UNIVERSITY OF HULL

Development of a Networked Virtual Reality (VR) Tanks game with Procedural Generation and Ray Tracing

being a dissertation submitted in partial fulfilment of the

requirements for the degree of

Master of

Computer Science for Games Development

in the University of Hull

by

Sohail Turner

January 2022

Word Count: 25,058

1 ABSTRACT

The paper describes the process of development of procedural, ray tracing and network features in the context of a VR game. The paper details how a methodology based on a game development process can be used to produce a VR game with networking, procedural, and ray tracing features. With a discussion of how these features can be implemented to improve a player's experience within a VR environment. The approach for the various procedural content is based on well-established algorithms, with an optimization to ensure the best-generated outcome for user experience. The paper discusses some approaches for procedural content which can be generated in conjunction with a VR game.

2 ACKNOWLEDGMENTS

I would like to acknowledge the generous amount of support provided by Huw Owen and Dr Qingde Li for all the generous amount time, help and information in helping me completing this project to a high standard.

I would also like to thank my family for the generous amount of support and guidance in helping me complete the project and help me get me through such a stressful time.

3 TABLE OF CONTENTS

4 INTRODUCTION

The main objective of the project was to research and explore the feasibility of developing a Virtual Reality (VR) networked game with ray-tracing and procedural generation in the context of a smooth player experience. The paper will explore how these modern techniques can be utilized together, to create such an experience, without ruining immersion or being limited by the hardware used.

Within the past decade, there has been a huge increase in the development and research of Virtual Reality technology, with a “fifth of Facebook employees are working on VR” (Byford, 2021), leading to a big increase in VR game development. With this development, there seems to be a lot of interest in implementing game techniques such as ray-tracing and procedural generation, but there seems to be a gap in hardware capabilities (as raytracing is so computationally expensive) and implementing these techniques while maintaining smooth FPS (frames per second). Because of this limitation, there seems to be a lack of research in the area which is evident with the lack of research papers and games utilizing the techniques together within a VR environment. This paper aims to contribute to bridging this gap and provide the basis of understanding to others who wish to develop a game of a similar caliber.

5 AIMS AND OBJECTIVES

The primary goal of the project undertaken was to understand the concepts of VR, network programming, ray-tracing, and procedural generation to create a VR multiplayer tanks game. It will provide a foundation for an in-depth analysis of integrating various APIs (application programming interface) to produce the game as well as justifications for algorithms utilized. The application developed follows a games development methodology. The focus of the project was the implementations of the various aspects discussed, with less focus on the gameplay as such the overall project is a prototype with the development of a framework to enable future development.

Objective 1- Development of a VR game

The first objective is vital as this will be the base of the project, to enable the other objectives to be achieved. The overall aim of this objective is to research and understand how Unity and its various APIs work, as well as how the Oculus package integrates and utilizes Unity and its APIs. Furthermore, a basic game will be implemented to allow for the basic game functionality and facilitate easy implementations of the other objectives especially the networking.

Objective 2- Implementing procedural content

This objective aims to research and conceptualize ideas for procedural content which can be implemented within the game. Sometime will be taken to study and understand how to implement procedural content in Unity alongside which algorithms can be used for creating various procedural objects. Testing of the content will be done to make sure it doesn't cause any bottleneck to the VR headset and stay at a constant FPS (Frames Per Second) rate to enable a smooth experience.

Objective 3- Implement various shaders

The objective will be to implement some shaders which utilize raytracing/raymarching algorithms to produce some real-time-photorealistic images alongside other shader techniques to visually enhance the game while remaining within the limitations of the Oculus Quest.

Objective 4- Implement and integrate networking

This objective will have a focus on researching and implementing networking into the game. As there isn't a focus on developing a custom networking system, research will be undertaken to determine the best available networking API for use within Unity. Sometime will be taken to research and understand how games developed with Unity use such systems to provide the ability to support network interactions between clients alongside achieving smooth player interactions. While understanding how these can be adopted to work in context of a VR game.

Objective 5- Adapt the procedural content to have networking functionality

This objective will focus on how to ensure that any randomness created from the procedural content, is not unique to one player but is shared with every player. As such research will be done to grasp an understanding of how games that employ similar aspects do so and how these such games ensure every user sees the same outcome in the procedural content.

RESEARCH QUESTION

A research question can be posed to indicate the direction in which the project will be developed. The research question: “What is the feasibility of integrating modern gaming techniques of Ray Tracing and Procedural Generation into a Virtual Reality networked Game while maintaining a smooth user game experience?”

6 LITERATURE REVIEW

Within the current generation of games technology, there are ever-advancing technological capabilities that have allowed for such complex aspects in games, that now there is a sense of when a new game comes out, it needs to employ these techniques to just grab a user's attention, let alone buy it. But with these advancements, there comes negatives, such as cost of development and development time, and if it is feasible within the engine being used to build and run it. An example of technology allowing for the development and implementation of one such technique is Ray tracing but at the cost of computational power.

This chapter provides an overview of techniques that may be deployed within the game. It will give a brief overview of the history of networking, raytracing, procedural generation, and virtual reality in terms of games, and how they have developed up to this point. Regarding Ray tracing, there will be an overview of the basic algorithm.

NETWORKING

As technology now being able to fully utilize and implement full ray-traced graphics, the feasibility and capability bring into question its ability to implement a ray-traced system in conjunction with a networked system where both users experience and witness the same outcome of effects. With old and new games, they utilize various robust networking systems', with modern games employing more robust and complex systems. As such the systems developed have allowed for such user interaction that users experience the same effects outcomes alongside maintaining a smooth interaction.

Network History

Within the history of networked multiplayer games, there is an extensive history, from how early games used PLATO network for support to modern games (Madhav and Glazer,2016) which employ highly sophisticated and robust systems with dedicated servers to support not just LAN gaming, but the ability to connect and play with a huge amount of people from different corners of the globe. For early-type games, Local-Area Networks brought a whole new experience as it created the ability for multiple players to connect locally and play. Doom is seen as the pioneer of this system. With the ever-increasing expansion of the Internet, there was huge popularity in online games such as Unreal (1998) along with consoles supporting the ability for online games in the early 2000s.

Star siege Tribes

Star siege Tribes is one such example of a multiplayer game with the ability for vast amounts of players to be within the same game session. The game employed a client-server network architecture in which the server coordinated the game and relayed the information between each client through this server (Madhav and Glazer,2016), which also contributed to catching cheaters. It worked through the information received by the server from the player such as a position change. If the servers predicted movement of the player was substantially different from the position change information received, this gave a good indication to the sever there was cheating occurring thus ensuring the person would be removed from the game. This architecture is still used widely within the games industry because of the security and adaptability it offers.

Within the game, it functioned through a variety of managers and techniques to support a smooth interaction and synchronization between each client. Detailed in Joshua Glazer and Sanjay Madhav's book: Multiplayer Game Programming Architecting Networked Games, they detail this architecture of Starship Tribes. They describe it as “At the lowest level, the platform packet module abstracts sending packets over the network. Next, the connection manager maintains connections between the players and the server and provides delivery status notifications. The stream manager takes data from the higher-level managers (including the event, ghost, and move managers), and based on priority, adds this data to outgoing packets. The event manager takes important events, such as “player fired” and ensures that this data is received by the relevant parties. The ghost manager handles sending object updates for the set of objects deemed relevant for a particular player. The move manager sends the most recent movement information for each player”. This description shows a robust implementation of an architecture which allows for a huge number of players to communicate and sync up to allow for interaction between each client without loss of experience.

Age of Empire

Another early game that implemented a different networked architecture is Age of Empire (1997). Within the game, it used a deterministic lockstep model which worked through a peer-to-peer manner. With each connected client receiving commands from the game which are then evaluated on each peer independently, with the game using a turn timer to synchronize the game over each client. The turn timer was used to store several commands over a period before sending them over the network, as such each time several commands were sent, they were not executed until two turns later which provided enough time for each peer to receive each turn command along with sending each turn command. A further note to be made in this architecture, is the important aspect of the deterministic simulation run on each peer. An example of this is the pseudo-random number generator which would be synchronized across each client. (Glazer and Madhav, 2016)

With each of these architectures, it shows that implementation of a robust system is possible even in early games, now as we are at a time where technology is fair greater in terms of capability, the use of highly advanced networking systems is possible within VR while maintaining a smooth user experience.

VR, AR, AND MR OVERVIEW

For virtual reality headsets, there are 3 distinctive ways in which user interaction in a computer-generated 3-dimensional virtual environment occurs. The basis of each approach is built on the concept of Virtual Reality (VR), which allows for interaction with a virtual environment in a seemingly physical way. But the other ways take it further as they offer support for interaction with real-world objects within the simulated environment. These other variations are known as Augmentation Reality and Mixed Reality.

Each uses specialized equipment to support this interaction and the equipment usually consists at the bare minimum, a headset with a screen and lenses to view the scene, speakers, and controllers, alongside a tracking system in the headset so ensure tracking of controllers and movement of the head-mounted display (HMD). The lenses for each eye act as the display, and through this it creates a stereoscopic 3D effect with the use of stereos sound.

Augmented reality (AR) differs from VR as it allows for an object that resides in the real world to be enhanced by computer-generated perceptual information along with multiple sensory modalities used such as auditory or haptic. (Schueffel 2017). The paper "Current status, opportunities and challenges of augmented reality in education" discusses AR and defines it as "a system that incorporates three basic features: a combination of real and virtual worlds, real-time interaction, and accurate 3D registration of virtual and real objects". (Hsin-KaiWu Silvia Wen-Yu Lee Hsin-Yi Chang Jyh-Chong Liang, 2013)

Mixed reality (MR) can be defined as the merging of real and virtual worlds to produce new environments and visualizations, where physical and digital objects co-exist and interact in real time. (Milgram, Paul & Kishino, Fumio,1994). Unlike Augmented reality, Mixed reality is a hybrid approach of reality and VR allowing for it to occur within either the virtual or real world.

With each variation, there is a fundamental idea that applications created for these environments offer a way to simulate and create new experiences within a virtual setting for use in a variety of applications ranging from video games to educational purposes. In recent years, huge advancements have been made regarding VR technology. As such there have been numerous inventions regarding these, one such being the Oculus Quest as the first fully standalone headset that supports itself. With these innovations, large amounts of technology companies are diverting huge investments into virtual reality technology research. As such VR is now a seemingly approachable idea as a new technology that could be capitalized on. One such company at the precipice of this research is Oculus, who since releasing their first VR headset in 2012 (Kumparak, 2014) have only invested more and more into research to keep ahead. Alongside being acquired by Meta (formerly Facebook) for around 2.4 billion dollars in March 2014. (Wood, 2020)

Figure 1: Oculus VR prototype (Pesce, 2015)

With these acquisitions, it has spearheaded huge amounts of investments and research into Virtual Reality technology.

The History of Oculus

Oculus's first headset was called the Oculus Rift and offered a wired experience using a high-end computer which subsequently reduced some interest in it as most people didn't own a computer that could handle it, which ultimately helped Oculus start investigations for standalone headsets.

The first standalone headset they released was in collaboration with Samsung and was called the Samsung Gear VR supported for Samsung Phones. This headset as such inspired Oculus to release their

first standalone headset called the Oculus Gear. (Oculus,2018) but compared to the Oculus Rift the capability of the Gear was underwhelming. This just furthered the idea that they needed to develop a standalone headset that could rival wired headsets like the Rift, in terms of capability. With this further innovation, the rewards of their research were paid off as they were first to release a high-end standalone headset, the Oculus Quest which rivaled the wired headsets in not just capability but also the functionality. In terms of headsets, it is revolutionary as it offers users the ability to use the headset wherever and not be confined to a room with a computer. It still is relatively weak in comparison to wired headsets as they also use the hardware of the computer, they are connected to but overall, it still offers a lot. Furthering this research, Oculus has released a second Quest with huge improvements, such as the new model which offers a more responsive experience overall, thanks to an improved RAM and chip specs. It also has a higher resolution display, which is a 50% sharper than its predecessor. Combined, these upgrades make the Oculus Quest 2 experience even more seamless and immersive. (Lynch, 2021). The specs for the Quest and Quest 2 can be found in appendix A. These improvements have helped the Quest to even more rival wired headset experiences along with introducing hand tracking which utilized AR technology and allows for the use of the headset with just hands. Further developments within VR headed by Meta, are the announcements of the Metaverse, in which they stated they are making it a more social media-type platform.

Metaverse

The metaverse is described as the next iteration of the internet, which supports online 3D virtual environments through a multitude of platforms such as a computer, VR, or AR (Newton, 2021). There are examples of limited versions of Metaverses already present in the current climate such as VRChat or Club Penguin as these games incorporated many aspects of social interactions in a virtual environment.

Figure 2- Example of a limited VR Metaverse - VR chat (Hunt, 2018)

Figure 3- Example of a limited metaverse, Club Penguin (Kelly, 2022)

Metaverse related problems

The focus for ambition for metaverses in the current technology is addressing the technological limitation within modern VR/AR devices and the ability to expand these metaverses into new different spaces such as business, education, and retail. Concerning metaverses there has been quite a bit of criticism as some companies like Facebook are using the method as PR (public relations) approach to building their image. Further concerns are seen in data protection or user addiction stemming from the current climate of challenges regarding social media and video games. With this, it can bring some ethical questions regarding the development of a metaverse application. With recent problems and accusations that have plagued Facebook, there is rising concerns that with the announcement of Facebook using targeted adverting, that there will be even less personal privacy alongside misinformation becoming more rampant. Furthermore, an algorithmic approach to tailor worlds based on persons specific beliefs, could have future consequences as users' perceptions of reality may be distorted, as the algorithms used are created to maintain and/ or increase engagement.

Implications of the Future

Microsoft is another company to show similar interest as they acquired the VR company AltspaceVR in 2017. With plans to integrate and support metaverse features within Microsoft Teams. (Warren,2021). A real-world implementation in terms of a Metaverse is in the announcement by the South Korean government, where they announced the creation of a national metaverse alliance to build a unified national VR and AR platform. (Sharwood,2021)

RAYTRACING AND RAYMARCHING

The idea of creating virtual worlds can be further enhanced using techniques within the field of 3D computer graphics such as Raytracing. Ray tracing is a technique utilized to create photorealistic images by modeling light transport and employing it within rendering algorithms to generate digitally enhanced pictures.

Raytracing is capable and very useful as it allows for the simulation of optical effects such as reflection, refraction, shadows, and motion blur. (Shirley and Morley, 2003). As it utilizes tracing the path of a ray, the same principle can be used regarding any physical wave with approximate linear motion. An example is sound waves, which can create a greater immersive experience within a video game regarding sound design as a simulation of reverberation and echoes, (Kastbauer, 2010) to achieve a greater game sense. Especially within a VR video game where sound plays a bigger role in a player's immersion compared, to non-VR.

The feasibility of raytracing algorithms within real-time applications has only become possible within recent years because of the computational cost and visual fidelity associated with them. The use of raytracing has been widely used and accepted within still computer-generated images, as well as visual effects utilized within TV and movies, as this use of ray tracing tolerates a long time to render. But with speed being a critical aspect in rendering each frame and the fidelity of hardware used for real-time applications for example video games, raytracing is less suited for this. Within major films, global illumination was achieved in computer-generated imagery with additional lighting, so is seen as faked raytracing. But ray tracing-based rendering eventually enabled physically based light transport to create true raytracing-based effects without the need for additional lighting. Some examples of early films taking advantage of path tracing are Monster House (2006), Cars (2006) (Pixar Animation Studios, 2006) and Cloudy with a Chance of Meatballs (2009) (Robertson, 2009).

Figure 4: Ray-tracer sharp shadows illustrated from Cars. (Pixar Animation Studios, 2006)

However, with some big strides and development within hardware regarding hardware acceleration, the use of real-time raytracing is becoming the norm within new commercial graphics cards along with graphics APIs following suit. With these strides, developers are allowed to employ hybrid approaches rendering with raytracing and rasterization. Implementations of ray tracing at the time of writing vary in the application and use of ray tracing or RTX. Such as Battlefield V which uses them for soft shadow and

reflection while engines like Unity or Unreal have introduced experimental implementations with the idea for them to be phased in and be useful to any sort of developer. But no suitable solution for VR as VR hardware doesn't use or take advantage of hardware acceleration.

Ray-Tracing History

The concept of ray tracing was first described by Albrecht Durer and employed in an invention known as Durer's door (Hofmann,1990) The invention consisted of a thread being attached to the end of a stylus while an assistant moves along the contours of an object to draw. The thread would pass through the door's frame and a hook acting as the projection center and cameras position, which in turn caused a ray to form on the thread and point away from the hook (Durers,1522) (Curtis et el. Luecking, 2013)

Figure 5: "Perspective Machine" illustration published in Albrecht Dürer's The Painter's Manual, 1525. (Dürer, 1525)

Further accomplishments regarding raytracing, was accomplished by Arthur Appel in 1968. He was the first to employ a ray-tracing algorithm for digital rendering and coined the term ray casting (Appel, 1968) He utilized the concept of ray tracing on a computer to generate shaded pictures. In this approach, the key difference between rasterization and ray tracing is the ability for the intersection between rays and objects to be analytically defined. This can be seen in how techniques in CAD software allow objects to be rendered to be defined mathematically. (Giugiu,2020).

Regarding modern ray-tracing techniques, Turner Whitted introduced new concepts related to ray tracing within his paper, Improved Illumination Model for Shaded Display (Whittard,1980). In this paper, he outlined how ray tracing can be used for not only determining visible geometry, but in how it can be used to compute shadows, refractions, and refractions through tracing additional rays towards the light

source and other objects within the scene. In turn, the results can be used to compute the final color of each pixel in a scene. With this approach even though realistic-looking results can be generated, it has no basis on using a physically accurate model for light transport. Hames Kajiya introduced a new form of the ray tracing algorithm known as path tracing to solve this problem and designed the mathematical model to define the intersection of light with surfaces and support simulation of a range of optical phenomena. (Kajiya, 1986) The algorithm focused on tracing the path of the light around the scene, with information being collected about the order in which objects were hit to produce the final color. With this introduction, it has become the standard rendering algorithm used within the industry for light transportation. A further point to be made is regarding the direction used for tracing the rays, which unlike in real life where photons travel from the light source and bounce around its surroundings until the ray hits the observer (e.g., eye), whereas to use the most optimized solution, ray tracing algorithms, deploy the ray of light from the observer of the scene to the light source. This technique is regarded as backward tracing. (Arvo,1986)

With the use of the technique, various technical implications must be considered. One such implication is that no object culling (frustum culling) can be used for the frame as a ray may be reflected behind the camera, other objects, or out of camera view, as such to be able to perform culling, the scene must be loaded into memory at run time to then apply any culling. This brings problems as GPUs have a limited amount of dedicated memory compared to the CPUs available shared memory, even though raytracing would be very suitable to be parallelized. A solution to this would be to stream the scene from the shared memory or disk but would introduce latency such that any benefits are outweighed by the negatives. (Giugiu,2020)

Further disadvantages of ray tracing in offline renderers, is the time needed to render a scene, even though they now make use of GPGPU (general-purpose computational unit) technologies (Direct Compute, OpenCL, etc.) and provide the ability for the algorithms to make use of the parallel power of the GPU(Kurachi,2011). Even with this ability, render times can be quite big in comparison to real-time engines. Another disadvantage that can be discussed is the digital noise produced in the final render which to reduce or even remove this noise the ray casts must be increased. Or in conjunction with a denoising algorithm used on the rendered scene frame but with this further complexity adds further computational time needed for the result to be rendered. A further disadvantage is even though rays can be cast in parallel with O(N) alongside using culling to improve the render, rays have poor cache coherence even more as each ray can travel in random unpredictable directions and hit various geometries.

Ray Tracing algorithm overview

Turner Whittard (1980) ray tracing combines eye and rays to light alongside recursive tracing to determine the color of each pixel. A breakdown of the process of ray tracing in the context of a rectangular viewport with a sphere intersection, but there are further shape intersection algorithms: triangle, cube, and plane. Each intersection equation uses the ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t\,\mathbf{d}$$

The sphere equation:

$$\mathbf{p} : (\mathbf{p} - \mathbf{c})^2 - R^2 = 0$$

Through combining both equations the intersection of the sphere be solved as:

$$(\mathbf{o} + t\,\mathbf{d} - \mathbf{c})^2 - R^2 = 0$$

Which can be written in the form:

$$f(x) = ax^2 + bx + c$$

Which will enable the roots to be found through the following equations:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$\Delta = b^2 - 4ac$$

Where the letter Δ (delta) is the discriminant and the sign of it will indicate whether there are two, one or no root to the equation. With Δ>0 representing two roots which in context of the ray intersection signifies the ray has intersected the sphere twice, Δ<0 signifies no intersection and Δ=0 representing 1 intersection, which can mean the ray originated within the sphere. Figure 6 illustrates these outcomes.

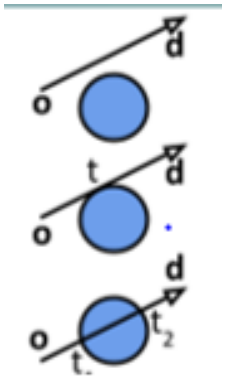


Figure 6: Sphere intersection outcomes (Ray tracing illustration- WikimediaCommons, 2017)

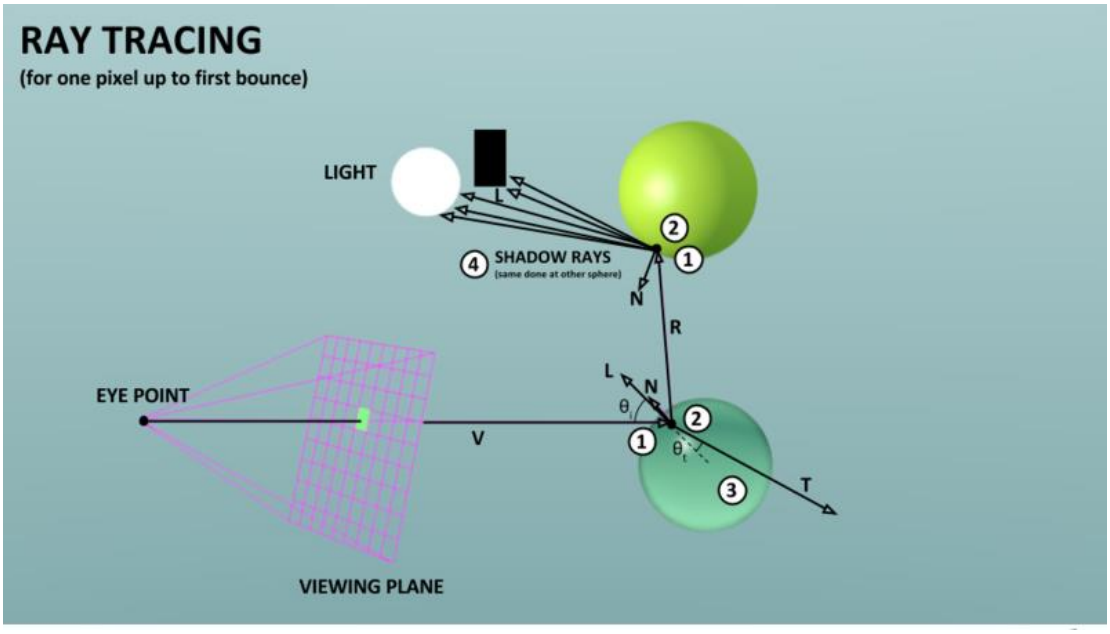


Figure 7: Illustration of the ray-tracing algorithm with a sphere intersection for one pixel (up to the first bounce) with a Phong-Blinn illumination. (Ray tracing illustration- WikimediaCommons, 2017)

Ray marching

Within the realm of raytracing, there is an overlap with procedural generation, with a technique within the Ray-Tracing family known as Raymarching. Raymarching allows for an implicit modeling technique that represents geometric shapes as mathematical functions. These functions are known as Signed Distance Functions (SDF) and provide an arbitrary point in the scene, specifying when a ray hits this position, a shape should be produced. With SDFs allowing for objects to not be defined by a data structure, they offer some key advantages. One such is the data efficiency for GPUs, as their computational capabilities evolved faster than their memory bandwidth, this enabled the purely mathematical SDFs to become competitive against 3D-texture/voxel/octree-based SDFs (Quizlez,2008).

Raymarching and Ray tracing parallels

There are some parallels when it comes to raymarching and ray tracing, as each technique, a position based on the camera, with the use of a grid which represents a pixel in the output image, rays are sent from the camera through each grid point. The difference lies in how the scene is defined, as such these changes affect how the intersection between the view ray and scene are found. Whereas in ray tracing, the scene is defined by explicit geometry: sphere, triangles, etc. With the use of different geometric intersection tests used to find intersections between the view ray and scene. But with ray marching, the SDFs define the scene. Any intersection between the scene and view ray is found, by starting at the camera and moving along the ray, bit by bit, where at each step a determination question is asked of "Does the SDF evaluate to a negative number at the current point". If it does evaluate to negative, an intersection has occurred, and the ray is stopped, else there is a continuation along the ray till the maximum number of steps along the ray are completed.

Ray marching algorithm

The ray marching algorithm is illustrated by the diagram in figure 8. The diagram outlines how raymarching is in terms of sphere tracing (Hart, 1996). Where the maximum step is used, as it is known not to go through the surface. With each step occurring through marching forward along the ray direction in increments which then uses the distance provided by the SDF to the surface to determine the shortest distance between the current point and a surface.

It illustrates, let p0 be the camera, with the blue line indicating the direction of the ray cast from the camera through the view plane, with the first step taken larger than others. The ray is marched along by the shortest distance to the surface. The steps recursively continue until the point of the ray has hit the surface, in this case, p4.

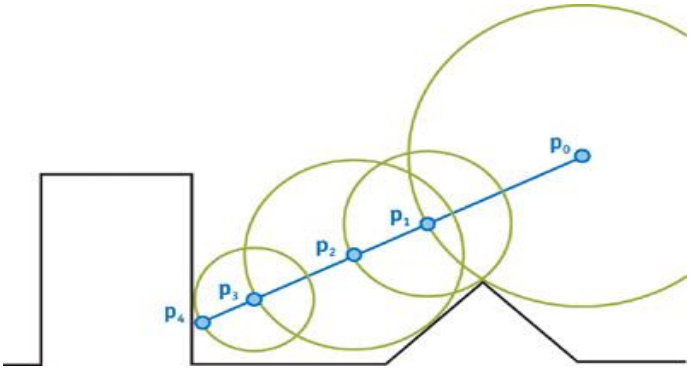


Figure 8: Overview of sphere tracing raymarching (Pharr, 2006)

SDF examples

The functions and images used within the section are adapted from Inigo Quilez (2008) site explaining signed distance functions. Some examples of the SDFs are illustrated by the mathematical function to represent each alongside a translation to HLSL. Let F(x,y,z) represent these functions.

