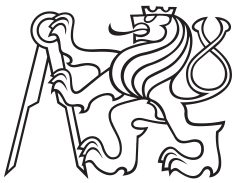
Bachelor Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Performance Optimizations in Unity

Jiří Kropáč

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kropáč** Jméno: **Jiří** Osobní číslo: **498990**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Výkonnostní optimalizace v Unity

Název bakalářské práce anglicky:

Performance optimizations in Unity

Pokyny pro vypracování:

Zmapujte optimalizační metody pro zobrazování detailních a rozsáhlých scén implementované v herním enginu Unity. Zaměřte se na podporu LOD, využití normálových map a paralax map, occlusion culling a optimalizace zobrazovacích dávek. Vyberte nejméně dvě ze zmapovaných oblastí a rozeberte je podrobně. Vytvořte nejméně tři testovací scény různé složitosti, které budou dobře ilustrovat funkčnost zvolených optimalizačních metod. Scény otestujte z hlediska rychlosti a kvality zobrazování na různě výkonném hardwaru. Podrobně popište doporučení pro využití zvolených optimalizačních technik v Unity - vhodnost pro různé typy scén, nastavení konkrétních parametrů v Unity. Diskutujte možnost využití automatického hledání optimálního nastavení pro danou scénu.

Seznam doporučené literatury:

[1] Lindstrom, Peter, et al. Real-time, continuous level of detail rendering of height fields. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996.
[2] De Floriani, Leila, Leif Kobbelt, and Enrico Puppo. A survey on data structures for level-of-detail models. Advances in multiresolution for geometric modelling. Springer, Berlin, Heidelberg, 49-74, 2005.
[3] Dietrich, Andreas, Enrico Gobbetti, and Sung-Eui Yoon. Massive-model rendering techniques: a tutorial. IEEE Computer Graphics and Applications 27.6, 20-34, 2007.
[4] Yoon, Sung-Eui, Christian Lauterbach, and Dinesh Manocha. R-LODs: fast LOD based ray tracing of massive models. The Visual Computer 22.9, 772-784, 2006.
[5] Áfra, Attila T. Interactive ray tracing of large models using voxel hierarchies. Computer Graphics Forum. Vol. 31. No. 1. Oxford, UK: Blackwell Publishing Ltd, 2012.
[6] Unity Manual. https://docs.unity3d.com/Manual
[7] Nanite Virtualized Geometry. https://docs.unrealengine.com/5.0/enUS/RenderingFeatures/Nanite/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **17.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Acknowledgements

I want to thank my supervisor, doc. Ing. Jiří Bittner, PhD, for helping me steer in the right direction by giving me tips and his insight on how to proceed. I want to also thank my parents and Kamila Houšková for their moral and physical support, pushing me to finish this thesis in their own way.

Declaration

I declare that I have done this thesis all on my own and that I have cited all the sources I have used in the bibliography.

Prague, 24th of May 2023.

Signature:

.....

Abstract

In this bachelor thesis, we will focus on performance optimization in the Unity Engine. We will look up and document the commonly used game performance optimization techniques and how they are implemented in the engine. Then we will test the Visibility Culling and Levels of detail optimization techniques in different game environments to determine how they impact the performance.

Keywords: Performance Optimization, Unity Engine, Occlusion Culling, Levels of Detail, Normal maps, Parallax maps, Draw call batches

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

V této bakalářské práci se budeme zabývat výkonnostní optimalizací v Unity Enginu, podíváme se a zdokumentuje nejčastější používáné metody, a jak jsou implementovány v enginu. Následně otestujeme účinnost optimalizačních metod: Redukování podle viditelnosti (Occlusion Culling) a Levels of detail v různých herních prostředí, abychom zjistili jejich dopad na výkon hry.

Klíčová slova: Výkonnostní optimalizace, Unity Engine, Occlusion Culling, Levels of Detail, Normalové mapy, Parallax mapy, Zobrazovací dávky

Překlad názvu: Výkonnostní optimalizace v Unity

Contents

Figures

Tables

Chapter 1

Introduction

A critical aspect of games, and software in general, is their performance, which can vary depending on many factors, for example, the hardware on which the software runs, the computational cost or the render difficulty. The game must run smoothly to ensure a satisfying player experience; in other words, optimising its performance is fundamental to its success and is an integral part of game development. Fortunately, most primary optimising techniques are built in the Unity Engine and applied during build.

1.1 Motivation

I chose this topic because I sometimes had problems with poorly optimised games. It is also a common discussion topic when new games are released in horrible states where the optimization is lacking, resulting in atrocious frame rates, bugs and crashes. And as an aspiring game developer, I wanted to learn some basics of game optimization for my future projects.

1.2 Thesis Structure

This thesis is divided into two main parts. In the first part, we will document the most popular optimization techniques: both types of bump maps, draw call batching, all aspects of visibility culling and the level of detail technique. In the second part, we will create different scenes and game environments, where we will look and test how occlusion culling and level of detail optimization techniques impact the performance of the scenes.

1.3 Related works

Rendering optimization is a popular topic for research in computer graphics. Their goal is to find methods that help render massive, complex meshes and environments. Both Visibility culling and level of detail, the techniques we will focus on and test, are mainly used for this purpose, as the implementation of their algorithms can be sophisticated but can help achieve a smooth performance.

In this thesis, we will use only discrete levels of detail, meaning we have predefined models that Unity switches when needed. Other types of LOD are progressive and continuous. The latter is the object of research to find algorithms that can render real-time continuous levels of detail. For example, the work of Peter Lindstrom et al. [1].

1.4 Unity Engine

The Unity Engine is a multi-platform game engine developed by Unity Technologies. The first version was released in 2005. The engine is used for 2D and 3D game development for all platforms, such as PCs, consoles and mobile devices.

Chapter 2
Documentation

The Unity Engine has several built-in methods to help optimise game software. During build, Unity performs basic optimization for the target platform. Other methods need to be enabled or baked in the editor. Each method is well documented in the Unity manual.

2.1 Profiling

Profiling helps the developer with finding bottlenecks that lower performance. Which helps furthermore optimise. It is recommended to profile from the early stages of development to the finished product, as it can help tackle performance issues much faster.

When profiling in the Unity editor, results are relative but serve as a good idea of how the game will perform. Profiling on the target platform is more critical, considering the highest and lowest spec devices.

Instead of assuming performance issues, the Unity Profiler can more accurately detect the source. It also shows what exactly is happening in each frame.

2.1.1 Profiler

The Unity profiler is instrumentation-based. It profiles code timings explicitly wrapped in ProfilerMakers and helps detect causes of bottlenecks or freezes during run-time; see screenshot 2.1.

The profiler can monitor CPU usage, GPU usage, rendering, memory, audio, video, physics (3D and 2D), network, UI, global illumination and virtual texturing. These profiler modules can be enabled and disabled as needed, tracking only those critical for the game. Unnecessary module recording can impact performance and affect the results.

I will be recording only the CPU and GPU usage and rendering to test occlusion culling.

In the detailed part of the profiler, the hierarchy, we see the different API calls in the selected frame. We can also find out if our game is CPU or GPU bound, which means that one unit needs to wait for the other to continue execution. If a game is CPU bound, the GPU waits and vice versa. The

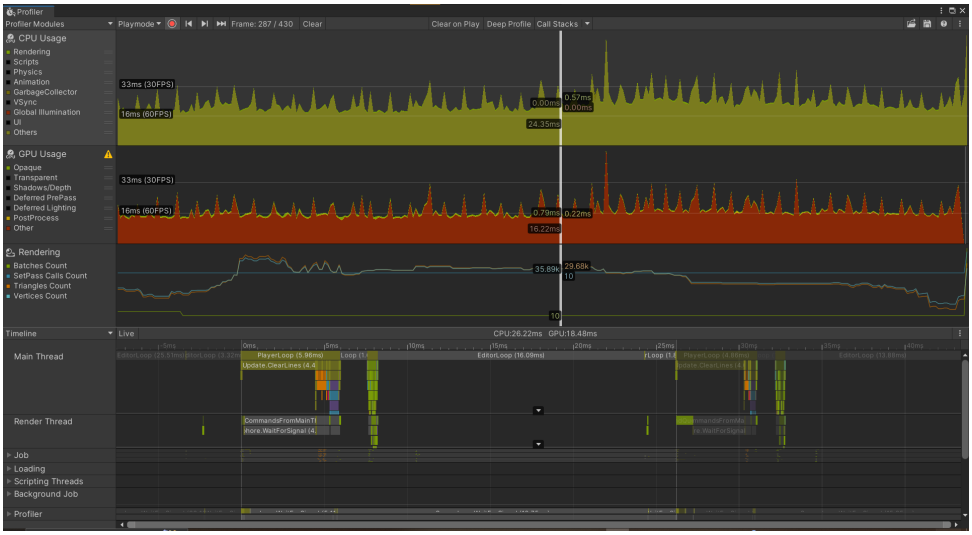


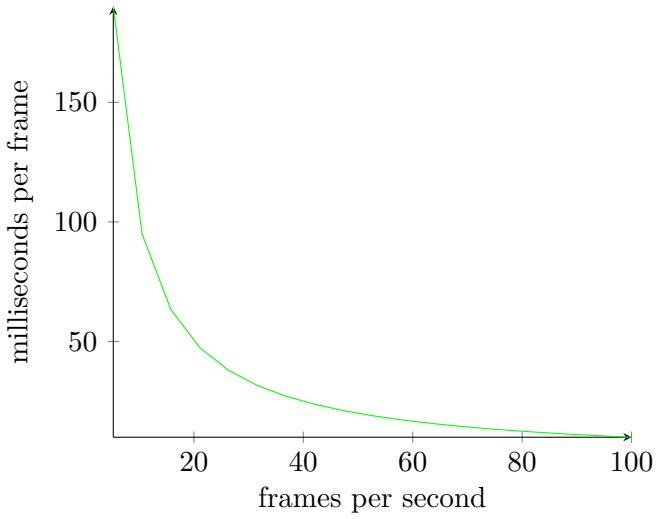
Figure 2.1: The Unity profiler window

gfx.waitforpresent API call, when the CPU is ready to render the next frame but needs to wait for the GPU to present the next frame in the profiler, will reveal whether the computer is GPU bound.

2.1.2 Frame Time

The frame time is the length of a frame in milliseconds. We look at the CPU and GPU time to determine the frame time and choose the higher. It is more accurate than frames per second as it shows how many frames can fit in a second if each frame were the same length. To determine the frame time from the FPS, we want to achieve, we divide 1000ms by the target FPS, as shown in this equation 2.1.

$$t_{cpu} = \frac{1000ms}{FPS} \tag{2.1}$$



For the three most popular target FPS counts, the frame times are defined as follows:

60fps -> 16ms;
30fps -> 32ms;
(VR) 90fps -> 11ms .

With the frame time being more accurate, developers prefer to benchmark it over the frames per second.

2.2 Normal maps

Normal maps are a type of bump maps used in computer graphics to simulate light reflections and shading of high-detail geometry on simple meshes. Rendering small details on geometry is very resource costly. We can use normal maps to simulate these details on a simpler mesh geometry.

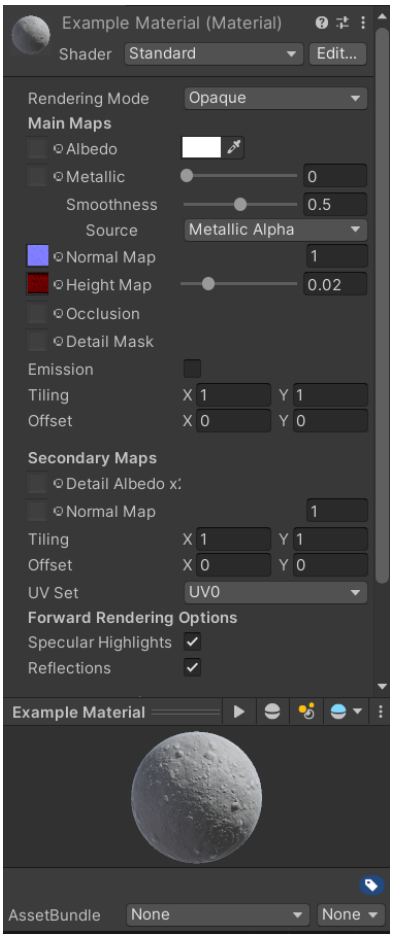


Figure 2.2: Screenshot of the material settings.

These blueish textures store the normal vectors of a mesh. The RGB values represent the XYZ coordinates of these normals. To ensure that all directions can be stored inside normal maps, each of the coordinates of the normal vector is added one and divided by two. For example, a normal vector of (0, 0, 1), the default value on a flat normal map standing for the Z-axis / up-axis, is stored as (0.5, 0.5, 1.0).

Generally, normal maps are generated in 3D modelling software from a high-detail model. To use them in Unity, first, the image's importer inspector needs to set the texture type as a normal map. Then the texture can be added to the normal map slot of the desired material; see Figure 2.2. We can see the normal map slot, where is placed a normal map texture from a terrain layer. And under it, a height map texture of terrain brush is placed in the height map slot.

