

Optimizing Architectural Visualization For Mobile VR Devices

Master Thesis

For attainment of the academic degree of
Dipl.-ing.für technisch-wissenschaftliche Berufe

in the Masters Course Interactive Technologies at St. Pölten
University of Applied Sciences, Masterclass Augmented and Virtual Reality

Submitted by:
Samuel Kreuz
it181518

Advisor: FH-Prof. Dipl.-Ing. (FH) Matthias Husinsky,
Second Advisor: FH-Prof. Dipl.-Ing. (FH) Mario Zeller,

St. Pölten, 30.08.2020

Declaration

- The attached research paper is my own, original work undertaken in partial fulfillment of my degree.

- I have made no use of sources, materials or assistance other than those which have been openly and fully acknowledged in the text. If any part of another person's work has been quoted, this either appears in inverted commas or (if beyond a few lines) is indented.

- Any direct quotation or source of ideas has been identified in the text by author, date, and page number(s) immediately after such an item, and full details are provided in a reference list at the end of the text.

- I understand that any breach of the fair practice regulations may result in a mark of zero for this research paper and that it could also involve other repercussions.

Date: _____ Signature: _____

Abstract

The process of creating architectural visualizations for mobile virtual reality (VR) devices differs in multiple ways from creating conventional VR applications. The thesis focuses on two of the most important factors performance and quality. It concentrates on four aspects, which have a major influence on both performance and quality: scene complexity, shaders and real-time lights, shadow mapping and post processing effects. The question "What strategies can be utilized to optimize high-end architectural scenes for mobile VR devices?" was answered by investigating how computationally expensive each of the aspects are and how they should be tested before implementation. This was achieved by performing four experiments using a mobile VR device and an architectural application specifically designed for these tests. The results show that it is crucial to test the limits of the target device and to decide as early as possible, what aspects need to be the focus of the application. Additionally, the thesis provides a guideline for what potential problems to look for when designing an architectural scene and what profiling tools to use to find performance issues.

Kurzfassung

Der Erstellprozess von Architekturvisualisierungen für mobile virtual reality (VR) Geräte kann sich von konventionellen VR applikation unterscheiden. Diese Arbeit beschäftigt sich mit zwei Faktoren die bei solchen Visualisierungen eine große Rolle spielen: Performance und Qualität. Auf die Aspekte, welche einen großen Einfluss auf beide dieser Faktoren haben, wird genauer eingegangen. Diese umfassen die Komplexität der Szene, Shader und Echtzeitbeleuchtung, Shadow Maps und zuletzt Post-Processing Effekte. Die Forschungsfrage die Beantwortet wurde, lautet "Welche Strategien können angewandt werden um hoch-qualitative Architekturszenen auf mobilen VR Geräten wiedergeben zu können?" Die Frage wurde beantwortet, indem getestet wurde, wie rechenintensiv die genannten Aspekte sind und mit welchen Vorgehensweisen diese getestet werden können. Getestet wurde anhand von vier durchgeführten Experimenten, welche anhand einer mobilen VR-Brille und einer speziell für diese Arbeit erstellten Architekturumgebung durchgeführt wurden. Die Ergebnisse zeigen, dass es wichtig ist, die Grenzen das Gerät, für welches die Applikation designt ist, zu kennen und dass man Designentscheidungen so früh wie möglich treffen muss. Weiters bietet die Arbeit eine Richtlinie, an welche man sie halten kann um Problemen rechtzeitig aus dem Weg zu gehen und eine Liste an Profiling Tools, welche bei der suche, nach Performance-Problemen helfen können.

Contents

1	Introduction	1
2	Theoretical Foundations	5
2.1	The Rendering Process	5
2.1.1	Graphics APIs	5
2.1.2	The Pipeline	5
2.1.3	Conceptual Rendering Phases	6
2.1.4	Tile Based Rendering	12
2.1.5	Mobile- and Desktop Hardware	14
2.2	Draw Calls and Batching	16
2.3	Light and Shadow	17
2.4	Aliasing	21
2.5	Stereo Rendering	22
2.6	Multithreaded Rendering	24
2.7	Profiling	24
2.7.1	Hardware	24
2.7.2	Measure Performance	25
2.7.3	CPU/GPU Bound	26
2.7.4	Profiling Tools	27
3	Method	34
3.1	Testing Methodology	34
3.2	Application Design and Scene Setup	36
4	Results	39
4.1	Experiment Nr. 01: Polycount	39
4.2	Experiment Nr. 02: Shaders and Lights	41
4.3	Experiment Nr. 03: Shadow mapping	44
4.4	Experiment Nr. 04: Post Processing	47
5	Discussion	52
6	Conclusion	56

Appendices	66
A Appendix	66

1 Introduction

Virtual Reality (VR) is booming in numerous market segments. Even though VR headsets are often perceived as gaming platforms for the consumer market, businesses have found benefits in areas like healthcare, education, engineering etc. With VR, the construction and architecture sectors have found new and more immersive ways to visualize their ideas and communicate them to others. The viewer instantly dives into a virtual three-dimensional environment. Traditional static 2D renderings can barely convey how big a room is and what it feels like. VR helps users to get a feeling for how wide or narrow a room is. However, the technology is not only used as a new tool to visualize projects and support architects in the process of construction, but can also help clients to become a part of the design process. Other than with pre-rendered images and videos, VR visualizations let users interact in real time with what they see. Clients can experience their future homes, which have not even been built yet, and make changes right away.

There are numerous different Head Mounted Displays (HMD) out on the market. Depending on which device a developer is aiming for, the workflow of designing an application can differ. Earlier VR devices like the Oculus Rift and HTC Vive are tethered devices, meaning they are connected to a PC via cable. The PC's components like the CPU and the GPU are doing all necessary calculations and image processing, while the HMD simply acts as a display and sensor device. Mobile wearable devices were the next step. The same technology that powers our smartphones is also powering these untethered VR devices. With every new generation of smartphones, mobile hardware is becoming more powerful and so are the all-in-one VR headsets. With the "Quest", Oculus has released its first all-in-one gaming system which includes untethered VR and six degrees of freedom, which means the HMD detects head rotations as well as where the user is positioned in the real world. The considerable benefit of using a mobile device as a tool for showcasing is its convenience, in contrast to being dependent on a stationary high-end PC. These devices are lighter than most laptops, do not take up much space and can therefore be transported easily and without much effort.

However, fully immersive mobile VR applications must be designed for low power consumption, so untethered HMDs can be lightweight, small and, most importantly, free from wires. To still gain the same level of visual quality as with high-end hardware is nearly impossible. This is where optimization becomes relevant. The application must be perfectly optimized

to fit the requirements of the hardware to run at the required framerate. Developers new to mobile VR devices who previously created content with non-mobile VR applications might have difficulties designing such scenes or converting them to be viewable on mobile devices. Creating photorealistic content for mobile devices requires solid knowledge of how the used hardware is constructed and how to design content for it. It is essential to know where problems originate from as well as how to analyse and solve them. As there are a number of ways to model a scene, add texture and light to it, as well as tweak several dozens of parameters, it takes effort to find the right settings for perfect performance.

There are certain aspects which contribute more to the aesthetic of an architectural visualization than others and can influence the appearance of an environment significantly. Next to the materials and colors, which are chosen during the design process, light is one of them. Both a natural light source like the sun or artificial interior light can increase realism when mindfully implemented into the virtual scene. Seen from a more artistic view, lights can serve as artistic tools. For example, in architectural visualizations light falling into a room can easily be exaggerated by increasing its brightness. A similar technique is used by photographers by over-exposing a poorly lit room. This can help to create a brighter looking environment. With light comes shadow, which in some cases can be expensive to compute. Furthermore, bloom effects and the use of strong colors by applying color grading have become standard for architectural renders and are therefore perceived as modern and noble.

Another aspect to consider when trying to improve immersion is detail. If a virtual environment looks too simple and is modeled by few polygons, the scenery is likely to look fake rather than realistic. Details can on the one hand be achieved by simulating them with materials and textures. This however has the disadvantage to become noticeable whenever the user closer inspects an object or looks at surfaces from a flat angle and are also expensive to compute. The second solution to this is creating details by actually modelling them. This will achieve the highest level of realism because the quality is not depending on the viewing angle. There are dozens of ways to increase performance and a lot of small changes may have a large impact. Trying to cover all of these aspects would go beyond the scope of this paper which is why it focuses on a limited number of aspects. The thesis concentrates on the few aspects which will lead to the most realistic looking visualizations, which are light, shadow, detail and visual effects as well as performance.

Many consumers expect console quality and long-lasting battery life on mobile devices. High-end PCs have the advantage of having large cooling systems and receiving continuous energy from large power supplies. According to a "Mobile App User Survey" from 2015, app users are quite intolerant of issues and will quickly abandon an app if it does not reach the performance expectations. Nearly all users saw the speed and responsiveness of applications as important. Therefore the satisfaction with an app is directly correlated with its

performance (Techbeacon). With virtual reality, performance becomes even more relevant since it can induce simulator sickness. Creating mobile experiences as long-lasting and as performant as high-end devices is a challenging task.

The thesis aims to study these characteristics to provide a guideline for designing and profiling an architectural visualisation application for mobile virtual reality (VR) devices. The thesis constitutes a beneficial resource for CG-artists and designers alike, as it provides compiled information about how the rendering process works, and which tools can be utilised to reach maximum quality and performance for such applications. The results not only benefits CG-artists and designers, whose job it is to design and model these architectural environments, but also developers.

It is better to design a scene with previously defined targets like the complexity of the scene, rather than modeling a scene regardless of performance limits and spending time on optimization later on. However, if an application already exists for high-end devices, the paper still conveys knowledge when trying to optimize such projects for mobile hardware. The reader can learn more about how to efficiently analyze applications and how to overcome the challenge of porting applications from high-power hardware to small mobile devices. All in all the thesis will give a deeper understanding of how mobile VR devices work and will teach techniques to be utilized to achieve the best possible quality and performance, starting at the very first design steps. The research question to be answered is: “What strategies can be utilized to optimize high-end architectural scenes for mobile VR devices?”. In the process of doing so, by running a number of experiments, the question of “What tools are useful when tracking down performance problems?” will be answered.

The experiments conducted in this thesis will include an architectural scene, specifically created for these tests. The scene will be created in Autodesk Maya and later be transferred to the Game Engine. The Game Engine, Unity by Unity Technologies will be used for implementation and testing of the scene. The experiments will be executed using numerous Profilers to analyse the application and locate where problems might occur. Profilers are essential analysis tools, which help to find bottlenecks and performance issues in applications. They collect information about each individual step, necessary to create the final image, for later evaluation. The experiments demonstrate how resource-intensive scene complexity, real-time lights and shadows, shadow maps as well as post processing effects can become. Multiple tests are run on the Oculus Quest VR headset to find out what is possible and where performance limits are reached. The results are respectively discussed after the tests and are summed up as a conclusion in the last section of the paper.

The first section of the paper, Theoretical Foundations, aims to deliver a basic knowledge and useful terminology for a better understanding of further subjects regarding the functionality of mobile hardware and software. It gives an overview on some of the most important topics. These will show how the underlying technologies and algorithms are utilized to

draw high quality images on the screen. It shows how traditional render pipelines were designed, why some might lead to problems on mobile devices and how these problems were solved. The major differences between PC hardware and mobile hardware regarding power and cooling are discussed and the basics of rendering techniques are explained. These cover light and shadow creation, stereo rendering and multithreaded rendering.

The second section will take a closer look at one specific use case of a VR visualization. Therefore the hardware used for the performance tests, Oculus Quest, will be examined and some profilers are introduced. The profiling tools can be used to analyze applications and are therefore the key instruments when optimizing. The test setup is described in detail for each of the experiments on the subjects of Polycount, Shaders, Post Processing and Shadow Mapping. In the final section, the test results will be analysed and discussed.

2 Theoretical Foundations

2.1 The Rendering Process

2.1.1 Graphics APIs

To fully understand how the soft- and hardware of any device share data, it is good to know what exactly an API is and why it plays an important role in making applications run. An API is an “Application Programming Interface”, which in basic terms allows applications to communicate with each other. Graphics APIs are the link between the application and the GPU and are used to control and program the GPU. With the fast development of GPUs, APIs evolve with them. Unity supports numerous low-level APIs such as DirectX (Direct3D), Metal, OpenGL and Vulkan. Which Graphics API to use, depends on, which platform the app developer is aiming for. Under the hood, some APIs might work differently. For example, OpenGL favours the cartesian coordinate system, treating the lower left corner of an image as its origin, while Direct3D sometimes defines the upper left corner for it. However the difference they make regarding performance is minor. Vulkan is the most recent API designed by Khronos and is the successor of OpenGL ES. Even though Vulkan has some benefits over the OpenGL API, Oculus does not recommend using it at the time of writing. It is still in an experimental stage, which is why OpenGL ES 3.1 is used for this project. (Luna 2016, April 19; Eising 2017, December 7; Akenine-Möller et al. 2018; Bedekar 2020, February 4)

2.1.2 The Pipeline

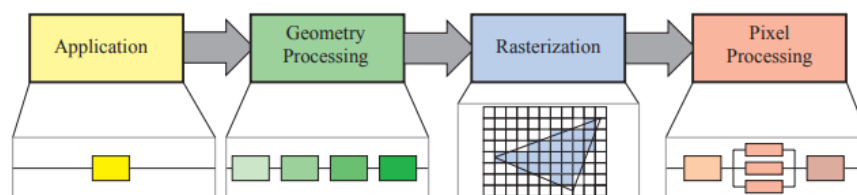
The core component of real-time graphics is what is called the “graphics rendering pipeline” or simply the “pipeline”. Rendering pipelines can be discussed on two different levels, the software API and the hardware implementation level. 3D graphics APIs (like OpenGL, Direct3D, Vulkan, Metal) provide a logical framework showing how a 3D scene is rendered. These functional stages describe a task which needs to be performed but does not describe the way this task is executed. This logical framework does not represent the actual hardware implementation on a device. The real hardware implementation level would describe the actual rendering instructions running on the GPU or the CPU. Different devices

may have different solutions to use their computing units and therefore the render pipeline on the hardware level will always be different to the logical framework of the API. The following section will cover the logical framework of the traditional rendering pipeline. It gives a basic understanding of how these pipelines were and are used on most devices. In later sections we will see how the pipeline was implemented for the use in mobile devices. (Cozzi and Riccio 2012; Akenine-Möller et al. 2018)

2.1.3 Conceptual Rendering Phases

The rendering pipeline is a fundamental concept describing the different stages data has to run through when being rendered. The main function of the pipeline is to take information like three-dimensional objects, light sources, camera position and angle, materials, textures etc. and use them to generate a two-dimensional image. It consists of several stages where each has a specific purpose and every stage works with the outcome of the previous one. Passing data from one stage to the next happens at the same time for every stage. Therefore they execute in parallel. If one stage is slower than the following ones, they are stalled until the data is finished processing. Since the slowest stage decides the speed of the whole pipeline, it is called the “bottleneck”. (Akenine-Möller et al. 2018)

The rendering pipeline can be divided into four stages: application, geometry processing, rasterization and pixel processing. The stages can consist of several sub-stages which can be pipelines themselves. This structure is the core of most real-time computer graphics applications and will therefore be described closer in the following chapters. (Akenine-Möller et al. 2018)



Source: Akenine-Möller et al. 2018, S. 12

Figure 2.1. Construct of a render pipeline with its four stages. Each of them may be a pipeline itself.

Stage 1: Application Stage

The first stage in the process is driven by the application itself and runs entirely on the CPU (central processing unit). CPUs usually include multiple cores, which process multiple threads in parallel. A later section will cover the topic of multithreaded rendering in detail. A

bigger number of cores lets the application stage compute a larger variety of tasks. These tasks could be

- application logic
- inputs from keyboard, mouse, sensors
- collision detection
- global acceleration algorithms (eg. culling algorithms)
- animation
- physics simulation
- network
- sound etc.

Because the application stage runs entirely on the CPU, the developer has full control of the process. The stage can also be used to decrease the number of triangles to be rendered in later stages via algorithms. For the pipeline the most important purpose of this stage is to identify potentially visible meshes, materials as well as light positions etc. Finally, all the data that might end up on the screen, is passed to the next stage. (Akenine-Möller et al. 2018)

Stage 2: Geometry Processing

The geometry processing stage deals with per-vertex and per-triangle operations and is the first step on the GPU. It can further be divided into four stages: **Vertex Shading -> Projection -> Clipping -> Screen Mapping**

Vertex Shading

The vertex shading (VS) stage includes two tasks, computing the position of each vertex and evaluating what the programmer wants as output data (eg. normal or texture coordinates). Traditionally vertex shading was, as the name suggests, a per vertex shader. Today the vertex shader is more of a general unit, to set up the data associated with each individual vertex and to transform the vertex positions. (Akenine-Möller et al. 2018; Alfonse 2019, April 8)

At first, the vertex position needs to be computed. On its way to the screen a model has to be transformed several times into several different *spaces* or *coordinate systems*.

- Model Space: this is the original space, the model is in, without being transformed

- Model transform: each model has a model transform to be positioned and oriented. When using instancing, a model can even have more than one Model transform
- Model coordinated: these are the coordinates an object has

After applying the model transform to the model coordinates, it is said to be in *world coordinates* or in *world space*. The world space includes all models after they have been transformed with their respective model transforms. Because it is only necessary to render objects which lie inside the area the virtual camera (and therefore the observer) sees, only those models are processed. Just like all other models, the camera has a position and a direction in world space. To make clipping and projection operations simpler, the view transform is applied, which transforms the whole scene for the camera to be at the point of origin. (Akenine-Möller et al. 2018)

To create realistic looking images, the shape and position of objects is not sufficient. We need to compute every object's material and the effect of how light shines on an object. There needs to be a description of how the materials and lights are calculated. These can be rendered in multiple ways, from simple colors to physically correct representations. The operation to determine what effect light has on a material is called shading. Shading is done by executing a shading equation on several points of an object. Depending on the shader, some can be processed during geometry processing, others may be performed during per-pixel processing. (Akenine-Möller et al. 2018)

In pipelines such as the OpenGL render pipeline, some additional but optional stages can be added after the vertex shader such as the Tessellation Shader, which can subdivide vertex data into smaller primitives. (Akenine-Möller et al. 2018)

Projection

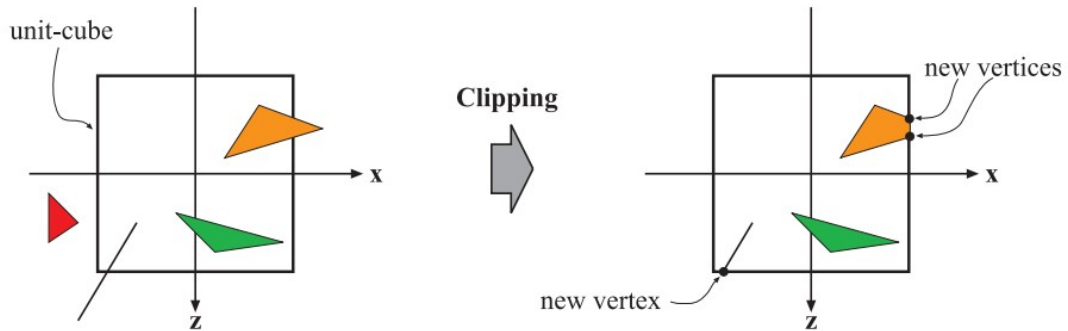
There are two commonly used projection methods, orthographic projection and perspective projection. Orthographic is one of several parallel projections. With the perspective projection, objects which are further away are smaller than objects nearby. The perspective projection mimics the way humans perceive the world and is therefore more important for us. The view volume, also known as frustum has the shape of a truncated pyramid and includes all objects which will be seen on screen.

