear VR (and Cardboard). However due to the fact that it connects to a PC, the performance is directly proportional to the PC specs (mainly GPU); a reasonably high-end PC is required to for a good VR experience.[6] This means that user has to also pay for extra to up their PC specs. Also it is not as easy to setup as Gear VR and Cardboard since the user is required to download software and connect multiple cables.

[10]

This product is quite pricey for most users and this doesn't even account for the PC requirements. The Oculus is for those who will pay for no compromise on the VR experience. This is the primary VR headset used by the VR team to prototype client models on the unity and Revit Software.

HTC Vive

The final high end VR display on this list is the HTC Vive. This is a relatively new high-end HMD which comes at a price tag of $800.[6] This product is manufactured by HTC and Valve.[7] It has the most VR content then the other displays since VR content is supplied by Valve (Steam) themselves. It offers mostly everything what the Oculus offers plus more - such as two external proprietary controllers and *Laser-powered Light House Tracking technology*.[7] Multiple cameras are setup in a defined perimeter, this allows the user to interact with virtual objects within the perimeter and thus provides maximum immersion compared to the other products. An example of this is shown in the image below:



It must be noted that to use this device, the product requires a specific perimeter for the setup (atleast 2mx2m); users may not have this much space.[7]

The Oculus rift and HTC Vive are very similar in terms of hardware, there is however a subtle price difference and that is most likely due to the extra feature (Light house tacking) offered by the Vive.

2.4 Opus VR device(s)

The primary focus of Opus is distribution of VR content in the form of architectural modelling for clients. This means that the cheaper the device, the more likely their client will get/use it. Distribution of 3D models/content is the main focus of Opus, this is why they plan to use Google Cardboard as the primary platform for content distribution for clients. Google Cardboard is the cheapest VR device (not exactly a device but a dock) which is easy to buy in mass quantities and distribute to clients. If clients want a better quality of VR immersion then they are free to buy their own VR device which may offer a more realistic (e.g. higher resolution) view of their model.

When I went to Opus, they initially introduced me to VR through the Google cardboard. After the introduction phase, Opus then revealed the *real* VR product - the Oculus Rift.

Opus primarily uses Oculus rift for development of VR content (3D VR models); to present and distribute the 3D models (in VR), they use Google Cardboard. In addition to that, Opus also plans to use Samsung Gear VR for testing on Samsung Galaxy devices; this will be a new addition to their arsenal of VR headsets. But since not all Opus clients will be willing to pay much for an introductory VR experience; Google cardboard remains to be the best option for getting clients interested in the technology which they can try from the freedom of their homes with minimal expenditure.

2.5 VR Applications

Virtual Reality is a rapidly emerging field which holds many applications in any field. Some examples of the industries where Virtual Reality can be used extensively in are:

- **Military**: can be used and used to some extent in flight simulation, medic training and vehicle simulation.
- **Healthcare**: Virtual Reality surgery - reduces invasive procedures from doctors, allows doctors to practice surgical procedures before practically applying them on patients
- **Entertainment**: Main hype around VR is its uses in entertainment; this opens up a new world to gaming and movies (Netflix)

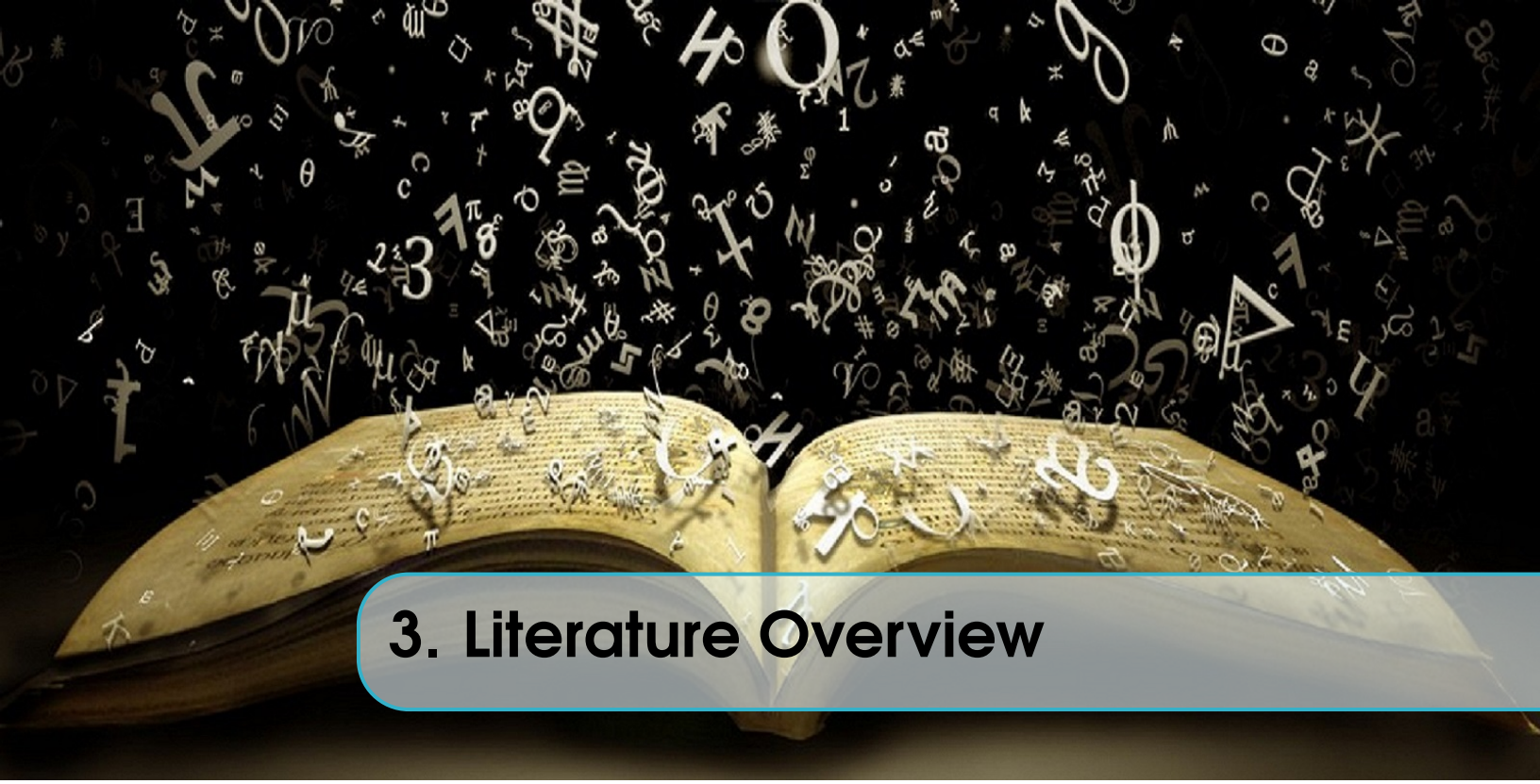

[11]

The above image is an example of full VR immersion via the use of a multi-directional treadmill and a proprietary controller (gun). The same concept can be used for military training.

- **Fashion**: In the Fashion industry VR can be used for 3D Modelling of wearable clothing
- **Engineering**: 3D modelling of general infrastructure and machinery such as bridges and vehicles etc.

- **Sport**: Can be used to provide a first person view into sports (via use of 360 cameras). This can be used extensively for analysis and identification of good technique. In addition to that, can be used as a vector for coaching new sportsman
- **Scientific Visualization**: A means of visually conveying complex 3D information. For example in biology it can be used to map the internals of the human body. In astrology can be used to produce a 3D map of space and the position of earth relative to other objects
- **Search & Rescue** - Virtual reality devices paired with remote controlled drones could be used to provide accessibility to hazardous environments while also improving safety by reducing exposure.
- **Construction** - Architecture modelling and visualization; this is the **main use case for Opus**

VR technology holds potential in every industry, these are just some of the industries where VR technology can be utilized. This is also the main reason why major companies are investing into VR technology (Google is one example).



3. Literature Overview

Developing a web based VR solution requires knowledge of various methodologies and technologies. To implement a successful solution requires researching previous literature which may provide the stepping-stones for initialization. Since the web platform will be incorporating virtual reality, WebGL and ThreeJS; it is vital to review literature within these respective fields of study.

3.1 Virtual Reality

Virtual reality can be used for many things, one paper of particular interest is using VR to allow museums to build and manage online exhibitions.[29] Content designers designed visualization templates of 3D model artefacts (of the exhibition) which were viewable in a VR based exhibition. Additionally, this exhibition was viewable on the internet.[29] This web based simulation fascinated me since one could portray a vast amount of information in first person through the internet.

Use cases such as this gave me motivation to work on an immersive web based VR platform which delivers a similar experience (VR walkthrough) to that of a VR exhibition. Such research also allows me to understand the possibilities of what a VR based solution can be used to implement; I can potentially extend the project to more than just having virtual walkthrough's. For example the implemented platform can potentially be enhanced to provide full simulations of virtual environments.

VR provides an insight and offers much more than just the ability to view and walk-through virtual environments. VR can also be used as a platform for training and improving certain skills. A study examined the impact of VR surgical simulation on improvement of psychomotor skills relevant to the performance of laparoscopic cholecystectomy.[30] The results concluded that surgeons who received VR simulator training showed significantly

greater improvement in performance than those in the control group and from this the report concluded that VR is therefore a valid tool for surgical training programmes[30].

Such research shows the true power of VR and how its capabilities can be extended to more than just a viewing/visualization tool. The findings are intriguing because I could potentially extend my WebVR application to incorporate training and skill improvement capabilities on top of just being a visualization/walkthrough platform.

3.2 WebGL

WebGL is a powerful platform which can be used to render very complex graphical data/visualizations with exceptional frame rates. Textures are materials which make models look realistic since they apply colour onto 3D models, however they come with their fair share of limitations - complex high quality textures can significantly reduce rendering times. A study had been conducted which researched whether it is possible to visualize large 3D volumes on mobile devices and WebGL.[31] The study showed that mobile devices (which run the WebGL) are able to successfully render large volumes of 3D textures.[31]

This discovery is very significant since my implementation of the WebVR platform heavily relies on the fact that users/clients are able to view complex 3D models (with potentially large textures) on their smartphones. The research proposes an algorithm which reduces constrains of texture size (allows larger textures to be rendered); this can be used as an improvement in my WebGL based solution since if I am constrained texture-wise. I can potentially utilize the algorithm to reduce the texture-size constrain and achieve a much more fluid and immersive scene (with higher FPS).

There are other studies which also support the fact that WebGL is a potent graphics library. Research has shown that 3D Web-based human machine interaction (HMI) running on WebGL has resulted in higher rendering performance compared to other alternatives.[32] The frame rate (FPS) and frame time metrics were used to evaluate the performance of the rendering system. A performance increase was observed when running the 3D GMI on WebGL in Internet explorer and chrome.[32] The report concluded that stuttering incurred less in the proposed 3D Web-based (WebGL) HMI compared to their chosen commercial HMI product.[32]

This implies that WebGL is a robust graphics library with unmatched performance, statistics like these motivated me to implement my solution utilizing this highly performant library.

3.3 ThreeJS

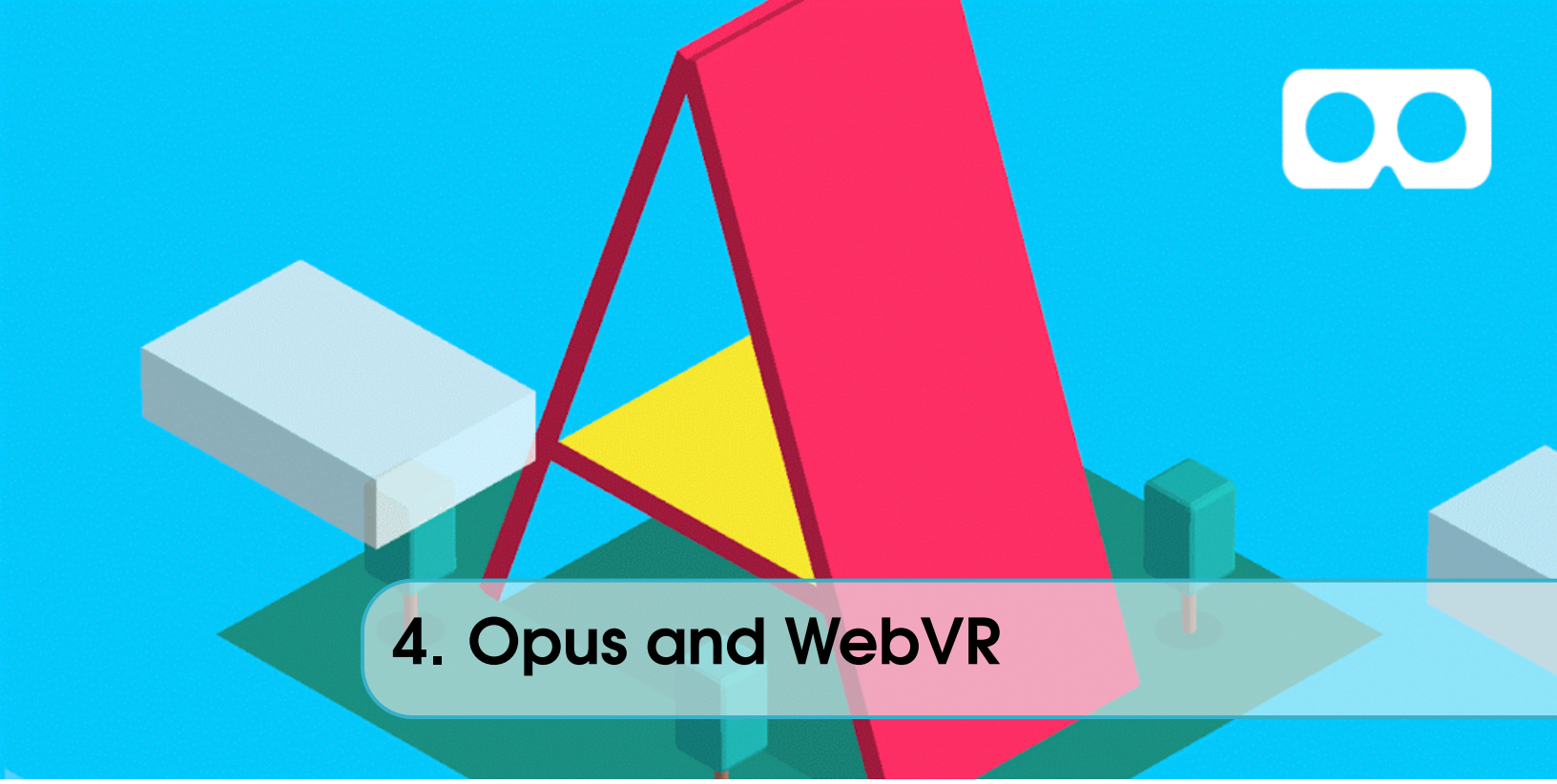
ThreeJS, according to some research has a very bright future ahead. Due to the fact that ThreeJS runs on WebGL which is a pluginless solution to render complex graphics on multiple platforms (PC, smartphones, tablets etc.), it is very convenient option for a gaming platform. Most online multiplayer games require an external client of some sort,

research had shown that ThreeJS with WebSocket can provide the same capabilities as those heavy client constrained games.[33] The results from the research show that ThreeJS with WebSocket based multiplayer online games can easily support the interaction of a small group of users.[33] With advances in ThreeJS and WebGL technologies, it may be possible to support large online multiplayer in the near future - which may completely reduce the requirement of having a game client.

Research such as this shows that ThreeJS holds tremendous potential; even at its infancy, it can support a small group of online users. This is important to my project and to Opus because in the future they plan to extend the capabilities of my WebVR solution to support multiple user walkthroughs. In such a type of walkthrough, multiple users should be able to see each others location within the walkthrough simulation (as sprites) in real time. Since my solution is implemented in ThreeJS, research has shown to prove that this is possible; this makes my project much more exciting and relevant in terms of incorporating clientless/pluginless multiplayer capabilities.

In another report ThreeJS (running over WebGL) has been used to visualise the space debris environment around the Earth.[34] The objective of this research was to reduce operator's effort in visualizing debris data which better helped them understand the situation. The researchers experienced a significant decrease in development time using the ThreeJS graphics library compared to building a traditional desktop application.[34] They also commented on major benefits of using such a method of implementation; such as no installation required (pluginless) for end users.[34] This is beneficial for operators and developers which makes distribution straightforward. Other benefits to ThreeJS included commercially friendly licence which significantly reduces risk of encountering commercial difficulties when running on top of existing software. Using the ThreeJS graphics library, they stated that the application user interface was highly responsive and gave a modern look and feel.[34]

Such positive reviews of the ThreeJS library as a successful online graphics rendering solution further convinced me to study the API. Upon further examination and research of this API, the details and ease of use motivated me to implement my web based solution in ThreeJS.



4. Opus and WebVR

We have briefly defined what VR is and also seen some of the applications which VR holds for different industries. Opus on the other hand wants to go one step further and that is utilizing WebVR.

4.1 WebVR

WebVR is an experimental javascript API which can be used to produce content for Virtual reality devices such as the Oculus Rift and Google Cardboard. WebVR makes heavy use of WebGL library which is also a recent web technology that is used to display graphically intensive content without the use of plugins. In simpler terms, WebVR is: making use of a stereoscopic camera and device orientation controls (accelerometer/gyroscope) in a scene rendered by WebGL. One could deduce an even simpler definition of WebVR; it is VR on top of the WebGL graphics framework in a web page.

4.2 Opus Objectives

Virtual Reality, as previously stated, has potential in any field. For Opus, they would like to utilize VR to provide clients with 3D virtual walkthroughs of virtual environment(s). By '*Virtual Environments*', we mean an architectural model proposed by the client themselves. A 3D virtual walkthrough would provide clients with an insight on how their design would look like (from first person perspective) before construction. The clients will then be better informed and thus could make improvements of their current model to obtain the *perfect solution* before proceeding to implementing it (construction). Opus commonly uses '*Computer Aided Design*' (CAD) software applications such as Revit, Sketchup, and 3DS Max to rapidly convert the CAD files to a format which are viewable in 3D through the Oculus Rift. CAD files are the format of models provided by their clients.

There are many benefits to using VR as a primary means of sharing 3D content. Two major factors are:

Client experience and communication of ideas

VR allows for clear and effective communication of ideas to clients. This is especially useful for less technical clients where conventional methods of presentation may not be as effective. A virtual walkthrough of a new road alignment (for example) could showcase the road design, bridge structures and landscape design, while giving clients a perspective of how the road will be experienced by its customers. The ability to deliver such a VR walkthroughs will also enhance the *wow factor* of Opus's work and has the potential to boost their reputation for delivering innovative and high-tech solutions.

Promotion

VR technology can also be used to promote Opus at schools, trade shows, conferences and other public events. Attendees could be given a tour of current projects that Opus is working on or perhaps experience a virtual earthquake which demonstrates innovative new structure designs. The VR team is working on such a project (in previous statement) and has managed to '*surprise*' attendees via presenting such a demonstration.

The VR team utilizes many resources for production of VR content to cater for their clients. These resources can be broken down to hardware and software requirements:

Hardware:

- Oculus Rift and Google Cardboard
- CAD-spec computer - for building models and presenting media
- Game controller to navigate virtual environments

Software:

- 3D modelling software - Revit, SketchUp, Unity and 3DSMax
- Model conversion software - primarily IrisVR

4.2.1 The Current Procedure

At the moment, clients must physically go to the Opus headquarters to test their architecture models. They give the model to the VR team which uses software to convert it to a format that is compatible with the Oculus rift (and/or Google Cardboard). Once the conversion process is complete, the client is able to view the 3D content through the VR headset.

This procedure requires the client to be physically present at Opus headquarters, thus the client is completely reliant on Opus to be able to view their own models (in VR). This is a time consuming procedure for both the client and Opus. What Opus wants to do is produce a WebVR solution; this means converting the model (provided by the client) into WebVR format and pushing it onto the web for the client to view through a VR headset (Google Cardboard). This will allows the client(s) to perform virtual walkthroughs of their models from the freedom of their home (rather than having to visit Opus).

4.2.2 The Desired Procedure

The procedure which they wish to implement for clients is as follows:

- A client sends their 3D model(s) to the Opus VR team
- The VR team converts their file (probably CAD) into VR format. In our case, this is the WebVR (WebGL with stereoscopic camera) format.
- Once the model is converted, Opus will post the new content onto the server into the client(s) account - cannot be public for privacy reasons
- The client will login to the Opus VR website, they will be able to view the VR content under their respective account provided they have a VR headset and/or a smartphone
- Note: Opus will provide Google Cardboard headset for free to their clients

Ideally Opus wishes to post WebVR content (compared to IOS/Android app) as it is compatible with potentially any smartphone. This is because for APK/IOS VR app, the user would have to download and install the program; WebVR is a multi-platform system which would allow the client to visit the URL and view the content instantly upon entering the web page without needing to download any plugins.

4.3 My Objectives

In the first semester, my task was to conduct research on virtual reality technologies. I had to research the VR headsets, their pros/cons, WebGL, WebVR, and other libraries such as ThreeJS. This is because it may even have been possible that there may there were established open source projects which were exactly what we (me and Opus) were looking for.

The difficulty of this development depends on what other developers in this field have already tried; since WebVR is a very recent and emerging field, there are not many resources available for me to utilize. Thus I failed to find any other projects similar to the one Im developing and I realized that I needed to develop one from scratch. Due to this, in the first semester I had to design an early proof of concept of how the final product should perform so I could start implementing early. In the second semester I must implement the design that

I had come up with. In this case I must implement a script that can load 3D collada models, display them on a webpage and allow a user to walkthrough the scene. Then I must extend the script to allow support for VR - which means implementing stereoscopic vision and device orientation controls within the scene (modifying camera and movement/interaction). Thus converting my WebGL scene into a WebVR one. The ideal product should be a script(s) which can be utilized by an html file to display the WebGL content on Opus's website. The resulting product should be user friendly - for Opus and their clients. The VR team should have control of the elements in the scene, they should be able to modify the parameters without delving into the code. Thus I must implement a GUI feature in addition to a load/save settings option so that Opus can customize each model/scene specifically to for each client.

4.3.1 Motivation

At the start of 2016 our course supervisor Dr. Manoharan Sathiamoorthy gave us a list of options for the possible BTech projects we could do this year. The first option (out of the 8 options) was the Virtual Reality showcase for Opus. I was interested in VR and VR modelling (game/content production) before the project options were even shown to me since VR appeared to be the next promising emerging technology. In my spare time, I would watch videos on YouTube on how VR worked, the devices, the entertainment and also how to produce basic scenes in VR (for game development). So naturally, I was inclined to choose the VR option for my project. A major factor which helped me select VR as my project option was the potential this technology had, it appeared to be '*the next big thing*'.

