Bachelor's thesis

Information and Communications Technology

2022

Mika Oksanen

3D INTERIOR ENVIRONMENT OPTIMIZATION FOR VR

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | Number of pages 44

Mika Oksanen

3D Interior environment optimization for VR

In this thesis, the primary question was to determine if Level of Details (LODs) were optimizing the performance of the VR platform inside Unity. The secondary question was to find other optimization methods and study how they optimize performance for VR. The thesis was commissioned by the research group Futuristic Interactive Technologies of Turku University of Applied Sciences for their multi-user application named TUAS VR Social Platform.

To achieve the objective of this thesis, an altered interior of a real-world building was created. Two copies of the same scene were created where one was not optimized, and the other was fully optimized. The test was carried out by first measuring the unoptimized scene and then comparing it to other optimization methods. The unoptimized scene was then compared to the optimized scene.

The conclusion was that LODs do not significantly affect performance at this scale. The most performance gains occurred with occlusion culling and bake lights. Implementing all optimization techniques in the scene at once resulted in significant performance gains. Nevertheless, the test was ultimately successful as it showed that optimization can improve performance with VR.

Keywords:

Unity, Blender, game development, 3D-modelling, LOD, optimization, VR

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2022 | 44 sivua

Mika Oksanen

3D-sisäympäristön optimointi VR:lle

Tämän opinnäytetyössä tavoitteena oli selvittää, tuovatko yksityiskohtaisuuden tasot (LOD) merkittäviä suorituskyky etuja VR-alustaan Unityn sisällä. Toissijainen kysymys oli löytää muita optimointimenetelmiä ja kuinka ne lisäävät VR:n suorituskykyä. Opinnäytetyön toimeksiantajana toimi Turun ammattikorkeakoulun Futuristic Interactive Technologies -tutkimusryhmä heidän TUAS VR Social Platform -nimiseen monikäyttäjäsovellukseen.

Opinnäytetyön tavoitteiden saavuttamiseksi luotiin muunneltu tosielämän rakennuksen sisusta. Tästä luotiin kaksi kopiota, yksi ilman optimointia ja toinen täysin optimoitu. Testi suoritettiin mittaamalla ensin optimoimattomasta rakennuksesta ja sitten vertaamalla sitä muihin optimointimenetelmiin. Lopuksi optimoimatonta versiota verrattiin täysin optimoituun versioon.

Johtopäätös oli, että LOD:t eivät merkittävästi vaikuta suorituskykyyn tässä mittakaavassa. Eniten suorituskykyä paranivat occlusion culling ja light baking. Kun kaikki optimointimenetelmät olivat käytössä, suorituskyky parani huomattavasti. Silti testi onnistui, koska se osoitti optimoinnin olevan loistava tapa parantaa VR:n suorituskykyä.

Asiasanat:

Unity, Blender, pelinkehitys, 3D-mallinnus, LOD-tasot, optimointi, virtuaalinen todellisuus

CONTENTS

Pictures

TABLES

LIST OF ABBREVIATIONS

Baking	Pre-computing information into a file to speed up the process
CPU	Central Processing Unit
Draw call	Rendering task to decide many objects to draw on the screen
FPS	Frames Per Second
GPU	Graphics Processing Unit
LOD	Level of Detail
PC	Personal Computer
RAM	Random Access Memory
Texture	Image applied on a surface of a 3D object
UV-map	2D presentation of the surface of a 3D object, which is used for texturing
UV unwrap	Operation where the surface of a 3D object is spread into two-dimensional information
VR	Virtual Reality

1 INTRODUCTION

The game industry grows year after year. During the COVID-19 pandemic, millions of new players were introduced to gaming. Video games are projected to take a 10.9% share of global spending in the entertainment and media sectors by 2026 (Read 2022). This results in game developers making new astonishing-looking games while simultaneously leading the audience to be more critical of what they want to consume. As a consequence of this, game developers tend to take shortcuts and cut corners when making the game and this can ultimately lead to a poorly optimized game that will ruin the player experience. Waiting for the next area to load or massive framerate drops can make the players stop playing the game altogether. Additionally, it can impact the game's sales if the critics give it a bad review. The poorly optimized game can even increase power consumption, which is essential, especially for mobile devices. These are all common issues resulting from poorly optimized games.

Optimization refers to increasing the performance of a video game for a better overall experience and opening the possibility of having the game running on multiple devices. However, every game is unique and will have different problems and challenges to overcome. Fortunately, Unity has a vast amount of manuals and guidelines to help get started, but knowing when to use one or the other is time taking.

This thesis aims to determine if Level of Details LODs were making substantial performance difference for the VR platform inside Unity. The secondary question is to find other optimization methods and study how they increase performance for VR. To achieve these objectives, the author created an altered copy of a real-life building interior of Koneteknologiakeskus Turku Ltd to test the performance of optimization methods. First, two copies of the same scene are created where one is not optimized, and the other is fully optimized. Then, to test the performance benefits, the author uses Unity's built-in profiler tool to observe optimization results. After that, the results are compared to the same scene without the optimization methods.

To create the building, a free 3D software named Blender is used to make all the models. Then, Adobe Substance Sampler and Adobe Photoshop are used to create the textures.

The thesis is structured as follows. Chapter 1 introduces the objectives of this thesis. Chapter 2 focuses on the tools and software used during the project. Chapter 3 explains different optimization principles that should be done before putting the objects inside a game engine. Chapter 4 discusses optimization principles that should be considered when making games with Unity. Chapter 5 describes the methods that were used for the environment. Finally, Chapter 6 goes through the results, and Chapter 7 concludes the research.

2 DEVELOPMENT TOOLS AND TECHNOLOGY

2.1 Virtual Reality

Virtual reality (VR) refers to the use of a computer-simulated virtual 3D environment (Lowood 2022). To get inside the world, the user uses a head-mounted display typically (HMD) that the person wears. VR headset immerses the user in the virtual world by preventing them from seeing the real world around them (Vince 2004).

For the best user experience, the VR should get a high enough framerate, and optimizing will be vital for achieving this. The targeted framerate can be hard to achieve because VR requires every frame to be drawn twice, once for each eye. This generally means that every mesh and texture is drawn twice, so the draw call count is double (Oculus 2022). Poor framerate can cause the game not to run smoothly, and this can lead to motion sickness for the user (Vince 2004).

2.2 Used Software

A game engine software is a development environment with functionalities and settings built-in for developing different 2D or 3D games (Gregory 2018). The most popular game engines out there are Unreal and Unity. They both have many features built-in and extensive documentation to help new and experienced game developers. For this project, the chosen game engine was Unity for its use already in the commissioner's project. This ensures that it is as easy as possible when the project is tested together.

When making games, the world needs something, or they are just an open world with many possibilities. For this, 3D software comes to the rescue to fill the void. 3D software is computer graphics software that allows the user to develop, design and produce 3D graphics and animations. The main principle is to work inside 3D space with three axes: X, Y and Z. 3D software are not all

equal, and some are more specific in another aspect than in another. Also, 3D software are often under a licence, which can be expensive. Blender is, on the other hand, a free, open-source 3D software. For this project, the author chose Blender for its benefits of being free, and the author has some experience with the software. Other excellent and popular 3D software are Autodesk Maya and 3DS Max, ZBrush, Houdini, and Cinema 3D.

While 3D objects might look good, they are missing a part to make them stand out, and that is textures. There are many different software to make textures, but the two most popular ones are Gimp and Adobe Photoshop. Gimp is a free and open-source image editor with lots of features. However, Adobe Photoshop contains far more tools to edit images, but the downside is that Adobe is paid software. Nevertheless, the author was familiar with Adobe software, so the textures were made with Adobe Photoshop and with Adobe Substance Sampler. Substance Sampler is an excellent tool for making textures out of images because it automatically creates different maps for the image. Photoshop was used to finetune these images then.

Creating the environment from scratch is no easy task, but luckily the Koneteknologiakeskus Turku Ltd interior was already captured by Matterports with their laser scanner cameras. The interior was accessed from the Koneteknologiakeskus website, and it could then be used to accurately see the interior without going there all the time. The website also had a measurement tool that helped to measure the premises and machines with high accuracy.