Before clipping, some GPUs perform additional optional stages like tessellation, geometry shading or stream output. The use of these stages depends on the GPU capabilities and the desires of the programmer. (Akenine-Möller et al. 2018)

Clipping:

When rendering an image, only the previously generated primitives fully or partially lying inside the frustum need to be passed on to the rasterization stage. If a primitive lies fully inside the viewing frustum, all its data will be passed on. Primitives completely outside will

not be passed on because they are not visible. The primitives that are partially inside the view must be clipped. Primitives are always clipped against the unit cube/view volume, defined in the previous stage. When combining multiple objects to one while modeling a scene, there are some aspects to consider, which are discussed in the section 2.2.

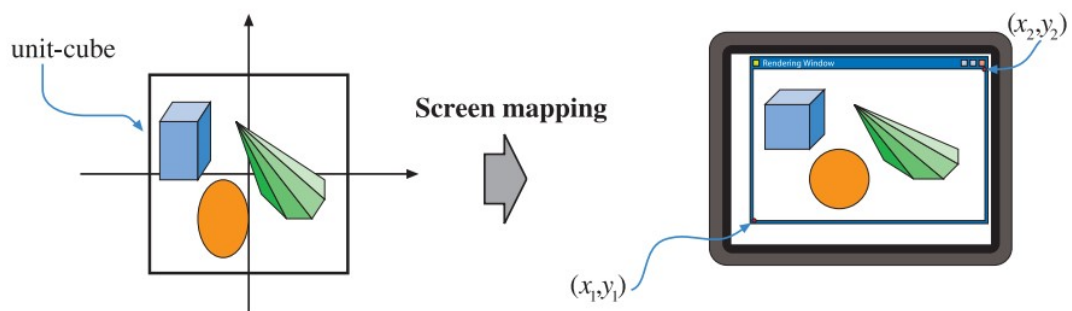


Source: Akenine-Möller et al. 2018, S. 20

Figure 2.2. Only the primitives inside the view frustum are kept. The ones outside are discarded. If a primitive lies inside as well as outside of the view frustum, it is clipped and new vertices are generated.

Screen Mapping:

Now that all potentially visible primitives are passed to this stage, they can be converted to two-dimensional space to create the “screen coordinates”. Screen Mapping is the process of finding the right coordinates on the screen. The X, Y coordinates are translated and scaled from a range of $(-1,1)$ to actual pixel coordinates on the screen. The Z coordinates, which describe the depth values are also remapped and passed on to the rasterizer. (Teschner 2016, January 13; Akenine-Möller et al. 2018)



Source: Akenine-Möller et al. 2018, S. 20

Figure 2.3. With Screen Mapping the correct screen coordinates for each vertex are found.

Stage 3: Rasterization

The rasterization stage has the task to find the correct color for every pixel. using the transformed and projected vertices as well as the additional shading data, coming from geometry processing. It finds all pixels which are “inside” of/overlap with triangles and renders them onto the screen. The way to determine if a triangle “overlaps” with a pixel depends on how the GPU pipeline is set up. One of the simplest methods is to use a single point sample in the center of each pixel. If this center point is inside the triangle, the pixel is considered being inside the triangle as well. Using one sample per pixel may lead to an artifact called “aliasing” which is why supersampling or multisampling aliasing techniques are commonly used. For every part of a pixel, which overlaps with a triangle, a fragment is generated. Fragments contain all information to paint the surface of the polygon, on the position of the pixel. These informations can be color, z-depth, texture coordinated, etc. Only the samples where a pixel overlaps a primitive, and which might therefore be seen in the final render, are sent to the next stage, pixel processing. (McCormack and McNamara 2000)

Strage 4: Pixel Processing

At this stage, all per-pixel or per-sample computations are performed on those screen pixels which overlap with primitives and will therefore be seen in the final render. The stage is divided into two parts:

1) Pixel Shading

In general, shaders are small programs which are executed on the GPU and run in parallel. The shader takes input data, containing all shading information and performs the shading process on a per-pixel level. Therefore, the program has the task to calculate the final color of each pixel. The result is one or more colors per pixel which are passed to the next stage. To do so, the so-called pixel shader (known as fragment shading under OpenGL) has to receive data like light, shadows, color of the light, etc. for the calculations. If objects have textures applied, these image files are also included into the process. The pixel shading stage is a programmable stage, meaning the developer can create self-written programs for pixel shading if he wants to. Over-complex shaders are a common cause for high performance costs. The more data the shader has to process, the longer it will take to process a single pixel. Unity has special simplified shader models designed for the use on mobile devices. These are usually designed in a simpler way, which makes them fast to compute at the cost of quality, compared to a high performance shader. When creating own shaders, it is recommended to keep them as simple as possible and perhaps move code from the pixel shader to the vertex shader. The advantage of self-designing shaders is for them to perfectly match the environment they are designed for. (De Vries 2017;

Unity-Technologies 2020e, August 25)

2) Merging

The very last step in the rendering process is the merging stage where all data comes together. Other than the pixel processing stage, the merging stage is not programmable. At this stage of the pipeline, it is still not known if a fragment is in front or behind another object, seen from the perspective of the camera. When the scene has to be rendered, the color buffer should only contain colors of the objects actually visible by the camera. This is why this final stage measures the distance from the camera to each fragment. If it is behind another object, the color value is discarded, if it is in front, it is written to the buffer. In most cases, the depth value to do this is found using a z-buffer algorithm. The z-buffer stores the z-value of the closest primitive for each pixel. The images below show an example of the final render and the stored depth values used to create it. (Akenine-Möller et al. 2018)

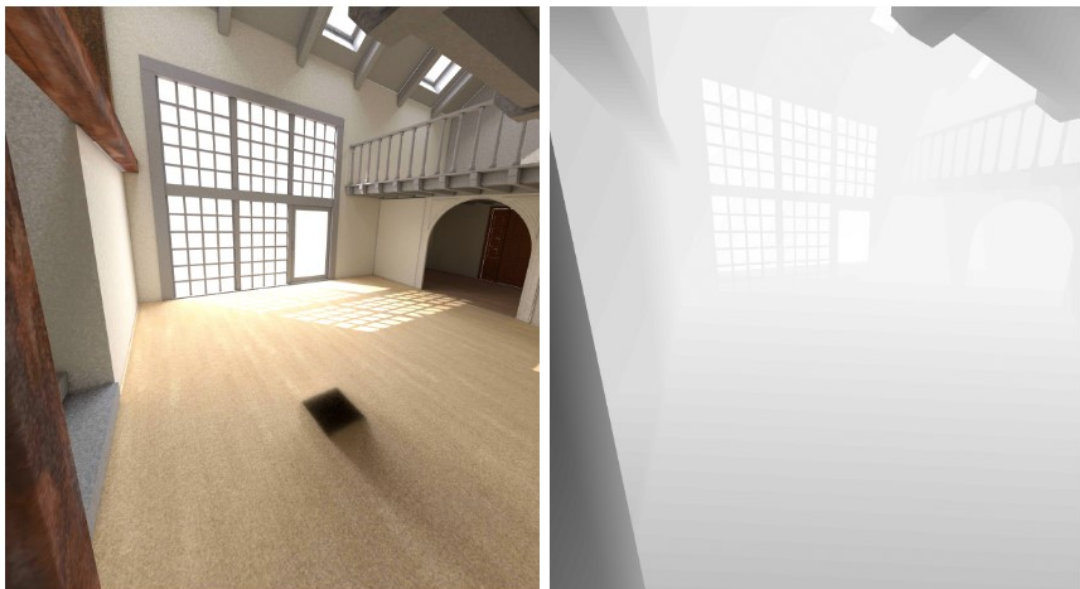


Figure 2.4. The color buffer, and therefore the final render, is shown on the left and the corresponding depth buffer which was used to find the depth value for each fragment on the right.

This Z-buffer algorithm is popular, because the order that objects are rendered does not matter. However the z-buffer can only store one single depth value for each pixel. This becomes problematic when rendering transparent objects, like windows. Transparent primitives have to be rendered after all opaque primitives have been rendered, in a separate render pass. This is why transparency is expensive to render and therefore is one of the major weaknesses of the z-buffer algorithm.

Traditional desktop PCs apply the entire render pipeline by processing triangles as soon as they are submitted. This is known as an Immediate-Mode Renderer ("IMR"). Using IMR, the graphics pipeline is run entirely from top to bottom for each rendered frame. This leads

to the graphics card accessing memory on a per-primitive basis, resulting in a large amount of data which has to be transmitted in the process of rendering. One solution created to reduce this high amount of bandwidth demand was Tile-Based Rendering. (Samsung 2020)

2.1.4 Tile Based Rendering

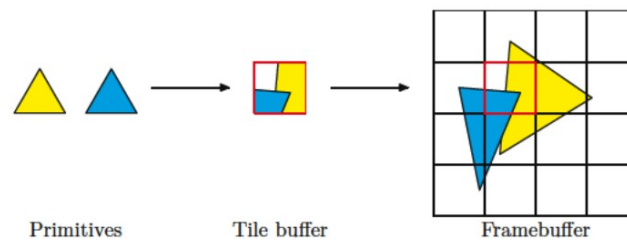
This section will examine what is referred to as “Tile-based rendering” or “Indirect Rendering Mode”. It is a particular way to arrange a graphics pipeline to make the rendering process more efficient for mobile GPUs. The majority of GPUs on mobile devices today use this tile-based approach instead of immediate mode rendering.

Traditional GPUs found in desktop PCs are usually designed as an “immediate mode” architecture, or IMR for short. They execute the vertex and fragment shader sequentially for each primitive in every draw call. The OpenGL virtual pipeline requires a lot of bandwidth and requires a lot of read and write operations per pixel. A typical use case can require a read from the depth/stencil buffer, a write back to it after processing and a write to the color buffer. According to Cozzi and Riccio (2012) this could easily lead to 100 bytes of data traffic for each pixel, when not considering texture compression. Therefore mobile devices need to be optimized for less bandwidth for power consumption reasons. This is where a Tile-Based architecture has its benefits. Is implemented in many mobile devices because it can reduce data traffic tremendously. This helps to minimize power consumption and reduces the cost of calculating transparency or anti-aliasing. On a basic level, tile based GPUs don’t store buffers like the depth buffer or the multisample buffer on the main memory as earlier hardware did. They write this data to a high-speed on-chip memory. This fast on-chip memory, called G-MEM (Graphics Memory) is specifically designed to store color values, stencils and Z-values. Due to the memory being close to where computations are carried out, data has a much shorter way to travel and can be written and accessed much faster. The internal memory is specially designed for high bandwidth and low latency. Because of shorter distances between the GPU and the storage unit, a lot less power is required to access it.(Sharp and Leger 2009, August 20; Cozzi and Riccio 2012)

However the size of the on-chip framebuffer has to be much smaller than previous framebuffers. According to Cozzi and Riccio (2012), it would just take too much silicon to implement a larger on-chip memory. This is why this on-chip framebuffer, also referred to as tile buffer, can only store a small amount of data. It can vary in size from 16x16 pixels up to 256x256 pixels per tile depending on hardware and pixel format. The way to generate a high resolution image using such a small buffer is to split up the screen space into smaller tiles, the size of the hardware memory. The GPU divides the output framebuffer into several smaller subregions also known as tiles. These tiles are where the name tile-based

rendering comes from.(Cozzi and Riccio 2012)

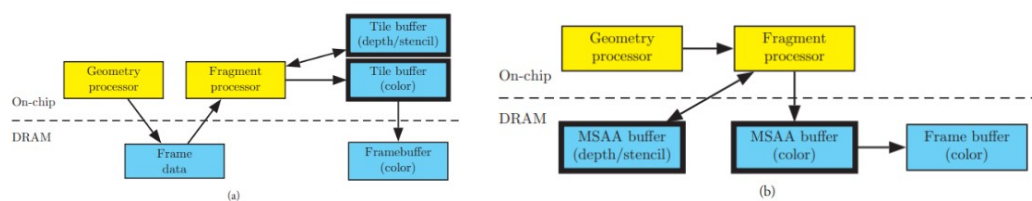
The entire scene will be rendered one tile at a time using a multi-pass technique. All primitives in the scene lying inside or touching the tile segment are rendered into the tile buffer and all primitives that lie entirely outside the tile window are discarded for rendering this tile. (Cozzi and Riccio 2012; Dasch 2019, November 19)



Source: Cozzi and Riccio 2012, S. 325

Figure 2.5. Primitives inside a tile are stored in memory (left), one tile is processed by rendering the intersecting primitives to the on-chip tile buffer (center) and is copied to the framebuffer in the last step (right)

Once the tile buffer is complete it is copied back into the general memory of the system, a process referred to as “resolve” or “heavy resolve”. This resolve is a very expensive task for the GPU as well as for the CPU, meaning it requires a lot of time to transfer data over the bus. The time it takes to transfer data is known as “resolve cost”. The bandwidth advantages of tile-based rendering are coming from only having to write a small amount of data back to the slow memory. Combining depth values, stencil values, overdrawn pixels and multisampling all happens on-chip. Tile-based rendering is therefore usually more efficient than rendering directly to slow memory.(Cozzi and Riccio 2012)



Source: Cozzi and Riccio 2012, S. 326

Figure 2.6. (a) shows the tile-based data flow. Yellow boxes indicate computational units, blue ones are a type of memory. The boxes with thick borders are multisampled buffers, so they have to be accessed multiple times. (b) shows the immediate-mode data flow where multisampled pixel data is transferred between the two buffers, taking up a lot of bandwidth.

The rendering process takes place in multiple phases to maximise efficiency:

- The first phase is the binning phase. It bins the geometry depending on their triangle

vertex positions into their corresponding tiles

- The second phase uses the binned data of one tile and rasterizes it, applies pixel shading and finally writes it back to memory (Cozzi and Riccio 2012; Sommefeldt 2015, April 2)

There are also some mobile GPUs using immediate mode rendering or which can switch between IMR and tile-based rendering. In 2013 Qualcomm introduced “FlexRender”, which lets Adreno GPUs switch dynamically between direct (immediate mode) and indirect (tile-based) rendering modes. It can analyze which rendering type works better for a given render target and automatically select a mode to render with. (Qualcomm 2015, May 1; Qualcomm 2018a, May 14)

Tile Based Rendering should not be confused with Tiled Shading. Tile Based Rendering is based on a hardware property a device offers. Tiled shading on the other hand is created on the software side and describes an algorithm which can be chosen to be integrated or not by the developer.

When profiling tools are not designed for the tile-based architecture, they might output incorrect data. The section “Profiling Tools” of this paper takes a deeper look at suitable tools to profile these devices and shows how to analyse the different render stages. (Cozzi and Riccio 2012; Arm 2020; Qualcomm 2020a; Samsung 2020)

The efficiency of Tile-Based Rendering comes from computing pixel values on the on-chip memory. Their hardware however has to be built for it. Even though it is most commonly seen in mobile hardware, desktop PCs are starting to use partially-tile-based rendering as well. The reason tile-based rendering was implemented into most mobile devices is energy consumption. The following paragraphs will deal with the differences of mobile and desktop hardware and show how they differ regarding power.

2.1.5 Mobile- and Desktop Hardware

Mobile and desktop GPUs are basically doing the same thing and have the same goal. They have to render images to a screen in a certain amount of time. One of the biggest differences is the available power these devices have available. When talking about mobile performance, energy consumption is a topic that can not be avoided. Most mobile devices are designed for a power-constrained environment and need to be power efficient due to their energy coming from a small battery. The two main factors when talking about energy consumption on mobile devices are battery life and thermal limitations. Due to the limited size of the device, mobile batteries have to be small. The components of modern high-end PCs on the other hand have a lot of power available and are housed in large enclosures. Mobile chipsets have to be a few millimeters thin. To provide a better user

experience the CPUs are built with more and more cores and higher processing frequencies to reach higher performance. This leads to much higher power consumption and heat generation. (Lei and Shengchao 2014)

Stationary PCs are often cooled by large metal heat sinks and fans. As the CPU and GPU heat up, it reaches the active threshold. If devices are actively cooled and therefore use a fan for cooling, the fan will turn on and cool down the processor. Devices with passive cooling systems don't use fans to cool down the components. If the temperature of a non-actively cooled component still rises, the system will automatically reduce the frequencies of the processors to decrease power usage and cool the device down again. If the temperature still does not decrease, damage might be caused to the components and the power should be removed from the critical components. (Lei and Shengchao 2014)

On mobile devices, cooling is usually done passively. However some systems, like the Oculus Quest, use active cooling where a fan is built into the HMD. This definitely helps the chip to stay at lower temperatures for a longer period of time. CPU cores can adjust their frequency depending on the task. If not all cores are needed, one single active core can run at the needed frequency while the others run at a low frequency or are turned off. The amount of energy a mobile device requires is usually related to the applications running on the device. It is therefore possible to influence energy consumption from the software side. Also the battery life of mobile devices depends on the content itself. Always using an application at the very limits of the devices capabilities will drain the battery fast and be frustrating for the user. It is important to think about how long users will stay inside an app and how much image quality a developer can sacrifice to still keep a good user experience. When developing an app it is a good idea to look at power consumption as part of the user experience. The hard part is finding the balance between performance and power consumption. Leaving a bit of headroom when designing apps is therefore recommended. (Kerin 2013, October 8; Garrard 2018; Jagneaux 2018, October 3)

One of the biggest power consumers is memory bandwidth. Computational tasks are cheap compared to the power it takes to move data. The further these informations have to be moved from one component to the other, the more power it will take, also leading to increased heat generation. When there are multiple read and write operations to different buffers, it leads to an excessive use of bandwidth and therefore power. This is why it is important to decrease the number of times, data has to be moved. The following section will show how data is usually transferred and how this process can be accelerated. (Cozzi and Riccio 2012)

2.2 Draw Calls and Batching

When processing a single frame, the hardware has to move a huge amount of data. All the textures, meshes and shaders can take up a lot of bandwidth. To display game objects on the screen, the game engine has to invoke the graphics API to render them. This is known as a draw call. These draw calls can be resource-intensive due to the graphics API having to process every single call separately. If not optimized in any way, each object in the scene would need a separate draw call. A large number of objects leads to a large number of draw calls which can lead to performance overhead on the CPU. Changes between draw calls such as switching to another material for a differently shaded object can lead to a performance decrease as well. Therefore the time a draw call takes to execute depends on the state of the previous draw call. If materials, meshes and textures changed from the last draw call, the following call will take longer as than if the materials stayed the same. (Doppioslash 2017; Unity-Technologies 2017a, October 26; Akenine-Möller et al. 2018; O. Developers 2020a; Murray 2020)

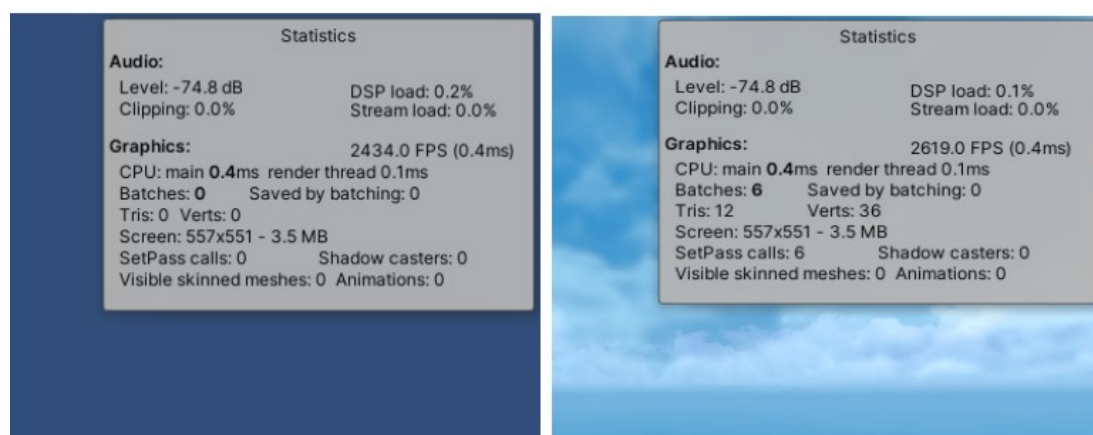


Figure 2.7. Using a Skybox in Unity increases the number of draw calls as well as batches.