Sphere

A sphere can be represented as an SDF. Where a sphere with the center origin at (x0, y0, z0) and a radius defined as 1. To determine if any 3D point lies on a sphere and satisfies the equation:

$$f(x, y, z) = \sqrt{x^2 + y^2 + z^2 - 1}$$

- f(x,y,z)<0, the point lies in the sphere;
- f(x,y,z)>0, the point lies outside the sphere;
- f(x,y,z)=0, the point lies surface of the sphere.

As a result f(x,y,z) is the distance between the point and the sphere surface along with its sign informing if the point occurs on the inside/outside/on the sphere surface. This function informs why it is known Signed Distance Function.

In the equation for the sphere, where p represents the center and s the radius:

$$f(x, y, z) = \|\vec{p}\| - s$$

```
float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```

Figure 9: HLSL sphere SDF

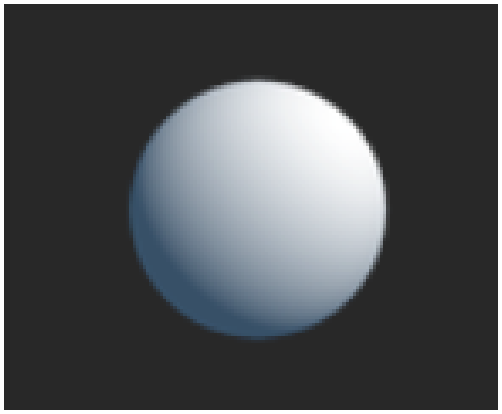


Figure 10: Sphere SDF

Box

In the equation of a box where the center is represented by C with b representing from the center to the first-quadrant corner:

$$Let\ \vec{q} = \|\vec{p}\| - \vec{b}$$
$$f(x, y, z) = min(max(q), 0) + \|max(q, (0, 0, 0)\|$$

```
float sdBox( vec3 p, vec3 b )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

Figure 11: HLSL box SDF

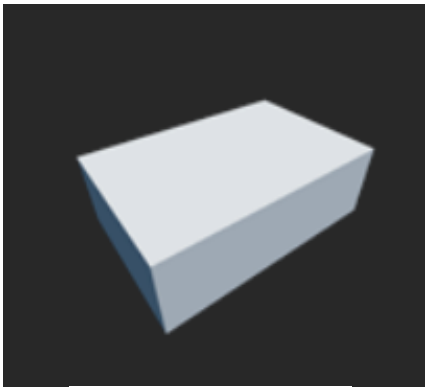


Figure 12: Box SDF

Constructive Solid Geometry

Along with the ability to create these implicit objects, there are methods known as Constructive Solid Geometry (CSG) which are Boolean operations for more complex geometric shapes with the ability to combine two surfaces. For CSG the fundamental idea is built on 3 primitive operations: intersection (∩), union (∪), and difference (−). The figures below show a pure application of these. But with these basic operations there are problems associated with the outcome of the operations as there are discontinuities in its derivatives as such the resultant surface of the unified object is not a smooth surface. To solve this problem there are further operations that enable the discontinuity of each to be removed.

Each of the parameters passed into each function represents a surface that will be utilized for the function application.

Union

The Union operation functions by using the closest surface which means it is equivalent to a set sum but not a distance sum. The Union equation can be represented as:

$$(f \cup g) = min(f(x,y,z), g(x,y,z))$$

```
float opUnion (float d1, float d2) {return min (d1,d2);}
```

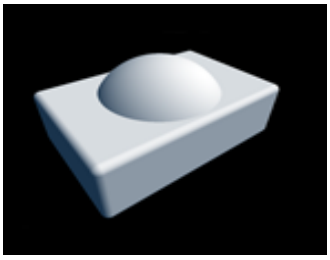


Figure 13: Union SDF

Intersection

The intersection function uses the further surface of the 2 passed in.

$$(f \cap g) = min(f(x,y,z), g(x,y,z))$$

```
float intersection (float d1, float d2) {return max(d1,d2);}
```

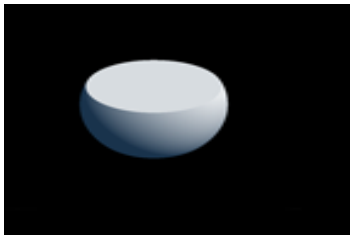


Figure 14: Intersection SDF

Subtraction

A function is inverted and used to intersect another. The subtraction equation can be presented as:

$$(f - g) = max(f(x,y,z), g(x,y,z))$$

```
float opSubtraction (float d1, float d2) {return max (-d1,d2);}
```

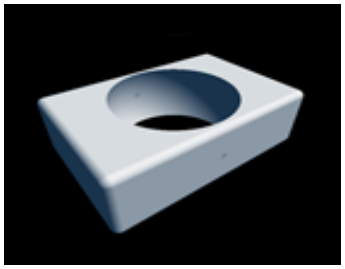


Figure 15: Subtraction SDF

Smooth Blending

The problem of the discontinuities can be solved through further operations which enable the discontinuity of each to be removed. One such example is discussed by Hoffman and Hopcroft (1985) in which they derived away to preserve the conservative property of the SDF, where shapes can be blended to create a smooth union in a sense. There are numerous functions for this union but the one derived by Hoffman is a union using “smooth min” (smin) function instead of a straight min function. With the various smooth min function, each offer tradeoffs in terms of quality and performance. A fast polynomial version recommended by Quilez is:

```
float smin(float a, float b, float blendRadius) {
    float c = saturate(0.5 + (b - a) * (0.5 / blendRadius));
    return lerp(b, a, c) - blendRadius * c * (1.0 - c);
}
```

Where a and b represent the 2 SDFs and blendRadius represents how smooth they should be blended.

Figure 16: Smooth union SDF

Increasing Performance

For rendering one concern is performance, a ray marching is performance-critical, some considerations to increase it can be done. At the cost of image quality, the performance of a ray marcher can be improved by increasing the minimum step size and accepted distance from the surface while also decreasing the maximum iterations allowed with resolution. But it can also be increased by improvements in the algorithm which won't affect the image quality but come at the cost of the simplicity of the naive method. Alongside there being more improvements, which can be considered such as over relaxation, or bounding sphere.

PROCEDURAL GENERATION

In terms of raytracing and raymarching, these techniques offer the ability to create procedural objects with photorealistic characteristics but another way in which procedurally generated objects can be created is using algorithms to generate large amounts of content, not just implicitly, for use in various applications such as graphics and video games. The approach is usually done through a mix of pre-made assets alongside algorithms, with the use of processing power and computer-generated randomness enabling procedurally generated content to be produced. It offers a huge advantage in being able to produce a wide variety of content and randomness in gameplay with little to no extra cost in processing power. As the concept is only limited by the ability of the algorithm creator, there is a huge amount of possibility to create huge amounts of complex or simple content to be deployed within a game while creating a sense of uniqueness and no repetition in gameplay compared to games which don't employ this approach such as Call of Duty.

One example where procedural generation is a focal point to creating the gameplay and content is No Man's Sky and No Man's Sky VR. Within the game, there are around 18 quintillion (Wilson,2015) unique planets and moons, generated algorithmically using computers. With this as well, there is graphically generated content, over a network system which allows for users to see the same effects and graphics effects as well as smooth networked interaction.

History of Procedurally Generated Content (PGC)

As it is an algorithmic approach the history of procedural generation can be traced back to the 1600s within the mathematical community and the idea of procedural structures can be dated back to even before humankind. (Korn and Lee, 2017) Within the natural structures of the environment as well as elemental there is to a degree these structures are generated in universal iterations of simplistic rules. Some examples are Romanesco which has a sort of procedural graphical effect. (Korn and Lee, 2017).

Figure 17: Romanesco Broccoli showing a simplistic and repetitive pattern

As for this possibility of PCG, there is a deep history of PCG being employed in games as early as 1952, an example is a project by Alan Turing and was an automated love-letter generator and is identified as the first known work of new media art (Aversa,2015). It was implemented onto the Manchester Mark 1 computer by Christopher Strachey. Further development in the field came around 1980 with the creation of Akalabeth by Richard Garriott's which is seen as one of the first games to utilize a seed to generate a game world and supplementing the need to store premade assets or data in memory not available on computers at the time. And gives an idea of why PCG is widely used as it allows for the compression of data. (Aversa,2015) One such game which employed PCG to a high ability was The Sentinel which with only 48 and 64 kilobytes to use managed to store around 10,000 different levels.

Elite

Another game that was able to compress and create lots of assets was Elite (Kionary, 2019) which used PGC to generate a universe. The universe is comprised of around 8 galaxies with each consisting of 256 solar systems, along with each solar system containing around 1 to 12 planets. A space station orbited each planet alongside each planet having a procedural name, personal terrain, and local details. While created for a 32KB computer. The seed of the universe was hardcoded with it being 4069 on release. To illustrate why how much space the content would take up, the newer version of Elite is considered. It utilized PCG to generate a 1:1 replica of the Milky way with more than a 400-billion-star system which in terms of space would use more than 400 Terabytes of space without the use of PCG. (Aversa, 2015)

In the past, there were 3 main reasons for use of PGC. The use of a PCG system allows for the automatization in designing and producing assets such as trees, rocks alongside the general environment. It can also help facilitate an increase in the amount of game content which isn't possible manually and randomness to improve games replay ability.

Modern PGC

Dwarfs Fortress

A game that took the idea of PCG to another level is Dwarfs Fortress which had a sophisticated fantasy world generator that considered a wide variety of aspects from weather to biomes to the geological distribution of materials, along with much more. It also built an extensive history for the various populations in the game as well as being influenced by races and the rise and fall of cities. Other parts of PCG it employed were poetry, monsters, animals, events, cities, and even more. This game inspired one

of the most popular modern games of Minecraft which utilizes PCG to the full extent and is seen as the basis of modern intrigue into PCG.

Types of procedural algorithms

In terms of PCG algorithms, there are 2 types: Teleological and Ontogenetic.

Teleological

Teleological refers to trying to simulate an accurate physical process of an environment such the result is a desired procedural output. In turn, the content will be generated such as in nature. Some example algorithms are Fluid Dynamics, Fractals, and Cellular Automata.

Cellular automata is one such example, coined by John Von Neumann in the 1950s, in which a grid of cells, where each have a state and with a set of rules which through influences by its neighbors and itself to help determine its state. Using the rules in several iterations, each cell can be adapted to take after their neighbors. Another example of a Teleological algorithm is the raindrop algorithm which is a way to simulate erosion. The algorithm works through the manipulation of a height field based on simulating raindrops falling onto the height field and moving lower points. The initial raindrops are used to remove height at the spawning, with the rain drops then depositing additional height to the lower points. (Doull, 2011)

Ontogenetic

Ontogenetic Algorithms refers to an approach in which the results of the algorithm are observed and then the algorithm attempts to replicate the results through ad-hoc algorithms. One of the earliest forms of PCG Ontogenetic algorithm is regarding Mazes as there are several decades of research within mathematics. Some example algorithms are Perlin noise, Random Traversal, Dijkstra's, and Midpoint Displacement.

An L-System is one such example and can be described as a parallel rewriting system and a type of formal grammar. (Aversa,2015) Botanist Aristid Lindenmayer was first to describe it as the way in how plant cells act alongside modeling plant development by growth processes.

A common technique used for the creation of procedural dungeons is the Voronoi-Delaunay Triangulation illustrated below.

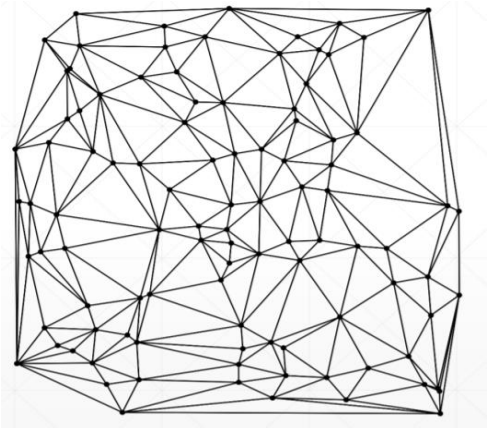


Figure 18: Representation of a Voronoi-Delaunay Triangulation (Aversa, 2015)

7 REQUIREMENTS

To enable development to start, several requirements and the approaches for these had to be chosen. Each requirement outlined, there were several options to choose from such that the best-suited option for each requirement matched the project's needs.

The first requirement choice was the VR headset, as the chosen game's development software had to be adjusted to enable VR development. As well as enabling any additional external software to be downloaded and integrated to support the headset. The use of a game's development software was determined, with the language used for development also being determined by this choice. The options for the networking API used would also be determined by the chosen game's development software. Regarding the options for procedural generation, numerous avenues could be explored, but to simplify the choice, a maze and some simple objects were chosen to be generated. Which would also be supplemented with the use of a ray marcher or ray tracer. The approaches for each would be based on existing algorithms suited for procedural generation.

VR HEADSET

The VR headset chosen for the project was the Oculus Quest. As discussed before, there was a wide variety of choices regarding the headsets available to use. With each headset, they came with both positives and negatives but the overall deciding factor in why the Oculus Quest was chosen was the portability and the ability to be used alone or in conjunction with a computer whereas most other headsets required a high-end computer to be able to be used and function. A further point to be made about the Quest compared to the other headsets available is the external cameras which allow for the Quest to be used anywhere and not constrained for use within a room with a computer. The specs offered are very good but compared to other headsets they aren't as good, as headsets that require to be connected to a computer also utilize the computer's hardware. This enables these headsets to have better-quality visual and video aspects. Ease of use and setup was the main factor in the headset chosen compared to graphics and performance.

The tracking system of the Quest functions through the freedom of movement of a rigid body in three-dimensional space. This is known as 6DOF or 6 degrees of freedom. The process of the system works through tracking the body position changes alongside translation on 3 perpendicular axes as well as tracking changes regarding rotation on 3 perpendicular axes. As such, 6DOF offers the ability to track surge, heave, sway as well as sway, pitch, and roll, represented in figure 19.

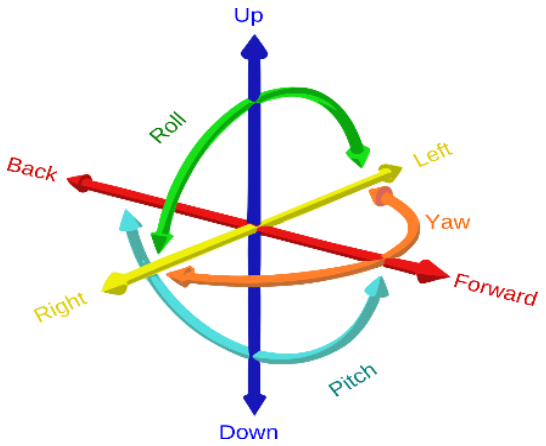


Figure 19- Representation of 6DOF (Wikimedia Commons, 2020)

The 6DOF system is an algorithm also known as SLAM (Simultaneous Location and Mapping) or Inside-Out. The process in which SLAM algorithms work is through noticing unique static features in a room and through comparison of the rotation and acceleration from the accelerometer & gyroscope of the headset against how these static features appear to move, which enables the headset to determine its position. The process is also known as dead reckoning as it tracks each moment to moment alongside tracking headset and controller this way. The HMD tracks each controller, through its cameras and intercepts any visible and/or infrared light coming from the LEDs contained within the controllers. (Lang, 2013)

OCULUS API

As the Quest is an Oculus product, they have created a package to support the development of VR within Unity. There is a variety of useful utilities offered in the package as well as offering easy integration alongside easy development of a VR application within Unity. A further point to be made about the package is it offers pre-developed scripts which add more functionality to Unity on top of enhancing a variety of Unity component scripts. The package also provides prefabs which provide the full VR support for Unity along with scripts that integrate the VR framework and Unity framework. The main parts of the package which provide the most useability of VR within Unity is the OVRplayer, OVRCameraRig, OVRGrabbable, and grabber. Each provides a different aspect to fully allow interaction within the VR environment.

An essential component offered within the package, which acts as the main interface between the user and VR is a component called OVRManager. This component is a singleton which the Oculus SDK is exposed to by Unity, and in turn, offers the functionality to provide the Oculus variables to establish the camera behavior. The component also provides access to the Head Mounted Display (HMD) state of the VR. Alongside providing access for the component OVRDisplay in terms of the pose and rendering of the HMD. The component OVRTrackers is also provided the ability for tracking each of the controllers pose, frustum along with the tracking status of the infrared tracking sensors contained in each controller,

GAMES DEVELOPMENT SOFTWARE

For the choice of games development software, it was either through a game's development software or a programmatic approach using OpenGL with a game's engine needing to be developed and integrating the VR API into this engine. As the focus of the project was not on game engine creation, the choice was to use a games development software with a programmatic approach for the various aspects so more time could be focused on the project's goals rather than set up. Regarding OpenGL, it is already integrated and utilized by several games' development software's so even with both paths offering positives and negatives, the games software approach offered the ability to utilize both OpenGL while not having to focus on the creation of an engine to support development. With this availability a more straightforward approach in development could be undertaken as the requirements for each objective are fulfilled.

With the choice of software to choose from, Unity and Unreal Engine are the 2 most standout and popular software available on the market. Each platform offers similar opportunities for the creation of games as well as offering packages to support VR game creation. The only difference is the scripting language used by each software and which is wanted for development. As such, C# was chosen as the language which ultimately decided the development platform used.

Unity

Unity is a games development platform that provides game developers with all the necessary tools and features to support games to be developed quickly, efficiently, and enabling easy creation of interesting game mechanics. As it is also classified as a game engine and is built on a foundation of a collection of various libraries. These libraries are linked to support for games to work and are accessed through the unified Unity API. For code development, Unity offers a framework that is derived from Monobehaviour which simplifies the game's process as the aspects needed for the specific project is only used. C# is utilized by the primary scripting API of Unity alongside using plugins and the Unity editor functionality: drag and drop. Within the software, it offers the basis of a games engine and supports the ability to extend this with custom scripts to allow for continuous adaptation and expansion.

Mesh and Shaders

For the implementation of the procedurally generated meshes, Unity offers the ability to use its custom mesh implementation, through specifying arrays made up of triangles and setting them to the mesh of a game object, to enable game object to be created.

Unity offers the ability to create shaders to enable programs to run on the GPU. With use of shader in Unity, it enables instances of a shader to be defined as a Shader object. With said object specific way in how Unity works with shader programs as it functions as a wrapper for shader programs alongside enabling multiple shader programs to be defined within the same file while informing Unity how it should use them. (Unity- Manual: The Shader Class, 2020)

C#

C# as stated is used by Unity and is described as an Object-Oriented language so fully supports OOP concepts: inheritance, encapsulation, abstraction, and polymorphism. It also allows for easy grouping of

objects of a similar caliber along with allowing the objects to be self-sustainable as opposed to more action and logic-driven languages.

NETWORKING

In the discussion of networking, an important aspect is the transmission protocol for transmission of the data packet over the network. The 2 best protocols suited and used for games are UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). Even though both are based on IP, which provides the ability to send a packet from a source to a destination, there is no guarantee a packet would arrive at the destination, nor it is the only data packet, or the packet isn't corrupted, or the sequence of the packets is received in order. So, with these limitations, in both protocols, some considerations need to be made to determine which is best suited for this project along with what packages to support the chosen protocol.

Protocols

TCP offers reliability, preservation of packet order, error detection which while not offered on UDP, the design of TCP causes higher latency. TCP provides a more reliable, ordered, and error-checked connection for both hosts, and is used in conjunction with other protocols like HTTP or FTP but it comes with a downside of latency. In TCP, latency is caused when a packet is sent from the host, the receiver expects an acknowledgment (ACK) but if none is received after some time (various reasons can affect this like a lost packet or no acknowledgment.), the packet is sent again. Moreover, TCP provides a guarantee that packets sent after cannot be processed until the first one has been received and processed. Because of this issue, many games use UDP in conjunction with a custom protocol to remove the latency as important aspects in-game rely on low or non-latency such as FPS. As such UDP with a custom protocol can be made to be more efficient compared to TCP. With this custom protocol, an example of a more efficient way is through the ability to manage various streams of data which ensures each stream will continue running even if there is a lost packet on one.

Networking APIs

As for the wants of the project, as the focus isn’t on the development of a custom networking system, the use of an existing library that offers each of the features discussed above, while also offering the ability to implement a custom protocol if needed. In the discussion of the library, a dependent factor is the type of architecture wanted and which is best suited for this type of game. A discussion of the various preexisting libraries and the features they offer, as well as what architecture they offer.

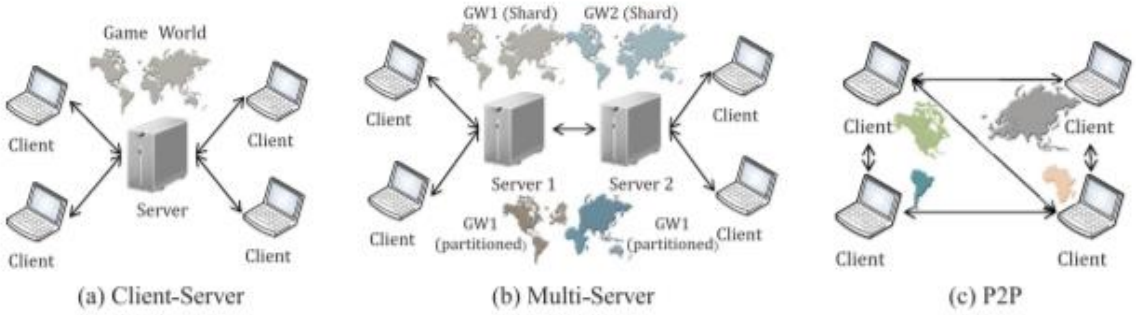


Figure 20: Different gaming architectures: (a) In a client-server architecture the server is responsible for maintaining the whole game world; (b) in a multi-server architecture either: (1) servers maintain separate game worlds (shards) or (2) the game world is divided into different zones maintained by different servers; (c) in a peer-to-peer architecture each peer maintains a part of the game world. (Yahyavi and Kemme, 2013)

Mirror

Mirror offers a similar approach to old Unity Networking UNET. Offers easy concepts to grasp that make development easy. Mirror offers the ability to switch between different Transport layers but offers a less built-in functionality which requires extra assets to enable use. Mirror offers a connection model of Client/Server, with offline LAN support and built-in functionality for the multicast discovery of LAN games and offline mode. It doesn't give the ability for built-in matchmaking or NAT punch through, requires 3rd party or self-development for relay support, and only offers a listen server but not full Peer to Peer connection. Supports cross-platform Desktop, Mobile, Web, Console. (General - Mirror, 2022)

Photon

Another Networking solution offered in Unity is Photon. It is described as a powerful yet easy-to-use networking solution and used mostly for room-based (short and small amount of player and playtime) multiplayer experiences. It offers a variety of features but one of its main draws is the ease of use and guaranteed connectivity between players as it deploys a semi-authoritative solution with the use of relay servers to keep the connection between players secure and connected. Players connect via an authoritative player known as host but connect via the relay server and not the host per se. For the connection options, it offers a connection model of Client to Client so is Client hosted. It offers a single-player offline mode, built-in matchmaking, and a built-in relay server, and supports cross-platform like Mirror. But it doesn't offer Offline LAN support, does not require a NAT punch through, and offers no support for Peer-to-Peer connection. (Photon Unity 3D Networking Framework, 2020)

Photon 2

With regards to Photon further adaptation has come about with a new Networking system built on the Photon relay and simple host migration implementation. Such it offers everything Photon does with additional functionality. Its connection works the same as Photon as it uses an authoritative player hosted by a client to client via a cloud server. It ensures 99% connectivity with no NAT problems. It offers the ability to connect to players directly so with matchmaking supported players can find other games by other players. LAN play is unavailable, so an internet connection is always needed for use of PUN2, but an offline mode is offered with the code written around the PUN implementation of a single-

player experience. A con though is the difficulty in a cheat prevention implementation. (Photon Unity 3D Networking Framework, 2020)

MLAPI

A newer networking solution being supported and adapted for Unity is MLAPI. It is the new official Unity solution, with it being the main networking solution pushed and will be supported fully by Unity going forward. It offers an authoritative player host with a client-to-client solution with a dedicated server offered. This allows for ease of avoidance in client hacking. But to enable the dedicated server to work the use of an external download is needed, but LAN support is not available. It offers the ability for access to different transport layers along with custom transport. Other features offered are the ability for offline mode as well as full support for synchronization with the use of the Unity features. One feature missing is matchmaking which seems to be in development but is not yet offered. Cross-platform is also supported. (MLAPI- Unity Multiplayer Networking, 2021)

With a comparison of the best available APIs offered for use in Unity, the decision of the networking API was the MLAPI. It offers a free to use model, with also offering the best suited architecture of a listen server for an FPS action 3D game.

Chosen Solution

MLAPI is an acronym and stands for “Mid-Level Application Programming Interface” and as stated in the name is a Mid-level API. As it is a mid-level solution, it offers a middle-level approach, so it provides access to the inner processes of a lower-level API while also giving the feeling of a high-level API. As such it has access to the central networked processes while maintaining a certain level of abstraction. (Bonzon,2020) The MLAPI framework aims to be Unity’s first official first-party Game Objects Netcode solution, with continuous development still occurring. (House,2020) A more in-depth discussion of what MLAPI offers can be found in appendix B.

PROCEDURAL GENERATION ALGORITHMS

In the discussion of the requirements, a discussion on the algorithms considered for the generation of a procedural map. This sub section will discuss the various algorithms considered and help determine the best suited for the type of project.

Cellular Automata

One such algorithm is Cellular Automata. It utilizes a grid which is the representation of the level. Within the approach, there are various steps to the generation, with each step causing a smoothing effect which creates a more organized level as each cell takes more after their neighbors. At the start, the grid is usually randomly filled with a variety of tiles, but for the project, only walls and non-wall tiles are considered. To adapt the cells, there are established rules, which inform each cell of its next manner based around its 8 neighbors. With the use of Time, each step is called which causes the grid to adapt per the rules, which with a repetition of each step, a grid adapts to resemble a level. (Shaker, Togelius, and Nelson, p.38, 2016) An example of this is illustrated in the figure 21.

