Improving the game development process on Unreal Engine 5

Bachelor's thesis

Degree Programme in Computer Applications

Spring 2024

Anton Satalin

HAMK
Häme University
of Applied Sciences

Degree Programme in Computer Applications Abstract
Author Anton Satalin Year 2024
Subject Improving the game development process on Unreal Engine 5
Supervisors Mazhar Mohsin

The objective of this thesis is to reach a conclusion regarding whether game development needs optimisations or not. Reasoning behind is the author's personal journey of figuring out how to optimise games himself and trying to put it into something practical. Author has attempted to use his prior knowledge in game playtesting to give reasons to the common issues occurring in games. The questions set in the thesis were asking about the importance of said improvements, their necessity, and the ways that they could be implemented.

The thesis sought to analyse and test the possible ways to improve performance and analyse who would be potential beneficiaries from optimisation. The thesis was done only on the author's behalf.

The outcomes of thesis were performance tests done on Unreal engine, which sought to research the influence on the performance graphical presets provided with the engine as a support for the majorly theoretical subject that was researched using predominantly literature review.

The thesis concludes with the estimation that even with the difficulties in setting up game developments improvements, their long-term benefits will be able to save much more time and resources than having to fix the issues that would arise without them.

Keywords Framerate, Performance, Unreal Engine, Blender
Pages 31 pages and appendices 20 pages

Glossary

FPS	Frames per second
TPS	Ticks per second
CPU	Central processing unit
GPU	Graphics processing unit
RAM	Random-access memory
FBX	***F****ilm****b****o****x*** file extension

Content

Figures

Tables

Appendices

1 Introduction

The topic of game optimisation has always been of great significance to the players, as with the release of any new game, the first thing that it will be judged by is not story, gameplay or graphics but rather how well it runs in the first seconds, as everybody wants to enjoy the game without having to wait for simple actions to take multiple seconds to be processed. On the author's side is the legitimate interest in figuring out different ways to improve performance without sacrificing too much graphical fidelity, which with some games could get so problematic, that in order to achieve proper, playable framerate some parts of a game would have to be sacrificed, with it usually being objects' shadows. Nowadays, taking into account the past experiences with game optimisation, playtesting, bugfixing and participation in an actual game development project, the author is committed to turn that experience into something presentable, so that this thesis could be used to support the author and his future carrier as a game developer.

But game optimisation has practical benefits: for example, well optimised games could be accessed by a much broader audience due to lowering the entry barrier that is attributed to hardware requirements. The newer market requirements such as "backward compatibility" and "cross-generation optimization" require the developers to be able to port their games to multiple platforms. (Polydin Game Studio, 2023)

But first, it is important to note that the topic of hardware and how much do different models of it affect the performance will be left out, as it is a topic for somebody who specialises in hardware much more than the author. The focus will be more on comparing multiple different game optimisation solutions to each other on a same laptop to reduce the number of variables that could tamper with the results.

For the practical part, the plan is to run optimisation solutions and gather the performance results. Specifically, it will include downloading a performance-intensive object from a public repository and then use those game optimisation solutions in order to test out by how much can the FPS be improved by using them. There will also be a hypothetical code optimisation, because in order to see serious changes to performance due to code, a lot must of code must be written either to make game performance-heavy or make serious optimisations to it. The base metric will be FPS (Frames per second). And for the questions that are planned to be answered during this thesis, they will be all related to game optimisation and are :

- Why is optimisation in games such a popular topic in the discussions?

- How would it be possible to improve optimisation in games and by how much?

- Is it worth it to make your game adaptable to changes in development?

2 Performance optimisation

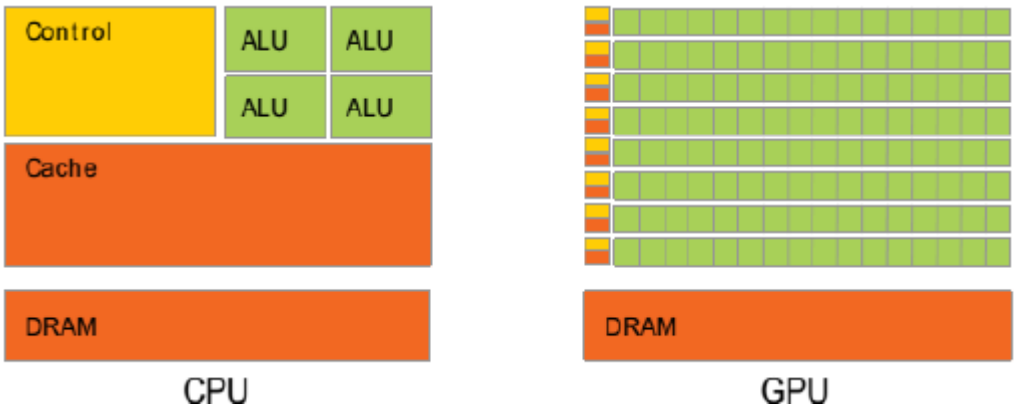
The thesis is going to attempt to gather many different solutions to the problem of bad game optimisation. Due to this kind of topic requiring already created project and the effectiveness of such solutions scaling directly to the amount of content already implemented. As an example, games that primarily convey a story first, do not have much content to optimise and polish. On the other end of spectrum are “competitive first-person shooter” type of games, for which usual optimisation will not be sufficient, it must be responsive enough for everybody – players input, server recognising that input and updating player actions in just a few milliseconds. (Intel Corporation, n.d. -a)

While understanding the end-product requirements for the project is important, it is essential to also understand the details from which the game worlds and objects are created – The objects that could be seen in games, are constructed from simpler figures, with preference being given out to squares and rectangles, due to them being easy to modify and to use in advanced modelling. (Oravakangas, 2015, pp. 25-31)

2.1 Rendering in computers

Most of the computers nowadays do have enough capabilities to run rendering engines, but all models of computers do have restrictions on how much can they handle before FPS starts to plummet – the older the model the less it can handle. The 4 most important parts of the computer that dictate how much can it process before slowing down are : CPU (Central Processing Unit), GPU (Graphics Processing Unit), RAM (Random Access Memory) and GPU Memory (Sometimes also referred as VRAM – Video Random Access Memory). CPU is responsible for processing everything from simple calculations to rendering, but the disadvantage is that it does not specialise in one specific thing but rather doing them all at ones, so it has less resources it could allocate to rendering than for example GPU who only does graphical processing, so its resources will not be used by anything not related to graphics. Figure 1 shows the difference between them, with CPU having less but more powerful cores and GPU having a lot smaller core, that could do multiple calculations at the same time.

Figure 1. CPU vs GPU Architecture (Thambawita, Ragel, & Elkaduwe, 2014)



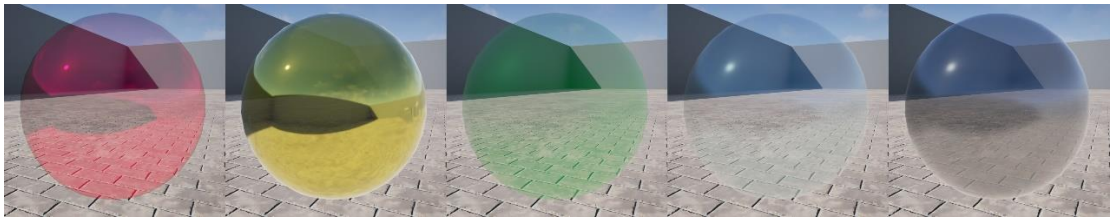
While CPU and GPU process graphics, RAM, and GPU Memory store it for usage. RAM is used by CPU to save information and results of processing logic such as location of objects, while GPU Memory store information about how to render them for example shadows, post-processing and etc. (Alton, 2020)

2.1.1 Unreal Engine

Before the discussion about the rendering optimisations in Unreal Engine can be done, the first moment that should be noted is creating an environment to render. The time, resources and performance requirements can change drastically depending on the requirements that the environment should fill. The environment could be something simple like a demonstration of an idea for the new project or maybe even a frame for a high-budget film.

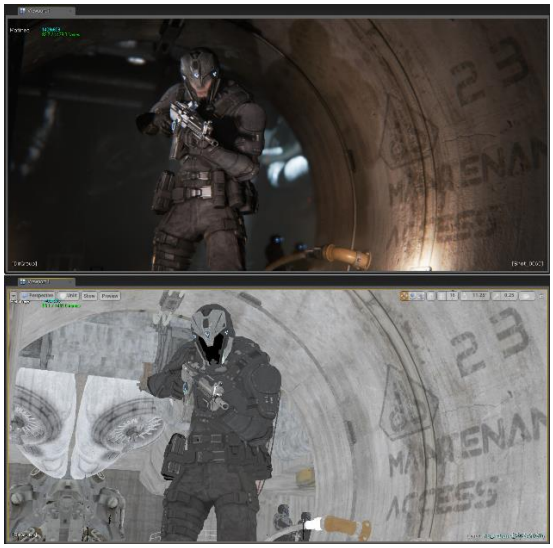
The environment consists mainly of textures which are like a wallpaper that could be applied on the objects. As it is in real world, it is impossible to make a concrete wall suddenly transparent, it requires special **materials** that are different from textures, as they dictate how will the object interact with lightning and whether, in this case, it will let light pass through it or not. Textures and materials are completely interchangeable: both transparent and opaque object can be painted, as it is seen with both transparent and coloured spheres on Figure 2, and covered in textures and the only difference that will be how will those changes interact with the surrounding world.

Figure 2 Multiple transparent spheres, tinted with different colours. (Epic Games, Using Transparency in Materials, 2024 -l)



While object and their parameters create the base of the environment, they still lack the important aspect of lighting. Without proper lighting there is either complete darkness or everything is too bright. While the problem of complete darkness is not that difficult to understand, the full brightness on every object may not be as problematic as complete darkness, it still recommended to not leave it as is. The reason for that is that fully lit objects, especially when they have the same colour could mess with perception of surroundings and make navigating such environments difficult. Figure 3 showcases just that with the second image being completely different from the first image, where the environment feels completely different and has problems with overwhelmingly confusing objects.

