HAW HAMBURG

BACHELORARBEIT

Visualizing Stage-Light Fixtures on Standalone Virtual Reality Headsets using the Unity Engine

vorgelegt am 07. Februar 2024

Tom Milter

Erstprüferin: Dipl. Des. M.Sc. Anke von der Heide
Zweitprüfer: Prof. Dr. Ing. Roland Greule

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Department Medientechnik
Finkenau 35
22081 Hamburg

Zusammenfassung

Während Meta ihre Vision des ”Metaverse”versucht zu fördern und nachdem, im Zuge der 2019 COVID Pandemie plötzlich viele Menschen Erfahrung mit Events machen durften, die hybrid und digital stattfinden mussten, stieg das Interesse an standalone Virtual Reality Brillen, also Brillen, die keinen Computer oder Kabel brauchen, um genutzt zu werden, stark an. Um eine virtuelle Kopie eines Konzertes oder anderer Bühne darzustellen, ist Event-Licht Pflicht. Diese Arbeit erforscht die Hürden und Probleme, auf die bei der Erstellung virtueller Klone von Effektscheinwerfern, die auf den Virtual Reality Brillen angezeigt werden können, gestoßen wurde. Die Forschungsfrage **”Wie kann eine immersive Lichtvisualisierung, die auf leistungsarmen standalone Virtual Reality Brillen läuft, erstellt werden?**

Indem Prototypen entwickelt und weiterentwickelt werden, wird ein Asset entwickelt, welches GDTF Dateien lesen und 3D Modele, die Lichtkegel und deren Auftreffpunkte mitsamt Gobo-Projektionen visualisieren kann. Diese virtuellen Scheinwerfer können dann auch mit realen Steuerkonsolen gesteuert werden.

Da während der Entwicklung einige unerwartet große Hürden entdeckt wurden, kann die Forschungsfrage nur zum Teil beantwortet werden und die nächsten Schritte, die zu erforschen sind, werden herausgestellt. Dennoch wird auch dargestellt, wie das Asset in einem realen Projekt bereits genutzt wird.

Abstract

Meta pushing their vision for the “Metaverse” and following the 2019 COVID pandemic, where digital events were suddenly a common experience for many, the interest in standalone virtual reality headsets - a kind of virtual reality headset where no auxiliary device is needed, rose drastically. To display a virtual clone of a concert or other stage, stage lighting is essential. This thesis explores the issues arising when trying to create a visualization of real stage fixtures on these standalone virtual reality headsets. The question: **“How can an immersive light-visualizer experience, running on low-power virtual reality devices be created?”** will be evaluated.

Through prototyping, an asset is developed that can import GDTF files, and display 3D models, beams, and gobo projections which can be controlled by real-world hardware.

Discovering some major hurdles in researching and implementing, the question can only be answered with the next steps to further research. Nonetheless, real-world usage is also documented.

Contents

List of Figures

List of Tables

List of Codeblocks

1 Introduction

1.1 Research Topic

Programs to create and display a virtual stage are called light visualizers. Light visualizers are heavily used for pre-production and pre-visualization of stages, performances, and architectural spaces and are often expensive and proprietary, preventing access for the general public (W. Robbins, 2014).
While some would say VR has arrived in consumers' homes and the first sectors of the industry (Gilbert, 2023), light visualizers are still mostly 2D. Leon Bantin, in his thesis "Realisierung eines Lighting Visualizers unter Verwendung der 3D-Engine Unity" already showed the feasibility and outlined the usefulness of a VR visualizer. (Bantin, 2021)
The global pandemic, which started in 2019, forced the world to isolate. The world reacted by making significant steps towards digitalization.
This led artists to explore more digital mediums, including virtual concerts with support for virtual reality headsets, which have been growing in popularity in consumer households since 2018 (XRToday, 2023,UTA, 2021a).
Game engines, which are used to display these virtual concerts, opened the possibility of not only experiencing concerts on a computer, phone, or TV but also in virtual reality. Users now could move around while experiencing a virtual concert.

The virtual world not only simulates the real world but also expands on it: Virtual concerts can accommodate many more viewers than would fit into a physical space. This was only possible before using video and live streaming. (UTA, 2021b). Additionally, floating islands, user-controlled fireworks, and other features are only a few examples that make these concerts unique and interactable.