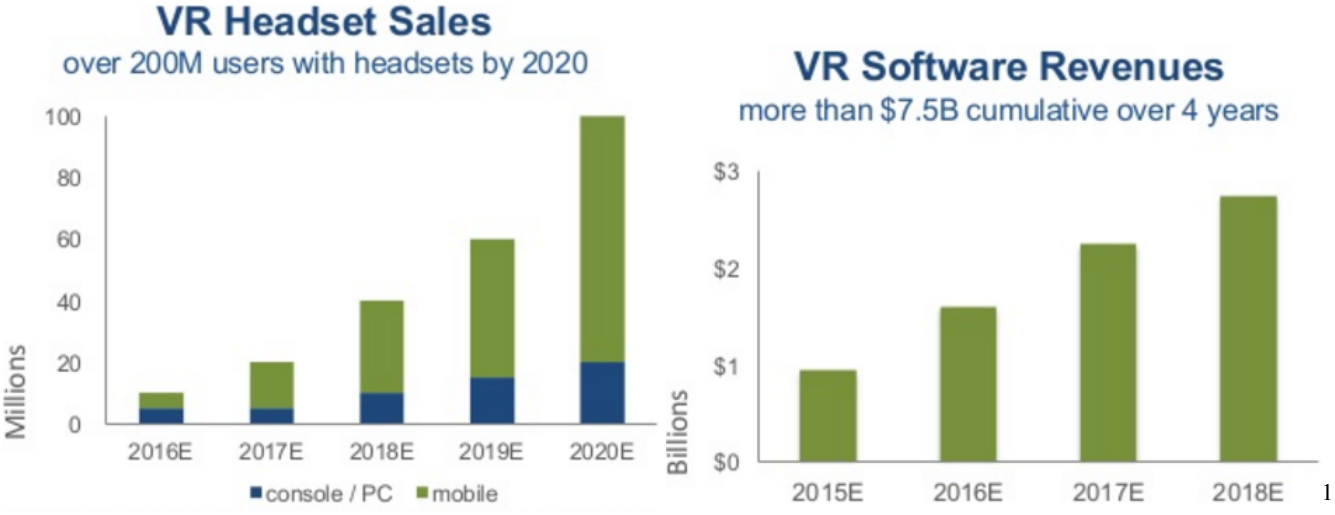
The Next Big Thing

It is not only me who believes that VR (and WebVR) may be the, many big companies such as Google, Facebook, HTC and even Sony believe that VR has tremendous potential. In fact many companies have also heavily invested in VR; Mark Zuckerberg, the owner of Facebook has bought Oculus for $2.2 billion in 2014.[12] In addition to that, eight of the top ten tech companies are invested in VR. These tech companies include:

- **Apple**: They do not reveal specific details on product developments, according to the report by Tim Bradshaw at *The Financial Times*, Apple has been building prototypes of possible headset configurations for several months".[13]
- **Samsung**: Samsung struck a strategic partnership with Oculus, this gave Samsung early access to Oculus software platform (Oculus store) for content. This partnership was also helped Samsung co-develop the Gear VR. It is poised to become one of the most owned headsets in the market due to its portability and quality of VR content delivery.[13]
- **Microsoft**: Microsoft has arguably the best research teams in Computer Vision, this is what enabled Microsoft to be the first to develop an Augmented Reality headset - the Microsoft HoloLens. Augmented reality delivers VR-like content on top of reality (somewhat analogous to Google Glass). Additionally the company offers Xbox One controller support for the Oculus rift.[13]
- **Google**: Google is one of the major players which has released the cheapest form of VR - the Google Cardboard. VR involvement has transformed from a "*20% project*" to a fill-on department in less than two years.[13] It has shipped more than 5 million headsets since inception; in addition to that the company is working on a VR version of the OS - Daydream, which will be released later this year.[13]
- **IBM**: IBM's cloud based *Watson platform* is one of the most technically advanced Artificial Intelligence system in the world.[13] This system is also beginning to power VR experience; this is important since this AI system could be used to bring unparalleled levels of depth and interaction of virtual characters (e.g. voice communication) in a VR environment.
- Intel: Intel is looking to use *RealSense* technology for mobile VR (like Samsung).[13] The company recently announced it will be releasing developer kit phones powered by this technology.[13]

- **HP**: This company has recently created as $999 "*VR Ready PC*".[13] This is a huge step for PC HDM's such as HTC Vive and the Oculus rift because they require high end PCs with specific requirement; with a PC like this, one would not need to buy all the required PC components since it would all come in 1 package.
- **Foxconn**: This company is the worlds largest electronics manufacturer; it has also worked previously to help develop zSpace which is an educational "*real world VR*" system.[13]

These 8 companies only represent a fraction of the whole VR ecosystem. Others such as Sony, LG, HTC and Facebook (who are not be in the top 10) are also making huge impacts on VR with their respective headset/VR-software launches.



[1]

Fig-2: VR future projection statistics

Analysing the stats above, we can clearly see that there is alot of potential for VR for the future. According to the estimations, VR is a big industry; by the end of this year, there will be atleast 10 million VR users around the world. We can also see that overtime, the mobile market will be dominant. Mobile VR is powered by smartphone, these offer portability which is always better than using VR in a static location. In addition to that smartphones get more technologically advanced (updated) in terms of specs every year; thus they will be able to support the demanding requirements of VR. An example of this is the new upcoming Android N, which is an android OS specifically built with a focus on VR.[14] Technological advances such as these really support mobile VR emerge as a successful technology which will make it more dominant than console/PC VR.

From these statistics and actions made by companies, we can clearly see the potential VR holds for the future. VR is proving to be a promising emerging technology which will have many use cases in the future; this is a major factor which highly motivates me in working with VR and WebVR technology (BTech project). My development practices can be useful in content production for the VR industry.

[1] http://www.slideshare.net/BDMIFund/the-emerging-virtual-reality-landscape-a-primer?next_slideshow=1

5. Methodology

The main focus of the first semester was to research on Virtual Reality and WebVR with minimal prototyping. Major prototyping of a proposed solution will occur in the second semester. The main research on VR and WebVR has already been covered in this report; now we will move onto prototyping. By prototyping I imply, the approach taken to implement a web based solution which will allow clients to perform VR walkthroughs in virtual environments (in 3D architecture models). Since the solution is web based - we must get accustomed with the web technologies required for implementation.

5.1 Web Technologies

Firstly to implement a WebVR solution for Opus, I have to become with familiar with certain web technologies such as basic HTML, javascript and CSS (in some cases - for code readability). These technologies are the basic building blocks required to make a basic WebVR scene which can be hosted on the web.

5.1.1 HTML

Hyper Text Markup Language (HTML) is the primary language used to display webpages. A set of markup tags describe the layout of the web document. Each tag emphasizes specific document content. With the introduction of HTML5, we are no longer limited to just drawing 2D rectangles on the screen. This is because of HTML5's **Canvas** API.

5.1.2 canvas

Initially developed by *Apple*, **canvas** is an HTML tag that can be used to draw complex graphics using javascript. [16] What canvas offers (that other tags don't offer) is that it allows us to address each pixel individually. In addition to that canvas also enables us to edit

images and video.[16] This significantly improves website performance since one does not need to download images from the network and can implement one within the document using canvas. However not all browsers (and versions) support the canvas tag, this is an important factor because we (Opus) want to cater for a maximum number of clients and thus browser compatibility is vital.

The supported browsers (and versions) are:

- Safari 2.0+
- Chrome 3.0+
- Firefox 3.0+
- Internet Explorer 9.0+
- Opera 10.0+
- IOS (Mobile safari) 1.0+
- Android 1.0+

[16] Since the WebVR solution will primarily be run on a smartphone, compatibility with IOS and Android browsers is essential.

Canvas - technical details

What happens when using canvas (for drawing) is that javascript is executed on the CPU, which interacts with the GPU to display the pixel on the screen. For a simple scene (2D), this is fast procedure; but for a complex scene it is not fast enough, this is due to JS being single threaded, which means you only access one pixel at a time. Major smartphone screen sizes in most cases have atleast 1280 x 720 pixels = 921,600 pixels; sequential pixel access is a major problem for such a large number of pixels. We can optimize this however through the use of WebGL - more on this later.

Implementation of the HTML itself is of least importance (to me) because Kodie (and his IT team) will be producing the frontend and backend of the website. To render graphics on a specific webpage I have to be able to append the canvas tag into the html file. Javascript will be used for appending the canvas tag and rendering the WebGL (WebVR) scene on the webpage.

5.1.3 CSS

Cascading Style Sheets (CSS) on the other hand is a language/script which is used for describing the presentation of a document written in HTML. It can be said that CSS is an extension to HTML since its primary purpose is to define the visual presentation of the HTML document. CSS implies a concept analogous to "*Separation of concerns*"; we separate the visualization/presentation of data (using CSS) from the raw data (in HTML). For WebVR implementation, the least used technology/language will be CSS; we are not concerned on the layout/presentation of the webpage, we are rather only interested in the content within the *canvas* tag of the webpage.

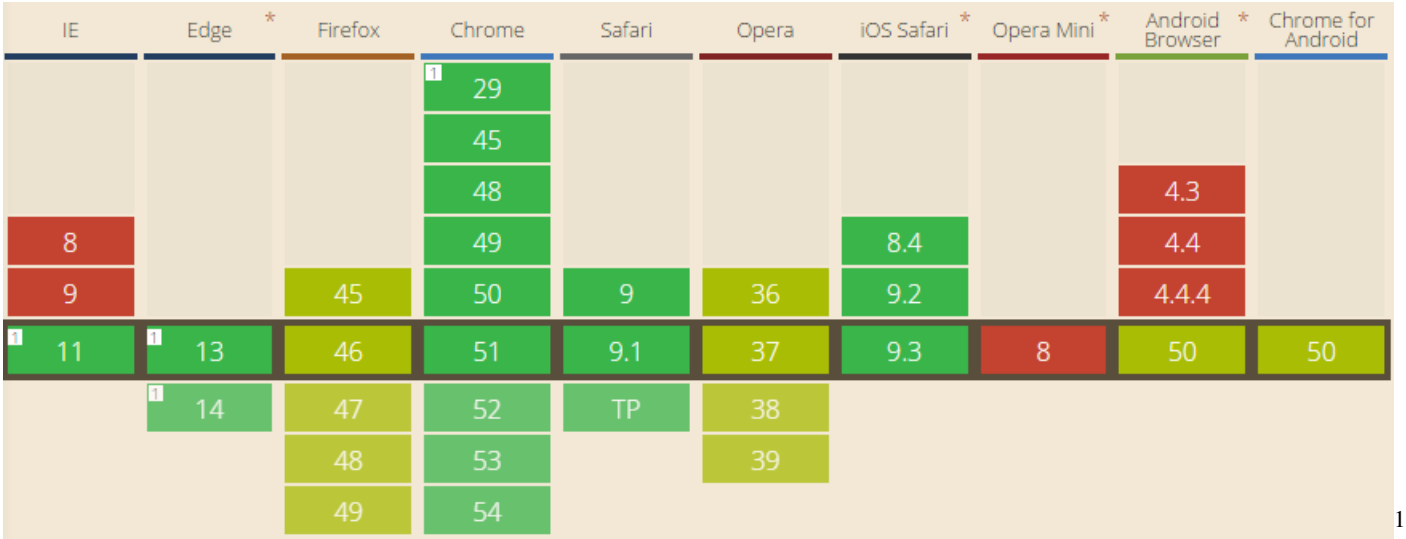
5.2 javascript

Javascript (js) is a high-level interpreted programming language. It's a fairly *easy* language to learn (compared to others such as C/C++) due to its dynamic and untyped nature. In addition to that it is also a very powerful multi-paradigm programming language which has support for object-oriented, imperative and functional programming styles. JS is also supported by all major browsers without the use of any plugins. HTML, CSS and javascript are the core programming languages of the web; since I will be implementing a web based solution, it is essential to know how to code in javascript. If we wish to use javascript for our webpages (which we surely will), then it must be referenced by the HTML webpage - usually done in the '*head*' tag of the html document.

To implement a WebVR solution we have to be able to display graphics onto the screen. A javascript API, WebGL allows us to do exactly this.

5.2.1 WebGL

Web Graphics Library (WebGL) is a javascript API which is used for rendering 2D and/or 3D computer graphics. WebGL is derived from OpenGL ES 2.0, it utilizes the HTML **canvas** tag for displaying graphics.[17] The most prominent feature of WebGL is that it can render graphics **without the use of any plugins** - on compatible browsers.[17] Since the implementation of a WebVR solution requires WebGL, it is important that the Opus's clients have a compatible browser that can display the WebGL rendered content. These browsers (and their respective versions) are shown in the diagram below:



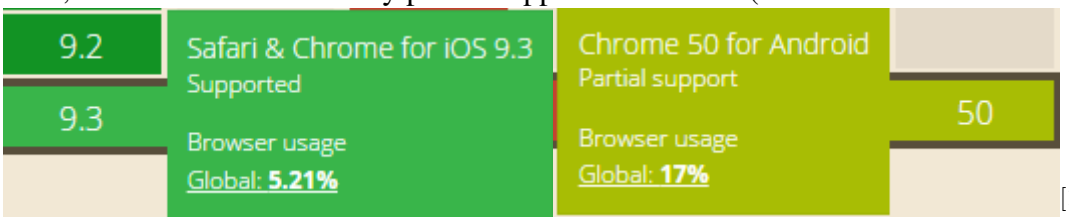
[1]

Fig-3: WebGL browser compatibility chart

- Red: Not supported
- Light Green: Partial support
- Solid Green: Fully supported

[1] http://caniuse.com/#feat=webgl

The browsers of most interest are iOS Safari, Android browser and Chrome for Android. This is because these are the mobile browsers (used in smartphones) which clients will use to view the WebVR solution (3D model in VR). IOS is fully supported so thats not a problem, but however there is only partial support for android (chrome and stock browser).



[18]

Fig-4: Chrome support for iOS and Android devices with WebGL usage statistics

When hovering over Safari/Chrome for IOS and Chrome for android, we see that these browsers have a reasonably high amount of WebGL global usage. Android has a whopping 17% WebGL browser usage, thus meaning that a significant number of clients will be most likely be using the Android browser for viewing WebGL content (and later our proposed WebVR solution). The partial support of WebGL for android could become a constraint on how we implement the WebVR solution since the android platform has a significantly large user base.

Technical details

When using simple javascript for rendering graphics in the canvas tag, the JS only has sequential pixel access (as stated previously). This is a major problem but, the canvas tag can be optimized using WebGL. This is because WebGL uses the GPU in parallel, it doesn't go through the CPU sequentially like normal JS would. This allows WebGL to access billions of pixels in parallel for rendering graphics in the canvas tag and thus significantly increases graphical performance of the rendered scene.

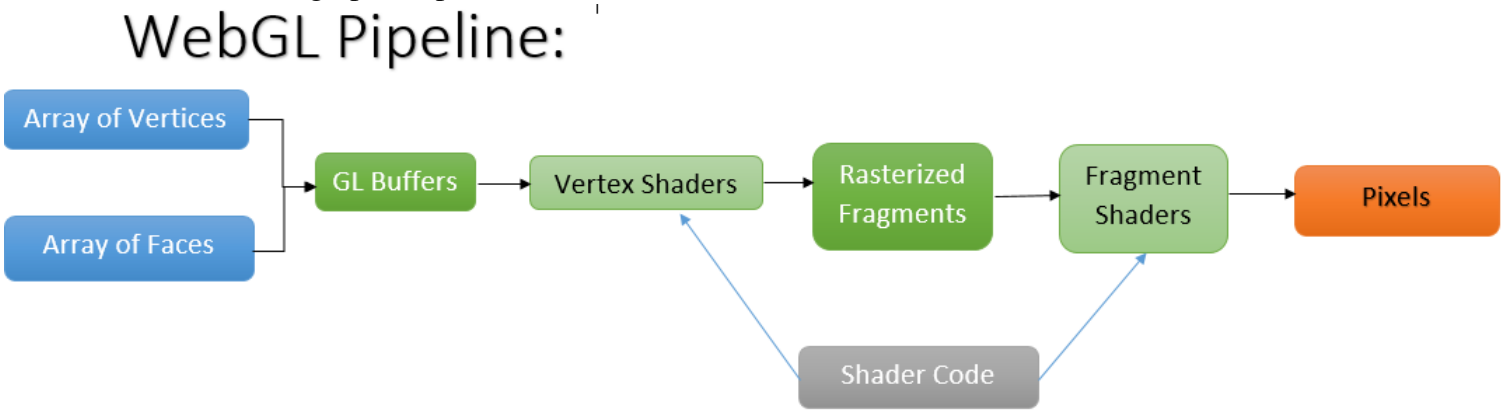


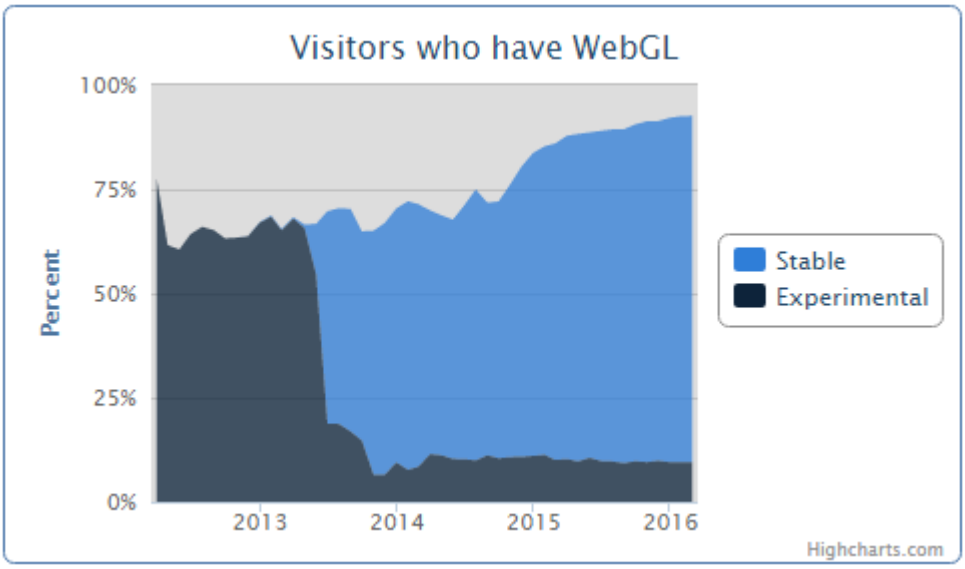
Fig-5: WebGL pipeline of underlying procedures for webpage graphics rendering

The above diagram illustrates a basic pipeline of how WebGL efficiently utilizes the GPU. Initially two arrays (Array of Vertices and Array of Faces) are uploaded from WebGL to GL

Buffers. A GL Buffer is a section of memory on the GPU (embedded in the GPU) which stores the vertices and faces. By uploading these arrays directly into the GL buffers is an example of how WebGL gets direct access to the GPU. The buffers then pass on data to the vertex shaders which return rasterized fragments. These fragments are filled with colour via Fragment shaders and output as a pixel on the screen. The Vertex Shaders and Fragment Shaders are coded in GL shader language (shader code) which is run on the GPU. We can also write in GLSL in WebGL, this is another way we can use to get WebGL direct access to the GPU. This whole process runs in **parallel** on the GPU for all pixels which results in optimal performance, this allows users to render reasonably complex scenes at high frame rates - 60 fps+.

Why WebGL

WebGL enhances user experience and in some cases allows viewers to get a better understanding of the concept (illustrated by the graphics). Before implementing a WebGL/WebVR solution we should analyse how prominent WebGL is as a web technology.



[2]

Fig-6: WebGL usage statistics - Experimental vs Stable builds

The following graph illustrates the percentage of users which have WebGL browser compatibility to view WebGL web pages in the time period of 2012-2016 (present). This site was last updated in February 4th 2016 (recent statistics), and shows us that a large majority of users (90%) have a WebGL compatible browser. These stats reinforce the fact that we should use WebGL to produce a WebVR solution as it is highly likely that clients will have a compatible WebVR browser which they can use to view their 3D VR model (WebVR solution).

[2] http://webglstats.com/

5.3 Prototyping

To implement a viable solution I have to *test out* or in other words prototype code via making use of the web technologies. The development phase consists of having a simple idea, applying coding knowledge to implement the idea. This can be for example as basic as rendering a cube in WebGL on the screen. Once you have basic understanding of how the code works, you can build upon it to produce complex objects/structures. The initial prototyping consists of thinking, testing, and thinking again; creating your own hypothesis and conclusions on how the code (and API/libraries) works, in the process learning what works and what doesn't. This is exactly how I prototyped, testing my hypothesis and seeing what works and what doesn't. I knew that a successful and reliable solution could only be implemented once I understood the fundamentals of the web technologies.

5.3.1 Workflow

Before actually getting into prototyping I have to plan out a path to follow; without any direction it would be very difficult to produce a solution. The desired workflow is:

- Retrieve a 3D model (collada file) from Opus which I can use to convert into a WebVR format
- Export the model to a JSON object which can be loaded by javascript for rendering
- Render the imported model in javascript using WebGL (within the Canvas tag of a webpage)
- Once we have a successful model which we can render, we need to implement controls to walk through the model and virtual environment. Movement will be based on keyboard controls. Remember buttons/keys are only functional if we view the scene on a PC browser, we will not have any buttons/keys when the scene is viewed on a mobile browser.
- Implement VR (device orientation) controls to be able to view the scene stereoscopically.
- Once we have an established solution for movement via keys, then we can expand on this movement system via using just our headset. For example since we are only interested in moving forward within a model, the movement in VR can be controlled by 1 button (Cardboard has 1 button input). This will provide with the viewer with a method to traverse the scene without using any keys.
- The final *ideal* solution should essentially allow the VR team to parse any 3D model (in the right format) into the script/program which will allow for pluginless VR walkthroughs of it on mobile and PC browsers.

From the workflow above, this means that throughout my prototyping procedure it is very important that I keep my code compatible to any input model, meaning that the javascript cannot be specifically designed for one 3D input model.

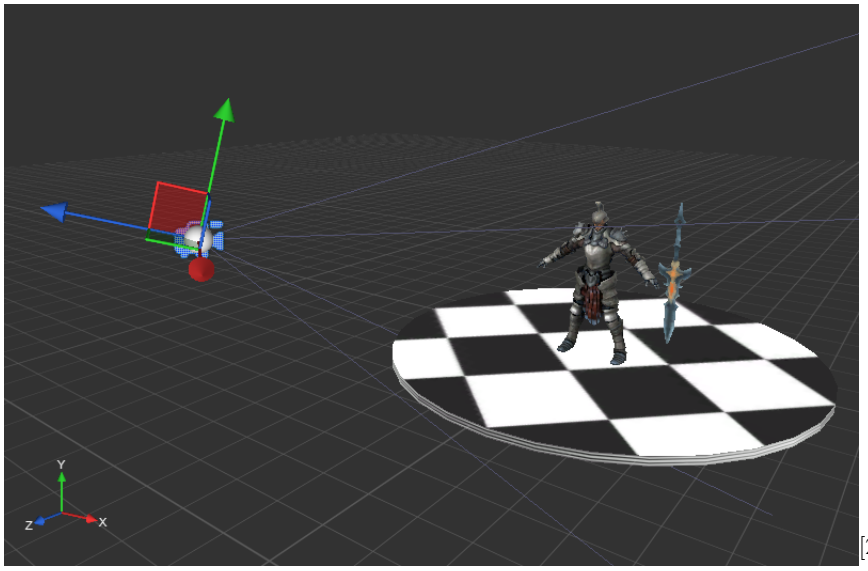
We can breakdown the workflow to 6 simple steps as a reference point to see my progress in implementing the VR solution.