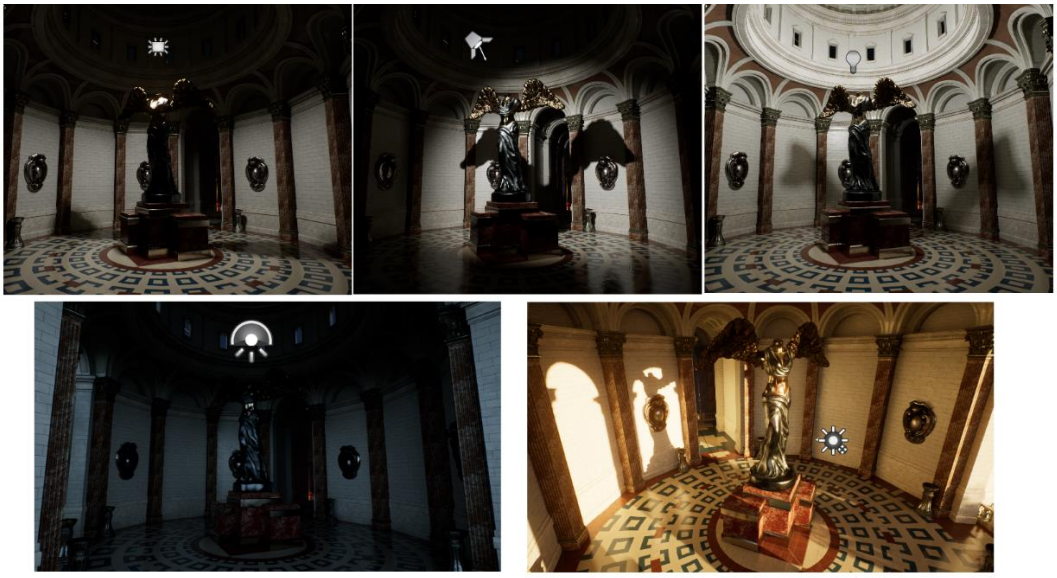
Figure 3. Difference between normal lighting (top) and full brightness (bottom). (Epic Games, View Modes, n.d. -m)



There are multiple ways how to set up lighting for the scene starting from simple static point light to dynamic spotlight. The first one will essentially be a light bulb lighting up the room around without the capability of moving or changing its attributes like colour on command, while the latter one is a complex system of spotlights, that could change its colours and move around freely to light up the surrounding differently, depending on what is happening on the scene. Just like in real life those

complex lighting systems are expensive (in case of computers they are, more correctly speaking, taxing on the computer's resources), hard to set up, and require a knowledge base to use effectively. All of the light types available in Unreal engine are shown in Figure 4, from the simple indoors lighting to more complex sun light with its shade. (Epic Games, Inc., 2024 -a)

Figure 4. Collage of different lighting types : Rectangular Area Light (top left), Spot Light (top centre), Point Light (top right), Sky Light (bottom left) and Directional Light (bottom right) (Epic Games, Inc., 2024 -a)



After work with lighting is complete, additional effects could be applied to further enhance the picture. There are a lot of different effects, but most of them specialise in either adding further realism or giving an otherwise generic image its own unique style. Category-wise the effects that could be split into one of the following : "Image effects", "Global Illumination" and "Reflections".

Image effects generally don't affect the performance, as they are only changing visual representation of an image and do not require heavy calculations that would use a lot of resources. (Epic Games, Inc., 2024 -b) Global illumination can be either be pre-rendered, which is better for simple, static environments, while dynamic illumination is a lot better for large, complex environments with day and night cycles, as everything is done on the go. The point of having global illumination is make the lightning more accurate to the environment. (Epic Games, Inc., 2024 -c)

Reflections, as it says, deal with reflective materials such as polished metals, water, glass and etc. (Epic Games, Inc., 2024 -d).

2.1.2 Blender

While Blender is capable of rendering scenes and whole videos, it is not an interactable like a game engine. Which does not make it useless in any way, as it is great for creating models and textures for the game engine. In its sense both Unreal and Blender can be completely independent but provide better customisability to the project if used together.

Due to blender being used in these types of projects as just a texturing tool and not many other features such as animation, physics and etc. . , it makes sense to talk only about the features that are being used instead of the massive library of tools that the Blender otherwise provides.

While Blender is used in this case to only create, edit, and port models and their textures, seeing how the rendered texture will interact with lights and different rendering engines will be useful to predict future issues with it, for example lighting artifacts, finding which as early as possible will save a lot of time trying to find a workaround in the final project.

While in the Unreal the parts of the environments were used, in the Blender part only different editors will be mentioned and render engines, as here, in Blender most of the interactions are done with objects and lighting is mostly used in testing to prevent visual and other bugs from being transferred into Unreal Engine.

The biggest part in creating objects in Blender is the “3D Viewport editor“ where most of the interactions with the items happen from adding a new item to editing shapes and using tools that could for example smooth out the edges. (Blender, 2024 -a)

While the content made in Blender could utilise additional elements, for example sculpting, which is great for creating life-like models, the necessity of using them depends on the model requirements for the project. And because again, every project is unique, talking about complex functions and reciting tutorial materials is always a gamble, as it is not always apparent, how knowledgeable is the person, what tools does the person prefer using, and how much of different tools does the person need to fulfil the project’s quota.

After the base model is created, it is time to apply textures to it, using the UV Editor. The UV Editor essentially tells the model which parts of an image it should use on the model and where to place them on the model. For the more complex texture and/or models, it will be necessary to edit the UV. (Blender, 2024 -b)

After the textures are applied, it is time to test it out with different render engines. By default, it should be Workbench, as it processes the models the quickest, which makes it comfortable to use while editing, without having to wait for new changes to be rendered.

Other render engines such as EEVEE and Cycles are oriented towards higher quality images and achieve it using different methods.

EEVEE is the more balanced in the performance department due to it just estimating how the light is going to fall, instead of accurately calculating it. This does create a lot of limitations to it due to it just predicting the lighting and staying light on the performance side. (Blender, 2024 -c)

Cycles on the other side, is the most accurate and adaptable rendering engine while still staying reasonable and friendly to users. The reason for that as they say is :

> We do think handling large amounts of geometry for smooth surfaces, displacement and hair is important. However, we also think that rendering algorithms based on long preprocessing and baking, which are often used in such Renderman engines, are not sufficiently interactive and user friendly for most users. (Blender, n.d.)

So that means that they would like to reduce the barrier to entry to new users without removing any features. The drawback is that it is the most performance intensive due to it not making any compromises to stay as accessible as it currently is. (Blender, n.d.)

Once the process of rendering is complete and models are tested for any visual bugs under different lighting arrangements, any visual bugs fixed, the models can be ported into Unreal Engine as a file or files in. fbx extension.

2.2 Optimising performance in games

There are many ways to optimise a game, from using visual tricks to make low quality textures look good, with special effects, to optimising code to run the processes faster with less mistakes. For this part, the focus will be on optimising the graphics and using certain tricks to it. Oksanen (2022, pp. 12-21), who has done the thesis on the similar topic, stated that optimisation keeps the flow of the game consistent at all the graphical level, usually to ensure the game has good qualities for it to be easily adapted for different devices with different processing capabilities. Mika also pointed out that the most important metric in modelling is the polygon count, which means that the models should have as little faces as possible, when it possible to do so, without changing the object’s appearance.

But in order to even start optimising, it is important to set up the environment for testing, as many issues may not be apparent in the beginning of the project. Finding one specific object that consumes all the resources with thousands of objects will be difficult. Separating the environment into smaller pieces and then finding the culprit is much easier.

In addition of setting the environment for testing, it is also important to consider testing the performance on different devices, that the game would be available on, starting from testing different computers in different price ranges and producers, if the game will only be available on computers. If there are plans for porting, then the game should be able to perform well on those platforms too, which means that additional optimisation may be needed for the weaker platforms such as mobile and less powerful consoles.

The Unreal documentation describes a decent amount of ways to optimise the application's performance, in this case the user interface with specific modules, still could be used in general optimisation, like reducing the amount of needed updates, for example instead of checking the player's health every game tick, it can be checked only if the player takes damage or is healed. While this is related more to fixing the code, and the performance improvement is just a side effect, the connection between good code and good performance is still essential in improving the project's optimisation. (Epic Games, Inc., 2024 -e)

In addition to the user interface, the Unreal Engine itself is able to provide additional optimisations such as rendering the perceived image in lower quality than usual, stretching the image to fit the screen, filing the gaps and rendering the updated image instead of the low quality one. The result most of the times is an image that takes less resources to render but still tries to keep the original quality. Sometimes during the process of filling the gaps some visual artifacts that may appear, reducing the quality of the image. (Epic Games, Inc, 2024 -f)

The other type of optimisation is related to the objects themselves and adding additional interactions to them that could improve performance. One of those examples is LOD (level of detail) technique and it is usually used to reduce the complexity of an object the further it is from the camera. The LODs can be set up either manually by creating a similar texture but with reduced complexity and detail, adding a condition when should it change to a less detailed one and the other way around, which is usually either distance or some sort of an action, for example passing through the room and reducing quality of the table in the room before, assuming of course those rooms are long enough for the quality reduction to not be noticeable, as it is the main concern of good LOD creation process – making sure that the process of the objects changing their LOD is not noticeable for the camera, as the suddenly changing textures could distract the player from achieving the goal of the game. (Epic Games, Inc, 2024 -g)

The more extreme technique that is in the same vein as LOD is Occlusion Culling, which instead of reducing the quality of distant objects, just refuses to render them at all, which could potentially save a lot of computer resources, if done correctly or consume even more computer resources, due to it having to render new things every time the camera moves. The other problem that it introduces is that every object should have appropriate bounding boxes, which themselves tell the information about the object's location, rotation and scale. The bounding boxes are also used to simulate intractability like walking into the door's bounding box can be used to simulate pushing it, if programmed properly. (Epic Games, Inc, 2024 -h)

There are different methods to apply Occlusion : the simplest one is to set two dots which will define how far from the camera will the object be visible, but in this case it is more accurate to say those dots are : minimum distance and maximum distance. The maximum distance to render can also be set in two distinctive ways – either as a simple value or base it around the size of the object. The latter could be used to stop rendering smaller objects like vases in a house on much shorter distances than mountains, monuments and other landmarks. This method of culling is very predictable, making it a good option either as a backup option for other types of culling or for more linear worlds where distances and object locations stay the same. But in the cases where the simple distance-based occlusion is not enough, other, more advanced methods of occlusion could be used, that check for whether the object itself is visible from the camera or not. These types of checks can occur either before the level is loaded or real-time. The benefit of occlusion done beforehand is, of course, significant improvement in performance, while staying subtle, which distance-based occlusion method cannot be, specifically in more detailed environments. The disadvantage of the pre-rendered occlusion is that of course, it is done before, so any changes in the lighting, object position during the gameplay will not be updated. For the occlusion to be able to update objects occlusion, it needs to be dynamic, hence the dynamic occlusion is needed for that. But again, due to the occlusion occurring every frame, a lot of requests asking CPU or GPU to render or not to render can cause massive strain on the computer, using only dynamic occlusion just by itself is not recommended for the bigger projects, it should be paired with other types of occlusion in order for the dynamic occlusion to work only on dynamic objects or to fill other occlusion methods' shortcomings. (Epic Games, Inc, 2024 -h)

2.3 Common performance issues and solutions to them

In this chapter, the focus will be more on the practical application of performance optimisation by utilising the more common issues that are encountered during testing. This sort of issues can be anything – low or unstable framerate, slow updates of the world, visual artifacts and etc.

Probably the most discussed performance issue is the framerate. Does it come due to game development advancing its graphical fidelity in order to look even more realistic, while it is still possible to find games that still take a lot of resources without offering much, or even games that look great and run well without much issue ? Why do certain games struggle on even the most modern devices and other run well even on older machines?

There could be a lot of reasons for that, but it would be unfair to compare two completely different games from different time periods, genres, perspectives and etc. , just due to the requirements for what is considered as a well optimised game are significantly different, for example if the comparison is drawn between a basic turn-based RPG and a competitive First-Person Shooter then on its own very similar to comparing a game of chess to a game of paintball in a sense that chess or in this case the RPG game only makes any updates to the game state when a move is done and otherwise keeps relatively still apart from few animated objects such as timer or in the game the characters idle movements, while in the case of paintball or its depiction in game, all the player positions, where the paintball gun is pointed, where will the paintball hit, will it hit another player, is player already hit and many other checks that are related to game's rules are constantly being updated in real-time in order to prevent many rule enforcement problems, such as players who should be out of the game, hitting another player and a hit being registered instead of ignored.

In addition to that a study with cooperation from Intel Corporation focused on framerate effects on player satisfaction concluded that correlation between game satisfaction and average framerate is heavily dependent on the game itself, as different genres and player's playtime in the game heavily influence the results. (Liu, Kuwahara, Scovell, & Claypool, 2023)

Again, this calls for carefully constricting the type of game that will be tested, which in this case will be photorealistic 3D game and the testing will probably consist of turning on and off certain features, such as shadows, real time rendering in Unreal Engine 5 and, on the Blender, side trying to make objects in different ways.