1.2 Motivation

Seeing the void between real stage hardware and consumer devices, we founded a team that creates the "XRevent Creator", a web-based, virtual reality-compatible solution for artists to create their own virtual concerts, stages, and exhibitions.

The development of a light visualizer inside Unity Engine began in 2019 when, as part of a course at the HAW Hamburg, software to create virtual events was developed and tested. In the time since, we made further contributions to the created software, founded the team committed to developing the software, and rewrote almost the complete stage-fixture solution in working on this thesis. Designing for VR has a host of challenges attached to it, not only in game design (Moore, 2022), but also in the requirements in performance. Not every guest of a virtual concert can have a powerful computer. Standalone virtual reality headsets, like the Meta Quest, lower the barrier to entry but also provide much lower performance (Unity Technologies, 2022b).

1.3 Goal

This thesis aims to answer the question "How can an immersive light-visualizer experience, running on low-power virtual reality devices be created?". Breaking down the question further leads to the following hypothesis and questions:
"How can Unity Engine obtain specifications and models of stage fixtures?" "What are the rendering limits and bottlenecks on standalone virtual reality headsets? How can these be subverted?"

While this thesis is also a part of the documentation, it aims to explain the reasoning behind the decision-making in developing the asset. The code and assets that are allowed to be publicly available, as well as technical documentation, can be found here: https://github.com/DarkTJ/Unity-GDTF.

1.4 Research Methods

Our main methods of research were: Examination of literature and online sources, including open source projects, and practical experimentation by prototyping inside the Unity Engine, supported by empiric performance measurements.

Early examination of the literature quickly revealed a gap between the features and functions Unity Engine provides and the documentation of those features. Especially newer features often either lacked usage and documentation for all hardware or had no documentation at all. Using open-source projects played a vital role in analyzing the functionality and usage of many features described in this thesis. Details and examples are described in section 3.1 on page 14.

Experimenting inside the Unity Engine by prototyping allows for quick assurance of the discovered, researched, and developed ideas. Because standalone headsets use a new and different architecture one can't rely on documentation alone (See chapter 3). Prototyping, as described in chapter 3.2, made rapid progress possible.

1.5 Structure

The thesis is structured into 6 chapters. The following chapter explains the essential points and basics to understand further topics. In Chapter 3, the design and greater structure of the asset and tests are described, while Chapter 4 follows by describing the implementation of those ideas, detailing arising problems and tested solutions. The results of the tests described in Chapter 3 are evaluated in Chapter 5. Here the insights are first plainly described, and in later parts interpreted in the context of each other. In chapter 6 a conclusion is formed and future works and possibilities are described.

2 Background

2.1 Terms and Abbreviations

Unity Engine

Unity is a powerful and widely used game development engine that provides a comprehensive set of tools for creating interactive experiences, ranging from video games to architectural visualizations and simulations. It is known for its user-friendly interface and flexibility, making it accessible to both beginners and experienced developers. It also supports almost all modern devices that have displays. Updates with new features and improvements are released regularly and, depending on the impact of a change, can alter how certain code executes. In the production of a game or other software, the versions of all the software dependencies are often "locked", meaning that all developers work with the same, fixed version. Often chosen, as they are created for this exact reason, are LTS-Versions, which are versions that only get minor updates to fix bugs. The tests contained in this thesis were run on Version 2023.1.8f1. While in development the project updated from 2022.3.4f1 LTS to 2022.3.7f1 LTS and finally to 2023.1.8f1.

Mesh

The shape of a 3D object is described with vertices. These vertices are connected with edges to form a mesh. While almost all Polygones are possible, Unity Engine uses primarily triangles.

Shader Language

The High-Level Shader Language (HLSL) is used to write shaders. Shaders are comprised of code that is run on the Graphics Processing Unit (GPU) and mostly used to calculate the image displayed to the user. (Unity Technologies, 2023a; Unity Technologies, 2019a). HLSL can also be used to calculate non-graphics related tasks, referred to as a "Compute-Shader" (Unity Technologies, 2021c). A detailed description of how a shader is programmed is done in Chapter 4.3.4.