3 OPTIMIZATION OF 3D MODELS

3.1 Optimization fundamentals

Video game optimization means increasing the performance for better visual experience and gameplay, regardless of the level of graphic setting. An optimized game means it can work on a wide range of different hardware and maintain consistent Frames Per Second (FPS) no matter the platform (Garney & Preisz 2010). This is especially important when games get imported to different consoles, mobile or, in some cases, to Personal Computer (PC). Regardless of the game's size, a handful of general principles and good practices should be followed. To ensure this happens, the graphic team and programmers must work together closely, or it can cause issues later in the development process (Unity 2021b).

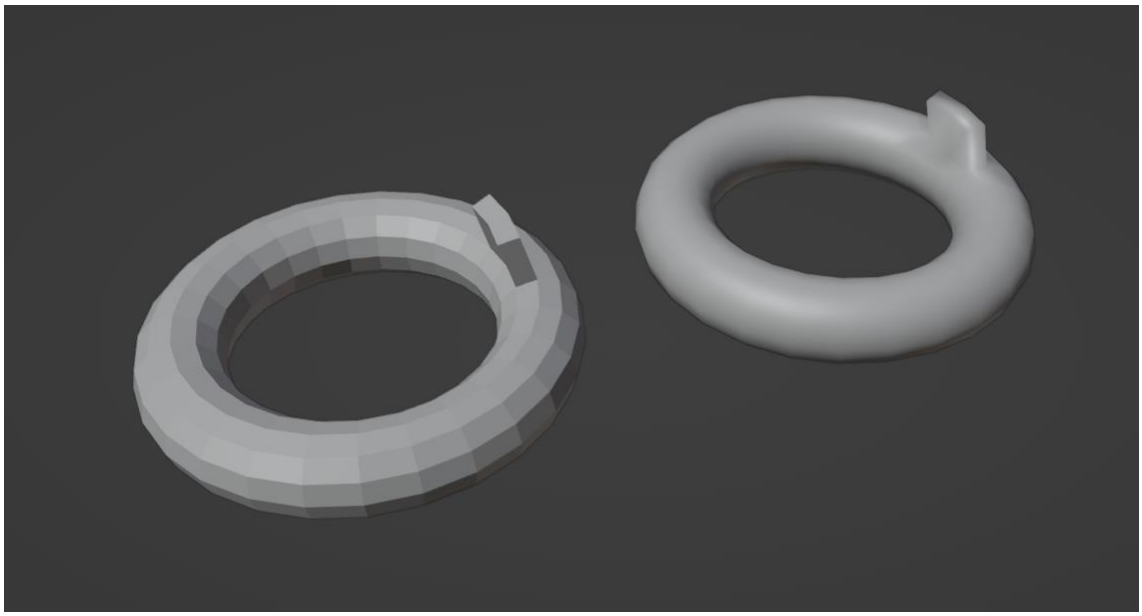
Visual optimization means optimization methods related to video games' visual properties and not programming. For example, visual optimization includes the 3D models and textures visible in the game and the game's lighting. In other words, visual optimization includes methods that are related to the rendering of game objects (Unity 2021b). For example, shadows and reflections created by lighting exposure should be calculated in advance so that the lighting does not have to be calculated constantly during gameplay because this would significantly impact the performance (Gonzalez 2017). One of the most straightforward and crucial visual optimization methods are hiding objects that cannot be seen, baking lighting beforehand, making Levels of details for the 3D objects and using suitable formats and different levels of resolution for textures (MIPs). When using these techniques for optimization, they are often relatively easy to notice when playing games. However, when done right, they are imperceptible by the players (Meshmatic 2020).

3.2 Fundamentals of 3D model optimization

The polygon count is something that the artist should keep an eye on when making models for video games because it plays a vital role in game optimization. It is especially crucial when making mobile games because they may need more performance to display a detailed model (Jie et al. 2011). Reducing the number of polygons is one of the essential optimization steps and should be considered from the beginning. When making 3D models, one must consider the number of polygons and the model's appearance. The tricky part is to know what is right. Adding too much detail to a model can affect the gameplay, but if adding too little detail, the more the model's appearance suffers (Unity 2021a). However, by reducing the polycount of the model, it will be easier for the artist to make changes later to the mesh and texture it (Meshmatic 2020). However, to know the desired polycount, we will need to know the target application of the model. One of the most significant benefits of having a well-optimized model is file size reduction (3D-Ace 2022). When making a game, especially when many people are working on it, it is to have it as small as possible file sizes to reduce the time it takes to open and download everything. It cannot be avoided easily but acknowledging it is a good thing.

When making 3D models with hard edges, they might look unnatural, as would be expected, as in the actual world, edges are rounded. This would lead to adding a bevel to every hard corner of the model to achieve this look. It would increase the polygon count by a thousand and simultaneously impact the object's performance and size way too much. To solve this problem, there are several techniques to make the object look more rounded and simultaneously smoother than it really is. Smooth shading is a common technique used to make this happen. It will trick the lighting into making the object look much smoother and does not alter the object's geometry but changes its appearance. (Blender Foundation 2020). Smooth shading does not change the object's silhouette, so when viewed from certain angles, we can still easily see the edge count, thus breaking the illusion as seen in Picture 1. Another way to make objects appear smooth is to add subdivisions to the objects. Subdividing is not

ideal for game assets because it will add more geometry to the object and impact the performance.



Picture 1. Flat shading on the left and smooth shading on the right.

3.3 LOD – Level of Detail

LOD stands for Level of Detail, and it is a process to reduce the mesh complexity and the details of the object when viewed further away from the camera (Arm Developer 2022a). Level of detail is a technique to speed up rendering by lowering the detail for objects which take up a small screen area and helps to lower the strain on the computer and allow more objects to be rendered while maintaining a high FPS. LOD are a widespread practice, especially in open-world games, where there are vast distances between objectives (Denham 2022). The most detailed version of the model will be seen when the player is closest to the object. The further way player goes from the object, the more it will be changed to a more primitive object with fewer details (Arm Developer 2022a). LODs are generally numbered to start from LOD0, which is the most detailed version and is viewed from close. From there, it will be LOD1 and can go up to as many LODs as it needs, as Picture 2 demonstrates. There is no set-in-stone number on how many LODs the object

should have. It depends on how important the object is and on the size of the objective. Many background objects can also have a 2D image as the last LOD, decreasing the CPU load furthermore (Bonet 2021b).

Picture 2. Same object with multiple LODs.

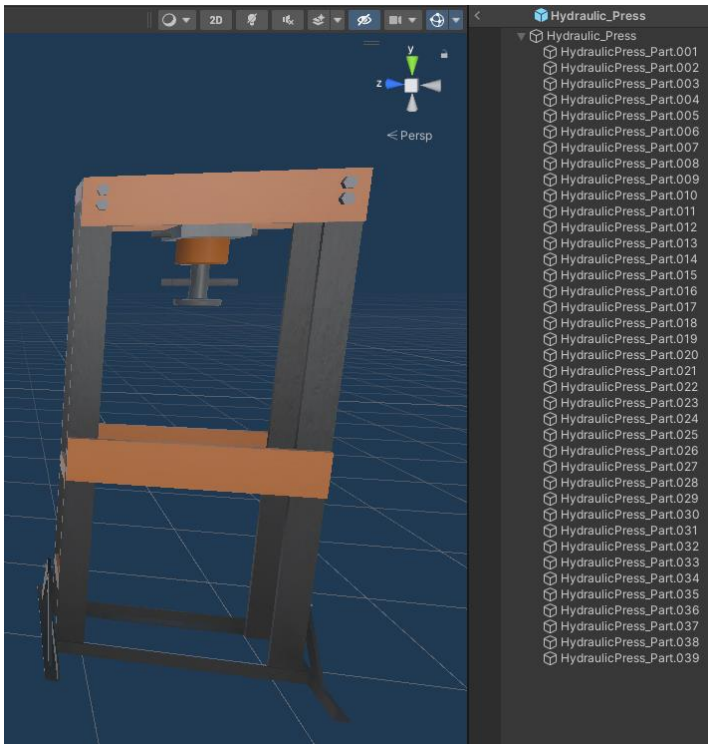
Making LODs and adding too many LOD levels can cost CPU workload, and the increased size of the LODs combined will also affect the file size. On the other hand, not adding enough levels can cause popping where the LOD change is noticeable, breaking the immersion, or the benefit of LODs might not be worth it anymore (Arm Developer 2022a). LODs are recommended when the player can see the 3D object up close and from afar, especially when the object has lots of small details. If the model is enormous, it is recommended to make many LOD levels to stop the popping effect. Nowadays, almost every game that does use 3D model will use some form of LOD system.