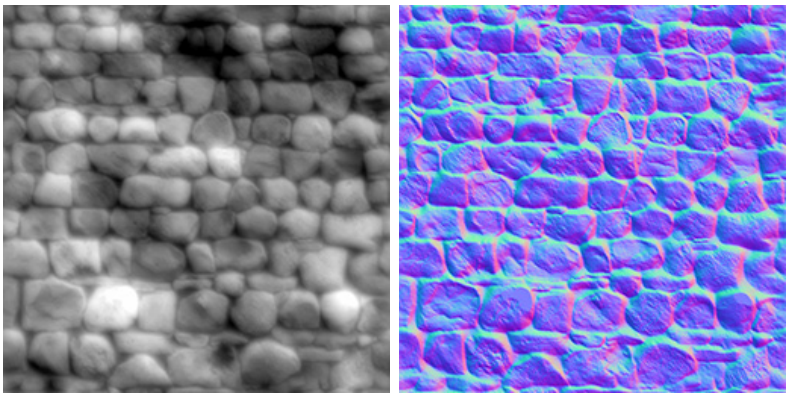
2.3 Parallax maps

Parallax maps, also known as height maps, are a type of bump map. Working similarly to normal maps, they are often used together to create a more realistic look of details on a flat mesh. The difference is the values these height maps represent. They are also more

complex and more performance expansive than normal maps.

These textures shift the visible parts of the surface texture by expanding the closer side to the camera of the bumps and reducing the sides facing away, simulating a surface-level occlusion effect of a 3D geometry. The data is stored in grayscale, where the light areas represent high areas and the dark areas the low areas of the texture.

Using them in Unity is similar to normal maps with these different settings, marking the texture source as grayscale. Instead of putting them into the normal map slot of the material, there added them into the height map slot just under the previous one.



(a) : Height Map **(b) :** Normal Map

Figure 2.3: Difference between a Height map and a Normal map of a rocky wall [6]

2.3.1 Occlusion maps

Occlusion maps are very similar to parallax maps. They serve as a complement to them, representing the areas, in grayscale, that should receive full indirect lightning and areas that shouldn't.

2.4 Draw call batching

Batching is an optimization technique where Unity combines meshes of GameObjects using the same material to render them in fewer draw calls. Unity renders the content of each frame by dispatching a draw call to the graphics API. This process takes time and resources, and sometimes the preparation for the call is more expensive than the call itself. Render state is the settings Unity sets on the CPU and GPU, allocating resources for the next draw call.

Unity has built-in two types of batching: static for GameObjects marked as static; and dynamic for moving GameObjects. All the rendering pipelines in Unity support static batching; the High Definition Rendering Pipeline does

not support dynamic. Both types batch the same type of renderers, Mesh renderers with mesh renderers etc. Renderers supported are Mesh Renderers, Trail Renderers, Line Renderers, Particle Systems, and Sprite Renderers.

2.4.1 Static batching

Unity creates vertex and index buffers for combined meshes transformed into world space. After which, it proceeds with a series of small draw calls, not reducing their number but reducing the number of changes of render states between these draw calls.

Static batching is mainly used with geometry already present in the scene but can also be prepared for generated meshes during runtime, involving the StaticBatchingUtility class. The geometry present at build time needs to meet a list of requirements for the batching to work with.

To use this optimization technique during build time, it must be first enabled in the project setting of the editor and then also enabled in the static editor flags in the inspector of the chosen GameObjects to participate in the batching.

2.4.2 Dynamic batching

Targeted more at old and low-end devices as for the current modern consumer hardware, this type of batching takes more resources and time than the draw calls.

Dynamic batching transforms the vertices into the world space of moving GameObjects on the CPU instead of the GPU. This can have a reverse effect, so it is essential to profile the application to determine whether this technique is helpful.

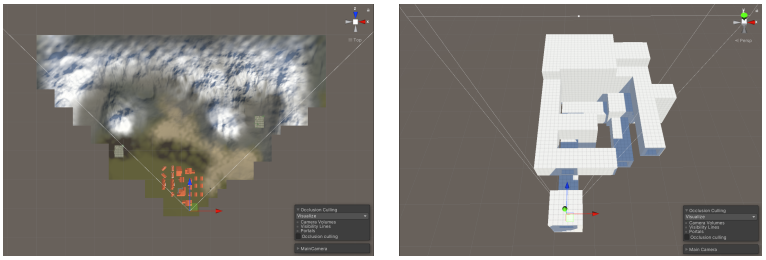
It is also enabled in the project setting of the editor. For dynamic geometry, such as the particle system, dynamic batching is by default enabled, so Unity will always batch it.

2.5 Visibility culling

Visibility culling is an optimization technique that saves rendering performance. It happens for each camera in the scene, where it renders only what can be seen. There are two types of visibility culling that Unity uses: one which doesn't render GameObjects outside of the view frustum, called frustum culling, and one which doesn't render GameObjects that are hidden behind other objects, called occlusion culling.

2.5.1 Frustum Culling

Frustum Culling is performed automatically on every camera. The culling is done by layers first, rendering only the GameObjects on the layers the camera uses, then removing any GameObjects outside the camera frustum.

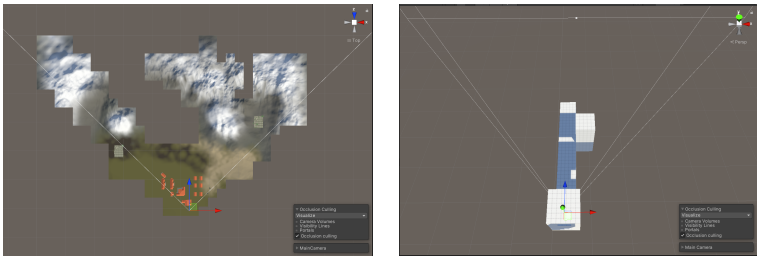


(a) : Demo 1 **(b) :** Demo 2

Figure 2.4: Example of Frustum Culling

Unity recommends organizing GameObjects into different Layers. There is a maximum of 32 layers, each of which can be assigned a value less than the farClipPlane in the layerCullDistances array. There is also a possibility to manually set per-layer culling distances using Camera.layerCullDistances, which allows for culling objects closer to the Camera than the default farClipPlane.

2.5.2 Occlusion Culling



(a) : Demo 1 **(b) :** Demo 2

Figure 2.5: Example of Occlusion Culling

Occlusion culling removes any GameObjects occluded by other GameObject, such as those that the Camera doesn't see. Thus lowering the render cost even more after frustum culling. On the other hand, it is a baked process.

Unity bakes the data during build. This data, which takes some disk space, and costs CPU time and RAM access, can also be baked manually in the editor. This way, the developer can control the parameters to ensure the best performance at a low data size.

Unity uses this data during run time to determine what the Camera sees. The scene is divided into small cells, and data is generated describing the geometry within the cell and visibility between adjacent cells.

2.5.3 Occlusion Data

For occlusion culling to work, we need first to bake the data that Unity will use to calculate what to render and what not. In the baking window, see

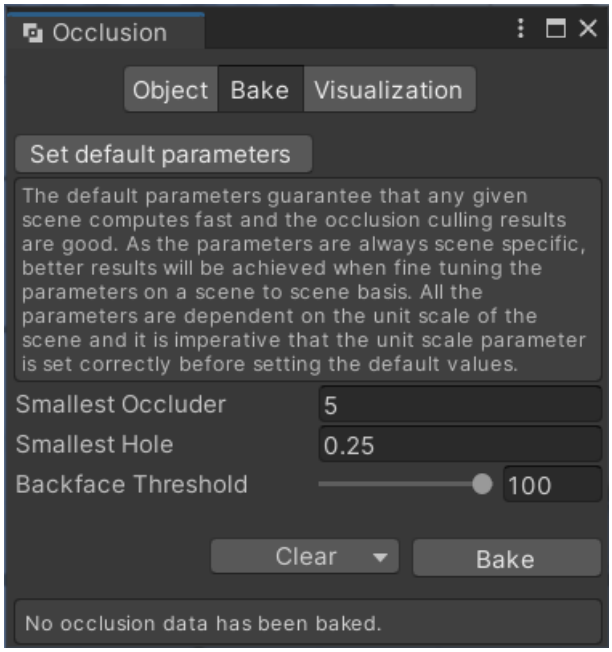


Figure 2.6: Occlusion Baking window with the three parameters, set at default values

screenshot 2.6, three different parameters can be set. Each impacts how fast the data will be baked, how much size the data will take and how the rendering will be affected.

The first parameter, Smallest Occluder (default set to 5), indicates the size, in meters, of the smallest GameObject that can occlude other GameObjects. The smaller the number, the longer the data need to be baked, and the bigger the data size on the disk will be. On the other hand, fewer objects can be rendered, as even a tiny object can occlude objects behind it.

The second parameter, Smallest Hole (default set to 0.25), indicates the diameter, in meters, of the smallest hole through which the Camera can see.

For both of these parameters, in general, for the smallest data file size and fastest bake times, we need to find the highest number that gives us the best render and performance result.

The third and last parameter, Backface Threshold (default set to 100), can be set smaller if we need to reduce the size of the baked data. But it can lead to visual artefacts. It denotes the limit percentage of backfaces a visible occluder geometry can have so it is not discarded. The default value of 100 never removes any area.

As the baked data works only in the scene it was baked, we can fine-tune each parameter in the different scenes such that they run smoothly. A larger scene would need a higher parameter for the smallest occluder than a smaller scene. It also varies on the GameObjects present in each scene. This means each scene needs to have its occlusion culling data.

2.5.4 Occlusion Area and View Volumes

Occlusion Area is a component, generally on an empty GameObject, that defines View Volumes in a scene. View volumes are cuboids in the scene representing an area where the Camera will likely be during runtime.

This helps during the baking process of occlusion culling as Unity will generate more precise data inside these areas and perform higher precision calculations when the camera is inside these view volumes — saving the need to have the same precise data in the whole scene, where the camera is less likely or unlikely.

Unity will generate view volumes during the baking process if no occlusion areas are defined in the scene. This can lead to unnecessarily large data sizes, longer baking times and resource-intensive calculations during runtime in more complex or large scenes.

2.5.5 Static and Dynamic objects

Now that we covered all aspects of visibility culling in Unity let's look at how it affects GameObjects in the scene.

Both static a dynamic objects are affected by the culling. But the baking process of occlusion culling will consider only GameObjects marked as static. In other words, only static GameObjects can be occluders.

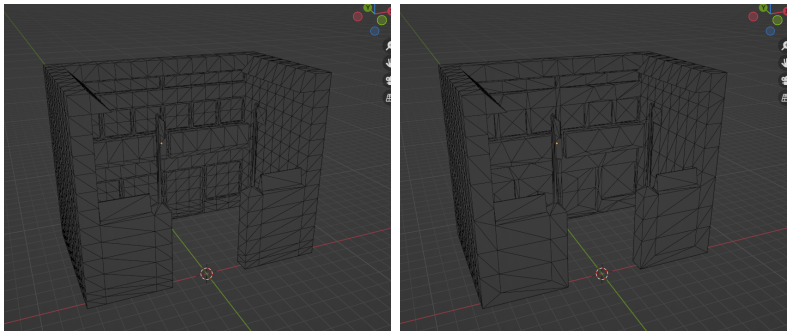
2.6 Levels of detail

Level of detail is a rendering optimization technique that reduces the number of GPU operations required to render distant meshes. The main idea is to reduce the number of triangles rendered of a mesh depending on its distance from the camera. For this, we create from high-detail object copies that have decreasingly fewer faces, called LOD levels. This type of LOD technique is called discrete LOD. We then have an ordered sequence of the different levels with decreasing resolution and accuracy. The corresponding mesh is selected from the sequence depending on the application's needs.[2] In Unity, as the camera moves further away, the high-detail mesh is replaced by lower-resolution meshes. Thus saving GPU operation and helping the performance of the scene.

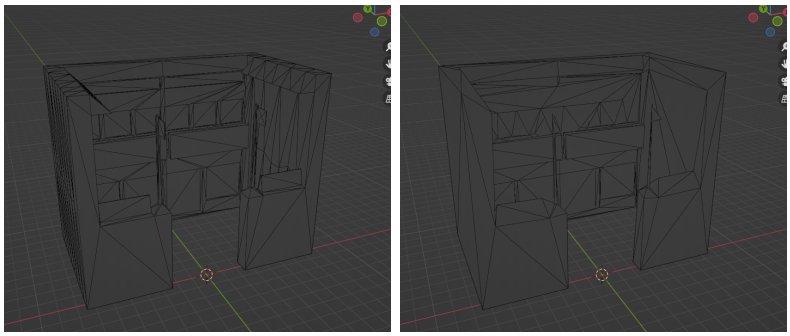
2.6.1 LOD group

To work with LODs in the engine, we must import all the levels either bundled together within the 3D modelling software or each level as separate models.

In the first case, Unity will automatically recognise it as a group of LODs and create the required GameObjects and their components. In the other case, all this configuration needs to be made manually by the user by adding a GameObject and adding the LOD group component. In both cases, the



(a) : LOD0 **(b) :** LOD1



(c) : LOD2 **(d) :** LOD3

Figure 2.7: The 4 Levels of Detail in Blender of the Spawn model used in Demo 4.

user can manually tweak the parameters and configurations of the LOD group component. See Figure 2.8 for the LOD Group component in the inspector.

Such as the thresholds that determine which LOD model to use. The thresholds are set as a percentage ratio of the GameObject's height and the screen's height. These percentages should be fine-tuned to minimise the pop-up effect. The effect is produced when the change of LODs is visible. For each level, where LOD 0 is the most detailed LOD level, a box of renderers shows the GameObject holding the Mesh used for that level of detail. Typically the GameObjects are children of the LOD group GameObject. Other GameObjects can be added, but Unity will prompt the user to add it as a child of the LOD group GameObject. Unity supports a maximum of 8 levels of detail, the first being LOD0. The object is culled after the camera is too far, and the object takes a tiny percentage of the screen.