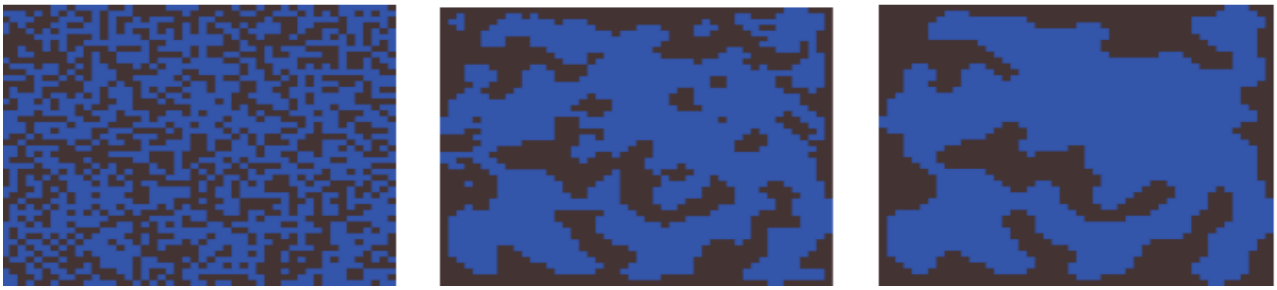


Figure 21: Cellular automata Map generation (Cook,2013)

Agent-based map creation

Within this approach, the idea is with the use of one or multiple agents which are given a variety of tasks that when undertaken produce a level of sorts. A general implementation of this is through an agent, that traverses a map, generating a trail behind it, with each step taken the agent will do one of two actions: place a room, or change direction, creating a chaotically generated level. (p.38, Shaker, Togelius, and Nelson, 2016) Figure 22 represents each action. Even though a map is created, it usually generates chaotic levels as there is such randomness in the paths along with the possibility of an overlap of the rooms. If these problems can be removed, the generated levels will feel organic and maybe even planned. But even this won't always cause the agent to work as intended.

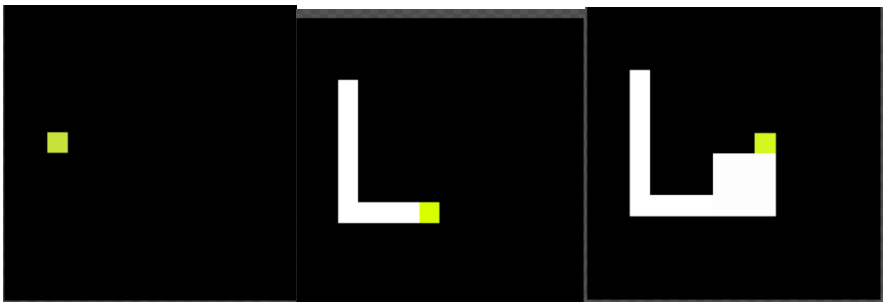


Figure 22: The yellow square represents the agent, is initially randomly placed within a level of walls(left), the agent digs through the level(middle) while it places rooms(right).

Cyclic PCG

Within a Cyclic approach, it is based around the idea to reduce the amount a player needs to backtrack through a level. Backtracking refers to if a dead end is reached, a retrace of steps is undertaken to find an area that doesn't have a dead end. To generate the main path in a cyclic approach, the use of cycles to avoid any dead-end generation unlike a tree in a graphing approach. (Dormans, 2015) Unexplored is one such game that uses cyclic PCG.

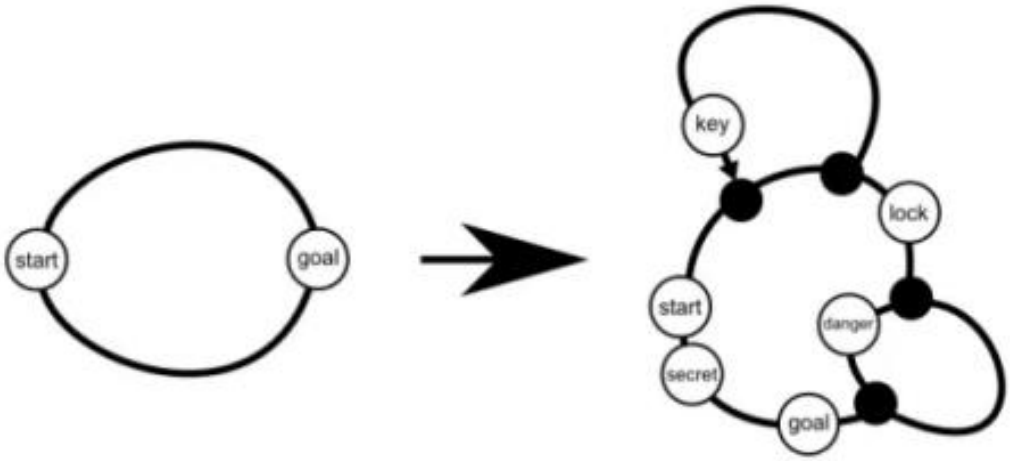


Figure 23: A cyclic level approach (Dormans,2015)

Search based- PCG

Another algorithm which could be used is a Search-based PCG which is based on the Darwin theory of evolution as such can be classified as an evolutionary algorithm. For this an evaluation of various algorithms is done, to determine the better-quality aspects of each, with only the better-evaluated parts kept. With the remaining content, it is used to replace the unwanted parts with a random modification. Further iterations are done, until either the remaining content is deemed to be of high value, or the maximum number of iterations is reached. A fitness function or an AI can be used to determine the best content by allocating scores when playing. (Valtchanov and Brown, 2021) (p.18, Shaker, Togelius and Nelson, 2016)

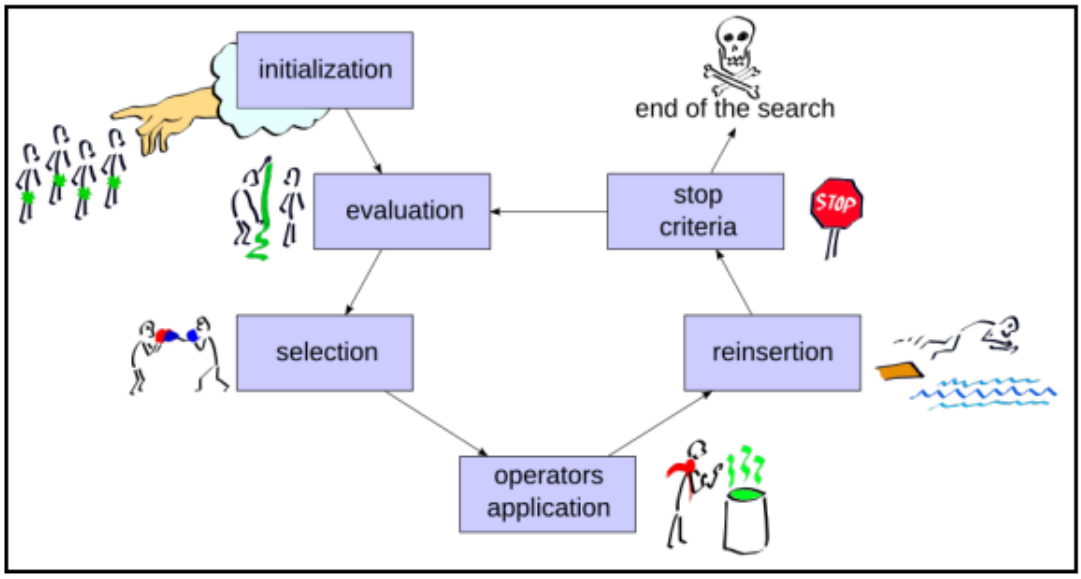


Figure 24: Evolutionary algorithm shown as general schema (Maturana, 2009)

Space partitioning

A final approach that may be undertaken is a Space partitioning generation. The idea of this, is through a generation of a basic layout of rooms, with several iterations done to recursively divide each area into smaller and smaller disconnected areas along a random axis, which once the rooms are small enough, doors and corridors are used to connect them. The main implementation of this approach is known as Binary Space Partitioning, where an area is continuously divided into 2 new smaller areas. (p33, Shaker, Togelius and Nelson, 2016) Through this method, there is a generation of a highly structured level along with connections between each room. An example of this generated level is illustrated in the figure 25.

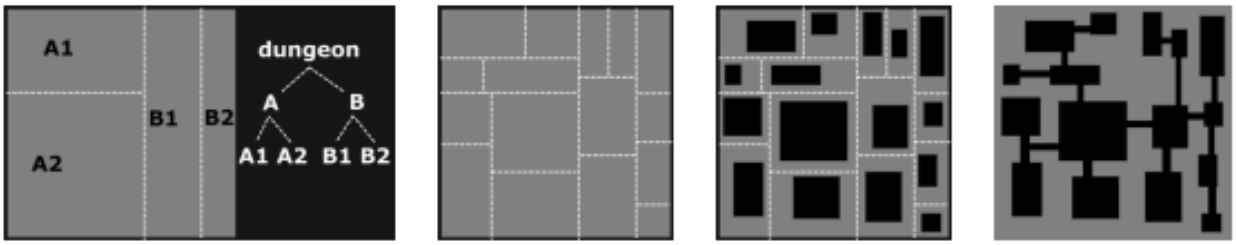


Figure 25: Dungeon generation with space partitioning. (Basic BSP Dungeon generation - RogueBasin, 2020)

MESH GENERATION ALGORITHMS

Regarding representing the created 2D maps into a 3D mesh, several algorithms can be used to extrude triangles from 2-dimensional into a 3-dimensional mesh. 2 algorithms best suited for this are Marching Squares and Delaunay Triangles.

Marching squares

The Marching Square algorithm is defined as a classical iso-value curve extraction method of a two-dimensional scalar field, also represents a special case of the Marching Cube algorithm in 2D. An overview of the algorithm can be described as where scalar fields are dispersed into quadrilateral mesh data, with each value at a vertex on each grid cell a comparison occurs between the pre-defined iso-value α, with each one greater than or equal to α, it is marked with "+", while others will be marked with "–". (Xu, 2015)

In the outcome that two vertices have different marked symbols, a further step is done where interpolation is computed to counter the isoline through the edge, and by using this edge, the intersection of the iso-value curve and the edge is obtained. (Vilchez, 2020) By connecting the edge intersections of each edge in order, the iso-value curve passing through the mesh grid is found. Where each vertex of the grid has an attribute value. Each square has a pre-defined iso-value assumed to be 4. The counter isoline can be extracted with value 4 by using linear interpolation. When the Marching square algorithm has been performed on each cell, the iso-value curve is found, with a mesh generation possible as each polygon in each square can be extruded by using triangles. An example is illustrated in figure 26 within a two-dimensional quad mesh.

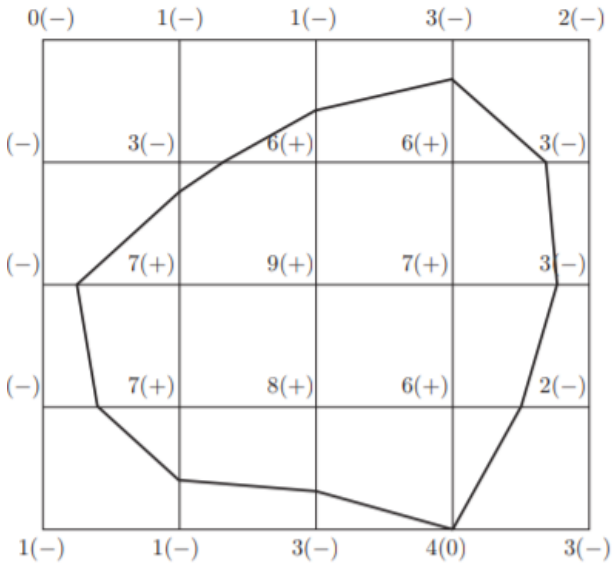


Figure 26: Marching squares algorithm on a 2-dimensional quad mesh (Xu,2015)

Delaunay Triangulation

One further algorithm that can be used for generating geometric data structures for meshing is Delaunay Triangulations (DT) which is based on the Voronoi diagram but through a principle of duality. The description of DT is described based off David Mounts' (2020) explanation.

He describes it as "The Voronoi diagram is made of a set of sites in the plane so is a planar subdivision, also noted as a cell complex. This can be defined as the dual of such subdivision is a cell complex. Where each face of the Voronoi, a vertex, corresponds to the site. Each edge of the Voronoi diagram which lies between 2 sites, within the dual connecting vertices of these 2 sites, seen in figure 27b with Pi and Pj, an edge is created. Each vertex relates to a face of the dual complex.

Under an assumption of general position where no 4 sites are collinear, each vertex of the Voronoi all has degrees three. As such it follows where each face of the resultant dual complex is triangles but with an exemption of exterior faces. A triangulation of the sites is the result of this dual graph, and this approach is known as Delaunay triangulation". The use of triangle extraction method can then be used to extrude triangles and create a 3D mesh.

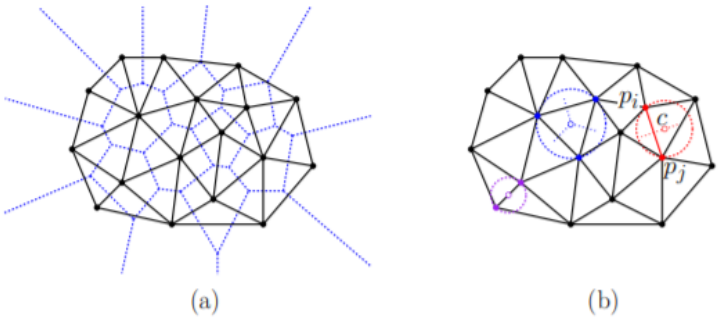


Figure 27: (a) The Voronoi diagram of a set of sits (broken lines) and the corresponding Delaunay triangulation (solid lines) and (b) circle-related properties. (Mount,2020)

With these algorithms, they are more general-purpose algorithms for the generation of a variety of shapes with distinct properties offered by each, but a more general discussion can be made regarding the algorithmic approach for procedural simplistic shapes. A discussion can be made regarding algorithms that can be used for generating one of the more basic geometric shapes, the sphere. For use in the project, the spheres will be used as the bullets fired from a tank as a sphere is perfect to represent a bullet.

PROCEDURAL SPHERE MESH GENERATION

There are numerous methods for generating a sphere, but 3 approaches will be discussed, these are UV sphere, Spherefied cube, and Icosahedron.

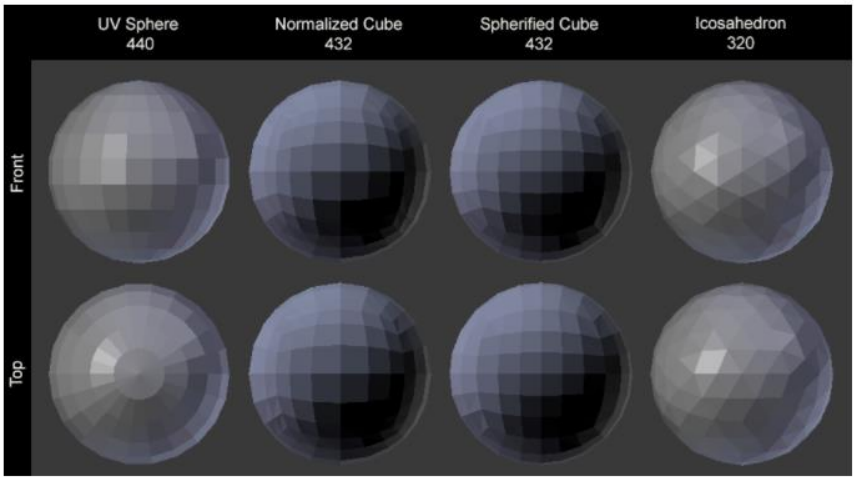


Figure 28: Front and top views of the four methods. Each is made of around 300 - 450 triangles. (Cajarvaville, 2016)

UV Sphere

The first implementation is the UV sphere, found within most 3D tool kits. For this sphere, it can be defined as a 3D closed surface in which the same distance(radius) from the origin for each point of the sphere is the same. This can be represented in the formula:

$$x^2 + y^2 + z^2 = r^2$$

The process of this method is where a limited number of points on the sphere are sampled as not all points can be drawn with the sphere being divided by meridians, stacks, or longitude (lines from pole to pole) alongside sectors or parallels (lines parallel to the equator). Each sampled point then connects to form the surfaces of a sphere alongside an outcome of how each point acts as near the center of the sphere each face is bigger whereas closer to the top there are smaller quads.

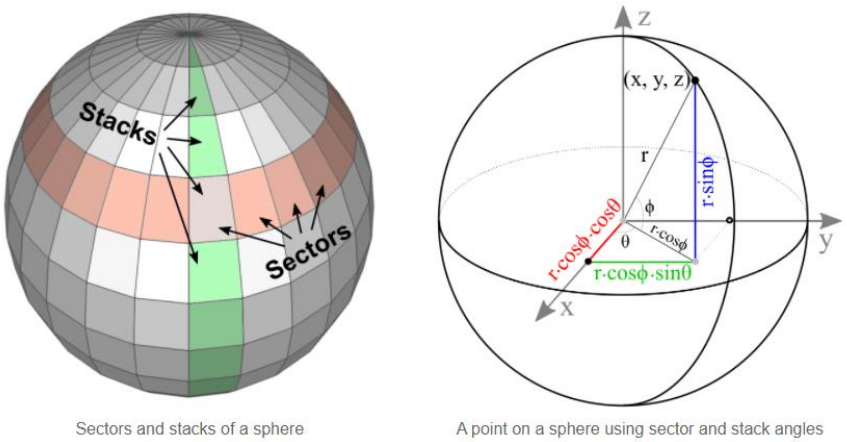


Figure 29: Finding a point on a sphere through using sector and stack angles. (Ahn,2021)

To further illustrate this point, an example can be given in terms of a parametric equation for the sector angle ϑ and stack angle ϕ in terms of corresponding to an arbitrary point f(x,y,z) on a sphere(Ahn,2022):

$$\begin{aligned} x &= (r \cdot \cos\phi) \cdot \cos\theta \\ y &= (r \cdot \cos\phi) \cdot \sin\theta \\ z &= r \cdot \sin\phi \end{aligned}$$

0 to 360 degrees is the range used for the sector angles, while the stack angles from the top to bottom each use 90 degrees, but the bottom is -90 degrees. The figure represents the equation of each step for both the sector and stack angles:

$$\begin{aligned} \theta &= 2\pi \cdot \frac{\text{sectorStep}}{\text{sectorCount}} \\ \phi &= \frac{\pi}{2} - \pi \cdot \frac{\text{stackStep}}{\text{stackCount}} \end{aligned}$$

Figure 30: Equation to calculate each sector and stack angle for each step (Ahn,2021)

Spherefied cube

A cube sphere method can be discussed as subdividing a cube by N several times to supplement the creation of a sphere. The special property of a cubed sphere is that the spherical surface of the sphere is broken down into 6 equal-area regions (+X, -X, +Y, -Y, +Z, and -Z faces).

Figure 31 illustrates how one of the 6 regions of a cubed sphere can be constructed. It is done by intersecting angularly equal-distant longitudinal and latitudinal lines from -45 degrees to 45 degrees. With the use of an intersection of the 2-plane equation, each vertex on a cubed sphere can be found. $\vec{n}_1$ represents the normal vector of the latitudinal plane alongside $\vec{n}_2$ representing the normal of the longitudinal, with the direction vector of the intersect line represented as $\vec{v} = \vec{n}_1 \times \vec{n}_2$. Each

vertex on the cubed sphere is then scaled through the normalized direction vector multiplied by the sphere's radius, $r\vec{v}$. (Ahn,2021).

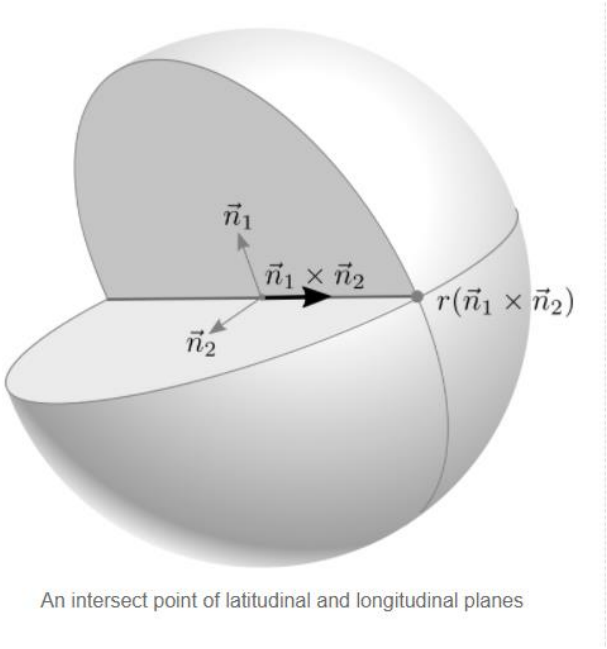


Figure 31: Illustration of how one of the 6 regions can be constructed (Ahn,2021).

$$+\text{X-axis vertices} = \begin{cases} \vec{n}_1 & = (-\sin\theta,\ \cos\theta,\ 0), \quad -\frac{\pi}{4} \leq \theta \leq \frac{\pi}{4} \\ \vec{n}_2 & = (\sin\phi,\ 0,\ \cos\phi), \quad -\frac{\pi}{4} \leq \phi \leq \frac{\pi}{4} \\ \vec{p} & = r(\vec{n}_1 \times \vec{n}_2) \end{cases}$$

Figure 32: Equation to calculate the positive X-axis vertices (Ahn, 2021).

The other 5 faces can be generated through this process or simply swapping and negating the vertices of the +X face. It also helps remove redundant calculation sine/cosine to enable optimization. There is some obvious deformation when points are closer to the cubes original corner with this method though.

Icosahedron

The most complex approach is the icosahedron which is defined as a polyhedron and is made up of 20 identical equilateral triangles, each triangle has unique properties and has the same area size, and each vertex is the same distance from its neighbors. The higher number of triangles are generated by a subdivision of each triangle so 4 triangles are created through the creation of a new normalized vertex in the midpoint of each edge. With the normalization, each vertex will lie on the sphere surface. This ruins the initial icosahedron properties as all the triangles aren't equilateral, and each area and distances between the vertices aren't the same across the mesh now. Using spherical coordinates, the 12 vertices of an icosahedron can be constructed, where 2 vertices each aligned on the north and south poles alongside the other 10 placed at latitude $\pm \tan^{-1}(\frac{1}{2})$ degrees and 72 degrees aside on the same latitude. (Ahn,2021).

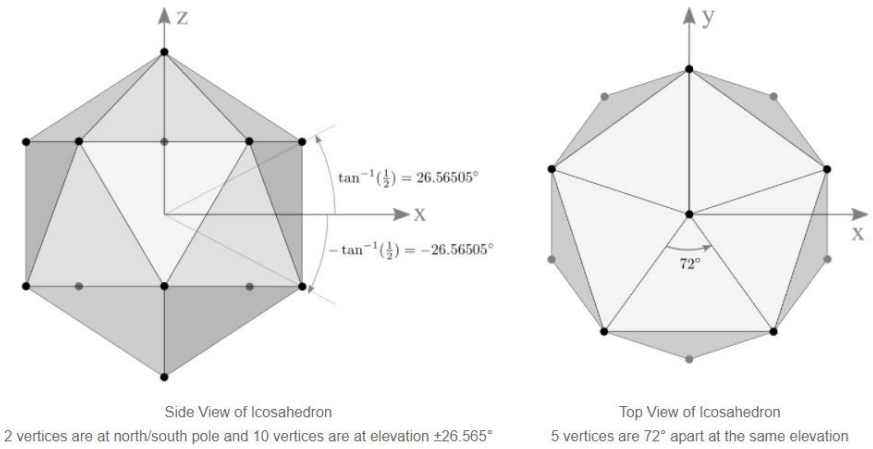


Figure 33: Orthogonal projection of a icosahedron (Ahn, 2021):

$$
\begin{aligned}
x &= r \cdot \cos\left(\tan^{-1}\left(\tfrac{1}{2}\right)\right) \cdot \cos(72 * n) \\
y &= r \cdot \cos\left(\tan^{-1}\left(\tfrac{1}{2}\right)\right) \cdot \sin(72 * n) \\
z &= r \cdot \sin\left(\tan^{-1}\left(\tfrac{1}{2}\right)\right)
\end{aligned}
$$

Figure 34: Equation to compute the radius using a latitude of 25.565 for a typical point (Ahn, 2021).

FUNCTIONAL REQUIREMENTS

For porting to the Oculus Quest, further software had to be installed. The software is known as ADB (Android debug bridge) and is an android development software, which once installed is integrated into Unity and enables Unity to port to the Quest as it runs on the Android OS (operating system). The ADB is a command-line tool that provides the functionality for communication between a computer and an android device. The ADB software is versatile and supports various actions such as debugging and installing apps. 3 components, a client, a daemon, and a server are used to create the functionality of the ADB. The process works, by an invocation of a command-line terminal that issues a common ADB from the client, the development machine. Each device involved runs the daemon, in a background process from the connected device. The server provides and manages the connection between client and daemon. To illustrate this, an example of the process, where a client checks for a running of an ADB server process which determines if it needs to start a server process. From this, a local TCP port of 5037 is bound, with it waiting to receive information or commands from the client. Connections are then created by the server for each connected device, which then locates android emulators on each device. A connection to the port is then made if an ADB daemon is found.

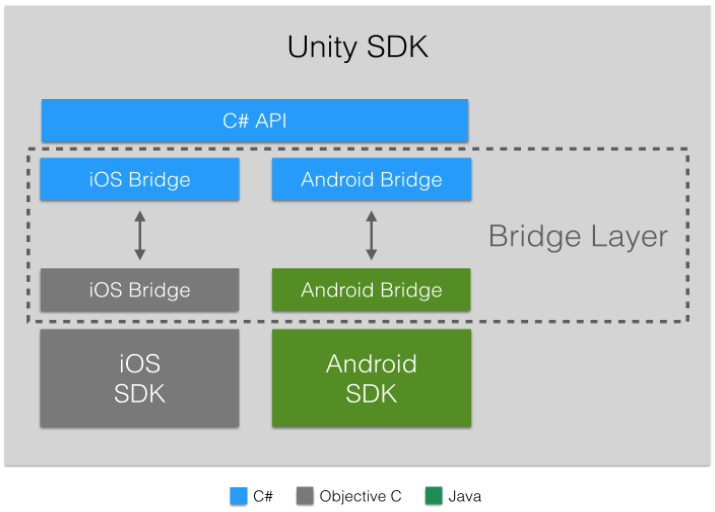


Figure 35: The process of how an ADB supports porting between the android APIs of Unity and ABD (Royal, 2018)

DEVELOPMENT CONSIDERATIONS

With the availability of GPU alongside at least an I5 core CPU some of the intensity of rendering in Unity for VR alongside porting, as both are very expensive computationally. With this availability, it will allow for sharing the workload between the CPU and GPU whilst also enabling crashes and slow project development to be avoided. One point to make about Unity when VR support is used, several automatic changes are done to alleviate certain problems with VR. These changes offered are orientation tracking, stereoscopic rendering, along distortion correction to the main camera.