Now with the reasoning why is good framerate important, it is time to list the most common issues causing low or unstable framerates.

Starting from having constantly low framerate without any input or any updates, which is likely caused by the device not having enough processing power, which could be either CPU or GPU, to update the image at the consistent rate. The source tells that "There are no benefits of having higher FPS than your monitor can display" which is only partially true, because again having extra resources to spare could help a lot when a lot of new objects or interacting with the server, where updating and displaying changes takes additional time. Saving resources is not insignificant as it

could help a lot with bottlenecks which will be discussed in the unstable framerate part. Possible solutions may include reducing the quality of the image by simplifying or removing a visual part of the game such as shadows, reflections, and lighting. Additionally, it is possible to manipulate the final image rendered in order to improve performance. This could be done by reducing the number of pixels rendered on screen which could give a significant boost to FPS. (Intel Corporation, n.d. - b)

Changing visual quality does not come without any potential problems. One of those problems could be that changing visual quality of the game does not provide sufficient improvement in framerate, due to it not being properly implemented or being way too extreme, actively crippling the gameplay by rendering objects only a few meters away from camera. Most of the time this happens due to the player not being allowed to modify the visual quality manually or is restricted by only using presets. The usual solution is usually on the developer side is to either allow the modification of graphics in every aspect, or being able to see how the graphics will change with every setting, either as a comparison or immediately updating the graphics. This is of course not a solution to the underlying optimisation issues with the game itself, which should be handled as soon as possible to keep the customer base as wide as possible regarding to their device's capabilities.

For the next common issues, the problems that were already mentioned will be omitted, unless they have other solutions.

The next issue is having a low framerate when the camera is moved, which could signify that optimisation methods such as Occlusion Culling are not working properly or used in the places where they should not be even used, such as completely removing all objects that are not in a view causes massive queries sent to GPU and CPU, which massively slow down the framerate in order to render the objects.

Solution for those sort of rendering issues could, of course be minimising the amount of updating done on turning the camera and, instead, do rendering updates based on distance, which occlusion has an option for. The other important aspect in fixing occlusion is to strike a balance between how many often do those kind of updates occur and how many much of the environment is updated at once – is the developer striking for seamless updates, which start rendering every single object at different distances in order for them to look like they naturally entered the camera's view without just popping into existence, or if the developer is creating a linear games where walls or other types of obstructions will be able to hide unrendered objects behind themselves. In the linear game in that case, it is better to only render those objects, that the player can see, and update only when its needed, all objects at once.

The reason why the balance must be struck is because single object updates tend to get worse the faster the camera moves, as speed of the camera is going to amplify the amount of updates needed per second of movement, which of course will increase the strain on the CPU and GPU. While for per-object rendering the speed of the camera is the problem, for the bulk updating method, the important part is to ensure that while the objects are invisible they should be blocked by proper, non-transparent walls and when they are required to be visible, they could be seen from all the angles, so that if the camera of the player in all possible conditions, will not see empty parts of the map, if they choose an odd angle to look from.

3 Code optimisation

While the active neglection of code optimisation may not noticeable right away, the problem with it is that the effects of good or bad code optimisation could not be seen without a lot of prior coding already done and by that point the effects of bad code optimisation are already slowing down the whole system and in order to fix all the issues of the game, code-wise at least, the developer could as well just make a whole new game, because rewriting an already established game would essentially require to not only require rewriting the parts that were critically broken but going into the root of the game and reconnecting all the other parts of the code, that was reliant on the unstable code and so on. It is simply easier to start anew, knowing the earlier mistakes, having new and updated requirements, newer tools and etc. instead of trying to forcefully push the new things into an outdated code.

Nevertheless, the code optimisation should not be disregarded as just a beneficial part of the development cycle, a well thought out code will become more valuable the more the content the game has.

3.1 Making the code easy to use

Code could be made easy to use by two usually common ways: segmenting the code into pieces of a puzzle, which will be talked about in its own chapter and making sure that each piece of puzzle is easy to understand and explain its role in the game. For that part of the code to be understandable, documentation could be written, where everything is explained in great detail and examples could be brought, but to be completely honest with the developer, the more complex is the code, the higher the chance that the documentation itself becomes too complex for anybody new to the development team and in addition to that, the documentation must be constantly updated and reevaluated to stay relevant.

This could become unbearable for the developers to maintain, so that they could, of course, hire a dedicated person, which will maintain the documentation, but this will be more of the same mistake, passing responsibilities to another person instead of addressing them, which regardless of approach, the usability of said documentation always suffers from difficult to understand code. If not addressed properly, difficult to understand code then becomes difficult to explain and finally, when the original person leaves the project, then the development team needs to essentially start all over again, due to them being unable to comprehend and use that person’s code.

So, making the code easy to understand includes giving the context appropriate names to the more complex variables, predicting what names will be taken for the new variables, which will assist in preventing any usage of confusing names before the coding phase is even begun.

Bringing a complex idea into a mechanic is a difficult process in game development, but it does not set a requirement to always make the code as complex as needs to be, as most of the times to turn that idea into something good, it just needs a few simple commands and functions to work well together and it will be the foundation for something greater that, for the average person, will look as something very complex and hard to achieve.

3.2 Getting rid of underused functionality

How would it be possible to isolate features that could be considered underused from the other ones? A good measurement of how much the feature is being used is testing out how does it interact with player – how often does the player utilise it, are the interaction positive for the player, how problematic is the feature for the developer in its implementation, as again there is no need to endure the feature and work with it, if it is constantly revolting against any modifications.

The question itself is a lot more complex than it may seem like, as it there are many possibilities what causes such issues and some of them may be invisible for one of developers and apparent for the other one. A well-balanced discussion with the goal of creating an overview on the functionality could help reach the final decision with minimal drawbacks. But it does not mean that every feature that was not liked should be removed completely, but rather just disabled, to ensure that if the feature is suddenly needed it could be easily restored, saving valuable time.

Once the underused functionality is found, many different options on dealing with it, could be used, from simply removing, which is usually reserved for game-breaking functionality or underutilised functionality in its early development phases where removing it will not harm the development process, to reworking or integrating it with other functionality. The integration can itself be done in many different ways : simply streamlining the behaviour of similar objects, like doors, which have opening and closing animations, may be unlocked or locked, requiring and checking for a key every time it is being interacted with; combining the new functionality with the already existing one.

3.3 Preparing for scaling and changing

As it was already stated in the previous chapters, separating the code into smaller pieces makes finding and squashing bugs in it much easier due to simply being able to answer which part of the code is causing problems, what is it responsible for and why does it not function or interact properly.

As an example of standardising of the objects was brought up in the earlier chapter, separation of the code comes very handy in being a baseline for the door example. In the cases, when a new door needs to be implemented by the developer, the part of the code that was already present in the code handling the different doors could help to reduce the amount of work needed to code a new door, so its implementation could be reduced to just setting a few variables and adding animations, if needed.

4 Evaluating the optimisation

Now that optimisation issues with their solution were collected, time to test their efficiency in improving FPS and see exactly how serious is the correlation between FPS and graphics and whether changes in them cause additional issues with the game.

4.1 Methods

For the testing part of the thesis Unreal Engine 5. 3. 2 and Blender 3. 6 will be used to test out the framerate changes with an application of different performance enhancement methods.

As for the device used, it will be Asus laptop Model M3502QA With AMD Ryzen 7 5800H with Radeon Graphics (integrated GPU) 3. 20 GHz Processor, RAM 16, 0 GB (15, 4 GB usable)

The graphics card is integrated, that means it is using CPU resources.

In the Unreal Engine Part, 3 different levels will be tested : a completely new map generated by Unreal Engine for new projects - FirstPersonMap, the map created during the Game development course and, if the results are inconclusive and there is time to spare, an open-source map created by Conrad Justin published on Sketchfab. Testing objectives will be to see how much does the average framerate changes with different graphical quality. The parameters that will be tested to see how much they affect the performance are : Real time rendering, graphics presets : Low, Medium, High, Epic (Realistic crashes Unreal Engine) ; Screen Percentage : 25%, 50%, 75%, 100%, 125%, 150%, 175%, 200%. Real time rendering is responsible for rendering objects with animations, effect on the graphics will be discovered in the testing; Graphics presets will dictate the quality of everything on the screen, such as shadows, lighting, textures, effects and everything else, while Screen percentage simply changes what percentage of pixels from the original image on screen should be rendered.

4.2 Performance tests

Before the testing could begin it is important to explain the methodology of how the tests proceed and how to read the results.

The tests take place on the edge of the map, using an angle from which most the objects are visible without having to rotate the camera to simulate the most performance intensive moments in games, where there are a lot of objects on screen, which often causes games to start lagging.

First test will always involves standing completely still, hands off the camera controls, which in this case is mouse and the camera is overseeing all the objects in one frame. For the lowest and highest framerate to count it must be stable, occurring within the one cycle, so that random frame jumps that occasionally occur from the device occasionally reassigning its resources to the different processes and reevaluating the amount of resources to assign for said processes, in order for the latter to function normally, will not distort the final results. The singular test on average approximately 30 seconds and had 3 stages :waiting for framerate to stabilise, finding commonly occurring highs and lows in framerates, recording accurate framerates or in cases it was not possible, mostly to fps heavily fluctuation, which causes the framerate counter to be unreadable in the decimal values, forcing the results to be rounded down to whole numbers.

Second test's only difference in execution is that instead of camera being completely still, it moves approximately 90 degrees with the middle point being located where usually first test takes place and the border being parallel with either walls in the second map or level edge walls in the first map. The movement is as smooth as possible while doing that manually, so that it could imitate real player moving their camera and inaccuracy caused by low fps input lag does make the final results more authentic to live testing, which is the goal. While automatic rotation could give more consistent results, it is unlikely to change final results too much, as tests are done for as long as they are needed to find consistent patterns in the framerate.

The tables of results start with defining the constants in the test and are read : Map which was tested and then 2 out of 3 variables, the third one is used in the testing and is omitted from the cell. Those variables are, in order of their appearance :Realtime setting, graphical preset, screen percentage (or sometimes also referred as image scaling percentage).

Now the testing process will begin itself, starting with map number 1 : FirstPersonMap.

First aspect that was tested was Realtime setting, as it was the more obscure setting before the testing process and its effect on the performance were completely unknown. Tests were run with real-time setting disabled and then with it enabled.

Immediately after the tests were completed, the results in Table 1 showcased that there was something wrong with framerate with real-time off and the next tests should determine with their result whether it stays consistent on different graphical settings or not. If the results of the idle test are disregarded then the conclusion that could be drawn from this test is that turning off real-time in the first map does in fact increase performance, by small margin, from the lowest increase in framerate of 1 fps to the highest of 4 fps. This is probably due to the map not having anything

substantial to benefit from real time rendering, except for clouds and tests on the second map will probably be able to tell more about the performance intensity of real-time rendering.

Table 1. Results of testing how does Realtime setting affect the framerate.

FirstPersonMap, Low preset, 100% Screen perc. Realtime setting :	Highest framerate while idle	Lowest framerate while idle	Highest framerate wile rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
Realtime ON	36.2 fps	35.9 fps	38.5 fps	37.7 fps
Realtime OFF	120.2 fps	119.9 fps	~42.7 fps	~38.9 fps

As it was stated in the previous aspect test, now it is time to test out how do the results of other aspects differ from having real-time rendering on or off. So, based the test results, written in Table 3, the suspicion of results in the idle test being skewed with were confirmed. This potentially means that once the Unreal engine finishes rendering all the objects in the view, it will stop any updates unless the camera or the object is moved. Stopped updates means that it cannot be used as valid example for game development, as it treats finished render of the map as an image instead of a dynamic world, where updates to the map, such as animations are being done constantly, for the game to feel alive instead of being just simply an image or a video. But that does also prove another thing, that completely still worlds are well optimised, as they only need to update when something happens. This is great for the games that are more image based or text based, but terrible for the games that rely even slightly on dynamic interactions with the world, as this technique become more useless the more movement happens on the screen, as it will be constantly updating everything regardless of whether it is using real-time rendering or not.