In the LOD group component, the user can also select the method of transitioning between different levels, called crossfading, and customise these transitions. Crossfading is a technique Unity uses to minimise poping, visible change in the mesh geometry when switching LODs, by rendering both the current and next LOD levels simultaneously with weighing similar to blending and happens inside the transitions zones.

Figure 2.8: Screenshot of the LOD group component used for the Spawn model from Demo 4

Chapter 3

Test Demos

In Unity, we created two new 3D projects using the core 3D template and the 2020.3.41f1 LTS version of the engine. The first project contains the first demo and its three different large scenes. The second project includes three different demos, each with one scene of a similar type. Each demo will be focused on a different game environment to test their interaction of the baked occlusion culling and the use of levels of detail.

In the first demo, we will study the interaction of the terrain component with occlusion culling. Will the terrain always be rendered, or will it be divided into smaller chunks which will be rendered when needed? And is occlusion culling effective in large open scenes?

The second demo will be more closed and compact to see if the occlusion culling is necessary for small scenes.

In the third demo, we will create a medium-sized scene that the camera will not visit in its entirety, which is ideal for testing the use of occlusion areas.

And finally, the last demo will have a less abstract geometry in the scene to test the use of occlusion culling and levels of detail.

All the demos were created and sampled on my laptop with these hardware specifications; see Table 3.1.

	Dell G5
CPU	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
RAM	32,0 GB
OS	64bit
GPU	NVIDIA GeForce GTX 1060 with Max-Q Design
VER	Direct3D 11.0
VRAM	6043 MB
DRIVER	31.0.15.3161

Table 3.1: Hardware specifications of my Dell G5.

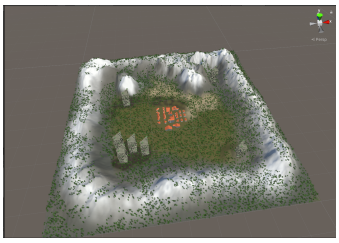
3.1 Demo 1 - Large terrains and Culling

The first demo will test the interaction of terrain GameObject and Occlusion Culling.

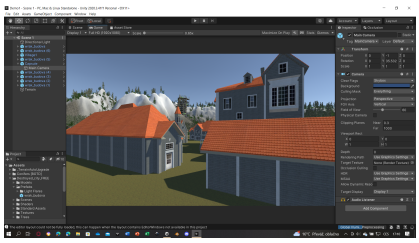
Each scene has the default setting for the camera component. After entering the play mode, the Camera will follow a predetermined path. This way, we can compare the performance from the same sample. The only scene where the Camera doesn't follow a path is the second one because of the large size of the terrain.

3.1.1 Scene 1

This scene will serve as a basic test of how the occlusion data works with the terrain. The terrain tile size, the camera setting and the parameters for baking the occlusion data are all set to default values.



(a) : Scene view



(b) : Game view

Figure 3.1: Demo 1, Scene 1, screenshots of the scene setup

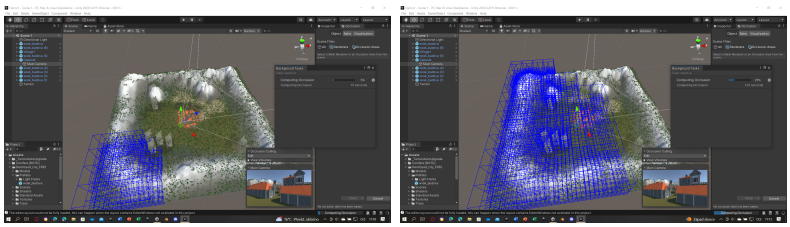
The terrain is modified to create a valley surrounded by mountains. We used the tools and assets from Terrain Sample Asset Pack to achieve that. Inside this valley lies a village created using houses from the House Pack asset bundle downloaded for free from the Unity asset store. We placed some destroyed skyscrapers from the Destroyed City asset bundle in three corners of the mountain range. See Figure 3.1 for an overview of the scene.

As we mentioned before, we baked the data using the default values. During the baking process, see Figure 3.2, Unity started to create view volumes, represented as blue wired boxes, from the scene's origin and covering the whole scene. This process took 30 minutes to complete and resulted in a baked data size of 6,5MB.

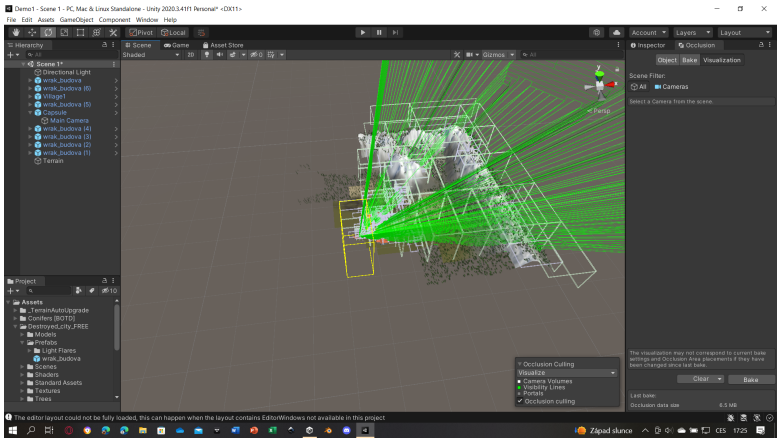
In Figure 3.2 subfigure b), we can see that the scene was divided into small cells by the view volumes. Each cell is rendered if the Camera would see objects in them. The Camera sends rays in the view frustum to calculate which cell it sees. The terrain is also not rendered in its entirety, only the visible parts. The objects behave as expected.

3.1.2 Scene 2

The second scene is much larger than the previous one, as the terrain tile is ten times bigger than the default size. Using the same assets to create a

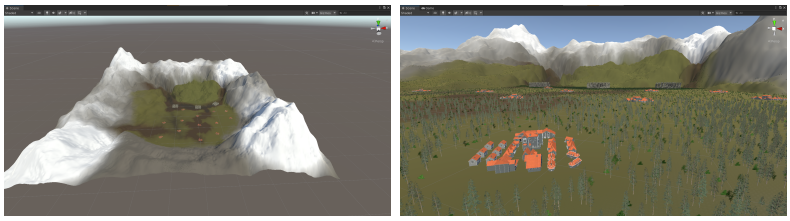


(a) : Beginning **(b) :** Mid progress

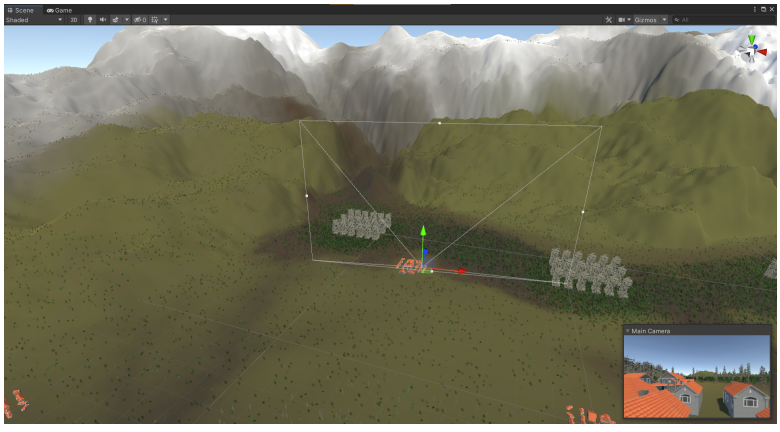


(c) : Result

Figure 3.2: Demo 1, Scene 1, screenshots of different stages of baking the occlusion culling data



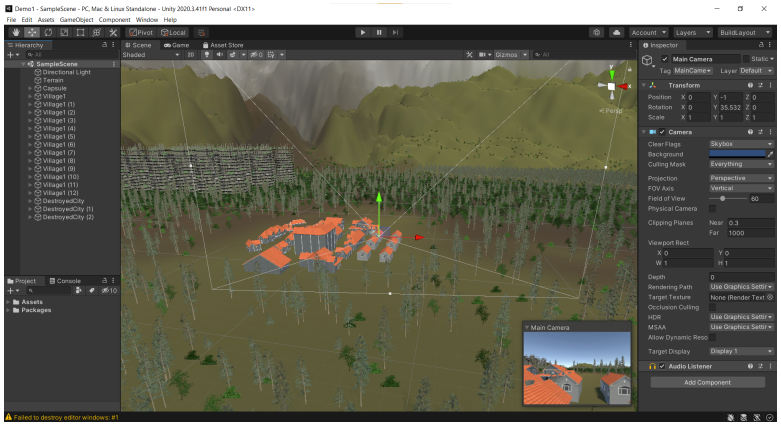
(a) : Overview of the terrain **(b) :** View of the valley



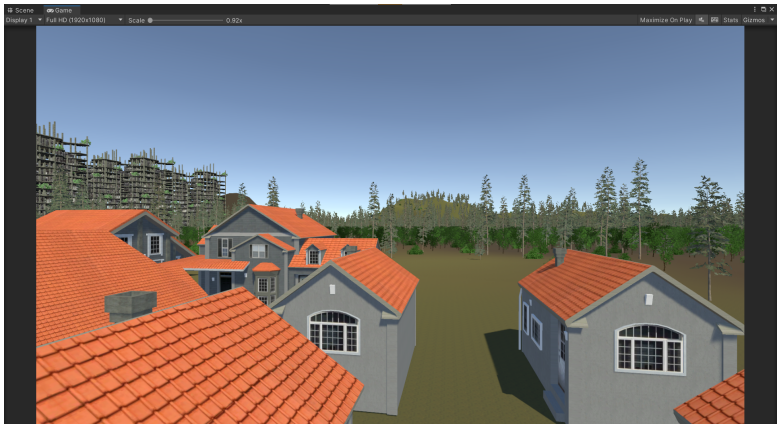
(c) : View with the location of the Camera

Figure 3.3: Demo 1, Scene 2, screenshots of the terrain and buildings

vast valley surrounded by mountains, inside this valley lies several villages with the same assets and house positioning as the one in the first scene. Complementing the villages are three cities formed by a grid of destroyed skyscrapers. See Figure 3.3 for an overview of the scene.



(a) : Overview of the scene with the inspector



(b) : Game view

Figure 3.4: Demo 1, Scene 2, screenshots of the camera setup

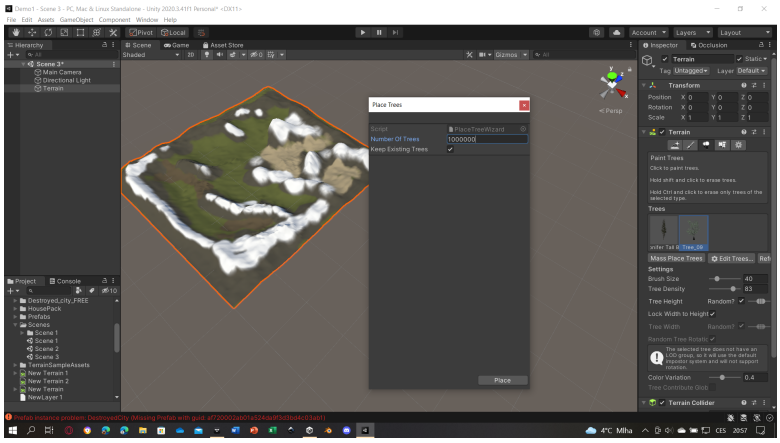
The Camera is placed on the outskirts of one of the villages. It is moved in the air to have a better view. We are using the default setting of the camera component. The Camera sees some of the houses in the village in the foreground and the background: destroyed skyscrapers on the left and a hill in the centre.

In this first version of the scene, on the default setting of the occlusion parameters, the baking process took more than 11 hours and resulted in an error. After this, we tried greater numbers for the smallest occluder parameter, yet even with values like 100, 500, etc..., the baking process took still too much time and resulted in the same error, as there was not enough space to calculate and store the data.

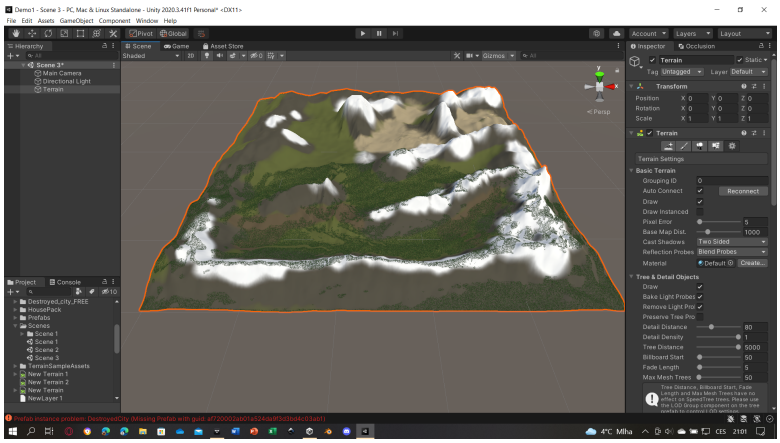
After several tries, we concluded this large-scale terrain was too big for my computer to bake the occlusion data.

3.1.3 Scene 3

I reduced the terrain size in the third scene compared to the terrain tile in the second scene. This time the terrain tile is only five times bigger than the default size. We used the Unity tree wizard to mass-place 1000000 trees; see Figure 3.5. The top image shows the Mass place tree dialogue window poped-up from the tree settings of the terrain component, and the bottom image shows the result of mass placing trees. The terrain is more varied, with several valleys and mountain ranges. But without any other assets. It is only a varied terrain with a lot of trees. These trees come from the Conifers[BOTD] asset bundle.



