General Guide to Performance in Unreal Engine

by
Gohde Lars

Version: 10.12.2025

Contents

General Guide to Performance in Unreal Engine

Abstract

Performance is a present and necessary topic in every project, because a smooth gaming experience is not just a "nice-to-have," but a basic prerequisite. This work examines various solution approaches for optimizing Unreal Engine projects in general and specifically for VR. This document is designed as a continuous work and is intended to encourage every reader to extend it with their own knowledge.

Preface

Before one begins with the actual optimization, one should address the fundamental perspective on performance. A lecture by Lars Thießen (Senior Game Programmer at Limbic Entertainment GmbH) at the GermanDevDays described this wonderfully with the metaphor of the budget. Every feature costs performance; however, this performance (or "money") is finite. One must learn to budget, to stay within the limits. Some features are more expensive and others cheaper. One must decide for oneself what is worth how much to one and how one manages to nevertheless remain within the given framework.

1. Fundamentals and Objectives

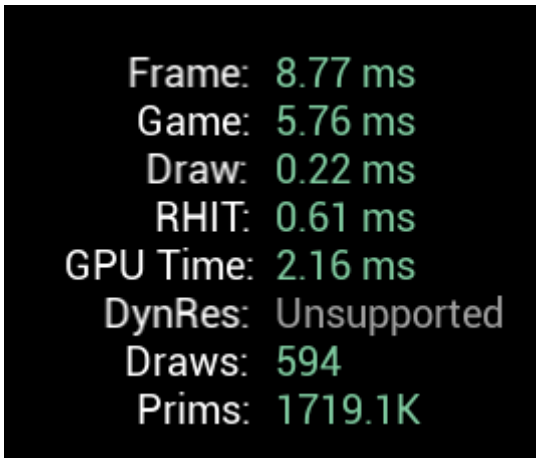
1. **Target Platforms and Performance Budget**

   Defining the objective is the first and most important step. This clarifies the question of whether optimization must continue or if the focus can be directed to other development areas. As soon as one develops for the PC, the objective becomes more difficult, since, in contrast to consoles or VR headsets, one has no exact hardware specifications that all players possess. One's own development computer is also usually unsuitable as a benchmark, as these are mostly high-performance PCs that are required for working with the Unreal Engine. Accordingly, a representative end device should be determined even before development, which serves as a performance reference. Consequently, it is derived from this which

Frames per Second (FPS) should be achieved. For PC, this is, for example, 60 FPS and for VR often 72 FPS for Mixed Reality and 90 FPS for Full-VR. (Tom, 2025)