3.4 Combining meshes

Every object that is visible in the scene needs to be drawn. In Unity, one of the CPU tasks is to send different rendering commands to the GPU. Rendering is resource-intensive for the CPU and can quickly add up over time (Meshmatic 2020). Combining meshes can be done inside the 3D modelling program, but

there can be tools inside the game engine to do that as well. For example, there is a robot arm with 80 bolts attached to it, but they never deviate from the original robot body, so they should be combined to the main objective. This will reduce the number of components from 81 to only one. Combining the meshes makes the CPU workload much lower (Meshmatic 2020). The reason being the CPU prefers to process one large model rather than many small ones. The cost of multiple meshes and one mesh is almost identical for the GPU, but the work done by the CPU to render multiple different objectives is much higher. Combining meshes into one can help boost the performance to the desired levels and reduces the amount of work the CPU and GPU need to do (Unity 2021f). Combining meshes not only reduces the workload but can also help keep the project organized. Picture 3 demonstrates how much more objects there are inside the editor. While combining meshes is a good practice, there are some things to consider when widely using them. When combining meshes, they should be close together. It is because almost every game engine has a visibility system that can determine whether the object should be culled or not. For example, if a player has a whole house made out of one object, it will render the whole thing visible even if the player sees a fraction of the house. This can impact the GPU, and its resources will be wasted because the player sees a couple of pixels. Also, when combining objects that do not share the same texture, it will not give any performance increase at all. That said, excessive merging can worsen performance by increasing the amount of work for the GPU (O’Conor

2017). The best solution would be to combine small objects that are close together because they will probably be on-screen at the same time.



Picture 3. Hydraulic press model made of multiple different meshes.

3.5 Optimizing textures

When creating textures, they should be as significant enough to meet the project's required quality. This can be vague because there are a lot of different factors, such as the platform the game is intended to be used, the art style of the game and the size of different objects. The artist should consider the objects in the background, or are they essential hero objects like a sword? The best advice when making textures for 3D assets is to make the size a power of two on each side (512, 1024, 2048, 4096). The textures usually do not need to be square. The height can be different from the width (KatsBits 2022). For example, 64 x 512 and 32 x 156 are both acceptable since they can still follow the power of two rule. Unity can more easily resize and compress the textures by following the power of two methods. Some compression tools also need this to function correctly (KatsBits 2022).

A great way to optimize textures is using a texture atlas, which is one large image that is made out of multiple smaller images packed together (Arm Developer 2022c). For example, a 4K (4096 x 4096) texture atlas can hold up to four 2K (2048 x 2048) texture maps, as Picture 4. demonstrates. However, these square textures can be easily split into smaller pieces, with three 2K textures and four 1024-pixel textures, for example (O'Conor 2017). Textures can be different sizes depending on what kind of texture map they are and how much detail they need. For example, an albedo texture could be 2048 x 2048 pixels, while a roughness map could be only 1024 x 1024 pixels. It is not a set rule but should be observed if it makes a too noticeable impact on the visuals.

Picture 4. Texture atlas.

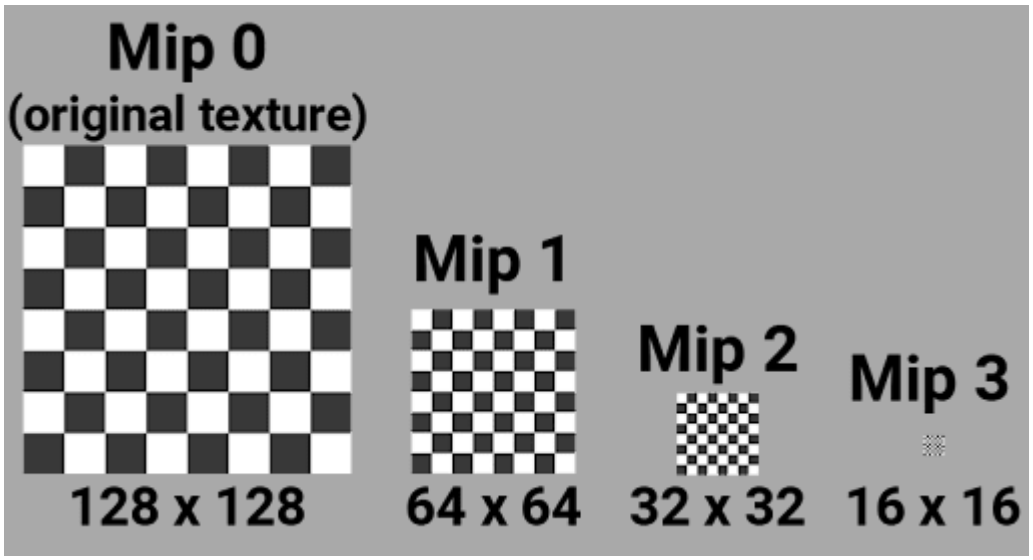
Unity can handle most common texture formats like PNG and JPG, although GPUs cannot use these formats directly. Instead, GPUs need to use different and more specialized formats to optimize memory usage and sampling speed. Unity does this by decompressing and recompressing the original file to be suitable for the GPU. It can also reduce the texture file size by a quarter without decreasing the image quality too much, and for this, choosing the right

compression tool for the right textures and the targeted platform (Unity 2021i). The recommended compression formats for PC are DXT1 and DXT5. DXT1 will give higher-quality compression but is slower and can only use RGB texture. DXT5 can use RGBA textures and has faster compression but loses in quality. For iOS, PVRTC gives a broader range of compatibility. On the newest iOS hardware, the recommended format is ATSC format. For Android, the recommended compression formats are ASTC and ETC (Unity 2021g).

3.6 Mipmaps

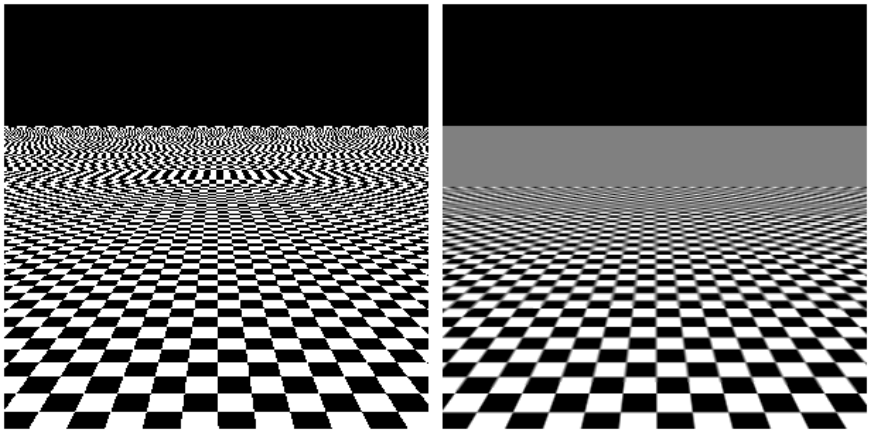
When making textures, artists want them to be of the highest quality possible. Unfortunately, this means that the texture sizes are large and can put a strain on the computer. However, their details are not noticeable, especially when using large textures far away, which will drain performance from the GPU, which is when Mip levels come in handy. Mips are stored in a mipmap containing different resolution versions of the texture, as Picture 5 shows an example of different MIP levels. Mipmaps are equivalent to LODs, but for textures (Unity 2021d). Artists can create mipmaps manually, but most game engines have tools to generate mipmaps automatically, but the original texture resolution must be in a power of two value (Jeremiah 2019).