To enable data recovery in case of loss or corruption, a private Git repository was set up on GitHub and used extensively alongside providing the ability to work from multiple computers. SVN was considered but the features offered by Git were more appealing as it offered more advanced branching or working without an internet connection.

8 GAME DEVELOPMENT PROCESS AND METHODOLOGY

The chapter discusses the methodology and planning process of the project. The methodology of the project was based on the methodology of developing a game. This methodology was chosen as there are parallels that can be drawn in creating this project and regular games even with the project outcome aim to investigate the possibilities of the different aspects, while the product is only a prototype it allowed for a methodical approach in conjunction to a useful plan.

Within the methodology, 4 vital aspects define and help influence the planning of the development. These are Mechanics, Story, Aesthetics, and Technology. With so many game genres and types of games created, each aspect isn't always defined and isn't always a big influence within the methodology.

For the aspect of the story, this refers to how part of the game's unique feel is created as it is a way of giving a player an overview of the game setting and what is required for the game to be played. Within this game, a story isn't a pivotal part as it is a multiplayer game as it is gameplay driven not story driven.

A game mechanics refers to the rules and procedures created as a guide for players alongside how the game responds to a player's actions. The mechanics implemented within the game involve the various objects within the game along with the various scripts created and attached to them. An example of a mechanic is the use of VR within the game enables different gameplay compared to a non-VR game. One such idea related to a VR mechanic is the user interaction with their role type, and how these different roles need different actions to allow this mechanic. An example is a driver, which uses buttons to cause the tank to turn whereas the engineer has physical interaction with objects.

The aesthetics of a game revolves around how to make a game different and unique compared to games within the same genre. Aesthetics are influenced by several aspects, but the main influence is the assets used within the game as the assets used to enhance the unique feel of the game and help distinguish the game from others of similar properties. So, the procedural content and the shaders will come under this as these are the uniqueness and the parts which will enable the game stand out compared to others in the same genre. For the aesthetics of this game, it is based on a tank maze game so more military style. This will be enhanced with some procedural effects and shaders.

The final aspect is related to technology and refers to what technology and platform the game is aimed for. This isn't concise as cross-platform allows for the game to be ported to another platform than just the one it was created for but involves a more abstract implementation of the game system to support the various APIs of the platforms to port to. Within the context of this game, the game is aimed for Oculus devices but mainly the Quest 1 and 2 but cross-platform can be implemented later. Medium technical needs are required as the game is a Virtual Reality game alongside it utilizing the 3D and 2D graphics and shaders as well as making use of the renderer system to create runtime objects. A further technical problem will be creating shaders that don't inhibit the overall running of the game and feed into the game's aesthetic to create a more vibrant atmosphere around the core aesthetic. One further point to be made is regarding a player's mood and how it can influence how they play regarding the analog utilized by the Oculus Quest controllers. As such implementation will take this into account, so that either way a user is playing whether it be standing up or sitting down, the immersive experience won't be ruined.

CONCEPT DESIGN

The concept design was styled after well-known tank battle arena games but with a twist to satisfy the core concept of the initial game idea. The main concept within the design was to have users be contained within a team that followed around a tank. Each team would consist of 3 roles with each role having different functionality: Gunner, Driver, and Engineer.

One such idea is when client does a role, the action would cause a drain on energy which eventually stops the client doing any actions until the energy points are replenished. The engineer will be responsible for ensuring this. To enable this the engineer would be updated continuously through energy bars with a low energy amount signifying a client on their team requires a replenishing of their energy and enable the role to continue doing. To ensure each role is up to date with the latest energy state of each player in their team, the energy bars and state have networking functionality which provides the ability to update each client regarding each teammate's energy state and points.

Furthering the concept idea through using procedural generation which will come in the form of various kinds. One such will be the representation of the map using a procedural algorithm with a mesh generation algorithm to produce a 3D visualization. Alongside the procedural spheres being used as bullets.

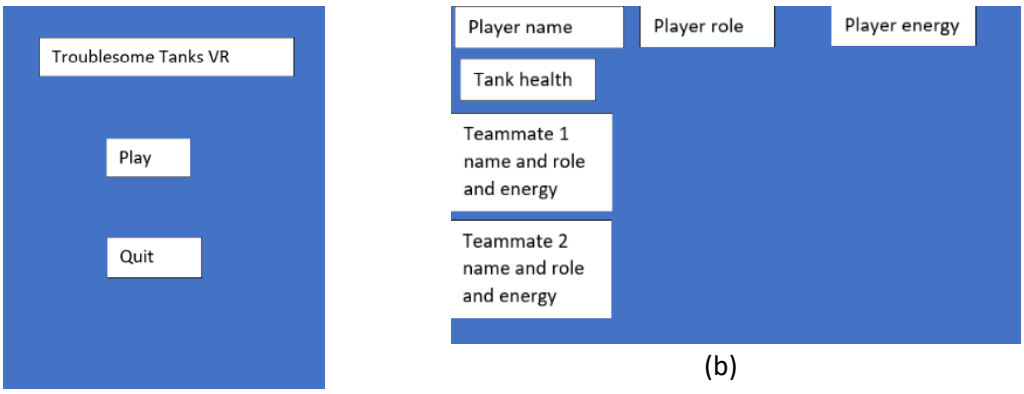
VISUAL DESIGN

The idea of the visual design was to use shaders to enhance the viewing experience and immersion of the game. But while keeping within the limitations of the hardware and producing a visually appealing game that enhanced the game experience of the user. A further note can be made in terms of the UI being designed to be visually appealing and not ruin any immersion within the environment.

UI DESIGN

For UI in games, it is usually an extra layer over the screen to inform the player of information relating to the game and update the player when new information needs to be relayed to them such as a nearby enemy or low health. Whereas with VR, UI is not just a 2D flat plain but will need to be molded to fit into the screen without losing any visual smoothness. Because of this, the use of 2 UI types were utilized in the game, one for the menus and one for gameplay. The menu UI was designed as a spatial UI and is attached to the environment.

But the game UI, was designed to not ruin user immersion as well as avoid health-related problems such as out-of-focused text causing eye strain or nausea. Alongside being designed as an informative way regarding any information relating to gameplay such as their teams' tanks' health.



(a) (b)

Figure 36: Example of the UI design for the menus(a) and main game(b).

9 IMPLEMENTATION

The project's focus was to implement a basic networking system compatible with a Virtual Reality game that contains the functionality of procedural generation and ray tracing. The focus was on if these various features could be implemented into a single game with a smooth FPS over a network, to bring forward how it can be done, and not necessarily implement anything new or cutting edge.

Using the methodology outlined in the methodology chapter along with the use of a basic plan with outlined outcomes and some leniency to allow for unexpected problems, a project was developed with most objectives fulfilled to create a VR game with networking, ray-marching, and procedural aspects. Within the project, there was 1 game level along with several menu scenes to set up the game by allowing player interaction with the UI.

The discussion of the various aspects developed is discussed by each scene and what was implemented within each, certain scenes were created to facilitate setup and to support a client's ability to enter the game as such there isn't much to discuss regarding these scenes. The main game scene is where most of the development occurred, so more discussion is made regarding that one. The chapter will give some in-depth detail of the various functionality researched and implemented, from the selected networking system implementation and its functioning on each client to the implementation of the procedurally generated content and shaders.

PLAYER

The player object within each scene refers to the use of the Oculus provided object known as the OVRplayerController. Each connected client to the game network would use this object on their client-side game as each object had attached the components to allow for player interaction with the game. In the OVRPlayerController, it had an object and component known as OVRAvatar which handles the tracking of the controllers through the SLAM algorithm so that real-time tracking and predictions of the controller movement can be done. Each player will follow through the scenes setting up their local client to then be able to connect to the game network to choose which role and team they are on. The player object will also provide each connected client the ability to interact with any UI contained in any scene.

UI INTERACTION

Where UI interaction was required, several elements were required to support the interaction as the UI within a VR application is very different compared to typical UI interaction such as mouse and keyboard or touchscreen. The solution to support this interaction was through an asset being attached to the main in-game camera which tracks the headset movement and creates the functionality to support user VR UI interaction. For the interaction the use of a ray caster attached to the headset in which the ray cast originated from the headset, and when a ray cast intercepted any UI or object in the user view, the interaction was possible. This enabled the use of input modules that recognizes user input and enables the user to interact with any interactable object through the headset having a gaze pointer attached. This would enable when the gaze point was over an interactable UI element such as a button, an event could occur on a click event.

NETWORKING SYSTEM OVERVIEW

The main functionality of the game is run on the server, with various NetworkVariables used to sync each client to the server's values. The main gameplay functionality is run through inputs received from each client, with the input corresponding to an action which causes the action to be played on the server. The server then updates the NetworkVariables to ensure each client has the same information. To ensure a more linear way of sending information to the server, each client would use the server RPC function offered by their tanks NetworkTankState component.

Architecture

The approach for the architecture was a host model with a compositional approach in which the client and server logic was split which avoids “god classes”. With this model, one of the six clients will act as the host/server with the use of a server- authoritative implementation. With this client utilizing the compositional aspect as the objects on its local game will have both server and client components. As such the approach, means each game object contains a server, shared, and client component. Enabling clients who are not the host to have all the server components disabled. This ensures the type of the client connected, only has components enabled related to their type and removes unnecessary functionality occurring.

As offered by MLAPI, the use of NetworkVariables (Appendix A) helps synchronize any variables which required synchronizing. The variables which are synced for the tanks are the position, rotation, speed, movement states. The variables synced for the players and tanks are the health/energy points amount alongside health/ energy states.

To ensure an organized approach the use of three separate assemblies: Client, Shared, and Server, with the Client and Server referencing the Shared assembly. The shared assembly holds any class which is shared between the server and client to enable specialized logic alongside easy access to NetworkVariables or RPCs. All the logic of the game runs at 30Hz, and matches the network update rate offered by MLAPI, with logic running through the FixedUpdate.

With how the RPC functionality offered by MLAPI works and the network components containing the RPC functions which each contain an event invocation. Which enables functions of scripts to subscribe to events being invoked through a callback. With the function running once the callback has occurred, enabling multiple functions to subscribe, and enabling multiple functions to run in parallel.

The main connection sequence is run through the NetworkPortal. Within this it contains the approval check, registering the Client and Server Message Handlers to support messages between the clients and the server. It also provides the events for invoking a host or a client.

SCENE AND GAME STATE FLOW

The flow for the scenes resembles a top-level Game State approach in such a way it resembles a 1:1 way. As within MLAPI, the functionality doesn’t support any clients being on a different scene than the server which considering this, the scene changes are host driven. With each scene controlled by the scene state of the server which also controls the transitioning of each client’s scene using the MLAPIs network scene management. Through this, each scene contains one SceneState prefab, which contains a client and server Scene State behavior. With this prefab controlling the global state logic of that scene

with the transitioning of each state depending on the scene. As such, a scene change will cause a state change. Within each state, there is several various RPC calls and events which with the event invocation stored in the RPC function, with a various functions of scripts subscribing to these events to allow for various actions to occur on the server and be relayed to each client.

The relationship of Game State and Scene is a 1-to-many as each scene corresponds to exactly one state, but a state can exist on multiple scenes. Each SceneStateBehaviour inherits from a NetworkBehaviour as it represents a discreate game state and offers a guarantee that only one Scene State can occur at a time.

For the scene state prefab, the Server SceneStateBehavour is only needed once the Server is running as such there is only 3 server states needed. The figures below represent the inheritance of each scene state in regards to the client and server scene states.

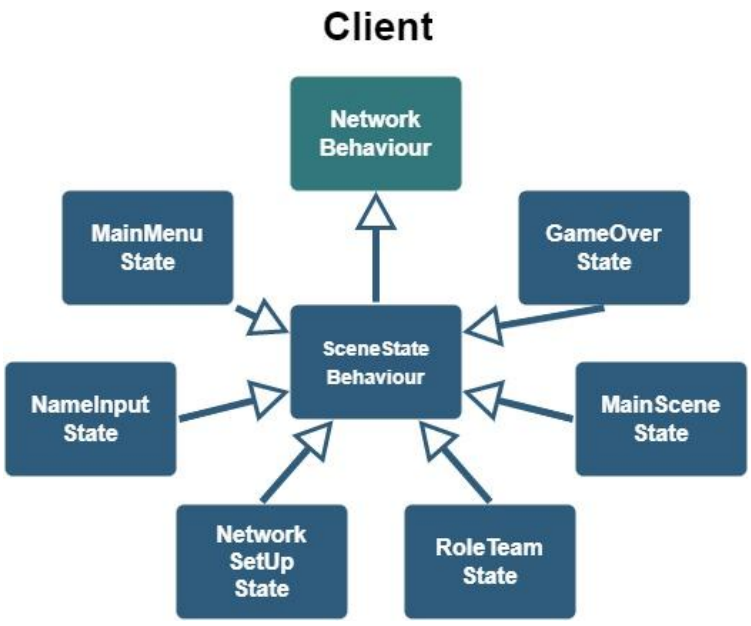


Figure 37: Client Scene State behavior inheritors.

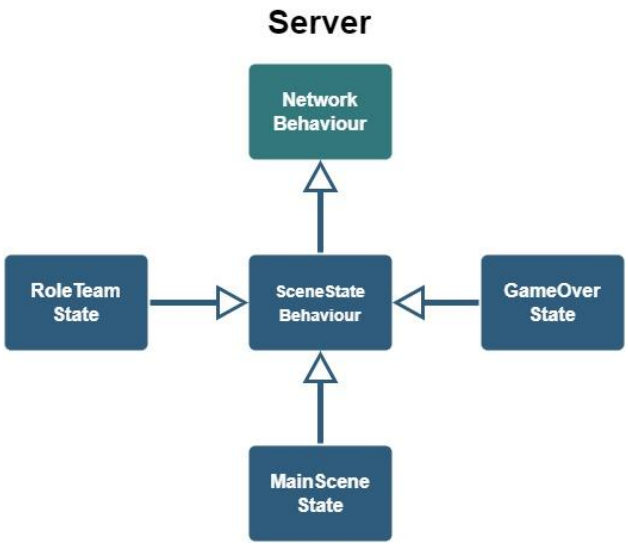


Figure 38: Server Scene State behavior inheritors.

Scenes

Within the main menu and name input scene, each was created to facilitate each client the opportunity to set up their local game before connecting to the server. In both scenes, the UI is within world space, not user viewport space. Each scene contained buttons to enable the user interaction with the Name input scene enabling clients to use the created keyboard to input a name or choose a random name, which will help users distinguish one from another once connected to the Server.

Network Set-Up Scene

The scene where the networking commenced is the Network Set-Up Scene and controls the logic for each client to choose to create a lobby or join a game. With the buttons contained within the scene, each has functionality attached, with a callback stored for when the client is happy with the IP and Port, the continue button will then enable the client's selected game running choice to occur. As such either the client will start a server and host the game or will join the created game.

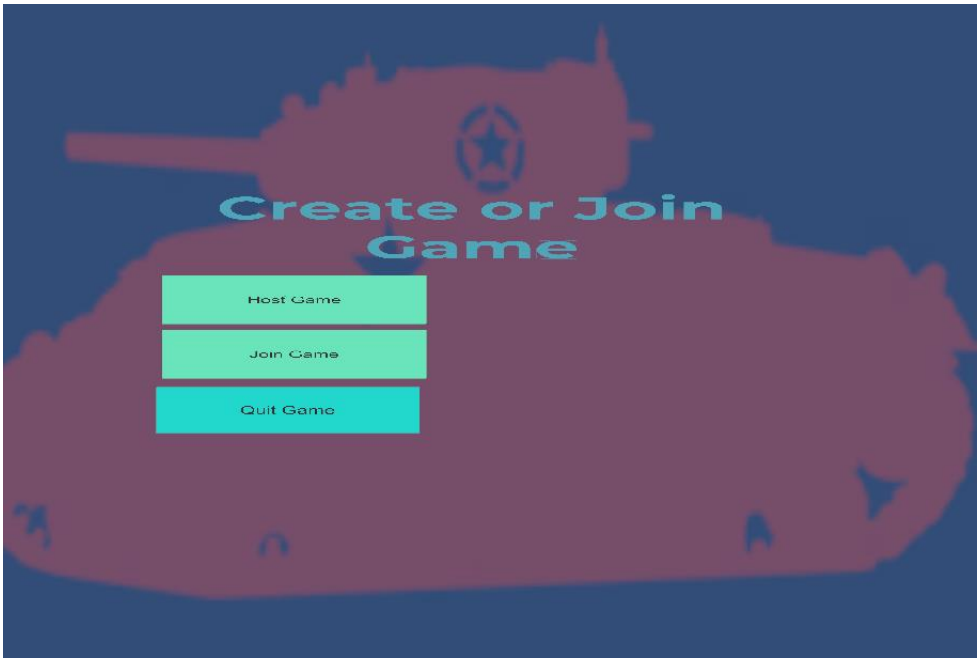


Figure 39: Network Set-Up Scene Menu

Role and team selection scene

Within the implementation, the first idea was to use a separate scene for a client choosing their team and role but to alleviate and optimize the scenes a bit, the scenes were merged, as it created an easier way for users to choose their team and role without having to wait for others to choose alongside easing some networking traffic between each client.

Scene logic

The flow of the GameState runs on various registered callbacks relating to the RPC client->server events. One such callback is OnConnectClient, which subscribes to an underlying MLAPI callback for when a client is connected to the scene, along with this callback occurring on the connected client.

As mentioned, each scene consisted of one GameState to ensure that the server could keep track of each client and which scene each client is on. Within this scene, the logic is handled by the ServerRoleTeamState, which handles informing each player of any choices made regarding roles selected. Which is done through a client->server RPC and callback which contains the information about the role selected alongside the seat index, clients name, client's id and team. This will be used to fill the UI information or inform the client that this seat is locked. Which in turn will update each client of the roles selected, as the use of a NetworkList offered by MLAPI, allows for functions to subscribe to a list change server side. Which if the server receives a role selection and updates the list, each client will use this information to then update the related UI with the information of the list. Enabling each client to be synchronized to the information of the server. The server will change scene for each client, once each client has readied up, with a RPC called with a Boolean parameter of true signifying to the server the client is ready. Enabling the server to then change scenes for each client through a event trigger. With the event trigger, saving each clients selected roles configurations of role and team into a RelayObject. The RelayObject, which allows for saving variations of different custom or Unity types between scenes,

will be used to ensure each client once spawned in the main game scene has the correct information set. The use of coroutine occurs to and changes each client's scene to the main game scene.

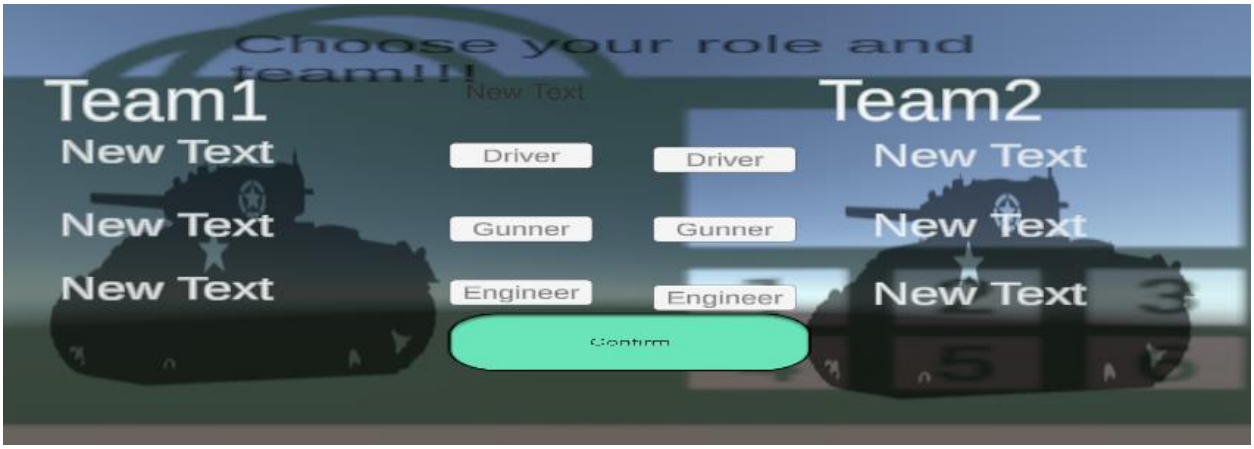


Figure 40: Role Team selection scene UI

Main Game

The main game scene is where the main gameplay and networking functionality to support user interaction occurred. Further points to be made in creating a further immersive feeling are with the use of a parallax shader in conjunction with a ray marcher.

UI

The use of dynamic UI across the screen space with it containing several pieces of information regarding their role, name, and energy bar. Furthermore, it contained the information of their teammate's role and energy along with the health bar related to their team's tank health. The local player's information is stored within a local singleton game manager to enable information to be easily accessed even after the player prefab is deleted on scene game.

Main scene state

The prefabs server state contains the state behavior of the level and handles the main game logic alongside spawning in each connected client with the correct attached information to facilitate each client to be able to do the functionality related to their role. Each player spawned has its network player state information all set along with the UI elements in relation to their teammates and tank health/energy.

Action flow

To illustrate how the actions flow across the Network and it supports the client sending the input action to the server with the subsequent gameplay occurring based on the received ActionData. A small discussion of the various created scripts and their purpose in context of the game and how they facilitate in helping actions be sent by a client to the server.

Network Actions

Data model: The actions and the data attached are defined by ScritpableObjects, with these objects organized by an Enum and made available through a singleton: GameData, which holds the ActionInfo.

Each action represents discrete action functionality to describe what the action and the outcome it should do. The actions are substantially data-driven.

Action sender: The main class which handles sending the actions to the server from each client as it listens for inputs, interprets them, and then calls the appropriate RPC on the RPCStateComponent. But as the aim is to run the actions through each client's tank, the use of the NetworkTankState component RPC. As such the action will then be received by the tanks RPC and any functions which are subscribed to the event invocation will be called and enabling the action to occur based on the action information sent by the client. It performs the actions in a sequence and ensures only a set number of actions can be done at a time. Alongside advancing the queue of actions based on the action information of each in the queue which will help determine if the action should occur or be removed.

Action: Action is an abstract base class for all actions which can occur on the server.

INetworkMovement: Interface containing network variables representing a objects position, rotation, and speed. This is inherited by any objects which require updating on both the client and server.

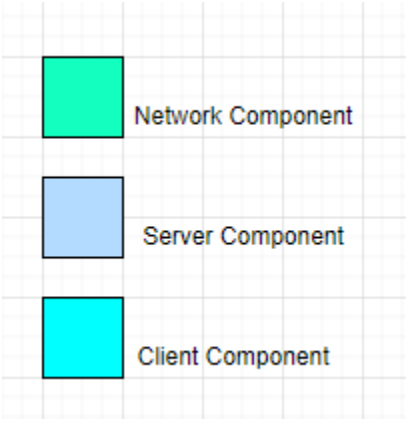


Figure 41: Color key for each components type

Tank

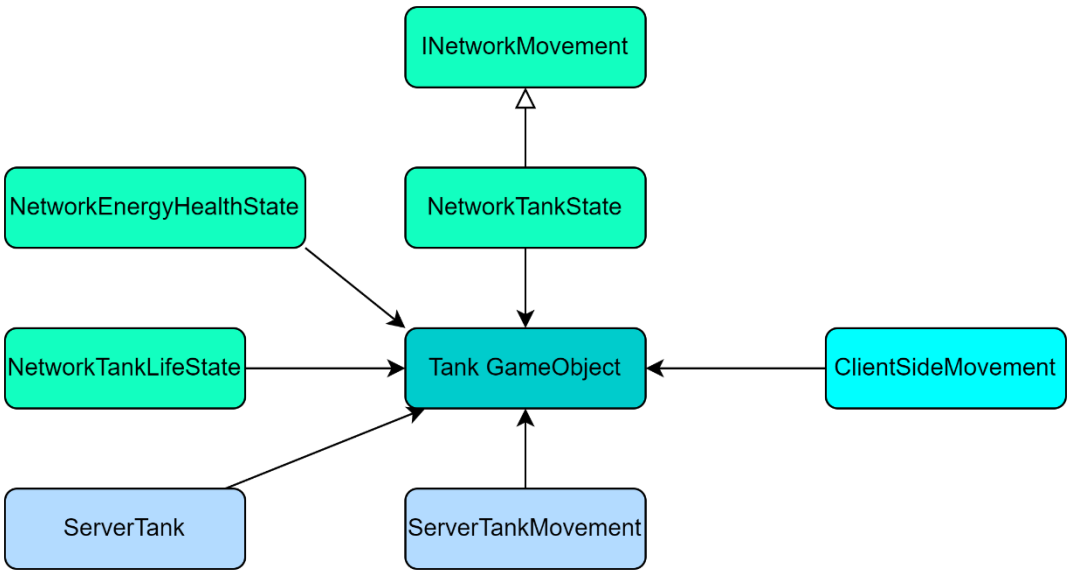


Figure 42: Top-level design of components attached to a Tank GameObject

NetworkTankState: This contains the NetworkVariables which store the state of each tank, as well as containing the server and client RPC endpoints. The endpoints don't handle any logic but read out any call parameters and raise events from them. The variables which the component handles are the position, rotation, speed, movement, and health state. The server will update each based off the gameplay actions and ensure each player is synced and up to date with the latest server values. It also stores each player of a team and its role, which will help with enabling a player access to another player information if needed.

ServerTank: The class functions to handle the actions received from each client. It will receive inputs from client inputs in terms of UI button events or gameplay. It will then execute each action received. The OnActionPlay function subscribes to the event of the tank action ServerRPC to enable when an action is sent by the client, the server can deal with it.