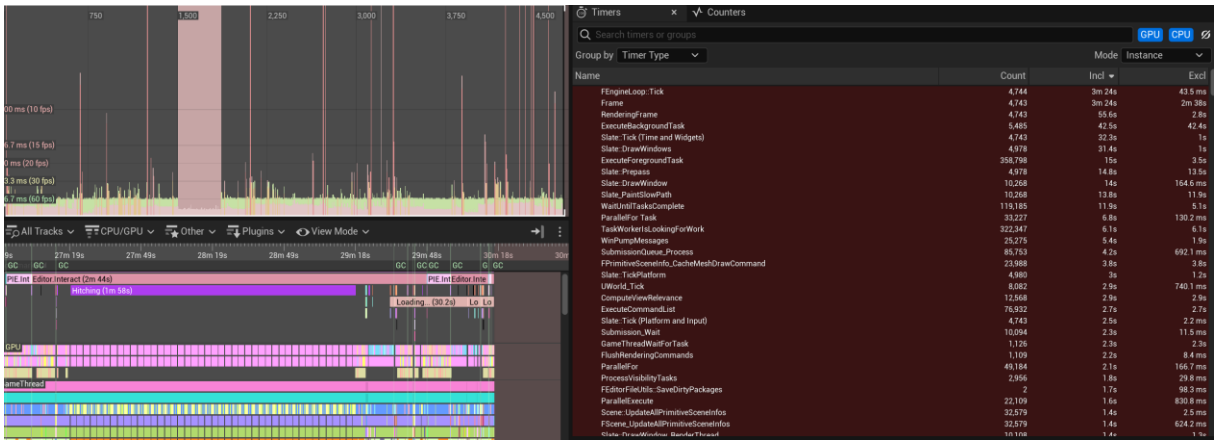
2. Tools and Profiling Methodology

Since the objective is now established, it must be checked which areas need improvements if the performance goal has not yet been achieved. The Unreal Engine serves here as the first point of reference for verification.



For a rough overview, the command "stat unit" can be entered in the console. This is the first indicator of whether the performance problems are a CPU or GPU bottleneck. A bottleneck describes that a hardware component limits the speed of the entire system. Example: If the GPU requires longer (measured in milliseconds, ms) than the CPU, this means that the GPU has more work compared to the CPU and takes longer. The CPU waits for the GPU for this long, which leads to the game running with fewer FPS. (Drake, 2022)

For detailed recordings and deeper analyses, there is the Engine-internal profiler "Unreal Insights". With this tool, one can record the game for a certain time and receive a breakdown of the frame time to see which processes took how long.



As an external profiler for VR platforms like the Meta Quest, the Snapdragon Profiler can be used, as a corresponding chip is installed in these devices. Generally, this profiling data is very important, but often difficult to interpret for laypeople. Fortunately, however, there is extensive assistance and instructions on the web. The obtained data are essential for investigating the specific places where optimization must occur.

In the following, general optimization approaches are found, which should basically be applied in every project to potentially already achieve the defined objective in advance.

2. CPU-Performance (Game Thread)

Should the Game Thread show a too high millisecond time (ms), which points to a CPU bottleneck, the following points should be investigated first:

1. **Ticking and Updates**

   Every Actor possesses the possibility to execute logic in "Event Tick". This function is only processed on the CPU if the tick is activated in code or Blueprint. If the tick node in the Blueprint is gray or removed, no further optimization of the tick is necessary, as it is not triggered.

If the tick is used, however, the following questions arise:

- Do I need the tick permanently?

  Example: A security camera should be permanently directed at the player. But does it have to be if the player is a room away or out of sight? Here, it can be detected through collision detection whether the player

is nearby. The camera can be controlled accordingly, whether the tick should be on or off.

- How high must the update rate be?

  By default, the update rate is set to 0.0 (i.e., every frame). However, it can be set to a higher value (e.g., 0.1 seconds for ten updates per second) to reduce the CPU load. (Drake, 2022)

2. **Logic and Code Optimization**

   This point is of a specific nature but should be checked regularly. Basically, it concerns whether the code logic can be simplified or designed more efficiently.

   Central questions here are:

   - Must the entire logic be executed in the Event Tick?
   - Can multiple nested loops ("Loop within a Loop") be avoided or their complexity reduced?
   - Is the use of Blueprint in critical, high-frequency areas replaceable by C++ to achieve a performance gain?