Prototyping procedure

1. **Setup the vital elements to render a basic scene in javascript; render a simple object within this scene (as a reference)**
2. **Load a model (collada file) into the scene, convert it to the right format (.dae) if required**
3. **Modify camera parameters so that I can walk through the model. Implement the necessary physics (gravity, collision detection) for fluid/realistic movement**
4. **Replace the normal camera with a stereo camera to make it VR compatible (Google Cardboard) with mobile devices**
5. **Allow camera movement in the scene without the use of keys - so the solution works on both PC and smartphone (VR)**
6. **Implement GUI controls so that Opus (VR team) can have full control of model, scene and VR parameters without requiring to modify any code**

5.3.2 3D graphics - basics

To perform any type of rendering of 3D content, we have to first understand how 3D graphics work. To display 3D graphics on a webpage (or any platform) we require 3 essential ingredients; a **camera** a **scene** and a **mesh**. The setup of such a model in a basic scene is shown below:



[22]

Fig-7: Components involved in rendering 3D graphics

In a scene we have a mesh; a mesh consists of a bunch of triangles. Shaders are used (remember GLSL shaders) which fill in the triangles with a *material* to give us a colourful complex objects. However we cannot see this object without a camera.

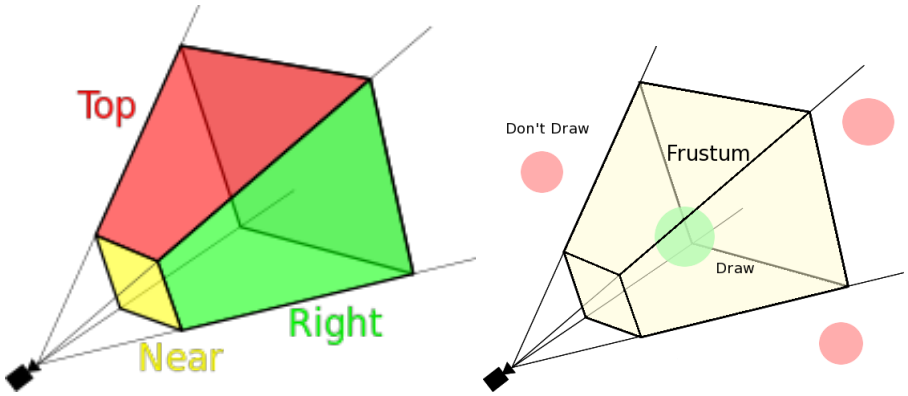
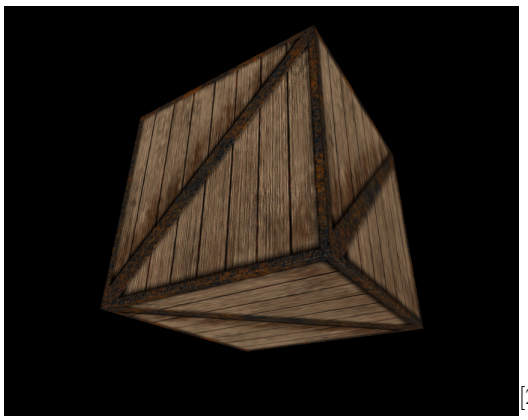


Fig-8: Camera frustum attributes - rendering areas

A scene can only be viewed through a camera; the camera casts our rays into the scene as frustum. A frustum is basically the camera view, everything in this frustum is exactly what the camera can see. Both the camera and the mesh have a position and a rotation in the scene. Static models have no rotation, but they do have a position. To be able to view the model, the mesh must be within the parameters of the frustum (camera view), this can be done by translating the 3D coordinates of the mesh to be within the frustum, or, a better approach would be to change the view (rotate the camera) so that the frustum contains the mesh. Once the frustum contains the mesh we need to render the scene; the renderer is used which projects all the 3D stuff within the scene onto the 2D webpage. Complex objects have alot of meshes, and thus have alot of triangles which need to be shaded with a material (via GLSL) before rendering; this is the reason why highly complex objects/models can be graphically intensive to render - this can affect the performance of the scene.

An example of such a simple 3D rendered object in a scene is shown below:


[23]

Now that we have an understanding of how 3D graphics work in general (and specifically WebGL) and also have a path to follow, we can begin prototyping. Remember that the initial step is to be able to convert a 3D model provided by Opus to a JSON format.

Since Opus uses two technologies for 3D modelling, Google Sketchup and Unity, it would be logical if we start experimenting with these technologies to see if we can export the model to **JSON** format from within these applications.

5.4 Google SketchUp

Rather than implementing a WebVR solution from scratch, I initially looked at open source software which may already do this or ease the implementation procedure. The VR team told me to experiment with Google SketchUp because that is one of the software's they use to produce/modify models for their clients.

Google SketchUp is a free 3D modelling computer software. It is used for a wide range of applications such as modelling architecture, interior design, engineering, film and also video game design. The application allows a user to load up and view CAD (Computer Aided Design) aka collada files, these are models of structures/architecture. Initially I experimented on Google SketchUp because the first model which Opus supplied to me was in CAD/collada format which could be viewed in SketchUp.

This software offers many tools to build 3D models; users can download basic models from the SketchUp website and modify those to produce more complex models - such as the one provided to me by Opus. Model design is not my job however, my role is to load and walkthrough those models in VR (on the web).

After tinkering around with the model, I started looking for online solutions to export the model to a JSON format. Exporting the model to a JSON format (WebGL) is essential in the implementation of a WebVR solution because it would be infeasible to manually convert the models to JSON - takes too much time. After some searching I came across a plugin produced by **TAK2HATA** (online name).[20] This plugin allowed me to export the model as a scene to a WebGL; essentially allowing me to skip the JSON conversion. The plugin converted the model to JSON and produced the javascript and html files which could load the model and display it on a webpage. Additionally, the exported WebGL webpage offered extensive control on how the model was viewed.

I have uploaded this model to: **http://zsar419.github.io/webgl_1/**

Usage:

- Use the arrow keys to move the camera in the scene (translation)
- The mouse changes the view of the camera (rotation)

This is a scene in WebGL, but not in a WebVR format; meaning that it is not a stereoscopic scene which allows a user to walk in the model. This export only allows a camera to view the model, it does not provide immersive walkthrough experiences. Rather it is an online tool which allows you to view models as if they are viewed in Google Sketchup.

An example of a scene in which the camera (person) can walk within the complex 3D model is shown below:

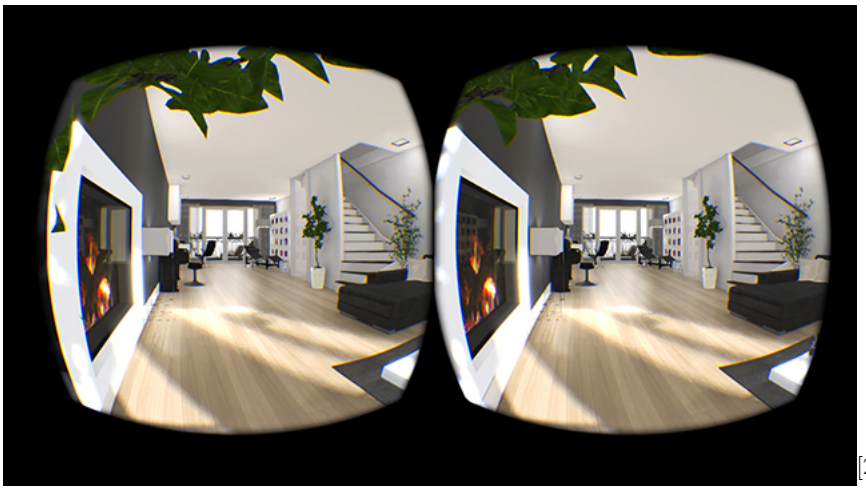
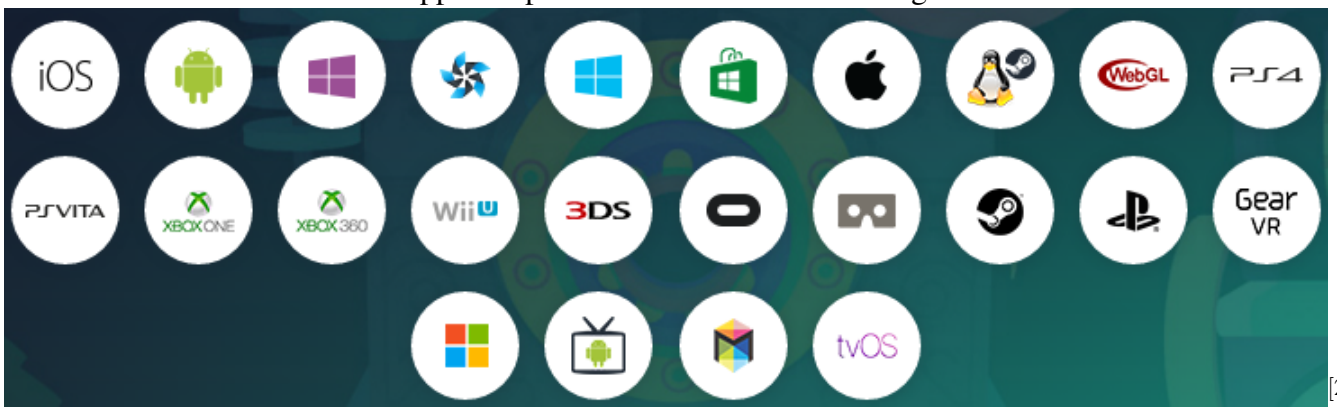

[24]

Fig-9: Example of a high quality model viewed in stereoscopic view (VR)

This is what we wish to achieve with the input model provided by Opus; immersion within the infrastructure. Since the exported WebGL code is very difficult to analyse (not developed by me), I moved onto another platform which could potentially ease implementation of a WebVR solution.

5.5 Unity

Unity is a cross-platform game engine which is primarily used to develop video games for PC, consoles, mobile devices and websites. A major upside of using unity compared to any other engine is that it follows the "*Build Once Deploy Anywhere*" model; this allows a developer to produce content only once and effectively distribute it to all platforms with a click of a button. The supported platforms are shown in the diagram below:

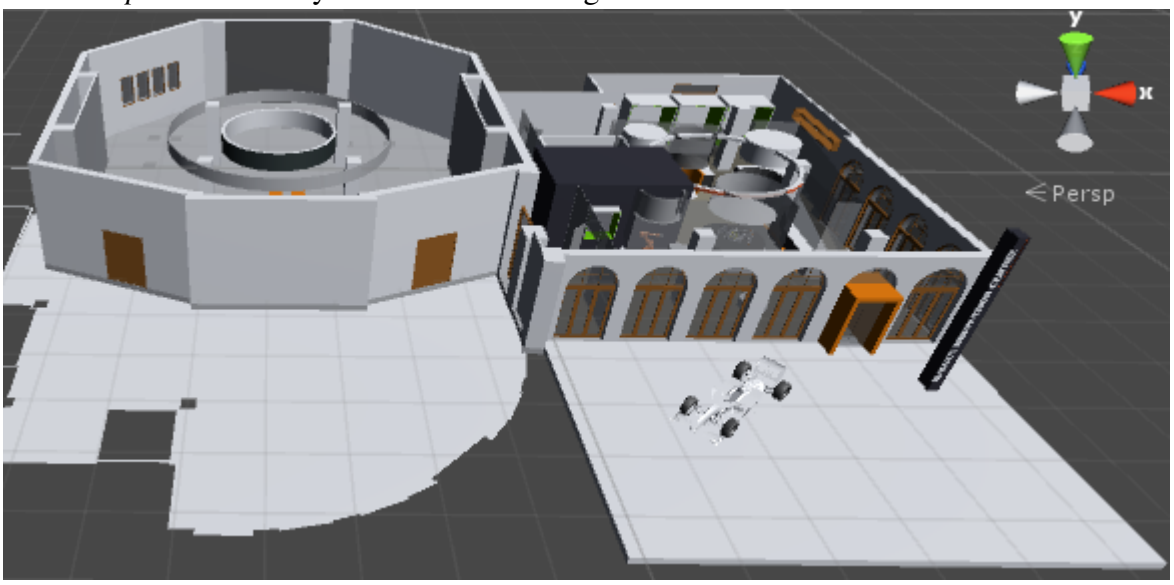


[21]

From the image above we can see that unity provides support for a vast majority of the platforms thus Opus can cater to potentially all clients using Unity. To me (and Opus), the platform of interest is WebGL.

To load the model in unity I exported the model from Google Sketchup as .dae (digital asset exchange) format. The exporting process output two files, *"sketchup-demo.dae"* and

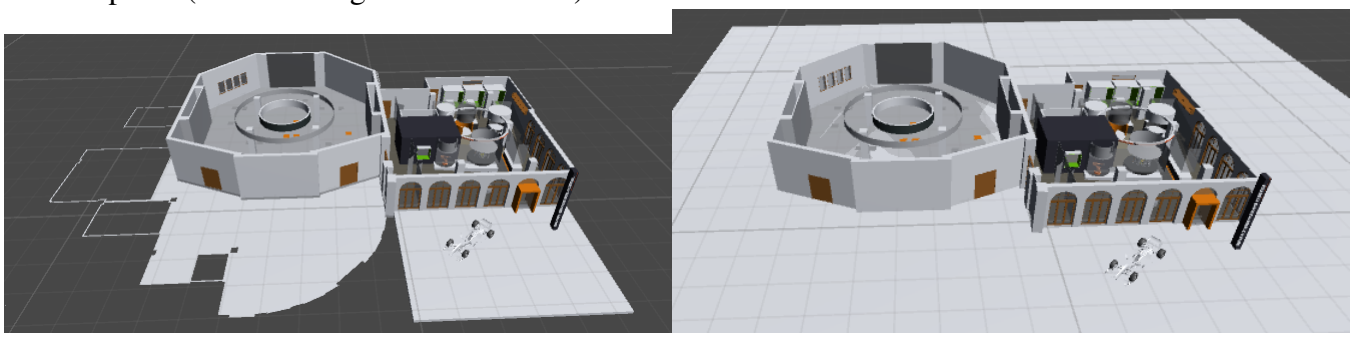
a textures file which contained all the textures which were present within the model. The *sketchup-demo* in unity is shown in the image below.



Now that the model has been successfully imported in unity, I have to be able to walk through it in first person (like how a real person would). To be able to do walk in a scene we need to have a plane which accounts for collision - prevents camera from falling through the floor (due to gravity). The plane also makes the scene look realistic - since we don't want a floating camera. To add a plane in unity we go to:

GameObject -> 3D object -> plane

Once the plane is added to the scene we have to scale the plane so that it represents the size of the model, we can do this by selecting the plane and looking at the "*Inspector* tab, then scale the X and Z axis; we dont scale Y because that will change the height of the plane (we want height to be at 1 unit).



The images above shows the 3D model without the plane on the left and the model with the plane on the right; as you can see now we have a floor which we can use to walk around in the scene.

Now that we have a plane (scaled to model size) we can add a camera - which will act as a person walking in the scene. To add a main camera:

GameObject -> Camera

Now that we have a camera, we can traverse through the model. The image below

illustrates the first person view of the camera within the model.

Fig-10: Model walkthrough on Unity

Note: There were certain steps involved in setting up this scene, Opus helped in the implementation of a player controller, shadows and other misc. settings

Unity - WebGL Export

Now that we have a functional scene in unity, we need to be able to test it. You can manually test the scene within the unity engine by pressing the play button, however this is only a testing method within unity. Clients will most likely not have the unity application and thus we need to export to WebGL so that it runs in a browser. Unity can export to WebGL; to do this however, we must first download the WebGL unity package - which can be done using the unity installer. After installing the WebGL package, to export the scene we go to:

File -> Build Settings -> WebGL

This will build the WebGL javascript files, CSS files and HTML files - basically everything that will allow you to view the same scene/game no on the web.

Unity WebGL conversion - technical details: The way that unity 3D works is that it converts the model/scene into C# code (changes are reflected in C#), it does so by compiling the code into IL code (intermediate language).[25] Then it uses the IL2CPP converter to create a C++ version of the code which is translated to javascript/WebGL.[25] This procedure also optimizes the output javascript/WebGL code and thus is an extremely convenient way to develop graphic heavy content for the Web.

But with these advantages, there are also some significant disadvantages which must be taken into account. Some of these are:

- Extremely difficult to debug, you can only debug through the console (very inefficient)
- Highly limited code accessibility and readability, it is very hard to understand the code produced by the conversion - even harder to modify
- Additionally, the output file size is huge - in most cases 300MB+ because 300MB is just the size of the unity library
- Due to the resultant large file size and difficult in code readability, the developer is very limited in code modification and in most cases cannot perform their own modifications

[25] The ease of development still (in most cases) heavily outweigh the downsides of Unity. In addition to that it is an emerging technology which most people are working hard on; there is alot of investment on the Unity3D platform and thus this technology will continue to improve.

After exporting the project to WebGL file(s) we click the *"index.html"* to open and view the scene. Due to the fact that the project is produced in unity, we are met with a unity splash screen. We can enlarge the scene to fullscreen to get a better full-screen view.

I have uploaded the WebGL model at:

http://zsar419.github.io/webgl_model/

Since we have been able to successfully implement the scene in WebGL, the next logical step is to convert the camera to WebVR. By this we mean convert the camera into a stereo camera so that the user can use google Cardboard to look around in the scene. But first I must check if the hosted website works on a smartphone.

5.5.1 Unity WebGL compatibility Issue

When I tried to access the hosted WebGL webpage from my android smartphone, the webpage failed to load on my smartphone. When searching for a remedy, I was shocked to find out that the Unity exported WebGL webpages are **not compatible with smartphones**. There are major compatibility issues with smartphones, this issue made me realize that implementation of a WebVR solution is not as straightforward, currently Unity does not support mobile WebGL and thus I cannot use Unity to implement a WebVR solution.

After using both these technologies (Google SketchUp and Unity) and failing to retrieve a viable WebVR solution, I realized that I had to implement a solution from scratch. Since WebVR is an emerging technology (very recent), there aren't a many resources available, thus implementation from scratch is a difficult task; nonetheless I must try. To produce a WebVR solution, this requires understanding of what is required, then planning towards it; testing out and getting results. Since there is no major support other than some examples, I realized this would be an incremental procedure in which I would have to read documentation (API's), acquire knowledge, then test out the knowledge in small incremental steps.

5.6 ThreeJS

Three.js (3JS) is a cross-browser open source and lightweight javascript library/API which is used to create and display computer graphics on a web browser. Its an API on top of WebGL which simplifies much of the WebGL code so the developer could focus more on the implementation rather than the repeated '*dirty work*'. For example shaders in WebGL are expressed via complex GLSL code, in ThreeJS to represent a shading, a mesh is only 1-2 line(s) of code which runs the same underlying GLSL code. This effectively means that we can render a scene with WebGL performance but using much simpler code. The source code for this graphics library/API is hosted on:

https://github.com/mrdoob/three.js/ and **threejs.org**

After some basic researching and looking at examples, I realized that I can study the distinct examples and integrate the key elements (code) from each of these examples to implement a WebVR solution. A ThreeJS example for a stereoscopic scene is shown in the picture below:

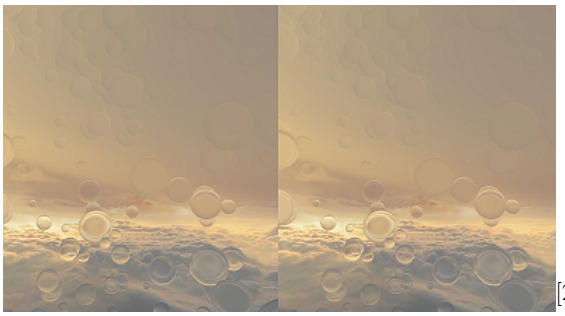

[26]

Fig-11: Using ThreeJS graphics library to render stereoscopic scene

The picture above shows a split screen**WebVR** scene of bubbles; using Google Cardboard a user can turn their head to view moving bubbles. You can view this scene at: **http://threejs.org/examples/#webgl_effects_stereo**

The scene above only incorporates a stereoscopic split screen, it does make use of any device orientation controls or any other key elements which are required for my solution. What Im trying to imply is that for the implementation of my final WebVR solution, I had to use bits of code from examples from the ThreeJS site (such as the one above for stereoscopic view).

ThreeJS will remove some of the complex WebGL code for 3D graphics and allow for a much simpler way to render the same scene. To start off with ThreeJS I have to be able to manually render a scene. Due to the fact that I was unsuccessful in implementing the WebVR solution using Google SketchUp and Unity, now I must follow the 6 step prototyping procedure (described before) so we can start producing our own WebVR solution (form scratch).

5.7 WebVR solution implementation (ThreeJS)

5.7.1 1 - Setting up the a basic WebGL scene

The first step is to setup the required variables in javascript to render a basic scene. To do this lets first make an html file which will be used to host the webpage. This can be a basic webpage with no content, but it must have the **canvas** tag.

To setup a basic webpage on which we can render 3D graphical content we will make a basic html file:

```
<html>
    <head>
        <script src="http://threejs.org/build/three.min.js"></script>
        <style>canvas { width: 100%; height: 100% }</style>
    </head>
    <body>
        <script>
            // Javascript will go here.
        </script>
    </body>
</html>
```

For prototyping a solution, we do not need to make our webpage visually appealing and thus we wont be using CSS. Now that we have a webpage and the canvas set up, we can use the canvas tag to run graphical WebGL content. To run WebGL content we need to use javascript; in addition to that we will be using the ThreeJS graphics API instead of coding in WebGL (which will run underlying WebGL). The script tag can contain all the javascript code (ThreeJS); however it is best practice to separate implementation from view (separation of concerns). Thus we will implement the ThreeJS code in a separate javascript file (which will be linked in the head of the html file).

To setup a basic 3D graphics scene we need to first create a scene, we do so by:

```
var scene = new THREE.Scene();
```

After we have setup a scene, we need to create a camera.

```
var camera = new THREE.PerspectiveCamera(75,window.innerWidth/window.inne
rHeight, 1,5000);
```

Creating a camera is not as simple, this is because ThreeJS has more than 1 camera, in here be are using a basic perspective camera. The camera requires four parameters; the field of view (degrees), the aspect ratio, the near clipping plane and the far clipping plane. The close and far clipping plane define the length of the camera view frustum, anything outside this range will not be rendered. After we have setup the camera we need to setup up the WebGL Renderer. These camera parameters imply that everything within 5000 units of the camera and within the field of view of the camera will be rendered in the canvas.

```
var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
```

```
renderer.setClearColor( 0x0000ff );
document.body.appendChild(renderer.domElement);
```

The code above creates an instance of a renderer with a set size to render on (inner parameters of the window). We then call its "setClearColor" function which sets the background colour, this is an easy and efficient way to set the scene colour since everything not rendered in the scene will be blue (which is defined by 0x0000ff in our case). The blue colour was chosen due to the fact that its the most natural colour for a sky. We then add this renderer to the canvas tag which will now allow the renderer to render the scene within the canvas.