The solution by the Unity Game Engine is batching. It combines multiple meshes together to draw them all in one go. Static batching combines multiple game objects which were marked as “static” in the Unity Editor to render them faster. Dynamic batching improves performance by taking small and similar meshes, grouping them together and drawing them at once. This already becomes important when thinking about which objects to combine in the process of modeling. If done manually, a good balance between combining barely all meshes and combining none has to be found. Unity’s built in batching algorithm has some advantages over merging meshes together manually e.g. meshes can still be culled individually. When for example two or more objects are combined manually, which are on opposite sides of a scene, both of them are going to be rendered even though only one of them can be visible at a time. The reason why it sometimes makes sense to keep ob-

jects separate which lie far from each other is frustum culling. Unity's integrated automatic batching algorithm might therefore be an advantage. However static and dynamic batching done in Unity can cause more memory, storage as well as CPU overhead.(Palandri 2018, September 26)

If meshes use different materials, they can't be batched. This is why it makes sense to only combine objects which are made of the same material. Using as few materials as possible can therefore lead to increased performance. Setting as many GameObject as possible to "static" and using as few different materials as possible helps Unity to batch those objects together. If two objects use the same material but a different texture, the textures can be combined to one single bigger texture to lower computing effort. Combining a number of textures to one is called "atlas" and is highly recommended by Unity and Oculus. Oculus recommends to target 50-100 draw calls per frame for the Oculus Quest. This should be taken as a guideline for establishing a baseline. Depending on the complexity of the scene, number of shaders, materials and textures these numbers will vary.(Doppioslash 2017; Unity-Technologies 2017a, October 26; Akenine-Möller et al. 2018; Oculus 2020)

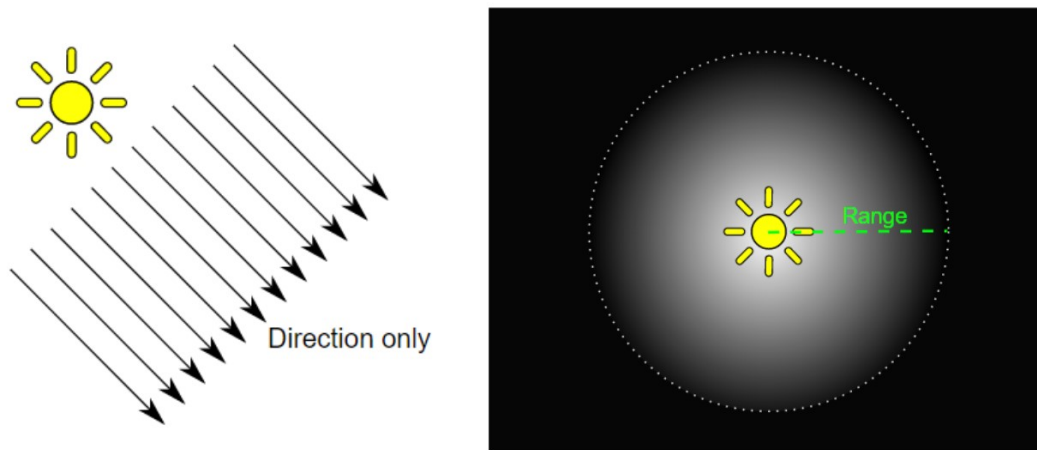
Most Unity shaders support multiple real-time lights. This leads to at least one more draw call per light. Draw calls for per-pixel lights can not be batched. Using real time lighting and shadows can therefore have a huge impact on render time. Especially using shadows can become very expensive to process.(Unity-Technologies 2017a, October 26)

2.3 Light and Shadow

One of the most important artistic and technical aspects for architecture visualizations is light and shadow. Shadows are necessary for realistic looking images and give the observer cues about where objects are in a scene. Especially in architecture, the use of light and shadow becomes a major aspect of design and impacts the mood of an environment significantly.

A directional light is one of the simplest forms of a light source. It is effectively infinitely far away and has a constant light intensity over the entire scene. All light rays created by the light source are therefore parallel. A light like this does not exist in the real world due to it having no specific location. These types of lights are abstractions that work fine as long as the distance from the light source to the scene is large enough. The most common use for directional lights is to simulate the sun. This light is therefore important when trying to simulate sunny indoor or outdoor environments. Other than directional lights, point (omnidirectional) lights have a distinct position in world space and can be imagined as small lights that uniformly send out light rays into all directions. The light intensity usually falls off with the square of the distance from the origin. If it does not fade off completely,

the distance the light can reach can also be clamped at a predefined distance. The render engine applies the effect of the light on all objects within the sphere of influence, which can make it quite cheap to compute. Point lights are typically used as artificial light sources such as lamps on a ceiling or wall. These lights are also referred to as dynamic lights because they can be rendered in real time.(DoppioSlash 2017)



Source: Unity-Technologies 2020f, August 25

Figure 2.8. Directional light with parallel light rays on the left and a point light with a specified distance on the right.

Shadow Maps

There are many different approaches on how to render shadows on any kind of object with low computational cost. One of those was a z-buffer-based approach called “Shadow mapping”. The scene is rendered from the position of the light source using the z-buffer. All polygons the light “sees” receive light and everything else is not illuminated and is therefore shadow. This map, with information about how far rays travel before they hit a surface, is later sampled, to find which parts of the scene are in shadow, when rendering the final frame. If a light source is surrounded by shadow casters, like a light bulb in a room, the shadow map can be created as a six-sided cube, similar to an environment map. These are called omnidirectional shadow maps. For distant light sources like the sun, the view range is set to include all shadow casting objects in the scene, which cast shadows into the viewing volume of the camera. Sunlight is simulated using an orthographic projection and the view must be wide enough to view all objects relevant to compute shadow maps. How detailed the shadows are is therefore dependent on what resolution the shadow map is, which is one of the biggest disadvantages of shadow maps. In Unitys URP and HDRP the shadow map resolution can be chosen for the main light and for all additional lights in the scene independently.(Akenine-Möller et al. 2018; Unity-Technologies 2020f, August 25)

For every real-time light which casts shadows into the scene, all meshes that cast or receive shadows require one additional draw call for shadow calculation. The cost of creating a shadow map is therefore quite predictable. The big advantage is that this process only

2 Theoretical Foundations

requires z-buffering. Lighting, texturing and writing to the color buffer is not needed, meaning the fragment shader does not need to run, like typical shaded draw calls do. However they still produce overhead on the CPU as well as the GPU side. The CPU has a draw call increase and the GPU has to project geometry correctly using the vertex shader to create the depth shadow map. How significant this overhead is and how render times are influenced by it will be tested in later sections.

The temporal depth images to calculate the shadows can be inspected using a frame debugging tool like RenderDoc. It shows the scene rendered from the perspective of the directional light. As with standard opaque objects, there might be several draw calls necessary for the temporal buffer to be finished, depending on how the scene is built. The draw calls following the shading calls are then sampling the scene's shadows.

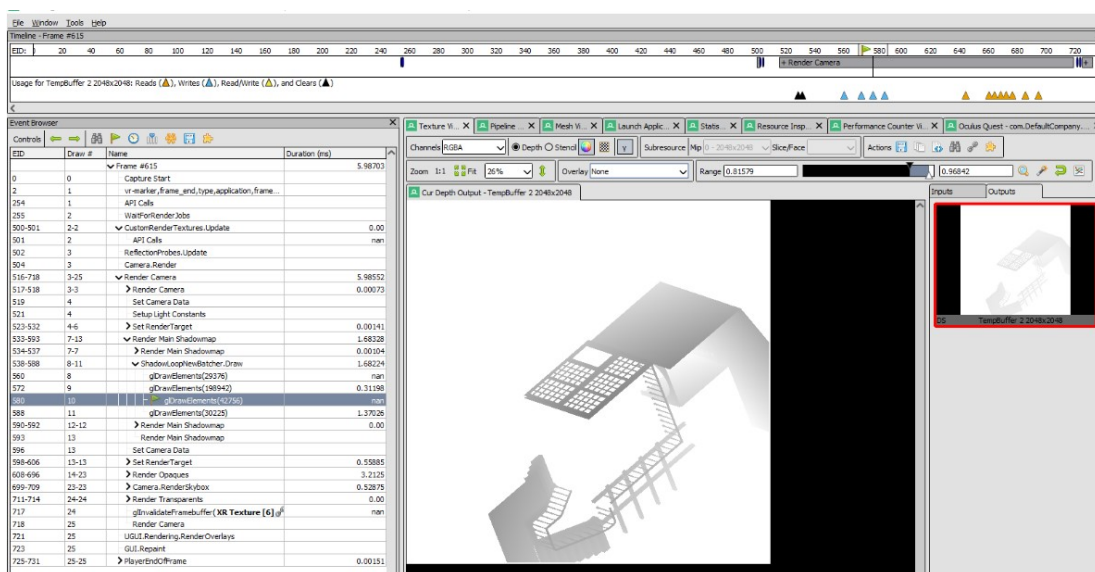


Figure 2.9. RenderDoc can be used to inspect the z-buffer for any light casting a shadow.

We can see that the shadow map is calculated before the room is drawn. By expanding the “ShadowLoopNewBatcher.Draw” element in RenderDocs Event Browser, we can see step by step in which sequence the shadow map is generated.

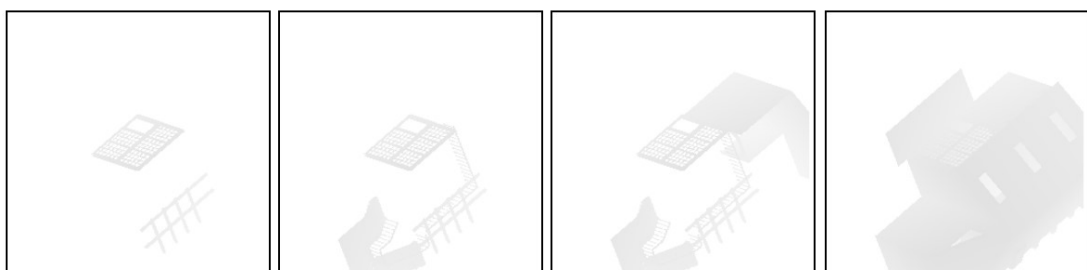


Figure 2.10

When planes are modeled as one sided, the objects will not cast any shadows in Unity

because their back faces are culled. Therefore both faces of the plane need to be rendered when using real time light and shadow.

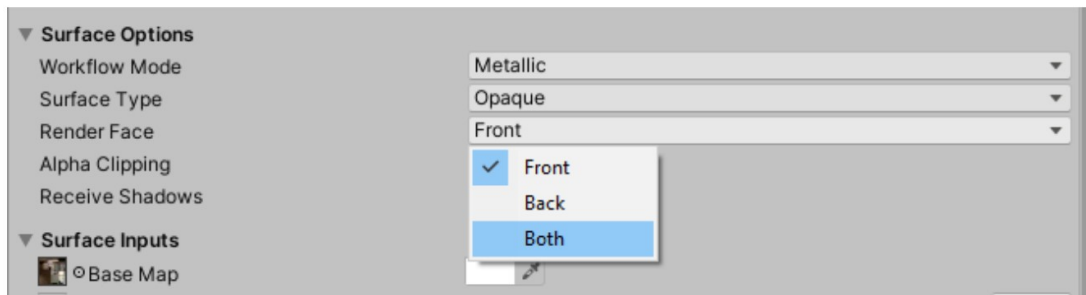


Figure 2.11. In Unitys Surface Options one can choose which sides of an object should be rendered.

Area or volume lights produce shadows that appear soft with fuzzy shadow edges. However these soft shadows can not be created by simply blurring the edges with a low-pass filter. The edges have to become harder the closer they are to the shadow-casting object. Real-time area lights have been a focus point and a goal to achieve for game developers. In recent years, some approaches were found and even implemented into some game engines. At the time of writing, there are not any widespread solutions for mobile hardware. Unitys High Definition Render Pipeline (HDRP) which targets high-end hardware for example, renders area lights in real-time but still can not compute shadows in a correct way. For mobile applications it is necessary to bake area lights and resulting shadow information into the textures. Hard-edged shadows on the other hand result from small, distant light sources and can be rendered in real time.(Unity-Technologies 2020b, February 3)

Texture Baking

When rendering non moving light sources in a scene, there are two ways to render them. One is the same way dynamic lights are rendered, that is, going through the entire shading pipeline and calculating its effect every frame, meaning it is rendered on-the-fly while the application is used. This can of course become very expensive to render. When baking textures, the effect of a light is added to the texture of the object itself. This happens offline in the development stage of the application and can therefore take a much longer time. The lighting information reaching the object is therefore “baked” into the texture of an object and becomes part of its texture color. After texture baking, all lights can be removed from the scene and the baked texture is applied to the object as a self illuminated material. The advantage of baked textures is that they are calculated much faster because the effect a light has on the object does not have to be processed every frame. A disadvantage is that this trick becomes obvious as soon as any of the objects or lights in the scene are moved. Further specular lighting can not be utilized.(Chaosgroup 2019; Unity-Technologies 2020d, August 25)

Texture baking does not necessarily have to be done in the game engine. It can happen in an earlier stage in the modeling software. Most 3D graphics modeling software ship with a native product-quality rendering engine like Arnold in Autodesk Maya or Mantra in the 3D animation software SideFX Houdini. Further, most of these applications can add external renderer software such as V-Ray as plug-ins. These render applications have the advantage of being very powerful, as they deliver fast and accurate results, compared to simpler render engines like Unity's integrated Progressive Light Mapper. The architecture visualization created for the later experiments was rendered using V-Ray for reasons of speed and quality. Rendering it with Unity's integrated texture baking system instead, would lead to less realistic looking textures. Further it would be more time consuming to achieve the same results with Unity's Progressive Light Mapper as with external render engines. (Unity-Technologies 2018e; Chaosgroup 2019; SideFX 2020)

When baked textures are used, they need to be self illuminated. The URP includes a material specifically for this purpose called the "Unlit Shader". Unlit shaders are used when an object does not need lighting. Because time consuming operations like calculating how light falls onto an object are skipped, the Unlit shader is perfect for low-end hardware like mobile devices. (Unity-Technologies 2020h)

It is required to have clean and unique UV mapping when baking light. There will be problems if light maps overlap or are not scaled in a correct way. If there are problems regarding the UVs, they probably have to be fixed in the modeling application.

2.4 Aliasing

When not using anti-aliasing, objects which don't perfectly line up with the pixel grid of the screen such as triangles, or circles, will create noticeable artifacts. Objects will show up in pixels as being there or not being there. There is no in between. This jagged look is why this artifact is often known as the "jaggies". Formally the problem is referred to as "aliasing" and the solution to it is "anti-aliasing". Aliasing becomes visible when dealing with low resolution displays. When talking about VR headsets this becomes even more important because the viewer is unusually close to the display. The pixel density therefore would have to be even higher for aliasing effects not to appear.

There are many different anti-aliasing algorithms. One that works well on mobile devices and is therefore recommended to use is multisample anti-aliasing (MSAA). Adreno GPUs by Qualcomm for example are specifically designed for a fast implementation of MSAA. Using MSAA on PC hardware usually comes with a lower cost than when using it on mobile devices. According to Oculus, depending on how much content needs to be rendered per frame and the GPU level the app runs on, using MSAA will increase the render time by



Source: Akenine-Möller et al. 2018, S. 131

Figure 2.12. Three different levels of anti-aliasing. Left: One sample per pixel, middle: four samples per pixel, right: eight samples per pixel. (in a grid pattern)

0.5 to 1.5 ms. When aiming 72 fps, meaning 13.8ms render time per frame, MSAA could take up more than 10% of a frame's render time. In Unity no anti-aliasing, 2x, 4x or 8x anti-aliasing can be chosen. Oculus recommends using MSAA for apps but not choosing a higher sampling than 4x. 8x MSAA might lead to a slightly better image quality and less aliasing effects but due to the high performance costs it is not recommended on mobile devices.(Qualcomm 2019; O. Developers 2020d)

Adreno GPUs use multisampling to reduce aliasing effects. It divides every pixel into a number of samples which are handled like individual pixels themselves. Each one has a color, depth and stencil value. When the final image is created, the samples are resolved to one final pixel color.(Qualcomm 2019)

2.5 Stereo Rendering

Other than non-VR mobile applications, virtual reality does not just have to render one image to the screen, but two. Depth perception is an important aspect of virtual reality applications. For each eye, a different viewing angle has to be rendered to create the illusion of a three dimensional space. Rendering the entire scene twice with a different view is not the most efficient way to generate those images. This would lead to a significantly higher rendering effort and therefore less time to render the applications actual content.

Since Version 5.6 Unity provides two rendering modes for VR applications. When using stereo rendering, two images need to be rendered which is done by rendering to each of the eye buffers sequentially. Therefore the application and driver overhead is usually doubled compared to non-VR applications. This approach which needs one draw call per eye is called Multi-pass rendering. The solution to this problem is Single Pass Stereo rendering. It saves render time by rendering the same scene from different viewpoints with a single draw call instead of two. It is accomplished by allowing this one draw call to render to a number of texture layers of an array texture at once. The vertex and fragment shaders then run for each of the texture layers in the array texture. Vertex positions and other variables,

which are dependent on the viewing position, such as reflections, are therefore rendered separately as before. This primarily decreases CPU usage, because the draw call count is lowered, while GPU performance is not affected as much.

Multi-pass Rendering

This is the traditional way to render a scene by rendering the scene to two different images, which are separately displayed to each eye. Stereo rendering therefore requires two passes.

Single-Pass Stereo Rendering (also known as Multiview rendering in OpenGL/Vulkan)

For this mode to run, special GPU hardware is needed. It renders the scene with just one geometry pass and extract the different informations for each eye. With this rendering mode, the vertex shader is only executed once.

As a developer, switching from Multi-pass rendering to single pass stereo rendering is one of the easiest ways to reduce draw calls. (Eskofier 2017, July 10; Lopez Mendez 2017, May 1; Unity-Technologies 2017b, June 20; ARM 2018, October 19; O. Developers 2019b; O. Developers 2019c)

RenderDoc can be used to verify if an image is rendered using Multi-pass or Single-pass rendering mode. When taking a closer look we can see that two “Render Opaques” events both take about 4ms each to render. Looking at the render textures we can see that each of them is rendering the scene once for each eye.

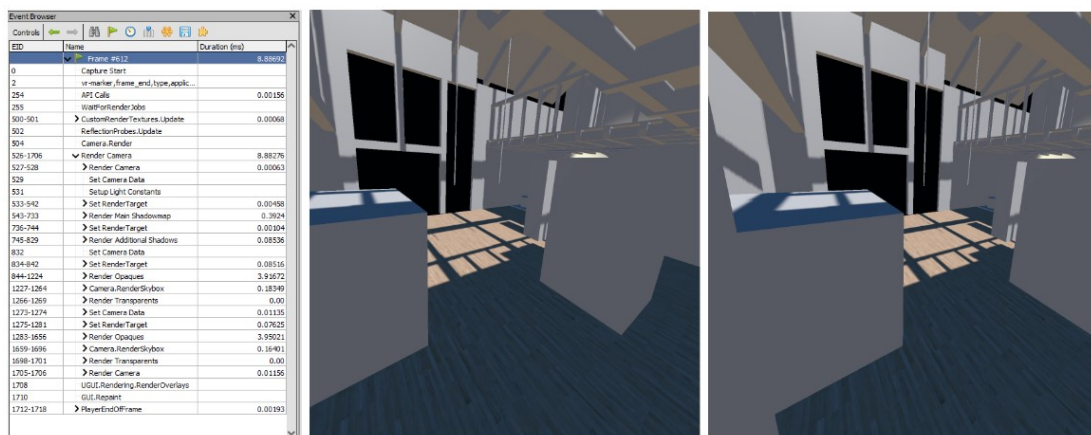
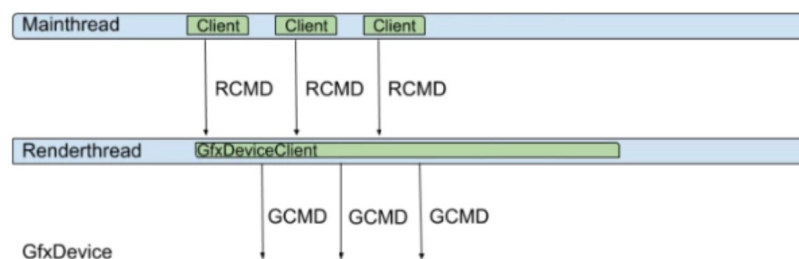


Figure 2.13. A scene rendered in multi-pass mode needs one draw call for each image rendered.