Mipmaps are commonly used for 3D objects that can be seen from a distance and up close by the camera. To calculate the correct mipmap level, GPU will need to perform calculations based on the texture's angle and distance from the camera, concluding how much of the texture is visible to the camera (Unity 2021d). The GPU does this by calculating the pixels, so it much considers the player's screen resolution to calculate it correctly. With all this in mind, the GPU will determine what resolution mip it will show (Arm Developer 2022b). If the texture is a very sharp angle or a great distance away, it will pick a lower resolution mip. In vies versa, if the object is close and slightly angled, it will choose a better quality mip.



Picture 5. Different Mipmap levels (Unity 2021d).

Mipmaps not only give a clear performance boost when used correctly but can also be used to prevent the formation of artefacts on large surfaces. (Roldan 2022). This kind of interference results from viewing a large texture in certain angles where the texel density is getting too high will result in moiré patterns on distant surfaces (Gregory 2018). When mipmaps are used correctly, the moiré patterns can be fixed, resulting in better visuals, as Picture 6 shows.



Picture 6. Moire Pattern before and after mipmapping (Roldan 2022).

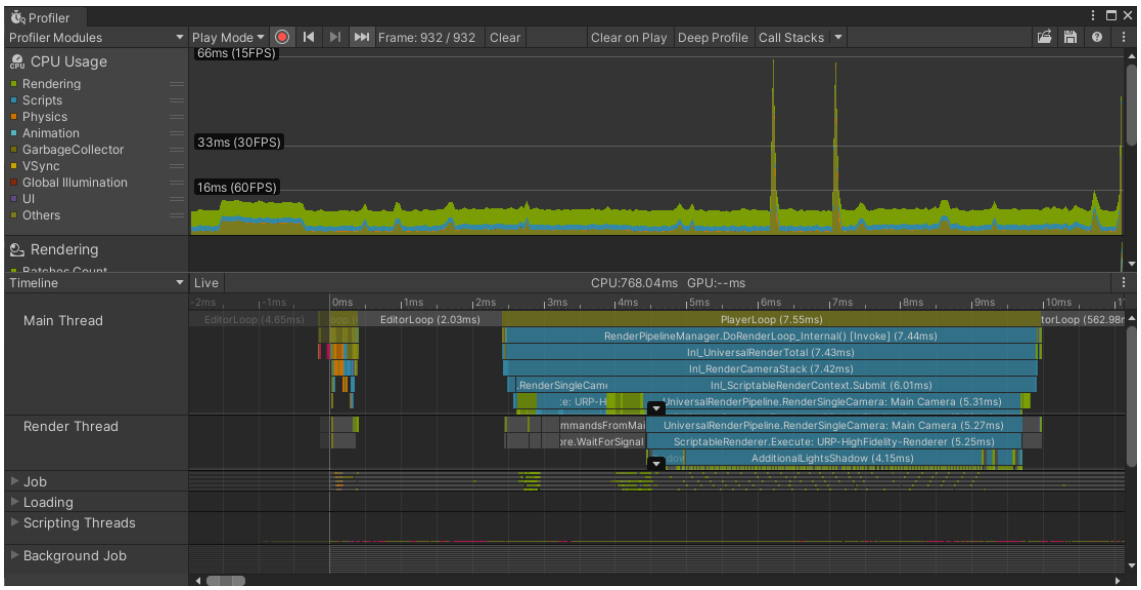
Mipmaps are excellent overall, but there are some situations when they are not a good option. When generated, Mipmaps will increase the texture's size on

disk and in memory by 33%. If the texture only uses the full resolution, such as User Interface (UI) textures, it will not provide any additional benefits. Additionally, if the object is always close to the camera, like the player model, there are no benefits in making a mipmap (Unity 2021d).

4 UNITY PROJECT OPTIMIZATION

4.1 Profiling

When making games, developers should always watch the scene's performance. The game could run smoothly, but there might be issues below the surface. For this, many game engines have built-in tools that allow the game developers to see how well-optimized their game is during the game's development and after the release. Unity's built-in profiler tool comes in handy in this situation. With the profiler tool and its many different modules, as seen in Picture 7, developers can examine many areas of the application's performance, such as audio, lighting, memory usage and CPU load. It is advised to use profiling with the targeted platform of the game it is intended to be used. For example, use profiling with the phone if the game is meant to be played with a mobile device. The application can still be tested inside Unity, but the results will differ (Unity 2022).



Picture 7. Profiler tool from Unity

With the profiler tool, developers can easily find problems in the projects, figure out the root causes, and iterate on those areas while simultaneously increasing

their application performance (Unity 2022). In addition, with the profiler tool, the developers can pinpoint the performance costs. Without seeing the information about the application, it leads developers to look for solutions with guesswork. It would lead to optimizing pieces of the application that do not require optimization at all, thus using resources for nothing (Unity 2021b).

4.2 Reducing draw calls

Reducing draw calls is one of the most effective ways to improve the game's performance. Draw calls contain information about shaders, textures and other rendering objects that the CPU will send to the GPU so that it knows what to render and how. Unfortunately, draw calls are heavy for the CPU, so the more there are, the more strain it puts on the CPU and simultaneously reduces the performance (Unity 2021f).

There are a few options to reduce draw calls, like batching (Unity 2021f). However, some tasks inside the modelling program should be performed before even importing the models into Unity. The reason is that if the objects have numerous meshes and each has its own material, it will quickly add numerous draw calls. This is because Unity needs to calculate the draw call for each material and mesh. For example, there is a model of a car with four wheels and a body that is not in the same mesh, and they all share the same material. Therefore, when added to Unity, it will need five draw calls when rendering. However, if everything were combined into one mesh, the draw calls would drop to one. The reason is that the objects only have one mesh and one material in the best case. Nevertheless, this only works for objects that will not have parts moving independently (O'Conor 2017).

4.3 Static batching

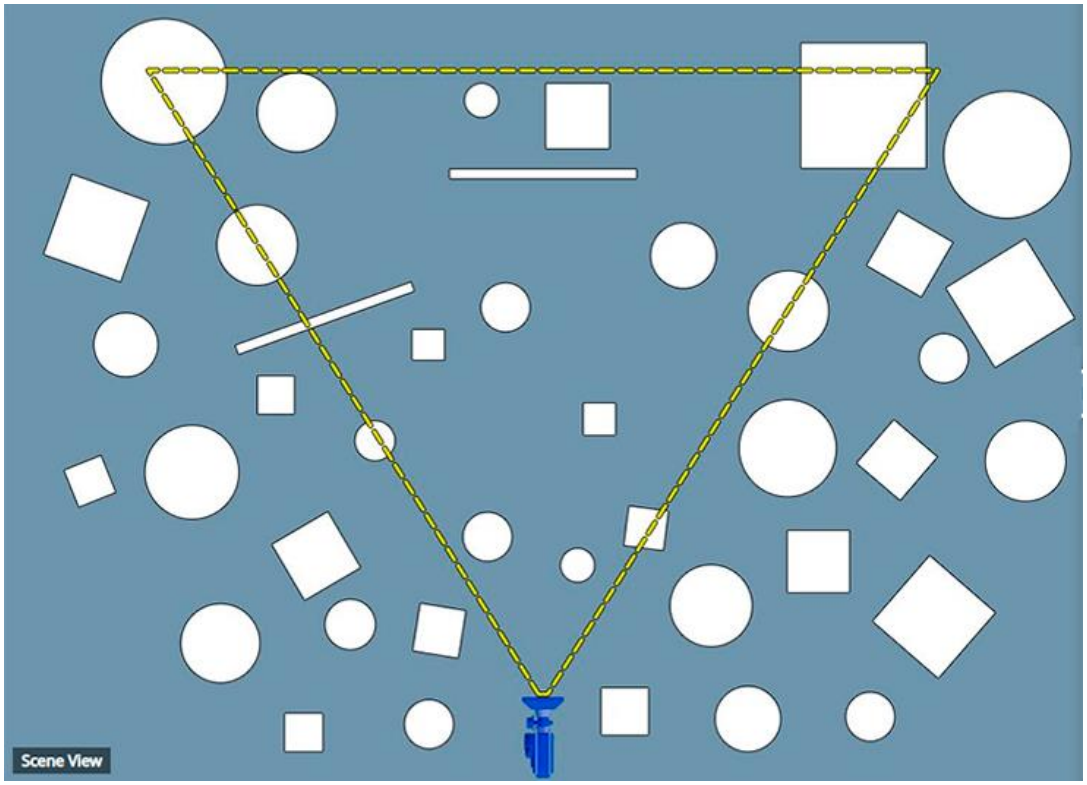
Game objects can be toggled to be static inside the Unity editor. Static means that the objects should not move inside the game (Gonzalez 2017). Unity can

pre-calculate information about the object and potentially improve its performance. Furthermore, it can be improved by adding static batching to the mix (Unity 2021h).