(a) : Mass place trees dialogue window



(b) : Scene view

Figure 3.5: Demo 1, Scene 3, screenshot of tree placing

This scene had similar problems as the second. The terrain was still too large. Even with the smallest occluder set to 500, the baking process resulted in an error. The only occluders present in the scene would be the mountain ranges.

3.1.4 Results

The terrain tile is divided into small tiles, which are then culled when necessary. Other GameObjects, like the trees and buildings, behave as expected. Occlusion culling on large terrains takes too much time to bake, and the data takes too much place on the disk. As Unity default uses frustum culling, it is the only effective culling on extensive plains. It also shows that occlusion culling will be more effective in smaller-scale scenes or with a high density of occluding objects.

3.2 Demo 2 - Small labyrinth and Culling

Considering the conclusion from the first demo that a more fitting scene would be a smaller and dense one, we created a labyrinth-like structure of rooms of different sizes and narrow corridors connecting them for the second demo. The narrow and short corridors are perfect for testing the occlusion culling.

The first attempt to create such a scene was using the ProBuilder tool from the Unity package manager. We made a simple complex with rooms of different sizes connected by corridors. After completion, we baked the occlusion using the default values. Upon hitting play when the data was calculated, nothing was culled, even if it was occluded. This discovery led to the conclusion that Unity culls whole GameObjects and not only vertices.

On my second attempt, we made small parts of corridors using the ProBuilder tool. These gave me more freedom and prefabs to assemble a labyrinth-like two-floored building. At some end, we created unique large rooms.

The camera travelled through most of the building, starting and ending its run at the same position and often changing directions and floors to constantly switching what was culled.

This scene's small scale will help me study the impact of each parameter used for occlusion culling baking.

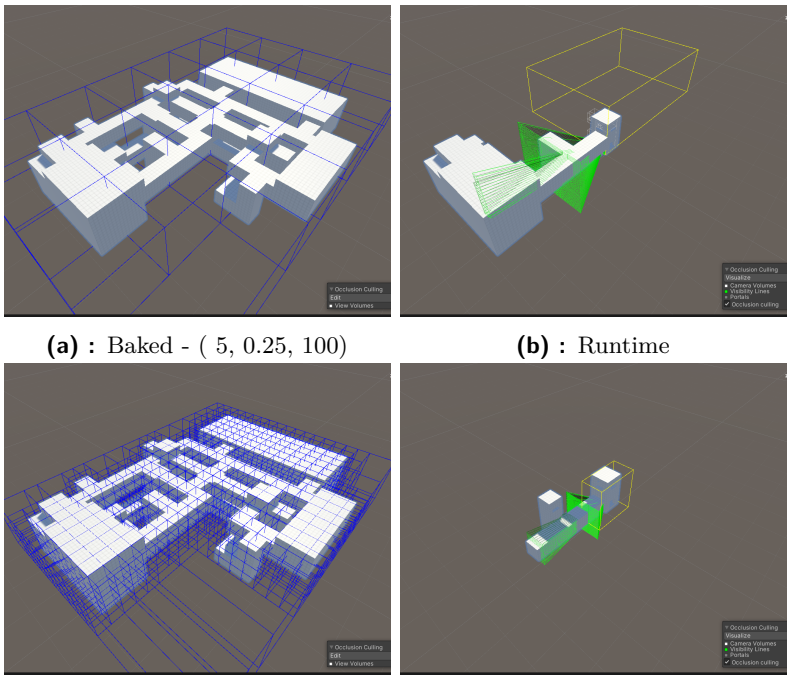
3.2.1 Profiling

In the main thread, we looked for the gfx.waitforpresent API call, and we found it, which means this demo is GPU bound.

In the table 3.2, we can see the impact of each parameter on the size of the bake data file. The file size is more significant when the first two parameters are set to a smaller value. For my scene, the third parameter doesn't impact baking or data size. The result of the first two rows can be visualised in 3.6. The smaller size of the smallest occluder creates smaller view volumes, the blue wired boxes, resulting in fewer parts of the scene to render.

SO	SH	BFT	ODS	
5	**0.25**	**100**	63.9KB	
1	**0.25**	**100**	416.7KB	SO = Smallest Occluder
1	0.10	**100**	367.9KB	SH = Smallest Hole
5	0.10	**100**	79.2KB	BFT = Backface Threshold
8	0.50	**100**	16.9KB	ODS = Occlusion Data size
9	**0.25**	**100**	19.3KB	

Table 3.2: Demo 2, Occlusion Baking Parameters. Default values are bolded.



(a) : Baked - (5, 0.25, 100) **(b) :** Runtime

(c) : Baked - (1, 0.25, 100) **(d) :** Runtime

Figure 3.6: Demo 2, Visualization of the Occlusion Parameters

3.2.2 Test 1 - Single frame sampling

I ran the scene with the profiler connected and recorded performance data for each baked data. We chose to look at the 417th frame of each sample, writing down the CPU and GPU time of the frame. The frames per second were calculated by using the equation 3.1 derived from 2.1

$$FPS = \frac{1000}{t_{cpu}[ms]} \tag{3.1}$$

and rounding down the result to a whole number. Results are in the 3.3 table. After occlusion culling, the FPS count increased from 44 to 91 using the default values. Because the FPS differs for each data, the 417th frame is different. When the FPS is higher, the frame will come earlier. It also shows how effective the occlusion culling is with the right parameters, dividing the frame time by 2. The second conclusion is that smaller values don't mean

better results, as the engine needs to calculate the data more often, frequently changing the rendered game objects. This support the hint given by Unity in its documentation to find the highest numbers for these parameters that achieve the best performance results.

SO	SH	DS [KB]	CPU [ms]	GPU [ms]	FPS
-	-	-	22.25	11.68	44
5	0.25	63.8	10.93	6.96	91
1	0.25	416.7	14.84	11.57	67
1	0.10	367.9	13.51	9.65	74

SO = Smallest Occluder, SH = Smallest Hole, DS = Data Size

Table 3.3: Demo 2, Test 1, Occlusion and Profiler Statistics - 417th frame

3.2.3 Test 2 - Single location sampling

Our second approach was to choose a location on the Camera's path, where a script marks the frame number, which we then analysed in the profiler. This way, we compared stats of a different frame but at a similar place. The number of frames, with the FPS, shows how, when optimising, more frames are rendered in a defined time.

In the table 3.4, we have written more statistics; the occlusion parameters and the size of the data taken on the disk; the frame that we analysed; the time that both processing units take to process their commands in the frame; the calculated FPS from the frame time; and from the rendering module the number of vertices rendered in the frame and the number of draw calls.

SO	SH	DS [KB]	Frame	CPU [ms]	GPU [ms]	FPS	VC	DC
-	-	-	818	19.25	19.12	51	18.3k	14
7	0.25	29.6	826	19.58	19.26	51	18.3k	14
5	0.25	63.8	834	19.43	19.64	50	18.3k	14
3	0.25	123.4	841	20.38	19.32	49	18.3k	14
1	0.25	416.7	835	31.53	21.62	31	18.3k	14
1	0.10	367.9	841	19.41	19.15	51	18.3k	14
3	0.50	122.0	815	19.96	19.22	50	18.3k	14
5	0.50	47.6	840	19.31	19.15	51	18.3k	14
5	1.00	34.8	835	19.30	19.13	51	18.3k	14

SO = Smallest Occluder, SH = Smallest Hole, DS = Data Size
VC = Vertices Count, DC = Draw Calls

Table 3.4: Demo 2, Test 2, Occlusion and Profiler Statistics

3.2.4 Test 3 - Camera path sampling

We took a larger sample in the third and final test using this scene. The camera travels along a predetermined path, with control points along the way. When the camera passes a control point, a script registers the profiler data and prints them into the console, such as frame time, vertices count and draw calls. There are 35 control points in total.

SO	**STATS**	**CONTROL POINTS**							
DS	[KB]	1	5	10	15	20	25	30	END
0	RT [s]	2,46	11,08	30,67	39,39	47,64	62,26	72,49	86,1
0	F#	1	1096	3715	4891	6025	8009	9415	11274
	FT [ms]	0	8,5	8,7	7,8	7,3	7,2	7,7	7,7
5	RT [s]	2,47	11,04	30,69	39,43	47,65	62,29	72,52	86,13
63,8	F#	1	1180	3826	4798	5859	7244	8592	10458
	FT [ms]	0	7,4	7,1	10,3	6,5	12,4	7,7	6,8
1	RT [s]	3,71	14,79	34,43	43,16	51,42	66,04	76,28	89,9
416,7	F#	1	1065	3548	4703	5745	7553	8864	10436
	FT [ms]	0	8,2	6,9	6,3	9,3	10,2	7,7	8,9
3	RT [s]	3	11,56	31,2	39,94	48,18	62,81	73,04	86,65
123,4	F#	1	1134	3923	5148	6327	8359	9702	11571
	FT [ms]	0	6,8	7,7	8	6,8	6,9	8,1	7,2
7	RT [s]	2,51	11,06	30,71	39,44	47,69	62,31	72,55	86,17
29,6	F#	1	1138	3876	5034	6100	8044	9434	11349
	FT [ms]	0	6,8	7,4	9,1	8,9	6,9	6,4	7,5
4	RT [s]	2,45	11	30,64	39,37	47,6	62,22	72,47	86,09
71,7	F#	1	1186	3997	5207	6360	8297	9691	11549
	FT [ms]	0	6,4	7,5	7,1	8	6,6	8,4	7,7

SO = Smallest Occluder, DS = Data Size
RT = Realtime, F# = Frame Number, FT = Frametime

Table 3.5: Demo 2, Test 3, Table of performances depending on the smallest occluder parameter at selected control points.

I took the data of every fifth entry and wrote it in a table to compare each occlusion parameter's impact on performance. First, we changed only the smallest occluder parameter and left the rest of the parameters at their default values to monitor its impact on the performance, see Table 3.5.

We can see that the best results are for the value of 3, but it had a problem. When the camera travelled through the longest straight corridor, the far wall didn't render, leading to the second-best result for the value of 4, which rendered everything that should. This discovery shows the importance of watching if everything renders as it should and that the culling doesn't cull what needs to be rendered.

So when we were changing only the smallest hole parameter, we used the value of 4 for the smallest occluder parameter as it yielded the best performance and visual results with the default value for the backface threshold

parameter, see Table 3.6 for the impact of the Smallest Hole parameter.

SH	**STATS**	**CONTROL POINTS**							
DS	[KB]	1	5	10	15	20	25	30	END
0	RT [s]	2,59	14,94	34,59	43,32	51,55	66,18	76,41	90,02
0	F#	1	1192	4107	5253	6320	8242	9582	11300
	FT [ms]	0	7,2	7,1	7,2	6,9	7,1	7,8	7,9
0.25	RT [s]	2,41	10,97	30,63	39,36	47,59	62,21	72,44	86,05
71,7	F#	1	1049	3331	4307	5279	7010	8217	9805
	FT [ms]	0	8,3	8,9	9,1	7,8	8,5	8,6	10,3
0.1	RT [s]	2,57	11,19	30,83	39,56	47,8	62,43	72,66	86,29
70,7	F#	1	1059	3432	4497	5508	7226	8357	9868
	FT [ms]	0	8,1	8,4	8,4	8,8	8,7	8,6	9,4
0.5	RT [s]	2,62	11,2	30,85	39,58	47,82	62,45	72,68	86,3
53,1	F#	1	956	3219	4245	5150	6671	7790	9351
	FT [ms]	0	7,7	9,6	8,8	9	8,7	7,8	8,3
0.75	RT [s]	2,4	10,97	30,61	39,35	47,59	62,22	72,46	86,08
44,5	F#	1	1057	3440	4510	5524	7196	8383	9949
	FT [ms]	0	9	8,5	8,5	8,5	8,8	9,2	10,9

SH = Smallest hole, DS = Data Size
RT = Realtime, F# = Frame Number, FT = Frametime

Table 3.6: Demo 2, Test 3, Table of performances depending on the smallest hole parameter at selected control points.

Both tables are also interpreted as graphs see 3.7.

3.2.5 Results

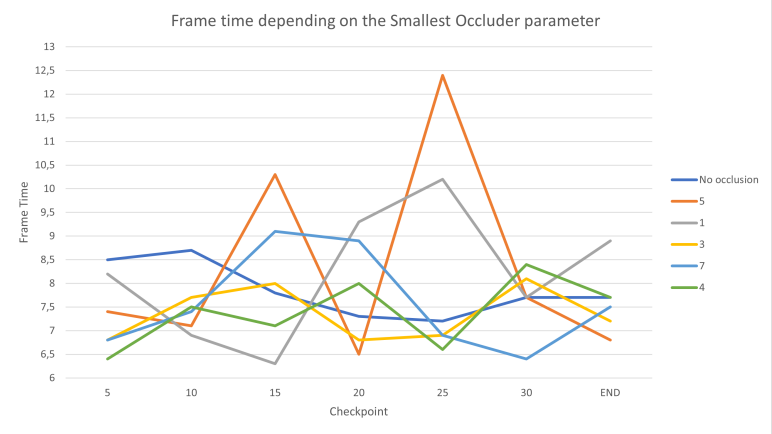
The first test had excellent and acceptable results. But as we wanted more samples some days later, the frame times were significantly lower. This is why the first test has fewer samples than the second.