Server Tank Movement: The Driver player of each team handles the movement flow of each tank. With when a client interacts with a button, the ActionOption is set to represent which driver action has occurred. The Client->Server RPC will contain the driver action type, which will cause the movement state of the tank to change based on the action type. A duration time will be used to inform the script of how long the action should occur till the tank becomes idle. With once the action has occurred, the server representation of the Tank will update the Network Variables of Tank Network position, speed, and rotation at a rate of 30FPS.

ClientSideMovement: The script is responsible for updating each client's version of an object based on the NetworkVariables

Player

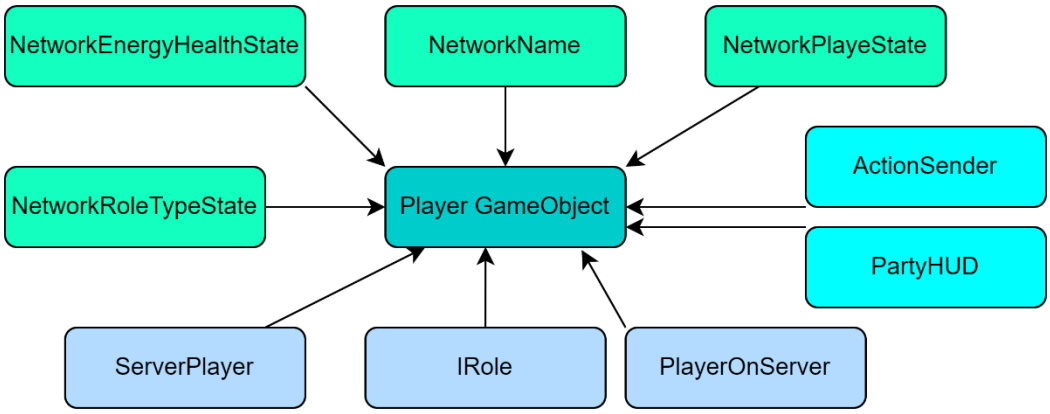


Figure 43: Top-level design of components attached to a Player GameObject

NetworkPlayerState: The scripts holds the information of the NetworkVariables of each player but in the case of the game, it only holds the energy bar points as well as the energy state of the player.

ServerPlayer: The script is handles updating the energy state of the player based on the energy points value. Which is achieved through the function OnEnergyChange subscribing to the RoleEnergyState value change which ensures when player has no energy left, the state will be changed to no energy. Which ensures that players with the energy state HasEnergy, to be able to send actions to the server.

ClientPlayer: The script is responsible for setting the objects related to the role chosen by each client.

PlayerHUD: The responsibility of this script is to update each clients UI elements in regards totheir own information alongside teammates and their tanks information.

Bullet

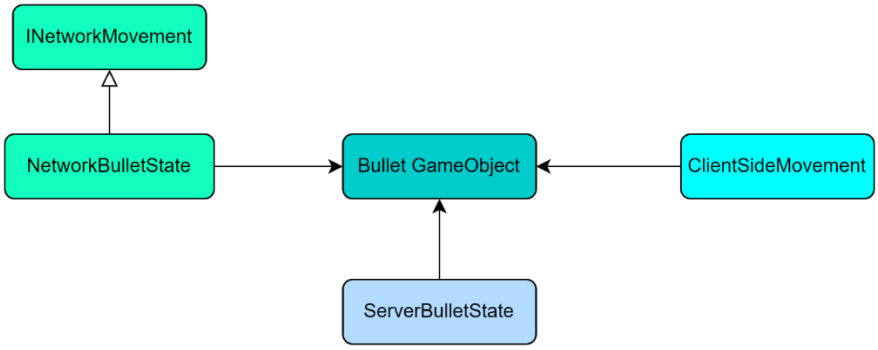


Figure 44: Top-level design of components attached to a Bullet GameObject

NetworkBulletState: Holds the network variables related to bullet regarding transform information alongside its speed and its team information

ServerBullet: The script is responsible for controlling the bullet on the server side alongside updating any of the network variables linked to it.

ActionDataRequested:

The class is comprehensive and contains all the information to enable any action to be played back on the server. It is the data packet which is serialized and sent to the server via the RPC. The packet contains the information for how an action should then be played.

GameData

Each of the actions which can occur are defined in GameData and are ScriptableObjects. GameData is a singleton class which makes actions available to anywhere in which it is required. Each action is defined by Action info and is stored in the Game Data singleton within an enum to ensure easy access.

Action Info

The action data which describes a single action, and the information needed to affect gameplay.

Role actions

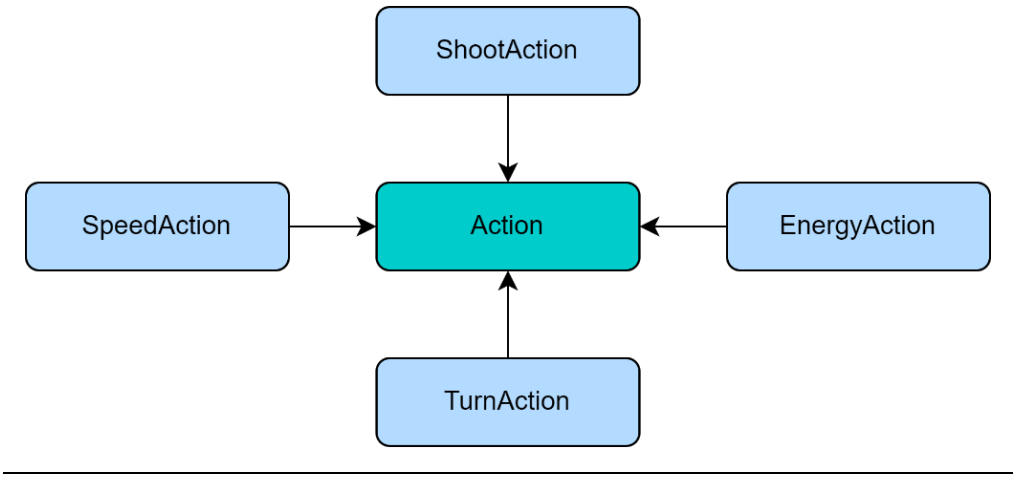


Figure 45: Top-level design of action inheritance

EnergyAction: The action related to when a player has run out of energy, the engineer can replenish the energy for the role with this action.

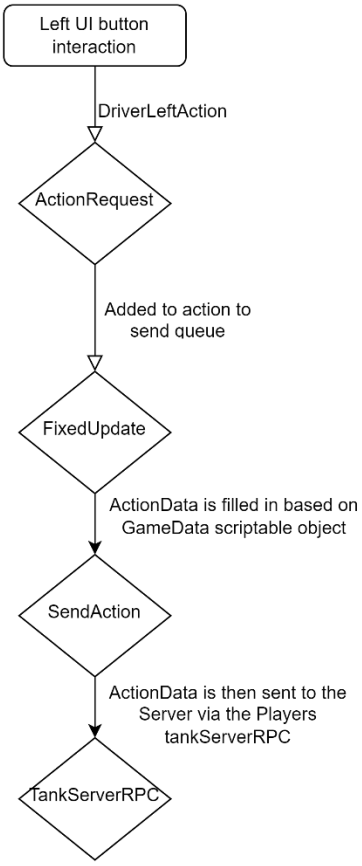
TurnAction: The action informs the tank the direction in which the driver has requested it turn.

SpeedAction: The action informs the tank to move forward for set duration.

ShootAction: The action informs the tank to shoot a bullet in front of it, with the bullet randomly chosen off the bullet objects provided in the GameData class.

An example of an action can be illustrated through the flowchart below, which provides an idea of how an input from the client is sent to the server and how that action is then played by the server.

Client-side action flow



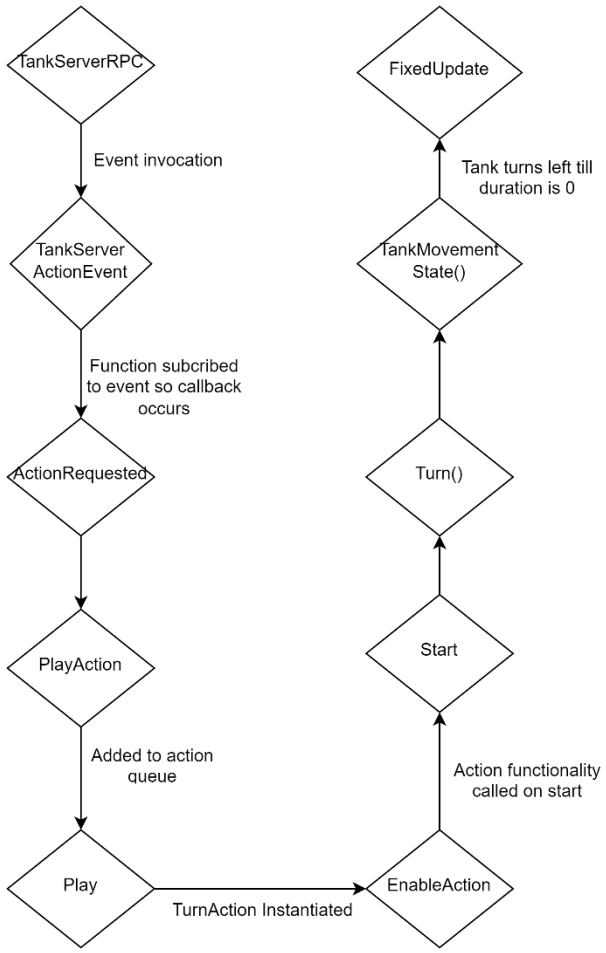
The action in question is a driver action of turning left. As seen the first step is the UI button interaction by the client, which provided to the ActionSender script through the function ActionRequest(). Within the parameters set in the function is the action type. This will cause the action to be added to the queue of actions to be sent to the server.

Within the FixedUpdate, the serializable ActoinDataRequested is filled out based on the scriptable object stored within the GameData which uses the actiontype to find the information related to the action.

Once the action has been filled out, the function SendAction() is provided the ActionDataRequested, which then sends the ActionData to the Server through the RPC component of the Tank.

Figure 46: Flowchart of a turn left action client side

Server-side action flow



With the action now sent through the RPC, the server-side representation of the RPC received the ActionData. Which invokes TankServerActionEvent.

A callback then occurs where the subscribed function ActionRequested, uses the ActionData to set in motion playing the action on server.

The process of this is done through checking the action can occur as other actions can stop this occurring. Which if the action is able to occur, the abstract base class function of Play() is called which will determine if the action is front of the queue. If so, the correct action is then instantiated based of the ActionType of the ActionData.

With this instantiation, the action will then be created, with it being added to the Unity pipeline process and enabling the actions functionality to then occur.

Figure 47: Flowchart of a turn left action server side

In this case, the action is to turn left, so the Turn() function will be called, which will then send a move state change to the ServerTankMovement. Which will cause the tank to subsequently turn for a duration for the amount set within the ActionData.

Within the fixedUpdate the Network position and rotation of the tank is updated to enable each client to be informed.

Procedural objects

Contained in the scene, is a procedurally generated map that was generated based on the Moore Algorithm of Cellular Automata and a flood fill algorithm to ensure a more organized map without disconnected areas. To visualize this map from 2D to a 3D visualization, the marching squares algorithm was implemented. Further procedurally generated content is 2 variations of a procedural sphere with one generated through tessellation and application of a parametric equation, with the other utilizing a ray marcher to produce an implicit sphere confined in a 3D object. The spheres were utilized as the bullets for the shoot action, with a randomly chosen one of the 2 used when the action occurs. One final note to make is the implementation of a parallax shader to produce a higher quality sense of immersion.

Procedural map

Cellular Automata

Cellular Automata was implemented as it gave the ability to implement a larger area of gameplay with some simple rules which offered more regarding the gameplay needs of the specific project, unlike the other algorithms discussed. The implementation of the Cellular Automata utilized the Moore neighborhood algorithm which comprises all eight immediate adjacent cells and is illustrated in the figure 48, alongside a simple rule to enable smoothing. As each cell can be represented as a square-shaped neighborhood it can be defined at each given cell (x0, y0) with the use of the Moore neighborhood the evolution of each cell will be affected by these neighbors. The Moore neighborhood of range r can be defined as:

$$N^{M}_{(x_0,y_0)} = \{(x, y) : |x - x_0| \leq r, |y - y_0| \leq r\}.$$

x-1,y-1	x,y-1	x+1,y-1
x-1,y	X,y	x+1,y
x-1,y+1	x,y+1	x+1,y+1

Figure 48: Moore neighborhood

The process of this is 2 steps. The first step involved is the initial fill, where 1(wall) represents an alive state and 0(floor) represents a dead state. Involved in the step a seed is used which represents what percentage of the cells should be alive. The next step, the smoothing step where several iterations, each cell based on their neighbor's state evolves to be more like their neighbors creating a smoothed map and reducing overall noise. The smoothing step involves the application of the rule known as the Life transition rule, in which at each step, 3 outcomes of the cell can occur. These are birth, survival, or death. Each outcome can be described:

1. Birth represents the cell transitioning from dead (0) to an alive (1) state as 5 out of 8 neighbors are alive. With the approach for the project, this means that the cell has changed from an empty to a wall.
2. A second outcome is survival, which means 4 neighbors were alive meaning it remains alive. In the context of the project, this means that the cell remains a wall.
3. An outcome is death in which the cell is alive but 5 out of 8 neighbors are dead causing it to transition to a dead cell. The death can be due to loneliness or overpopulation. In the context of the project, this is a wall (1) becoming an empty (0).

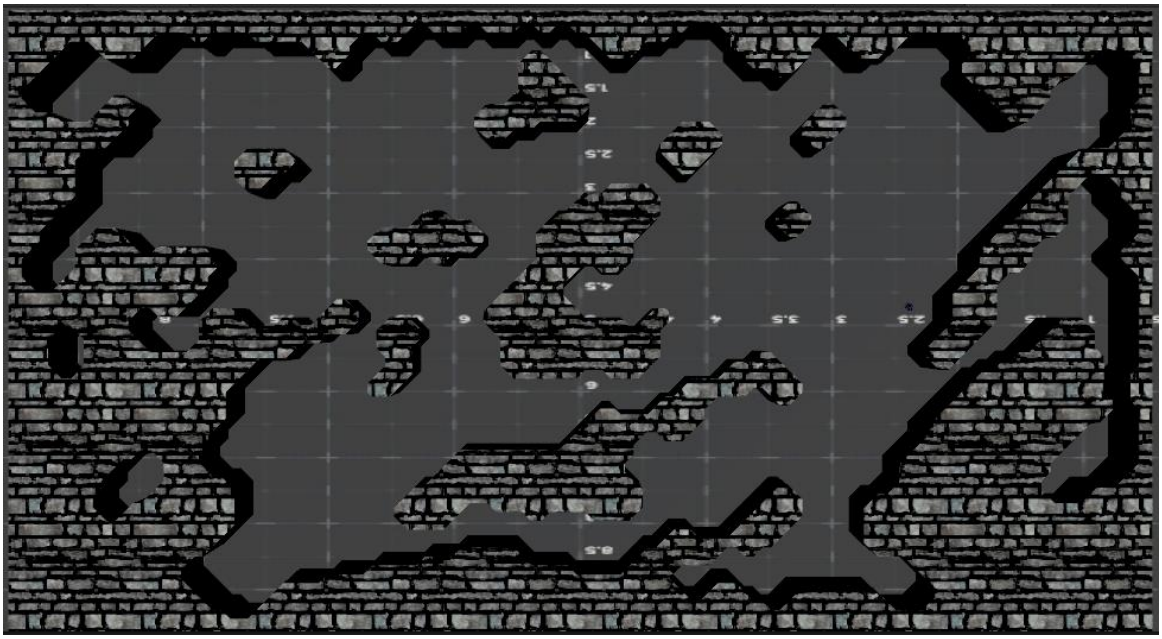


Figure 49: Cellular automata, Seed = 50, Intial fill = 45, without flood fill

After the smoothing step, a more organized map is created such as figure 49, but in the context of the game, there are certain problems encountered as there is still not much control over where the areas should be filled with walls and where shouldn't be while also creating unplayable areas. As seen in figure 49. This is known as the isolated cave problem and can be solved with the use of either a flood fill algorithm or horizontal blanking. Each solution is described on the website Rogue Basin (2016).

Solutions to Isolated cave problem

Flood Fill

There are two implementations of the flood fill algorithm which can be used. One such way is where a random open point on the map is flood-filled, which will cause any open point outside the flood-filled portion to be adapted and changed into a wall. It can then be used to check if the portion of the map which has been flood-filled is more than the allowed threshold percent of the map but if the map is below the threshold the whole cellular automata and flood fill commences again.

Another such implementation is through finding each area made of open points and determining if the size of the open points region is above the threshold size wanted, with the biggest then used as the main region. Furthermore, other regions deemed big enough are then connected to the main region through a passageway which uses the shortest distance between each region. With the shortest distance found by a gradient calculation between each connection.

Horizontal blanking

Horizontal blanking is an approach that involves blanking a horizontal strip of walls, 3 to 4 blocks tall, in the middle of the map after the random fill but before the automata commences. If the rules are sufficient, the horizontal strip width will help prevent any continuous walls from forming and stop any disconnected sections from being created.

Chosen solution: Flood fill algorithm

The chosen solution to solve the isolated room problem was the flood fill implementation but using the latter approach discussed. As such, each room would be connected to the biggest rather than the culling every room apart from the biggest. The idea for implementation came from Rouge Basin cellular automata (2016) with the implementation of the passageways based on the approach by Sebastian Lague (Procedural Cave Generation-Flood fill algorithm, 2014). As ensuring each room could be connected by a passageway it is imperative that each passageway is wide enough, as such the implementation of this ensured each passageway would be the size needed.

Fundamentals of the algorithm

The overview of the flood fill algorithm can be described as where a cell is selected, and each adjacent cell, no diagonal, of a similar metric is found with repetition until the metric is different than the original cell. This indicates that the edges of the region were found and enables all the regions of the map comprised of the type wanted for each region to be stored.

Algorithm implementation

Finding each region

Through an iteration of the map, each region and its size are found. The size of the region is then used to determine if it is above the threshold size wanted for a region, if not all the regions' cells are adapted into walls. With a region found, it will store various information about the cells it is comprised of alongside any rooms in which it shares edge tiles with. Once each region above the threshold is found, the main room is set through a sort that determines which is the biggest of the regions.

Main room connectivity

The next step is to ensure each region is connected to the main region. This is achieved by using 2 lists of the regions to enable each region to be compared against another region. An iteration through the list of regions while comparing the second list of rooms against the first to find any regions which are connected. The comparison is achieved using each region's edge tiles to determine where 2 regions share the same edge.

Shortest distance

With each region and its connected regions found, a determination test is needed to find the shortest connection distance between each room and the main room and if it is possible to create a connection. The shortest distance is found using the edge tiles where the lowest distance between each edge is used to determine the shortest distance.

Each edge tile and the distance between each is iterated through to ensure the shortest distance between each room is found with the region than having a bool set to inform the passageway creation stage that this region is the one to connect to the main region. Each region and regions connected to this region, are now accessible to the main region. If a region and its connected regions are deemed not to be accessible to the main region, a connection is found by force through finding the shortest distance to a room with a connection to the main room or a connection straight to the main room.

Passageway Creation

Each passageway uses the shortest distance to be then placed through using the gradient of the connection as each passageway can be represented as a line which helps find the gradient. The shortest gradient found will be used to ensure the shortest passageway between the main room and the disconnected room. With each possible passageway and tiles collected, a gradient accumulation is done based on the closest tile and furthest tile distance. A line is then drawn between each tile coordinates, with a step accumulation to determine which passageway gradient is shortest, with the shortest then used as the final gradient. The use of a circle calculation is done to ensure each passageway is wide enough.

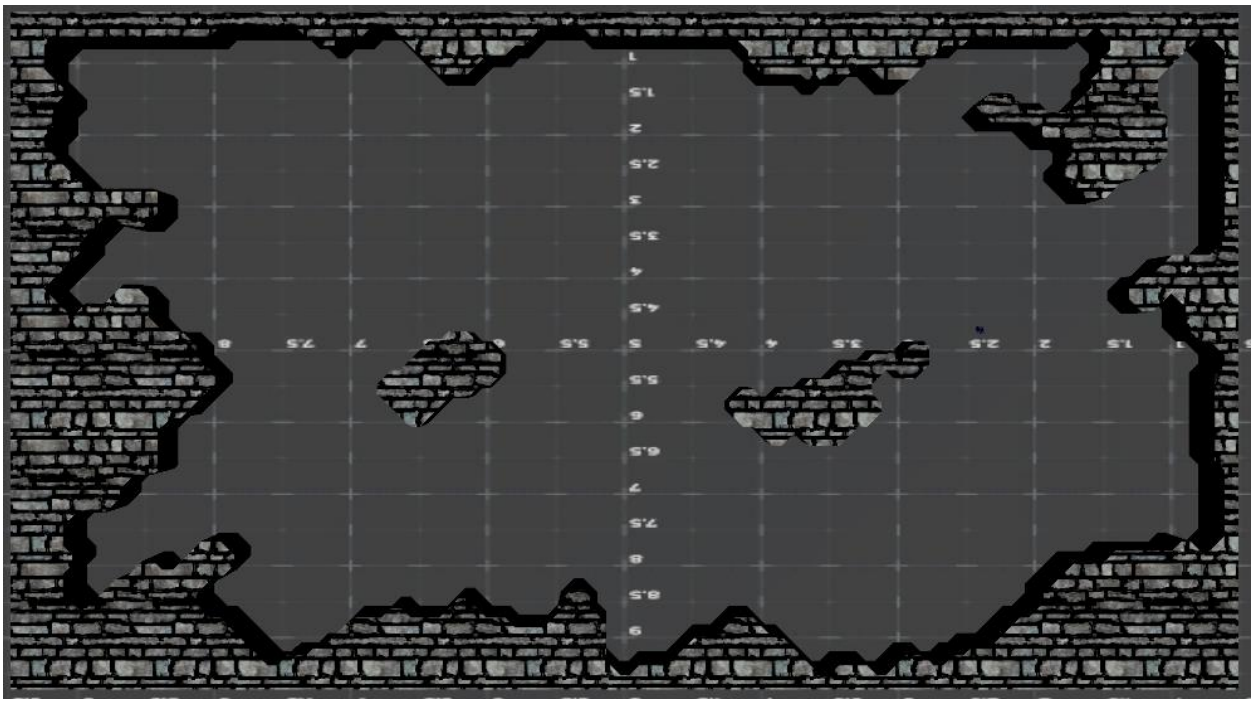


Figure 50: Cellular automata, Seed = 50, Intial fill = 45, with flood fill

Marching squares

The marching squares algorithm facilitates the ability to visualize and convert an implicit 2D surface into a 3D polygon mesh. The algorithm can be used to determine if an arbitrary point lies within the bounds of an object and consists of various steps but the main 2 steps, one to partition the space into smaller cells, then each cell an approximation is done to determine the surface as polygons.

In the circumstances of the project, the marching squares algorithm was deemed to enable the best mesh representation. As Delaunay Triangulation is more suited for terrain generation rather than a controlled system with the ability to create a map and find connections.

The implementation is based on the marching squares approach by Sebastian Lague (Procedural Cave Generation- Marching Squares algorithm, 2016), as it provided the necessary outcome alongside the approach for generating the geometry of each square and edge found with the interpolation. An analysis

of this approach is discussed alongside its utilization within the project.

Start

As it is a polygonization of a 2D map, the 2D array representing the map is provided after the flood fill has been completed. The 2D map and how it is represented gave basis to the marching squares algorithm to enable the isolines of each square to be found and linked up to enable triangles to then be extruded from the found isoline.

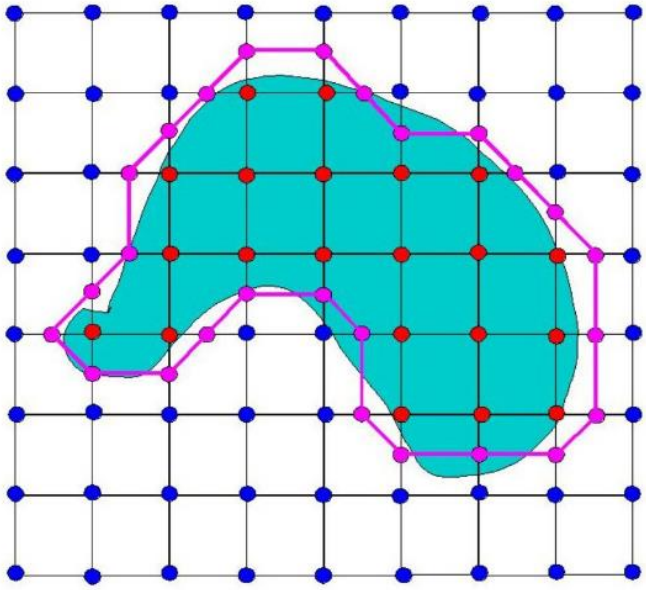


Figure 51: Marching squares algorithm example. Where the blue dots represent outside the object and red inside. The purple dots show the intersection of the surface with the squares. Each purple dot is connected to form the isoline and provides an approximation of the original surface. (Anderson, 2015)

Algorithm implementation

Step 1

The algorithm process works by iterating through each scaler field (cell) with each consisting of 8 nodes, with the 4 corner nodes acting as the control nodes and are also referred to as scaler values, with these 4 nodes used to form an imaginary square. The 4 the control nodes for each corner represent a bit value which is used to find the topological state of the square while enabling classification of each vertex with the in or out index.

These 4 normal nodes represent the middle points between the corner nodes and will be used to enable the different triangulation configurations and makeup within the square and generate the 2D polygon to represent the part of the iso-surface which passes through the square.

Step 2

The creation of a grid of squares, the size of the map with each control node-set. Each control node of the square has its bit amount set through an application of a threshold to find the binary of each square

where a control node will either be an iso-value of 1 or in this case the use of bool to specify if it is active 1 or unactive 0 which represents inside the isoline or 0 representing outside the isoline. With the square's final bit amount determining its topological state(configuration) and classify the contour location for each edge and determine the actual index of the square's polygon indices array.

The index array is a precalculated array of 2^4 possible of 2D square polygon configurations within the square. The bit values of each corner control node will have a bit amount set with the use of a lookup table to generate the configuration, with a walk around each cell in a clockwise direction, with each bit added to the square whole bitwise value through bitwise OR and left shift. Most significant is top left, with the resultant 4-bit index, based on the 16 different configurations.