Now that we have a scene, camera and renderer; we need to create an object/mesh to render, in this case we will be rendering a basic cube.

```
var geometry = new THREE.BoxGeometry(500, 500, 500, 0, 0, 0);
```

This creates a cube with height width and depth of 500. The box geometry defines the vertices of the cube - in this case a box structure. We need to colour the geometric faces with a material:

```
var material = new THREE.MeshBasicMaterial({color: 0xfffff,
wireframe: true});
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

We colour the cube with a blue coloured material (defined by *0xfffff*), in addition to that we make it a wireframe model which will help us view the animations. To actually create the cube we need to use the geometry and its mesh, the next statement combines the geometry and the mesh to produce the cube. The last statement adds the cube to the scene at world coordinates 0,0,0 (by default).

Now that we have a scene setup, we must remember that the position of the camera is at the origin (0,0,0) while the position of the cube is also at the origin, meaning that our camera is probably within the cube, we must move it back to be able to see the cube. We do so by:

```
camera.position.z = 1000;
```

Now the camera is at a reasonable distance from the cube, this will allow the cube to be within the camera's frustum (view).

We cannot see the cube yet, this is because we have no light source in the scene. To implement a light source we add the following code:

```
var d_light = new THREE.DirectionalLight(0xffffff, 0.5);
d_light.position.set(0, 1000, 0);
d_light.castShadow = true;
scene.add(d_light);
```

There are many types of lights in ThreeJS, these include ambient, hemisphere, directional and spotlight. Each light type has special properties which distinguish it from the rest, for example only directional light and spotlight can cast shadows; however, the radius of the

spotlight is much lower than that of the directional light. In our case Im using the directional light in the scene; the colour of this light is white (0xffffff hex) with an intensity of 0.5 defined within the constructor. I then allow it to cast shadows, this will only cast shadows on objects which can receive shadows (more on this later).

Once we have added a light source, to view the scene, we render it using the following code.

```
function render() {
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
render();
```

This is an automatic render loop since the method (renderer.render(...)) continuously calls the render method by default (doesn't end) after reaching end of file. The *requestAnimationFrame* allows the user to pause the WebGL when it is not in the foreground - for example when the user switches to another tab the WebGL scene pauses automatically.

To animate the cube we need to rotate/translate it:

```
cube.rotation.x += 0.01;
cube.rotation.y += 0.01;
```

In this case we add the above code to the render function so that with each frame the cube rotates along the X and Y axis.

Now we also have an animation in our scene; the basic setup of a WebGL scene is complete. You can view this basic scene at:

http://zsar419.github.io/webgl_cube/

It is very important that we understand what is going on in the scene (hence my explanation) because these fundamental concepts will be extended upon to implement the remaining steps of our prototyping procedure.

Now that we are able to render a basic scene we can move onto step two, which is loading the 3D model (provided by Opus) into our scene.

5.7.2 2 - Load a model (collada file) into the scene

To render a model into the scene, all its vertices (polygons) must be loaded up by a script. To load up a model, it must be in a specific format. Due to the fact that 3D models will most likely be used by other applications which are most likely not web based, it is unlikely the 3D model format will be JSON. After consulting the format with Opus VR team, they stated that the format of the 3D models will be **digital asset exchange** (**".dae"**) format - which is a collada extension. If the 3D model is not in that extension, the VR team would convert it (via using *Revit* or *Google SketchUp*) to ".dae".

A model under this format usually has 2 files, one file is has the actual ".dae" extension, this defines all the 3D models vertices, faces and texture mappings. The second file is a folder of the same name which contains the textures that are applied between vertices (faces) of the model. An example of the 3D model given to me is shown in the picture below:

After realizing that the models are in ".dae" format, I had to find a collada loader for ThreeJS. Luckily to my surprise, the ThreeJS library had such a collada (".dae") loader.

To my excitement I tried out the loader, first I added the ThreeJS loader script into the head of the html file:

```
<script src="http://threejs.org/examples/js/loaders/ColladaLoader.js">
</script>
```

After getting access to the loader, I added the following code in the script to load the *sketchup-demo.dae*, my custom 3D model (provided by Opus):

```
function loadModel(name){
        var loader = new THREE.ColladaLoader();
    loader.options.convertUpAxis = true;
    loader.load('Model/'+name, function ( collada ) {
        model = collada.scene;
        model.position.set(0, 0, 0);
        model.rotation.set(0, 0, 0);
        model.scale.set(1, 1, 1);
        // Casting shadows for all children
        model.traverse(function(child) {
                child.castShadow = true;
            child.receiveShadow = true;
        });
        scene.add(model);
        }, function ( xhr ) {console.log( (xhr.loaded / xhr.total * 100) +
                '% loaded' );}
    );
}
```

To load a 3D model, I defined the function "loadModel" which takes in the model name as parameter. In my case, this function is called with the parameter "sketchup-demo". In the following lines we define the ColladaLoader, and call the load method. According to the right hand rule, we want the Z-axis to be the depth axis (which is in/out of the page) and thus to enforce this, we set the convertUpAxis" variable to true - otherwise model will be loaded in an incorrect orientation. To enforce good practices and ease of use, this code will load models only in the "Model" folder, therefore the relative path of the sketchup-demo is "Model/sketchup-demo.dae", additionally, the load function is adaptive since it loads the model based on the name input parameter. This essentially allows us to load multiple models by calling the function with different input parameters thus allowing the code/application to scale.

The next line assigns the variable name ”model” to the loaded model in the anonymous function. This is important because now we can access the model and modify its state outside the scope of this function, basically allowing for post-loading modifications or *modifications on the fly* - (importance explained later). The next few lines allow you to set position, rotation and scale (for which Ive shown the default values). The ”model.traverse” is a vital in-built function which allows the code to recursively traverse all the children of the model and apply a certain function to them; in my case Im making the model realistic by allowing its components (children) to cast and receive shadows in the scene. The directional light allows the children to cast and receive shadows. The next line closes out the load function and also logs the status in the console, this gives real-time feedback in the console (for debugging purposes) which shows the current loading state/progress of the model. It must be noted that larger models will take longer to load than shorter ones due much more data being packed into the same collada file - thus it is helpful to have an indicator of load status.

Once I have been able to load up the model, I adjusted the model parameters such as scale, and the player parameters such as rotation and position (manually via code) to get a good view of the model. Additionally, I managed to change the background colour from black to blue. This is done by the renderer; I implemented is to that the default rendering colour is sky-blue (the colour of the sky) to make the scene more realistic. This was done by the following code:

```
renderer.setClearColor(0x1E90FF);
```

Where ”0x1E90FF” is the chosen sky-colour. The model is shown in the image below:

Fig-12: Loaded demo model (PC) and blue background - adding realism

Now that we are able to successfully load a model, the next step would be to allow for player/camera movement in the scene.

5.7.3 3 - First person camera controls

To have an immersive scene, the user must be able to interact with elements within the scene. Initially we will implement a WebGL walkthrough and extend capabilities to VR.

Thus the first step would be to cater for PC controls - mouse and keyboard based interaction within the scene. We want the mechanics of the walkthrough (PC) to be very intuitive and familiar to that of other applications, thus we will implement movement and rotation which is similar to that found in first-person shooter (FPS) games.

The camera represents an entity in the virtual world, in our case its the user; so we will now refer to the camera as the **player**. There are two types of movement within the scene, player rotation and player translation. In typical fps games, player rotation is provided primarily by the mouse, while player movement (translation) is provided by the keyboard. We will use the same approach in for PC based controls.

person player rotation

Upon searching for examples on the ThreeJS library, I found that an example which used first person controls similar to what I wish to implement. The following example was used as the template for implementing controls for PC based WebGL controls:

https://threejs.org/examples/#misc_controls_pointerlock

Upon studying the underlying code, I realized that "**PointerLockControls**" are responsible for providing fps mouse based "look" controls, this allows you to control the player rotation within the scene. I managed to strip down the implementation of the example (stated above) to obtain the barebones player rotation (mouse rotation). To use player rotation one must import the following script in the html head:

```
<script src="js/controls/PointerLockControls.js"></script>
```

Then, the following code allows fps look-based movement (via mouse) within the scene:

```
var controls = new THREE.PointerLockControls( camera );
function lockMousePointer(controls) {
        document.addEventListener( 'pointerlockchange', (()=>{controls
                .enabled = document.pointerLockElement === document
                .body?true:false}), false );
        document.body.addEventListener( 'click', (()=> {document.body
                .requestPointerLock()}), false );
}
```

The first line links the camera to PointerLockControls - basically allowing the camera to rotate using mouse movement. But this wont occur yet, we need to add event listeners to do so. I've abstracted the implementation of fps based controls into a single function "lockMousePointer" which has two event listeners with anonymous functions. One anonymous function allows player to rotate based on mouse movement (similar to fps), while the other event listener checks if the user is currently focussed in the WebGL application. If the users focus is not within the application, then mouse movement is ignored, thus reducing performance load on the application when not in use.

First person player movement Now that we have successfully implemented player rotation within the scene, we must implement movement (translation) via keyboard controls.

In my case, I would want the player to move around in the scene using the four arrow keys and also *WASD* controls. For this we would need a keyboard event listener and also define other characteristics such as player movement velocity.

For this type of movement, we will declare four variables, for four different types of movement - forward, backward, left and right. We will initially set these to false and then add the following event listeners:

```
function addPCControls() {
        document.addEventListener( 'keydown', onKeyDown, false );
        document.addEventListener( 'keyup', onKeyUp, false );
}
```

The onKeyDown() and onKeyUp() methods are called based on the keydown and keyup events. Those methods (not shown in this report - due to length) basically set the movement variables to 'true' when their respective key is pressed, and false when that key is released. For example, when pressing 'W' (or arrow key up), the "forward" boolean variable is set to true, when that key is released, the variable is returned back to false. In addition to these variables, we will have another variable "canJump", this will regulate player movement (such as jumping - with SPACE) within the scene (more on this later). To move the player in the scene based on these variables we declare the following method:

```
// Rendering Movement Changes
function renderPCMovement(player) {
        var time = performance.now();
        var playerSpeed = 1600-(player_c.speed*100);
        var delta = ( time - prevTime ) / playerSpeed;
        velocity.x -= velocity.x * 10.0 * delta;
        velocity.z -= velocity.z * 10.0 * delta;

        // Prevents forward movement when colliding
        if ( moveForward) velocity.z -= 400.0 * delta;
        if ( moveBackward ) velocity.z += 400.0 * delta;
        if ( moveLeft ) velocity.x -= 400.0 * delta;
        if ( moveRight ) velocity.x += 400.0 * delta;

        player.translateX( velocity.x * delta );
        player.translateZ( velocity.z * delta );

        prevTime = time;
}
```

The following method is called within the **render()** method for each frame. This method requires a player (parameter) since I believe it is good practice to separate controls from scene rendering; thus I implemented PC controls in another script - hence the player must be accessed by parsing it in as a parameter to the method. The "player_c.speed" is a variable

defined outside the scope of this function which controls player movement speed within the scene; this will allow for fine tuning the speed later if required. As shown on the code above, the four movement variables regulate player translation within the scene. The other remaining variables (”prevTime” and velocity vector) are initialized to 0 outside the scope of this method.

Using the code above, the ”player” is able to move and rotate in the scene. But before we can do that, we haven't actually declared the player neither have we linked movement to the player. The following code will do so:

```
function setPlayerControls(){
        player = controls.getObject();
    scene.add( player );
    lockMousePointer(controls);
    addPCControls();
}
```

The following function assigns an uninitialized player variable to the controls object variable. The controls variable (declared above) was a modification of the camera, this basically assigns that to the ”player” - this causes the player to be the parent of the camera (in the hierarchy) and thus camera movement and rotation occurs upon player movement and rotation. For player rotation to occur, we call the *lockMousePointer()* method. For integrating player movement, we implement the keyboard event listeners using the *addPCControls()* method.

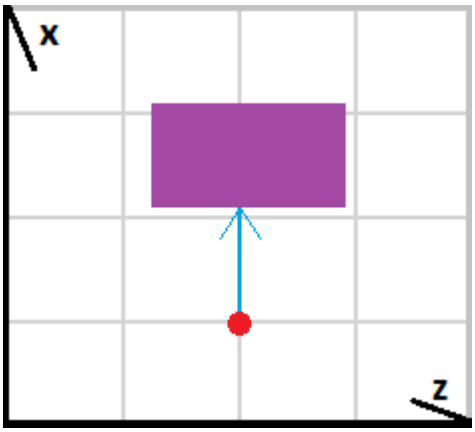
Now we have a functioning WebGL scene which incorporates basic player movement and rotation. This scene will provide the barebones experience for a walkthrough on PC; but however there are still many inherent issues. First of all, the player can move through walls, additionally there is no collision detection (physics) in the scene. Collision detection and prevention are vital for any sort of immersive walkthrough experience. There's also absence of gravity within the scene, so the player can walk off the model or ledges (in the model) and not fall down. Additionally, movement is restricted to the X and Z axis, the player cannot move on the Y axis; in other terms, the player cant traverse stairs. One way to address all these issues is to implement **Physics** within the scene. In our WebGL (and later WebVR) solution, implementation of physics will be split up into three parts, **collision detection, gravity and terrain traversal**.

(1) - Collision detection

A common way to address the issue of collisions is to use ray casting. Ray casting is a procedure in which specific ray(s) (with desired lengths) are cast from a certain 3D location in a certain 3D direction. Upon intersection with objects, they trigger an event which stores current state information such as the location and object of intersection. In the case of ThreeJS, rays store the coordinates 3D of the intersection with the object into a list. This means that if casted rays do not intersect with any objects, the ray intersection list is empty due to absence of intersections. Using this information it is very easy to detect collisions with objects.

For our implementation, only 1 ray will be cast at the player position in the forward

direction of the player (basically where they are facing). Only one ray is required because we are only interested in detecting forward collisions (at 90 degrees). This is because later when we extend our PC WebGL solution to incorporate VR, in VR the player can only move in the direction they are facing (forward) due to limited controls (only 1 button). Thus there is no need to cast multiple rays which will not be effective in VR and also use up unnecessary resources. The design of such a ray casting for forward (player direction) collision detection is shown in the images below:



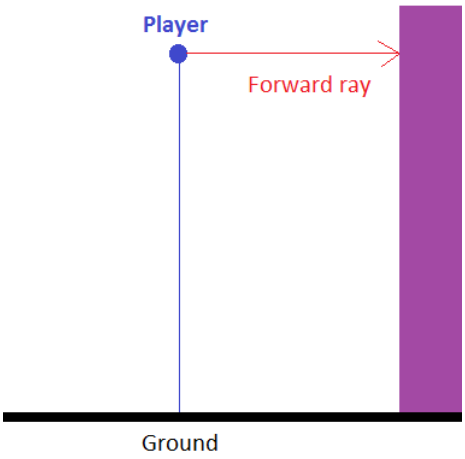


Fig-13 (Left): Ray-cast in player direction on XZ axis (2D plane) - Collision detection top-view
Fig-14 (Right): Ray-cast in player direction - Collision detection side-view

The image on the left shows the ray cast on the XZ axis from a top-down view and how it would detect intersections with 3D objects in the scene. The diagram on the right illustrates how it would seem in 3D coordinates; from the players perspective (side view).

This concept for collision detection can be easily implemented in ThreeJS. First we must initialize a ray caster which is done using the following code:

```
var raycaster = new THREE.Raycaster( new THREE.Vector3(), new THREE
    .Vector3(), 0, 15 );
```

Where the first two parameters are the ray position and direction respectively in 3D world coordinates (vector). The second parameter is the offset from which the ray actually starts (0), and the final parameter is the ray length (15 units). According to the ThreeJS documentation, initialization of the ray caster is a computationally intensive task, to avoid this, we just update the rays position (based on player position) and the rays direction (based on player direction). This is also the reason why the first two parameters (vectors) in raycaster constructor are zero - since they will be updated in render method. All of these raycasting parameters (position, direction, near, far/length) can be dynamically set via ray methods - thus potentially allowing us to use one ray caster to cast multiple rays. This is because we can loop through the ray caster and set position and direction into other vectors of interest (left, right, backwards and diagonals); if we would like to have collision detection

for the other player directions, then implementing the rays for those directions (relative to player) would be a trivial task.

Now that we have a ray caster setup, for collision detection we must cast a ray in the direction of the player (forward) and check if a ray has any intersections. This can be done by the following code:

```
function getForwardCollision(){
        var pos = player.position.clone();
    var direction = camera.getWorldDirection();
    var forward = new THREE.Vector3(direction.x, 0, direction.z);

    // Forward raycast
    raycaster.set( pos , forward);
    var intersections = raycaster.intersectObjects(scene.children,true);
    if ( intersections.length > 0 ) return true;
}
```

Ive generalized the forward collision detection to a method - which makes it easier to use for potential future developers. The first three parameters get the player position, camera direction (which is player direction) and forward vector (based on player direction). It must be noted that we are NOT interested in the y-component of the forward vector, this is because collision detection must occur for objects directly in front of player (not above) - this is why we set the forward vectors y component to 0 (90 degrees form player). The problem is best visualized by understand the illustration below:

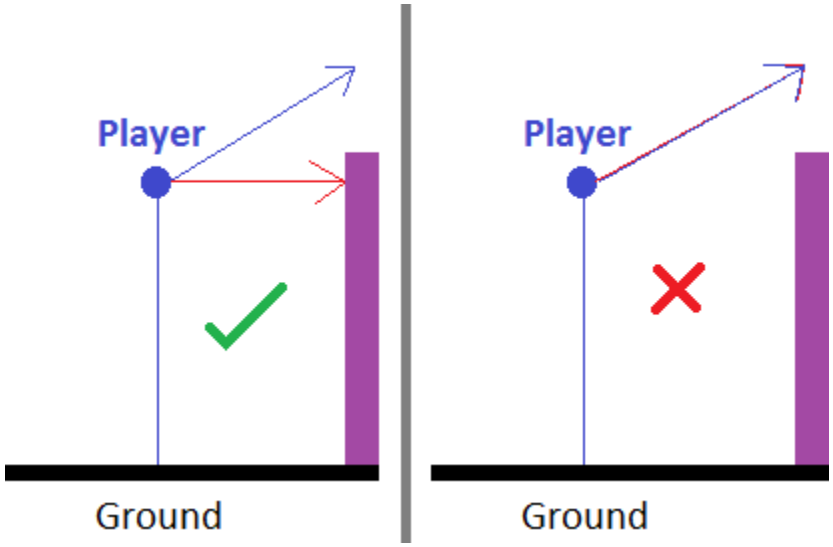


Fig-15: Ray-cast in player direction (side-view) illustrating ray with no vertical component (left) vs ray with vertical component (right)

The blue line shows the player current look direction while the red lines shows the ray casts. The image on the left shows how an ideal collision detection will be able to detect a forward

collision - when the **Y-component of the forward ray is 0**. The right scenario shows the ray being cast in the forward direction of the player **including the Y-component of player direction**; this will cause the collision to miss and allow player to get too close to objects - which is unwanted behaviour.

After setting the rays position (on player) and direction (facing forward - relative to player), we check if the ray casted forward from the player intersects with anything in the scene. The ***intersections*** variable is a list of objects which the ray intersects with. Since all we want is to check if any intersection occurs, we check if the length of the list is greater than 0, if so, that means the player is colliding with something in front - thus returning true for the collision. Since we want forward collision detection to be always present for the player, it must be performed at every frame, thus the *getForwardCollision()* will be called preferably within the ***renderPCMovement()*** method (which is called by the render() method at every frame).

Within the *renderPCMovement()*, we will replace the following line of code:

```
if ( moveForward ) velocity.z -= 400.0 * delta;
```

with:

```
if ( moveForward && !getForwardCollision()) velocity.z -= 400.0*delta;
```

Velocity.z is forward movement based on player current direction; it is relative to player (rather than scene). The *getForwardCollision()* checks the state if an object (e.g. wall) is directly in front of player, if it is not, then allow forward movement (in player Z coordinates - in front of player), otherwise disable forward movement. This forward collision detection mechanic is illustrated in the diagram below:

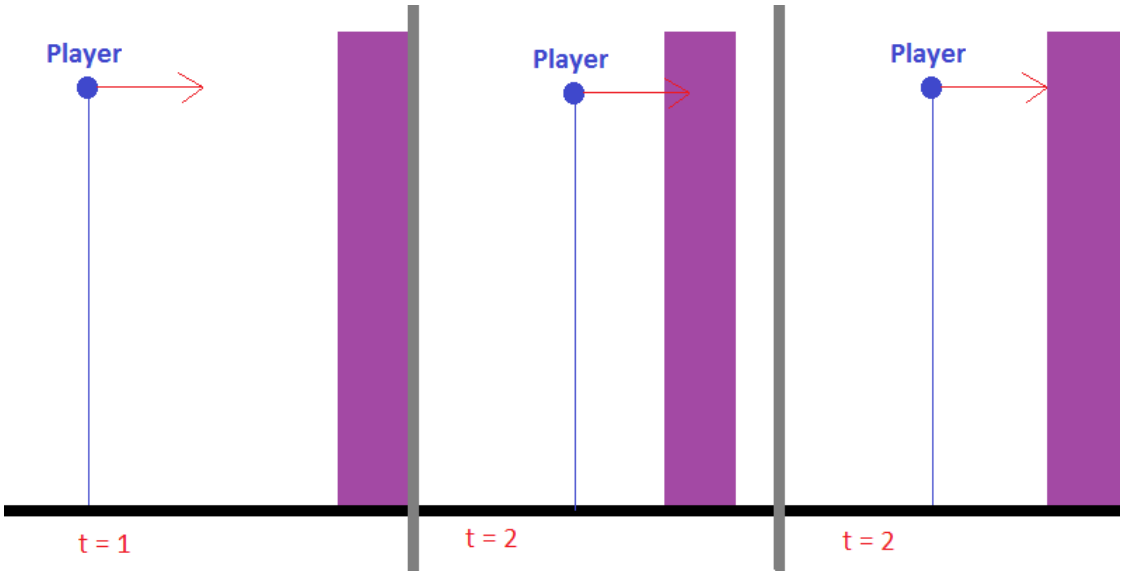


Fig-16: Ray-cast in player direction (side-view) - ray object intersection disabling player movement

As shown by the diagram above, at the specific time intervals, as soon as a ray collision occurs, the movement of the player is disabled (at t=2) and the player is set back to the position at which ray intersection (collision) initially occurs.

So basically the implemented code disables forward movement in the direction a collision (from the casted ray) occurs; once you look towards a collision free-direction, the forward movement will be enabled again (due to no forward ray intersection). Now that we have successfully implemented forward collision detection, the WebGL scene is more realistic due to the fact that we can walk within the boundaries of the 3D model. To implement more robust movement however, we require an implementation of gravity.

Physics (2) - Gravity

Gravitational physics within the scene is a vital component since it makes player movement in the scene more realistic. In fact, any realistic 3D engine obeys the laws of gravity. Implementation of gravity will also allow for natural player jumping within the scene - for PC.

In reality, the forces of gravity are always acting upon us; thus in the game world (walkthrough), the forces of gravity will always be acting upon the player. This will keep the player in a constant *falling state*. Also just like in reality, a person stops falling when they contact the ground; translating the same concept into the game world, a check must be implemented which will stop the player from falling through floor(s) if they surpass it. Implementing this idea into code, we get the following implementation:

```
function applyGravity(player, delta){
        velocity.y -= 9.8 * 10 * delta;
        player.translateY( velocity.y * delta );
        if( player.position.y <= groundHeight ) {
                velocity.y = 0;
                player.position.y = groundHeight;
                canJump = true;
        }
}
```

I decided to place the implementation of gravity within a function. In our case, this function will be called inside the *renderPCMovement(player)* function - before "*prevTime=time*" to be exact. What this function does is at each frame it applies gravity and translates player down. If the player surpasses a specific height (groundHeight) then we reset player velocity and set the player y-position to the height above ground (groundHeight). The groundHeight is basically the world y-coordinate of the floor (of the model/plane) added to a constant height above ground we want player to be. For example this can be declared as:

var groundHeight = model.position.y + 25;

In this case, the player will always be 25 units above the model floor (if it falls down) and thus will never fall below the 3D model floor.

Additionally we also set the jump variable ***canJump*** to be true. Upon pressing the spacebar (on PC), we want the player to perform a smooth jump, we must cater for this in

the ***onKeyDown()*** function which is called via the keyboard event listener. At the pressing of space, we want to change the vertical velocity of the player so that it increases. This is done by the following code:

```
if(space is pressed){
        if(canJump === true)
                velocity.y = 50;
        canJump = false;
}
```

Upon pressing of spacebar, the vertical velocity is set to 50, only if the player can jump. From the previous code in the *applyGravity()* method, the player can only jump when they are in contact with the ground. This simple functionality produces very natural jump movement since when the space is pressed, this causes the vertical velocity to be set, this causes player to move on its y-axis. The player would continue to rise/jump up indefinitely; however, in the *applyGravity()* method, the ever-present gravity force is constantly applied downwards on the player. This produces a natural *parabolic* downward motion which imitates jumping in reality. Using such an approach, a player is able to successfully jump within the scene (on PC).

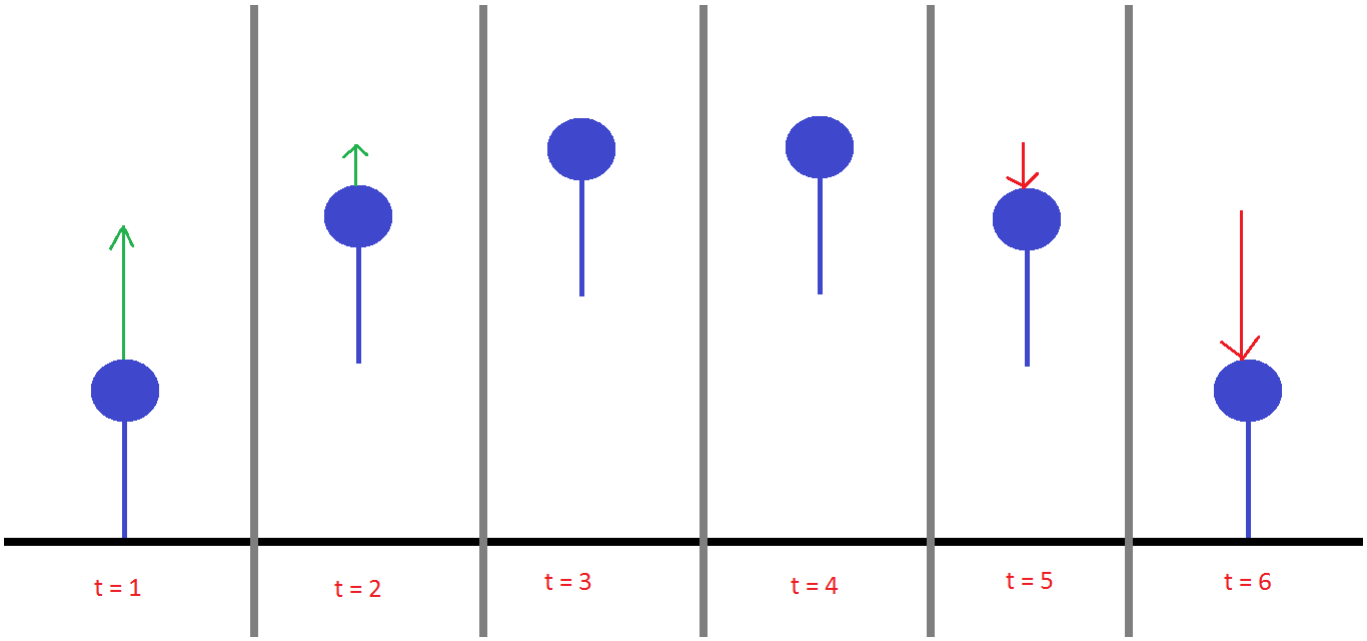


Fig-17: Parabolic effects of gravity on player over period of time - imitating reality

The image above shows the effect of gravity on an object - in our case, its the player. There is a constant gravitational force acting downwards upon the player; when the player jumps, the initial force upwards is very high (vertical velocity); with time gravity decrements this upward motion until there is no upward motion. Then the gravity force (shown in red) is the only movement which acts downwards on the play (ever-present) until the player hits the ground.

EXTRA: Large Plane

When walking through the scene (on PC) I realized that 3D models can also be small and thus the player cannot spawn within the model. This caused the player to spawn in mid-air and fall down continuously (due to absence of ground). Thus I implemented a basic large plane so that the player always spawns on a plane and does not fall indefinitely. The simple implementation is:

```
plane = new THREE.Mesh(
        new THREE.BoxGeometry(5000, 5,5000),
        new THREE.MeshPhongMaterial( { color: 0xffffff } )
);
plane.receiveShadow = true
plane.position.set(0, 0, 0);
scene.add(plane);
```

The plane is 5000 units wide and deep and 5 units high. The important thing to note is that material of choice, in this case it is *MeshPhongMaterial*. This is because the ”Phong” material allows other objects to cast shadows to it - since we want a realistic scene, it is always a good option to allow a plane to receive shadows (which is also declared). Now that we have implemented a large plane, the player will not fall upon initialization (due to gravity).

Now that a player can move within the scene and fall off from ledges/terrain (within model), we still haven’t implemented a way for the player to traverse the terrain. Basically this means that, the player cannot walk up slopes or stairs which may be present in a model. This may not be an issue for single floor 3D models, but however it is a major issue for buildings/structure which have multiple floors. For immersive 3D walkthroughs, the user must able to realistically walk through all floors within a model and fall off edges if they are not in contact - dynamically.

Physics (3) - Terrain traversal

Currently the *groundHeight* is a hardcoded value which the player cannot surpass (cannot fall below that height). Player height and movement is based on this value and will not adjust for other objects within the scene. The result of this problematic scenario is illustrated below:

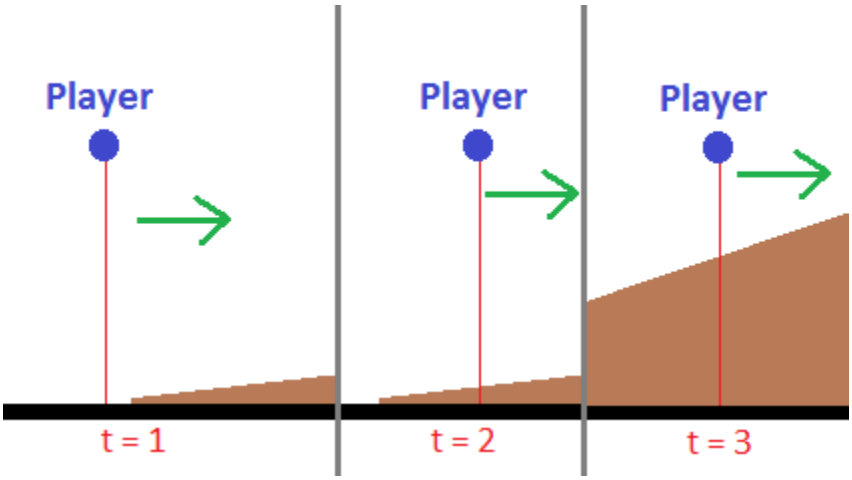


Fig-18: Player failing to traverse terrain - unable to walk-up slopes

As shown above, this issue becomes significantly problematic as one would be constrained to performing walkthroughs on only flat ground - which is not a realistic situation.

To fix this issue, we need to dynamically adjust player height above ground (groundHeight) based on the current terrain the player is on. Thus to perform terrain traversal, we must again use ray casting. To address this issue, a ray will be cast from the player head (camera) to the ground, we will obtain the y-position of the first intersection that occurs with the ray and increment that onto groundHeight, the effect that this will have is that the player could potentially traverse any/all types of terrain.

```
var gravitycaster = new THREE.Raycaster( new THREE.Vector3(),
        new THREE.Vector3(), 0, playerHeight );

function getGravityCollision(pos){
    var downward = new THREE.Vector3(0, -1, 0);
        gravitycaster.set( pos , downward);
    var intersectionsGravity = gravitycaster.intersectObjects
        ( scene.children, true );
        if ( intersectionsGravity.length > 0 ) {
        groundHeight = player.height+ intersectionsGravity[0].point.y;
    }
        groundHeight = 0;
}
```

First we define a ray caster which will be the gravity raycaster. Then we set the rays direction vector (*downward*). After setting the ray position (from player) and direction (to the ground) we again check if the ray collides with anything in the scene; in this case its objects exactly below the player. If a collision occurs (length of list greater than 0), then we get the first list element with which the ray collided with. We obtain the collided elements y-world coordinate and add it to the player height; this allows the groundHeight variable to *dynamically* adjust to the terrain below it - essentially allowing the player to

traverse any terrain. The implemented method (*getGravityCollision()*) is called within the *renderPCMovement()*.

The gravity ray-caster can be visualized by the following diagram:

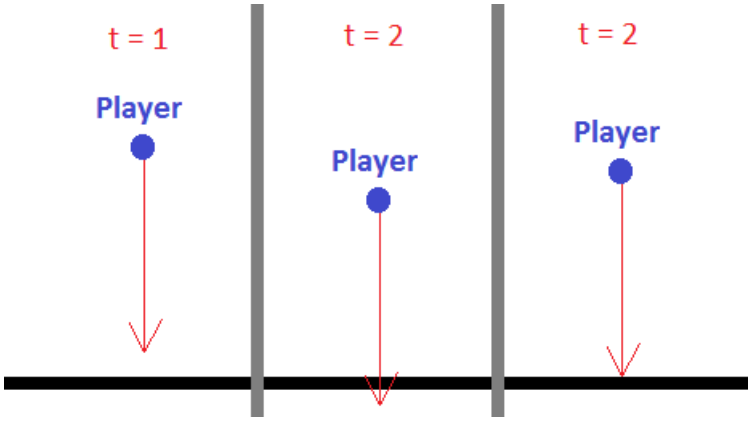


Fig-19: Gravity ray-cast from player to ground (length = player height), Resets player y-position upon ray-ground intersection

As shown by the diagram above, when the player falls, as soon as the ray intersects the ground (and surpasses it), the player position is automatically set to be above ground; which prevents player from continuing to fall. After implementing such a gravity based approach, the player will *get above* any all objects below it. For slopes, this would result in the following scenario:

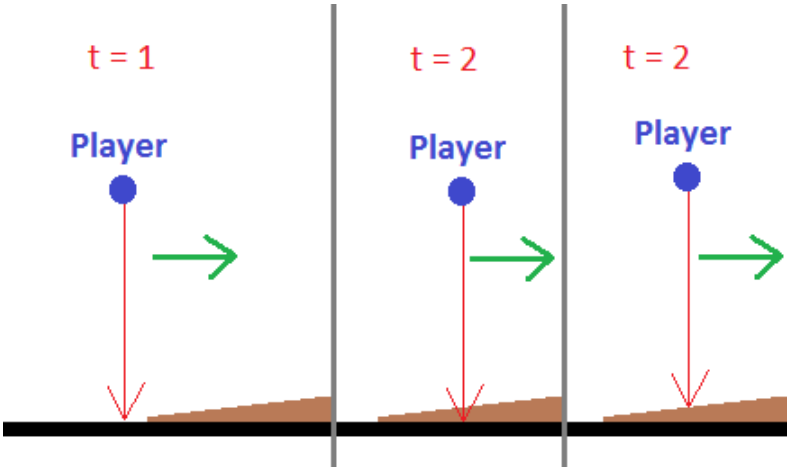


Fig-20: Player not able to successfully traverse all upcoming terrain using gravity ray-cast - can now walk-up slope in Fig-18 (and all slopes/stairs/terrain

Such an implementation will allow the player to naturally traverse any upcoming terrain that is between the player height and the ground.

However, there are some issues with this implementation since we dont want to traverse all terrain. For example, if there is a high gate/wall in front of the player (still lower than

player height), we may not want the player to walk on-top of it. To address this issue, we cast a ray in the players forward direction, but at a lower height (y-position) on the player. The height of this ray is the *stepSize*, which determines the height of forward terrain which can be traversed. An illustration of this ray cast to detect lower collisions is shown in the diagram below:

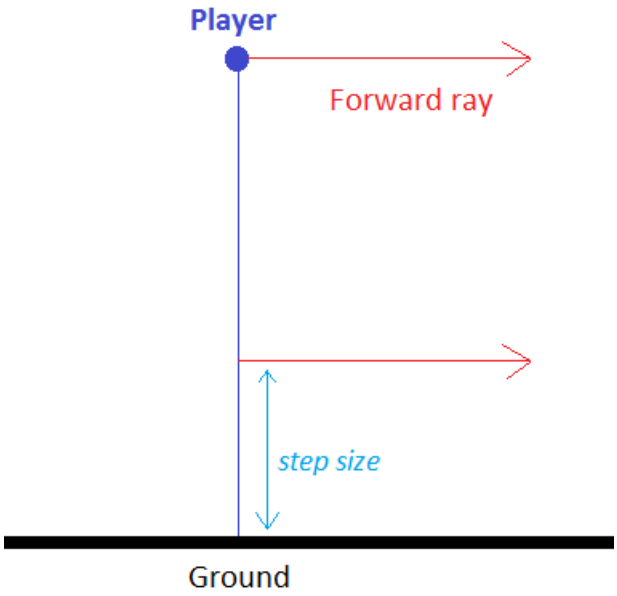


Fig-21: Bottom ray casted at height of *stepSize* which regulates player movement such that player doesn't traverse high upcoming terrain (big stairs etc.) - unwanted behaviour

Such methodology will prevent player from climbing/traversing fairly high objects which should not be passable. The way this works is that the forward - lower collision will detect if the upcoming terrain is traversable, if it is lower than the stepSize, the player will *climb* (or walk on top of it); if not, the player will be prevented from moving forward in that direction (acting like a wall).

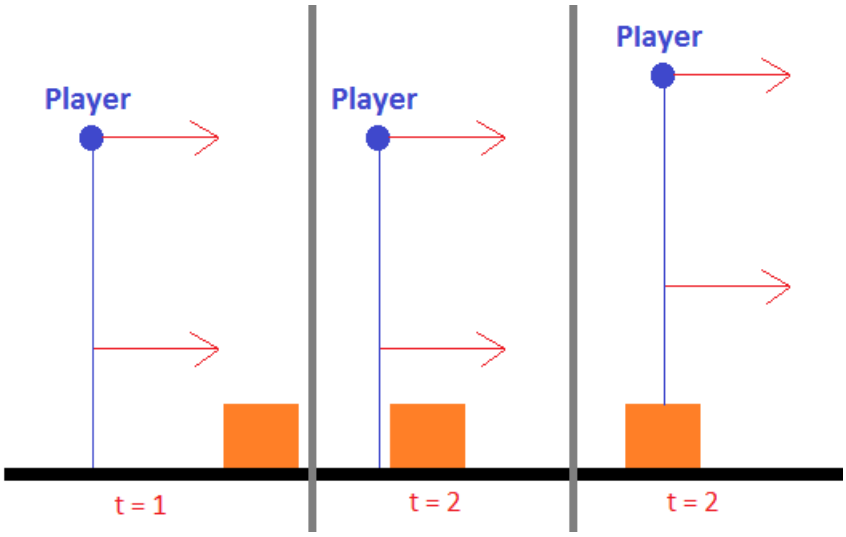


Fig-22: No bottom ray collision with upcoming object - player ”steps over it”

In this scenario, the upcoming terrain is traversable since its height is lower than that of the acceptable *stepSize*.

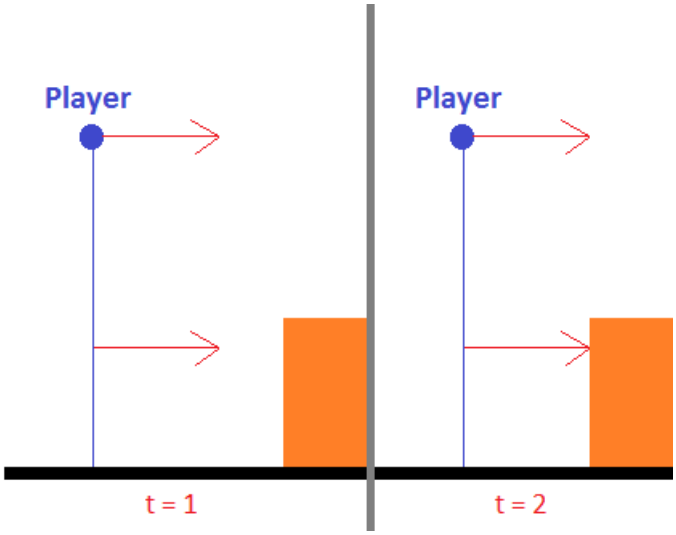


Fig-23: Bottom ray collision occurs with upcoming object - player does not ”step-over” object However, in this case the upcoming terrain cannot be traversed due to the fact that its height is larger than that of the acceptable *stepSize*.

We have successfully implemented basics physics which will allow user walkthroughs within the WebGL scene. The user/client can now use the arrow keys to walk (forward, backward, left, right), jump, walk within models and traverse floors (go up and down) - on PC. The application however is only functional on PC, you can view the webpage on a mobile system but due to lack of controls, you cannot move or perform any interactions. We will now start extending the capabilities of our WebGL solution to support VR to develop the final WebVR solution.

5.7.4 4 - VR smartphone compatibility

In the development of a WebVR solution, we must first study other examples and learn best practices. Fortunately, there are ThreeJS examples and documentation available online which assisted me in the extension to VR. Such examples include the ones on the following webpage:

https://toji.github.io/webvr-samples/

First we must import the correct javascript files from the ThreeJS library into the head of our index.html file. We make the following imports:

```
<script src=".../DeviceOrientationControls.js"></script>
<script src=".../VREffect.js"></script>
<script src=".../webvr-manager.js"></script>
```

The DeviceOrientationControls allow the camera in the scene be controlled by smartphone orientation (via accelerometers). The VREffect allows the camera in the scene to render in stereoscopic view - which can be viewed by Google Cardboard. The webvr-manager allows one to seamlessly transition between VR and non-VR mode on smartphones; thus allowing 3D models to be viewed (and walkthrough) even if one doesn't possess the Google Cardboard.

The following code is used to implement smartphone VR compatibility:

```
var effect = new THREE.VREffect(renderer);
var manager = new WebVRManager(renderer, effect, { hideButton: false,
        isUndistorted: false });
function setOrientationControls(e) {
        if (!e.alpha) return;
    controls = new THREE.DeviceOrientationControls(camera, true);
    controls.connect();
    controls.object = player;
        setInterval(() => { controls.update(); }, 15 );
        window.removeEventListener('deviceorientation',
                setOrientationControls, true);
}
window.addEventListener('deviceorientation',setOrientationControls,true);
```

The first line implements a *VREffect* which will allow the camera to render in VR (stereoscopic view). However, this cannot occur without the manager, the *WebVRManager* uses the current renderer, the effect (stereoscopic) and two more parameters to allow the stereoscopic (and normal) rendering to occur. The hideButton attribute basically allows the manager to render 2 buttons; the first regulates access to and from VR in smartphones; the second allows the user to view scene in fullscreen. These two buttons are shown in the image below:

Fig-24: Buttons/options available on mobile devices (smartphones)

Next we define a function *setOrientationControls(e)* which is implemented as an event listener. The first line returns if the user is not using smartphone, thus meaning that when the code is run on PC, thus function will exit (not function) - so it only applies to smartphones. If a smartphone is detected (when visiting webpage), then we change the previously declared controls variable which was *PointerLockControls* (for mouse based fps viewing) to *DeviceOrientationControls*. We connect these controls and extend its functionality by making it equal to the player. This basically means that now the player view direction is controlled by the orientation of the smartphone. We then invoke the built-in ***setInterval*** at 15ms. This built-in method basically creates a separate thread which continuously updates the controls every 15ms (even after main function has exited). 15 seconds was chosen since we want a minimum of 60 frames per second; for 60 frames to be rendered in 1 second (1000ms) we do 1000/60 which gives us 15ms. This thread will asynchronously update the camera view based on device orientation every 15ms.

We then remove the event listener; this is important because now we do not need to listen for whether a smartphone is accessing the webpage or not. We have already set the desired variables (device orientation controls), updated the player and created a thread which will update player controls; thus we can now remove this listener and the controls will continue to be updated. Removing the event listener is there to reduce computational overhead since we have already established the desired smartphone controls.

However, the scene will still not be rendered in VR because we are not using the manager to render. We must replace the **renderer.render(scene, camera);** with **manager.render(scene, camera);** within the render function. This will give us:

```
function render(){
        requestAnimationFrame(animate);
        manager.render(scene, camera);
        renderPCMovement(player);
}
```

Now we are able to successfully render a scene on PC and smartphone. We are able to control the player direction (and camera direction) via smartphone orientation and we also have the option to view the scene in VR (smartphone).

The following image illustrates the player within a scene in PC:

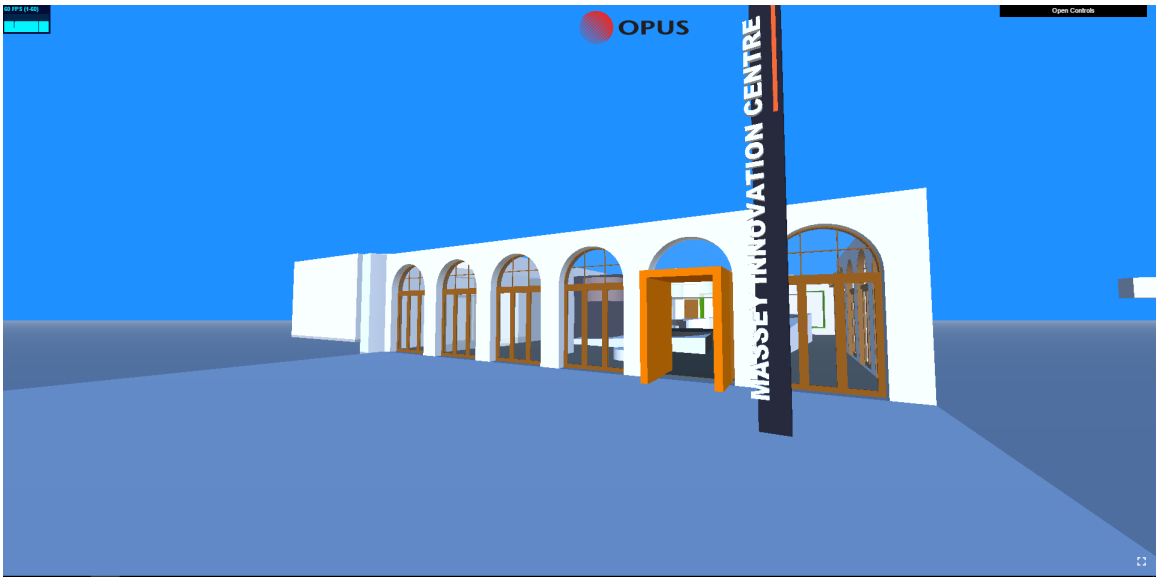


Fig-25: Webpage with loaded demo model (and set parameters) on PC

The following image illustrates the player within a scene in a smartphone (normal):

Fig-26: Webpage with loaded demo model (and set parameters) on smartphone

As shown by the images above, the scene on PC does not have support for VR (bottom-right icon) while the same scene has support for VR - when visited on a mobile device.

Upon clicking the VR icon on the smartphone device, the scene goes into VR mode. The following image illustrates the player within a scene in a smartphone (VR):

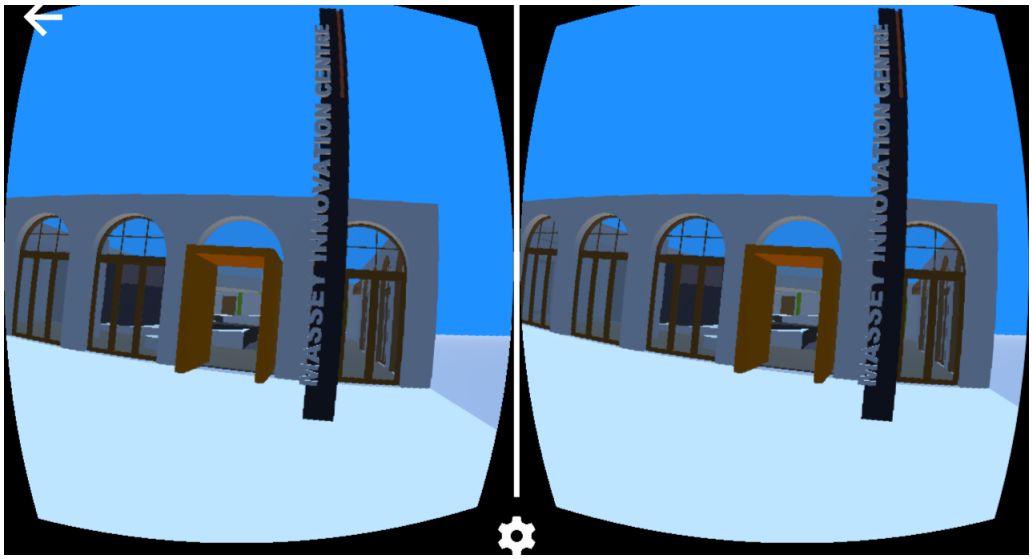


Fig-27: Webpage with loaded demo model (and set parameters) on smartphone in VR

The back button (left-arrow) allows the user to go back to the normal view while the *Options* button (center-bottom) allows the user to set-up which cardboard device they are using (2014 version vs 2015).

We have now successfully implemented a WebVR solution since a user/client can view 3D models on smartphones in VR (Google Cardboard) from the freedom of their homes. However, there are still some issues, there is no *walkthrough in VR*, this is because the player cannot move in VR.

5.7.5 5 - VR controls

When implementing VR controls, we must remember that when we use the Google Cardboard to view the scene in VR on a smartphone, the user will only have access to **one button**. This button is embedded within the cardboard headset, upon clicking/pressing it touches the smartphone screen. Using this concept, we can define a listener which will check if a user has clicked the screen, if so, then the player will move forward in the direction the player is currently facing. To do so we add the following simple code:

```
var vrMovement = false;
document.addEventListener("mousedown", () =>{
        vrMovement = !vrMovement;
        moveForward = vrMovement==true?true:false;
});
```

We implement an event listener which will detect mouse presses, this is also an intuitive way to detect if the smartphone screen has been clicked (via button on Cardboard). If button is

pressed, the *vrMovement* variable toggles its state; if the state is true, then move the player forward based on *moveForward* - which is checked in the *renderPCMovement(player)* method. This code essentially simulates the effect of a user holding down the 'W' or arrow 'Up' key on PC.

We have now successfully completed implementing the WebVR solution since we are able to perform immersive VR walkthroughs on smartphones (via Google Cardboard) within 3D models. This solution will however only work with one model (*sketchup-demo.dae*) due to the fact that the model parameters will have to be adjusted for each 3D model manually. This will require Opus VR team to manually delve into the code to make changes - thus is an inefficient methodology to make changes. One way to address this issue would be to implement online GUI controls which can be used to modify scene variables dynamically and then save the state. This will make scene modification a much more intuitive approach for the VR team, allowing them to dynamically view changes and perfect scene parameters to deliver immersive walkthroughs for their clients.

5.7.6 6 - GUI controls (online scene modification capabilities)

To implement such intuitive GUI controls, once again I searched online for ThreeJS examples. I came across an open source GUI design known as **dat.gui** at:

https://workshop.chromeexperiments.com/examples/gui/#1–Basic-Usage

To use these GUI controls, we need to download the source javascript and import it in the html head tag:

```
<script src="../dat.gui.js"></script>
```

The ThreeJS GUI (**dat.gui**) has a plethora of features such as:

- Event listeners
- Buttons
- Text (displaying)
- Drop-down menus (with selections)
- Variable state toggles
- RGB colour selectors
- Ability to save, load (of JSON) and revert states of all the features

The most prominent feature of *dat.gui* is the ability to save and load the settings of the GUI state. We can use the other features to dynamically set variable values and states; then we can potentially save the state of the modified parameters into a JSON file which will be automatically read upon consecutive webpage loading. Additionally, dat.gui has a feature which allows the VR team to set profiles. This basically allows the VR team to play around with elements in the scene and dynamically adjust values and view their effects within the scene in real-time. The Opus team can then save the state of the scene (with the modified parameters) into a JSON file, these settings will be loaded up by clients who visit the webpage. This is such a powerful feature which will simplify scene modification to a level that wont require the VR team to read/touch any code.

The technical implementation details of *dat.gui* will not be discussed in this report, however the control which the VR team will have on the elements within the scene will be

explained. The goal of this GUI implementation is to allow Opus to have maximum control over scene elements. The diagram below shows the GUI and the elements over which the VR team will have controls (will be able to modify dynamically):

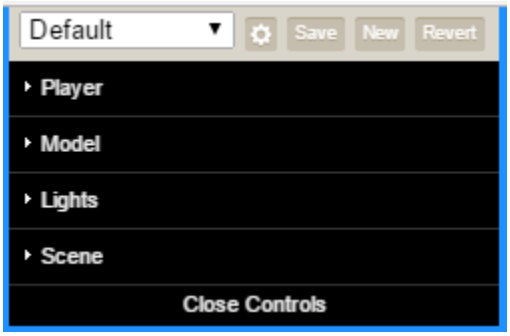


Fig-28: WebVR architecture intuitive GUI controls on important scene elements

Lets first discuss the adjustable **player GUI controls**:

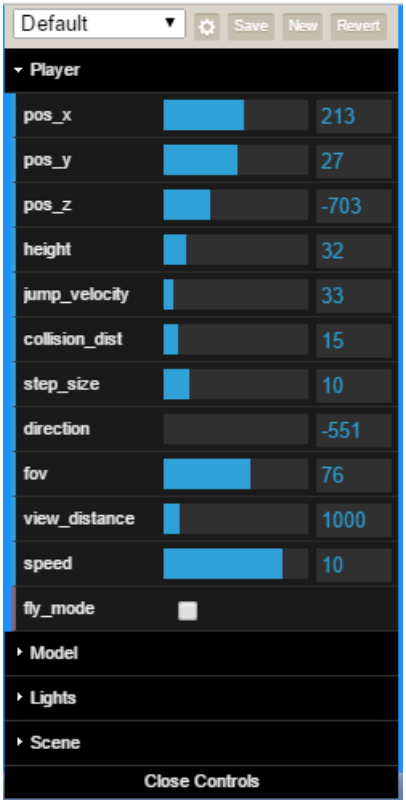


Fig-29: GUI player controls

- The player world coordinates (X, Y, Z) can be manually set in the range of [-2000 to 2000]. Additionally, the user can walk in the scene, the current coordinates will be updated in the GUI which will allow the user to see the current player world coordinates at all times.

- The height can be set in the range of [1 to 200]. This is the player height (camera) above the ground. Upon intersection with the ground, the height of the player will be set to this value above the ground.
- Jump velocity can be set in range of [0 to 500]; this regulates height and speed of jump
- The player forward collision distance can be set based on the *collision_dist* value in the range of [5 to 100]. This basically controls how close a player can get to the wall before a collision is detected (disabling forward movement).
- The *step_size* determines the stepSize which allows the player to traverse terrain. This can be set in the range of [1 to 50].
- The direction controls the rotation around the player Y-axis, this only effects horizontal rotation, not vertical. The direction parameter essentially allows the user to set the direction player is facing when game is initialized. Additionally, this parameter is updated constantly. The range is [-180 to 180] (degrees).
- FOV aka field-of-view can be set in the range of [10 to 120]
- Speed determines player speed within the scene. This can be in the range of [1 to 12] with 1 being slowest to 12 being fastest
- the *fly_mode* is an interesting state variable, this is basically used for debugging if one wished to walkthrough scene without gravity and collision detection (thus essentially flying through scene). Thus the *fly_mode* is an indicator of *physics* within the WebVR application. When turned off, there's no physics (collision detection or gravity).

The model controls are also essential controls which the VR team must be able to modify:

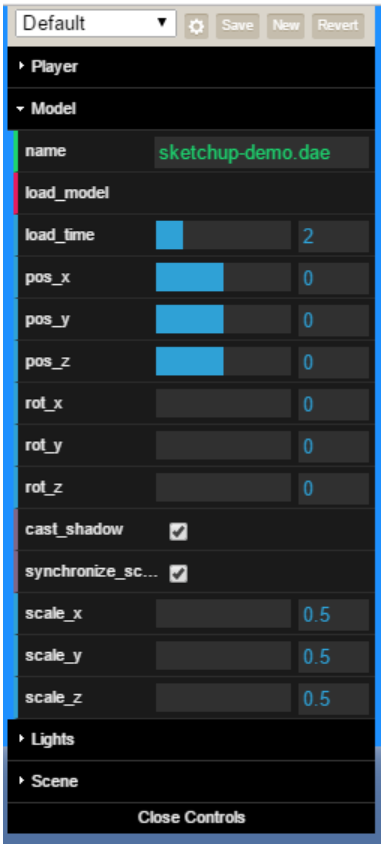


Fig-30: GUI model controls

The first input allows the user to state a name (if they know) of a model and the *load_model* button below allows the user to load a model within the **Model/** directory. This is especially important for the VR team since they will have access to the folders and can dynamically load other models (by name) under the specified directory. The user has full control over the world coordinates position (range[-1000 to 1000]), the rotation (range[0 to 360]) around all axis and the scale (range[0.1 to 100]) in all axis. The state *cast_shadow* controls whether the model can cast a shadow on the plane. The *synchronize_scale* allows the user to scale the loaded model equally on all axis - which makes it easier if one simply wishes to make the 3D model larger (common approach).

Another important element(s) which heavily influence immersion are lights within the scene; thus it is essential that the VR team (or other users) can modify these parameters via GUI dynamically. In the implemented WebVR solution, I have given the user control over 4 lights: Ambient, Hemisphere, Direction and Spotlight (2 of these). Both ambient and hemisphere lights effect the whole scene regardless of their position (source) within the scene. The directional and spotlight light source position matters, their distances from the 3D model (and player) will determine the amount of light that reaches it. Additionally, both these lights are the only sources of light which allows objects in the scene to cast and receive shadows. All light sources have a status (on or off) and an intensity. The intensity is unique/specialized to each light source.

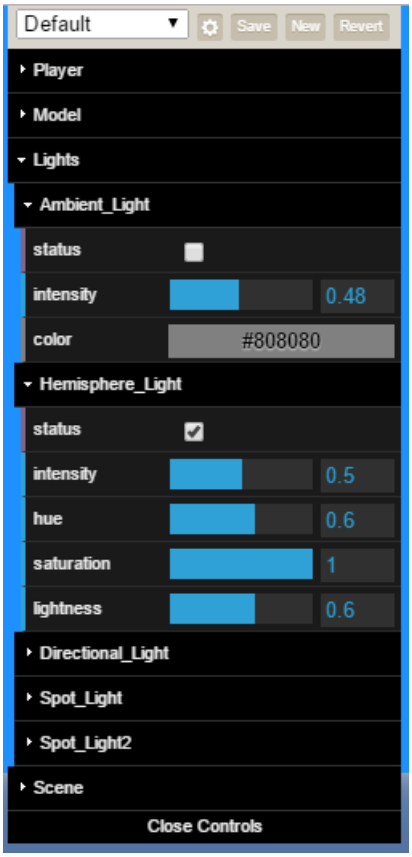


Fig-31: GUI light controls - ambient and hemisphere lights

For the Ambient light, I have given the user access to modify/set the status (on/off), intensity (light strength) and colour (selectable by colour-picker visual) of the light. This light is applied equally to all elements within the scene. For the Hemisphere light, the user can set its status and colour in the HSV/HSL colourspace.

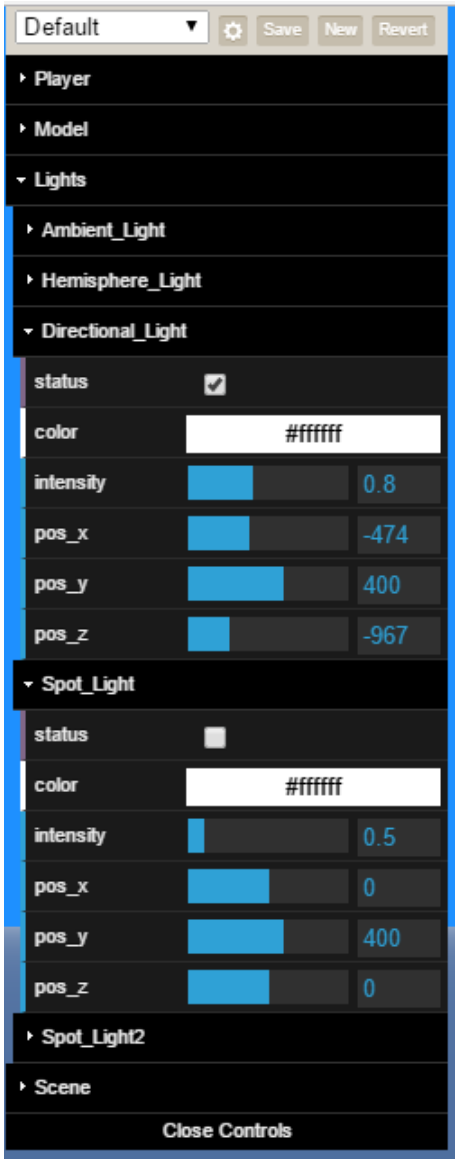


Fig-32: GUI light controls - directional light and spotlight

For directional light and spotlight, in addition to setting status and intensity of the light, the user can also modify their colour(s) (in RGB colourspace) and the light source position. I have also allowed the scene to contain a maximum of 2 spotlights (due to their inherent small radius).

The scene controls are just fine-tuning controls which are not very significant but are still somewhat useful:

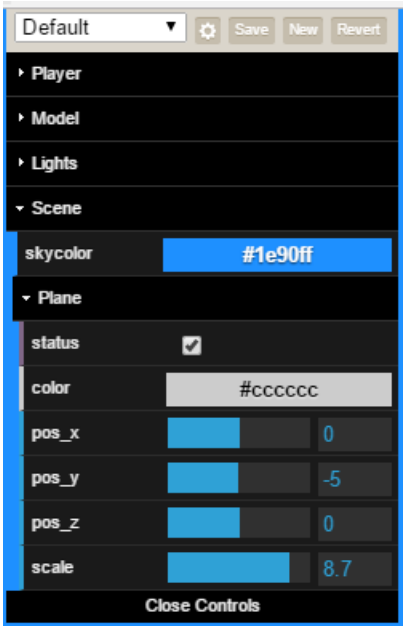


Fig-33: GUI scene controls - sky (background) and plane

In the scene, the user can adjust settings of the background (sky) colour in addition to setting the world position, scale and colour of the plane. The sky and plane colours is quite useful for setting the *theme* of the scene. For example one can darken the sky colour to make it appear as if it is night; on the other hand the user can change the colour of the plane to green to make it seem as if the model is on grass (on a field). The significance of even these basic controls is shown in the example below:

Fig-34: Significance of Scene controls in setting the walkthrough mood (night-time theme)

As shown in the screenshot above, the 3D model is still the same (sketchup-demo), however by just modifying the sky colour and plane colour, we have effectively (very easily) changed the theme of the scene.

Now lets move onto the final most important feature of the GUI, the ability to create, set, save and load profiles. After making changes to GUI elements, the user has the ability to create a new profile by clicking ”New”; additionally they can overwrite the current (in this case ”Default”) profile by clicking ”Save”. After saving the scene, the user can revert changes made to the previous save state by clicking ”Revert”. All of these profile controls are straight forward however, the ability to load and save after closing the session is of most significance. In my implementation of the GUI controls, there are two methods of saving depending on the use case.

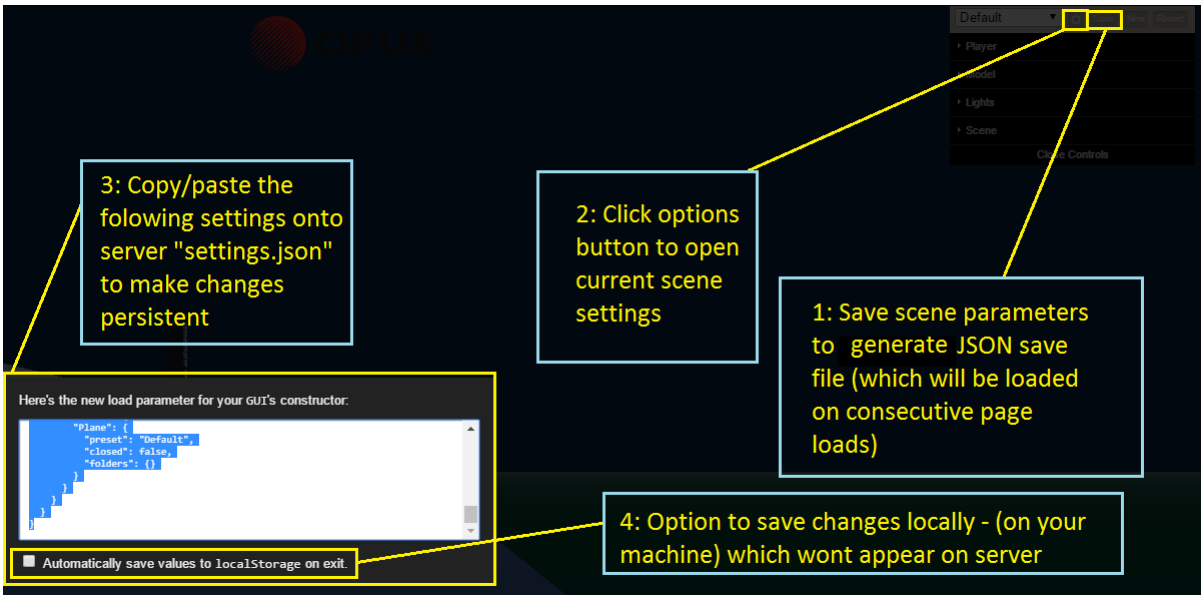


Fig-35: Procedure for saving the current (modified) scene state so that consecutive webpage loads will load the saved scene - option to save locally (browser cache) and globally on web server (via modification of settings.json file)

If the Opus VR team wishes to set model settings and want to reflect these changes globally (on the web server) than after making changes, they must click the ”Save” button. Then click the options button which will show up the JSON code of the states/values all the elements in the GUI. The VR team can select these and copy/paste them to a **settings.json** within the **Resources** folder. Now, whenever an Opus client visits the model webpage, they will view the modified scene; basically allowing Opus to set-up custom designed VR walkthroughs for clients.

If a client wishes to take manual control of elements in the scene, they can also modify the parameters but however they cannot save changes to the *settings.json* file (which resides on webserver). However, they can select the option to ”save values to localStorage”; this allows the clients to have the same control over scene elements as the VR team, create their own immersive VR walkthroughs within their models and then save them in the localstorage of their machine so the changes persist upon exiting. Thus the implementation of GUI controls is a win-win for both Opus and their clients since they have the ability to modify all elements within the scene and save these changes without touching any code.

5.7.7 EXTRA: Performance statistics

To test the performance of the WebVR scene, I implemented a **stats manager** which shows a real-time visualization of frames-per-second (fps) in the scene. This is very helpful since the Opus VR team (and other potential users) can test out which objects (and processes) within the scene can decrease the performance (fps). Such a dynamic real-time performance information allows one to identify technological bottlenecks and constraints within the scene and thus strip-down the insignificant elements to improve the overall ”feel” of the scene and walkthrough.

To access the stats manager we must first import the following script in our html head:

```
<script src="../stats.min.js"></script>}
```

Then we implement the following code in our main javascript document:

```
var stats = new Stats();
document.body.appendChild( stats.domElement );
```

This will give us a stats manager display on the top-left corner of our scene which will show the real-time FPS within the scene. However there will be no data within this display due to the fact that we must update it within our *render()* method; it can be done by placing the following line in the *render()* method:

```
function render(){
        ...
        stats.update();
        ...
}
```

With such an implementation we can now obtain real-time performance statistics and data within the scene. This is shown in the screenshot below:

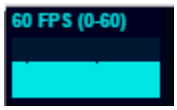


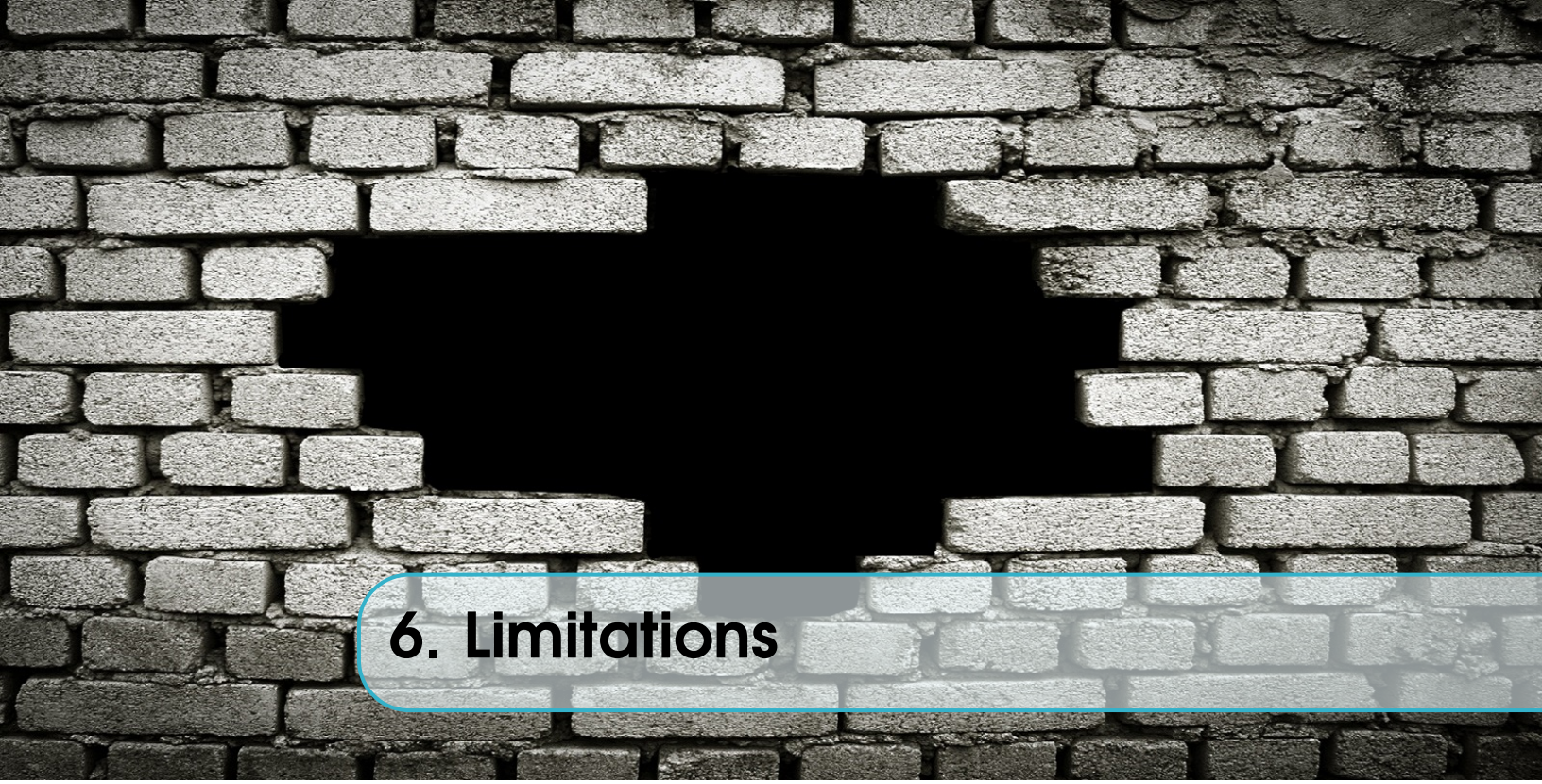
Fig-36: Scene frames-per-second monitor - real-time performance analysis

WebVR solution (wrap up)

By following the 6 workflow steps, I have been able to successfully implement a robust WebVR platform which can allow for immersive and customizable VR walkthroughs for Opus and their clients. The platform can allow walkthroughs in PC (using mouse and keyboard) while also allows VR walkthroughs on smartphones (via Google Cardboard). Additionally, each client can design their very own custom walkthrough's since they can themselves control scene parameters and save the changes locally in their persistent web-caches. This architecture has potential to be one of the most robust and customizable modelling and viewing platforms.

The final solution of this WebVR platform incorporating the *sketchup-demo* can be found at:

http://zsar419.github.io/webvr_prototype/



6. Limitations

Even though we have a successful WebVR solution implemented for Opus, there are some limitations. A few of the major limitations which can have a significant effect (negative) on the VR walkthrough experience delivered will be stated in this section. If I, or a future developer that wishes to continue my work, are able to address these limitations, then the WebVR solution will be much more plausible for commercial use by Opus. Currently the WebVR platform seems like a *"Development"* build due to the following inherent limitations.

Issues with large 3D models

The most significant issue of this WebVR platform is limited compatibility of large 3D models/structures. These are normally indicated by their respectively large file sizes. Large models possess alot of objects with vertices, these vertices are made of vectors which represent the position of the vertex in 3D world coordinates. In other terms, **larger models have a greater polygon count**. Additionally, models can have textures/materials which overlay the surface of the child objects - walls, chairs, roof, doors etc. There are two factors which increase model size; its polygon count and number of textures. The greater the poly-count and textures, the more information/data there is to render and thus the more computationally intensive the rendering becomes. This will lead to problems such as:

- Disproportionately large amount of time taken to initially load the model (compared to smaller 3D models): The user/client will have to wait for longer period of time for the scene to load large 3D models; greater amount of time taken is an inconvenience which may frustrate some users and may not make the experience worthwhile.
- Lower performance: Due to the fact that a large amount of 3D modelling data must be rendered, this will become very computationally intensive for the CPU and GPU of the device. The result of this will be lower rendering the scene with lower fps. A lower fps on (atleast 20-30) may be acceptable on PC and will not decrease the quality

of the walkthrough. However, the effects of lower fps are significantly magnified in VR (on smartphones) and thus a lower fps on smartphones will significantly reduce the quality and immersion of the VR walkthrough experience.

- Magnified performance drops on technologically inferior devices (smartphones): Since I have developed a WebVR platform which is multi-platform (can be accessed on smartphones and PCs), a large model will result in greater discrepancies between the qualities of experience provided between different devices. A smaller less complex model will render at 60 fps across all devices (in ideal conditions), however large 3D models will render faster in PCs compared to smartphones (as highlighted previously). Some users may be surprised to find such large discrepancies if they view the same model on two different devices.

I have briefly tested the 3D models provided by Opus and categorized the models based on file size - from the quality of experience they delivered (FPS). The categories are shown in the table below:

Categories:	Model Size (mb):
Small	0 - 5
Medium	5 - 10
Large	10 +

Fig-37: Categories of 3D models based on file sizes (which results from polygon-count)

Upon further testing of the models, I can confidently state that the greater the file size of the 3D model (larger it is), the more computationally performant it becomes to render it, resulting in a lower quality of walkthrough experience. These issues are further magnified on technologically inferior devices (smartphones). Thus, if one wants to experience fluid VR walkthroughs (high FPS), then it would most preferable to use small models - especially for mobile devices.

No dynamic control over elements within the model (children)

Even though I have given the user control over most of the elements within the scene - via the GUI; there are instances at which the user must perform manual work. For example, when the VR team export a 3D model to collada, they will have no control of the elements within the model after it has been loaded. When the 3D model is rendered on the webpage, they may realize that the model possess some complex 3D objects which are computationally intensive to render (due to greater high polygon count) and thus they may wish to remove that object (which is a child of the model) from the scene. But they will not be able to do this in an intuitive way, the VR team will have to go back to the 3D modelling program (Google Sketchup), remove the object from the scene, re-export it to collada format and

reload the webpage. There are too many steps involved in remove just one child object from the scene which is an inherent issue of this WebVR platform.

I have described this situation with an example of a complex object (car) within a scene compared to the same model having no car.

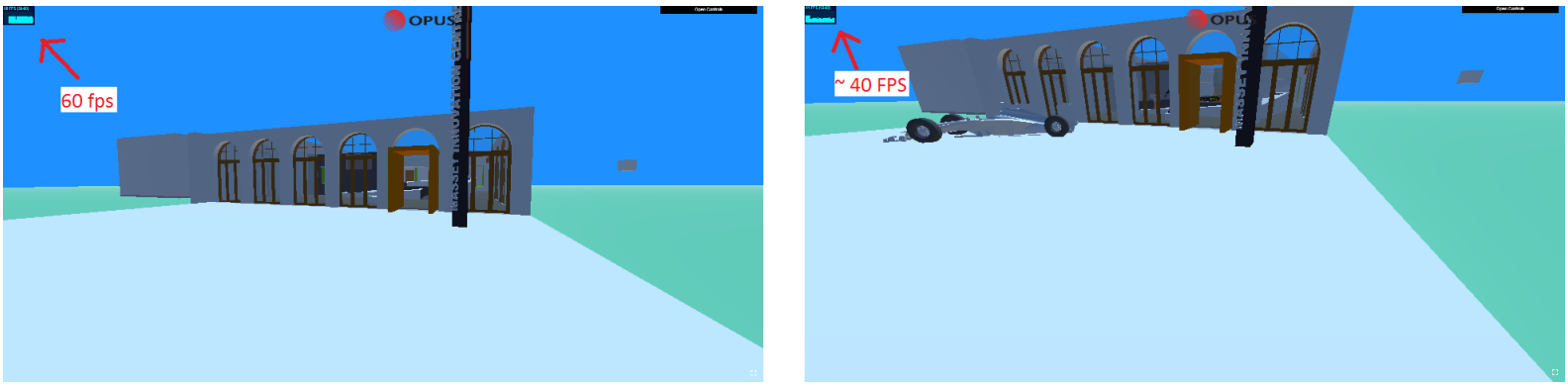


Fig-38: Model with no car (left) vs model with car (right) - same 3D model demonstrating the significance of complex objects (high poly-count car) which causes lower FPS

As you can see from the screenshot above, the 3D model with the complex object (car) has a lower performance (determined by FPS) compared to the 3D model with no car. There has been a 20FPS drop in the scene by having incorporating a car within it (high poly-count). Thus we need a way to address this issue dynamically without having to go back to other modelling software to modify the model.

An improvement will be suggested in the *Potential Improvements* section.

Terrain traversal issues for irregular objects

We had successfully implemented basic terrain traversal within the scene, however there are some situations in which it will not work as expected. This is due to the fact that my implementation for traversing the terrain (walking over objects) is not ideal for walking on irregular terrain/objects. To step over obstacles such as stairs/slopes, Im casting a horizontal ray from the player at the height of stepSize. If there are irregular objects directly in front of the player, which are located above the stepSize and below the player head (such as ledges), will cause the player to walk over them. The issue can be visualized in the illustration below:

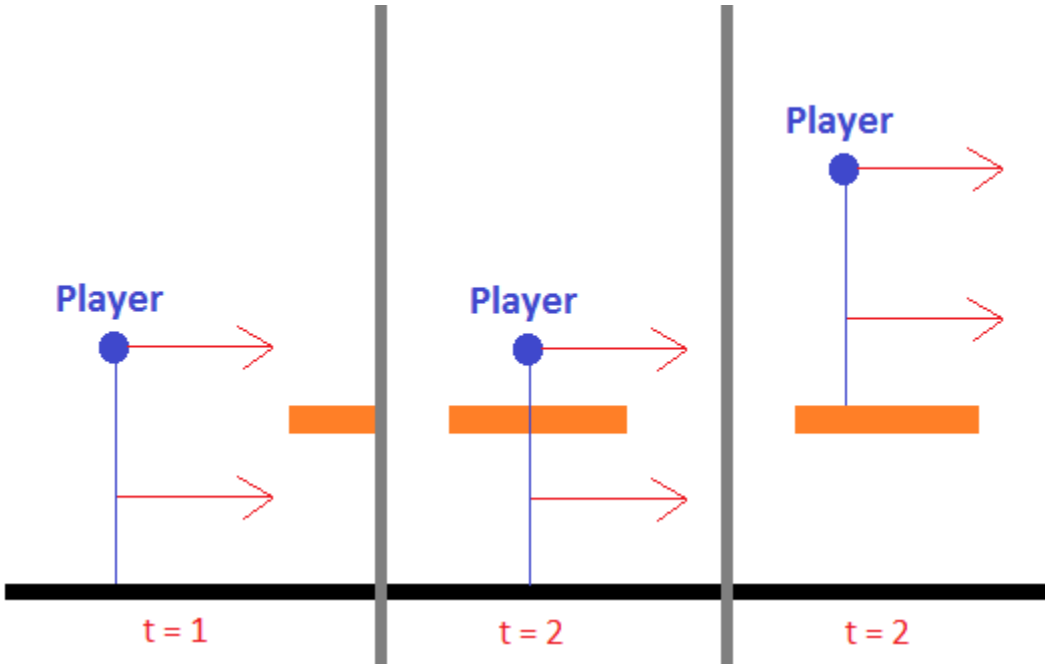


Fig-39: Unwanted Player terrain traversal upon rays failing to detect objects between them

This is an inherent issue which should not occur since we do not want the player to traverse terrain or objects directly in front of player which are higher than the stepSize. I encountered this problem while walking towards a table in the *sketchup-demo*:

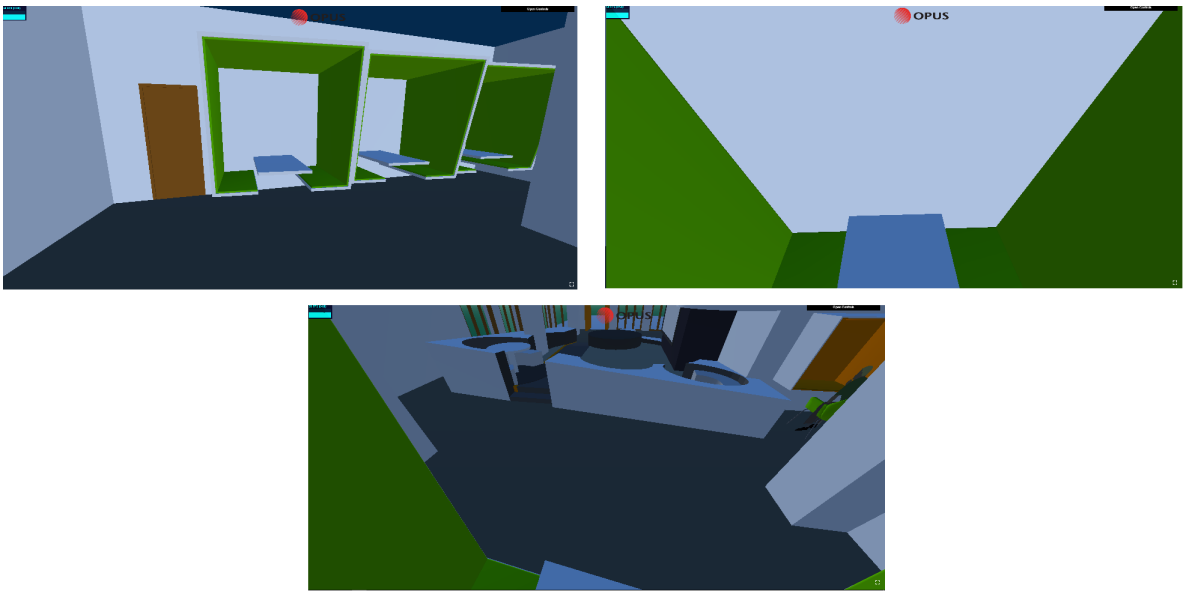


Fig-40: Unwanted terrain traversal (stepping over table) scenario within model walkthrough due to collision detection failure

As the player approaches the table (ledge-like), the forward rays (head and stepSize) will not detect the approaching object (table) and thus the table will go between both the rays. Upon failing to detect the table, it caused the player to get on-top of it (which is unwanted behaviour) - a current limitation.

A potential fix for this issue is also shown in the *Potential Improvements* section.

Model unable to receive shadows properly

Every WebVR scene should try to incorporate use of shadows within the 3D model if possible. Use of shadows make up for immersive walkthroughs within the scene. However I encountered a major problem while trying to implement shadows in my scene. The *sketchup-demo* model within the scene can cast shadows onto the plane, but cannot receive shadows properly. Upon enabling the *recieveShadow* for all child elements of a model (as shown in section 5.7.2 - code), inconsistent random shadows appeared as horizontal lines on the model. This issue is shown in the screenshot below:

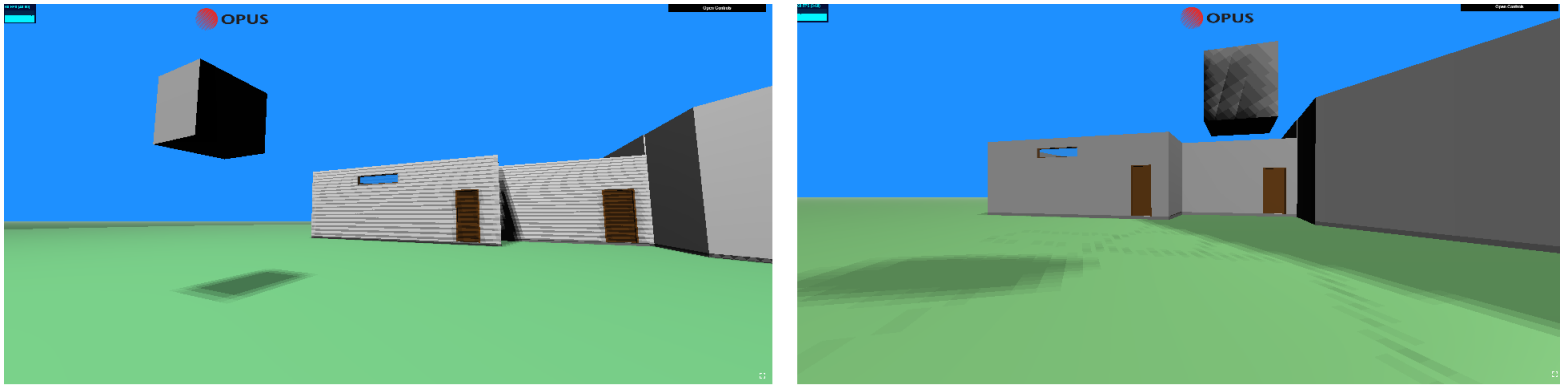


Fig-41: Inconsistencies in rendering shadows within within scene - random shadows on model (left; random shadows on ground and cube (right)

As shown from the images above, there is an inherent problem with shadows. On the left image, there are horizontal shadows being cast on the model - which should not occur (undefined behaviour). On the right image, random lines are being cast on the cube and also on the floor - which also should not happen. Due to such inconsistencies with the shadows, I refrained from incorporating them in the WebVR architecture.

Additionally, from some basic testing, I realized that including a spotlight which casts shadows causes a performance drop (lower FPS) by around 10 frames (per second). This may not be a problem on PC, but on mobile devices the effect is more significant and thus it is preferable that one should avoid using spotlight if they wish to have good rendering performance and fluid experience.

Due to performance drops and shadow rendering inconsistency (on loaded 3D models), I have disabled the model (and its children) from receiving shadows entirely. However they can cast shadows onto the plane.

Lower fps in VR - technological constraint

Even though I have successfully managed to implement a viable WebVR solution, there are some inherent issues with VR walkthroughs on smartphones. One major issue is with VR itself; from performing VR walkthroughs within smartphones (via Google Cardboard), I have observed that there is a decrease in performance within the scene. The observed performance loss is due to a decrease in fps within the stereoscopic VR view. This is a significant issue due to the fact that a lower fps directly correlates to loss of quality and thus decreasing immersion factor when performing VR walkthroughs.

A lower rendering performance is occurring most likely due to **dual-rendering**. Dual-rendering is basically when the renderer has to render two identical (yet slightly distorted) views of the same scene - in other terms, the renderer has to render the scene twice. The renderer is rendering the scene twice due to the inherent stereoscopic view in VR - one for each eye. The only way to address this issue (currently) is to perform VR visualizations/walkthroughs in less complex models (smaller models) with low poly-count.

Cannot adjust VR parameters

Another inherent issue related to the VR aspect is that the user is unable to adjust certain VR parameters. The VR team notified me of this problem when they were testing out my WebVR platform. They stated that would be great to be able to manually control variables related to VR in the GUI. This is also a limitation for the hardware; for example older Google Cardboard headsets (and potentially future ones) may have different eye-separation distance, lens-to-display distance, lens distortion co-efficient and other differences which may not be apparent. This implies that that future Google Cardboard iterations could potentially make the VR implementation obsolete. Thus it is essential that the user can dynamically adjust these VR variables via GUI controls.

This could be something I, or a future developer interested in progressing my WebVR solution, could look into.

7. Discussion

In this section we will comment on the viability and success of the implemented WebVR solution based on performance metrics such as scene frames-per-second and the relative quality of the walkthrough experience on both PC and mobile devices. Success and viability of the proposed WebVR architecture is a relative experience since some may like the platform very much while others may not. Thus the following comments (answers to research questions) are based on my perception of the proposed architecture.

Addressing First research question

The **first** research question proposed in section 1.2 of this report was:

Will the initial WebGL solution be successful in allowing users/clients to perform plugin-less 3D walkthroughs of models on the PC platform?

This question was answerable after the implementation of workflow step 3 (First person camera controls). After incorporating player controls (including physics system), the user was able to walkthrough 3D models on PC without being required to download any plugins. Later after implementing GUI controls, the WebGL walkthrough could be modified according to user preferences due to user having control of most elements in the scene. The success of the walkthrough largely depends on the loaded 3D model. Since we dynamically load models on runtime, the size of the model determines the quality of the walkthrough experience. As stated in the limitations, a larger model will result in a slower scene and thus reduce quality of experience. However since we are just commenting whether the architecture allows for walkthroughs or not; the initial WebGL (for PC) solution does allow users to perform plugin-less walkthroughs within the 3D models.

Addressing Second research question

The **second** research question proposed in section 1.2 of this report was:

Will the extension of the WebGL solution to VR (WebVR) on smartphones (and Google Cardboard) be viable for users to perform 3D VR walkthroughs?

The success of WebVR largely depends on the implementation of the initial WebGL solution since WebVR is an extension (to support VR on smartphones) of WebGL. Due to smartphones being technologically inferior devices compared to PC's, it is highly likely that a slow scene in PC (due to large model) will be even slower on the mobile platform. Thus, again the success and viability of the WebVR platform is largely dependent on the 3D model loaded into the scene. Small 3D models (with low poly-count) will result in more fluid movement which will seem more natural (in VR) while larger models will slow down the fps within the scene and cause delays which may be too disorientating for VR use. Regardless of this fact, the WebVR solution functions optimally with small 3D models, and thus is a viable web based platform to perform 3D VR walkthroughs.

Since my implementation of a WebVR platform is the first of its kind (novel), it was likely that this will have issues and limitations. However my solution is a step forward in progressing web technologies since it allows users to perform walkthroughs on multiple platforms (PC and mobile) and optionally in VR without requiring them to download any plugins. The ability to load in any 3D collada model (of reasonable size) and perform a walkthrough in it (on PC or VR - smartphone) is a powerful visualization tool to convey information and share experiences. This is enough for me to comment that the proposed WebVR solution is successful, viable and has great potential.

8. Potential Improvements

In this section we will discuss potential ways in which we could possibly address the limitations of the implemented WebVR architecture.

Model crunchers

As stated in the *Limitations (section 6)*, we are constrained to using relatively small 3D models (less than 10mb) if we wish to experience an immersive web based VR experience. This is a big constraint because the user may still want to perform a walkthrough of the larger 3D model, be it in lightweight. One way to address this issue is to use model crunchers such as the one offered by **blender**[1]. What a model cruncher does is that it reduces the polygon count within a 3D model (sometimes upto a specified threshold). When reducing the polygon count of a model, the complexity of the model decreases while the overall template is retained. I assume the model crunchers locate areas in the model with high polygon densities and remove a fraction of those polygons to reduce the overall polygon count of the model. Such a solution will allow the user (or VR team) to be able to use larger 3D models (due to decreased poly-count) for performing walkthroughs which may not have been viable before.

Dynamic model object removal

Another way to address the issue of scaling down large 3D models (or any model) is to be able to remove elements dynamically within the scene. As stated before, the VR team has to manually remove/adjust internal model parameters (objects) via the use of other software (Google Sketchup, Revit etc.). One way to ease this issue would be that once the scene is

[1] https://www.blender.org/

loaded, the user can shoot objects within a model to make them disappear. This would work how it does in conventional First-person-shooter games (FPS). A ray would be cast from the center of the screen (on the player) and travel a certain distance (probably as far as camera can see) in the forward direction of the player. Upon ray intersection (with objects - model children), the object would disappear (be removed from the scene) thus allowing the user to dynamically reduce scene polygon count and check for performance changes (via real-time stats display). Additionally, we could extend the functionality to reverse this by adding the destroyed objects to a stack so the user can press [*CTRL* + *Z*] to revert the changes if they wish to do so.

The destroyed/removed objects within a model can be saved to a list/stack (for the respective model) which will be loaded everytime with the scene. Upon loading, the scene will automatically remove the objects within that list, thus allowing the solution to scale and prevent repetitive manual object removal.

Such intuitive controls will greatly benefit Opus and further streamline the online VR experiences they can provide to their clients.

Dynamically adjusting VR parameters (for Google Cardboard)

Another issue which was stated in the *Limitations* was the fact that theres no GUI controls to modify the VR based parameters when performing walkthroughs on mobile devices (in VR). The lack of such controls may significantly hinder the VR experience since different users may require different settings. Additionally, as stated before, newer Google Cardboard headsets may have compatibility issues with the current WebVR architecture and thus make it obsolete. The way that google addresses this issue is that each new release of Google cardboard headset comes with its respective QR-code.