The third test will now test the same parameters as the second one but now with the real-time enabled. The results of the third test on the first map written in Table 4 confirm that turning real time on provides more accurate results in the idle tests and in the further tests, even though it could prove something in the moving camera tests, it is still not worth it to have half of the results completely unusable in the research. In addition to that the conclusion drawn in the first test (Table 1) that the framerate difference in moving tests that real-time setting has received another support for itself from comparing moving tests results in Table 3 and Table 4 with in both cases the

difference in framerate being from 1 to 4 fps, excluding only the estimate results, which had much higher difference, but due to it being more early tests, it could be possible that guidelines for testing, written earlier, were not followed, which why those results could be inaccurate for those specific cases. On the other hand, the results without any accuracy issues, do follow that trend with the lowest difference between lowest framerate in epic preset being 0.9 fps and the highest being 2.5 fps in highest framerate in high preset, which could have been higher, if the some of the results in the second test did not have any accuracy problems. Another thing that could have been proven if there were not any inaccuracies is whether the highest framerate has the highest difference between real-time settings and vice versa with the lowest framerate.

For the fourth and the last test on the first map will be seeing how much does the performance change from reducing or increasing the amount of pixels on the screen. The results from Table 5, as expected show that reducing the amount of pixels by half does, somewhat inaccurately, double the framerate and same with increasing the amount of pixels, while it is not accurate enough to estimate that every 25% increase/decrease will be contributing exactly $\frac{1}{4}$ fps of the original 100% in terms of framerate. The other theory that did receive confirmation from tests 3 (Table 4) and 4 (Table 5) is that the highest framerate while idle is almost the exact same as the lowest one during the turning test, albeit with some decimal differences in the framerate. The probable reason for this kind of similarity is that during the process of turning there are a few frames that are exactly the same with the idle tests. This concludes that the rotation tests were done correctly, because they were able to demonstrate the whole spectrum of how much objects' count affect the performance, from rendering the whole room to just some part of it.

Onwards to the second map : Game engine development project (or simply GameDevProject)

The first test on the second map does already show in Table 2 that, compared to the same test in the first map (Table 1) the performance will be lower by a significant margin, but more accurately, about 10 fps in all the cases with the deviation being from as little as 0.1 fps and as much as 1.2 fps for the accurate results.

Table 2 Results of Realtime setting affecting the framerate on the second map.

GameDevProject, Low preset, 100% Screen perc. Realtime setting :	Highest framerate while idle	Lowest framerate while idle	Highest framerate wile rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
Realtime ON	26.1 fps	25.1 fps	~28.6 fps	26.5 fps
Realtime OFF	~120.1 fps	~119.9 fps	~31 fps	~27 fps

The second (graphic preset) test for the second map gives a lot more interesting information than the first one. Considering the results of the same test in Table 6, starting with finding the minimal framerate supported by Unreal Engine being 8.0 fps, as it does not go any lower no matter the conditions and no changes to the camera position would affect that. The exception for that is moving test results on epic preset, which for some reason outputted higher framerate than the high preset. One possible explanation for that kind of a situation is that in order to preserve workability, Unreal started trying to improve framerate, for example, if it detects framerate consistently being under 10 fps, it will ask the user whether it should forcibly improve the framerate to a usable level or not.

Now comparing the result to the other tests, which in this case Table 6 will be compared to Table 4, the most noticeable difference is that on the second map the framerate drops significantly lower from low preset to medium preset, 10.8-12.8 fps difference in first map and for the second one the numbers are 16.6 ; 17.9 and 19.4 fps, if the approximate result is included and it is even more apparent on the Figure 18 (low preset) and Figure 19 (medium quality), where a lot of changes in the lighting affect the performance due to their complexity and overall improvement to casting shadows on objects.

For the last test on the second map, Table 7 has lower overall framerate than Table 5, but their scaling is very similar to each other, which mean that scaling works independently of other factors such as level complexity and is likely completely unaffected by other graphical settings.

4.3 Code influence on performance

The problem with measuring the affect of the code on the performance is often it is not as easy to comprehend the results and decide whether problems come from coding part of the project or graphical side. Low framerates could come from millions of different reasons in the bigger projects. The tests in the earlier chapter did not have any of its performance drops related to the coding, as there was not any code responsible for modifying the level and its objects.

There is an option to add something that would purposefully reduce the performance via code, but there will be way too many different variables that just cannot be standardised for the tests to be fair and clean. For example : where should the object be in the view, how far from the camera, what are the rotation axis of the object, how big the object has to be, how detailed the object will be in triangles and vertices (measurements of object complexity), plus adding that object into already existing graphical tests and finding any difference in framerate, and that is only a fraction of possible variables in the testing. This does not discourage those kinds of tests, but they deserve to be in its own thesis instead.

For the code-related problems that could arise in possible further development of the second map (Game engine development project map) could be making sure that objects that are in direct proximity to the player and in their view should be rendered while objects obstructed by walls or other objects should be left unrendered. Important parts of the process will be to strike balance between how seamless the objects should appear and minimising the amount of the objects rendered to potential save a few fps. This could also be a good test – testing how to better render new objects and remove old ones, their effect on performance, game flow and complexity.

5 Results

Starting from the theoretical part of the thesis, it was found discovered that the reason why the topic of optimisation is so contested, is due to the many different types and genres of the games available to the developer, each one with its requirements and quirks. From something like information-based games, whose gameplay revolts around mostly reading information and then interacting with said information, will it be finding differences between images or choosing how to respond to something in text-based RPG games, these kinds of games will benefit from some graphics, but they are usually already done separately from the game, making most optimising methods completely pointless. But not all games are information-based games, there are of course much more interactive games, which rely on their graphics and for their developers figuring out what role do the graphics play in conveying their idea is essential in delivering a well-rounded product that would be liked by their customers. The developer's decision is further complicated by them having to make an important decision about what should be the game's main focus : will it be realistic environments for the player to feel themselves as a tourist in a completely different place and setting from the one they are currently in ; will it be a test to their skills and reaction speed in the more fast-paced reactive games that have the highest standards of optimisation in order for the game to almost instantaneously give feedback on their actions or maybe other millions of game examples, from which all benefit greatly from optimisation and improved framerate.

The bigger problem lies within trying to balance the graphical quality and good framerate, as many areas of the game, where fps could be improved it requires sacrifices in the graphics unless the developer works extra on creating an illusion, where objects and materials look better than they actually are or the developer spends a lot of time adjusting the settings and testing them until the result is satisfactory. As an example of illusion textures could imitate uneven surfaces such as rocks, without having to render each one of them separately. Illusions can of course be quickly broken if the player is observant, so making sure that those optimisations are thoroughly tested in the live environment, so they could, for example, hide simple graphics while keeping the framerate high.

Simply using the game settings to improve framerate has proven to be problematic as they often are too extreme/too weak and could easily disrupt gameplay or visual fidelity of the game. If the player chooses to use image scaling, then that just causes the game to lose its image clarity, as it simply reduces the amount of pixels on the screen, with the exception being the newest image scaling solutions that attempt to fill empty spaces, sometimes doing so incorrectly and creating visual artifacts, due to them being new. If the player tries to reduce visual quality then alongside loss of good graphics, it could still be insufficient, as it can be seen in Table 6, where significant framerate increase only appears on the lowest setting compared to Table 4, which was done on a much more simpler map and shows much lesser drop in framerate from preset to preset.

Overall, it suggests that improvement the game development via optimisation of game performance and more thoughtful approach to game development is overall a good idea, even though many difficulties will be faced in the development of said improvements, they will still save a lot of time and nerves by preventing future issues.

6 Summary

This topic was very much on top of all the topics the thesis writer wanted to research and does of course support all that time he spent trying to make games work on his weak laptop and computer and somehow managing to have a stable, decent framerate even on the newest releases. This was the reason why he decided to research this topic.

During the process the thesis writer improved his own knowledge about both Unreal Engine and Blender, as some of the material was picked specifically to understand something that was missed from the game development course, such as what do different settings in Unreal Engine do and how much do they affect the framerate and what is the difference between the engines in Blender.

The question that was answered best was the second one, a whole set of tests was created to answer it, with first one and the last one just still with the lingering feeling that not everything was brought up that should have been done. Regardless of the doubts, the writer believes that he answered his questions well.

It is of course disappointing that a lot of the parts of the topics had to be dropped due to the sheer volume of research that this topic requires to get valuable results, and it is of course not the topic that should be picked for Bachelor, it is more fitting for Masters or even PhD in technology as it could be expanded in many ways and a lot more tests are required to get valuable results. If there would be an opportunity to make further research in this theme, it would be a great pleasure to expand it with more different tests, more topics such as lagging while moving, lagging on destruction, crash prevention, ai pathing improvement, in-game debug mode importance and etc.

References

Alton, L. (1. April 2020). *What Kind of Computer Do You Need for 3D Rendering?* Noudettu osoitteesta IEEE Computer Society: https://www.computer.org/publications/tech-news/trends/what-kind-of-computer-do-you-need-for-3d-rendering

Blender. (2024 -a). *3D Viewport - Introduction.* Noudettu osoitteesta Blender 4.1 Manual: https://docs.blender.org/manual/en/latest/editors/3dview/introduction.html

Blender. (2024 -b). *UV Editing - Introduction.* Noudettu osoitteesta Blender 4.1 Manual: https://docs.blender.org/manual/en/latest/editors/uv/introduction.html

Blender. (2024 -c). *EEVEE - Introduction.* Noudettu osoitteesta Blender 4.1 Manual: https://docs.blender.org/manual/en/latest/render/eevee/introduction.html

Blender. (n.d.). *Cycles Design Goals.* Noudettu osoitteesta Blender Developer Documentation: https://developer.blender.org/docs/features/cycles/design_goals/

Epic Games, I. (2024 -l). *Using Transparency in Materials.* Haettu 9. May 2024 osoitteesta https://dev.epicgames.com/documentation/en-us/unreal-engine/using-transparency-in-unreal-engine-materials

Epic Games, I. (n.d. -m). *View Modes.* Haettu 9. May 2024 osoitteesta https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/Viewports/ViewModes/

Epic Games, Inc. (2024 -f). *Optimizing and Debugging Projects for Real-Time Rendering.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/optimizing-and-debugging-projects-for-real-time-rendering-in-unreal-engine

Epic Games, Inc. (2024 -g). *Creating and Using LODs.* Noudettu osoitteesta Epic Dev Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/creating-and-using-lods-in-unreal-engine

Epic Games, Inc. (2024 -h). *Visibility and Occlusion Culling.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/visibility-and-occlusion-culling-in-unreal-engine

Epic Games, Inc. (2024 -a). *Light Types and Thier Mobility.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/light-types-and-their-mobility-in-unreal-engine

Epic Games, Inc. (2024 -b). *Post Processing Effects.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/post-process-effects-in-unreal-engine

Epic Games, Inc. (2024 -c). *Global Illumination.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/global-illumination-in-unreal-engine

Epic Games, Inc. (2024 -d). *Reflection Environments.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/reflections-environment-in-unreal-engine

Epic Games, Inc. (2024 -e). *Optimisation Guidelines.* Noudettu osoitteesta Epic Developer Community: https://dev.epicgames.com/documentation/en-us/unreal-engine/optimization-guidelines-for-umg-in-unreal-engine