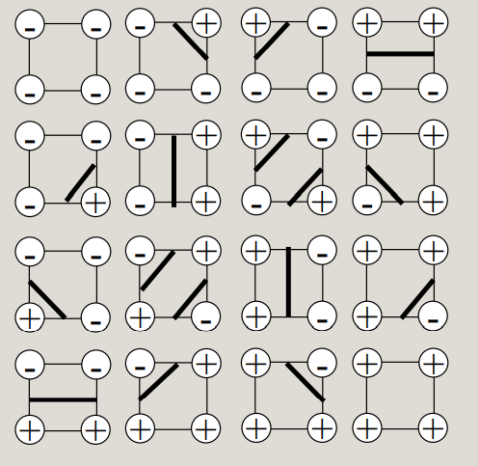


Figure 52: Configurations of each square based off the 4-bit index value

Step 3 Triangulation

The next step is a triangulation of each square to enable each squares configuration to be found, out of the 16 specified. For each square, the generated polygons can have each vertex placed in the position dictated along the edge of the square through a linear interpolation of the two scaler values connected by that edge to find the exact contour line along the edges of the cell.

Step 4: Triangle Extrusion

The vertices to extrude the triangles are found using the contour line and the squares it connects to. A function to generate the vertices for each square is MeshFromPoints(), which assigns and generates a list of vertices for that square alongside adding each vertex created to a list. This is done through the Params parameter which enables an unspecified number of nodes to be passed into a function. The squares configuration from the 16 is used to determine how many triangles made up the square alongside which nodes are the ones used as the vertex locations.

Step 5: Polygonise

The final step of the algorithm is the creation of the polygonal mesh as it consists of the triangulation of all squares to extract and extrude the triangles based on the vertices created from each square's configuration. Each square stores the indices to vertices in the vertex list, with a minimum of three vertices in each, which enables the indices to create a single triangle within the mesh object. The process of this is done by determining if a triangle is an outline edge by checking if 2 vertices share only

1 triangle. A clockwise generation must always be done when the outline edge is being found. The dictionary of the vertex and list is used to determine if 2 vertices share a triangle by comparing a vertex against the list of each vertex with if more than 1 triangle is shared the vertex is not an outline edge. Through this a 3D mesh is created based off the 2D array provided from the cellular automata and flood fill.

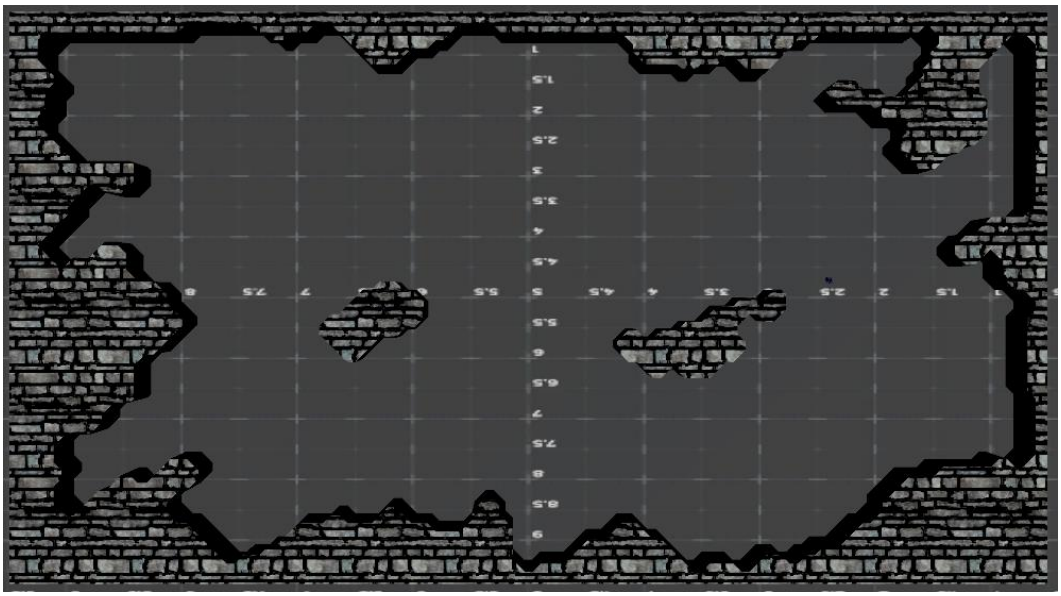


Figure 53: Cellular automata, Seed = 50, Initial fill = 45, with a flood fill algorithm and Marching Squares algorithm to extrude triangles to form a mesh

Parallax shader

In building the aesthetic of the game and visually enhanced experience, the implementation of a parallax effect for the generated maze was achieved. The technique was used as it boosts the textured surface's detail while conveying a sense of depth alongside normal mapping enabling a realistic outcome. It enables the real-time approximation of displacement mapping with a parallax effect, through sampling depth information from a height texture. The depth information is then used in each pixel's UV coordinates which are adjusted at render-time, with the adjusted UV, a ray is traced from the adjusted point until it intersects the view vector. As such it creates the illusion of depth as the viewer's eye moves across a scene. The new coordinate found is then used to sample the remaining textures.

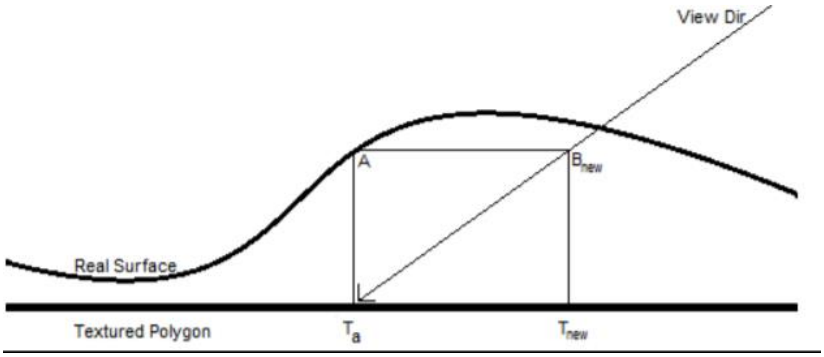


Figure 54: Offset example where the result is only an approximation. (WhitePaper, 2012)

The formula for the technique can be represented as:

$$T_{new} = \frac{Ta + (V_{(x,y)} * h_{scaled})}{V_z}$$

Where Tnew denotes the offset texture (UV)coordinates. The process of finding the new offset $\overrightarrow{AB_{new}}$ is through a conversion of the view, denoted as V, into tangent space and normalizing it. With the now normalized view, the x,y components are scaled based on the z component which causes x,y to now lie on the plane of the surface, alongside the Vz running perpendicular to the surface. With sampling the height value from the height texture, which lies in the range 0 to 1, it can be scaled and biased to account for the surface being simulated and multiplied by V(x,y), with the offset then found through the new vector being added to the original UV coordinates.

But to improve on this, a further step was undertaken and is known as steep parallax mapping, as instead of sampling the height texture once, it takes multiple samples enabling a better pinpoint of the vector A to B. It achieves this improvement by dividing the total depth range into multiple layers of the same height/depth. With each layer then sampling the heightmap, with the texture coordinates being moved along the ray vector within the raymarching loop. This keeps occurring until a sampled depth value is less than the depth value of the current layer. With this it visually improves the effect through generating more detailed sillhouettes compared to when only one sampling occurs.

Further enhancements are with the use of a bump scale to illustrate the effect further, which with use of the UnpackScaleNormal offered by Unity, a further bump scale can be used with the normal map sampling. With the normal map then transformed into tangent space. The application of the effect is shown in figure 55.

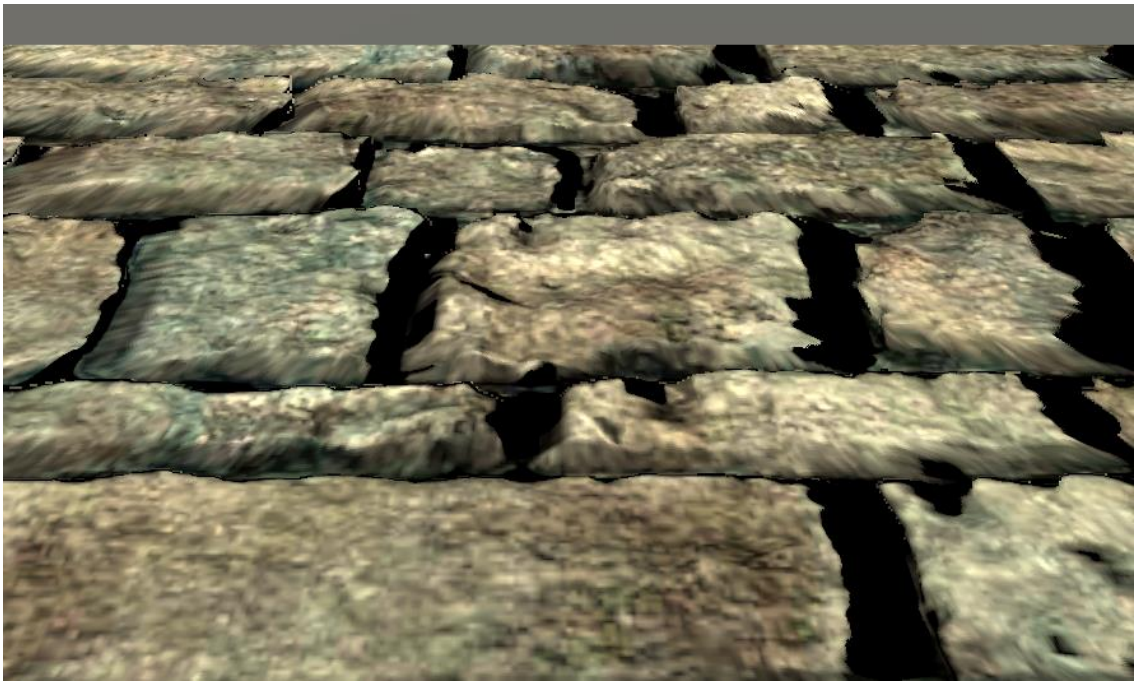


Figure 55: Steep Parallax effect

Procedural spheres

Tessellated sphere

Within the implementation of a sphere, one approach, as discussed in chapter 3, is the implementation of a UV sphere using a parametric equation. Within the current rendering pipeline technology of shader model 5 and GPU technology, there is the ability to do a process known as the tessellation which enables a programmer to subdivide a polygon via the GPU into smaller triangles. This is achieved through the Tessellation stage which comprises of 3 stages: the Hull shader stage, the Tessellation stage, and the Domain Shader stage.

The Hull shader enables a tessellation factor to be set to inform the GPU how much subdivision should occur. Within the Domain shader stage, there is the ability to utilize parametric equations on vertices. A small overview and purpose of each stage of the tessellation process to provide an understanding of how the sphere was achieved. Even though it can work with triangles, quads, or isolines, the same domain must be set for both the domain and hull shader as they act on the same one.

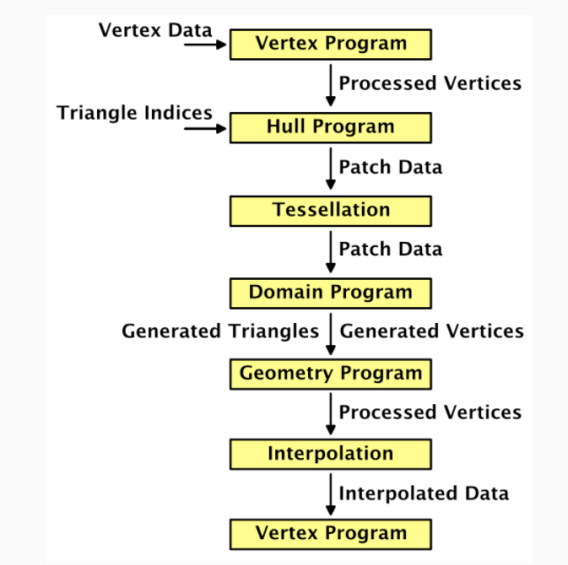


Figure 56: Shader Model 5 pipeline with Shading alongside Tessellation (Flick, 2017)

Hull Shader

The Hull shader is the first stage, which operates on a surface patch and receives the output from the Vertex shader in the form of the vertex data. The surface patch comprises of a collection of mesh vertex data. The Hull shader comprises of 2 parts which runs in parallel on the GPU, the Constant Function, and the Hull Function. Figure 57 illustrates this. With both parts being able to access the full primitive information with the purpose of each to do any per-primitive calculation.

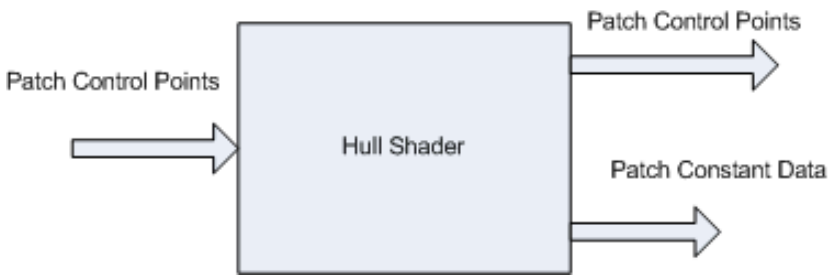


Figure 57: Input and output of Hull shader stage (Microsoft, 2020)

Constant Hull function

The Constant Function is invoked once per vertex with the purpose to output the tessellation factor for use in the Tessellator which is the parameter that defines how much the Tessellator will subdivide the provided primitives. Within this function various information needs to be set for each patch to inform the GPU how the patch should be subdivided for the provided vertices, with a quad specified as the domain, meaning the output control points must be set to 4, and to ensure each patch is evenly partitioned, the specification of "integer" is set for the partitioning. As the domain specified is a quad, this means the patch will have 6 tessellation factors to affect the subdivision.

Figure 58 outlines how each of the 6 tessellation factors of the quad can be adapted by a tesselation factor, but to ensure a symmetrical subdivision on each, a base of 100 is set.

```
PatchConstantOutput PatchConstantFunction(InputPatch<TessellationControlPoint, 4> inputPatch, uint patchId : SV_PrimitiveID)
{
    PatchConstantOutput output;
    float tessellationFactor = 100.0f;
    output.edges[0] = output.edges[1] = output.edges[2] = output.edges[3] = tessellationFactor;
    output.inside[0] = output.inside[1] = tessellationFactor;
    return output;
}
```

Figure 58: Constant function on a Domain patch

Main Hull Shader

The main Hull Function is invoked per-vertex or control point for the patch, alongside the additional argument which specified the control point(vertex) type it will work with. The Hull shader then outputs the tessellation to both the control points of the Domain shader and Tessellator.

Tessellation stage

The next stage is the Tessellation stage which is responsible for the subdivision of the primitives. Based on the tessellation factor received from the Constant function, it enables the Tessellator to create new triangles which compose a regular grid with texture coordinates (UV) varying from 0 to 1. With the new triangles passed into the Domain shader as texture coordinates. (Valdetaro, Nunes, Raposo and Feijó, 2010) There are 3 ways in which the tessellation is achieved but "Integer" was used as it enables a more symmetrical subdivision compared to the fractional ones.

Domain shader

The Domain shader is the final stage of the Tessellation process which is invoked once per newly generated vertex outputted from the Tessellator. Furthermore, being provided the UV coordinates for

each vertex through the SV_DomainLocation semantic which uses these coordinates to derive the final vertices. The weight of each of the 4 control points is established based off the X, Y, and Z coordinates.

The main purpose of the domain shader is to interpolate the new vertices along the passed in control points and transform the parametric UV coordinates into world space coordinates. This is where the spherical parametric equation to create the parametric surface can be applied. The equation can be presented as:

$$f(x, y, z) = (r * \sin\ \varphi * \cos\theta, r * \sin\theta, r * \cos\varphi)$$

R represents the radius of the sphere and the $\varphi = [0..\pi], \theta = [0..2\pi]$, as the UV coordinate received by the Domain shader from the Tessellator is within the range [0..1], By doing $\varphi = [U * \pi]$and $\theta = V * 2\pi$ and applying the parametric equation onto each of new vertices of the cube to ensure each is converted into spherical coordinates such that any point of the object is in 3D space, and can have its position determined based on the origin position and use of θ and φ. The passed in vertex data of the normal also has this parametric equation applied to ensure lighting is properly calculated onto the sphere.

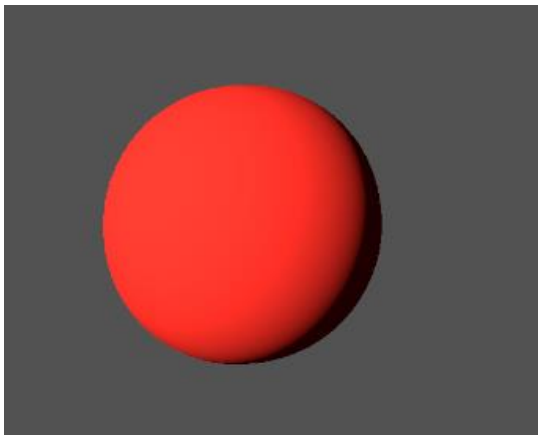


Figure 59: Phong shaded Sphere created through tessellation and an application of a spherical parametric equation

Interpolated Data

To ensure the now interpolated vertices are used, within the Pixel shader, a further Vertex shader is used which is passed the generated interpolated vertex data of the Tessellated mesh from the Domain shader to then be passed into the fragment shader stage. Phong (1975) lighting is then be used to enable real-time illumination.

In the initial implementation of the tessellated sphere, various unnecessary vertex data was shader from the CPU to the GPU causing significant performance issues, to alleviate this only necessary vertex data attributes were passed through as such only the vertex position and normal were needed.

Implicit sphere

A further approach made for generating some geometric shapes is using the ray marcher algorithm to generate an implicit sphere, with it then being able to be extended for any other simple geometric shapes and enable the use of the mesh colliders of Unity. This implementation is based on a sphere

tracer in object space as it is more intuitive. The algorithm contained within a shader consists of various steps, but the main presentation of the sphere is based on a 3D SDF.

```
for (uint x=0;x<Max_Step;x++)
{
   sample=pos+direction*step;
   float dist = sphereSDF(sample,0.5f);
   if(dist<Thickness)
   {
       hit=true;
       break;
   }
   step+=dist;

}
if(!hit)
  discard;
```

Figure 60: Ray marching loop within the Pixel shader

Ray marcher algorithm

Through using the sphere tracer ray marcher with a max step of 32, where each iteration step, the equation of each step is represented as:

$$S = p + d * step$$

S represents the current point being sampled along the ray, with p representing the current position, while d represents the direction, and the step represents the current step count. Each iteration step, S is passed into the sphere SDF. A test against how deep the ray is allowed to go is done to determine if there an intersection if so, the iteration stops with the calculation of light with the use of a calculated gradient. To ensure the shape is contained within the bounds of the shape, a depth clipping is done.

As lighting models require surface normals to be able to enable calculation of the surface color at a given point. A such the normals for an implicit surface F(x,y,z) must be found, with an implicit surface, the gradient acts as the normal:$\nabla F(x, y, z)$. And so can be represented as:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

But instead of taking the real derivative of the function, an approximation through sampling points around the point on the surface through ε for the approximation on each axis:

$$\vec{n} = \begin{bmatrix} f(x+\varepsilon, y, z) - f(x-\varepsilon, y, z) \\ f(x, y+\varepsilon, z) - f(x, y-\varepsilon, z) \\ f(x, y, z+\varepsilon) - f(x, y, z-\varepsilon) \end{bmatrix}$$

With the use of the gradient estimation of the surface of the SDF at each point the Phong reflection model can be used enable light to be calculated onto the sphere.

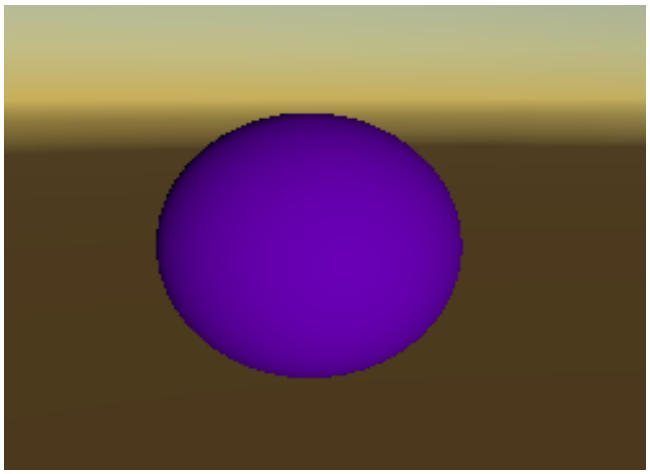


Figure 61: Implicit sphere with a gradient estimation to enable Phong lighting

To enable the implicit objects to be compatible with rasterization and post-processing while also enabling use within the game alongside explicit Unity game objects. The depth buffer was used and to enable depth to be accounted for. This is achieved by finding the distance along each ray to find the position at which the closest mesh-based object lies. With it enabling the ray marcher to exit its loop if it passes this point as it informs it there lies an object in front. With the depth buffer used by an occlusion occurring to enable the SDFs to query and yield a depth value within the depth buffer.

```
float4 tracedClipPos = UnityObjectToClipPos(float4(sample, 1.0));
depth = tracedClipPos.z / tracedClipPos.w;
```

Figure 62: Depth clipping for the depth buffer

Network procedural features

In the approach in the project for adapting the various procedural aspects, the various created shapes except for the maze, they were able to be used across the network for each client. In this, as the meshes of each were created on runtime, to allow them to be functional over the network, each mesh had a network object component attach, which created this functionality.

Game Over

The final scene is triggered through one of the tanks health states becoming dead, as such will cause each client to be taken to the final scene. With the message set on this scene based on the outcome of

their tank's health state which if the state is dead, on the state change, the clients local GameManager boolean variable representing the end game state message to be set.

10 CONSTRAINTS

One of the main constraints encountered in development was how much content and processes could be loaded into the game without bottlenecking the VR as well as not running the immersion by a varying rate of FPS depending on how much content is within the player's area on the loading of the content. So, to ensure a constant set rate FPS, and to enable the most optimal FPS of 72, as stated on the Oculus website, certain practices could be done. One of the better practices is to only have around 50 to 100 calls per frame. Another such idea to keep at the optimal FPS is through the limitation of game objects within each scene along with only using simplistic objects so the overall geometric complexity is low and reduces the number of triangles drawn within a scene. As such to ensure the best performance for the Quest 1 around 350k to 500k triangles should only be contained within a scene.

Another big constraint was time which played a role in various problems encountered. Certain parts in which time played a part, was the initial set up of the project. Another time constraint which played a big part in early development was having to load the project onto the VR to test any aspect implemented which could take around 20 minutes or more depending on how much content was being loaded which if any problems with scripts or objects were encountered the project would not load. Which meant time was needed to find and fix the problem to enable testing. The solution to this was the found with some research mid-way through development in which Oculus provided a way to run Unity from the computer on the VR through their Oculus link which is only a recent development in the last year.

A big constraint encountered, was the testing of the networking functionality. As the chosen architecture of a client hosted model, an assumption was made about MLAPI which enables easy setup of a local server and enable other clients to connect. But for the client hosting model, it requires either a relay server of an additional service or the use of port forwarding. The use of the additional service wasn't wanted as such the latter was chosen. With the creation of the architecture the implementation of the port forward was done but problems with the developer's router, made this unfortunately not possible. As such testing a viable connection was done using another headset when possible. Which when testing the networking functionality regarding the game many assumptions were made about MLAPI along with assumptions that implied on set up of a client's connection to the server, that assumes if a user can carry on through each scene. This is also related to the functions offered by the MLAPI in the scripts where the NetworkStart() function is only called when there is a viable connection.

11 TESTING

In testing the various features, the aim was to test if the with the various implemented aspects it still ensured a smooth FPS or if optimizations were required to improve performance. Other elements were tested with the certain assumptions made regarding if the test was successful. These assumptions are related to the networking functionality, as implied by MLAPI is that if there is a viable connection, then scripts and functionality would occur on each client. As such the testing was done on a local host client in which data packets could still be sent from the client to be received by the server running on the client.

3 test case scenarios were undertaken to determine if all the processes together could still ensure a smooth FPS. The Gunner role and the shoot action seemed like it would provide the best scenarios as instantiating an object can be a somewhat expensive process. So as such it enabled various levels of stress testing to occur. Each scenario there was the creation of the procedural map alongside the parallax shader applied, each scenario consisted of different amounts of action packets being sent to the server. Allowing for a determination to occur in determining if the FPS is smooth with all the information being received alongside if there is any bottleneck occurring and if it is occurring on the CPU or GPU. The Oculus package offers a profiler which provides the ability to see various information about the current game. It is especially useful as it offers how many triangle, vertices and draw calls are occurring alongside information about how the CPU and CPU is handling the information. It also provides the current FPS of the process. Three stress tests were conducted with use of the profiler.

The first stress tests no action packets were sent, with only the procedural map and parallax shader contained within the scene. As such the profiler details that: the triangle and vertices amount are acceptable, but the draw calls are way above the optimal amount. As such the time for rendering per frame is around 23ms which alongside the FPS only being 31.4 indicates there are more optimization improvements which can be made.

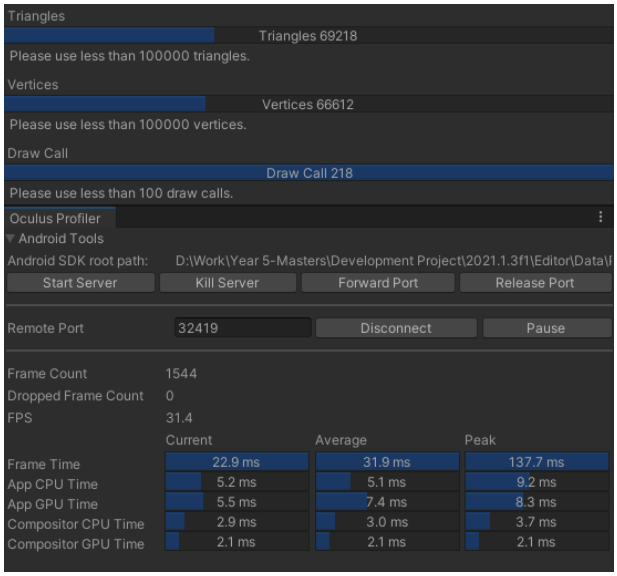


Figure 63: Test case scenario 1 with no action packet sent but with the presence of the procedural maze with the parallax shader attached

2 further test scenarios were done with use of shoot action packet. These scenarios were undertaken to determine which of the bullet objects ensured a smoother FPS alongside optimizing the game. Both test scenarios, included sending 25 shoot action packets, but with only either the implicit sphere created, or the domain shader generated sphere.