The second test was made because the results would represent the optimising technique more. But in the end, it didn't go as well as we had hoped; the real difference was only in the frame captured. We can see better the difference in the frame times than in the FPS.

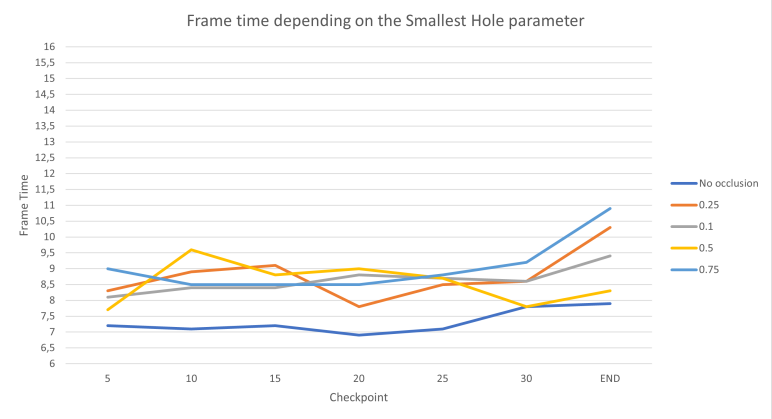
To my surprise, better results and more stable frame times throughout the path were generated in the third test without the occlusion culling. This led me to the conclusion that the scene is not complicated enough. The baked data worsened the stats because the engine calculated unnecessary optimization and often switched the GameObjects that needed to be culled.

3.3 Demo 3 - Medium factory and Occlusion Areas

After creating a large scene with terrain too big for the occlusion culling data to bake and a small scene full of corridors, too small for the occlusion



(a) : Depending on the smallest occluder parameter



(b) : Depending on the smallest hole parameter

Figure 3.7: Demo 2, Test 3, Graphs with the results of the test, comparing the frame times

culling to make a difference, we created a medium-sized scene of a factory. This factory comprises two levels of rooms of different sizes connected with corridors. The data will be sampled the same way as the last test above.

The camera travels a set path and visits only a small percentage of the scene; see Figure 3.8. Starting from the left side in the middle, where an entrance is located, the camera then turns left and goes through a corridor to the first stairwell. Goes up the stairs and into the second block of the factory. In the middle of this block are located stairs which the camera takes to go down to the ground floor of the building. At the end of the block, goes up another stair to go to the back sectors of the facility. Take the last stairs and forward to the vast laboratories where the camera exits the factory. The camera didn't go through the whole building, so there is no need to calculate occlusion data for the entire scene, and we can practice using occlusion areas.

After baking the occlusion data with the parameters of smallest occluder: 3, smallest hole: 0.25 and backface threshold: 100%, the view volumes inside occlusion areas are smaller than those outside them, see Figure 3.9. They are

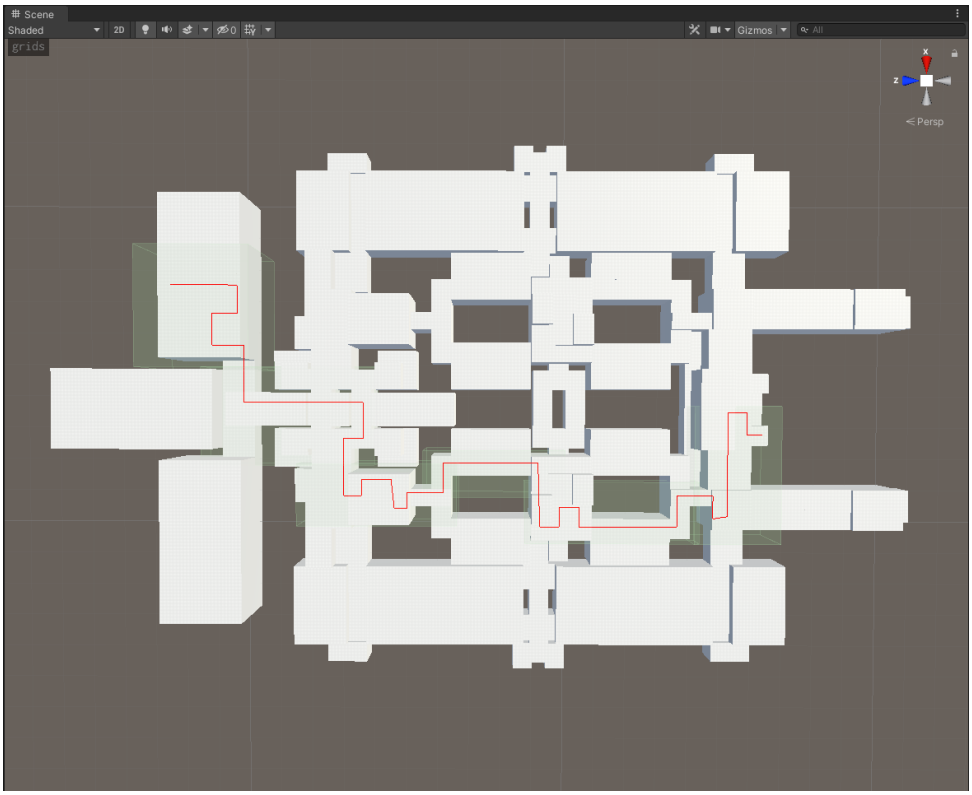
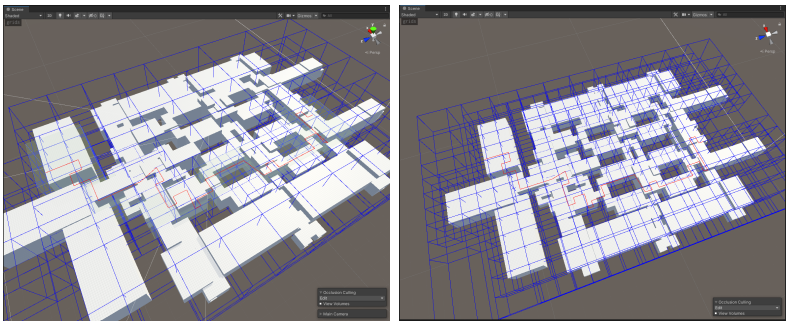


Figure 3.8: Demo 3, Top view of the scene with the path of the camera in red and occlusion area as green boxes



(a) : With Occlusion Areas **(b) :** Without Occlusion Areas

Figure 3.9: Demo 3, View of the scene with view volumes

also smaller than the standard view volumes generated by Unity without the use of occlusion areas. This means the calculations are more precise inside the occlusion areas, as advertised in the Unity documentation.

OPT	**STATS**	**CONTROL POINTS**							
OA		1	5	10	15	20	25	30	35
DS[KB]									
NO	RT [s]	3,3	15,97	28,77	37,17	48,73	58,13	72,94	84,08
NO	F#	1	533	1166	1559	2056	2458	3075	3509
-	FT [ms]		19,8	23,1	22,9	21,5	21,7	23,4	27,4
	FPS		51	43	44	47	46	43	36
	VC [$\cdot 10^3$]	0	27,3	165,5	145	78,9	97,3	29,7	20
YES	RT [s]	2,83	13,15	25,93	34,37	45,95	55,37	70,21	81,38
NO	F#	1	548	1174	1543	2025	2406	2979	3396
276	FT [ms]		19,3	23,1	24,6	22,7	23,4	31	26,2
	FPS		52	43	41	44	43	32	38
	VC [$\cdot 10^3$]	0	25,6	22,9	26,7	26,5	21,7	16,8	20,1
YES	RT [s]	3,02	13,22	26,02	34,46	46,09	55,5	70,34	81,53
YES	F#	1	450	979	1269	1691	2013	2547	2943
255,2	FT [ms]		28,1	35,4	26,7	32,3	27,8	28,8	27,4
	FPS		36	28	37	31	36	35	36
	VC [$\cdot 10^3$]	0	25,6	22,9	26,7	26,5	25	16,8	14,3

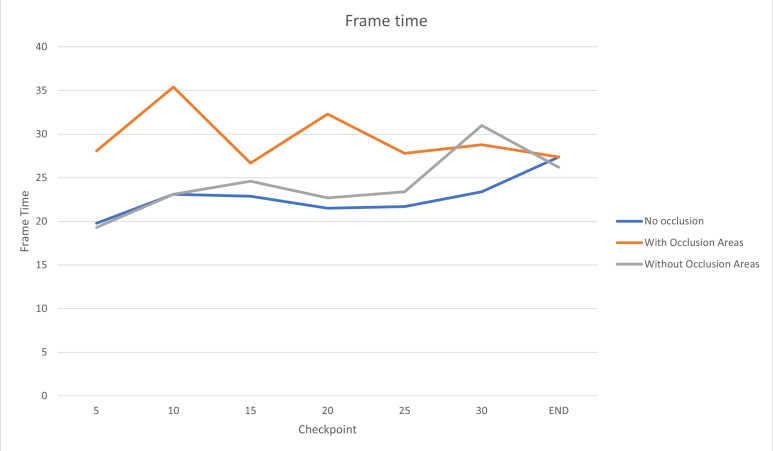
OPT = With optimization, OA = With occlusion areas, DS = Data Size
RT = Realtime, F# = Frame Number, FT = Frametime
FPS = Frame per seconds, VC = Vertex count

Table 3.7: Demo 3, Table of the profiler statistics at selected control points

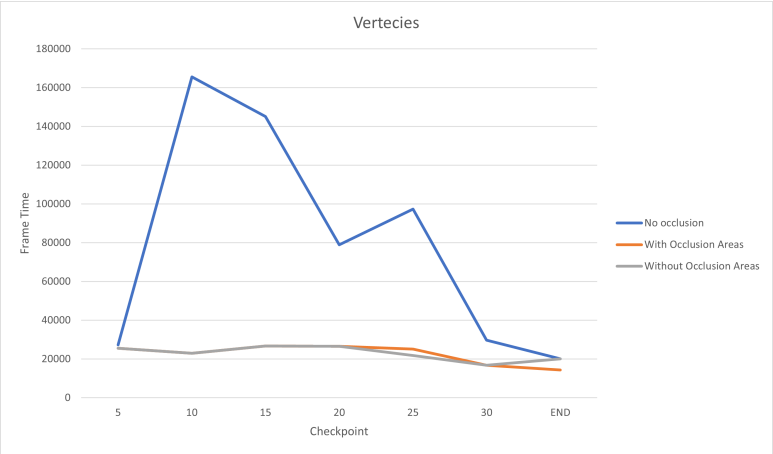
The scene's geometry is still too simple, and the calculations for the culling are more expansive than without the occlusion. But we can see a slight difference between the data size, frame times and vertex count when using occlusion areas, and when not, see the upper graph of 3.10. The size is 276KB without occlusion areas and 255,2KB with. It is better to use an occlusion area to reduce the data size when we know that the camera will be only in a particular area of the scene; we can also see how a considerable amount of vertices are saved by rendering with occlusion culling.

3.4 Demo 4 - FPS map for Culling and Levels of detail

For the last demo, we created a small FPS map; see 3.11. In the scene, we can see facades of buildings delimiting the play area. They are high enough to occlude large parts of the scene. The wide zigzagging main road is complemented by narrower streets that can be used as shortcuts. In several places, there are small inside yards between the facades. Four largely open squares offer more space for fights around the game mode capture checkpoint



(a) : Graph of the frame times



(b) : Graph of the vertices count

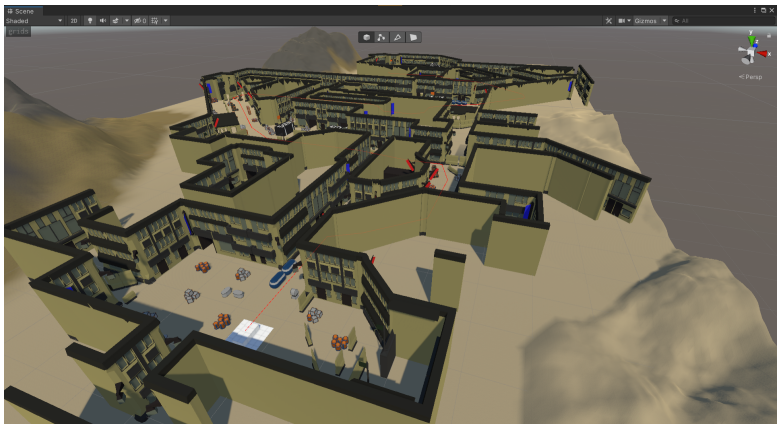
Figure 3.10: Demo 3, Graphs of chosen statistics at selected control points

on the main path of the objective.

We modelled the facades of buildings in Blender with four levels of detail, and most of the small props are from the SNAPS prototype assets package made by Unity. The facades were saved as blend files into the project's asset folder. This way, when the file was saved in Blender, it also updated its geometry in Unity. The second benefit was that Unity automatically created and set up the LOD groups for each model.

The camera travels a predetermined path along the buildings on the main road; see Screenshots 3.11 represented as the red line. It serves as the payload needing to be escorted by the attacking team if the map was fully implemented with game-play features. It is only a static scene to test occlusion culling and LOD optimization techniques.