2.6 Multithreaded Rendering

Depending on what a platform supports, there is single threaded and multithreaded rendering. With singlethreaded rendering, the “single client” needs to execute on this one available thread. This single client has the job to execute all rendering commands and performs the rendering process through the underlying graphics API. Because it is all done on the single main thread, it takes away important frame time which could be used for any other commands running on the main thread. Multithreaded rendering is automatically enabled if the device supports it. With this enabled, the single client running on the main thread forwards all rendering commands to a second thread called the “renderthread”. This can greatly benefit performance.



Source: Unity-Technologies 2019, November 26

Figure 2.14. The main thread forwards all rendering commands (RCMD) to the renderthread which is used for rendering tasks only.

Generally, multithreaded rendering should be enabled whenever possible. According to Unity, when profiling with multithreaded rendering active, incorrect results may occur. In general, it is recommended to deactivate multithreaded rendering when profiling, and activate it in the final build. Because some very low-end devices will not benefit from it, it is also recommended to profile the use of single- and multithreaded rendering to see how much of a difference it makes.(Unity-Technologies 2019, November 26)

2.7 Profiling

2.7.1 Hardware

Before starting to develop any application, taking a closer look at the platform the application will be designed for, is generally a good idea. If an application is designed for one specific device, the app should always be tested on the same devices, the end users will use. Finding out if a device has an active or passive cooling system can give indications on potentially occurring heat problems. Hardware vendors usually offer documents which

sum up the most important specifications.(Qualcomm 2017; G. Developers 2020, May 18)

The Adreno GPU is part of Qualcomm's SOC (System-on-a-Chip) hardware known as Snapdragon. As with any GPU, it supports the CPU with rendering tasks to meet the level of performance for mobile games as well as doing simple tasks such as drawing the user interface. The Adreno GPU series consists of devices purposely built for mobile application. They were designed to communicate with mobile APIs with usual constraints like the limited amount of power in mind. The Adreno GPUs use a tile-based rendering approach. The Oculus Quest uses a Qualcomm Adreno 540 GPU which supports OpenGL ES up till Version 3.2. The CPU is a Qualcomm Kryo 280 CPU with eight cores and a maximum clock speed of up to 2.45 GHz.(Qualcomm 2015, May 1; Qualcomm 2017; Qualcomm 2018b, October 2)

Further it can be helpful to have benchmark data or information about device limitations to get a better understanding of what a device is capable of. Oculus for example provides baseline targets for VR applications and some information on what tools to use to find bugs. (Oculus 2020)

The most accurate way to find out what a mobile device is capable of is testing it by yourself. This is why a list of profiling tools will be presented in this paper in section subsection 2.7.4. At first however it is necessary to understand how performance can be measured.

2.7.2 Measure Performance

There are several ways to measure the performance of an application. When talking about performance, people often say a game runs at a specific number of FPS (frames per second). The frame rate describes the number of images rendered per second and is a common measure in games. Even though frames per second might be the ultimate goal of performance optimization, it is easier to measure it in milliseconds (ms). When profiling, the time for each operation is measured in milliseconds, to get an idea of which processes need how long to compute. This is referred to as the budget. The maximum time in milliseconds one frame takes to compute can be calculated as following.

$$\text{frame time in ms} = \frac{1000}{\text{target frames per second}}$$

Most modern games try to achieve a framerate of 60 FPS. Therefore to achieve 60 FPS, the Budget has to be 16ms. If the target frame rate is 30 FPS, the maximum time to calculate one frame is 32ms. With Virtual Reality latency can create very uncomfortable effects. Simulator sickness, as it is called, appears when the displayed image does not match the users expectations or perceptions through other senses. Therefore, the lower

the lag between the actual movement of the players head and the matching display image the better. 90 FPS is a common frame rate for tethered VR devices. Apps for the Oculus Quest HMD should target a frame rate of 72 FPS, which is 13.88 ms for the entire render process for both eyes. This is the targeted render time for the architectural scene created for this paper. (Turner and Schell 2017, July 10; Akenine-Möller et al. 2018)

2.7.3 CPU/GPU Bound

If an application does not achieve the required performance target, there are multiple reasons for it. If an application is bound, it does not reach the aimed performance due to high processing demands. One can distinguish between a CPU bound and a GPU bound application.

To improve application performance, it is important to have a basic understanding of the relationship between the GPU and the CPU. As described in the graphics pipeline section, there is a certain amount of work the CPU has to do before passing data on to the GPU. Rendering code produces all the necessary data the GPU needs, to render the image, and is therefore executed at the end of the CPU rendering process. The CPU hands off the data to the GPU and as soon as the GPU receives the data, it can start rendering the frame. After the CPU has handed over the data, it can start to process the next frame. GPU and CPU are therefore working in parallel. If the CPU takes longer to execute all of its code, the GPU has to wait for it to finish. This leads to an increased render time due to the GPU not having the necessary data to fulfill its task. This is what is called CPU-bound, meaning that the GPU was waiting for the CPU.

If the CPU waits for the GPU to finish, even though the CPU is already done with its work, it is called GPU-bound. To start rendering a frame, the GPU has to finish rendering the previous frame first. The most common reason to be GPU-bound is that the GPU is taking too long to render the previous frame. (ARM 2020)

Architecture visualizations usually have the benefit to barely include any demanding code, complex physics simulations or animations. These are mostly computed on the CPU side. How much time the CPU spends on these tasks depends on how much interaction the developer wants. If complex code or a number of physics simulations need to be computed, it is likely for the application to become CPU-bound. Otherwise being CPU-bound usually will not occur. Most architectural visualizations will spend the majority of time rendering graphics and will therefore have a higher chance to become GPU-bound.

Figure 2.15 shows the profiling data from the architecture model designed for this paper. It views the timeline of the Unity Profiler which tells us how much time each task needs for computation. For this specific frame, the CPU took 14.15ms to compute its tasks. The

GPU only took 7.02ms. This tells us it's the CPU which wasn't fast enough and therefore the scene is CPU-bound.(Unity-Technologies 2020c, August 25)

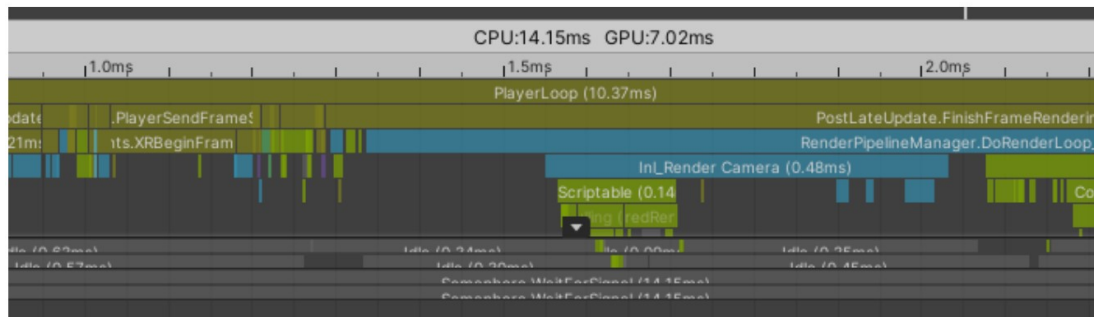


Figure 2.15. The Unity Profiler Window shows the GPU and CPU render times at the top of the window.

Finding out if an application is CPU or GPU bound is the first step when encountering problems. Unity's recommended approach is to use the Unity Profiler to do so. In some cases the task "Gfx.WaitForPresent" will show up in the profiler window. If this task takes up a majority of the frame render time, it is an indication that the CPU is waiting for the GPU to finish rendering. Therefore the application is GPU bound.

As mentioned in the section 2.6, single threaded rendering should be utilized when profiling with the Unity Profiler to have an accurate representation of how long certain tasks need to execute.

2.7.4 Profiling Tools

Profiling tools give information about how an application is performing and can help to find bottlenecks. They help to identify the cause of problems like low frame rate or high memory usage and help to see how design and programming choices affect performance and power consumption. Which characteristics are valuable for profiling is different to every application. It is important to know that with any kind of profiler, there is a certain amount of overhead. Profiling always takes a small amount of performance cost. However it is unlikely for the profiling tool to lower the overall application performance significantly, because the effects on performance are usually minor. Oculus recommends to generally use relative numbers for value comparison when profiling, rather than putting too much attention to absolute numbers.(Qualcomm 2013; O. Developers 2019a, August 6; Unity-Technologies 2020a, April 27)

Some profilers don't work on every application without modifying it and need the application to provide access to certain data. For example, to profile an app using the Unity Profiler, the app must be marked as a "Development Build" in the Build Settings before creating the

build. When disabling the option, the app will run faster, but will prevent the developer from using the Profiler API methods. The option should therefore only be used when profiling and should be deactivated for the final build.(Unity-Technologies 2020g, August 25)

A number of Profiling Tools will be inspected in the following. Each of them will be described shortly. Some of them will be used for application analysis later. These are the ones which will be covered more in depth. The tools that are covered are Unity Profiler, Unity Profile Analyzer, Unity Frame Debugger, Snapdragon Profiler, OVR Metrics Tool, RenderDoc and Systrace.

The Unity Profiler

When developing in Unity, the Unity Profiler is the first tool to go to. It is Unitys integrated profiling tool built right into the Unity editor. Profiling can be done in two ways: The application can be run in the Unity Editor itself or on a different device. When profiling directly inside the Unity Editor, the application uses the hardware components, the Unity Editor runs on, which would only make sense when developing for a tethered VR-device. To profile applications aimed to run on mobile devices, it is always recommended to profile directly on the target device. Therefore the project has to be built onto the target device as an application. The device can be connected via wireless network or directly by connecting it to the PC.(Unity-Technologies 2020a, April 27)

The Profiler can gather performance data in areas such as CPU, GPU, memory and audio. It is useful to find specific areas of a game which don't run as expected. It is possible to track down problems and break them down to find out where it might originate. One can clearly see which script, asset, camera affects the applications performance.(Unity-Technologies 2018d)

An advantage of the Unity Profiler is that it can be controlled via script. The profiler class can be useful when trying to profile just a specific section of an application. With "Profiler.BeginSample" and "Profiler.EndSample", only the code lying in between those code lines will be captured. Also on standalone devices the profiling data could be saved as a binary file right on the device for later inspection. Starting and stopping the profiler via script will be used in the testing section of the paper. By default, Unitys Profiler only records the last 300 frames and dumps all information recorded previously. The number of frames to capture can be increased but this also increases the Profilers overhead and therefore memory usage can become more performance intensive.(Unity-Technologies 2018b; Unity-Technologies 2020g, August 25)

The data gathered in the Unity Profiler is visualized as a chart for inspection. To further analyse or even compare two data sets, the profiling data can be transferred to Unity's

Profile Analyzer.

Profile Analyzer

With Unitys Profile Analyzer two data sets of recorded profiling data can be compared and analyzed. The tool can show minimum, maximum, mean and median values over a selected range of frames. It is especially useful when analyzing not just a single frame but a longer period of time. This can be done for multiple sequences at once, to see how they differ from each other. It is useful to show differences in performance after making a change in an application or to debug a scene. These changes could be in the code, the assets, project settings, Unity version or platform. A big advantage of the profile analyzer is the possibility to calculate the mean value of multiple parameters over a time span.(Homewood 2019, October 9)

When trying to reach a specific frame rate which should be constant over the entire time the application is used, the Profile Analyzer is a great tool to do so. This tool will be used in later chapters to analyze the profiled data sets.(Homewood 2019, October 9)

Snapdragon Profiler

Because the tests for this paper are performed using the Oculus Quest as mobile VR head-set it makes sense to try profiling tools which were specifically designed for the platform. Oculus Quest runs on a Qualcomm Snapdragon 835 System which is manufactured by Qualcomm Technologies. For any device running on a Snapdragon chip, Qualcomm offers a profiling tool specifically designed for them called the Snapdragon Profiler. It helps to analyze CPU, GPU, memory, thermal information and more. Its biggest advantage compared to other profiling tools is that it has access to a lot of hardware information. Hardware states like power consumption, temperature and information about CPU as well as GPU frequencies can be read from the device and shown as a graph in real time.(Qualcomm 2020b)

Snapdragon Profiler has the ability to take a so called Snapshot Capture, which captures a single frame for analysis. However compared to RenderDoc, Snapshot Capture does not offer a lot more information than RenderDoc does. RenderDoc can give a far deeper insight into individual steps of the render pipeline. One important thing to know when trying to use the Snapshot Capture mode is to verify that the Android application has the permission to write to the internal storage of the VR device. Otherwise the snapshot capture will not work and will not show any error message. This can be changed in Unitys Player Settings under "Write Permission". As soon as the app is opened the first time on the device, access to data has to be allowed by the user.

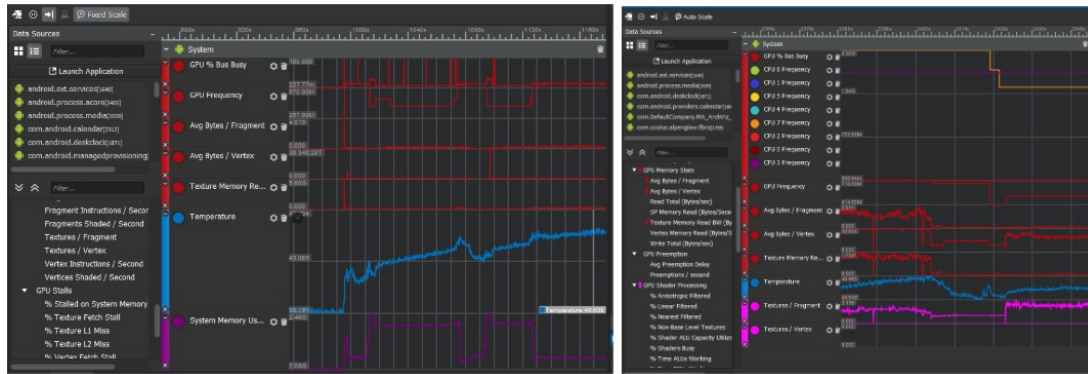


Figure 2.16. Left: The temperature of the Oculus Quest rises after the architectural scene runs. Right: Snapdragon Profiler providing information about data such as Bytes/fragment and processor-frequency.

If any component of a mobile device, like the CPU or the GPU, has too high of an operating temperature over a longer period of time, it will lead to a frequency collapse. The system scheduler and the power monitors will tell the components to lower their frequency for the device to cool down. This leads to lower performance and therefore the app might not be able to render frames in time. Monitoring the temperature of a device can be useful for several reasons. For this paper, the temperature data was used to measure how high the device's temperature is before starting a test run.(Schwartz 2016, December 15)

Unity Frame Debugger

The Frame Debugger is included into Unity and lets the developer freeze the application on one particular frame for analysis. The tool works similar to the Snapdragon Profiler and RenderDoc. It shows the individual draw calls, necessary to draw the frame. It can provide information about what the GPU is doing and provides a lot of information about how Unity prepares the data to be rendered. It shows all the necessary stages to show the final image on screen. An advantage of the Unity Frame Debugger is the ease of use when developing an application in Unity. It does not take much time to open the tool window and start profiling. The disadvantage compared to other profiling tools is the lack of data it provides. RenderDoc for example gives a way deeper look into the render pipeline and the content of frame buffers. That is why RenderDoc was chosen for most profiling tasks for this thesis.(Unity-Technologies 2018a)

OVR Metrics Tool

This is a tool by Oculus which is used to display performance metrics right in the view of the HMD. It is basically a head-up overlay in front of the application that provides informa-

tion in real time. The tool can show data like frame rate, heat as well as GPU and CPU information. For example, one can use the time an application needs to render a single frame to find out if an application is CPU or GPU bound. A second mode of the Metrics Tool called “Report Mode” saves the data of a VR session to later export it as a CSV with PNG images. A big advantage is that this tool can be used with every app, regardless of how it was built. Because the tool gathers data from the device itself rather than from the application, it is not necessary to make any changes to the application before using it. This can be very helpful to analyse any kind of VR application on the device or to see how efficient other applications are. (Trevor 2019, October 25; O. Developers 2020c)

RenderDoc

RenderDoc is a graphics debugger tool, similar to Unity’s Frame Debugger. It captures single frames and a huge amount of data like all graphics API requests and data stored in buffers at the time of the capture. All events that happened to render a single frame are sorted in the order they were issued, which is great to analyse how the frame was created on the device. This gives a look behind the applications rendering scheme for developers to verify if it is rendered the way it is supposed to do or to find bottlenecks along the graphics pipeline. By seeing exactly how many draw call requests were sent from the CPU side, the developer can verify if there are any unnecessary draw calls or something does not work as expected. The tool is completely free and has an open source license. Further the depth buffer can be inspected, which usually is not available anywhere. It shows if MipMaps are created as expected and as well as cascaded shadow maps, if activated. With RenderDoc the developer can verify if frames are created in Single-pass or Multi-pass rendering mode. Further it can show the different rendering stages of the render pipeline and which one are skipped. Another advantage of RenderDoc is that correct object culling can be verified easily. (Ferreira 2019, December 5)

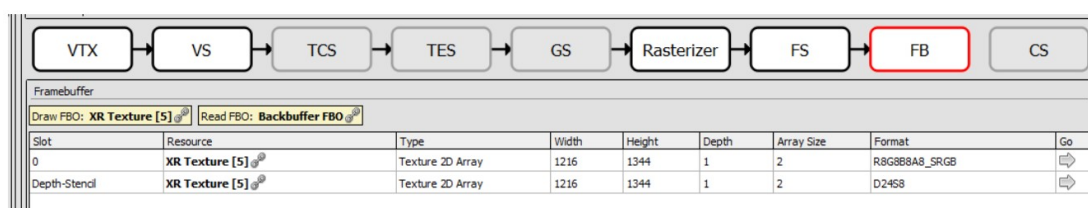


Figure 2.17. Inspecting the Framebuffer of an Oculus Quest application using RenderDoc.