For the results of the scenarios, it helped determine that the implicit sphere would be best to allow for the best optimizations as the process is handled fully on the GPU, whereas the domain sphere involved information from the CPU to the GPU each draw call which in conjunction to the maze and attached shader caused extra strain on each. This is evident in the 2 figures below. In which can be noted for the draw call even though both are above the optimal draw call, the implicit sphere still has less calls. Further notes to make is even though both spheres are heavily dependent on the GPU, the app GPU time for the tessellated sphere is significantly higher then the implicit sphere.

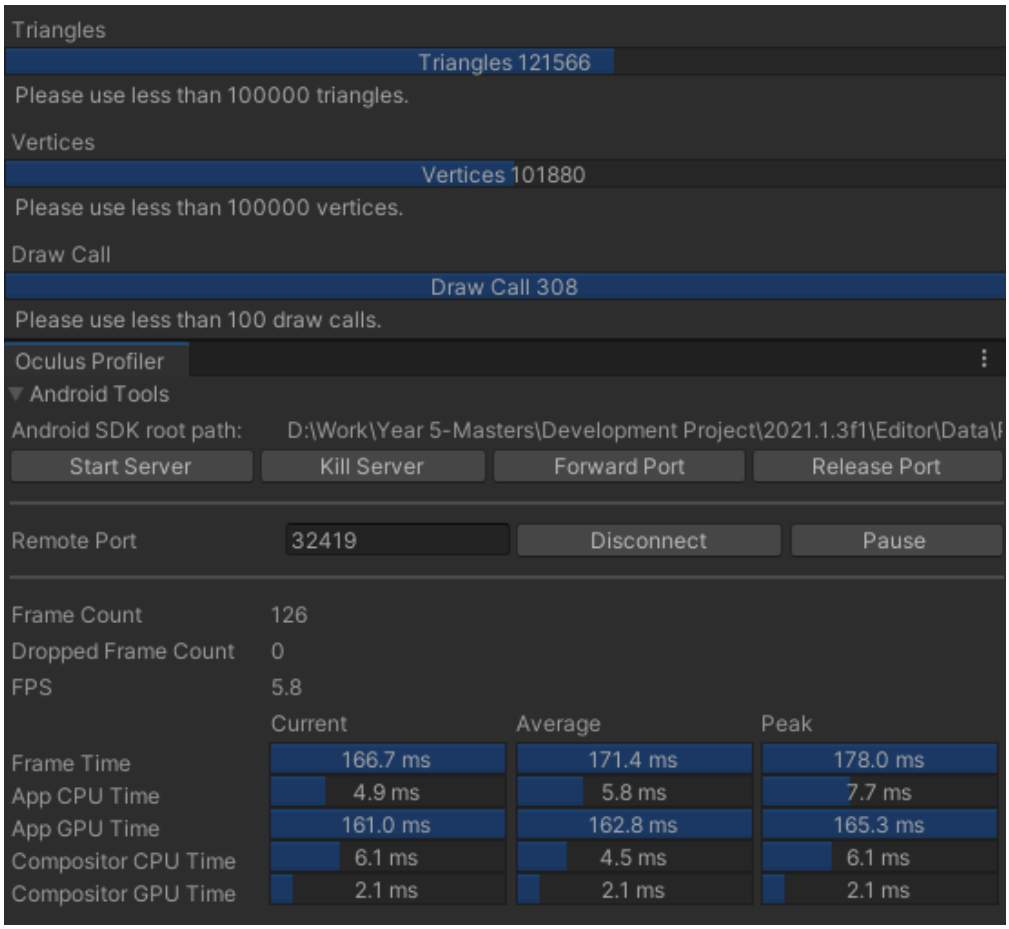


Figure 64: Test case scenario with 25 action shoot, tessellated sphere, packets alongside procedural maze with the parallax shader attached

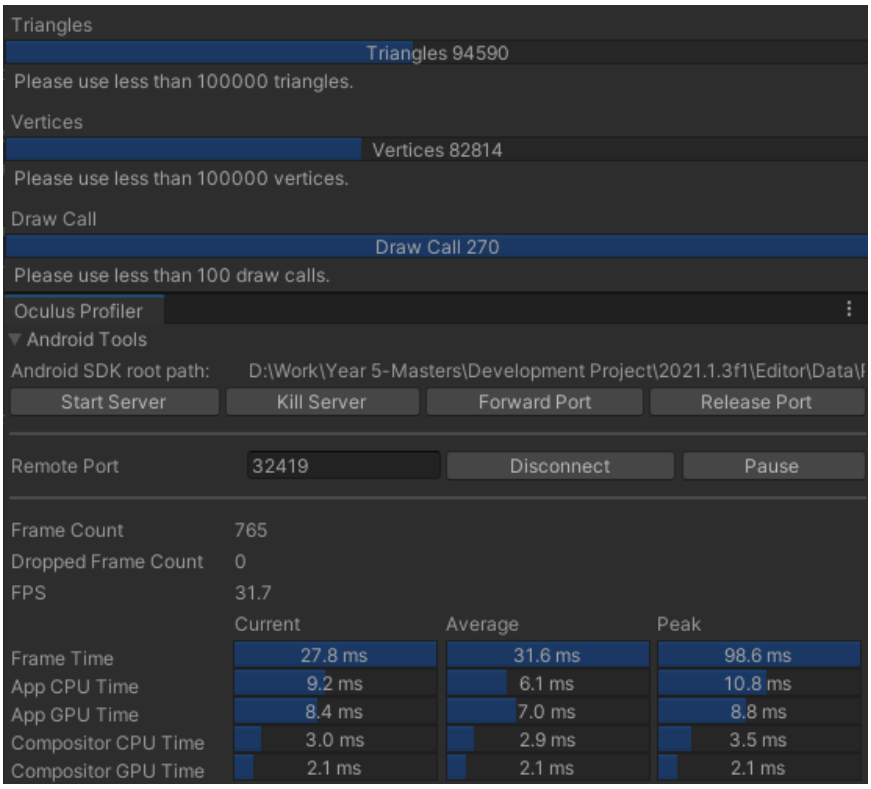


Figure 65: Test case scenario with 25 action shoot, implicit sphere, packets alongside procedural maze with the parallax shader attached

12 EVALUATION

Outlined in the initial plan are 5 primary objectives describing the outcome hoping to be achieved throughout development. As an overall summary, the outcome is far from the initial idea, with various problems encountered, or time spent on functionality which didn't require as much focus such as the networking. But to some extent each objective is fulfilled to a degree, with some being fulfilled to a high level while other only partially.

The first objective outlined in the plan was to develop a VR game to facilitate the other objectives to be achieved. As such the objective was achieved, as various gameplay was created, and then adapted to work within a network environment. In conjunction to a game utilizing VR being developed, there was various interesting game mechanics created. The mechanics are related to each role as each had different mechanics and abilities associated with them such as the driver controlling the movement and speed of their team tank, or the health/energy system employed controlling the ability of each role. As a result, the game can be seen as a VR implementation of the concept discussed in the methodology chapter.

Another objective outlined was the implementation of procedural content within the project. In this objective, it is fulfilled to a high degree as there is evidence of various procedural content deployed within the main game. One example would be the maze, which uses cellular automata as the algorithm basis to produce a randomly seeded maze. With the use various other algorithms to ensure the best map and 3D representation was achieved.

A final point to make regarding this objective is the creation of the UV sphere through tessellation as using this approach enabled the GPU to handle the heavy load of the process, as the Domain shader runs efficiently like a Vertex shader because of the simplistic data flow. A big advantage of using the tessellation pipeline for the triangle amplification process over a Geometry shader or the CPU, is it enabled wasted ALU (arithmetic logic unit) cycles to be removed, as with the Tessellator being a localized element it can act as state machine and requires little input while producing compact output data in the form of an index buffer with a 2D coordinate per outputted vertex. Whereas when a Geometry shader is used, it required ALU cycles alongside buffer space. So, the Domain shader enables the shader Units to require less buffer and can focus on the shading functionality.

Another objective set out was to implement some shaders regarding improving the overall game look and user experience. Within this objective, there were a variety of aspects within it, as it didn't just focus on implementation of ray tracing/ marching, it also had a focus on shaders in terms of a better user experience. One such example is the use of the shader for the procedural maze, a parallax bump map shader which utilized ray marching to further the quality of the object. The parallax shader enhanced user experience and immersion into the game as the use of raymarching with a height map enabled a real time displacement "bump" effect. As the main advantage and virtue of using the parallax effect is the efficiency of it compared to techniques of a similar caliber such as "parallax occlusion mapping" and "relief mapping." These provide a better image quality but at a cost of extra texture lookups, but parallax mapping provides the cheapest real-time technique for displacement mapping.

But there are several disadvantages related to the implemented parallax shader which could be solved with the use of the other techniques. The 2 main problems which can ruin the effectiveness of the

shader are the lack of self-shadowing as well as the assumption, as described in the figure 54, doesn't always constitute to true is that the points A and Bnew will always lie on the surface of the 'real surface'. As such the effectiveness is ruined as there are artifacts which occur at points of a steep height change, which becomes even more evident at points where there is a vertical height change. This is evident in the implemented technique when applied onto the walls, illustrated in figure 66, and not the top of the generated map. With no visible parallax effect alongside being visually unappealing.

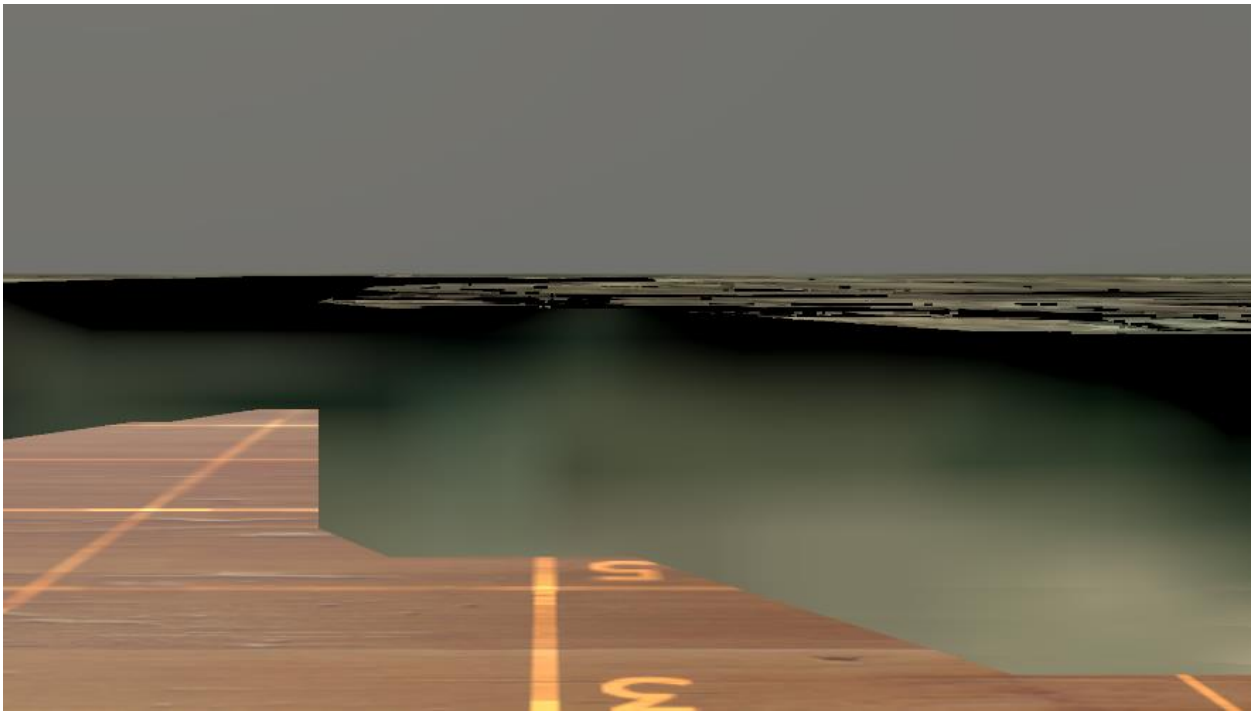


Figure 66: Application of the parallax shader onto the walls of the procedural map

A disadvantage is the self-shadowing and the lack of them, as when a height map is used, certain areas which are higher than others won't cast a shadow over the rest of the height map. But even with the attached problems, the objective can be seen as somewhat fulfilled as the parallax effect still provides a suitable solution which provides artistically pleasing effects but there is further improvement which can be achieved to create a more visually appealing game. Alongside this, with silhouettes caused by the parallax effect, they are always aliased, as MSAA only does so on the edges of geometry and not texture effects.

One of the more time-consuming objectives was the implementation of the networking functionality and adaptation of the created game to support it as such. A functional implementation was achieved as there is the ability for each client to connect to the game, and set up their role and team, and then enter the main game level alongside other clients. With each client also being able to send serializable actions to the server and whilst also receiving various information about what is occurring in game.

Within the implementation certain problems were encountered with MLAPI, in context of initial player spawning and placement, as such the gameplay approach was reevaluated. So instead of the player following around its tank with its position updated based on its tank movement. The objects for a team are held in static positions above the level, with on the initial spawn, the player would be designated the

objects related to their chosen role. An example of this would be through the client which selected the Gunner role. The client would have the Gunner component class attached to its local player, with the button relating to the Gunner actions within the scene allocated to the button contained in the Gunner component. Which enabled a button click event to be attached and provides the client the ability to send an action request to the server filled out with the action information, in this case the shoot action. And enable the action to occur on the server.

Even though the focus of the project was not on the networking implementation, more time was spent on trying to create a functional networking system with use of MLAPI as such reduced the time for implementation of other project features as such the implementation of more procedural content was taken away. But an outcome of this is a deep understanding of the MLAPI functionality enabling future developments with its use while alleviating some of the misunderstandings found in conducting this project.

A final objective set was to adapt the procedural content to be supported over the network such that each client would see the same outcome in the content. There are two aspects in relation to this objective and the procedural content. One aspect is related to the two procedurally generated spheres and the procedural maze the other one. For the spheres, as these are created objects and are used as bullets which are stored in an action array, as such they just required the NetworkObject alongside, functionality to support the objects to work on both the client and server. As such each object is made up of Network, Server, and Client component to facilitate the networking of the object. So as such can be seen as fulfilling this objective. But in respect to the maze, the object could be set up to be networked as a NetworkObject component could be attached but the outcome of the maze regarding cellular automata over a network was unfortunately not implemented. So as an objective it can be seen as somewhat complete, with an understanding gained from the research done around how various games apply networking to their procedural content, as such can be further investigated for future development of the project.

Overall, for the project, there were various aspects which required a good depth and understanding, with regarding each objective a lot of research was done, which enabled understanding for future accomplishments in the game to produce a fully-fledged game rather than just a prototype. But as a final product, the result is unsatisfactory, as a lot of time seems to be used in research rather than development, which can be seen in the outcome. As there is a lot of improvement to be made, with the focus of the project seemingly more of integration and research.

13 CONCLUSION

To conclude the project, the primary question this paper aimed to answer during the different stages of research and development, “What is the feasibility of integrating modern gaming techniques of Ray Tracing and Procedural Generation into a Virtual Reality Networked Game while maintaining a smooth user game experience?”. To answer the question based on the research and implementation, it can be answered yes, it is possible to some degree. As discussed, a huge pitfall encountered was time as well as testing in the VR environment. There were various avenues involved in the development which involved substantial time for research and understanding, as such when an aspect took longer to understand and implement such as the networking, it affected the rest of the project. Even though each objective was somewhat achieved, the project was too ambitious for what was wanting to be achieved.

In the work done throughout development of the project, it has helped achieve a variety of accomplishments. One such accomplishment is the understanding of using a VR system in conjunction with a networking system to create a multiplayer game experience. Along with this it helped in understanding how the various APIs and systems of Unity can also be used to support a VR multiplayer experience. A further point to make regarding ray tracing, as there is a variety of ways in which it can be achieved without full modern RTX support, but this comes into troubles depending on the number of objects involved in this process. Or with use of a shader which employs a ray marcher alongside utilizing Unity inherent shader values such as light, but this does not consider, objects around so reflection or refraction cannot be achieved.

The overall project was based on the project plan, which helped define the project aims and objectives. But with the project plan, each objective didn’t give a full concise description of the best approach to achieve that objective. As such created certain problems arose when a decision was needed in deciding the best way to accomplish the objective.

But even with these problems, the plan enabled for an initial starting point, and gave a cohesive plan outline such that the project moved in an understandable direction while giving leeway for learning and an in-depth understanding of the various aspects outlined related to VR game creation with the networking, procedural generation, and ray tracing through a game development process. Which enabled the appropriate solutions to be found to satisfy each objective. The plan and methodology outlined in the methodology chapter, especially the methodology, had a big influence in implementation as it gave a structure to allow for a steady process of implementation. The project plan can be found in the appendix B, with it outlining the various steps which influenced the methodology alongside a simple plan for the various objectives. This enabled for easy comparison of goals and to determine if each are on track. While providing a base game to be made and further adaptation to satisfy the various alongside supplementing the ability for more sophisticated features to be added.

One aspect of the initial scope of the project, which was researched, and certain implementations were tried but full utilization of it within the project never really came to fruition. The idea for this was to visually enhance the game through ray tracing and create a more immersive game and enjoyable experience for a player. There were parts which utilized this aspect such as the parallax bump map which used a ray march algorithm to visual improve the shader and another shader which produced a ray marcher sphere object.

With the overall outcome of the project a prototype which can be built on further in the future. But in the outcome of the project, the developed game could have been vastly improved with better time planning and more concise decisions for each objective alongside. Furthermore, each objective could have been more defined to enable a smoother approach and enable less time needed in researching to find a suitable solution to satisfy each objective.

14 FUTURE WORK

As seen in the testing chapter, there is still more optimizations which need to be made to ensure the most optimal FPS throughout each scene. One such way is through reducing the number of vertex attributes being submitted to the vertex shader as this can help in performance gain. As to ensure full precision for shaders only POSITION is required as such if extra attributes are required, these could be compressed into other channels for half precision.

The use of a compute shader to parallelize the mesh generation process on the GPU or offered within Unity there is the ability to remove the data being copied from the CPU Ram to the GPU vRam or back. As Unity provides the ability to render the data directly from the GPU without needing to copy it. The functionality is known as Graphics.DrawProcedural and enables rendering to occur without the need for GameObjects. The process of the DrawProcedural is through a compute shader which only requires the information related to the current vertex index. As such the unnecessary data is not created and never exists on the ram.

A point to make regarding further possibilities of procedural content, is the use of a procedural terrain in conjunction to the procedural map, which with use of varying heights for the terrain and in the context of the game could create new gameplay dynamics and help create a more unique gameplay experience in comparison to other VR tank games on the market.

A further way to improve the game experience is through mesh deformation as such when there a point of impact such as on the maze, the vertices involved in the impact can be displaced alongside applying a texture to illustrate the mesh deformation and amplify the effect.

To improve the Parallax Mapping, the use of a Parallax Occlusion Mapping, which takes the notion of the Steep Mapping approach but improves on it through linearly interpolating between each depth layer for before and after the collision.

Further improvements to the game are to use marching squares algorithm in conjunction to the SDFs to enable an isosurface extraction. As it will enable the isosurface to extract triangle meshes from the SDFs which with use of CSG operators, discussed in background, can extract complex shapes made from the use of the CSG operations. Through this, more of the generated objects can be created implicitly and alleviate generating the triangles on the CPU as with the use of a compute shader the work can be achieved through parallelizing the process on the GPU. Furthermore, it will provide the ability for meshes to be morphed and manipulated with further application for terrain manipulation such as in No Man's Sky.

To enable raytracing compatible with Unity, a proposed way based off the implementation outlined in the paper by Turner Whittard. Using a compute shader rendering to a render texture, the ability to use

the various intersection algorithms alongside enabling explicit and implicit objects to simulate the ray tracing together to achieve the light simulation and enable shadows, refraction, and reflection alongside other ray tracing features. To enable explicit objects to be used within the shader, compute buffers can be used to pass the mesh object information including triangles to the compute shader and enable the ray tracing loop to occur. A triangle intersection algorithm can then be used to determine if there any ray interactions between any mesh object within the scene. But there is a lot of considerations to made because of the intensity of ray tracing on a GPU, it recalls how intense the process could be on VR where the hardware is not as capable.

A final point to be made regarding optimizing the game to its full potential and reducing the strain of CPU to the bare minimum. An idea for this optimization can be achieved by the implementation of an Entity Component System (ECS). The system will enable code to be compiled faster within the engine alongside other aspects. In turn the GPU will be able to render faster as similar types are cached and packed together at compilation. Whereas in the Unity framework even though similar types a cached together, it caches the data across non-contiguous memory, and does so in a more randomized manner. Which can cause the RAM to become slower if too many objects are used.

With the implementation of the ECS framework, the entities are the parts which make up the game and environment, with each entity made up of its own set of components, which control the behavior of each entity and the systems control these components. This approach is very similar to how Unity works, but the data caching on the CPU is more randomized. With OOP, the compiling is more optimized but with ECS it enables entities with a particular component attached to be easily manipulated through using the system controlling that component. This enables it to be more efficient as it uses a data-oriented compilation system approach. The difference in the optimization of each approach is illustrated in figure 67.

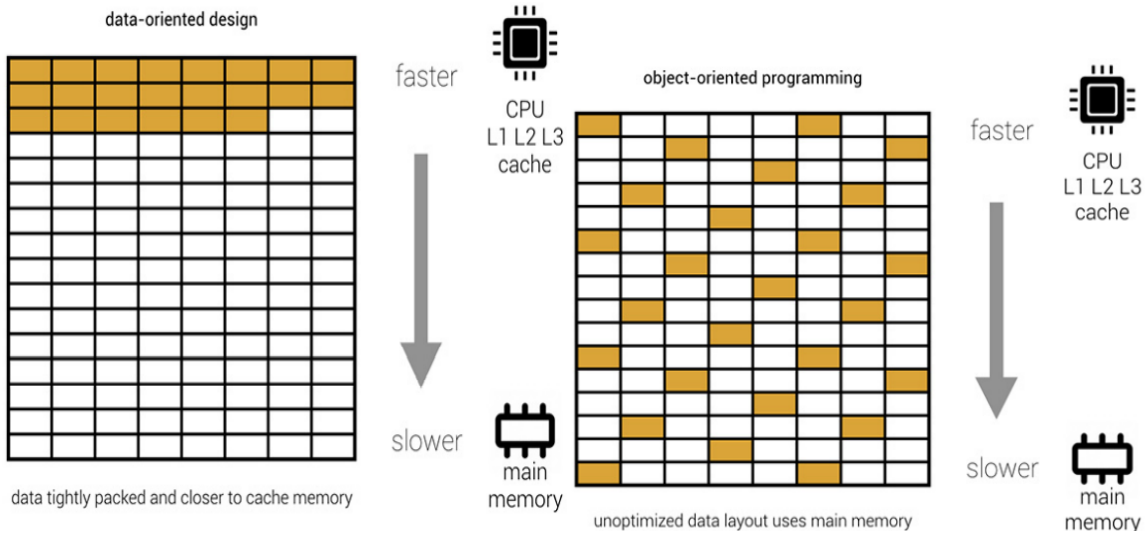


Figure 67: Caching of data (Lin, 2020)

15 APPENDIX A: OCULUS QUEST AND QUEST 2 SPEC COMPARISON

Device	Oculus Quest 2	Oculus Quest
Starting Price	$299	$399
Pixels per eye	1832 x 1920	1440 x 1600
Screen refresh rate	72Hz at launch, 90Hz to come	72Hz
Weight	503 grams	571 grams
Tracking	Internal cameras	Internal cameras
Battery capacity	Two to three hours	Two to three hours
Processor	Qualcomm Snapdragon XR2	Qualcomm Snapdragon 835
RAM	6GB	4GB
Storage	64GB or 256GB	64GB or 128GB

Figure 68- Comparison of the Oculus Quest and Quest 2 specs (Forbes, 2019)

16 APPENDIX B: MLAPI

MLAPI offers a variety of components which build the framework functionality. Such components offered are the Connection approval, ability to limit max players allowed on the server, NetworkObject, NetworkBehaviour, Modularity, NetworkVariable along with the ability to Read and write to and from a buffer at high speeds. Further offerings are the ability to have a Physics system running along with custom Transports which can be written.

The Netcode for all the GameObjects high level components and which the RPC system relies upon is the NetworkObject and NetworkBehaviour components. Each object which contains a NetworkObject is provided functionality by it alongside providing the networking components functionality.

On runtime each object which has a NetworkObject is assigned a NetworkId which is associated to 2 objects across the Network. So, allows for RPC calls to be made to specific objects across the network and the call to be replicated on each client. It also offers the ability to specify ownership of the objects so that only an owner of the object can manipulate it.

NETWORKBEHAVIOUR

The most useful component offered within the MLAPI framework is the NetworkBehaviour class. The class is abstract and derives from the base Unity scripting class of MonoBehaviour. The NetworkObject attached to each object owns each NetworkBehaviour attached to the object. The main idea behind the NetworkBehaviours is to contain any RPC methods and Network Variables. The functionality of the RPCs of the component works through a message sent with the params of the object along with the NetworkId of the object and the index of the NetworkBehaviour. NetworkBehaviours also offer the ability to synchronize the state of the same objects across the network using NetworkVariables and RPCs.