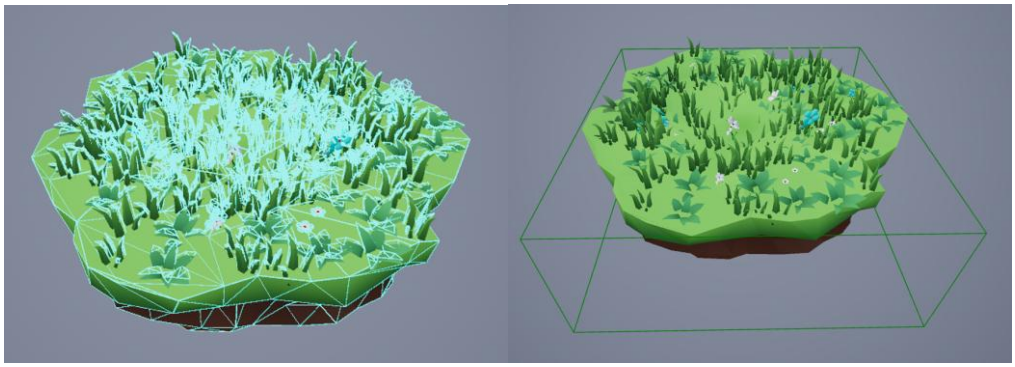
3. **Object Pooling**

   Object Pooling describes the reuse of objects (e.g., projectiles), instead of constantly creating and destroying them. This approach is more cost-effective for the CPU, as the object only has to be initialized once. Instead of destroying the object and reloading it, it is merely deactivated (e.g., placed outside the world or hidden) and reactivated at a new position when needed (by adjusting the Transform and resetting all necessary variables).

4. **Collisions**

   Simple vs. Complex Collision: When one imports a mesh, a shape with more surfaces than a box is chosen by default, although a simple

geometry suffices in most cases. This should be removed and replaced by a simpler one. Furthermore, there is the setting to use Complex Collisions. This takes into account every polygon and is thereby very accurate, but also significantly more cost-intensive.



Collision Channels: Ignoring unnecessary collision channels helps performance. If an object (e.g., a decorative tree) only has to interact with the player, but not with projectiles or the environment camera, these interactions should be set to "Ignore" in the Collision Preset. This reduces the number of collision tests that the Game Thread must perform per frame, as the collision information is only sent to the necessary channels and evaluated there.

3. GPU-Performance (Rendering Thread)

1. **Draw Calls and Geometry Optimization**

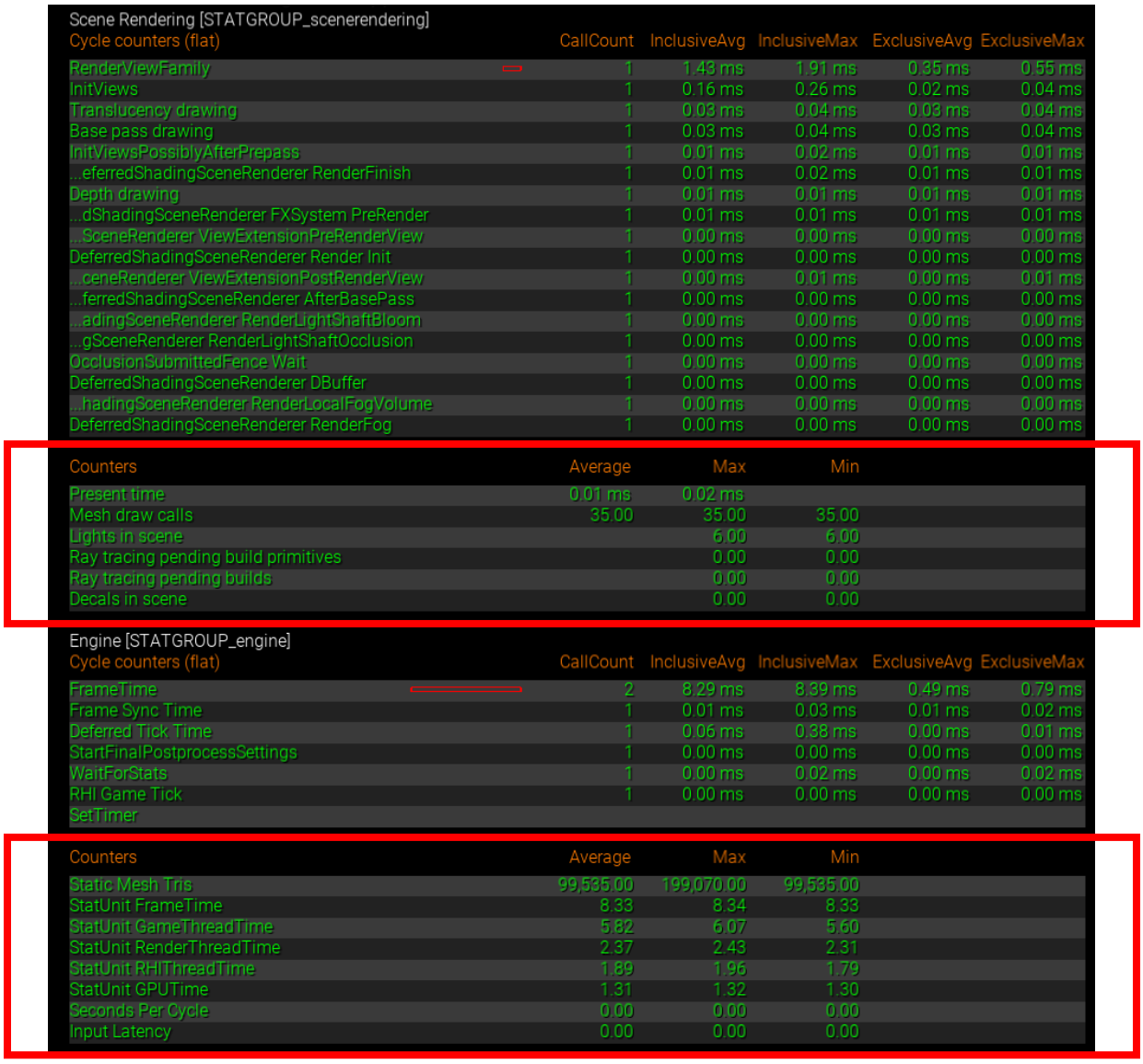
Instanced Static Meshes (ISM) and Hierarchical Instanced Static Meshes (HISM) are the most important point here. Basically, identical geometries are rendered as a single object thereby.