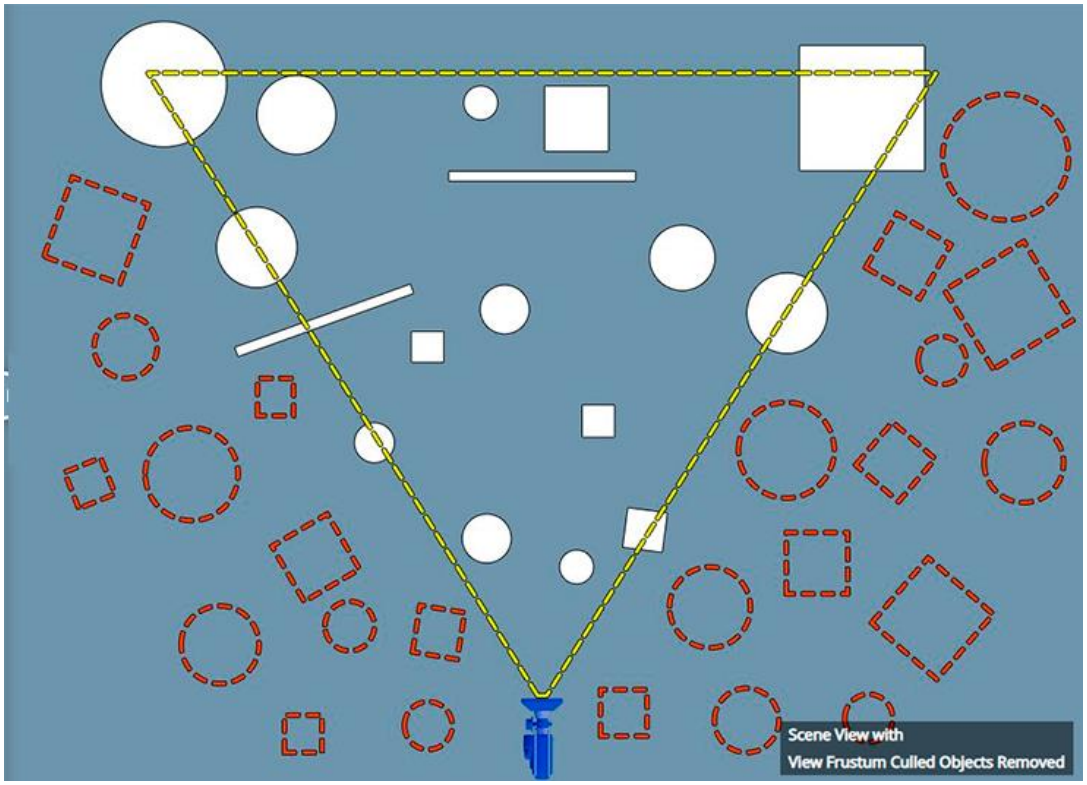
Static batching is a method that combines multiple meshes that are not moving. However, for this to work, the game objects must be marked static and share the same material. Marking objects static reduces the number of draw calls, increasing the performance, but the drawback is the increased amount of memory cost (Unity 2021h).

4.4 Frustum Culling

Frustum culling is Unity's default culling method, of which the primary purpose is not to render objects outside the camera's field frustum (Unity 2021j). Picture 9 shows how frustum culling works by dividing a pyramid-like shape into a far-clipping plane and a near-clipping plane. The clipping planes will determine whether the object needs to be rendered (Six 2021). The near-clipping plane is the closes point that the camera will be rendering objects visible, and the far-clipping plane is the farthest the objects can be visible. Therefore, anything that is closer or further away from the clipping planes will not be rendered (Autodesk 2020). Without any culling, the game would render everything, even if the player does not see the objects, as demonstrated in Picture 8.



Picture 8. Scene without culling (Unreal Engine 2021).



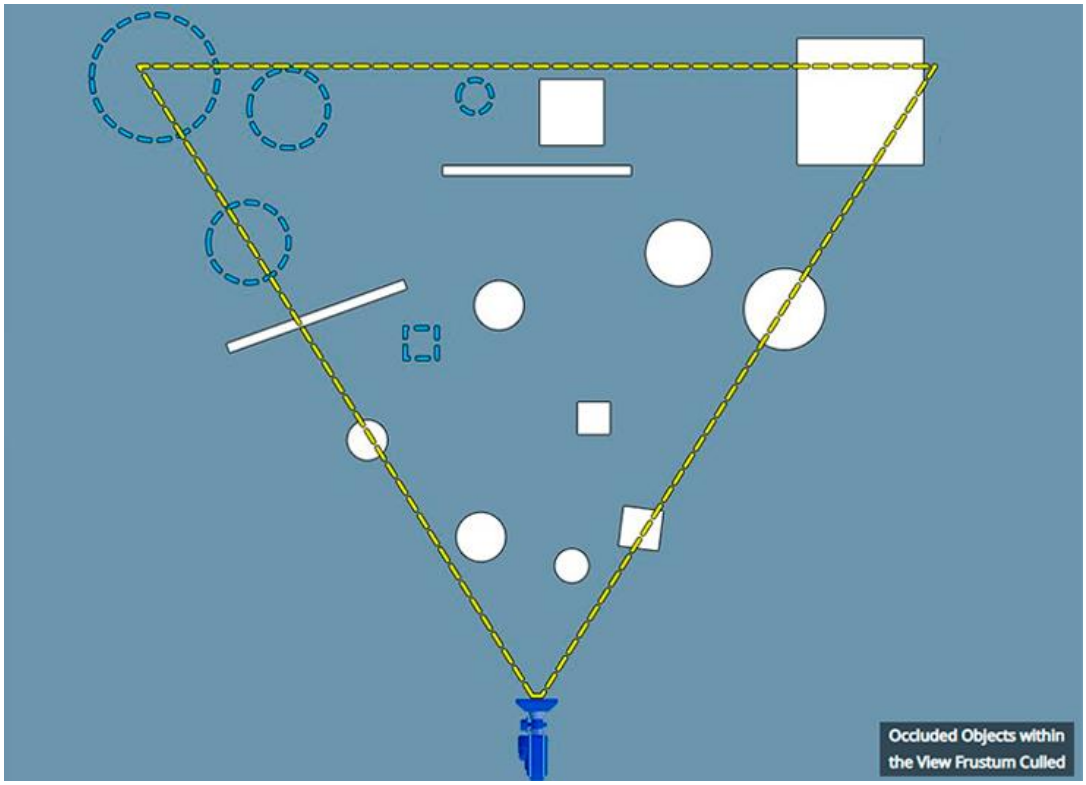
Picture 9. Scene with frustum culling (Unreal Engine 2021).

Frustum culling is an excellent method to optimize scenes because the GPU does not need to render all the objects, only the ones within the camera's field frustum. However, the downside to frustum culling is that it cannot hide objects that are behind other objects. This can lead to hundreds or even thousands of objects being rendered even if they are not seen by the camera, which will cause unnecessary work for the GPU and CPU (Unreal Engine 2021).

4.5 Occlusion Culling

Occlusion Culling is a process designed to reduce the processing power required to render a scene by not rendering the parts that are not visible to the player at runtime. In other words, the objects are not rendered if they are outside the view frustum or hidden behind other objects closer to the camera, so the method heavily relies upon other objects making visual barriers (Unreal Engine 2021). In addition, occlusion culling must be set up and baked ahead of time, unlike frustum culling (Unity 2021e).

Occlusion culling is very useful, mainly when used in an environment with many corridors or inside buildings (Unity 2021e). The inside of a building is an excellent example because objects are more likely to be hidden behind other objects, like doors and walls, as seen in Picture 10. However, with this in mind, occlusion culling is not very useful when used outside with vast space between objects, and everything has a clear line of sight.