When gathering time durations for each draw call, RenderDoc runs a process to find an estimated time for each draw call. This can give a rough estimation of how long certain draw calls take to execute on the graphics unit. Due to tile based rendering needing multiple passes to render a scene instead of a single one, the timing for draw calls can not be calculated accurately. When the tool was tested on an Oculus Quest scene, the frame

render times varied between +0.5ms up to +-5ms. The relation between the individual draw call times however can be accurate enough to see which draw calls might be responsible for poor performance. So draw call times should always be compared to other event times which cover a similar amount of pixels rather than relying on absolute time values. (Ferreira 2019, December 5)

Different from other debugging tools is that it is not necessary to package apps specifically for RenderDoc to be able to analyse them. Any captured frames can be saved on the computer RenderDoc is installed to view them at a later time. However, to reload them, the target device must be connected to the PC because RenderDoc usually reads data directly from the buffers inside the target device. (Palandri 2018, September 26; Karlsson 2020)

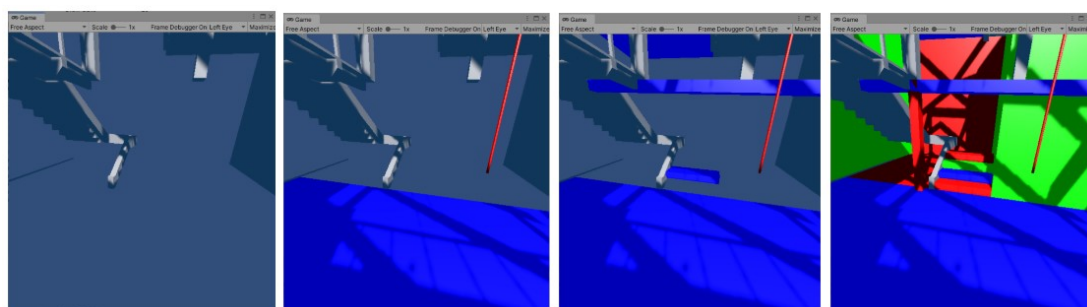


Figure 2.18. RenderDoc visualizes in what order objects are drawn to the screen. On tile-based devices however this is incorrect because this drawing process happens for each tile individually instead of once for the entire frame.

Systrace

Systrace is a profiling tool which comes with the Android Developer Tools. It can record detailed logging information about any android device. Systrace can either be run from a console window or from the Android Debug Monitor which is included in the Android SDK. For this paper, both variants were tested.

Most GPU profiling tools, such as RenderDoc have the problem of delivering incorrect draw call timings due to the nature of tile-based rendering. With Systrace, the tile-based approach can be visualized in detail. It can help to get a deeper understanding of how a frame is generated using the tile-based architecture.

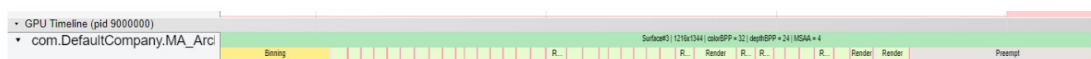


Figure 2.19. Systrace providing information about how a frame is rendered using tile-based rendering.

The first yellow processing block in Figure 2.19 is called “Binning” is the first of two stages.

2 Theoretical Foundations

In this stage, the triangle vertex positions for all draw calls are calculated. They are assigned to the corresponding partition of the drawing surface. This is done by executing simplified versions of the vertex shaders.

The following render steps are marked in light green and red. One chunk of the light green block, denoted as “Render”, describes the time it takes to render all vertices and fragment operations for one bin/tile. In this stage the full version of the vertex shaders are re-executed.

After the “Render” phase, a shorter process called “Store Color” is executed. Indicated by the color red, it shows the time it took for the previously calculated color values are copied from the fast on-chip memory to the slower general memory.(O. Developers 2020e; Lee and Palandri 2020, June 19)

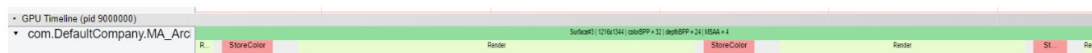


Figure 2.20. Using Systrace to take a closer look at how individual tiles are rendered.

Selecting one of those processes, gives further information on what is happening at this stage. As it can be seen in Figure 2.21, selecting Surface#3 gives information about the frame which is composed. For example, it shows how many separate bins the rendered image was split into and what dimensions the bins have.(Lee and Palandri 2020, June 19)

1 item selected. Slice (1)	
Title	Surface#3 1216x1344 colorBPP = 32 depthBPP = 24 MSAA = 4
User Friendly Category	other
Start	3,357,098 ms
Wall Duration	6,204 ms
Self Time	0,318 ms
Args	
surfaceId	"3"
width	"1216"
height	"1344"
colorBPP	"32"
depthBPP	"24"
stencilBPP	"8"
MSAA	"4"
MRT	"1"
numberOfBins	"60"
binWidth	"128"
binHeight	"224"
renderMode	"HwVizBinning"
colorAttributes	"0x3"
depthAttributes	"0x3"
stencilAttributes	"0x8"
newFrameSurface	"0"
contextId	"0x92faad50"
processName	"com.DefaultCompany.MA_ArchViz_Shadow_2048"
startTime	"1596712197972634"
endTime	"1596712197978838"
numSurfaceStages	"124"

Figure 2.21. Systrace can provide detailed information about the tile-based render process.

3 Method

3.1 Testing Methodology

On most mobile devices, the cost of rendering time can be measured by profiling a sequence of a few seconds while the application is run. Many profilers can be started and stopped via script, as described in the previous “Profiling Tools” chapter. For anything more granular than the render time per frame, the profiling information might not be useful on some profiling tools. This is because on tile-based architecture, commands are not processed in the order they are submitted by the application. Vendor-specific tools might be needed to get a deeper insight on what is going on at a hardware level.

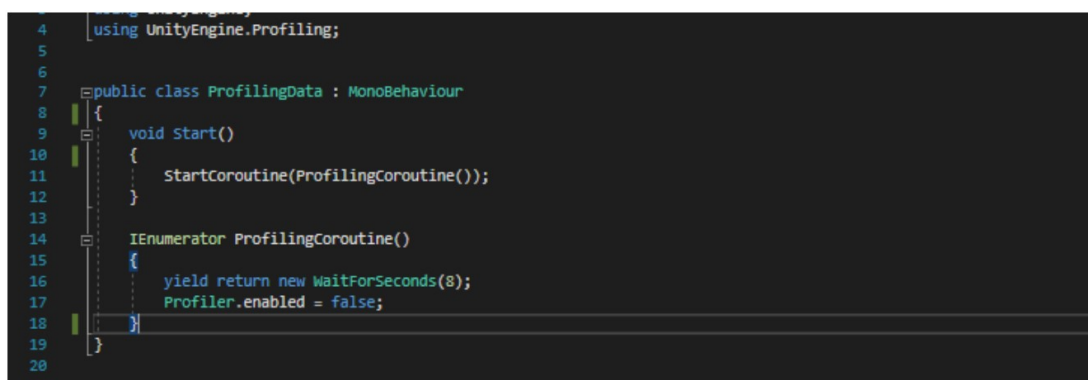
In the following sections, two methods for analyzing data were utilized.

The first approach was to record and replay a sequence of a pre recorded game play. Therefore a few seconds of a player moving through the virtual environment were recorded using a script. For each moving object in the scene, consisting of the virtual cameras for the eyes, the hand controllers and the entire camera rig, the position and rotation data was captured every frame and saved to an asset file. The captured sequence also includes the controllers position and movement as it was recorded. The sequence consists of natural player movements like inspecting the room from the ground to the roof by looking around the scene and teleporting around. The only thing that differs from a real walk-through is the absence of the teleportation visualization due to it being only visible when the corresponding buttons on the controllers are actually pressed. The recorded sequence was shortened to 576 frames after recording, which results in a sequence of exactly 8 seconds when replayed at 72fps. A second script was used to replay those movements for the camera rig. When building an application to a VR headset, the device does not use the positioning data it gets from the device’s sensors, but rather uses the data stored in the asset file. Using this method, the application can simulate the exact same movement through the scene, over and over again. In the Unity Profiler preferences, Frame Count was set to 576 frames as well. This automatically stops the profiling process as soon as 576 frames were captured.

Android devices offer two methods of remote profiling. Remote profiling allows the developer to profile applications, running on a different device. For example, the profiler can run on a stationary PC while the application to profile runs on a mobile platform. After build-

ing the application and running it on the mobile VR device, the Unity Profiler can record data like CPU and GPU usage, physics, memory, render time etc. for every frame. The recorded profiling data was then saved to a binary file. Using Unity's Profile Analyzer, the binary information could be imported for analysis and comparison with other binaries.(AB 2019)

An advantage of profiling using the Unity Profiler is that the profiler can be controlled via script. When capturing using the Unity Profiler, a small script to disable profiling was created to stop the recording sequence after 8 seconds. This assured the profiler would stop automatically after capturing the sequence.



```
4 using UnityEngine.Profiling;
5
6
7 public class ProfilingData : MonoBehaviour
8 {
9     void Start()
10    {
11        StartCoroutine(ProfilingCoroutine());
12    }
13
14    IEnumerator ProfilingCoroutine()
15    {
16        yield return new WaitForSeconds(8);
17        Profiler.enabled = false;
18    }
19
20 }
```

Figure 3.1. The script used to run the Unity Profiler as soon as the application is started.

The profiling data recorded for every frame of the replayed sequence was then averaged, resulting in a single value for comparison. The reason for averaging a number of frames and not picking out a single frame is because of the varying frame calculation times due to tile based rendering. As we can see in an example below, the average render time is the exact target frequency of the Oculus Quest, if the device can handle the complexity of the scene, which is 13.888ms per frame. Further using this method, the developer can know if the target frame rate was not only reached by one single frame but if it stayed consistently over a longer period time.

The second approach was to capture just a single frame of the scene and inspect all data coming with it. Snapdragon Profiler and RenderDoc both include features to do so. For all tests, the same frame and view was captured for further comparison. The frame captured for analysis was the last frame of the previously recorded sequence which is frame number 576. Snapdragon Profiler and RenderDoc can capture a predefined frame, meaning they will gather all possible informations as soon as frame number 576 is reached. Capturing just a single frame can help to get a deeper understanding of how a frame is created by showing in which order objects were drawn to the screen.

Depending on what was tested, one of those two testing approaches was chosen. For example, to illustrate how different model complexities affect performance, the sequence was

Num. of Tris	Sha. Res.256	Sha. Res. 512	Sha. Res. 1024	Sha. Res. 2048	Sha. Res.4096
25536	13,89	13,89	13,89	13,89	13,89
100457	13,89	13,89	13,89	13,89	14,04
224738	13,89	13,89	13,92	14,01	16,1
397371	14,29	14,31	14,6	15,53	19,17
619364	16,17	16,3	17,17	18,44	22,74
890381	19,55	19,57	20,89	22,17	27,04
1210422	23,72	23,82	25,24	27,1	31,91
1579487	28,52	28,31	29,85	32,57	38,44

Figure 3.2. Averaged frame render times for a number of frames. The green values indicate an average frame render time of 13,89 which is the time to reach for.

replayed and the average render time per frame was measured and used for comparison. For tests like the analysis of post processing effects, a single frame was analysed to see how exactly the effect was applied.

To assure that throughout all tests made, the device temperature would be on a level of about 50° celsius, the Snapdragon Profiler was used to verify it, before any experiments were carried out.

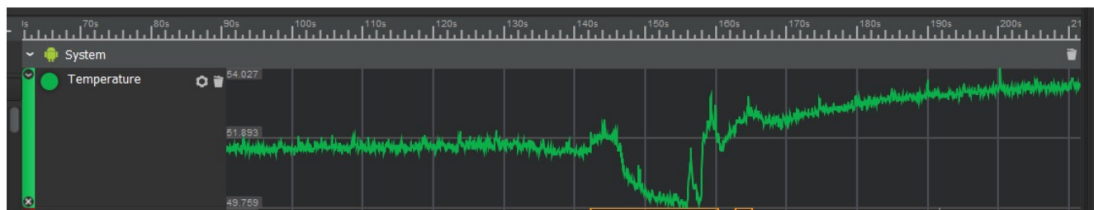


Figure 3.3. Snapdragon Profiler showing temperature data from the Oculus Quest. The graph shows how temperature changes as soon as the application is started. The middle section is where the application is being loaded. After the application is started, the temperature rises constantly.

Each test was executed with the same initial temperature. If the temperature would be higher on one test than on the other, the device might reach its temperature limits earlier and start to lower the CPU or GPU frequency leading to false measurement results.

3.2 Application Design and Scene Setup

For testing purposes a 3D interior model of a house was modeled in the 3D modeling and render software Autodesk Maya 2018. The model has one large room which extends towards two floors and a smaller entrance area. The scene was modeled using as few

polygons as possible, resulting in a 25524 triangle environment including the buildings structure, walls, doors, windows and stairs including handrails. By modeling all geometries using real-world scale, correct size ratios between objects were guaranteed. Real-world scale models can help when adding external assets and don't have to be resized in Unity. Some objects like doors or lamps were modeled in another Maya project and imported in a later step. The entire scene was modeled using a 5x5cm grid. All polygons were designed to fit to this grid or were placed along it. Combining meshes was simplified and guaranteed vertices of multiple objects to be at the exact same position for merging.(Autodesk 2020)

Reducing Geometry

From inside a room, the observer can only see the inner walls. The outer walls only become visible when stepping outside the room. What is invisible does not need to be rendered. Therefore, the back side of the walls will not be processed, which is known as "backface culling". This is why the walls were modeled using a one sided plane, facing inwards, which was extruded on the edges until a closed room was created. When backface culling is active in the modeling software it can also help the modeler to have a better view of the inside.

Some objects, like the entrance door were modeled two sided, so the person viewing the final application could open the door. If the door is meant to stay closed and therefore will only be visible from the inside, the outer faces could be removed. When removing all faces invisible from inside the building, the scene has about 250 less faces (being 500 less triangles). Backface culling can therefore be an easy way to reduce geometry without any loss of quality.

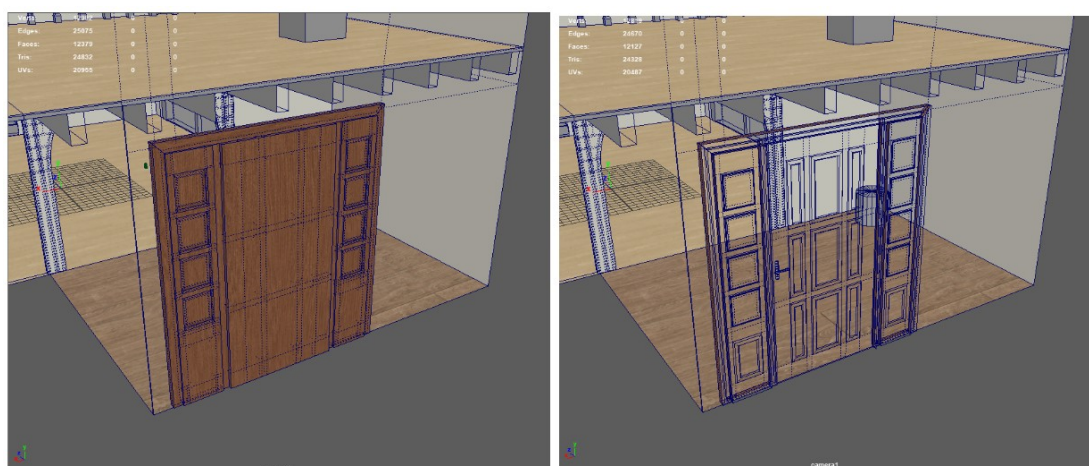


Figure 3.4. Reducing geometry at the main door of the scene.

Combining meshes

The meshes in the scene were combined manually rather than leaving it to Unity's automatic batching. The main reason for doing so was to reduce the number of textures by manually creating texture atlases. Texture atlases were created after manually com-

binning objects and UV-mapping them. Each texture was chosen to have a dimension of 2048x2048 pixels to maintain a lot of details. Before combining the different objects, the texel size for each one of them had to be adjusted. Texels describe pixels in an image texture. The texture size has to be chosen, depending on how many details of a texture should be visible. The floor for example will in most cases be viewed from a distance of 1,5m and above. The ceiling will most likely be viewed from a ever further distance, which is why it will not be necessary to texture those areas with small details and large texture sizes. The final structure of the building which includes floor, walls, doors, stairs and windows consists of seven objects and textures.

Texture baking

As described in the light and shadow section, it is the developers choice to light bake textures early in the modeling stage or at a later time in the game engine. For this test, all light was baked into textures in Autodesk Maya using the V-Ray V3.6 rendering software.

Testing with baked materials: create Unlit materials with textures and add them to every of the 13 scenes upload scene without realtime light in it onto Oculus Quest play and profile add profiling data of each scene to Unitys profile analyzer to calculate the mean render time per frame

4 Results

4.1 Experiment Nr. 01: Polycount

One of the major decisions when modeling 3D objects of any kind is how many details are required. A lower number of polygons used in the meshes leads to a faster application due to every vertex, edge and face needing computing resources. On the other hand, the more polygons are used when modeling, the more detailed and therefore realistic the scene looks. Small polygons can give a better control over shapes of objects, especially when dealing with rounded surfaces. This test examines how the mobile VR device Oculus Quest performs under different scene complexities.

Modern desktop PCs can usually handle a large amount of polygons without any problems. Performance on mobile devices however can differ tremendously on different devices. The number of maximum polygons possible in a scene is therefore limited by the platform the application is built for. According to the general guidelines from Oculus, applications for the Oculus Quest should target 50,000-100,000 triangles or vertices per frame. For comparison, the guidelines for the tethered HMD Oculus Rift states that each frame should be limited to a maximum of 1-2 million triangles or vertices per frame. These limits are highly dependent on the content of the VR-application and the state of the device and should therefore only be taken as a guideline. By testing different versions of a scene, one can easily find out how the device reacts to changes under specific circumstances. (Unity-Technologies 2018c; O. Developers 2020b)

For testing purposes the previously modeled scene was subdivided multiple times to simulate scenes with lower and higher complexity. With Autodesk Maya's "Add Subdivision" tool, edges can be added to polygons, without changing the object's shape. With the linear subdivision option, selected faces or edges can be subdivided into smaller components using an absolute number in the U and the V direction. The floor in the entrance area for example consists of one four sided face in the original model. When being subdivided with a value of 2 by 2, the element is split into 4 faces. When doing the same procedure again, one will end up with 16 faces and so on. By repeating these steps 12 times, 13 versions of the scene were created reaching from 25 536 to 4 160 472 triangles. All versions were then transferred to Unity using the FBX file format.

After importing the 13 mesh variants to Unity, materials and textures had to be applied. For the first experiment, an unlit material using the textures previously baked in V-Ray were used. The eight textures had a size of 2048x2048 pixels and were compressed using Unity's built-in ETC2 sRGB texture compression. The Unity project included no lights and a skybox with the solid color white.

Results

The graph in Figure 4.1 shows the averaged time in milliseconds on the y-axis and the number of used polygons on the x-axis. The sequence of frames was calculated using different complexities of this specific use case. The minimum time a frame has to be visible is 13.8ms which is the frame rate the device is designed for. The Oculus Quest seems to handle scenes with more than 1 million triangles quite well. When using more than 1.2 million triangles, the frame rate starts to drop below 72 fps. Because applications should always be calculated with a bit of an overhead, it is not recommended to not use more than 1 million triangles. This applies to scenes using unlit materials and texture baked lighting.