(a) : Top view



(b) : Side view

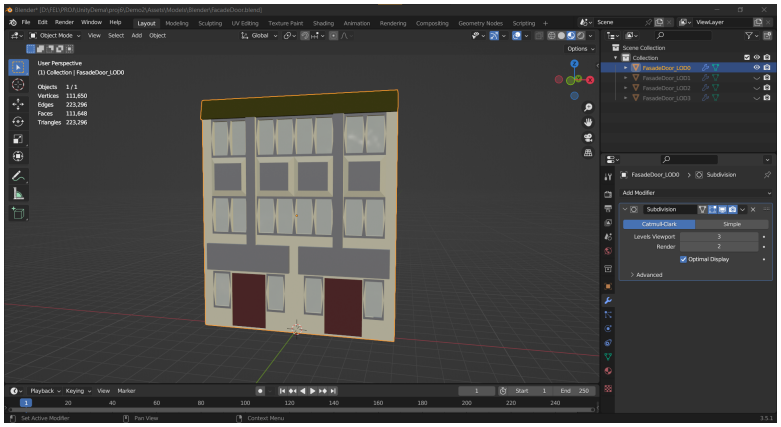
Figure 3.11: Demo 4, Overview of the scene. The path of the camera is represented as a red line

3.4.1 Models and their Levels of Detail

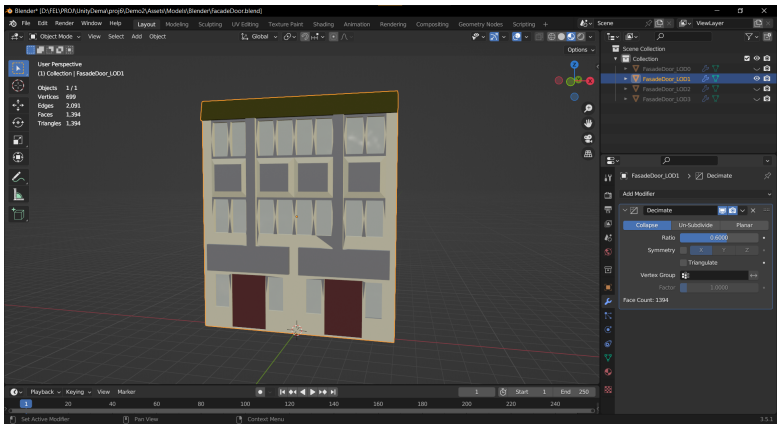
Most models that we modelled for this demo have four levels of detail. The LOD 1 - 3 levels use the decimate modifier from the previous level with a smaller and smaller ratio. These ratios also vary from each model as the geometry is different. Only the LOD 0 level has its face count increased by the subdivision surface modifier to compensate for the lack of other processes costing resources, such as physics, gameplay logic etc.

Let's look at one of the facade models, see Figure 3.12 and its levels of detail. The statistics displayed in the upper left corner are transcribed into Table 3.8 as well as the decimate ratio of the LOD 1 - 3. We can also see little imperfections in these levels of detail created by the materials and the face collapse by the decimate modifier. The fewer details, the more LODs are uglier; finding and fine-tuning the distances at which the models are swapped is essential.

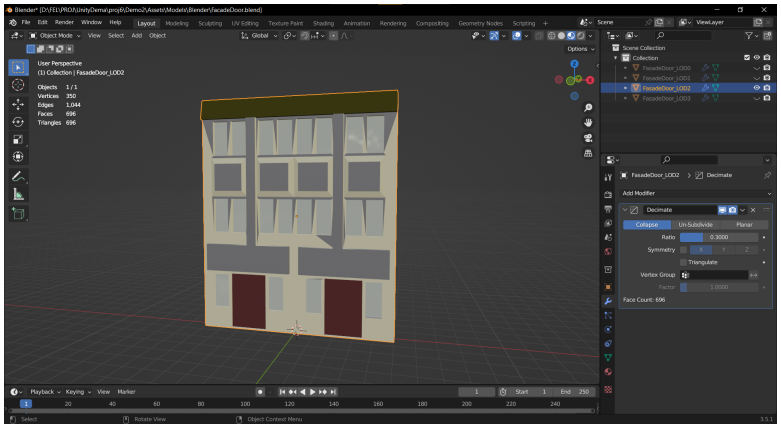
In Unity, the models were represented as a bundle of models with an empty parent object with the four levels as child objects. For each parent object, a



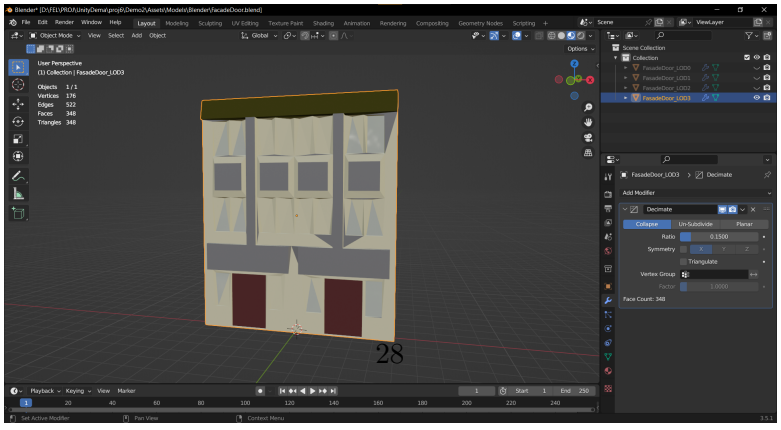
(a) : LOD0



(b) : LOD1



(c) : LOD2



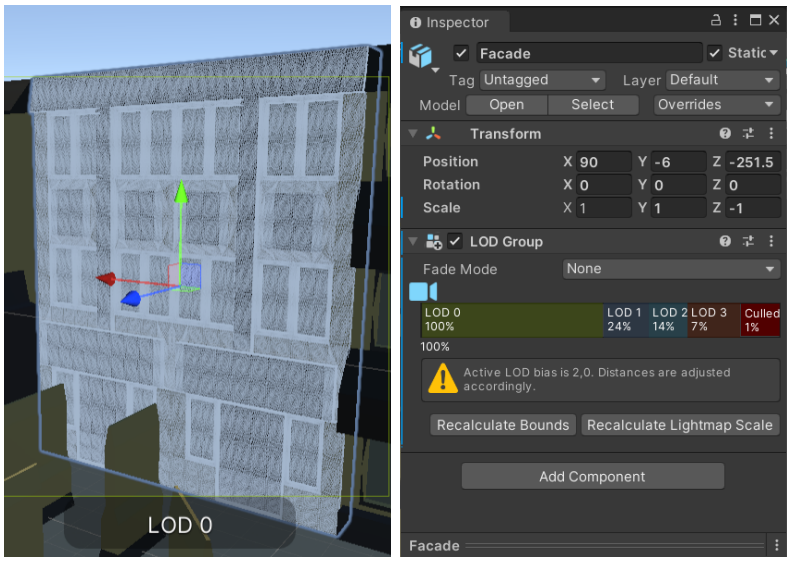
(d) : LOD3

Figure 3.12: Facade model 1 and its different LODs in Blender

LOD	Vertices	Edges	Faces	Triangles	Decimate ratio
0	111650	223296	111648	223296	-
1	699	2,091	1394	1394	0.6
2	350	1044	696	696	0.3
3	176	522	348	348	0.15

Table 3.8: Facade model 1 - LOD statistics

LOD group was assigned, containing at each LOD a model as the renderer that should be rendered when the object takes a certain percentage of the screen — no need to set it up manually. The LOD group cannot be changed inside the asset. But when we create an instance in the scene, we can modify each component of the different GameObjects making up the model as we want. In the scene, the hierarchy of the GameObject remains the same, an empty parent containing the LOD group, see Figure 3.13 and four children, each representing one level of detail. We need to be careful when selecting the objects inside the scene view as it happened to have selected one child / one level, the one that was being rendered, instead of the whole group/parent GameObject.



(a) : View of the model in the scene

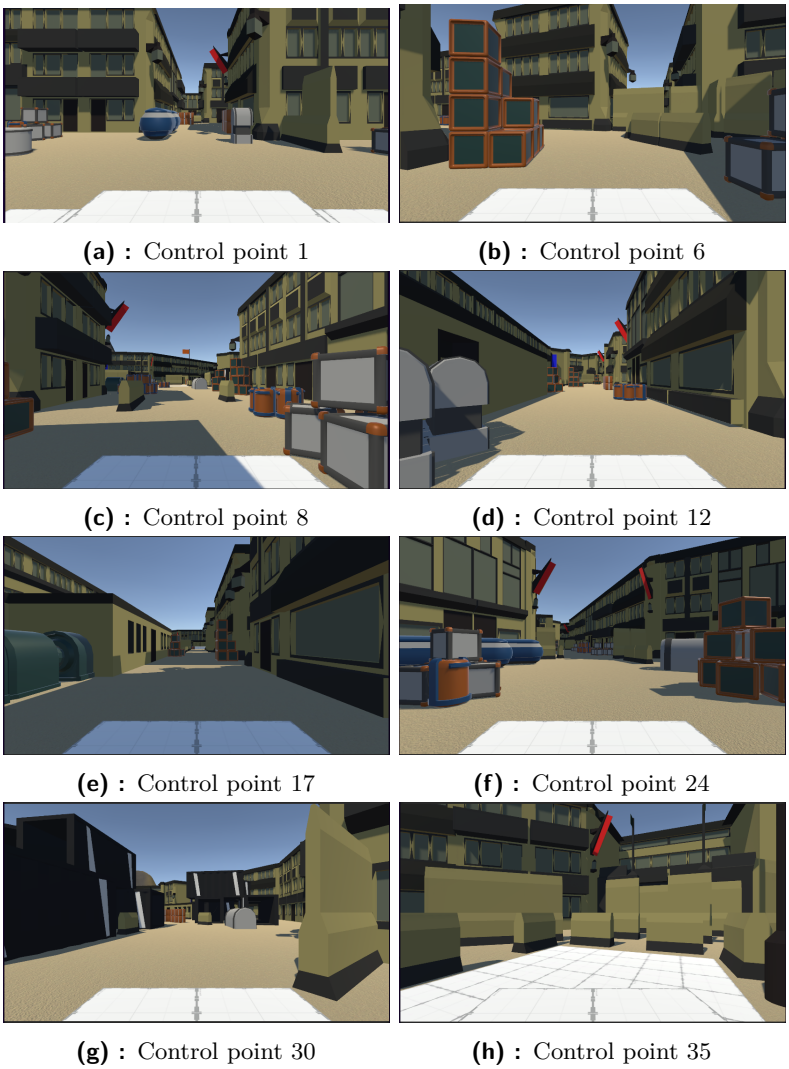
(b) : Inspector with the LOD group component.

Figure 3.13: Demo 4, Screenshots of the Facade model inside Unity.

3.4.2 Test preparations

On this map, we will conduct four tests: one without any optimization, one with LOD, one with occlusion culling, and one with both optimization techniques enabled.

The control points at which we sampled data were changed. Instead of each



(a) : Control point 1 **(b) :** Control point 6

(c) : Control point 8 **(d) :** Control point 12

(e) : Control point 17 **(f) :** Control point 24

(g) : Control point 30 **(h) :** Control point 35

Figure 3.14: Demo 4, Camera view at selected control points

fifth, we chose those with long sights of view to test the LODs; see Figure 3.14 for the camera view at these control points. We presume that at these chosen places, there is a higher probability of being bottlenecks. We can also assume that the performance between them will be more stable and better.

The first control point is located at the start of the camera path. From the camera, we can see the street by which the camera will leave the first square area. At the far end of the street is a building that should be rendered with a lower LOD. The second control point, indexed 6, is in the middle of the far-right street. From this, the street straight is blocked by wall props. Control point 8 is located at the crossing of the main street and the narrow street seen from the first checkpoint. We can see entering the second square with an orange flag indicating the first capture checkpoint. With an index of 12, the following control point is just outside the square entering the main street and providing a long sight with the defender spawn point at the end.

Going near this spawn point, the camera turns right and passes by control point 17, from which we can see in the distance building delimiting the third square with the second capture point. The following sample is taken from the control point 24 just at the exit of the square. Here the camera sees a long street at a 45 angle. The second last sample is taken from control point 30 when the camera enters the last square. In this square is the final capture point, if captured, resulting in a win for the attackers. The square is filled with giant black containers. And finally, the last control point, 35, is located just in front of the last defender checkpoint, which we can see barricaded.

The first run of tests will be on my notebook, the Dell G5 see Table 3.9 for its specification, in the editor. Then, we will rerun the same tests on the Dell G5 and on two other notebooks borrowed from the department; see Table 3.10 for the specifications of the Lenovo Ideapad Flex 5 and Table 3.11 for the specifications of the Lenovo Thinkpad T440s. For the second run of the test, we will build four different builds, one without any optimization, one with LOD, one with occlusion culling, and one with both optimization techniques enabled.

	Dell G5
CPU	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
RAM	32,0 GB
OS	64bit
GPU	NVIDIA GeForce GTX 1060 with Max-Q Design
VER	Direct3D 11.0
VRAM	6043 MB
DRIVER	31.0.15.3161

Table 3.9: Hardware specifications of Dell G5.

	Lenovo Ideapad Flex 5
CPU	AMD Ryzen 3 4300U with Radeon Graphics 2.70 GHz
RAM	8,0 GB
OS	64bit
GPU	Integrated AMD Radeon(TM) Graphics
VER	Direct3D 11.0
VRAM	3774 MB
DRIVER	31.0.12044.24003

Table 3.10: Hardware specifications of Lenovo Ideapad Flex 5

The results of the first run of tests, made on the Dell G5 in the editor, are written in Table 3.12. Here we will compare the impact of each test together. The second run compares the technique on different hardware; the results are in separate tables.