Picture 10. Scene with occlusion culling (Unreal Engine 2021).

It is good to remember that occlusion culling will be more performance-heavy for the CPU in runtime. This is because Unity uses the CPU to calculate the occlusion culling. However, when using occlusion culling correctly, it will also improve CPU performance by lowering the draw calls, and with this, it will also improve GPU performance (Bonet 2021a). Therefore, Occlusion culling should be considered to be used when the game is GPU-bound (Unity 2021e).

4.6 Light Baking

Good lighting can make the game look great and more alive. Lighting gives the game life but can also simultaneously break the game's performance. Real-time lighting can make the scene look great, and it is also able to change during runtime. The drawback is that, as the name implies, it is real-time, so Unity needs to calculate the light hitting different surfaces and the shadows it creates

in every frame. Real-time lighting can be very costly for performance, especially on mobile or low-end hardware (Gonzalez 2017).

In such a case, light baking will come in handy. Light baking means the lighting has been calculated beforehand inside the game engine and saved on the disk as lighting data. Baked lighting will reduce the rendering cost of shadows and shading because the lighting has been calculated in advance (Unity 2021c). When developing games, it is recommended to use baked lights as often as possible. The downside is that the baked lighting cannot be changed during runtime, so the game objects must be set as static objects (Gonzalez 2017).

5 MAKING THE ENVIRONMENT

5.1 Background and objective

This thesis was commissioned by Turku University of Applied Sciences research group called Futuristic Interactive Technologies for their multi-user application made with Unity, named TUAS VR Social Platform. The object was to create a copy of a real-life building interior of Koneteknologiakeskus Turku Ltd and to make it modular for easy reconfigurations with VR in mind. The main objective of this thesis was to determine if LODs were making substantial performance benefits for the VR platform inside Unity. The secondary question was to find other optimization methods and how they increase performance for VR. However, to get the maximum performance benefits of LODs, several other optimization methods must be considered alongside LODs. After that, the Koneteknologiakeskus building is to be used inside the social platform projects.

The social platform is meant to be played on almost every type of PC, no matter how low-end hardware is used. The koneteknologiakeskus has a 3D space presentation on their website that was very helpful when making the interior. When all of this is considered, the environment could not be overly detailed so that it could be used inside the TUAS VR Social Platform.

5.2 Modelling the environment

Because this project was to be done so that it would be added to the social platform project later, the models could not be too detailed or high poly. Keeping the model's polygon count low as possible ensures having a broader range of hardware. However, the models should still have enough polygons to test the LODs effectively.

The modelling should always start with conceptualizing and blocking out the scene, as seen in Picture 11. The blockout dimensions were obtained from the Koneteknologiakeskus website, which has a 3D scanned model of the building.

For this project, the first part was to make the modular parts for the wall, ceiling and floor. These were done because they were the basic building blocks for the future. With them, it was easier to make other machines and smaller props further down the line and get them to the correct dimensions. After making the blockout, the critical big machines, like the CNC machine, were next in line. These are important because the commissioner wants them to be included inside the building. After the essential machines were done, it was time for the smaller props and less important background objects.

Knowing the project goals is essential since they will affect how they go with the production process. Changes that need to be made later on might cause problems with the whole project (Galuzin 2017).

Picture 11. Blockout of the Koneteknologiakeskus.

When making 3D models, the artist should roughly know what the end product should look like, mainly the art style and polygon count. There is no set-in-stone value for polygon count, but because the models are supposed to be not photorealistic, the polycount is reasonably low (Meshmatic. 2020). This helped with the modelling process in the long run when clearly seeing what needs to be achieved. Knowing the purpose of the model in the scene is also always helpful. Most of the models are only static without any movement or functions, so most

of the models are made from a single mesh to help reduce the draw calls. (O'Conor 2017).

5.3 Making the LODs

When making LODs, it is not always the best idea to make them for every object in the scene, but this will highly depend on the project. For this project, the machines would be a significant and essential part of the scene, so they needed LODs. When making LODs, the artist should keep an eye out for the object's silhouette so that it will not change too much from the original. This will help mitigate the noticeable popping effect later (Arm Developer 2022a).

LODs can be done mainly by two approaches, both with advantages and disadvantages. The first approach to making LODs is to reduce the geometry manually. Manually reducing is useful when wanting more freedom and maintaining a cleaner topology of the object. This is more useful when making objects that need to change shape, like characters. The second approach is much easier and quicker than manually making, and it is by using the Blenders decimation modifier. The decimate modifier will try to reduce the geometry without changing the object's silhouette too much. The drawback with this is that the decimate modifier will make some mistakes that might need fixing manually. For best results, it is best to use both for easy and quick progress. Only manually making the LODs will take extra time, and the average player might not notice the difference.

All LODs were made using Blender's decimate modifier for quick results, but some models also needed manual adjustments for better visual results. The amount of LODs depends highly on the object type and its importance for the scene. Most of the objects had three LODs, so LOD0, LOD1 and LOD2 were quite enough for the project size. Objects bigger or more prevalent in the scene were given more levels, as demonstrated in Picture 12. It is good to acknowledge that not every object in the scene needs to have LODs. For

example, in this case, the floors and roof are only scaled versions of a plane, so there is no need to do LODs. It would only worsen the look of the scene.

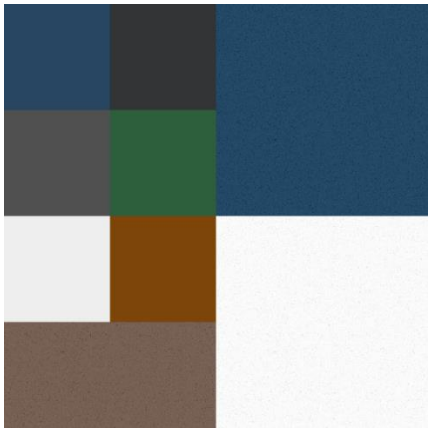
After the LODs were done for the models, they were exported to Unity by FBX-file. Inside Unity, if everything goes right, it should automatically add the LOD Group component. With the components, it is easy to change LOD group distances or add and delete LODs entirely.

Picture 12. From left to right. Different LODs of the same object decrease in detail inside Unity.

5.4 Making the Textures

The texturing process first started by marking down the most important and most used materials inside the Koneteknologiakeskus. This was done by visiting the place and taking pictures of the surfaces. Then, the images were imported to Substance Sampler, where they would be more edible. The primary purpose was to get the textures tileable and also, at the same time, make Normal maps for them. After making several tileable textures, they were imported to Adobe Photoshop to make them into one big texture Atlas to reduce

the needed materials. For the project, there was a total of 3 different texture atlases created. Texture atlases are a great way to improve performance while minimizing the project size and helping Unity with batching. Texture atlases will also help with the organization of the project by having fewer materials, as seen in Picture 13.



Picture 13. Texture atlas with more textures.

5.5 Baking the lights

After the models and textures were in place, for the most part, it was time to add lighting. With lighting, the whole scene will burst to life, making it feel more natural. When adding light to the scene, they are set to real-time by default, as seen in Picture 14. Real-time lighting is good when needing to change the light effects, or the objects need to be moving inside the scene. The downside is that it is very costly, and it will not be the best choice when making a scene for VR. However, because the scene has nothing that is moving, it is the best option to have. With bake lighting, the performance benefit will be huge compared to the real-time lights.

To start baking, the lights in the scene need to be selected as baked lights. After that, it is good to identify which objects are good candidates to bake lightmaps. Generally, all medium or large static objects are suitable for this. Small props and massive objects like terrain are not good lightmap targets. All the big props, like the machines and walls, were good candidates for this

project, as demonstrated in Picture 15. For this to take effect, selected objects need to be marked to be included in the global illumination from the lightmapper. (Unity 2020).