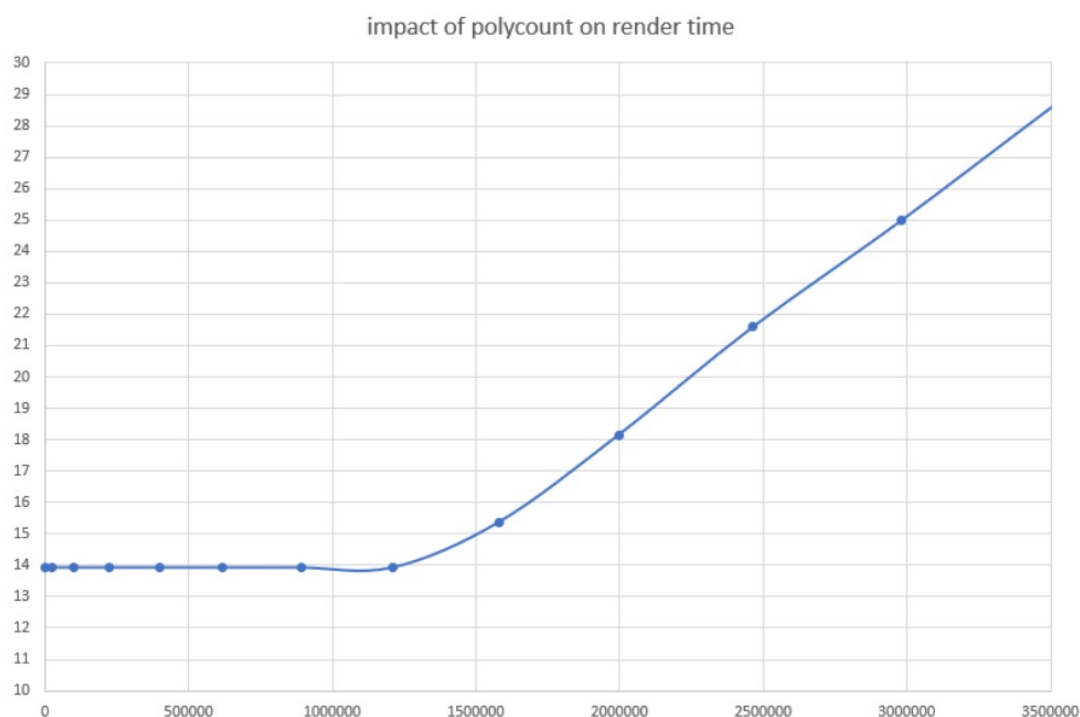


Figure 4.1. Impact of the number of polygons used in the scene on the time to render the sequence.

Many architectural applications might not be as static as this one is. The scene neither has any real time lights to simulate any changes of the light situation, nor are any reflective or transparent objects included. The next test will take a closer look at what happens if real time lights are added to this scene and how different shaders affect performance.

4.2 Experiment Nr. 02: Shaders and Lights

To create more dynamic scenes, animations or interactions can be used to add a bit of realism. Being able to interact with the scene can be more immersing and even fun for the user. Developers might not always want static, non-interactable scenes. This is one of the most important decisions prior to starting an architecture visualization project. One way to achieve this is by adding dynamic real time lighting. One way to implement light calculated in real-time would be to simulate the time of day, to show the observer how light can affect a room at a specific daytime. This can be implemented by using Unitys directional light source as the sun, rendered in real-time mode.

To use real-time light and see the effects of how it reacts to objects, the correct shaders have to be used. Unlit shaders can not be used for this task because, as their name suggests, they are not designed to be affected by real-time light. Other than Unitys unlit shaders, their lit shaders have a number of options to adjust the material's look such as reflectivity, transparency and emission. Instead of just sampling the color of the texture and passing it on without computing anything, lit shaders can be affected by realtime lights and can therefore react to various lighting conditions in a physically correct manner. Objects can cast and receive shadows, light probes and reflection probes can be utilized. Those shading tasks lead to a much higher computation cost.

The goal of this second experiment is to discover how much the change from a Unity Standard Unlit shader to a Unity URP Lit shader affects performance.

At first, the materials of all eight objects making up the scene were changed to use URP Lit shaders. These shaders are typically used for materials which are influenced by surrounding light sources. No lights were used in the first test run, to demonstrate how the change of shaders can affect performance. A "Specular Workflow" was chosen and the Specular Map was set to a black color. All other options were disabled. This leads to a similar look as when using the Unlit shaders because the surrounding light sources don't contribute to the surface of objects.

The test results are shown on the graph below in orange, next to the results of the previous test, using unlit shaders in blue:

We can see how changing shaders which visually look similar, has an impact on performance. When using Lit materials, the Oculus Quest can not reach the targeted frame rate when using more than 750 000 triangles. With unlit shaders the limit is at about 1.2 million triangles. This probably comes from Unity Lit shaders checking if any light sources impact the object, even though there are no lights in the scene. The Lit shaders are obviously designed for use in real time lit scenes, nevertheless this test shows that when applying them incorrectly, they can create unnecessary overhead. This is one reason, custom shaders

4 Results

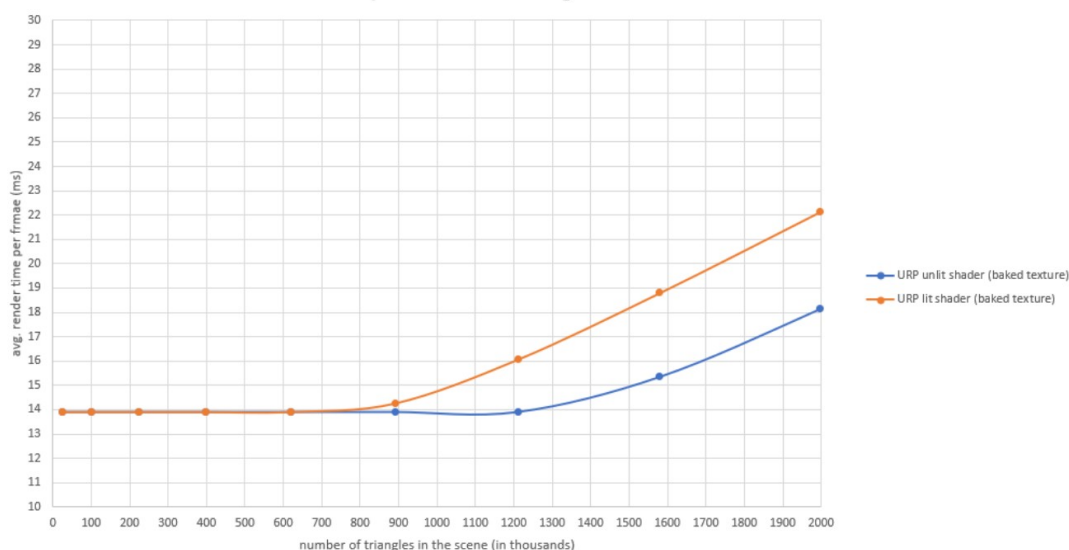


Figure 4.2. Comparison of Unity's Unlit shaders to their Lit shaders with no light sources in the scene.

are often developed for one specific application. It avoids having unnecessary queries and decreases overhead. Of course, using a lit shader for a scene without light sources does not make much sense in a real-world scenario. However, it illustrates why pixel shaders are often self-written by developers. By programming a shader model, it can be perfectly customized for a specific use case without having any unnecessary function calls. Unity's standard shaders might be easy to implement and work well for most use cases but they won't get every bit of performance out of an application.

In the following phase of the experiment, a single directional real time light is added to the scene. The test simulates one of the most common use cases of a single real time light used to represent the sun. The directional light has shadow casting enabled for all objects in the scene, as they are the most significant aspects of directional lights. This certainly leads to much higher performance costs. The test will show how high this cost is and therefore be helpful to know how much processing power it needs. The shadow resolution was chosen as 2048x2048 pixels. This resolution leads to high quality shadows which when moving the light source, don't show any artifacts and look most realistic for the scene. How performance changes when using different shadow resolutions is tested in a later experiment.

The result is added to the diagram of the previous test results as a gray graph for comparison.

The graphs in Figure 4.3 show how big of an impact the decision of using a real-time light has, when shadow casting is activated. For this architectural scene, with the use of a shadow map resolution of 2048x2048 pixels, scene complexity should not exceed 200 000

4 Results

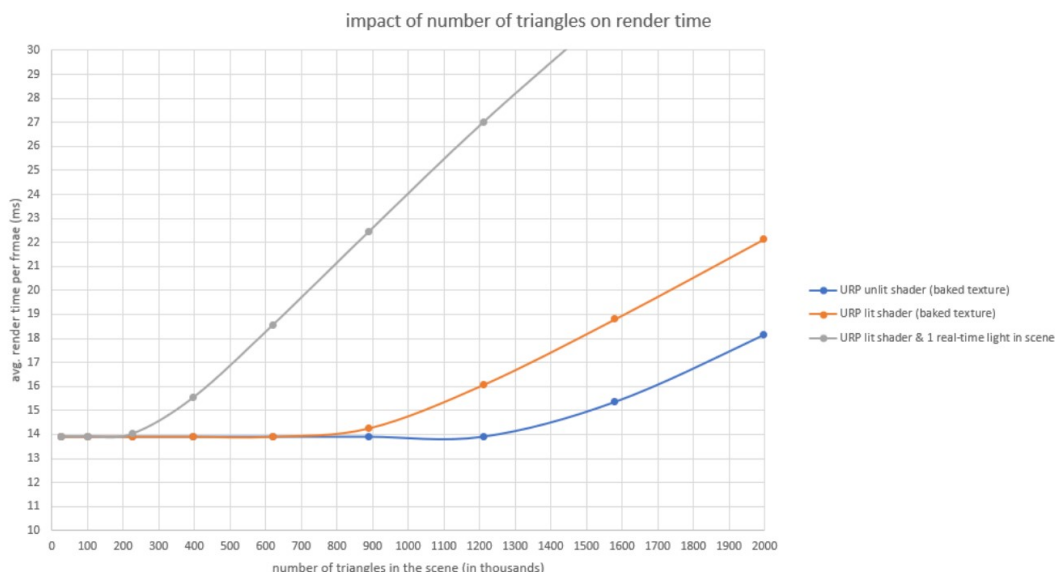


Figure 4.3. Comparison of one real-time directional light in the scene with casting shadows activated to the previously tested scenes using no real-time light.

triangles. Compared to using baked lighting, one has to use 1 million less triangles in this scenario to reach the same frame rate on the Oculus Quest.

The last test involving lights, shadows and shaders adds another real-time point light to the previously tested scene. Point lights can act as interior lights which can be switched off and on by the user or as candle lights which can be moved through the scene. Different to the main directional light, this light source has shadow casting deactivated, due to Unity recommending to not use more than two shadow casting light sources on mobile devices. The light source was placed on a spot where it is visible for most of the replayed sequence. It has a range of 5 meters, and therefore affects about half of the geometries in the scene. The Light Intensity value is set to 3 and the render mode is set to Realtime.

Using a second real-time light in the scene impacted performance further. However, compared to the difference of using no real-time light and using one, the second light certainly affected the scene less. The scene complexity still would have to be reduced by 50% to stay at a frame render time of 13.88 ms. In this particular scene, not more than 100 000 triangles should be used to model the entire scene. This is more than ten times less than when using no real-time lights at all.

Sometimes, having a physically incorrect looking shadow can still be more convincing than having no shadow, because our eyes are very forgiving about how shadows look. A simple blurred circle as a texture placed underneath an object can sometimes help to show that the object is close to the ground.

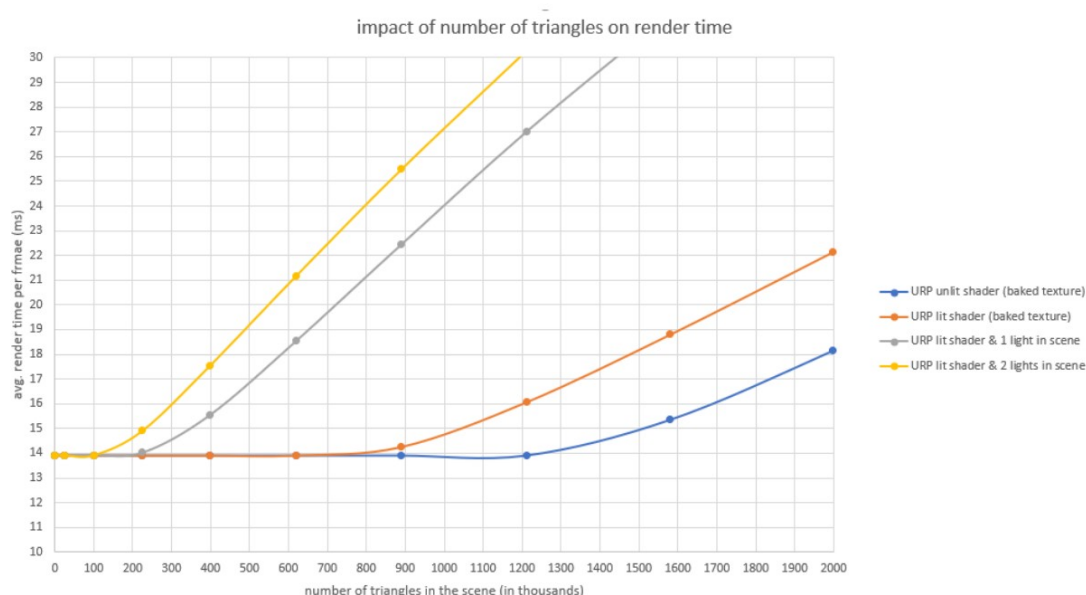


Figure 4.4. The results of adding a second real-time light (yellow graph) into the architectural scene, compared to the previously found values.

4.3 Experiment Nr. 03: Shadow mapping

Baking all light information into textures is one of the most efficient ways of simulating a lit environment. The downsides to this technique is that most objects have to be static and the lighting situation can not be altered. Even though a workaround with exchanging textures runtime could be used, a lot of pre rendering is necessary and flexibility is restricted. Realtime lights are usually rendered per frame, which makes them very flexible. With a real time directional light as the sun, different times of day can be simulated by changing its position and the strength and color temperature of the light can be modified. The directional light can be linked to a procedural sky system which emulates the correct color of the sky depending on the position of the sun. When a scene uses real time lighting, real-time shadows are usually used too. In architectural VR simulations shadows can have a huge impact on scenes realism and the players immersion. Even though it is possible to deactivate shadow casting at all, it would not contribute to the realism of the scene and might even distract the viewer. As mentioned in previous chapters, shadows help to locate objects in space, especially when they are further away from the viewer.

Highly detailed shadows are more expensive to render but can give the scene a much more realistic look. Low resolution shadows may be computed fast but tend to become very blurry, edgy or start to flicker. As with many techniques, the challenge is to find the perfect balance between quality and performance. Mobile devices can only render a limited amount of real-time lights before the performance drops fast and the application will not be able to reach its target frames per second.

The last of this series of experiments gives an overview of how different shadow qualities affect performance and how they perform in relation to each other and scene complexity. To be more specific, it shows how different shadow maps resolutions influence performance in a textured architectural scene using Unity's URP lit shaders.

For the test seven architectural models differing in complexity were used:

- Model 01: 25536 triangles
- Model 02: 100457 triangles
- Model 03: 224738 triangles
- Model 04: 397371 triangles
- Model 05: 619364 triangles
- Model 06: 890381 triangles
- Model 07: 1210422 triangles

All models consist of eight separated objects, each of which has a 2048x2048 resolution texture using Unity's "Normal Quality" ETC2 compression. Mipmap creation was deactivated and none of the textures used transparency. The scenes Multi Sample Anti Aliasing value was set to the recommended value of 4x.

The directional light was used with the standard Unity settings. Soft Shadows were chosen as shadow type and Shadow Cascades were deactivated. The "Shadow Resolution" parameter in the pipeline settings has five different shadow resolutions to choose from:

Map resolution 01: 256 x 256 pixels

- Map resolution 02: 512 x 512 pixels
- Map resolution 03: 1024 x 1024 pixels
- Map resolution 04: 2048 x 2048 pixels
- Map resolution 05: 4096 x 4096 pixels

The five shadow map resolutions were tested with each of the seven different architectural models. The walk-through sequence, recorded for tests with multiple frames, was built and copied to the Oculus Quest. Each of the 35 scene versions were then replayed on the device. The recording process of the sequence is described in detail in the "Testing Methodology" section. While it was replayed on the Oculus Quest, the Unity Profiler was used on a separate PC to capture the profiling data and save them as .data files.

To compare and evaluate the captured data, Unity's Profile Analyzer was used. The aver-

4 Results

age frame render time was calculated for each of the tests and the resulting values were compared. The chart below visualizes the test results using a line graphs for each of the texture resolutions.

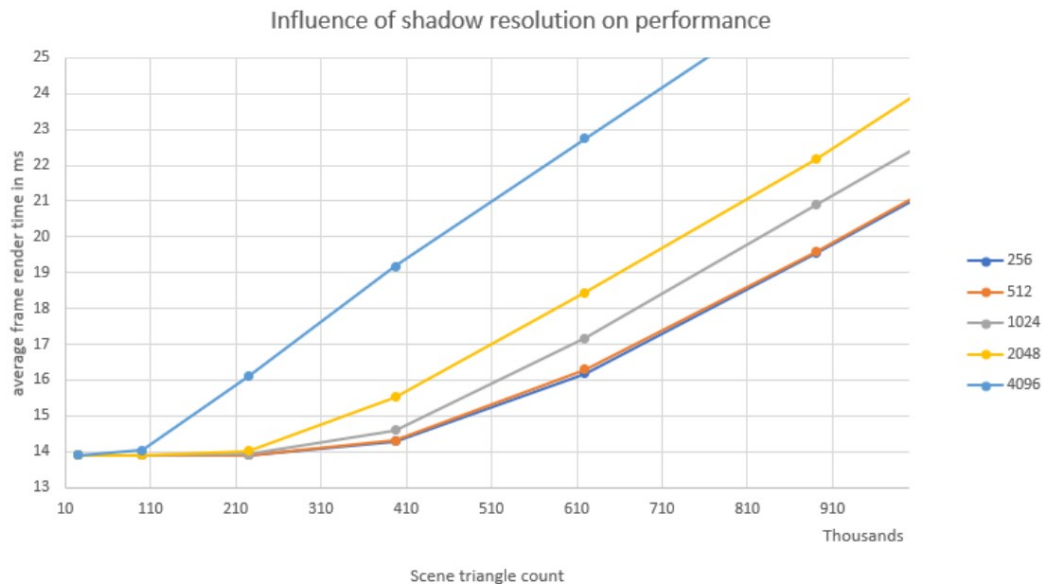


Figure 4.5. The graph shows how performance-intensive different shadow texture sizes are compared to each other in an architectural VR application run on Oculus Quest.

Looking at the graph in Figure 4.5, it seems like the impact different shadow map resolutions have is not evenly distributed. It does not have any impact on the performance when using a texture size of 512x512 rather than 256x256 pixel for the shadow map. Those two texture resolutions performed the same on all models tested on. Using a shadow resolution of 4096 highly affects performance. When using such a high shadow resolution, the graphics card will use a large amount of rendering time on this one task. Looking at Figure 4.6 which shows a capture of one particular frame in RenderDoc, one can see that the time to render a 4096x4096 pixel shadow map takes about half of the entire frame render time. This is why the use of real-time light and shadow maps should be considered carefully.

516-712	3-25	Render Camera	21.56146
517-518	3-3	Render Camera	0.00099
519	4	Set Camera Data	
521	4	Setup Light Constants	
523-532	4-6	Set RenderTarget	0.00797
533-587	7-13	Render Main Shadowmap	10.96839
534-537	7-7	Render Main Shadowmap	0.00083
538-582	8-11	ShadowLoopNewBatcher.Draw	10.96458
584-586	12-12	Render Main Shadowmap	0.00297
587	13	Render Main Shadowmap	

Figure 4.6. Calculating the shadow map uses about 50% of the entire frame render time

4.4 Experiment Nr. 04: Post Processing

Post-Processing applies effects and filters to the rendered scene, before it is displayed on screen. It can add color grading, cinematic effects and give a scene the look of a more natural look.. Because Unitys integrated Post-Processing package was mainly designed to give games a more cinematic look, not all effects can and should be used on virtual reality applications. This test aims to give an overview of how effects, which might have additional benefits for VR applications, impacts render performance for VR devices. For the test, the Post-Processing Stack coming with the URP was used. The following effects are included in the package:

- Bloom
- Channel Mixer
- Chromatic Aberration
- Color Adjustment
- Color Curves
- Color Lookup
- Death Of Field
- Film Grain
- Lens Distortion
- Lift, Gamma, Gain
- Motion Blur
- Panini Projection
- Shadows, Midtones, Highlights
- Split Toning
- Vignette
- White Balance

Effects like “Lens Distortion”, “Death Of Field” or “Motion Blur” are not designed to be used in VR. Effects that have an impact on color and light representation like “Bloom”, “White Balance”, “Color Lookup”. Color Adjustments are more likely to be useful and have the potential to enhance VR experiences.