For each test, we will first describe how we prepared the scene and its setting for the test, then we will examine how the technique used in the test improves the performance of the scene on my notebook, and finally, we will

	Lenovo Thinkpad T440s
CPU	Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz 2.70 GHz
RAM	12,0 GB
OS	64bit
GPU	NVIDIA GeForce GT 730M
VER	Direct3D 11.0
VRAM	986 MB
DRIVER	24.21.13.9836

Table 3.11: Hardware specifications of Lenovo Thinkpad T440s

analyse how the build performs on the three notebooks. In the end, I will sum up the results of this demo.

3.4.3 Test 1 - Not Optimised

As Unity automatically created LOD groups for my models, we needed to disable them to run an unoptimised test for reference. But disabling the component had resulted in rendering all levels at the same time. To force the use of only the first level, LOD0, we created a simple script that takes the LOD group component and calls the ForceLOD method. The script was then added to each LOD's parent game object. With that set, we could record the first test. This generated control statistics to compare how the optimization techniques helped improve the scene's performance.

In the editor, my notebook keeps an unstable, but never the less low frame time, with large fluctuations by displaying a large number of vertices.

The build, see Table 3.13, has a more stable performance on the Dell G5 than in the editor, frame time of an average of 20.5 ms, which can be a result of the build-in optimization made by Unity during the build process. Almost stable performance, frame time of an average of 56.5 ms, was recorded on the Ideapad. The worst performance, with high fluctuation of the frame time and an average of 169.5 ms, has the build on the Thinkpad.

3.4.4 Test 2 - Levels of Detail

Before running this test, we disabled the force LOD script and tuned each threshold of the levels. We minimised both the pop-up effect and the vertex count.

The build performed better than when not optimised. We can see in Table 3.14 a stable 16.6 ms frame time overall for the Dell G5. The performance on both the Ideapad and Thinkpad is similar, with less change in frame time during the build run. Though for the Ideapad, the FPS dropped, negating any optimization. The average frame time for the Ideapad is 84,6 ms and 92,6 ms for the Thinkpad.

LOD OC	STATS	CONTROL POINTS 1	6	8	12	17	24	30	END
	RT [s]	2,52	17,72	22,59	37,4	53,3	77,19	100,7	112,16
NO	F#	1	578	758	2073	4045	5191	7215	8867
NO	FT [ms]	0	27,3	20,1	9,6	34,6	15,5	12,3	5,1
	FPS		37	50	104	29	65	81	196
	VC [$\cdot 10^6$]	0	95,89	66,83	26,56	128,47	46,92	33,2	7,12
	RT [s]	2,57	17,81	22,68	37,51	53,37	77,28	100,78	112,25
YES	F#	1	1053	1381	2932	5052	7132	9449	11321
NO	FT [ms]	0	14,1	16,1	9,2	13,4	11,1	8,5	5,1
	FPS		71	62	109	75	90	118	196
	VC [$\cdot 10^6$]	0	40,15	48,43	24,97	38,61	34,78	19,59	7,12
	RT [s]	2,63	17,9	22,73	37,56	53,42	77,32	100,84	112,3
NO	F#	1	1178	1502	3100	5379	7387	9950	11977
YES	FT [ms]	0	20,3	13,1	7,5	15,7	16,6	6,8	5,6
	FPS		49	76	133	64	60	147	179
	VC [$\cdot 10^6$]	0	51,36	40,78	21,74	51,90	34,91	18,57	7,08
	RT [s]	2,54	17,82	22,69	37,52	53,37	77,29	100,81	112,27
YES	F#	1	1435	1908	3668	6018	8781	11542	13362
YES	FT [ms]	0	9,9	10,5	7,6	9,4	8,1	8,4	5,9
	FPS		101	95	132	106	123	119	169
	VC [$\cdot 10^6$]	0	27,8	33,43	20,69	26,29	26,79	18,57	7,08

LOD = Using Levels of detail, OC = Using Occlusion Culling
RT = Realtime, F# = Frame Number, FT = Frametime
FPS = Frame per seconds, VC = Vertex count

Table 3.12: Demo 4, table of comparison between optimization techniques

3.4.5 Test 3 - Occlusion Culling

We baked the occlusion culling data for this test using occlusion areas and the default parameters (smallest occluder - 5, smallest hole - 0.25, backface threshold - 100). This resulted in a data size of 251.8KB and enabled the force LOD script.

This time no change in the performance of the build on the Dell, resulting in the same perfect average of 60FPS, see Table 3.15 The performance on the Ideapad is a little bit better than in the previous test, with an average frame time of 39,4 ms. The fluctuation is similar when using levels of detail. On the Thinkpad, the performance dropped slightly but remained nearly identical to the second test.

3.4.6 Test 4 - Occlusion Culling and Levels of Detail

The occlusion data were identical, and the force LOD script was disabled.

Again no change in the performance of the Dell G5. In Table 3.16, we can see that the performances remain in the same order between them, where the Dell G5 is the best followed by the Ideapad, and last, the worst performance

HW	STATS	CHECKPOINTS							
		1	6	8	12	17	24	30	35
DG	RT [s]	5,99	21,82	26,67	41,52	57,42	81,31	104,82	116,3
	F#	1	557	747	1632	2546	3659	5070	5759
	FT [ms]	0	25,7	17,8	16,7	33,3	16,7	16,7	16,6
	FPS		39	56	60	30	60	60	60
	VC [$\cdot 10^6$]	0	95,88	66,83	26,56	128,13	48,86	33,2	7,12
IP	RT [s]	2,61	17,78	22,65	37,42	53,35	77,23	100,75	112,2
	F#	1	183	240	591	1039	1393	1966	2381
	FT [ms]	0	87,8	66,7	38,9	86,6	52,2	41,1	22,2
	FPS		11	15	26	12	19	24	45
	VC [$\cdot 10^6$]	0	96,1	66,41	26,57	128,5	46,92	33,2	6,56
TP	RT [s]	4,92	20,93	26,29	40,95	61,81	85,88	109,30	120,69
	F#	1	77	100	245	476	607	853	1049
	FT [ms]	0	255,3	205,3	91,9	319,7	167,6	103,2	43,3
	FPS		4	5	11	3	6	10	23
	VC [$\cdot 10^6$]	0	96,63	67,38	26,56	133,43	47,24	33,2	7,12

DG = Dell G5, IP = Ideapad, TP = Thinkpad
LOD = Using Levels of detail, OC = Using Occlusion Culling
RT = Realtime, F# = Frame Number, FT = Frametime
FPS = Frame per seconds, VC = Vertex count

Table 3.13: Demo 4, Table comparing the not optimised builds on the different hardware.

is again recorded on the Thinkpad. Despite this, the frame times lowered for both of the Lenovo notebooks resulting in the best results of an average of 37.7 ms and 74.7 ms, respectively.

3.4.7 Results

Figure 3.15 shows the use of occlusion areas and the generated view volume. As we already know, here, too, the view volume is smaller, resulting in more precise calculations inside the occlusion areas. The occlusion areas are placed around the camera path, which is represented by the red line. We can also see how certain facades close to the camera have more triangles than those further. This is because of the use of LODs.

The first run of tests is visualised in Graphs 3.16. We can see the performance of the demo in the play mode of the editor. For the second test, we can see the massive drop in vertices rendered when using the Levels of detail optimization techniques. This is expected when the difference in vertex count of the first and second levels of detail is significant. The frame time has also dropped accordingly. Results in the third were as expected, from the knowledge we gathered during the previous demos, as the occlusion culling optimised the scene's performance. Interestingly it was less effective than LODs. Combining both techniques yielded the best results and had a more

HW	STATS	CHECKPOINTS							
		1	6	8	12	17	24	30	35
DG	RT [s]	4,64	20,13	25,01	39,86	55,7	79,63	103,15	114,61
	F#	1	873	1166	2057	3008	4444	5856	6544
	FT [ms]	0	16,7	16,7	16,7	16,7	16,7	16,7	16,7
	FPS		60	60	60	60	60	60	60
	VC [$\cdot 10^6$]	0	40,14	48,43	24,97	38,66	34,77	19,59	7,12
IP	RT [s]	2,6	18,16	22,6	37,98	54,2	78,25	102,01	113,52
	F#	1	139	199	366	607	875	1176	1388
	FT [ms]	0	110,5	105,5	76,2	93,2	91,9	69,3	45,4
	FPS		9	9	13	11	11	14	22
	VC [$\cdot 10^6$]	0	40,54	47,67	25,15	38,45	34,75	19,78	6,34
TP	RT [s]	2,65	17,9	22,87	37,49	53,44	77,36	100,91	112,36
	F#	1	126	165	333	578	813	1091	1319
	FT [ms]	0	118,7	128,7	86,6	89,9	112,1	69,9	42,2
	FPS		8	8	12	11	9	14	24
	VC [$\cdot 10^6$]	0	40,34	48	24,98	36,88	34,75	19,77	6,56

DG = Dell G5, IP = Ideapad, TP = Thinkpad
LOD = Using Levels of detail, OC = Using Occlusion Culling
RT = Realtime, F# = Frame Number, FT = Frametime
FPS = Frame per seconds, VC = Vertex count

Table 3.14: Demo 4, Table comparing the builds optimised using LODs only on the different hardware.

stable frame rate.

Again we can see how many vertices are saved when using the optimization techniques, using both for the best results and Levels of Detail for better performance than Occlusion culling. This can be produced by the LOD group component that culls the object when it takes less than 1% of the screen. In other words, on top of rendering fewer vertices by using a less detailed level of detail, the LOD technique also culls those not visible on the screen, the occlusion culling doing only the latter.

HW	STATS	CHECKPOINTS							
		1	6	8	12	17	24	30	35
DG	RT [s]	4,08	19,52	24,4	39,24	55,09	79,01	102,54	114
	F#	1	873	1166	2057	3008	4444	5856	6544
	FT [ms]	0	16,7	16,7	16,7	16,7	16,7	16,7	16,7
	FPS		60	60	60	60	60	60	60
	VC [$\cdot 10^6$]	0	51,79	41,11	21,76	51,89	34,91	18,57	7,08
IP	RT [s]	2,6	17,62	22,54	37,32	53,21	77,14	100,65	112,12
	F#	1	334	432	856	1392	1970	2630	3125
	FT [ms]	0	53,4	45,5	32,2	53,3	40	30,3	21,1
	FPS		19	22	31	19	25	33	47
	VC [$\cdot 10^6$]	0	50,16	40,75	21,75	49,82	33,92	18,57	6,52
TP	RT [s]	2,67	17,83	22,69404	37,45	53,42	77,32	100,8	112,24
	F#	1	145	184	362	617	851	1140	1387
	FT [ms]	0	142,1	111	77,7	122,1	102,1	66,6	37,7
	FPS		7	9	13	8	10	15	27
	VC [$\cdot 10^6$]	0	53,08	41,4	21,92	50,22	33,92	18,97	6,52

DG = Dell G5, IP = Ideapad, TP = Thinkpad
LOD = Using Levels of detail, OC = Using Occlusion Culling
RT = Realtime, F# = Frame Number, FT = Frametime
FPS = Frame per seconds, VC = Vertex count

Table 3.15: Demo 4, Table comparing the optimised builds using Occlusion Culling only on the different hardware.

HW	STATS	CHECKPOINTS							
		1	6	8	12	17	24	30	35
DG	RT [s]	4,13	19,67	24,55	39,39	55,24	79,16	102,69	114,15
	F#	1	877	1170	2061	3012	4448	5849	6537
	FT [ms]	0	16,7	16,7	16,7	16,6	16,7	16,7	16,7
	FPS		60	60	60	60	60	60	60
	VC [$\cdot 10^6$]	0	27,79	33,42	20,7	26,31	26,8	18,57	7,08
LN	RT [s]	2,61	17,62	22,48	37,32	53,15	77,12	100,63	112,1
	F#	1	383	507	960	1517	2253	2946	3442
	FT [ms]	0	37,7	40	31,3	36,7	33,3	28,9	21,1
	FPS		27	25	32	27	30	35	47
	VC [$\cdot 10^6$]	0	27,79	33,6	20,7	25,9	26,80	18,58	7,08
TP	RT [s]	2,7	17,78	22,66	37,51	53,36	77,34	100,85	112,32
	F#	1	172	227	422	688	1009	1319	1565
	FT [ms]	0	91	92,1	73,3	76,6	85,5	66,6	37,7
	FPS		11	11	14	13	12	15	27
	VC [$\cdot 10^6$]	0	29,07	33,03	20,88	26,28	26,79	18,58	6,52

Table 3.16: Demo 4, Table comparing the builds optimised by both optimization techniques on the different hardware.