Fig-42: QR-code on Google Cardboard headset - scannable via Google Cardboard app

The picture above shows the QR-code on the Google Cardboard headset; it can be scanned by the Google Cardboard application[2] - which then outputs a config in the smartphone storage. We can potentially utilize this config file (read it from smartphone) to set the parameters of the VR effect in our WebVR application. Thus allowing the WebVR architecture to remain compatible with the latest Cardboard Headsets. On top of this; we could add GUI controls for the VR effect (currently not implemented) which will allow user to have even more fine-grained control over the VR parameters. Such implementations will increase the longevity of our WebVR solution by being compatible with more/all VR headsets.

Improved terrain traversal

As shown in the *Limitations* section, terrain traversal is an issue which can be encountered upon irregular objects/surfaces within the scene. The chances of a user encountering such scenes may be too high (since many people don't use ledges), but however is an issue because it would cause unwanted behaviour. Since two rays are cast (at the players head and below the player - at step size), the player must be able to detect the area between the head and step size (to prevent objects from getting in between. To address this problem, a ray could be cast diagonally from the player head to the ground (until step size). Such an intuitive implementation can be visualized in the illustration below:

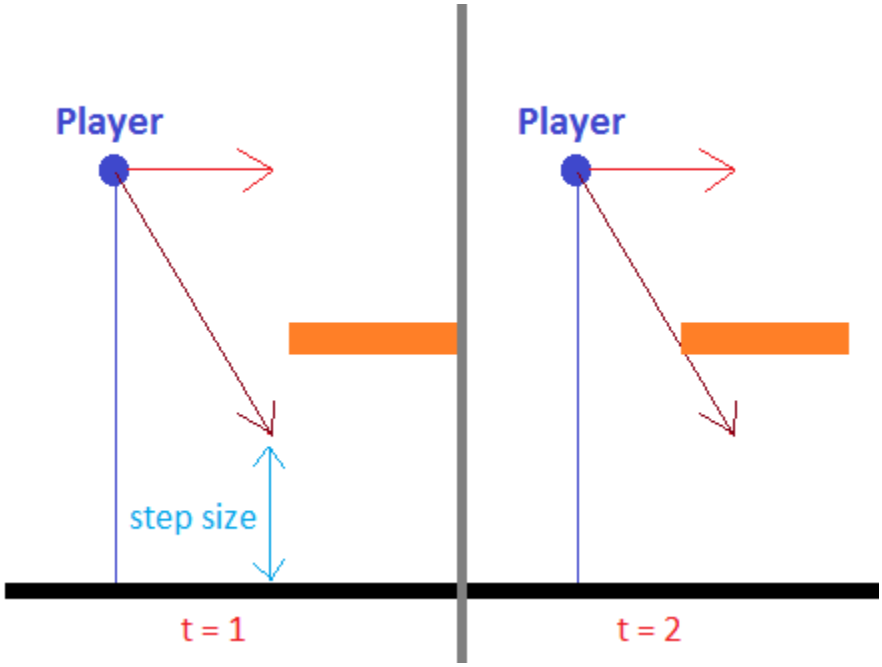


Fig-43: Diagonal ray casted from player to stepSize height which better detects upcoming terrain/objects - addresses issue illustrated in Fig-39; prevents Fig-40 scenario from occurring

[2]https://play.google.com/store/apps/details?id=com.google.samples.apps.cardboarddemo&hl=en

Any collision with this ray would suggest that the upcoming forward terrain is impassable and thus will prevent the player from traversing such terrain/objects - solving the terrain traversal issue from the limitations section.

Greater support for other 3D model formats

Although not a major limitation, but we could expand the capabilities of this WebVR solution so that a user can load in different formats of 3D models, for example (.JSON, .Obj, .3DS, .Mtl, .XML etc.). To implement such compatibilities with other format types would require the solution to have multiple types of loaders such as JSONLoader[3] or ObjectLoader[4] for example and thus may require variables to be adjusted to incorporate such a variety of possible model formats. It is likely that different format types may consist of different file sizes (and/or polygon count) for the same 3D model, thus it is possible that other formats may deliver more smooth (and more immersive) experiences without having to make any trade-offs. This could potentially reduce the limitations of large 3D models (slower scene) and thus allow some models to be viable which previously may have not been.

[3] https://threejs.org/docs/api/loaders/JSONLoader.html
[4] https://threejs.org/examples/webgl_loader_obj.html

9. Conclusion

This report covers a small set of potential benefits presented by virtual reality technologies. Many other opportunities exist, but these will only become apparent through experimentation and familiarization with the technology. Opus-VR allows this experimentation to occur with very little cost or risk and has the potential to significantly change Opus for the better. From the research provided, technological advances in VR has made it a promising technology which can be used for a vast majority of applications. In our case - the Architecture has been developed for 3D modelling of buildings and structures on a website for Opus's clients.

In this report I proposed a web based VR platform which would allow users to perform pluginless VR walkthroughs from the freedom of their homes. The discussed the design decisions of through each step of the implementation procedure and also described the procedure in detail so that one could follow and understand the fundamental underlying concepts. The novel architecture was implemented via utilizing open source ThreeJS graphics library which works on WebGL. I extended the initial WebGL (PC based) solution to incorporate VR, this allows people to perform walkthroughs without the use of any buttons/controls (other than one) on their smartphones via utilizing the Google Cardboard headset. The WebVR architecture is primarily implemented for Opus International Consultants - VR team. If the architecture seems promising, it is possible that they may ask me to implement other features and fix any bugs so that they could use it as commercial software for their clients. I defined a 6-step procedure which was strictly followed for the implementation. I have comprehensively documented the implementation procedure for the main elements within the project; outlining the implementation challenges and flaws within the solution. Additionally I have stated limitations and potential improvements to those limitations (if there were any).

The discussion outlines my thoughts on the viability and success of the implemented WebVR solution. According to some basic testing, I have suggested that the WebVR architecture is a successful solution in allowing users and clients (for Opus) to perform 3D

VR walkthroughs. It performs very well on PCs due to superior hardware and thus can be used as an introductory tool for performing online clientless walkthroughs. However on smartphones, the compatibility is limited/constrained; a user can perform VR walkthroughs in small 3D models (file size), but not larger ones (which may work normally on PC) due to inferior mobile hardware. This implies that solution does not scale well with model size, larger models take longer to load and also reduce scene rendering speed; thus significantly deteriorating the VR experience. Due to these factors, I conclude that my WebVR platform is a successful solution in allowing a user to perform walkthroughs on 3D models (PC), but a *barely-viable* solution for VR walkthroughs on smartphones.

The proposed WebVR architecture is significant because it would advance the current state of the VR solutions - especially online ones. This is because there are yet to be any intuitive web based VR walkthroughs of which the scene parameters could be adjusted and fine-tuned *"on-the-fly"*. I utilized many examples from the ThreeJS webpage in the implementation of this solution. This shows how you can incorporate many simple features to produce a WebVR platform. In my solution Ive integrated model loaders, web based GUI controls (dat.gui), first person mouse controls (and keyboard controls), physics and collision detection (via ray casting), device orientation and VR controls. No example or solution currently exists on the ThreeJS examples page (or on Google) which seamlessly integrates all these technologies/implementations. Additionally, I could potentially open-source a stripped-down version of this project to allow other developers can get an insight onto what is possible with ThreeJS. This way I can contribute to advancing the state of ThreeJS and thus convey the potential of what WebGL can do.

9.1 Significance for Opus

Implementation of new and disruptive technologies such as virtual reality will help to position Opus as a leader in the delivery of high-tech, innovate and efficient services. My WebVR architecture may be such a technology which will leverage Opus's advantage and make them stand out compared to their competition. If Opus were to successfully integrate the WebVR into their arsenal of software modelling and/or visualization tools; then they would be one of the first to allow clients the freedom to perform easy and intuitive pluginless VR walkthroughs within 3D models - on the web. Utilizing novel and innovative technologies such as the one proposed allows Opus to stay ahead of its competition. Even if the proposed solution may not be one that is viable for commercial use; Opus can further develop and improve upon it to make it viable since *the architecture is promising*.

10. Future Work

According to the statistics shown in *section 4.3.1*, VR use cases highlighted in (section 2.5), and many other indicators such as the release of *Google Daydream headset*[1], emergence of VR-ready PCS/Laptops[2,3], and general advances in VR technologies, the future of VR looks promising. The implemented WebVR platform is a novel architecture which would further advance the state of VR and deliver intuitive introductory experiences for those who wish to ”try” out these technologies. The architecture is however heavily dependent on external factors (which the WebVR solution has no control over).

Extension to multiplayer

Extension to support multiplayer capabilities is on a web-based clientless platform is an interesting concept. There is literature which is currently exploring this in area of research for the ThreeJS library - as described in section 3.3. Additionally, on top of research, development efforts being made to advance ThreeJS multiplayer capabilities. There are tutorials[4] to set-up basic multiplayer capbilities on ThreeJS using sockets; theres also emergence of multiplayer real-time games such air-hockey[5]. Such a multiplayer extension on the proposed WebVR architecture can for example allow many people to experience Walkthroughs within the same 3D models. This will make the experiences more collaborative and fun; allowing users to make more insightful decisions based on discussions with others. If we in-corporate such capabilities, this can allow Opus-VR team to arrange virtual web-based VR tours which will greatly enhance VR experiences for their clients. In the general sense, any sort of multi-player experience expands the life-span of

[1] https://vr.google.com/daydream/
[2] https://www3.oculus.com/en-us/oculus-ready-pcs/
[3] http://www.geforce.com/hardware/10series/notebook
[4] http://marcostagni.com/threejs-socketio-multiplayer/
[5] http://blog.artillery.com/2012/05/realtime-multiplayer-3d-gaming-html5.html

platforms and makes them more fun/interesting to use.

Web-based dynamic FPS platform

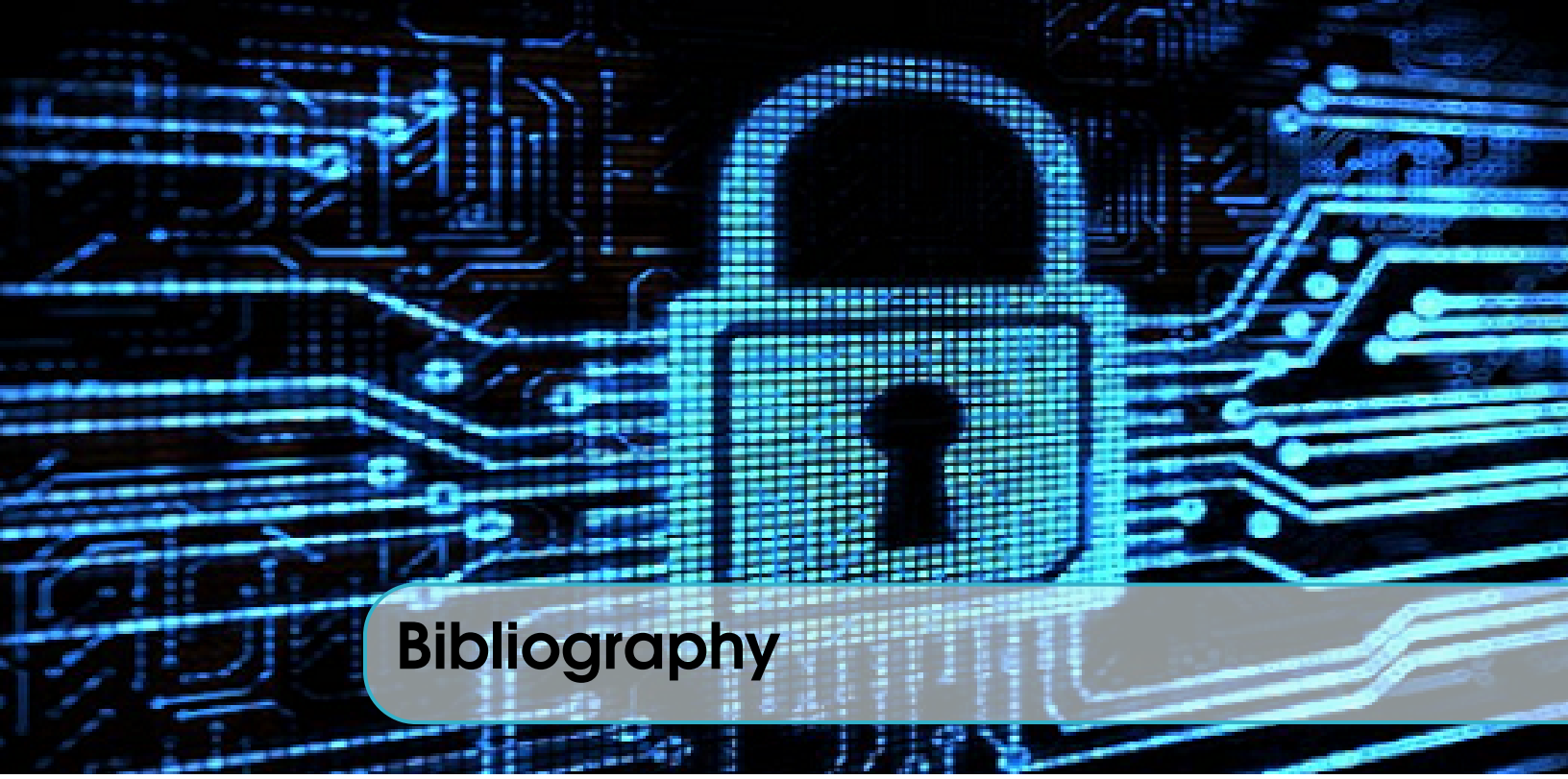
The implemented architecture already supports the main first-person-shooter elements such as first person controls, movement and physics (gravity, collision detection, terrain traversal). All one would need to add to this platform is the ability to shoot (via guns), moving entities or enemies, some basic level design and give the user an incentive to play. These are the basic elements which could be quite easily implemented to produce a FPS online pluginless game-engine. The levels could for example be models loaded up into the scene. Players could play on both mobile and PC platforms. To top it all, we already have support for VR - which further immersifies the experience. Implementation of these features (and multiplayer) would effectively make this a web-based game engine.

Advances in WebGL and ThreeJS

WebGL was released in March 2011 while development of the open source ThreeJS graphics library (which works on top of WebGL) began in April 2010.[35],[36] Due to the infancy of both these technologies, the libraries/APIs will not rival performance to those of already established PC/smartphone VR applications; but however what these technologies do offer is the ability to view (and in our case walkthrough) complex 3D renderings without having to download any software. Additionally, WebGL is multi-platform and only requires a supported browsers (very common) to be run. Further development in WebGL, greater browser-support and optimization and maturity of the ThreeJS library (from a development perspective) will naturally allow the implemented WebVR architecture (which is based on these technologies) to advance and become more viable/successful.

This implies that the proposed WebVR architecture is a self-advancing, self-scaling web platform that will improve along with the improvements of WebGL, Web-Browsers and rendering hardware (devices - PC/smartphones). The architecture will mature with time (assuming development efforts in web technologies and hardware continue).It is likely that over a period of time, a 3D model which may not have previously been viable (for rendering), may become viable due to the advances of the stated technologies - without changing the implemented architecture itself.

For this BTech project, I implemented a basic WebVR prototype to display the potential of such novel web based VR technologies. The WebVR solution may not be successful in a commercial setting (for Opus-VR), however it does show that development in this field could give promising results. The report (and implemented solution) can give researchers (or developers) an insight on web based VR technologies which may lead to increased development efforts in the industry (for this field). Such success will allow my work to be continued and be put to better use for the web and VR industries.



Bibliography

[1] Home Opus NZ. (n.d.). Retrieved June 01, 2016, from http://www.opus.co.nz/

[2] History Of Virtual Reality - Virtual Reality. (2015, December 25). Retrieved June 01, 2016, from http://www.vrs.org.uk/virtual-reality/history.html

[3] Thank you from Oculus Oculus Rift: Step Into the Game. (n.d.). Retrieved June 01, 2016, from https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game/posts/1458224

[4] Retrieved June 01, 2016, from http://vrfocus.com/wp-content/uploads/2014/10/VRKarts_1.png

[5] Mobile VR pros and cons - UploadVR. (2015, November 09). Retrieved June 01, 2016, from http://uploadvr.com/pros-cons-mobile-vr

[6] Spec Comparison: The Rift is less expensive than the Vive, but is it a better value? (2016, April 05). Retrieved June 01, 2016, from http://www.digitaltrends.com/virtual-reality/oculus-rift-vs-htc-vive/#:w8h_c272yfhtma

[7] HTC Vive Review: A Mesmerising VR Experience, if You Have the Space - Road to VR. (2016, April 05). Retrieved June 01, 2016, from http://www.roadtovr.com/htc-vive-review-room-scale-vr-mesmerising-vr-especially-if-you-have-the-space-steamvr/

[8] Retrieved June 01, 2016, from http://blogs-images.forbes.com/erikkain/files/2014/06/Cardboard-2.jpg

[9] Retrieved June 01, 2016, from http://www.samsung.com/us/explore/gear-vr/assets/images/desktop/GearVR_Hero_Gold.png

[10] Retrieved June 01, 2016, from https://cdn2.vox-cdn.com/thumbor/eIQgeQJ0845uTw0cSiN-Hf0qV5s=/0x0:1920x1080/1280x720/cdn0.vox-cdn.com/uploads/chorus_image/image/48508449/oculus-rift-image_1920.0.0.jpg

[11] 5 Virtual Reality Accessories That Could Change Gaming Forever - Games Under Pressure. (2014, January 13). Retrieved June 01, 2016, from http://gamesunderpressure.com/features201417best-vr-accessories/

[12] Will Virtual Reality Be Next Big Thing Across Technology? (n.d.). Retrieved June 01, 2016, from http://finance.yahoo.com/news/virtual-reality-next-big-thing-224200034.html

[13] 8 of the Top 10 Tech Companies are Invested in VR and AR. (2016, March 08). Retrieved June 01, 2016, from http://uploadvr.com/8-of-the-top-10-tech-companies-invested-in-vr-ar/

[14] Daydream is Google's Android-powered VR platform. (2016, May 18). Retrieved June 01, 2016, from http://www.theverge.com/2016/5/18/11683536/google-daydream-virtual-reality-announced-android-n-io-2016

[15] The Emerging Virtual Reality Landscape: A Primer. (n.d.). Retrieved June 01, 2016, from http://www.slideshare.net/BDMIFund/the-emerging-virtual-reality-landscape-a-primer?next_slideshow=1

[16] Roberts, A. (2014, February 20). Introduction to the HTML5 Canvas Element. Retrieved June 01, 2016, from https://www.sitepoint.com/web-foundations/introduction-html5-canvas-element/

[17] WebGL Specification. (n.d.). Retrieved June 01, 2016, from https://www.khronos.org/registry/webgl/specs/1.0/#1

[18] Can I use... Support tables for HTML5, CSS3, etc. (n.d.). Retrieved June 01, 2016, from http://caniuse.com/#feat=webgl

[19] WebGL Stats. (n.d.). Retrieved June 01, 2016, from http://webglstats.com/

[20] T2H EXPORT WEBGL — SketchUp Extension Warehouse. (n.d.). Retrieved June 01, 2016, from https://extensions.sketchup.com/en/content/t2h-export-webgl

[21] Build once deploy anywhere. (n.d.). Retrieved June 01, 2016, from http://unity3d.com/unity/multiplatform/

[22] HOWTO: Setup a basic scene. (n.d.). Retrieved June 01, 2016, from http://doc.xenko.com/1.6/manual/getting-started/howto-setup-a-basic-scene.html

[23] = Three.js / examples. (n.d.). Retrieved June 01, 2016, from http://threejs.org/examples/#webgl_geometry_cube

[24] ArchiVision Architecture (DK2) - NEW Kitchen demo - SDK0.4.0. (n.d.). Retrieved June 01, 2016, from https://forums.oculus.com/vip/discussion/11272/archivision-architecture-dk2-new-kitchen-demo-sdk0-4-0

[25] Unity3d is not ready to build WebGL/HTML5 web games with. (n.d.). Retrieved June 01, 2016, from http://blog.doublecoconut.com/unity3d-webgl-is-it-viable/

[26] Three.js / examples. (n.d.). Retrieved June 01, 2016, from http://threejs.org/examples/#webgl_effects_stereo

[27] Google Daydream Launch Date Confirmed — VRFocus. (n.d.). Retrieved June 01, 2016, from https://www.vrfocus.com/2016/05/google-daydream-launch-date-confirmed/

[28] Amadeo, R. (2016, May 18). Gear VRs for everyone! Google turns Android into a VR-ready OS: Daydream. Retrieved June 01, 2016, from http://arstechnica.com/gadgets/2016/05/android-vr-os-gets-a-virtual-reality-mode-and-vr-ready-smartphones/

[29] Wojciechowski, Rafal, et al. "Building virtual and augmented reality museum exhibitions." Proceedings of the ninth international conference on 3D Web technology. ACM, 2004.

[30] Grantcharov, TEODOR P., et al. "Randomized clinical trial of virtual reality simulation for laparoscopic skills training." British Journal of Surgery 91.2 (2004): 146-150.

[31] Noguera, J., and J. Jimnez. "Visualization of very large 3D volumes on mobile devices and WebGL." 20th WSCG international conference on computer graphics, visualization and computer vision. WSCG. 2012.

[32] Muennoi, Atitayaporn, and Daranee Hormdee. "3D Web-based HMI with WebGL Rendering Performance." MATEC Web of Conferences. Vol. 77. EDP Sciences, 2016.

[33] Chen, Bijin, and Zhiqi Xu. "A framework for browser-based Multiplayer Online Games using WebGL and WebSocket." Multimedia Technology (ICMT), 2011 International Conference on. IEEE, 2011.

[34] Biamonti, Davide, Jeremy Gross, and Daniel Novak. "High performance WebGL for visualization of conjunction analysis." SpaceOps 2014 Conference.

[35] Khronos Releases Final WebGL 1.0 Specification - Khronos Group Press Release. (n.d.). Retrieved October 24, 2016, from https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification

[36] M. (n.d.). Mrdoob/three.js. Retrieved October 24, 2016, from https://github.com/mrdoob/three.js/releases?after=r4

The future lies in Virtual Reality...