NETWORKING TIME: SERVER AND LOCAL TIME

In the context of Network Timing and ticks, MLAPI offers two values one for ServerTime and LocalTime. This is because MLAPI Netcode for Gameobjects utilizes a star topology (Unity,2021), such that clients only communicate with the Server/Host and never between each other. As such there is a time delay in messages being transmitted and received over the server. Which outlines which RPCs or NetworkVariables don't occur immediately on the other clients. As such NetworkTime allows time for consideration of transmission delays. (Unity,2021)

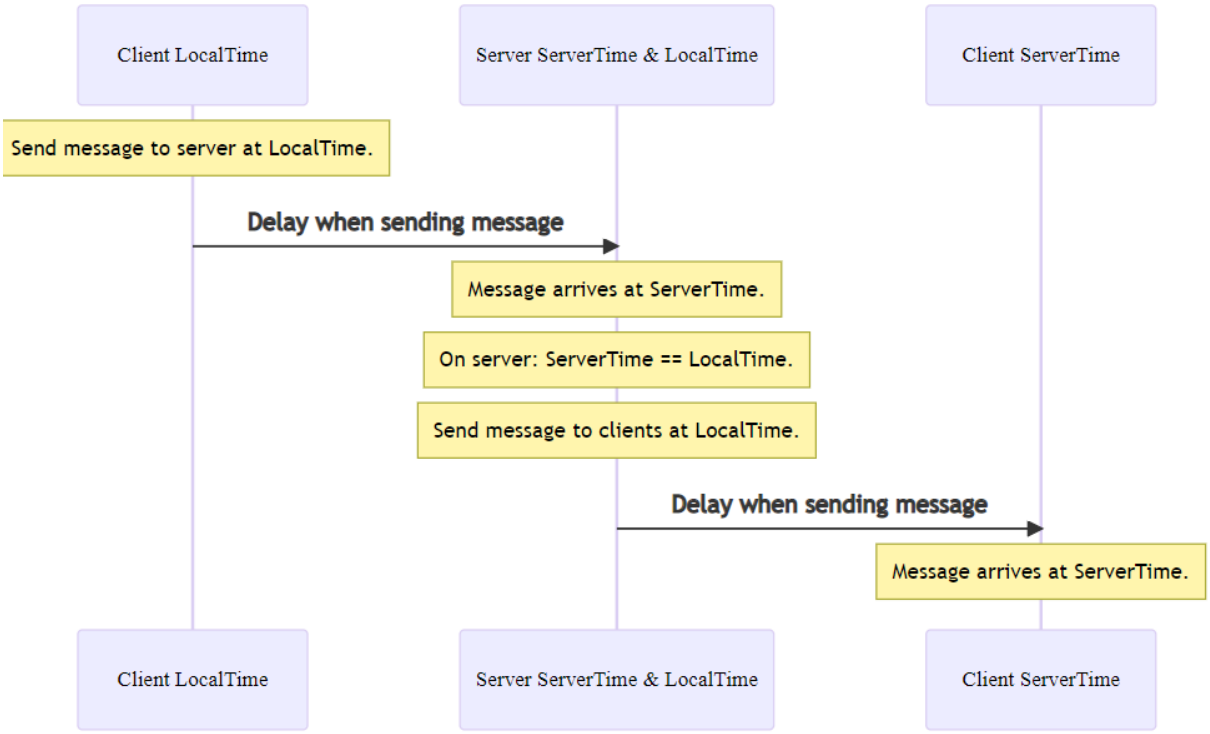


Figure 69: The process of sending messages across the Network with the use of the different Time variables. (Unity, 2021)

NETWORKING COMPONENTS

The main networking components which create the functionality for the Network are the NetworkManager, NetworkTransform and NetworkAnimator.

NetworkManager

The main component which handles all the aspects related to the setting of the Network, any prefabs to be initiated on runtime on the Network as well as each scene name within the project. It also requires a transport component as this is responsible for IP addresses and additional settings.

It provides the functions to start a server along with either hosting a game or joining as a client. The host acts as a client and server simultaneously. For creating a connection and connecting to a game, the NetworkManager uses the IP and Port specified in the Transport component to route to the game or create the connection.

NetworkTransform

The main component which handles the synchronization of the various transforms of an object at runtime is the NetworkTransform as it provides the functionality to synchronize server objects onto a client. It furthers this synchronization through the ability to enable interpolation of objects. This allows for any transform changes to be smoothly interpolated onto each client along with offering interpolation buffers for incoming data with a little delay which helps produce additional smoothing on the transform values which in turn creates a smoother transform synchronization.

Messaging system

Further aspects offered in the MLAPI architecture are a few key areas which provide easy development of various networked game features through the intricate messaging system. One such area offered is Remote procedure calls (RPCs). There are 2 types of RPCs within the network system, one is ServerRpc which are calls from a client to run on the server and ClientRpc calls that are calls from the server and run on the clients.

As stated on the Unity multiplayer Website, RPCs are very reliable so there is a guarantee, that code will be executed on each client but this won't always provide what is needed as reliability may not be wanted for certain events which aren’t critical such as sound effects or particle effects.

ServerRPC

The idea behind ServerRPC are calls made by the player object on a client to be relayed onto the server which in turn relays these to the other players. To increase security with these calls, commands can only be sent by the client’s own player object which stops other players controlling any other client’s player object. With the ServerRpc, a client invokes the method to be run server side.

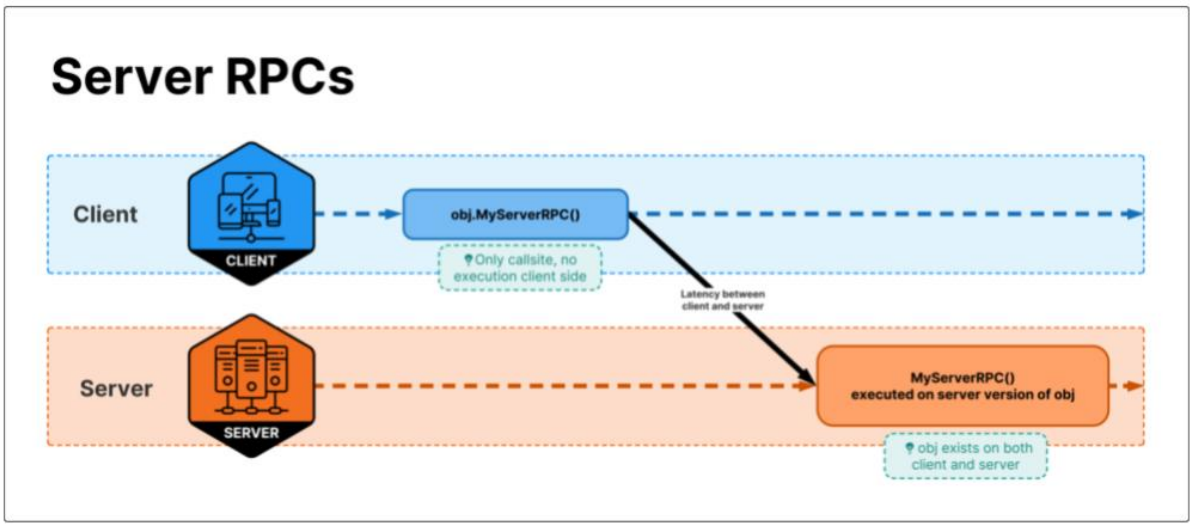


Figure 70: How a ServerRpc is invoked by a Client onto a Server.

With the MLAPI architecture it also offers the chance for Client Hosts to control ServerRpc calls from client as well as calls from the client Host to be relayed onto the local client's game.

ClientRPC

To create the functionality to call a ClientRPC, the server will call the ClientRPC which in turn relays call to the other players.

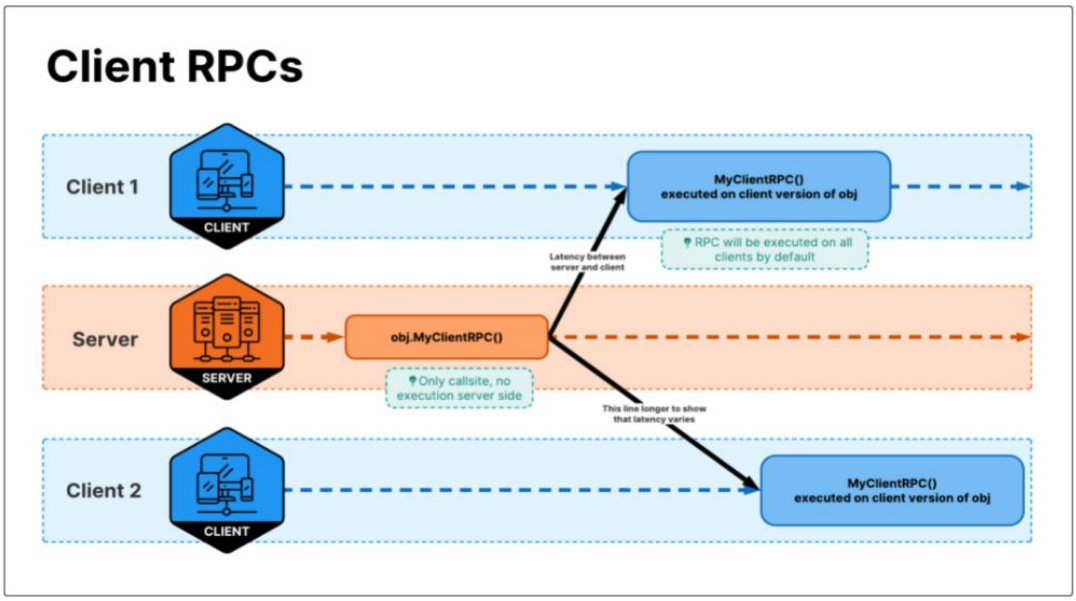


Figure 71: How a ClientRpc call is invoked onto each client by the server. (Unity,2020)

Custom Messaging

As stated before, it also offers a way to send custom messages across the messaging system, which is done by use of a thin layer referred to as “Custom Messages”. This allows for implementation of game specific behaviors and custom targeting. The messages can be created in 2 forms named and unnamed. Unnamed messages offer the ability to create a single sending of sorts channel which is useful for a game specific custom messaging system. Named messages use the messaging system already offered by MLAPI.

Sterilization

MLAPI offers custom serialization built-in support which supports native C# types as well as Unity primitive types, along with offerings the ability to further extend serializable for the network for User defined types through the interface INetworkSerializable.

RPC vs NetworkVariables

RPC vs NetworkVaraibles bring into question which way is best for syncing the various objects across the system as if the wrong data syncing is used bugs can occur and the code can become more complex than needed. As both offer the ability to send messages over the network, the logic and design of the messages is what will decide which is best for each network message. RPCs are best for one off transient event where the information is only needed for the point of receiving whereas Network Variables are useful for information which needs to persist over time. (Unity,2021)

NetworkUpdate loop

Network update loop refers to updating of the various networking systems outside the usual Unity Monobehaviour cycle as such the Network Update loop offers infrastructure for this such thing. It works through use of Unity’s low level Player Loop API and offers the ability to register and update the NetworkSystems using the NetworkUpdate methods which can be then executed at specific NetworkUpdateStages that can occur before or after the Monobehaviour logic execution. The running of the Network look is detailed below:

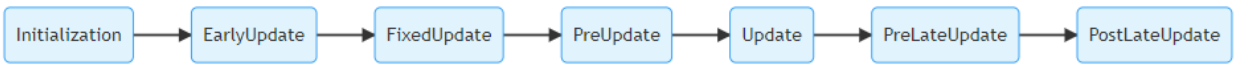


Figure 72: MLAPI NetworkUpdate loop (Unity,2020)

APPENDIX C: RISK ANALYSIS

Risk	Alleviation	Likelihood	Severity	Impact
An injury occurring while the VR is in use	While in use, a marked-out space will be made available, with all obstacles removed to remove the chance of injury. The Quest offers built in functionality which allows users to set the area of safe play. Which informs a player when they go out of the playing area bound through the user being shown their real-world surroundings.	2	2	2*2 = 4
Health related problems from use of the VR such as a headache	A use of a warning message and timer informing a player they require a break.	2	1	2*1=2
Data corruption/ Computer problems	With the use of a cloud storage, such as git, the project is saved online alongside enabling it	2	2	2*2=4

	to be available on multiple computers			
Software compatibility failure	Ensure there is a copy of when the software worked together so it can be easily reloaded.	2	3	2*3=6
Problems with the VR headset	The use of HIVE in University to enable use of other VR headsets	1	4	1*4=4
Requirements are not fulfilled	Create an efficient plan which enables proper time management with a revaluation of the objectives and depth.	2	2	2*2=4
Unexpected difficulties in development	Ensure in the plan there is time allowed for difficulties encountered so they can be dealt with at the time or later.	1	1	1*1=1
Illness and mental illness	Ensure a healthy diet along with time allowed for exercise and recreation.	1	2	1*2=2

APPENDIX D: PROJECT PLAN

#	Task Name	Description	Duration (Weeks)
1	Develop a VR tank game	A simple VR game with various functionality	2
2	Implement procedural content	An in-depth study of various procedural algorithms. Some considerations for mesh representation will be done alongside	5
3	Implement shaders for textures and Ray tracing	Creation of various shaders for use as textures or objects within the game	5
4	Implement, integrate, and adapt the game to support Networking	An in-depth study of various networking libraries with considerations for the protocol and architecture wanted.	3
5	Adapt the procedural content to have networking functionality	Ensure the created procedural content is supported over the Networking.	2
6	Final Report	Improve and finish the dissertation	4

17 REFERENCES

Ahn, S., 2021. OpenGL Sphere. [online] Songho.ca. Available at: https://www.songho.ca/opengl/gl_sphere.html [Accessed 20 January 2022].

Anderson, B., 2015. https://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html. [image].

Appel, A., 1968. Some techniques for shading machine renderings of solids. [online] New York. Available at: http://graphics.stanford.edu/courses/Appel.pdf [Accessed 15 August 2021].

Arvo, J., 1986. Backward Ray Tracing, Chelmsford MA: s.n.

Byford, S., 2021. Almost a fifth of Facebook employees are now working on VR and AR: report. [online] The Verge. Available at: https://www.theverge.com/2021/3/12/22326875/facebook-reality-labs-ar-vr-headcount-report [Accessed 6 January 2022].

Cajaraville, O., 2016. Four Ways to Create a Mesh for a Sphere. [online] Medium. Available at: https://medium.com/@oscarsc/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4 [Accessed 20 January 2022].

Cook, M., 2013. Generate Random Cave Levels Using Cellular Automata. [online] Game Development Envato Tuts+. Available at: https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664 [Accessed 9 December 2021].

Docs.blender.org. 2021. Denoising — Blender Manual. [online] Available at: https://docs.blender.org/manual/en/latest/render/layers/denoising.html [Accessed 22 January 2022].

Doull, A., 2011. Rain Drop Algorithm - Procedural Content Generation Wiki. [online] Pcg.wikidot.com. Available at: http://pcg.wikidot.com/pcg-algorithm:rain-drop-algorithm [Accessed 6 January 2022].

Dormans, J., 2018. A Handcrafted Feel: 'Unexplored' Explores Cyclic Dungeon Generation. [online] CONTROL500. Available at: http://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation [Accessed 6 January 2022].

Dürer, A., 1525. Painter's Manual: A Manual of Measurement of Lines, Areas, and Solids by means of Compass and Ruler. [image] Available at: https://blogs.unimelb.edu.au/librarycollections/2017/08/13/finding-durers-perspective [Accessed 23 January 2022].

Flick, J., 2017. Shading with Tessellation. [image] Available at: https://catlikecoding.com/unity/tutorials/advanced-rendering/tessellation [Accessed 23 January 2022].

Giurgiu, M., 2020. Raytracing with RTX. MSc. University of Hull.

Glazer, J. and Madhav, S., 2015. Multiplayer Game Programming. [The Addison-Wesley]: Addison-Wesley Professional.

HART, J. C. 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. The Visual Computer 12, 10, 527–545. 5

Hofmann, G., 1990. Who invented ray tracing?. The Visual Computer 6,. pp.120–124.

Hoffman, C. and Hopcroft, J., 1985. Automatic surface generation in computer aided design. [online] pp.92–100. Available at: https://www.semanticscholar.org/paper/Automatic-surface-generation-in-computer-aided-Hoffmann-Hopcroft/bb1607d2c0ed6d066887b1dbdf1e32b001b20ee4 [Accessed 20 January 2022].

House, B., 2020. High level guide to evaluate Unity networking solutions. [image] Available at: https://blog.unity.com/technology/choosing-the-right-netcode-for-your-game [Accessed 24 January 2022].

Hsin-KaiWu Silvia Wen-YuLee Hsin-YiChang Jyh-ChongLiang, 2013. Current status, opportunities, and challenges of augmented reality in education. [online] Available at: https://www.researchgate.net/publication/235703112_Current_status_opportunities_and_challenges_of_augmented_reality_in_education [Accessed 11 November 2021].

Hunt, C., 2018. Take a crash course on VRChat, the social app that's heating up. [online] Windows Central. Available at: https://www.windowscentral.com/beginners-guide-vrchat [Accessed 6 January 2022].

Kajiya, J. T., 1986. The rendering equation. SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques, 20(4), pp. 143-150..

Kastbauer, D., 2010. The Next Big Steps In Game Sound Design. [online] Game Developer. Available at: https://www.gamedeveloper.com/audio/the-next-big-steps-in-game-sound-design [Accessed 17 September 2021].

Kelly, M., 2022. Club Penguin Online shuts down after receiving copyright claim from Disney. [online] The Verge. Available at: https://www.theverge.com/2020/5/15/21260122/club-penguin-dmca-disney-takedown-cponline-online [Accessed 6 January 2022].

Kionary, 2019. Procedural Content in Video Games. [online] Medium. Available at: https://kionay.medium.com/procedural-content-in-video-games-2134c8ff7779 [Accessed 21 November 2021].

Kumparak, G., 2014. *TechCrunch is part of the Yahoo family of brands*. [online] Techcrunch.com. Available at: https://techcrunch.com/2014/03/26/a-brief-history-of-oculus/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAABoBq7TxDIlvMPqyWUjW8f7pyr4ePPIbQs0vs91nvYGwY79tUmokjG0C_DOA7MyrgTnqSLCkhHvX88tE3Fzcj9pajLps2Rdpxl_myS8XkCdmr7dAiM5NXQDxgSTISa7DIyiO0U8wpCei7Dkn54TguUTCPazaePSHbS4P2I968yxb [Accessed 7 November 2021].

Kurachi, N., 2011. The Magic of Computer Graphics. s.l.:A K Peters/CRC Press..

Lang, B., 2013. An Introduction to Positional Tracking and Degrees of Freedom (DOF). [online] Road to VR. Available at: https://www.roadtovr.com/introduction-positional-tracking-degrees-freedom-dof [Accessed 23 January 2022].

Lague, S., 2016. Procedural Cave Generation- Flood fill algorithm. [Source code] Available at: https://github.com/SebLague/Procedural-Cave-Generation/blob/master/Episode%2009/MeshGeneration.cs [Accessed 23 January 2022].

Lague, S., 2016. Procedural Cave Generation- Marching Squares algorithm. [Source code] Available at: https://github.com/SebLague/Procedural-Cave-Generation/blob/master/Episode%2009/MapGenerator.cs [Accessed 23 January 2022].

Luecking, S., 2013. Durer, Drawwing, and Digital Thinking - Steve Luecking. [online] Brian-curtis.com. Available at: http://www.brian-curtis.com/text/conferpape_steveluecking.html [Accessed 17 August 2021].

Lin, W., 2020. Entity Component System for Unity: Getting Started. [online] raywenderlich.com. Available at: https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started [Accessed 20 January 2022].

Lynch, G., 2021. Oculus Quest 2 review. [online] TechRadar. Available at: https://www.techradar.com/uk/reviews/oculus-quest-2-review [Accessed 10 January 2022].

Maturana, J., 2009. File:Evolutionary Algorithm.svg - Wikimedia Commons. [online] Commons.wikimedia.org. Available at: https://commons.wikimedia.org/wiki/File:Evolutionary_Algorithm.svg [Accessed 6 December 2021].

Microsoft, 2020. Hull shader stage: input and output. [image] Available at: https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/hull-shader-stage--hs- [Accessed 23 January 2022].

Milgram, Paul & Kishino, Fumio. (1994). A Taxonomy of Mixed Reality Visual Displays. IEICE Trans. Information Systems. vol. E77-D, no. 12. 1321-1329.

Milmo, D., 2021. https://www.theguardian.com/technology/2021/oct/28/facebook-mark-zuckerberg-meta-metaverse. Guardian, [online] Available at: https://www.theguardian.com/technology/2021/oct/28/facebook-mark-zuckerberg-meta-metaverse [Accessed 18 November 2021].

Mirror-networking.gitbook.io. 2022. General - Mirror. [online] Available at: https://mirror-networking.gitbook.io/docs/general [Accessed 24 January 2022].

MLAPI-Docs-multiplayer.unity3d.com. 2021. MLAPI- Unity Multiplayer Networking. [online] Available at: https://docs-multiplayer.unity3d.com/blog/tags/mlapi/index.html [Accessed 24 January 2022].

Mount, D., 2020. Delaunay Triangulations: General Properties. [ebook] Available at: https://www.cs.umd.edu/class/spring2020/cmsc754/Lects/lect12-delaun-prop.pdf [Accessed 20 January 2022].

Mtx, M, 2013. Pixar's Lightspeed Brings New Light to Monsters University. [online] Thisanimatedlife.blogspot.com. Available at: https://thisanimatedlife.blogspot.com/2013/05/pixars-chris-horne-sheds-new-light-on.html [Accessed 2 October 2021].

Pesce, M., 2015. Oculus Rift Crescent Bay Prototype. [image] Available at: https://www.flickr.com/photos/pestoverde/16568187562 [Accessed 10 January 2022].

Pharr, M., 2006. GPU gems 2. Upper Saddle River: Addison-Wesley.

Phong, B. T. (1975). Illumination for Computer Generated Pictures. Available online: https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf [Accessed 03/01/2022]

Photonengine.com. 2020. Photon Unity 3D Networking Framework. [online] Available at: https://www.photonengine.com/en/PUN [Accessed 24 January 2022].

Pixar Animation Studios, 2006. Ray Tracing for the Movie 'Cars'. [online] Available at: https://graphics.pixar.com/library/RayTracingCars/paper.pdf [Accessed 22 January 2022].

Quilez, I., 2008. Inigo Quilez. [online] Iquilezles.org. Available at: https://iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm [Accessed 20 January 2022].

Quilez, I., 2008. Inigo Quilez. [online] Iquilezles.org. Available at: https://iquilezles.org/www/articles/distfunctions/distfunctions.htm [Accessed 20 January 2022].

Robertson, B., 2009. Food for Laughs | Computer Graphics World. [online] Cgw.com. Available at: https://www.cgw.com/Publications/CGW/2009/Volume-32-Issue-9-Sep-2009-/Food-for-Laughs.aspx [Accessed 10 October 2021].

Robertson, C., 2020. Oculus Quest vs. Oculus Quest 2: what's the difference?. [online] The Verge. Available at: https://www.theverge.com/21433030/oculus-quest-2-vr-headset-specs-comparison-htc-valve-microsoft [Accessed 19 November 2021].

Roguebasin.com. 2020. Basic BSP Dungeon generation - RogueBasin. [online] Available at: http://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation [Accessed 20 January 2022].

Roguebasin.com. 2016. Cellular Automata Method for Generating Random Cave-Like Levels - RogueBasin. [online] Available at: http://www.roguebasin.com/index.php/Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels [Accessed 20 January 2022].

Royal, M., 2018. Communication architecture between Unity Android Bridge and Android SDK Bridge. [image] Available at: https://unitylist.com/p/i3x/Geomapping-with-Unity-Mapbox [Accessed 21 January 2022].

Shaker, N., Togelius, J. and Nelson, M., 2016. Procedural Content Generation in Games. pp. 18, 33, 38, 42

Shirley, P. and Morley, R., 2003. Realistic ray tracing. Natick, Mass.: AK Peters.

Schueffel, P., 2017. The Concise Fintech Compendium.. Fribourg.

Sharwood, S., 2021. South Korea creates 'metaverse alliance' to build an open national VR platform. [online] Theregister.com. Available at: https://www.theregister.com/2021/05/18/south_korea_metaverse_alliance [Accessed 21 November 2021].

Turner, T., 1980. An Improved Illumination Model for Shaded Display. [online] New Jersey. Available at: https://www.cs.drexel.edu/~david/Classes/Papers/Whitted80.pdf [Accessed 4 November 2021].

Unity Tecnologies, 2020. Unity - Manual: The Shader class. [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/shader-objects.html [Accessed 21 January 2022].

Unity Technologies, 2021. Unity Network loop. [image] Available at: https://docs-multiplayer.unity3d.com/docs/advanced-topics/network-update-loop-system/about-network-update-loop/index.html [Accessed 23 January 2022].

Valdetaro, A., Nunes, G., Raposo, A. and Feijó, B., 2010. Understanding Shader Model 5.0 with DirectX 11 - PDF Free Download. [online] Docplayer.net. Available at: http://docplayer.net/33071925-Understanding-shader-model-5-0-with-directx-11.html [Accessed 22 January 2022].

Valtchanov, V. and Brown, J., 2012. Evolving dungeon crawler levels with relative placement. [online] Available at: https://dl.acm.org/doi/10.1145/2347583.2347587 [Accessed 6 January 2022].

Vilchez, J., 2020. SURFACE DESCRIPTION AND EXTRACTION FROM IMAGING-BASED DATA FOR CONCRETE MATERIALS. Masters. Aachen University.

Warren, T., 2021. Microsoft Teams enters the metaverse race with 3D avatars and immersive meetings. [online] The Verge. Available at: https://www.theverge.com/2021/11/2/22758974/microsoft-teams-metaverse-mesh-3d-avatars-meetings-features [Accessed 6 December 2021].

WhitePaper, 2012. New UV Coordinate. [image] Available at: http://imgtec.eetrend.com/sites/imgtec.eetrend.com/files/download/201402/1514-2181-1480-2147-parallaxbumpmappingwhitepaper.pdf [Accessed 24 January 2022].

Wikipedia Commons, 2015. 6DOF. [image] Available at: https://commons.wikimedia.org/wiki/File:6DOF.svg [Accessed 23 January 2022].

WikimediaCommons, 2017. Ray Tracing Illustration First Bounce. [image] Available at: https://commons.wikimedia.org/wiki/File:Ray_Tracing_Illustration_First_Bounce.png [Accessed 23 January 2022].

Wilson, M., 2015. How 4 Designers Built A Game With 18.4 Quintillion Unique Planets. [online] Fast Company. Available at: https://www.fastcompany.com/3048667/how-4-designers-built-a-game-with-184-quintillion-unique-planets [Accessed 13 October 2021].

Wood, 2020. When Facebook bought Oculus did it make the VR firm less relevant? - Marketplace. [online] Marketplace. Available at: https://www.marketplace.org/shows/marketplace-tech/when-facebook-bought-oculus-did-it-make-the-vr-firm-less-relevant [Accessed 8 August 2021].

Xu, G., 2015. High-order Extraction Method of Iso-curvature Lines on B-spline Surfaces. Journal of Information and Computational Science, 12(10), pp.3945-3952.

Yahyavi, A and Kemme, B, 2013. Peer-to-peer architectures for massively multiplayer online games. ACM Computing Surveys, [online] 46(1), p.11. Available at: https://www.researchgate.net/publication/262233529_Peer-to-Peer_Architectures_for_Massively_Multiplayer_Online_Games_A_Survey [Accessed 15 November 2021].