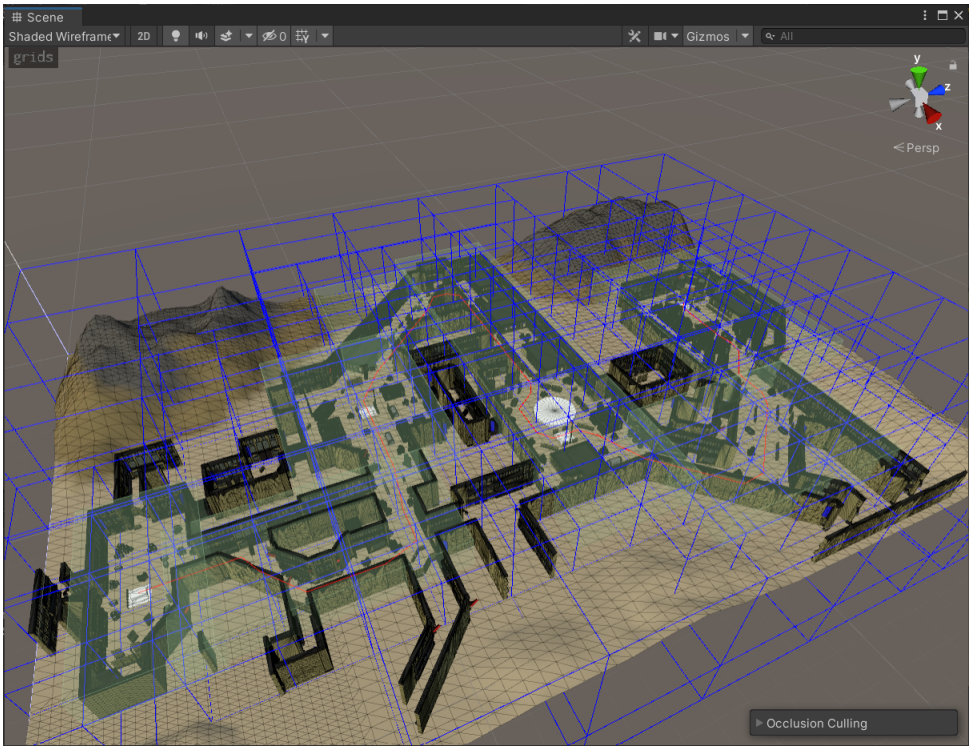
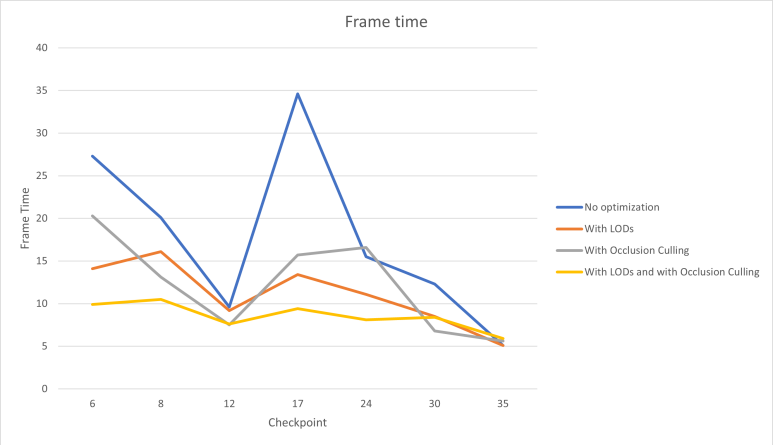
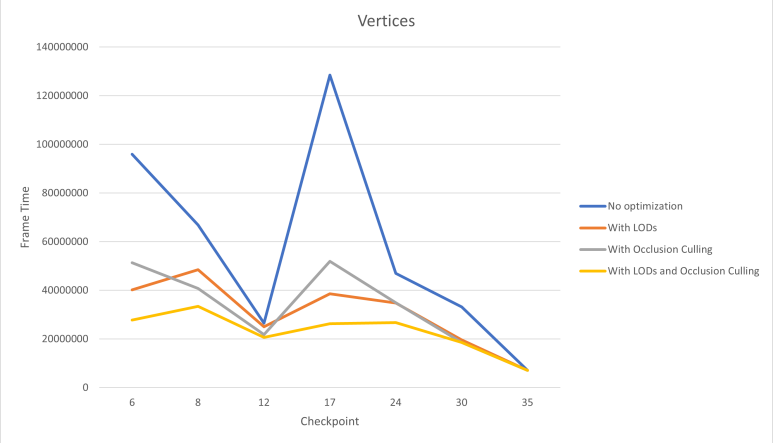


Figure 3.15: Demo 4, Shaded wire-frame view of the scene with the path of the camera in red, occlusion area as green boxes and baked view volumes as blue wired boxes.

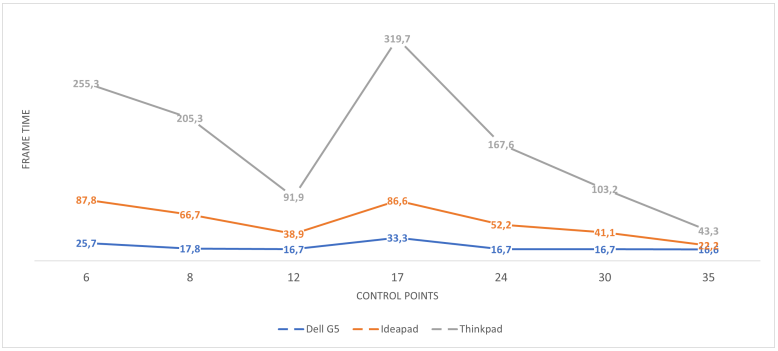


(a) : Graph comparing frame times in selected cooptimisednts

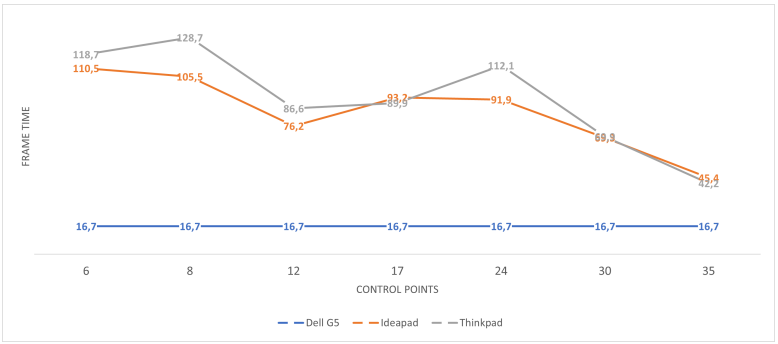


(b) : Graph comparing vertex count in selected control points

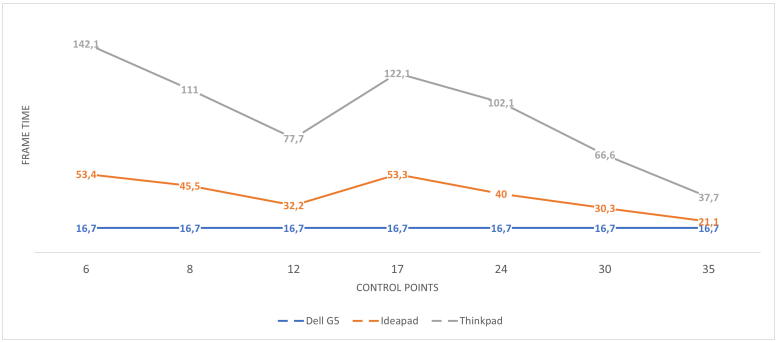
Figure 3.16: Demo 4, Graphs comparing different optimization techniques.



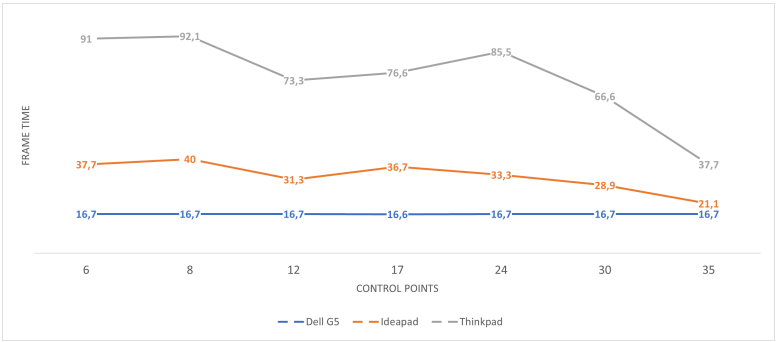
(a) : Test 1 - Not optimised



(b) : Test 2 - Level of detail



(c) : Test 3 - Occlusion culling



(d) : Test 4 - Both techniques

Figure 3.17: Demo 4, Graphs of the frame times recorded on the different hardware.

Chapter 4

Conclusion

Performance is a crucial part of a game. We must ensure a smooth frame time to deliver an excellent player experience. Fortunately, we have at our disposal many tools to succeed in that. The most common ones have been discussed in this thesis. We now know how there are used in Unity and how simple it is to use them.

> "Visibility culling is the other essential ingredient, in addition to LOD techniques, to make applications output sensitive." [2]

This quote sums up why we looked at Visibility culling and LOD in detail and tested these methods.

Occlusion culling is an effective rendering optimising technique by limiting the number of objects that need to be rendered, optimising many resources that the GPU can use more efficiently. The only down side it needs to have baked data to calculate the culling, taking up space on the disk. By wisely choosing its parameters, we can achieve great results. And not always smaller numbers have better results, as changes frequently the GameObjects that need to be rendered. An excellent way to find the ideal parameters is to know how big the smallest GameObject is that we want to use as an occluder and choose the smallest occluder parameter value. Although, even like this, finding the best parameters requires a lot of tuning. The best use of these techniques is in environments with a more dense number of Game Objects occluding one another. For example, rooms that are connected by corridors, a dense cityscape with high buildings and streets.

Levels of Detail need a little more work outside the engine by modelling different levels for a model. But in the engine, it is straightforward, as Unity can automatically assign the LOD group component to the models. But once set up, it offers a significant difference in performance as it effectively reduces the vertex count for details that are not visible at longer distances. The tunning of the parameters is also simpler and is better visualised than for the occlusion. We need to find the sweet spot where we cannot see a visible change between the two Levels of Detail. This technique mainly benefits more open environments with long view distances and detailed models scattered around the scene, where the player can move freely.

By combining both techniques, we achieved better and more stable performance. But it doesn't end here, as Unity disposes of a great number of other methods that helps polish the run of the game builds. Combining different optimization techniques, which are well documented, is recommended to ensure smooth performance and, thus, player experience.

Bibliography

[1] Lindstrom, Peter, et al. *Real-time, continuous level of detail rendering of height fields. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996.*

[2] Dietrich, Andreas, Enrico Gobbetti, and Sung-Eui Yoon. *Massive-model rendering techniques: a tutorial. IEEE Computer Graphics and Applications 27.6, 20-34, 2007.*

[3] Unity Technologies *Optimise your game performance for consoles and PC ebook, 2020 LTS Edition*, 2021

[4] Unity Technologies *Unity Documentation,* `https://docs.unity3d.com/Manual/index.html`

[5] Unity Technologies *Unity Documentation, Profiler,* `https://docs.unity3d.com/Manual/Profiler.html`

[6] Unity Technologies *Unity Documentation, Normal Maps,* `https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html`

[7] Unity Technologies *Unity Documentation, Parallax Maps,* `https://docs.unity3d.com/Manual/StandardShaderMaterialParameterHeightMap.html`

[8] Unity Technologies *Unity Documentation, Occlusion Maps,* `https://docs.unity3d.com/Manual/StandardShaderMaterialParameterOcclusionMap.html`

[9] Unity Technologies *Unity Documentation, Draw Call Batching,* `https://docs.unity3d.com/Manual/DrawCallBatching.html`

[10] Unity Technologies *Unity Documentation, Static Batching,* `https://docs.unity3d.com/Manual/static-batching.html`

[11] Unity Technologies *Unity Documentation, Dynamic Batching,* `https://docs.unity3d.com/Manual/dynamic-batching.html`

[12] Unity Technologies *Unity Documentation, Occlusion Culling,* `https://docs.unity3d.com/Manual/OcclusionCulling.html`

[13] Unity Technologies *Unity Documentation, Occlusion Areas,* `https://docs.unity3d.com/Manual/class-OcclusionArea.html`

[14] Unity Technologies *Unity Documentation, Level of Detail,* `https://docs.unity3d.com/Manual/LevelOfDetail.html`

[15] Unity Technologies *Unity Documentation, LOD Group,* `https://docs.unity3d.com/Manual/class-LODGroup.html`

[16] doc. Ing. Jiří Bittner, Ph.D. *Optimalizační metody pro hry, lecture from course B4B39HRY - Počítačové hry, DCGI FEL ČVUT*

[17] Ing. Ladislav Čmolík, Ph.D. *Levels of detail, lecture from course BE4B39VGO - Vytváření grafického obsahu, DCGI FEL ČVUT*

[18] Unity, *How to profile and optimize a game | Unite Now 2020, video,* `https://youtu.be/epTPFamqkZo`

[19] Unity, *Capturing profiler stats at runtime | Unite Now 2020,* `https://youtu.be/i2IpJHUyZLM`

Appendix A

Used assets

Here is the list of assets and tools that I used from the Unity Asset Store or Package Manager in alphabetical order:

Conifers [BOTD] - `https://assetstore.unity.com/packages/3d/vegetation/trees/conifers-botd-142076`;

Destroyed City FREE - `https://assetstore.unity.com/packages/3d/environments/sci-fi/destroyed-city-free-6459`;

House pack - `https://assetstore.unity.com/packages/3d/environments/house-pack-35346`;

ProBuilder - available in the Package Manager;

ProGrids - available in the Package Manager;

Snaps Prototype | Sci-Fi / Industrial - available in the Package Manager;

Terrain Sample Asset Pack - `https://assetstore.unity.com/packages/3d/environments/landscapes/terrain-sample-asset-pack-145808`;

Appendix B

Attachments

As an attachment of this thesis are the Builds for Demo 4 located in the **bin/** folder. All the files for both Unity projects are in the folder **src/**. The latex file is in the folder **latex/** folder. Every image this thesis uses is saved in **images/**. And a **README** file is added with a detailed hierarchy, locations of the files and the link to the GitLab Repository.