Intel Corporation. (n.d. -a). *Game Optimization Methodology.* Noudettu osoitteesta Intel Corporation Web site: https://www.intel.com/content/www/us/en/docs/gpa/user-guide/2024-1/game-optimization-methodology.html

Intel Corporation. (n.d. -b). *How to Fix Your Low Frame Rate.* Noudettu osoitteesta Intel US Web site: https://www.intel.com/content/www/us/en/gaming/resources/how-to-fix-your-low-frame-rate.html

Liu, S.;Kuwahara, A.;Scovell, J. J.;& Claypool, M. (2023). The Effects of Frame Rate Variation on Game Player Quality of Experience. *In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)* (ss. 1–10). New York, NY, USA: Association for Computing Machinery. Noudettu osoitteesta https://web.cs.wpi.edu/~claypool/papers/frame-variation-chi-23/paper.pdf

Oksanen, M. (2022). *3D Interior environment optimization for VR.* [Bachelor's thesis, Turku University of Applied Sciences]. Noudettu osoitteesta https://urn.fi/URN:NBN:fi:amk-2022122031159

Oravakangas, L. (2015). Game Environment Creation: Efficient and Optimized Working Methods. Kajaani, Kainuu, Finland. Noudettu osoitteesta https://urn.fi/URN:NBN:fi:amk-2015120419488

Polydin Game Studio. (10. November 2023). *The Essentials of Video Game Optimization.* Noudettu osoitteesta Polydin Studio: https://polydin.com/video-game-optimization

Thambawita, V.;Ragel, R.;& Elkaduwe, D. (2014). *To Use or Not to Use: Graphics Processing Units for Pattern Matching Algorithms.* ResearchGate. Haettu 9. May 2024 osoitteesta https://www.researchgate.net/figure/CPU-vs-GPU-Architecture_fig1_27022259

Appendix 1: Material management plan

There is no material outside of the thesis, everything that was done was documented in the thesis in its fullest. The images have references to the sources where they have been taken from, the only edits on them were done to combine them together from the same page. That was done in Paint application, and the author assumes no right of original images nor the edited ones. All the images were taken from public sources or by the author from his own pre-thesis products. Agreement on sharing and making the repository public with his teammates from game development was already done before the thesis.

Appendix 2: Testing Tables

Table 3. Test results of using different graphical presets to test their effect on the framerate, with Realtime setting disabled.

FirstPersonMap, Realtime OFF, 100% Screen perc. Graphics Preset :	Highest framerate while idle	Lowest framerate while idle	Highest framerate wile rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
Low Preset	120.1 fps	119.8 fps	~46 fps	~37 fps
Medium Preset	120.1 fps	119.9 fps	~30 fps	25, 6 fps
High Preset	120.1 fps	119.9 fps	21.7 fps	20.0 fps
Epic Preset	120.0 fps	119.8 fps	16.4 fps	15.4 fps

Table 4 Testing the effect of the different graphical presets on performance, with Realtime setting enabled.

FirstPersonMap, Realtime ON, 100% Screen perc. Graphics preset:	Highest framerate while idle	Lowest framerate while idle	Highest framerate wile rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
Low Preset	36.5 fps	34.7 fps	37.1 fps	34.9 fps

Medium Preset	23.7 fps	23.5 fps	25.3 fps	24.1 fps
High Preset	18.2 fps	17.9 fps	19.2 fps	18.4 fps
Epic Preset	14.8 fps	14.2 fps	15.3 fps	14.5 fps

Table 5. Results of testing different image scaling percentages and thier effects on performance.

FirstPersonMap, Realtime ON, Low Preset Screen percentage :	Highest framerate while idle	Lowest framerate while idle	Highest framerate while rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
25%	~85 fps	~83 fps	~93 fps	~85 fps
50%	~65.1 fps	~63.9 fps	~68 fps	~65 fps
75%	~47.6 fps	~46.8 fps	~49.5 fps	~47.6 fps
100%	34.2 fps	33.9 fps	~36.2 fps	~34.8 fps
125%	22.5 fps	22.2 fps	23.6 fps	22.9 fps

150%	16.7 fps	16.5 fps	18.0 fps	17.6 fps
175%	12, 8 fps	12, 4 fps	13, 3 fps	12.8 fps
200%	9.7 fps	9.4 fps	10.5 fps	10.0 fps

Table 6 Results of tests oriented at figuring out how do the different graphics presets affect the performance

GameDevProject, Realtime ON, 100% Screen perc. Graphics preset :	Highest framerate while idle	Lowest framerate while idle	Highest framerate wile rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
Low Preset	26.2 fps	25.9 fps	~28.1 fps	26.3 fps
Medium Preset	9.6 fps	9.3 fps	8.7 fps	8.4 fps
High Preset	8.0 fps	8.0 fps	8.0 fps	8.0 fps
Epic Preset	8.0 fps	8.0 fps	9.7 fps	8.4 fps

Table 7 Results of tests oriented on figuring out how does the image scaling percentage affect the performance.

GameDevProject, Realtime ON, Low Preset Screen percentage :	Highest framerate while idle	Lowest framerate while idle	Highest framerate while rotating ~ 90 degrees	Lowest framerate while rotating ~ 90 degrees
25%	~73 fps	~70 fps	~85 fps	~73 fps
50%	~51.1 fps	~49.9 fps	~59 fps	~51 fps
75%	35.4 fps	34.7 fps	~38 fps	~35 fps
100%	25.3 fps	24.8 fps	~27.3 fps	25.1 fps
125%	17. 6 fps	17.2 fps	18. 6 fps	17.5 fps
150%	12.9 fps	12.6 fps	13.7 fps	12.7 fps
175%	9.2 fps	9.2 fps	9.8 fps	9.4 fps
200%	8.0 fps	8.0 fps	8.0 fps	8.0 fps

Appendix 3 : Graphical Differences of different settings

This is the collection of images that do not fit into half page size and/or require smaller details to be visible in order to understand the differences between images.

Figure 5 FirstPersonMap with Realtime setting disabled, low graphics preset and 100% image scaling setting.

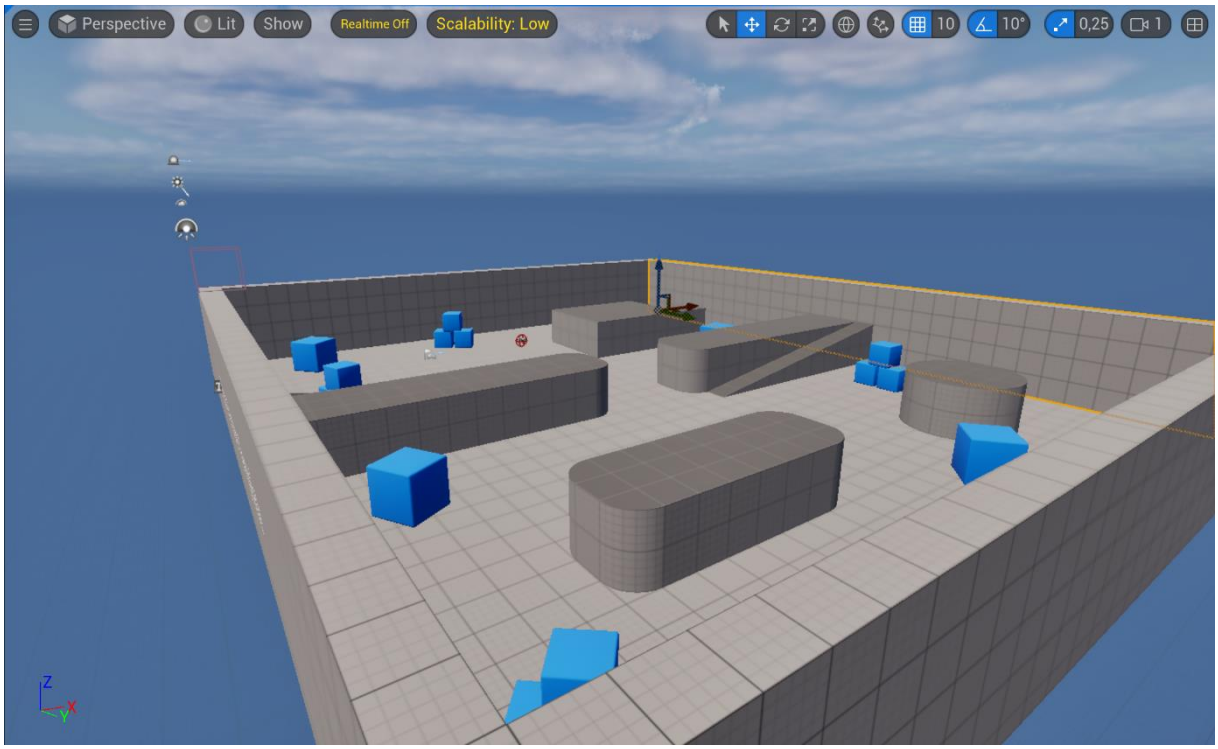


Figure 6 FirstPersonMap with Realtime setting enabled, low graphics preset and 100% image scaling setting.

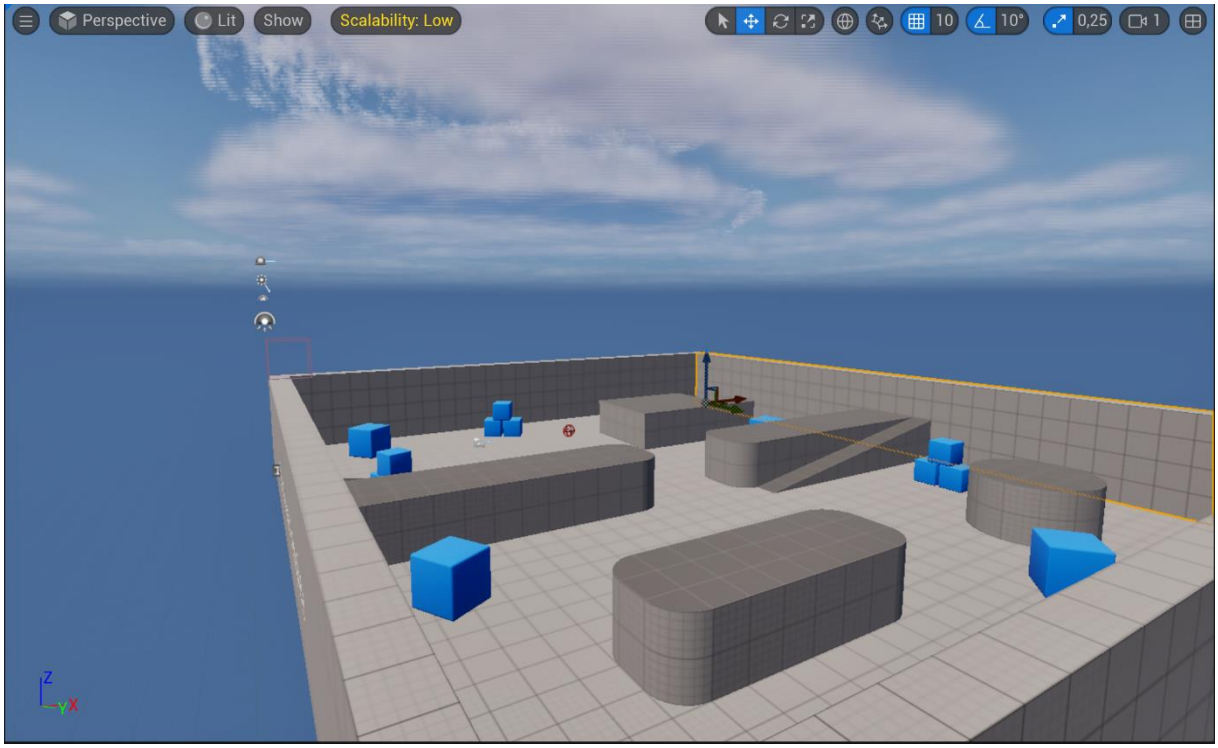


Figure 7 FirstPersonMap with Realtime setting enabled, medium graphics preset and 100% image scaling setting.