> Example: On a map, there are 100 identical buildings, which are each individual Static Meshes. If one looks only at the Draw Calls for this mesh, this results in 100 Draw Calls. However, if one uses one of the two systems and groups all 100 buildings together, the Draw Call reduces to one, as they are treated as one mesh.

Should one integrate every identical mesh into one of the systems? No. One should do this in chunks and check beforehand which objects are simultaneously in the field of view, to not render unnecessary geometry.

Difference ISM/HISM: ISM renders the mesh always the same and does not support Level-of-Detail (LOD) information. HISM, on the other hand, uses LODs and is thereby, depending on the case, more performant than ISM. (Learn how Drifter Entertainment leveraged elegant optimizations to bring Robo Recall to the Oculus Quest, 2019)

"stat scenerendering" provides information about the draw calls and "stat engine" about the triangles. (Dimitrov, 2024)



Scene Rendering [STATGROUP_scenerendering]

Cycle counters (flat)	CallCount	InclusiveAvg	InclusiveMax	ExclusiveAvg	ExclusiveMax
RenderViewFamily	1	1.43 ms	1.91 ms	0.35 ms	0.55 ms
InitViews	1	0.16 ms	0.26 ms	0.02 ms	0.04 ms
Translucency drawing	1	0.03 ms	0.04 ms	0.03 ms	0.04 ms
Base pass drawing	1	0.03 ms	0.04 ms	0.03 ms	0.04 ms
InitViewsPossiblyAfterPrepass	1	0.01 ms	0.02 ms	0.01 ms	0.01 ms
...eferredShadingSceneRenderer RenderFinish	1	0.01 ms	0.02 ms	0.01 ms	0.01 ms
Depth drawing	1	0.01 ms	0.01 ms	0.01 ms	0.01 ms
...dShadingSceneRenderer FXSystem PreRender	1	0.01 ms	0.01 ms	0.01 ms	0.01 ms
...SceneRenderer ViewExtensionPreRenderView	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
DeferredShadingSceneRenderer Render Init	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
...ceneRenderer ViewExtensionPostRenderView	1	0.00 ms	0.01 ms	0.00 ms	0.01 ms
...ferredShadingSceneRenderer AfterBasePass	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
...adingSceneRenderer RenderLightShaftBloom	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
...gSceneRenderer RenderLightShaftOcclusion	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
OcclusionSubmittedFence Wait	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
DeferredShadingSceneRenderer DBuffer	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
...hadingSceneRenderer RenderLocalFogVolume	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
DeferredShadingSceneRenderer RenderFog	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms