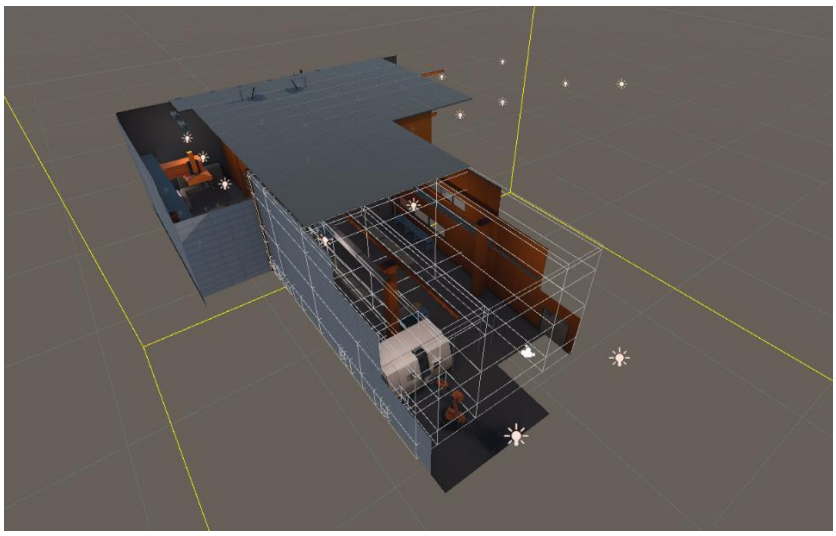
When baking lights, it is highly recommended to start with low lightmap resolution for fast iterations and tweak it afterwards. However, this can lead to artefacts that should be fixed with higher resolution. Furthermore, a higher resolution should be made relatively late in the development because light baking can take up to a couple of hours in higher resolutions. For this scene, the lightmap resolutions were kept relatively low because the scene was for concept purposes and will be iterated more in the future.

Picture 14. Interior with real-time lights.

Picture 15. Interior with baked lighting.

5.6 Adding Occlusion culling

Adding occlusion culling can be a massive help for FPS, but in some cases, it can also worsen the performance. Unity uses frustum culling by default, so the occlusion culling needs to enable inside the camera. In addition, occlusion culling only works with static objects, so it was a good candidate for the scene because nothing was moving. The first to get occlusion culling to work is to set the objects to be occluder and occlude and bake the occlusion data. Picture 16 demonstrates the scene where occlusion culling is enabled.



Picture 16. Scene with occlusion culling enabled.

6 TESTING THE ENVIRONMENT

6.1 Examining the optimization methods

The testing was done with a reasonably powerful desktop with a windows 10 operating system. The CPU is an Intel Core i5-8600K 6-core (6 threads) 3.6GHz (Boosted up to 4.30 GHz), and the GPU is Nvidia GeForce RTX 2070 8GB GDDR6. The system RAM is 32GB DDR4 3400MHZ dual channel. The VR headset used to test was Meta Quest 2.

Because the project needed to be made from scratch with the help of the Matterports 3D environment from the Koneteknologiakeskus website, some things were made in advance before putting them on the scene. Mainly the texture atlases because they are proven to give better performance. Making textures and materials for all would have been unnecessary work, so the texture atlases were always present in the testing. Testing was done to determine if the optimization methods would increase the performance was done by first taking performance data from different places of the scene. After that, adding one of the optimizing methods to the scene and compare the results. Lastly, compare the unoptimized scene with a fully optimized scene. The unoptimized scene had five different points to take the measurement. The average FPS was 36, Batches and draw calls were 16378 and CPU time was 27.7ms. The average triangle count was 4.1 million. Picture 17 shows one of the measurement points.

Picture 17. Unoptimized interior

6.2 Testing LOD system

The main focus was to determine if LODs will help performance with VR. The outcome was unexpected because the performance did not increase as much as was anticipated. The FPS only increased a bit, but batches and draw calls only lowered a couple hundred on average. CPU and GPU were both also almost unchangeable. Only the triangle count was impacted hugely by reducing the average triangle count to 1.4 million from 4.1 million. From this, it is clear that the LOD system was not worth it in itself, considering the time and effort it took to make them. The results would have been much more noticeable if the scene had been much larger in size with more objects with LODs.

6.3 Baked light testing

Real-time lighting is notoriously known to drain the performance of projects. Therefore, the best solution was to bake the lighting beforehand. The baking process took around 10 minutes for this scene, but the result was more than expected. The average FPS went from 36 to 70, which is a massive 94.4% increase in FPS. Also, the batches and draw calls decreased from the starting 16 000 to only 3500 on average, which is a 78.1% decrease. The testing was done multiple times to ensure that they were right. This was such a massive boost for the performance that it is highly recommended to bake the lighting for the interior building. The only downside to baking is that it takes time, but if the benefits are almost double, it is still worth doing.

6.4 Testing Occlusion culling

For this test, all of the objects were converted to static objects. It was a crucial step for getting the occlusion culling working. The scene needs to be baked like the lighting, but it took much less time. The occlusion bake was done with the

default values. The FPS average was the same, but there were areas where the occlusion culling could have worked better, and the FPS almost got up to 60 FPS. Triangle count also dropped an average by half, but the most significant change was with the Batches and draw calls, averaging around 3500. That is almost the same as with the baked lighting. When all of this is considered, it is pretty effortless to bake the occlusion data to the scene. However, it did lower the batches and draw calls considerably, so it is wise to do occlusion culling, at least for this scene.

6.5 Fully optimized scene

The result was quite promising when all of the above were put together to make the most out of the scene. The fully optimized scene was then compared to the unoptimized version to see the difference.

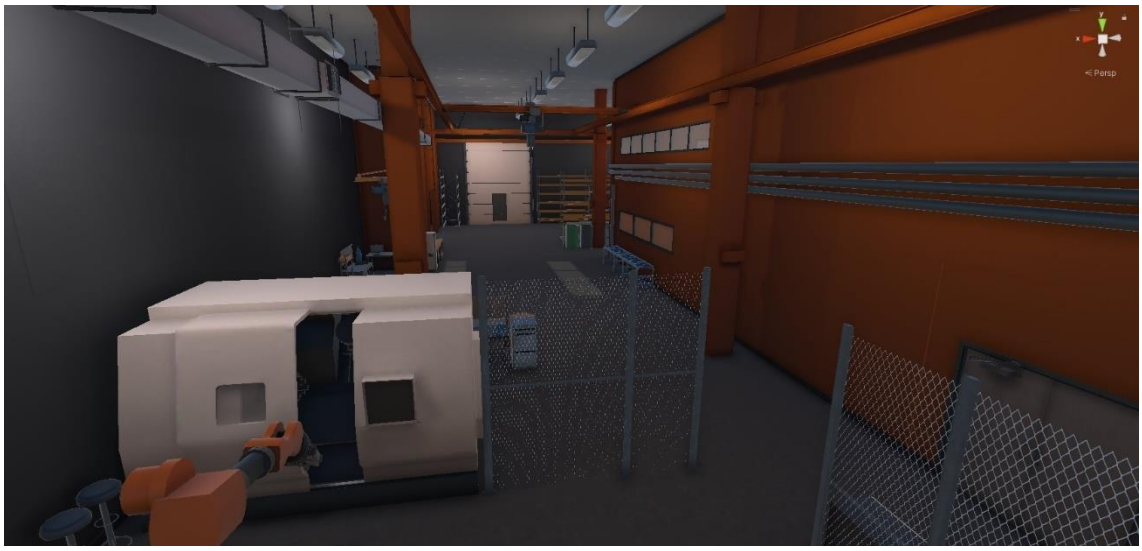
Table 1. Comparison between unoptimized and optimized scenes.

Scene	FPS	Draw calls	triangle count
Unoptimized scene	36	16 378	4.1 million
Optimized scene	72	650	200 000
Difference	100% Increase	96% Decrease	95% Decrease

The difference was huge as the average FPS was 72, and the average Baches and draw calls were low as 650. The average triangle count was slightly over 200 000 from the starting at 4.1 million. This indicates that optimizing has a massive role in improving performance. The FPS increased on average by 100%, and the batched and draw calls decreased an immense 96%, so they were almost at the minimum. The only downside to this is the time and effort it will take to make this all happen and how they might alter the game's look.

When comparing the result, the clear winners were the occlusion culling and the baked light. The worst contester was the LOD system, which frankly took the

most time to make. It is easy to say that, when making small interior scenes with low poly objects, do not worry about LODs at the start but focus more on the light baking and occlusion culling. Picture 18 shows the fully optimized scene from one of the measurement points.