4 Results

This experiment includes the Bloom, White Balance and Color Lookup Effects. RenderDoc is used to capture and analyse single frames rendered directly on the Oculus Quest. Frame Captures with the post processing effects activated are compared to the same frames without applying the effect. The virtual environment the snapshots are taken from is a texture baked environment with unlit materials and therefore no real-time lights in the scene. This test was performed using a scene consisting of 100 457 triangles. A scene of this complexity was chosen, because of the previously performed tests. It gives the designer a usable number of triangles to create a simple architectural scene, while not spending all resources on mesh rendering. The reason this test is performed using a frame capture tool, instead of just measuring the average frame render time, is to explore how the effects are created and possibly finding out why they might consume a lot of resources.

To set the effects up, the “Volume” script was added to a Unity GameObject. Inside this script, a “Post Processing Profile” has to be created, which lists the different post processing effects. It is necessary to activate the “Post Processing” option in the “CenterEyeAnchor” of the OVRCameraRig, for the camera to render post-processing effects. The first screen capture included the unmodified scene which should serve as a reference.

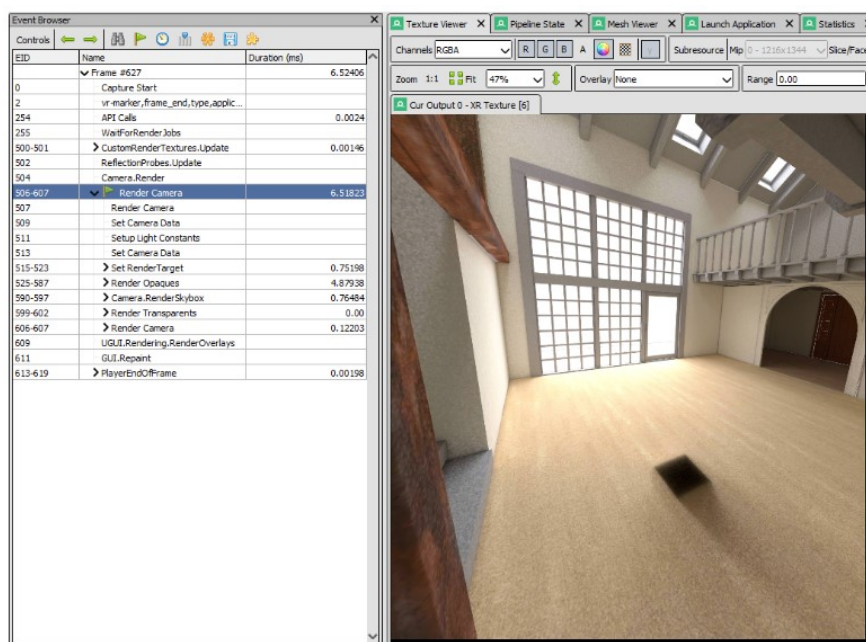


Figure 4.7

White Balance

When trying to either remove color cast, so items appear in a correct white, or when adding a colder or warmer feeling to the rendered image, the white balance comes in handy. The effect can be used to simulate different day times by changing the color temperature of the scene to a cold looking tone when representing daytime and a warmer look when simulating dawn.

4 Results

Unity's white balance effect consists of two properties, "Temperature" which is used to set the color temperature and "Tint". With the temperature slider, higher values result in a warmer, more orange looking image, lower values in a colder looking image containing more blue colors. The "Tint" slider can be used to compensate for magenta or green color tints. For demonstration purposes and better visibility, the color temperature is altered in an exaggerated way which results in a very cold looking image. The "Temperature" slider was set to a value of -50 while the "Tint" slider wasn't modified:



Figure 4.8

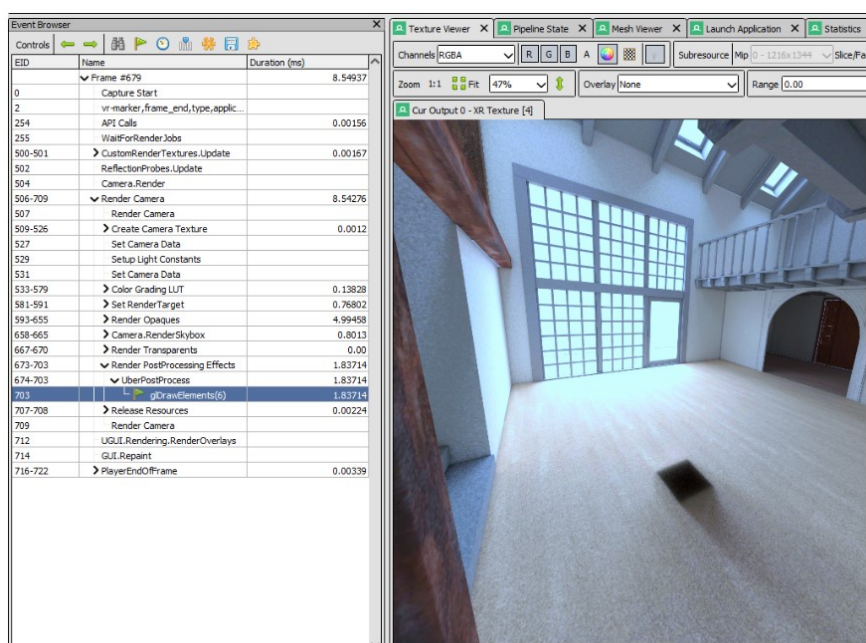


Figure 4.9

By calculating the time duration for each draw call with RenderDoc, we can see that the color grading effect took about 1.83ms to render. Considering the time to render the entire frame took about 8.54ms, the color grading effect was about 20% of the entire frame-rendering time. A further test changing the value of the Color Temperature to -100 did not have any impact on rendering times and resulted in the same render time percentages. Adding a "Tint" effect did not affect performance either.

Bloom effect

Bloom gives the impression of a very bright light and therefore blurs and brightens up areas around very bright spots in the image. Using this effect can create very elegant looking environments. Especially in places where high contrasts occur, hard edges between light and dark areas might create an unnatural look.

4 Results

In this testing setup, a “Bloom” effect with an Intensity value of 20 was added to the texture baked scene. The render time for the entire frame took about 11 milliseconds. The Post Processing Effect took 4.46 ms of that time which is 44,6

Taking a closer look at underlying draw calls of the Bloom effect, one can see that the effect consists of a number of down-sampled images. In this case the Bloom effect added 22 additional events, each taking between about 0.02ms and 0.52ms to calculate, before the color grading effect was added in the last draw call.

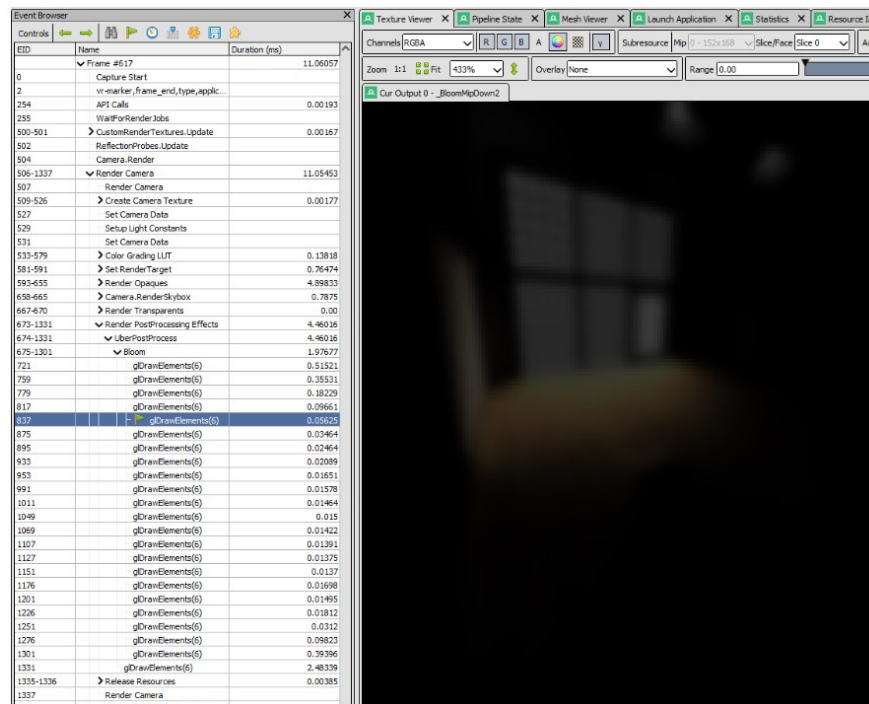


Figure 4.10

The same scene was tested with an increased intensity value of 100, which didn't have any impact on performance. Having two post processing effects activated, didn't take about as long as rendering both of the effects in sequence.

When not deactivating the Post Processing effects the correct way, a draw call might be executed for it even though the effect will not be visible in the final image. Unity seems to do this, regardless of the “Volume” script or its GameObject, being deactivated. This is why deactivating the “Post Processing” option in the camera properties is necessary for the API not to send this draw call.

In general, post processing effects seem to be quite resource-intensive and therefore might not be the best way to add certain effects. Color temperature for example might be more efficient to apply, by choosing the correct color before baking the textures. Also color grading effects can be baked into the textures color information rather than applying it while rendering. This has the advantage of saving one or more render passes but therefore can

4 Results

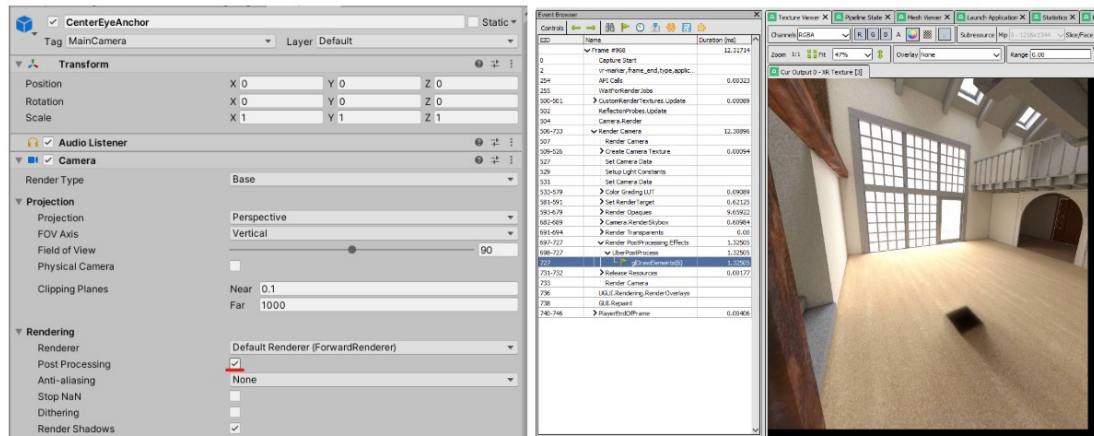


Figure 4.11

not be modified at runtime.

According to Oculus, the post effects are expensive because the input texture is first written from tile memory to the external memory and then used again as data input. This can lead to long frame render times because resolves are so expensive to compute. (Ferreira 2019, December 5)

5 Discussion

The outcome of the four tests has at least two findings. One is the actual outcome of the experiments, which is device specific but can be useful for anyone developing for the same or a similar device. The second finding is that the test results show how the capabilities of mobile devices can be tested. Knowing which profiling tools are suited best for a certain problem can be difficult to know, if developers have not tested them on their own. The results show the considerable difference a certain change in parameters can make regarding performance. It shows how a combination of a complex scene with real-time lights will not be achievable on mobile VR devices like the Oculus Quest, but might be when applying texture baking.

The results of the experiments can be used to describe which steps are necessary in what order to achieve the best possible result for an architectural visualization. They therefore provide an answers to the question of "What strategies can be utilized to optimize high-end architectural scenes for mobile VR devices?" By combining the outcome of the experiments with guidelines from multiple sources like Oculus, Unity, Snapdragon etc. these key rules and guidelines were found to follow when designing architectural visualizations for mobile VR devices:

1. **The capabilities and limits of the target device should be tested first.**
Knowing how many triangles, lights, textures, etc. a device can handle is the very first thing to find out.
2. **Deciding if real-time light is necessary or a scene can work without it should be one of the first decisions to make.**
If dynamic real-time light is not necessary, baking light into textures should be considered.
3. **Scene complexity should be kept as low as possible when using real-time lights. Models can contain more details when using baked lighting instead.**
When dynamic light is necessary, the target device should be tested to know how much detail can be obtained in the scene.
4. **When using real-time lights, performance tests on shadow map resolutions should be performed to find a good balance between performance and quality.**

Testing different shadow map resolutions can help to see how they perform visually as well as regarding render times.

5. Complex shaders should be avoided. The simpler the shader model is, the better.

Using the wrong shaders can lead to unnecessarily long render times. Shaders specifically designed for mobile devices should be tested and compared for best performance on individual applications.

6. Post processing effects should only be used when absolutely necessary.

Color changes can be baked into textures themselves which does not add any overhead to the rendering process. Baking LUTs into textures themselves can serve as a solution.

These are some further guidelines gathered during research which should also be considered:

- Leaving headroom will increase battery life and keep power consumption low
- Using texture compression will decrease texture file size and therefore render times
- Combining meshes manually or by using Unity's Batching will decrease draw calls
- Using as few different materials as possible will also decrease the number of draw calls
- Atlasing textures will lower the number of textures needed

With these test results, it becomes apparent that knowing the performance limits of the device an application is designed for is crucial. When trying to reach a certain performance target, profiling becomes inevitable. The developer depends on the profiling tools available and has to know where and how to use them. An important first step in the optimisation process is knowing where to start.

The most significant piece of information, which performance tests help to establish is, which algorithms and decisions are more performance-heavy than others. This will help to decide where to look for problems and where to start optimizing. To find and analyze a certain problem, the right tools are required. By testing multiple profiling tools in multiple areas, their strengths and weaknesses can be determined. For the experiments in this paper, a number of applications were examined and tested.

The reason these profiling tools were chosen over others for testing, is because they had certain benefits, such as providing more information than others, when profiling an Android based application for the Oculus Quest. Depending on the platform one is developing on, the tools might certainly vary. Every GPU, driver and application is different and will lead

to different results when profiling performance. This is why the only way to determine if an application reaches the expected performance goal is by testing it in as many ways as possible. The following list can be seen as a starting point for profiling applications.

To profile mobile applications and find performance issues or bottlenecks, the following tools are recommended:

- Head up displays such as the **OVR Metrics Tool** are useful as real time diagnostic tools, while using the HMD. They can be used to monitor the application while using it, to find certain areas of the scene which might have issues while rendering. Some performance decreases might not be perceptible by the viewer instantly. This could be light sources, reflective objects, complex geometries which are only causing problems when in frame. Because there is no additional hard- and software necessary to profile, it is easy and fast to use.
- **Unity Profiler** in combination with **Profile Analyzer** are the ideal tools for analysing short sequences of game play and determining if an application is CPU- or GPU-bound. When developing in Unity, they are the easiest tools to use for code analyses as well as testing graphics performance. Both of the tools are designed for profiling the tasks that are run on the CPU. The Unity Profiler shows which processes are taking up the most time to process and can visualize how performance varies over time. Both of the tools were used for the majority of experiments conducted for this paper. Due to their ease of use and fast as well as the advantage of being able to analyze the profiled data using the Profile Analyzer, were some of their key benefits.
- For GPU specific analysis, **RenderDoc** is a good tool for visualizing the chronological order of how the scene is built. This can help to determine if the application actually does what it is supposed to do or if any unwanted draw calls are executed. It delivers a variety of data on how a single frame is created, which is helpful for establishing why certain draw calls, like post processing effects, take so much time to compute. Furthermore, it is useful for examining what information is stored in the device's buffers and if geometries are culled correctly. Visualizing the input and output geometries can be useful for anyone creating self written vertex shaders.
- To get a more in-depth look at what processes are taking place, vendor-specific tools can be helpful. They can access internal performance metrics and therefore help to find what parts of the pipeline might cause bottlenecks. Tools like **Systrace**, which were specifically designed for tile-based profiling can give a tile by tile information on what is happening on the device. As a whole, it offers less information than other Profiling Tools like Snapdragon Profiler. However, it lets developers take a closer look at how frames are rendered on a hardware level, showing how individual bins are rendered and how long it took. At the time of writing, the tool is quite rudimental

and offers only a limited number of features to help finding bottlenecks or any other problems. However, it is a useful tool for accessing more pieces of information about the device itself.

- Platform specific profiling tools have the advantage of having access to a large amount of hardware information, which other applications cannot access as easily. For devices based on chipsets by Qualcomm, like the Oculus Quest, this profiling tool is the Snapdragon Profiler. These tools can receive information about thermal data, battery consumption, memory usage as well as CPU and GPU data in real-time, making them useful for monitoring and capturing live data at any time. **Snapdragon Profiler** can even trace data when the device is in standby mode which none of the other tools tested were capable of, except for Systrace. Because it can visualize what happens before and after opening an application, it is useful for gathering information about power consumption and thermal data.

Problems occurred while testing: While performing these experiments, some problems occurred using the Post Processing Stack. In order to compare how a frame is rendered when Post Processing is active versus to when it is deactivated, the same frame was captured twice. The first frame was captured using the Post Processing Volume script, placed on a newly created GameObject. After capturing the frame with the effects applied, the GameObject was deactivated using the “enable/disable Object”-checkbox at the top of the GameObject.

Doing so leads to all effects being deactivated in the Editors View as well as in the final application. Looking at the captured frame however, it can be observed that the effect is being rendered as before, it simply is not applied to the final framebuffer and therefore will not be visible in the final VR application. This occurrence exemplifies why profiling is important. Without analyzing what exactly was happening, these Post Processing Effects would have been computed on every frame without ever being visible for the end user. Therefore, it is generally a good idea to delete any unused objects from the Scene hierarchy, as long as they will not be used within the application at a later time.

6 Conclusion

Virtual reality can be found in various fields, including the area of architecture and construction. Due to the technological development of mobile VR devices in recent years, it was only a matter of time for architects as well as software developers to start implementing architectural scenes for those small devices. This mobile VR hardware requires specific performance handling and if done right, can lead to photorealistic results. This thesis helps to find a way to achieve such a high level of quality. Therefore it is necessary to know how these devices are designed. Having this solid knowledge of how the underlying hardware of today's mobile devices works is essential for tracing down hardware and software problems.

When planning to create a mobile architectural application, measuring device capabilities should be the very first step, even before starting the modeling process. Conducting performance tests can show when a device's performance limit is reached. When designing, modeling and implementing a scene first without certain targets, the developer might end up spending a considerable amount of time improving it afterwards. Thus, it is crucial to establish goals before starting the design process. If, for example, the maximum possible polygon count is defined early, modelers will know exactly what scene complexity to aim for. The content can be created to perfectly fit the device resources and will not reach the limits of the device in any situation.

Depending on the device specifications and performance limits, developers have to make the decision on how dynamic they want their scene to be. This decision will influence the further workflow and design process. They have to decide where they are willing to spend the majority of frame render time and which tasks should be prioritized. Certain features will have to be neglected or abandoned entirely because they will not benefit the application enough to justify their performance cost. Post processing effects might be one of those. However, for some features, there might be different solutions which lead to a less expensive result. For example, baking post processing into the textures rather than computing them in real time will not lead to any performance decline and will therefore have barely any drawbacks.

There are thousands of decisions to make when developing applications. Some of them have a major impact on how applications perform. This depends on the application itself, which is why bottlenecks are application-dependent and can not be solved in a general

manner. The performance problems are unique to every application and there is no secret trick to make an application run fast. Profiling tools can help to identify where bottlenecks occur and can give further insight into the processes of rendering a frame. Even if no performance problems seem to occur, there might be unnecessary draw calls which do not contribute to the final render in any way. Having a comprehensive knowledge of how the rendering process works on a hardware and software level and being familiar with the right tools for solving problems are key to creating high-quality architectural visualizations. When applied properly, the resources of the mobile VR device will be perfectly utilized and the user experience will highly benefit from it.