Counters	Average	Max	Min
Present time	0.01 ms	0.02 ms	
Mesh draw calls	35.00	35.00	35.00
Lights in scene		6.00	6.00
Ray tracing pending build primitives		0.00	0.00
Ray tracing pending builds		0.00	0.00
Decals in scene		0.00	0.00

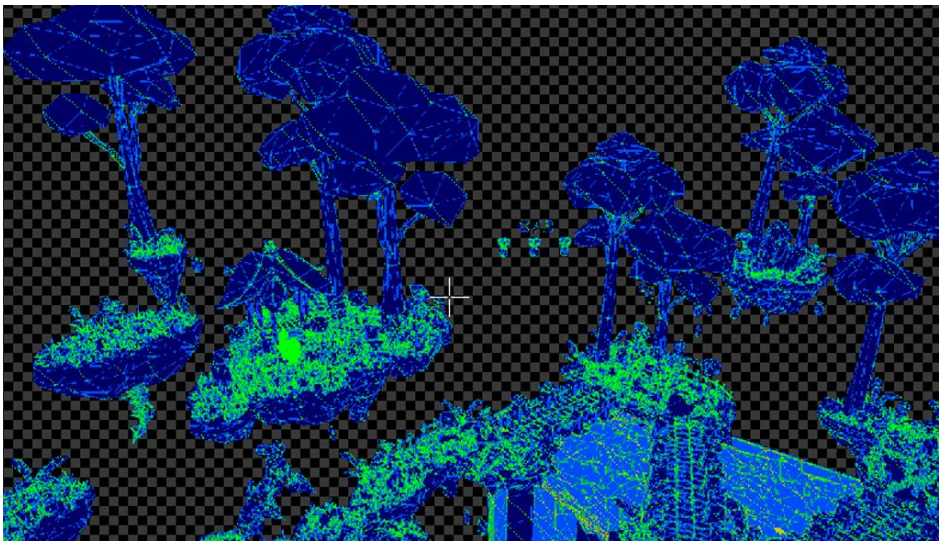
Engine [STATGROUP_engine]

Cycle counters (flat)	CallCount	InclusiveAvg	InclusiveMax	ExclusiveAvg	ExclusiveMax
FrameTime	2	8.29 ms	8.39 ms	0.49 ms	0.79 ms
Frame Sync Time	1	0.01 ms	0.03 ms	0.01 ms	0.02 ms
Deferred Tick Time	1	0.06 ms	0.38 ms	0.00 ms	0.01 ms
StartFinalPostprocessSettings	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
WaitForStats	1	0.00 ms	0.02 ms	0.00 ms	0.02 ms
RHI Game Tick	1	0.00 ms	0.00 ms	0.00 ms	0.00 ms
SetTimer					

Counters	Average	Max	Min
Static Mesh Tris	99,535.00	199,070.00	99,535.00
StatUnit FrameTime	8.33	8.34	8.33
StatUnit GameThreadTime	5.82	6.07	5.60
StatUnit RenderThreadTime	2.37	2.43	2.31
StatUnit RHIThreadTime	1.89	1.96	1.79
StatUnit GPUTime	1.31	1.32	1.30
Seconds Per Cycle	0.00	0.00	0.00
Input Latency	0.00	0.00	0.00

2. Level of Detail (LOD)

Through the implementation of LODs, meshes lose detail level with increasing distance. The Viewport mode "Quad Overdraw" is helpful here. It indicates whether too many polygons are being rendered in a small space (which points to inefficient LODs or unnecessary geometry).