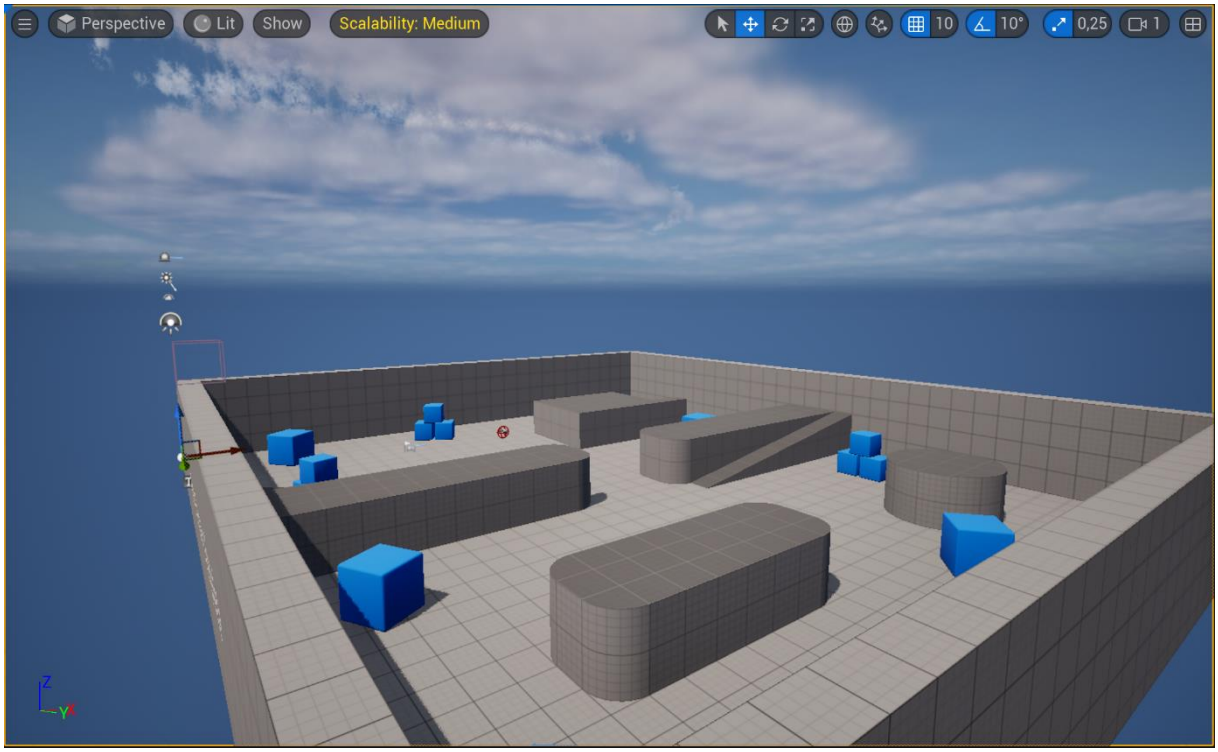


Figure 8 FirstPersonMap with Realtime setting enabled, high graphics preset and 100% image scaling setting.

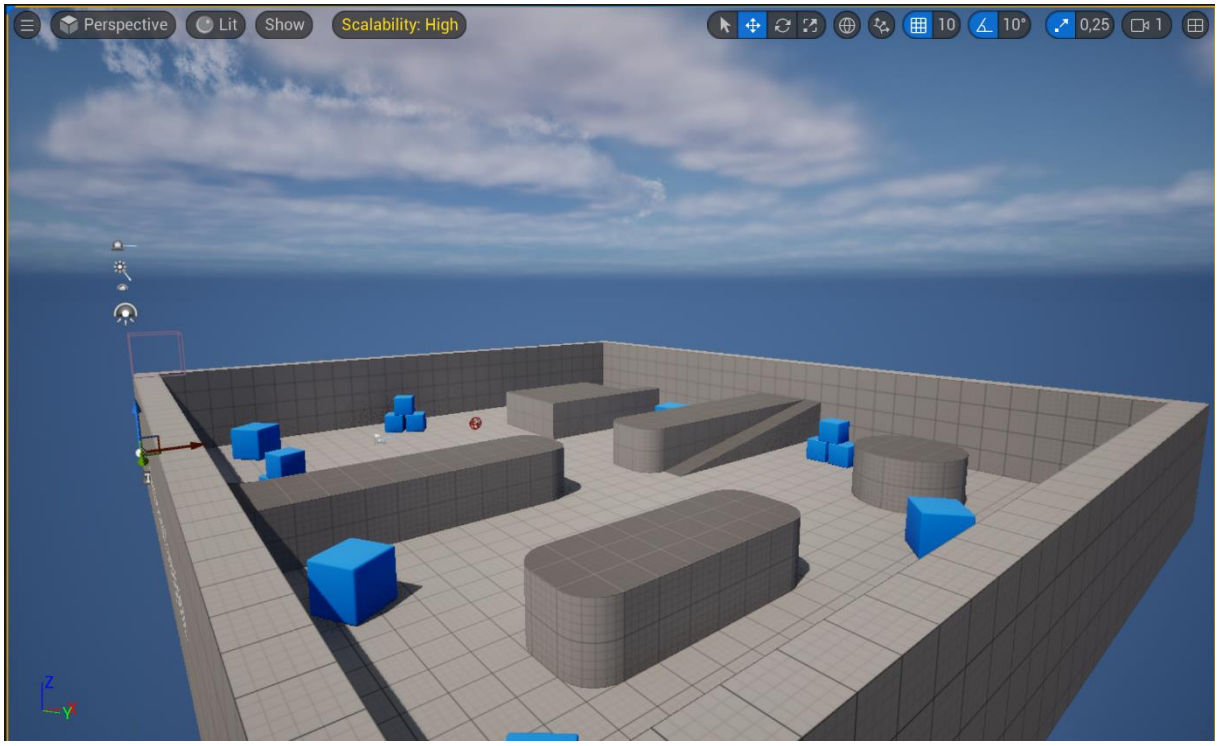


Figure 9 FirstPersonMap with Realtime setting enabled, epic graphics preset and 100% image scaling setting.

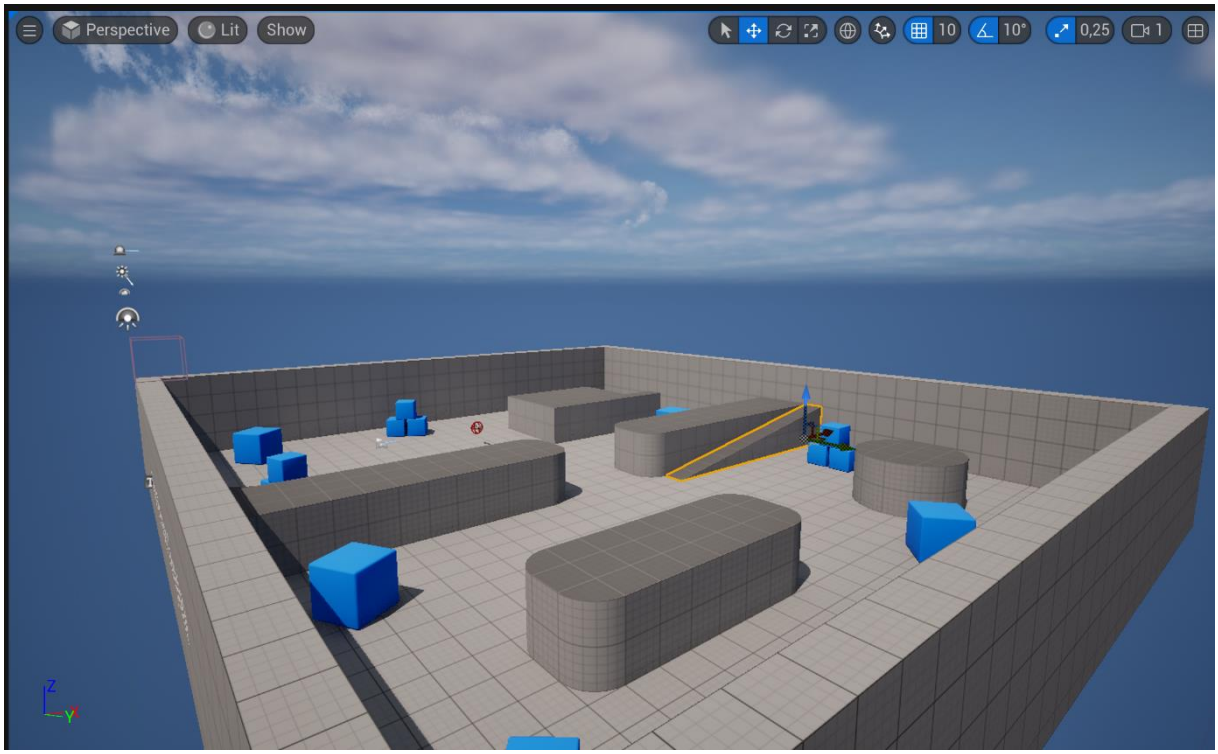


Figure 10 FirstPersonMap with Realtime setting enabled, low graphics preset and 25% image scaling setting.

Figure 11 FirstPersonMap with Realtime setting enabled, low graphics preset and 50% image scaling setting.

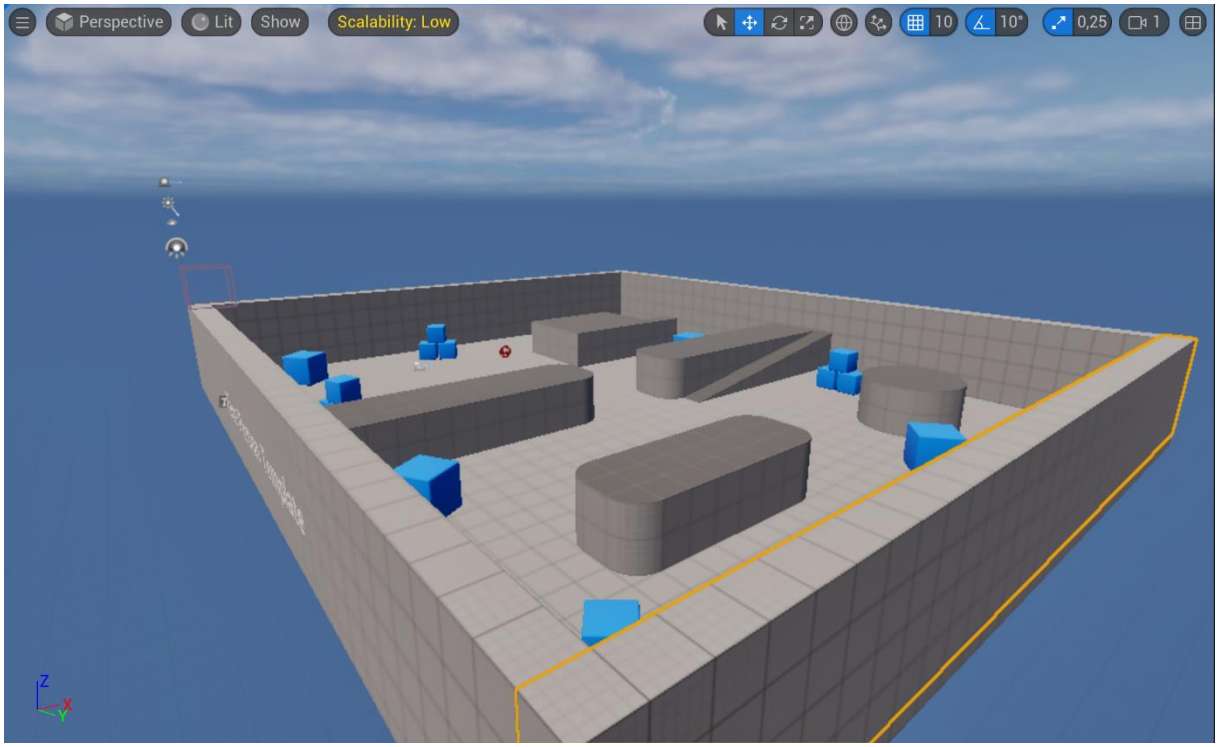


Figure 12 FirstPersonMap with Realtime setting enabled, low graphics preset and 75% image scaling setting.