With future mobile devices, hardware capabilities will increase and so will the techniques and algorithms which accelerate the render process to reach even better and more realistic results. With these new technologies, new ways to test and profile applications will appear. Further research could consider the question of what the workflow from a BIM or CAD modeling software till the final mobile VR visualization could look like. With the fast development of technology, real-time light calculation via ray tracing on mobile devices might also evolve to a valuable topic for future research. Furthermore, real-time area lights might be introduced for a use in mobile devices.

Bibliography

- McCormack, J., & McNamara, R. (2000). Tiled polygon traversal using half-plane edge functions, In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware - HWWS '00*. the ACM SIGGRAPH/EUROGRAPHICS workshop, Interlaken, Switzerland, ACM Press. <https://doi.org/10.1145/346876.346882>
- Sharp, C., & Leger, J. (2009, August 20). *QCOM tiled rendering* [Qualcomm technologies]. Retrieved August 29, 2020, from https://www.khronos.org/registry/OpenGL/extensions/QCOM/QCOM_tiled_rendering.txt
- Cozzi, P., & Riccio, C. (2012). *OpenGL insights*.
- Kerin, A. (2013, October 8). *The thermal efficiency behind smartphone trends* [Qualcomm]. Retrieved August 30, 2020, from <https://www.qualcomm.com/news/onq/2013/10/09/thermal-efficiency-snapdragon-processors-under-screen-and-behind-trends>
- Qualcomm. (2013). Power performance white paper. Qualcomm Technologies, Inc. Retrieved May 2, 2020, from <https://developer.qualcomm.com/qfile/27292/trepn-whitepaper-apps-power.pdf>
- Lei, Z., & Shengchao, G. (2014). Thermal management of ARM SoCs using linux CPUFreq as cooling device, 6.
- Qualcomm. (2015, May 1). Adreno OpenGL ES developer guide. https://developer.qualcomm.com/qfile/28557/80-nu141-1_b_adreno_opengl_es_developer_guide.pdf
- Sommefeldt, R. (2015, April 2). *A look at the PowerVR graphics architecture: Tile-based rendering* [Imagination]. Retrieved August 30, 2020, from <https://www.imgtec.com/blog/a-look-at-the-powervr-graphics-architecture-tile-based-rendering/>
- Luna, F. (2016, April 19). *Introduction to 3d game programming with DirectX 12* [Google-Books-ID: gj6TDgAAQBAJ]. Stylus Publishing, LLC.
- Schwartz, R. (2016, December 15). *Profiling VR apps for better performance* [Qualcomm developer network]. Retrieved August 30, 2020, from <https://developer.qualcomm.com/blog/profiling-vr-apps-better-performance>
- Teschner, M. (2016, January 13). *Image processing and Computer graphics - rendering pipeline*. Computer Science Department University of Freiburg. Retrieved July 22,

- 2020, from https://cg.informatik.uni-freiburg.de/course_notes/graphics_01_pipeline.pdf
- De Vries, J. (2017). *LearnOpenGL - hello triangle* [Learn OpenGL]. Retrieved August 29, 2020, from <https://learnopengl.com/Getting-started/Hello-Triangle>
- Doppioslash, C. (2017). *Physically based shader development for unity 2017: Develop custom lighting systems*.
- Eising, P. (2017, December 7). *What exactly IS an API?* [Medium]. Retrieved May 29, 2020, from <https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f>
- Eskofier, D. (2017, July 10). *Unite europe 2017 - every millisecond counts: How to render faster for VR*. Unite Europe 2017. Retrieved August 30, 2020, from https://www.youtube.com/watch?v=hNEVHYr1_CM
- Lopez Mendez, R. (2017, May 1). *How to get the most out of mobile VR in unity*. ARM. Retrieved July 2, 2020, from https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiqt-Kd48LrAhXxRBUIHbRWB1EQFjAAegQIBBAB&url=https%3A%2F%2Fdeveloper.arm.com%2F-%2Fmedia%2FFiles%2Fpdf%2Fgraphics-and-multimedia%2FHow_to_Get_the_Most_Out_of_Mobile_VR_in_Unity.pdf&usg=AOvVaw3u9Ap8c5NuTmJVe1kzHVeW
- Qualcomm. (2017). Snapdragon 835 mobile platform product brief. Qualcomm Technologies, Inc. Retrieved July 26, 2020, from <https://www.qualcomm.com/media/documents/files/snapdragon-835-mobile-platform-product-brief.pdf>
- Turner, K., & Schell, M. (2017, July 10). *Unite europe 2017 - performance optimization for beginners*. Retrieved August 30, 2020, from <https://www.youtube.com/watch?v=1e5WY2qf600>
- Unity-Technologies. (2017a, October 26). *Unity - manual: Draw call batching*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/DrawCallBatching.html>
- Unity-Technologies. (2017b, June 20). *Unity - manual: Single-pass stereo rendering for android*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/Android-SinglePassStereoRendering.html>
- Akenine-Möller, T., Haines, E., & Hoffman, N. (2018). *Real-Time Rendering, Fourth Edition*.
- ARM. (2018, October 19). *OpenGL ES SDK for android: Using multiview rendering*. Retrieved August 30, 2020, from <https://arm-software.github.io/opengl-es-sdk-for-android/multiview.html>
- Garrard, A. (2018). *Moving mobile graphics: Mobile graphics 101*. Samsung R&D Institute UK. https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-20-66/siggraph_2D00_2018_2D00_mmg_2D00_1_2D00_101_2D00_andrewg.pdf

- Jagneaux, D. (2018, October 3). *Oculus quest to have 'active cooling' from internal fan* [UploadVR] [Section: Article]. Retrieved August 30, 2020, from <https://uploadvr.com/oculus-quest-to-have-active-cooling-from-internal-fan/>
- Palandri, R. (2018, September 26). *Oculus connect 5 - reinforcing mobile performance with RenderDoc*. Retrieved August 30, 2020, from https://www.youtube.com/watch?v=CQxkE_56xMU&t=88s
- Qualcomm. (2018a, May 14). *FlexRender* [Qualcomm]. Retrieved August 30, 2020, from <https://www.qualcomm.com/videos/flexrender>
- Qualcomm. (2018b, October 2). *Snapdragon 835 mobile platform* [Qualcomm]. Retrieved July 9, 2020, from <https://www.qualcomm.com/products/snapdragon-835-mobile-platform>
- Unity-Technologies. (2018a). *Unity - manual: Frame debugger*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/FrameDebugger.html>
- Unity-Technologies. (2018b). *Unity - manual: Getting started with the profiler window*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/ProfilerWindow.html>
- Unity-Technologies. (2018c). *Unity - manual: Modeling characters for optimal performance*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/ModelingOptimizedCharacters.html>
- Unity-Technologies. (2018d). *Unity - manual: Profiler overview*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/Profiler.html>
- Unity-Technologies. (2018e). *Unity - manual: The progressive lightmapper*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/progressive-lightmapper.html>
- AB, S. S. (2019). *Remote profiling*. Retrieved August 30, 2020, from <https://memprofiler.com/online-docs/default.htm#!remoteprofiling.htm>
- Alfonse. (2019, April 8). *Rendering pipeline overview - OpenGL wiki*. Retrieved August 29, 2020, from https://www.khronos.org/opengl/wiki/Rendering__Pipeline__Overview
- Chaosgroup. (2019). *Texture baking - v-ray 3.6 for maya - chaos group help*. Retrieved August 30, 2020, from <https://docs.chaosgroup.com/display/VRAY3MAYA/Texture+Baking>
- Dasch, T. (2019, November 19). *PC rendering techniques to avoid when developing for mobile VR* [Oculus developers]. Retrieved August 29, 2020, from <https://developer.oculus.com/blog/pc-rendering-techniques-to-avoid-when-developing-for-mobile-vr/>
- Developers, O. (2019a, August 6). *Getting started w/ the unity GPU profiler for oculus quest and go | oculus*. Retrieved August 30, 2020, from <https://developer.oculus.com/blog/getting-started-w-the-unity-gpu-profiler-for-oculus-quest-and-go/>

- Developers, O. (2019b). *Multi-view*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/native/android/mobile-multiview/>
- Developers, O. (2019c). *Using single pass stereo rendering and stereo instancing*. Retrieved August 30, 2020, from https://developer.oculus.com/documentation/unity/unity-single-pass/?locale=es_LA
- Ferreira, C. (2019, December 5). *How to optimize your oculus quest app w/ RenderDoc: Getting started + frame capture | oculus*. Retrieved August 30, 2020, from <https://developer.oculus.com/blog/how-to-optimize-your-oculus-quest-app-w-renderdoc-getting-started-frame-capture/>
- Homewood, L. (2019, October 9). *Optimize your game with the profile analyzer - unite copenhagen 2019*. Retrieved August 30, 2020, from <https://www.youtube.com/watch?v=0lzqdDdE9Tc&t=960s>
- Qualcomm. (2019). *Anti-aliasing with adreno* [Qualcomm developer network]. Retrieved August 30, 2020, from <https://developer.qualcomm.com/software/snapdragon-profiler/app-notes/anti-aliasing-with-adreno>
- Trevor, D. (2019, October 25). *OVR metrics tool + VrApi: What do these metrics mean? | oculus*. Retrieved August 30, 2020, from <https://developer.oculus.com/blog/ovr-metrics-tool-vrapi-what-do-these-metrics-mean/>
- Unity-Technologies. (2019, November 26). *Optimizing graphics in unity* [Unity learn]. Retrieved August 30, 2020, from <https://learn.unity.com/tutorial/optimizing-graphics-in-unity>
- ARM. (2020). *Arm mobile studio | optimization advice for graphics content on mobile devices* [Arm developer]. Retrieved August 30, 2020, from <https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/advice/cpu-bound>
- Arm. (2020). *Mali GPU application optimization guide*. Retrieved August 30, 2020, from <https://developer.arm.com/documentation/dui0555/a/introduction/the-mali-gpu-hardware/tile-based-rendering>
- Autodesk. (2020). *Maya software | computer animation & modeling software | autodesk*. Retrieved August 30, 2020, from <https://www.autodesk.com/products/maya/overview>
- Bedekar, N. (2020, February 4). *Vulkan support for oculus quest in unity (experimental) | oculus* [Oculus developers]. Retrieved May 29, 2020, from <https://developer.oculus.com/blog/vulkan-support-for-oculus-quest-in-unity-experimental/>
- Developers, G. (2020, May 18). *Multithreaded rendering support | ARCore* [Google developers]. Retrieved August 30, 2020, from <https://developers.google.com/ar/develop/unity/mt-rendering?hl=de>
- Developers, O. (2020a). *Draw call cost analysis for quest*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/native/android/mobile-draw-call-analysis/>

- Developers, O. (2020b). *Guidelines for VR performance optimization*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/native/pc/dg-performance-guidelines/>
- Developers, O. (2020c). *Monitor performance with OVR metrics tool*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/tools/tools-ovrmetricstool/>
- Developers, O. (2020d). *Multisample anti-aliasing analysis for quest*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/native/android/mobile-msaa-analysis/?device=QUEST>
- Developers, O. (2020e). *Use GPU systrace for render stage tracing*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/tools/tools-gpusystrace/>
- Karlsson, B. (2020). *RenderDoc documentation*. Retrieved August 30, 2020, from https://renderdoc.org/docs/getting_started/faq.html
- Lee, J., & Palandri, R. (2020, June 19). *Improving GPU profiling on oculus quest | oculus*. Retrieved August 30, 2020, from <https://developer.oculus.com/blog/improving-gpu-profiling-on-oculus-quest/>
- Murray, J. W. (2020). *Building virtual reality with unity and SteamVR*.
- Oculus. (2020). *Testing and performance analysis*. Retrieved August 30, 2020, from <https://developer.oculus.com/documentation/unity/unity-perf/>
- Qualcomm. (2020a). *Avoid GMEM loads* [Qualcomm developer network]. Retrieved August 30, 2020, from <https://developer.qualcomm.com/software/snapdragon-profiler/app-notes/avoid-gmem-loads>
- Qualcomm. (2020b). *Snapdragon profiler* [Qualcomm developer network]. Retrieved August 30, 2020, from <https://developer.qualcomm.com/software/snapdragon-profiler>
- Samsung. (2020). *GameDev - build* [Samsung developers]. Retrieved August 29, 2020, from <https://developer.samsung.com/GameDev/resources/articles/gpu-framebuffer.html>
- SideFX. (2020). *How to render using mantra*. Retrieved August 30, 2020, from <https://www.sidefx.com/docs/houdini/render/render.html>
- Unity-Technologies. (2020a, April 27). *Diagnosing performance problems - 2019.3* [Unity learn]. Retrieved August 30, 2020, from <https://learn.unity.com/tutorial/diagnosing-performance-problems-2019-3>
- Unity-Technologies. (2020b, February 3). *HDRP area lights* [Unity learn]. Retrieved August 30, 2020, from <https://learn.unity.com/tutorial/hdrp-area-lights>
- Unity-Technologies. (2020c, August 25). *Unity - manual: CPU usage profiler module*. Retrieved August 30, 2020, from <https://docs.unity3d.com/2020.1/Documentation/Manual/ProfilerCPU.html>

- Unity-Technologies. (2020d, August 25). *Unity - manual: Light mode: Baked*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/LightMode-Baked.html>
- Unity-Technologies. (2020e, August 25). *Unity - manual: Optimizations*. Retrieved August 29, 2020, from <https://docs.unity3d.com/Manual/MobileOptimisation.html>
- Unity-Technologies. (2020f, August 25). *Unity - manual: Types of light*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Manual/Lighting.html>
- Unity-Technologies. (2020g, August 25). *Unity - scripting API: Profiler*. Retrieved August 30, 2020, from <https://docs.unity3d.com/ScriptReference/Profiling.Profiler.html>
- Unity-Technologies. (2020h). *Unlit shader - universal RP 7.1.8*. Retrieved August 30, 2020, from <https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.1/manual/unlit-shader.html>

List of Figures

2.1	Graphics Rendering Pipeline	6
2.2	Clipping Process	9
2.3	Screen mapping process	9
2.4	Color Buffer and Depth Buffer of an architectural scene	11
2.5	Primitives to tile buffer	13
2.6	Tile buffer data flow	13
2.7	Tile buffer data flow	16
2.8	Different Types of Light Sources	18
2.9	RenderDoc Shadow Analysis	19
2.10	Shadow Map Generation	19
2.11	Unity Surface Options	20
2.12	Aliasing and Anti-Aliasing	22
2.13	Multi view analysis in RenderDoc	23
2.14	Multithreaded Rendering	24
2.15	Unity Profiler Example	27
2.16	Snapdragon Profiler	30
2.17	Graphics Pipeline in RenderDoc	31
2.18	Render order visualization in RenderDoc	32
2.19	Systrace visualizes tile-based rendering approach	32
2.20	Close-up of tile-based rendering using Systrace	33
2.21	Systrace Parameters	33
3.1	Script used to run the profiler	35
3.2	Showing averaged profiling data	36
3.3	Snapdragon Profiler showing temperature data from the Oculus Quest	36
3.4	Reducing geometry at the main entrance of the scene	37
4.1	Results of experiment 01	40
4.2	Results of experiment 02 - graph 01	42
4.3	Results of experiment 02 - graph 02	43
4.4	Results of experiment 02 - graph 03	44
4.5	Results of experiment 03 - graph 01	46

4.6	2048x2048 pixel Shadow Map in RenderDoc	46
-----	---	----

Appendices

A Appendix

CameraRig movement recording and playback for testing purposes

The following section will explain how the test setup to simulate the same VR-user movement multiple times was created.

The so called "MotionTool" can be used by developers to record position and rotation data of Game Objects while in Unitys play mode. The information is stored in a .motion file from which the motions can be replayed. Originally intended for capturing a 360-degree video by a company called Telerik, the tool can also be used to capture the movement of a camera rig, the play area and the controllers, to be replayed later. This has two purposes: By simulating a VR walkthrough which is repeatable, we can change certain details in the scene or the rendering process, run the scene with different parameters and compare the two walkthroughs regarding performance. The second reason is to ensure the tests are repeatable and the experiments can be re-run at any time without obtaining different results.

After a free registration on their website, the tool can be downloaded for free. It is licenced under the Open Source MIT licence.

To get started with the tool, they provided two video tutorials which describe how the scripts can be used for replaying the movement of an avatar:

- Baseline Tutorial: <https://www.youtube.com/watch?v=ppDjy4bTy2M>
- Working with VR avatars: https://www.youtube.com/watch?v=_HP-smcbtnXQ

To replay the movement of the player on the VR device itself, some changes have to be made. In the following section, the necessary steps to record and replay VR movements on a device will be explained.

After importing the package into Unity, there are two scripts which are important for this to work. MotionRecorder.cs and MotionReplayer.cs. To record motion, the MotionRecorder script has to be attached to any GameObject, preferably the PlayerController. The "Custom Origin" parameter needs to stay empty. Under "Motion Data", a newly created .motion file

has to be linked. Three elements have to be created under the “Nodes” drop down where the following game objects from the PlayerController need to be assigned: The OVRCameraRig (or any other camera rig) itself, the TrackingSpace and the CenterEyeAnchor.

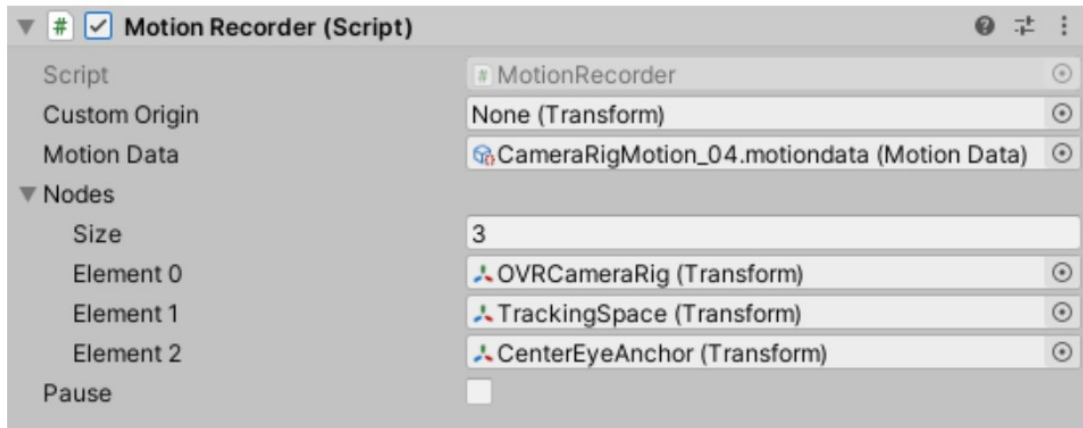


Figure 6.1

Recording movement

The easiest way to record the player's movement is to plug in the HMD via cable (when using untethered devices) and run the scene from the PC first. As soon as the play button is pressed, the scripts will start to log the positions of the HMD for every frame. These informations will be stored inside the previously created .motion file. The recording stops as soon as the play mode is canceled.

Replaying movement:

To replay the exact movement in the VR headset, the Motion Recorder script has to be deactivated and the motion Replayer script needs to be added and activated. The previously saved Motion Data File must be added and the transforms of the GameObjects must be added by clicking the “Try Auto-Assign” button. The OVR Camera Rig Script in the OVR Camera Rig GameObject has to be deactivated, otherwise the device will still try to use the VR devices position and rotation data. As soon as the scene is started, the Motion Replayer will update the position of all the objects inside the “Nodes” list which will simulate the exact same movement previously recorded.

Have fun.