3. Materialien and Shaders

Materials directly influence the number of Draw Calls. Example: A mesh with one material generates one Draw Call, a mesh with three materials generates three Draw Calls. The goal is therefore to reduce the number of materials to the bare minimum.

4. Master Material and Material Instance

To reduce the shader compilation time, one should not create a separate material for every color variation. Instead, a Master Material with parameters is created. Subsequently, one generates Material Instances that only change the parameter values. These instances share the shader data of the Master Material and thereby save compilation time.

5. Settings in the Material

- **Blend Mode:** *Opaque* and *Masked* materials are preferable. In the Material Complexity Viewport, the value should lie in the green area. If transparency is necessary, Additive is more performant than *Translucent*.

- **Shading Model:** *Unlit* requires significantly fewer instructions than *Lit* but does not react to light. Lighting must be simulated in this case.
- **Fully Rough:** Saves further instructions and is especially useful for VR titles.
- **Shading Rate:** Default value is 1x1 and sufficient in most cases - check if this is set correctly. (GDXR, 2023)

6. **Settings in the Project Settings**
    - **Android Material Quality:** Can be globally reduced or deactivated if features are not needed.

7. **Textures**
   Large textures can strongly impair performance, even if the Shader Complexity Viewport remains inconspicuous. A 3000x3000 px texture is, for example, useful for drawing in programs like Procreate, but should be exported in a smaller resolution - as small as possible. Especially color gradients (Gradients) can often be drastically reduced, for instance from 3000x3000 to 30x30 px or smaller, if the gradient is not read directly from the texture, but calculated procedurally using the extreme values.

8. **Lighting and Shadows**

First, one should check "Light Complexity" in the Viewport to recognize and reduce light overlaps. Independent of the position of the light sources, the following factors influence performance:

- **Mobility:** *Stationary Lights* are the fastest to render, but must be baked before the build. *Movable Lights* are the most expensive and should only be used if necessary.
- **Shadows:** Set *Cast Shadows* to Off if the light does not have to cast shadows.

- **Attenuation Radius:** The larger the radius, the more expensive the calculation. If the light is to reach further, Use *Inverse Squared Falloff* can be deactivated and the Light Falloff Exponent reduced. The light remains equally expensive thereby but covers a larger area.
- **Draw Distance:** Set a Max Draw Distance, as the default value is 0 (infinite) and causes unnecessary calculations. The Max Distance Fade Range saves no performance, but ensures a smoother transition when fading the light in and out.

9. VFX

- **Sim Target:** The simulation on GPU is faster. In this case, the Bounds must be set to Fixed.
- **Meshes in Effects:** Effects with complex materials or many material instructions should be avoided.
- **Pooling:** Important for performance and memory management. Even without a fully-fledged pooling system, an effect can be preloaded by saving a reference to an already spawned effect. (PiranhaSauce, 2024)

Explanation: The game manages its resources like a workbench. On the workbench lie all tools that are currently needed — such as hammer, nails etc. (meaning the Actors in the scene). When a tool is no longer needed, no reference to it exists anymore and it is not in the scene, it is taken from the workbench to create space (memory). If the tool is needed again later, it must first be fetched, which costs time. To avoid this, one can save a reference to the effect in a permanent Actor. Thereby the "hammer" always remains on the workbench. This consumes more memory, indeed, but enables immediate access without long loading.

10. Post-Processing

A peculiarity for VR games is that Post-Processing effects are not included in every project. Should one want to use these, however, large performance losses occur. For this, one must activate "Mobile HDR".