Shader Graph

The Shader Graph is a tool that Unity Technologies first introduced to Unity Engine in 2018 and officially implemented into the core engine technology in 2019 (Unity Technologies, 2022c. Instead of writing code to define shaders, a developer or artist can use nodes to create a shader. The nodes are compiled to typical shader code afterward Lever, 2022.

Light inside Game Engines

To create the simplest form of a Scene one just needs a mesh, a Light, and a Shader to calculate the Impact of the Light on the mesh. Multiple approaches to calculating light exist: Phong Shading is a very simple algorithm and a slightly modified version is called "SimpleLit" inside Unity Engine (Greule, 2021, p. 359; Unity Technologies, 2022e). More complex shaders use values for roughness, bumps, metallic, and more to calculate realistic looking shader in real time (Unity Technologies, 2022d). Also used are raytracing calculations, essentially simulating each ray a light-source emits and bouncing it off surfaces. This technique is mostly used for non-realtime renders and only the most powerful hardware can run real-time raytracing calculations (Unity Technologies, 2023c; Greule, 2021, p. 361).

Deffered vs Forward Rendering

Deferred and *Forward* are two rendering methods Unity Engines support. They differentiate in how each light source is calculated and how the amount of light sources influences performance. Bantin made a clear case to use the deferred renderer (Bantin, 2021, pp. 7–8) but his argumentation cannot be transferred to this work as his work focused on a computer-based VR system using many of the built-in Unity lights. We chose not to use the deferred renderer as the fake beams implemented in Chapter 3.5.2/4.3 are transparent objects and the documentation for these assets discourages the deferred renderer (Bantin, 2021; Dasch, 2019; Ferreira, 2019; Unity Technologies, n.d.).

Light Visualizer

A light visualizer is a piece of software that, using light simulation, shows the user a stage, architectural structure, or other scene and calculates the influence of light on and in that scene. Almost all light visualizers are limited to a 2D render, displayed on a normal computer screen. Some are specifically made to pre-visualize a stage for the light artist to create his light show beforehand, allowing input of control data from purpose-built light control consoles. The existing solutions in the live entertainment industry are a mix of hardware and software

platforms that are scattered, proprietary, and expensive. The tools and production pipelines are convoluted, difficult to use, and often impair the overall creation and production processes. Unreal Engine, Unity Engines' biggest competitor, added a "DMX-Plugin" in 2019 with the reasoning: *"Lately, there has been increasing interest and demand for Unreal in the context of live events and permanent digital installations throughout the world." [...].* This Feature sadly has not been developed further and is still missing substantial features (Epic Games, 2022).

GDTF File Format

There are many file formats available to exchange information about lighting fixtures. Like the existing stage visualizers, these are often proprietary to their respective software/hardware and developers, such that fixture manufacturers are required to put in a lot of work to support their fixtures on different platforms. The General Device Type Format (GDTF) aims to unify. Using GDTF, it is not only possible to describe and exchange data about lighting fixtures, but also other devices used in stage productions. The format was developed by the industry-leading companies MA Lighting and Robe Lighting, to provide a standardized way to communicate information about lighting fixtures and is specified with a DIN SPEC (VPLT, 11).

Virtual Reality

Milgram places Virtual Reality (VR) on one of the far ends of the "Real-Virtuality Continuum", opposite of the real environment (Milgram and Kishino, 1994). VR is mostly used in this thesis in conjunction with VR headsets, wearable devices including a screen (also called Head Mounted Displays (HMD)). The user gets a virtual world displayed in front of his eyes and can move in the real world to also move in the virtual world. Because of this, it is still considered "Mixed Reality" by some research as the Users still interact with the real environment (Skarbez et al., 2021). In a commercial context "mixed reality" and "augmented reality" are almost always used synonymously, placed more towards the real environment in Milgrams Continuum (Milgram and Kishino, 1994). This thesis focuses on VR and uses the Meta Quest and Quest 2 which are both mainly VR devices, subsequently called "VR headset". Both are standalone headsets, meaning all hardware needed to power the devices is packed inside it. No external cable to a computer is needed, providing a large amount of freedom and accessibility, but also limiting processing power similar to that of a modern smartphone.