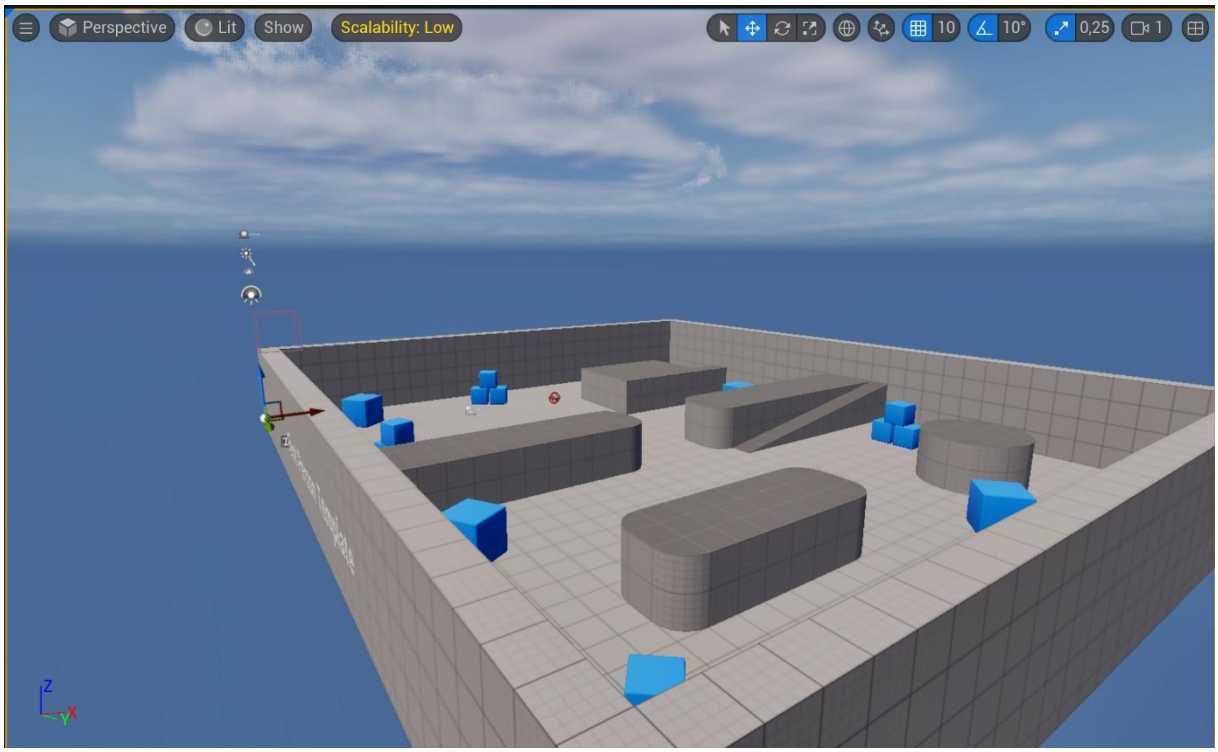


Figure 13 FirstPersonMap with Realtime setting enabled, low graphics preset and 125% image scaling setting.

Figure 14 FirstPersonMap with Realtime setting enabled, low graphics preset and 150% image scaling setting.

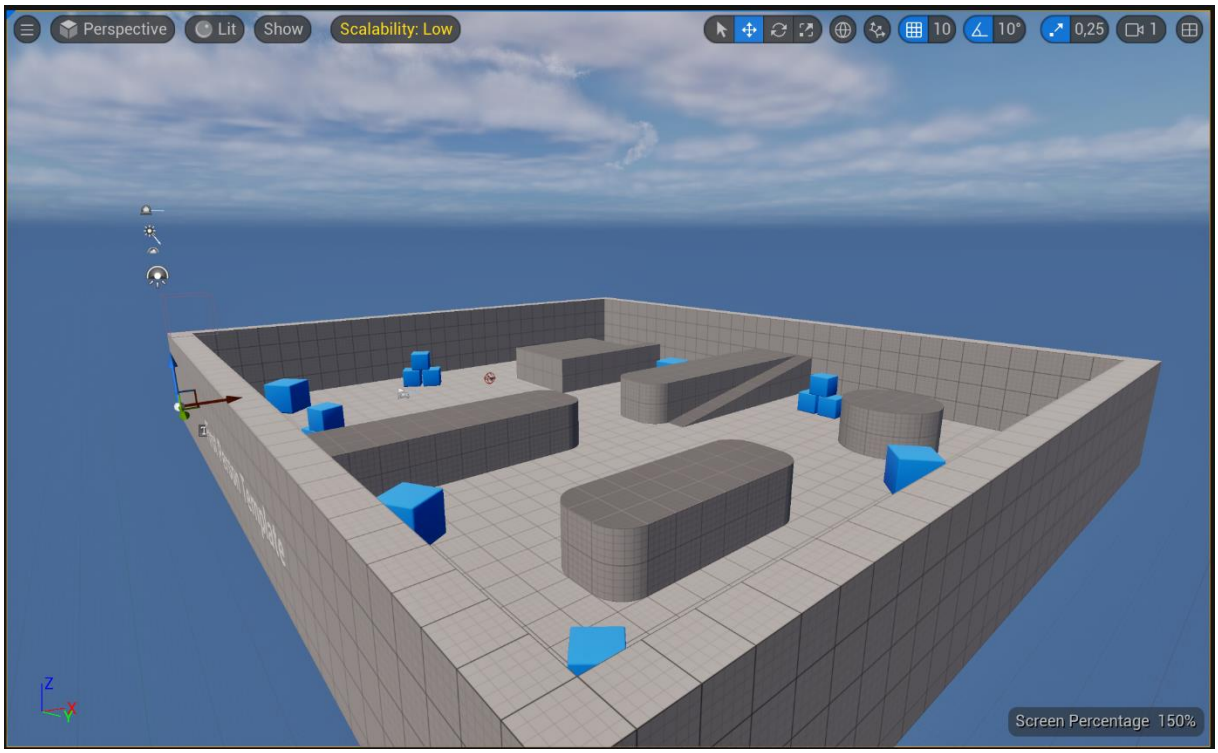


Figure 15 FirstPersonMap with Realtime setting enabled, low graphics preset and 175% image scaling setting.

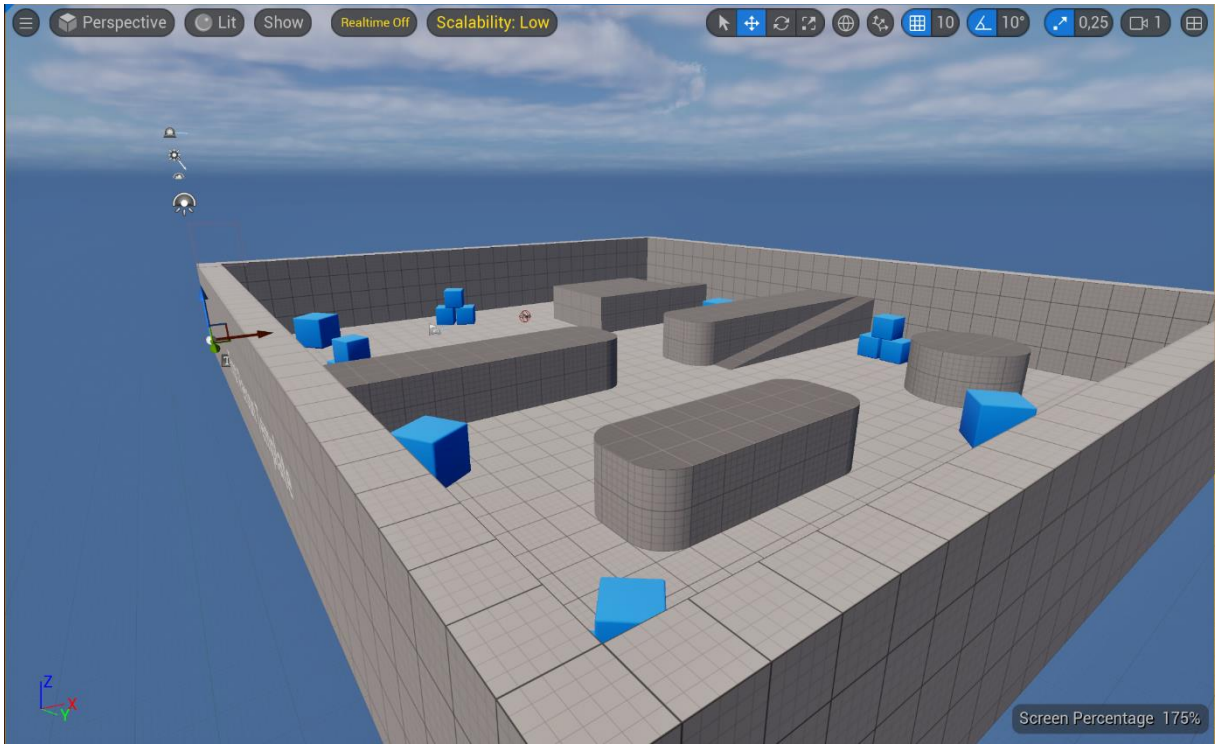


Figure 16 FirstPersonMap with Realtime setting enabled, low graphics preset and 200% image scaling setting.