4. Memory and Build Optimization

1. **Hard vs. Soft References**
   Hard references are to be avoided if possible. Why? This can be explained with the workbench metaphor: If an Actor has a hard reference, i.e., contains an Object Reference as a variable or uses the node "Cast to...", all referenced objects are loaded simultaneously and placed on the "workbench". This leads to the Actor only being functional when all referenced assets are completely loaded. Thereby the loading time extends, and the cache is filled with unnecessary data. Better in most cases are Soft References. These can be loaded and unloaded asynchronously. However, one cannot work with them directly as with a "Cast" node, but uses Interfaces or other systems instead.

2. **Plug-ins**
   Unreal Engine has many plug-ins activated by default that are not needed. An example here is "SpeedTree". As the name suggests, it is responsible for the import and management of trees in a game, should one determine this. Even in an empty scene, the plug-in is activated or also in scenes that indeed contain trees, but do not access the plug-in. It is accordingly worthwhile to clean the project (after a backup) to improve performance on the one hand and to reduce the size of the game on the other.

3. **Project Settings**
   - **Maps to include:** Specifying maps prevents all unused Assets and Actors from being included during the build. Thereby the build time and file size shorten.
   - **Lumen und Nanite:** Cause high performance costs.
   - **Forward Renderer:** Particularly preferable for mobile platforms, as it is faster, but offers fewer features.
   - **Make Binary Config:** Shortens the loading time.
   - **Full Rebuild:** Recommended for the final version.
   - **Include Prerequisites:** Can be deactivated to reduce the file size.
   - **Mobile Anti-Aliasing:** *MSAA* offers a good balance between quality and performance.

- **Instanced Stereo und Mobile Multi-View:** Recommended for VR games.

Further settings are project-specific and can enable additional optimizations. (Drake, 2022)

4. **Engine**

   Especially for VR development, there is an Oculus-VR-Fork of the engine, which includes improvements in light calculation and Occlusion Culling.

5. Sustainable Workflow

1. Gamedesign

Optimization does not begin only with the implementation of features. Most cost-effective is to adjust the Game Design. The exchange with the group can significantly reduce the load on programmers and artists. Does one actually need 10,000 Orcs in the intro sequence to portray an army or can the feeling also be conveyed with 100?

2. Timing

Optimization is not the last thing one tackles. It should begin as early as possible and one should profile continuously. It facilitates the work if one knows the budget of a scene and works towards it, instead of having to reduce everything in retrospect.

3. Maintainability

Changes, especially global ones, should be documented so that oneself or a teammate is not surprised why features do not work or were changed again.

References

Dimitrov, P. (2024, April 25). *Why Your Unreal Students Project Has Low FPS*. Retrieved from https://petedimitrovart.com/p/draw-call-tips-students-art-level-design

Drake, Z. (2022, Mai 10). *Showdown on Quest Part 2: How We Optimized the PC VR demo for Meta Quest 2*. Retrieved from Meta Horizon: https://developers.meta.com/horizon/blog/showdown-on-quest-part-2-how-we-optimized-the-pc-vr-demo-for-meta-quest-2

GDXR. (2023). *Understanding Textures And Optimizing Materials For Mobile VR in UE 5.1 - Part 1*. Retrieved from https://www.gdxr.co.uk/learning/understanding-textures-and-optimizing-materials-for-mobile-vr-in-ue-51-part-1

Learn how Drifter Entertainment leveraged elegant optimizations to bring Robo Recall to the Oculus Quest. (2019, Mai 24). Retrieved from Unreal Engine: https://www.unrealengine.com/en-US/developer-interviews/learn-how-drifter-entertainment-leveraged-elegant-optimizations-to-bring-robo-recall-to-the-oculus-quest

PiranhaSauce. (2024, Juni 17). *Optimizing Niagara | Measuring Performance*. Retrieved from Unreal Engine: https://dev.epicgames.com/community/learning/tutorials/0qPO/unreal-engine-optimizing-niagara-measuring-performance

Tom. (2025, November 16). *Notes on VR Performance*. Retrieved from GitHub: https://github.com/authorTom/notes-on-VR-performance