Picture 18. Fully optimized scene.

7 CONCLUSION

In this thesis, the primary question was to determine if Level of Details (LODs) were optimizing the performance of the VR platform inside Unity. The secondary question was to find other optimization methods and study how they optimize performance for VR.

For this, the author made an interior scene of an existing building with some of the machines and props inside. From this same scene, there was an unoptimized version that was compared to the optimized version of the same building. After the two scenes were compared, it was determined if the optimization method should be considered for future work. Some of the tests were carried out multiple times to eliminate undesirable results. The last test was conducted when all of the optimization methods were in use at the same time.

The tests clearly showed that for this scale, the LODs were not giving the hoped results, and there was almost no performance boost. The LOD system only impacted the triangle count considerably. This could have been avoided if the author had made the objects more detailed. The light baking and occlusion culling gave the best results by almost doubling the FPS and decreasing the draw calls and batches by an immense amount. When all the optimization methods were used, the performance was significantly better.

The LODs should be much more detailed for future work if tested in a small environment. All the optimization methods could have been more noticeable if the scene was more considerable in size, or there would have been more complex models. For future work, the Matterports laser scanned environment should also be used for its much more detailed models. Therefore, LODs with VR should be researched again in the future to uncover the benefits in full.

REFERENCES

3D-Ace, 2021. 3D Modelling for video games: How to create beautiful assets. Referenced: 20.9.2022. Available at: https://3d-ace.com/blog/3d-model-optimization/

Arm Developer. 2022a. Level of Detail – LOD. Referenced: 23.9.2022. Available at: https://developer.arm.com/documentation/102496/0100/Level-of-Detail---LOD

Arm Developer. 2022b. Mipmapping. Referenced: 25.9.2022. Available at: https://developer.arm.com/documentation/102073/0100/Mipmapping?lang=en

Arm Developer. 2022c. Texture atlasing. Referenced: 25.9.2022. Available at: https://developer.arm.com/documentation/102696/0100/Texture-atlasing?lang=en

Autodesk. 2020. Clipping Planes. Referenced: 10.10.2022. Available at: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/Maya-Rendering/files/GUID-D69C23DA-ECFB-4D95-82F5-81118ED41C95-htm.html

Blender Foundation. 2020. Shading. Referenced: 30.9.2022 Available at: https://docs.blender.org/manual/en/latest/scene_layout/object/editing/shading.html

Bonet, R. 2021a. How to Use Occlusion Culling in Unity — The Sneaky Way. Referenced: 29.9.2022. Available at: https://thegamedev.guru/unity-performance/occlusion-culling-tutorial/

Bonet, R. 2021b. Level of Detail (LOD) Tutorial for Unity 2021+. Referenced: 23.9.2022. Available at: https://thegamedev.guru/unity-gpu-performance/lod-level-of-detail/?utm_source=gamasutra&utm_medium=post&utm_campaign=unity_gpu_performance_lod_level_of_detail#why-do-you-need-lods-in-your-unity-project

Denham, T. 2022. What is LOD (Level of Detail) in 3D Modeling?. Referenced: 23.9.2022. Available at: https://conceptartempire.com/3d-lod-level-of-detail/

Galuzin, A. 2017. Making Difficult Level Design Decisions. Referenced: 15.11.2022 Available at: https://www.worldofleveldesign.com/categories/productivity_goals/making-difficult-level-design-decisions.php

Garney, B. & Preisz, E. 2010. Video Game Optimization. Boston. Cengage Learning PTR.

Gonzalez, J. 2017. Maximizing Your Unity Game's Performance. Referenced: 24.10.2022. Available at: https://cgcookie.com/posts/maximizing-your-unity-games-performance

Gregory, J. 2018. Game Engine Architecture, Third Edition. Page: 643—644. Referenced: 1.11.2022 Available at: https://books.google.fi/books?id=EwlpDwAAQBAJ&printsec=frontcover&hl=fi#v=onepage&q&f=false

Jie, J., Yang, K., and Haihui, S. 2011. Research on the 3D Game Scene Optimization of Mobile Phone Based on the Unity 3D Engine. Referenced: 25.9.2022. Available at: https://ieeexplore.ieee.org/document/6086340

Jeremiah, 2019. Learning DirectX 12 – Lesson 4 – Textures. Referenced: 28.9.2022. Available at: https://www.3dgep.com/learning-directx-12-4/#Mipmap_Filtering

KatsBits. 2022. Make Better Textures, The 'Power Of Two' Rule & Proper Image Dimensions. Referenced: 25.9.2022. Available at: https://www.katsbits.com/tutorials/textures/make-better-textures-correct-size-and-power-of-two.php

Lowood, H. 2022. virtual reality. Encyclopedia Britannica. Referenced: 5.12.2022. Available at: https://www.britannica.com/technology/virtual-reality

Meshmatic. 2020. 10 best practices when optimizing 3D files for VR. Referenced: 20.11.2022. Available at: https://meshmatic3d.com/technical/optimize-3d-files-ar-vr/

O'Conor, K. 2017. GPU Performance for Game Artists. Referenced:1.11.2022. Available at: http://fragmentbuffer.com/gpu-performance-for-game-artists/

Oculus, 2022. Guidelines for VR Performance Optimization. Referenced: 5.12.2022. Available at:

https://developer.oculus.com/documentation/native/pc/dg-performance-guidelines/

Read, S. 2022. Gaming is booming and is expected to keep growing. This chart tells you all you need to know. Referenced: 11.12.2022. Available at: https://www.weforum.org/agenda/2022/07/gaming-pandemic-lockdowns-pwc-growth/

Roldan, F. 2022. ANTI-ALIASING PROBLEM AND MIPMAPPING. Referenced: 25.9.2022 Available at: https://textureingraphics.wordpress.com/what-is-texture-mapping/anti-aliasing-problem-and-mipmapping/

Six, J. 2021. Frustum Culling. Referenced: 11.10.2022 Available at: https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling

Unity, 2020. How to build Lightmaps in Unity 2020.1 | Tutorial. Referenced: 18.11.2022. Available at: https://www.youtube.com/watch?v=KJ4fl-KBDR8

Unity, 2021a. Creating models for optimal performance. Referenced: 20.9.2022. Available at: https://docs.unity3d.com/Manual/ModelingOptimizedCharacters.html

Unity, 2021b. Graphics performance fundamentals. Referenced: 22.9.2022. Available at: https://docs.unity3d.com/2021.3/Documentation/Manual/OptimizingGraphicsPerformance.html

Unity, 2021c. Light Mode: Baked. Referenced 14.11.2022. Available at: https://docs.unity3d.com/Manual/LightMode-Baked.html

Unity, 2021d. Mipmaps introduction. Referenced: 25.9.2022. Available at: https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html

Unity, 2021e. Occlusion Culling. Referenced: 10.11.2022. Available at: https://docs.unity3d.com/Manual/OcclusionCulling.html

Unity, 2021f. Optimizing draw calls. Referenced: 22.9.2022. Available at: https://docs.unity3d.com/2020.3/Documentation/Manual/optimizing-draw-calls.html

Unity, 2021g. Recommended, default, and supported texture formats, by platform. Referenced 28.9.2022. Available at: https://docs.unity3d.com/Manual/class-TextureImporterOverride.html

Unity, 2021h. Static batching. Referenced 15.11.2022. Available at: https://docs.unity3d.com/2023.1/Documentation/Manual/static-batching.html

Unity, 2021i. Texture compression formats. Referenced 28.9.2022. Available at: https://docs.unity3d.com/2020.1/Documentation/Manual/texture-compression-formats.html

Unity, 2021j. Understanding the view Frustum. Referenced: 11.10.2022. Available at: https://docs.unity3d.com/Manual/UnderstandingFrustum.html

Unity, 2022. How to profile and optimize a game | Unite Now 2020. Referenced: 10.11.2022. Available at: https://www.youtube.com/watch?v=epTPFamqkZo

Unreal Engine. 2021. Visibility and Occlusion Culling Settings. Referenced: 10.10.2022. Available at: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/VisibilityCulling/

Vince, J. 2004. Introduction to Virtual Reality. London. Springer.