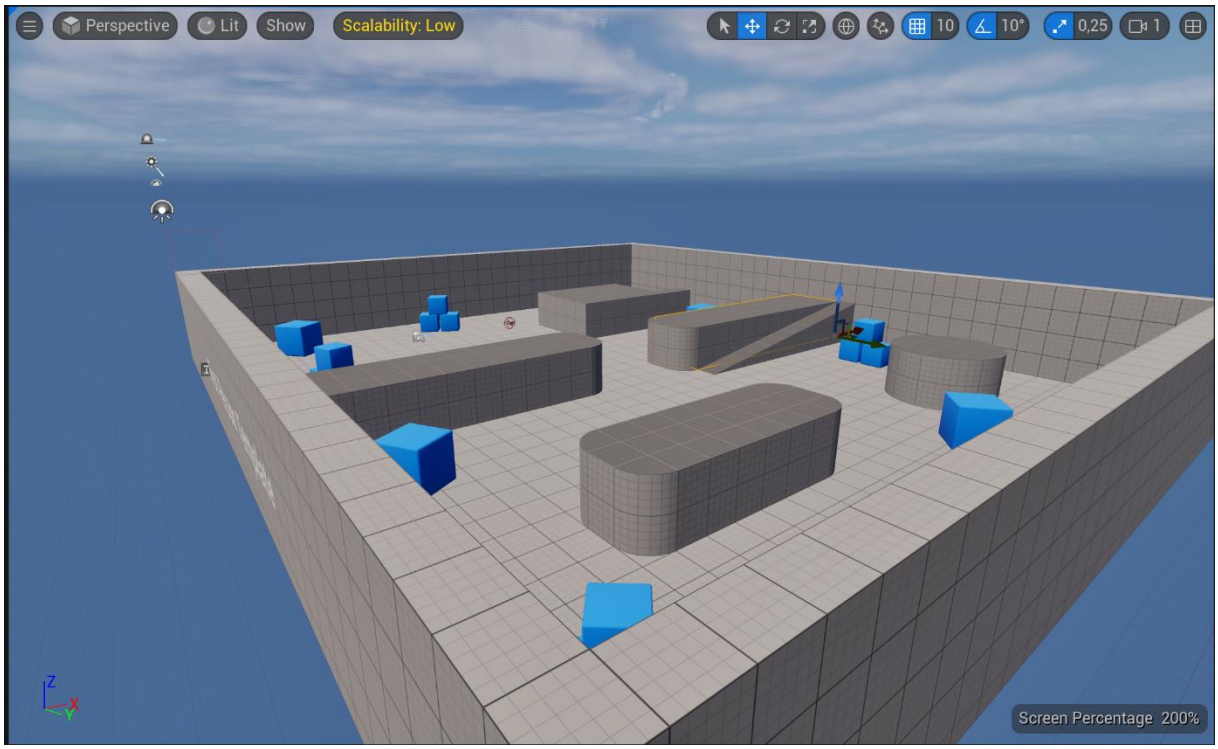


Figure 17 GameDevProject with Realtime setting disabled, low graphics preset and 100% image scaling setting.

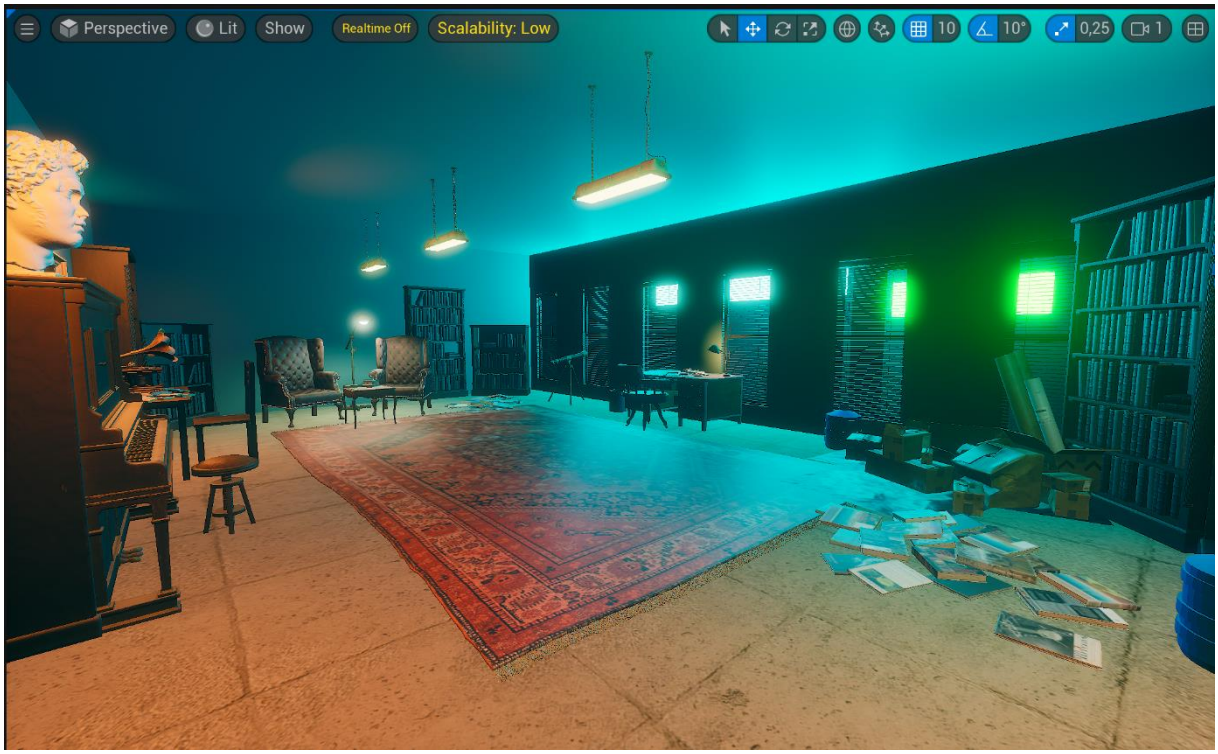


Figure 18 GameDevProject with Realtime setting enabled, low graphics preset and 100% image scaling setting.

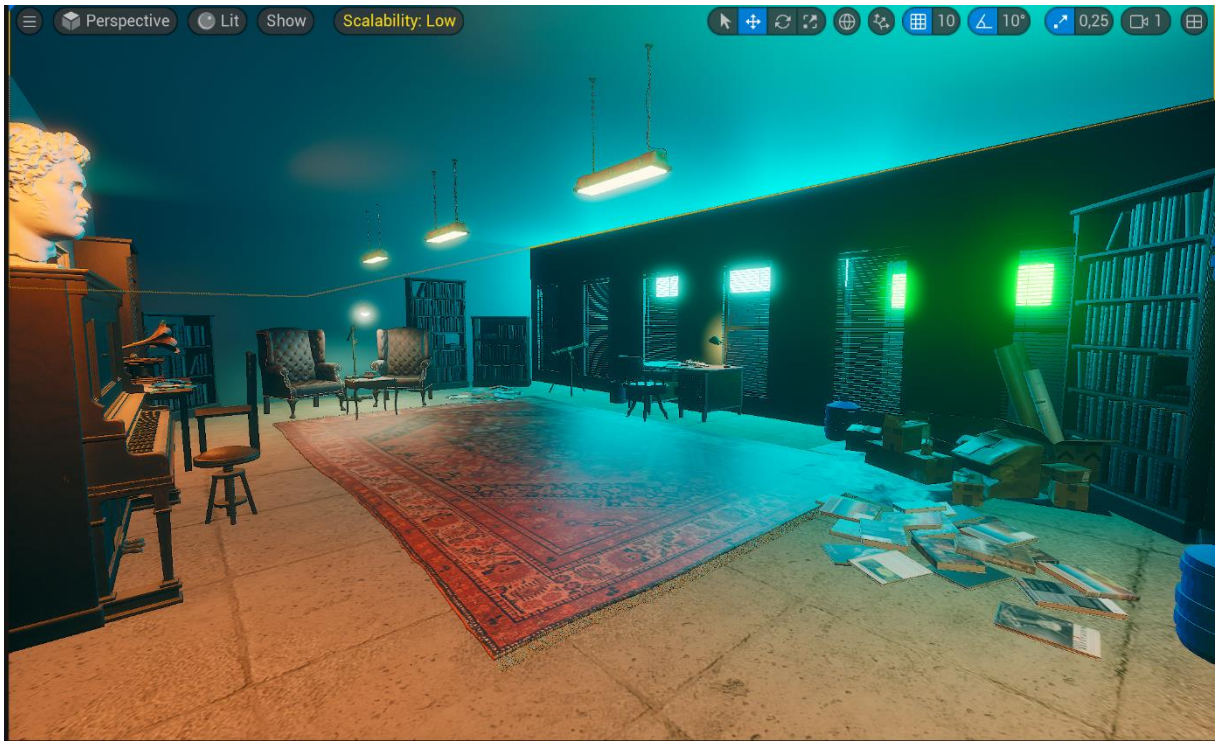


Figure 19 GameDevProject with Realtime setting enabled, medium graphics preset and 100% image scaling setting.

Figure 20 GameDevProject with Realtime setting enabled, high graphics preset and 100% image scaling setting.

Figure 21 GameDevProject with Realtime setting enabled, epic graphics preset and 100% image scaling setting.

Figure 22 GameDevProject with Realtime setting enabled, low graphics preset and 25% image scaling setting.

Figure 23 GameDevProject with Realtime setting enabled, low graphics preset and 50% image scaling setting.

Figure 24 GameDevProject with Realtime setting enabled, low graphics preset and 75% image scaling setting.

Figure 25 GameDevProject with Realtime setting enabled, low graphics preset and 125% image scaling setting.

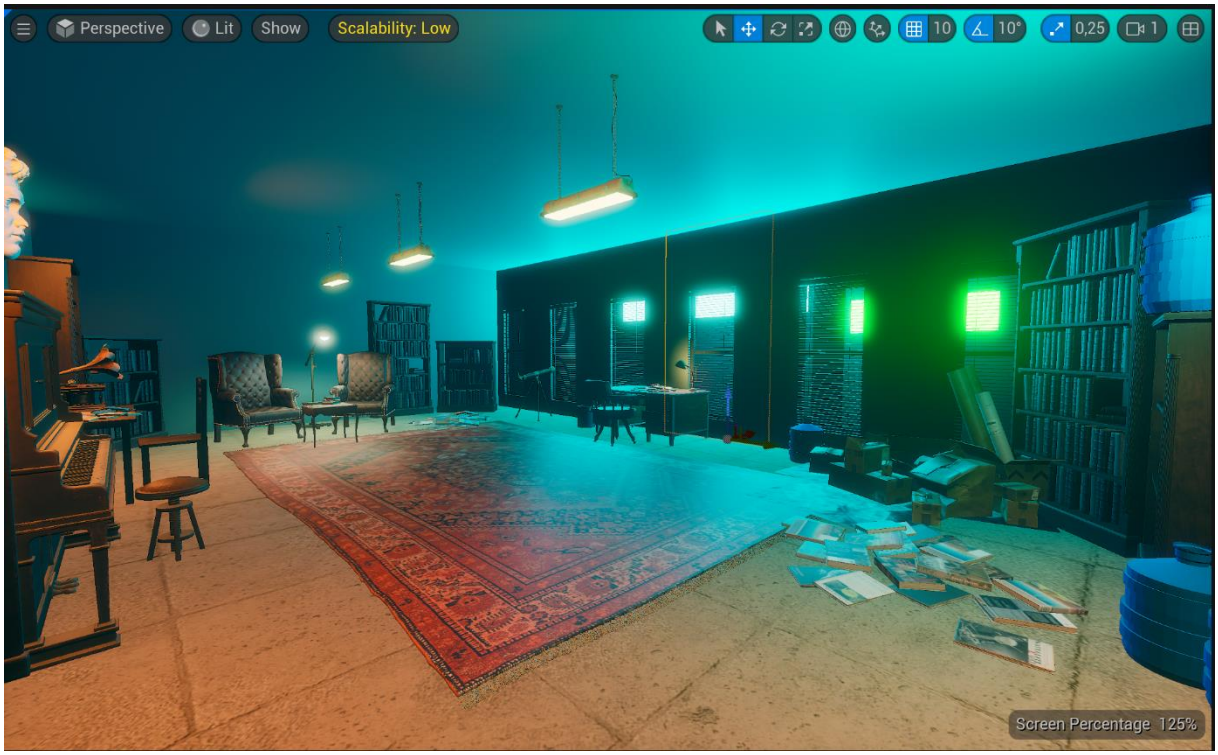


Figure 26 GameDevProject with Realtime setting enabled, low graphics preset and 150% image scaling setting.

Figure 27 GameDevProject with Realtime setting enabled, low graphics preset and 175% image scaling setting.

Figure 28 GameDevProject with Realtime setting enabled, low graphics preset and 200% image scaling setting.