Single-Pass versus Multipass Stereo Rendering

Because VR headsets have two screens, one for each eye, they have to render each frame twice from two slightly different points of view (POV), called stereo rendering. The default stereo rendering mode is called Multipass, rendering each POV independently from the other. As they are rendering mostly the same objects the Single-Pass rendering mode was introduced: Instead of loading all data of one object to the GPU, rendering the object for one POV, discarding the data, only to load it again for the other POV later, the data is used to render the object for both POVs, saving to load it multiple times (Unity Technologies, 2022j; Ferreira, 2019). This is called Instancing, picked up upon in chapter 4.3

Tile-Based Rendering

In contrast to modern GPUs using Immediate rendering, low-power devices such as the Oculus Quest use a tile-based GPU (Ferreira, 2019). By splitting a frame into smaller tiles, the amount of memory and bandwidth needed can be significantly reduced. This also reduces the power requirements (Molnar, 1994; Dasch, 2019). For developers, a host of challenges are connected to this type of hardware, especially when creating complex graphics and shaders. Working on this thesis revealed that a lot of literature and guides are only considering computer-based hardware using Immediate rendering (see chapter 3.1 and chapter 5.1.1).

Art-Net & DMX

DMX512-A (DMX) is a digital control protocol used to control fixtures on stages. Sending up to 30 packets per second containing 512 bytes over XLR Cables (on Universe). Art-Net expands on top of DMX, using Ethernet cables and infrastructure to send up to 32768 Universes of DMX data.

Unity Performance Analyser

The Unity Performance Analyzer is a toolbox made available by Unity Engine to analyze and debug games made with Unity Engine. While it can collect a high amount of data when the game is run on a computer it falls short when analyzing games running external hardware (VR headsets for example).

OVR Metrics Tool

The OVR Metrics tool is an important instrument developed for virtual reality on the Quest and Quest 2. The tool is provided by Meta/Oculus, allowing the capture of metrics directly from the hardware, ranging from frame rates and rendering latency to more advanced metrics like "Average Vertices Per Frame" or "Average Instructions per Fragment". These metrics can be used to gain insights into the connections between hardware, displayed graphics, and code. While the data can be shown to the user as a graph inside the VR headset, it is also possible to collect and log data in the CSV format (Meta, 2019).

Time.deltaTime

The function `Time.deltaTime` returns "The interval in seconds from the last frame to the current one" (Unity Technologies, 2019b). This is the most important statistic for performance analyses. When the intervals are smaller, the performance is better. For reference values see Table 2.1.

deltaTime	FPS	Notes
33 ms	30 fps	Youtube and other Online Videos, US-TV Standard (Verttermann, 2021)
16 ms	60 fps	Minimum Frame Time for VR (Meta, 2022a)
13-11 ms	75-90 fps	Targeted Frame Time for VR (Meta, 2022a)

Table 2.1: Reference Values for Frame-Times

2.2 State of the Art Technology

Figure 2.1 shows a selection of virtual concerts categorized and arranged in relation to their **Accessibility/Performance** (*Is special hardware needed?*), their **Immersion/Interactibility** (*Can the user influence others or the performance?*), and their **usage/emulation** of real stage hardware (*Can real stage-hardware be used to control the experience? Keyword: Digital Twin*). Four categories are additionally shown through coloring, explored in the following sections.

2.2.1 The Virtual Concert

As consumer devices were getting more powerful and game engines more efficient, access to virtual concerts was made possible for the everyday user. Best known and setting a world

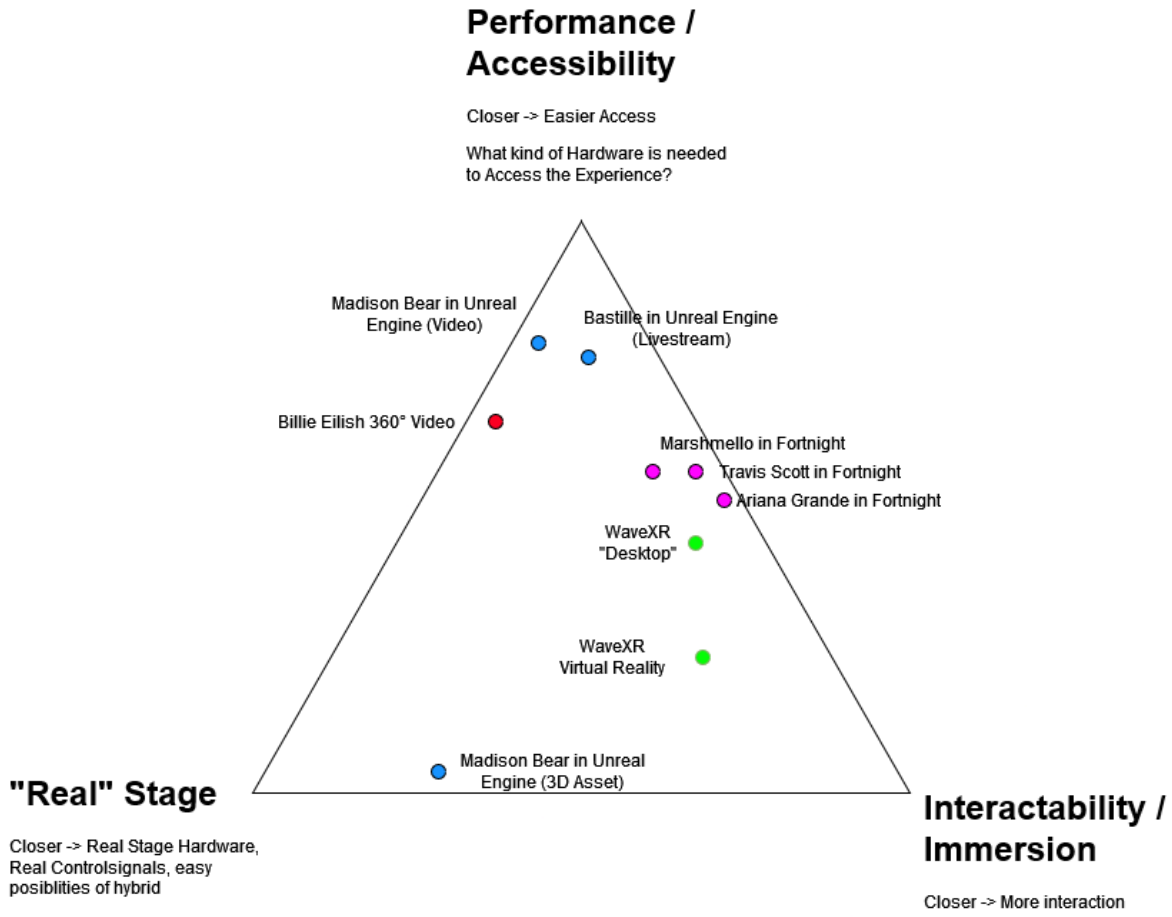

Figure 2.1: A Diagram placing a selection of Virtual Concerts in 3 Categories.

Figure 2.2: Marshmellow Concert inside Fortnite

Figure 2.3: Ariana Grande Event inside Fortnite

record for the "Largest music concert in a videogame" (GuinnessWorldRecords, 2023) are the virtual concerts hosted inside Fortnite. These concerts balance in the middle between interactability and accessibility, they are grouped with the *Pink markers*. Needing to download software onto a computer or mobile device capable of running said software represents only a small barrier to entry. The first concerts still showed a more classical stage (Fig.: 2.2). Later events were more abstract experiences, with artist avatars being scaled to giant size and visitors floating on an invisible floor (Fig.: 2.3).

2.2.2 The Virtual Concert "Experience"

Unreal Engine, the game engine Fortnite is based on, has since made big investments in creating possibilities for large-scale, realistic concerts (XRToday, 2022). They called the events they created "Virtual Concert Experiences" (Epic Games, 2023b). This phrasing has been used for two kinds of Events. **Blue markers** are 2D Videos of digitally created, digitally enhanced concerts, or 360° Videos (XRToday, 2022). The Madison Beer "Experience" (Fig.: 2.4) for example has been released as a simple Video, rendered from a 100% virtual stage. Everything (even the artist herself) was created in 3D inside Unreal Engine.

Figure 2.4: Madison Beer Virtual Concert

Figure 2.5: Secret Sky Music Festival

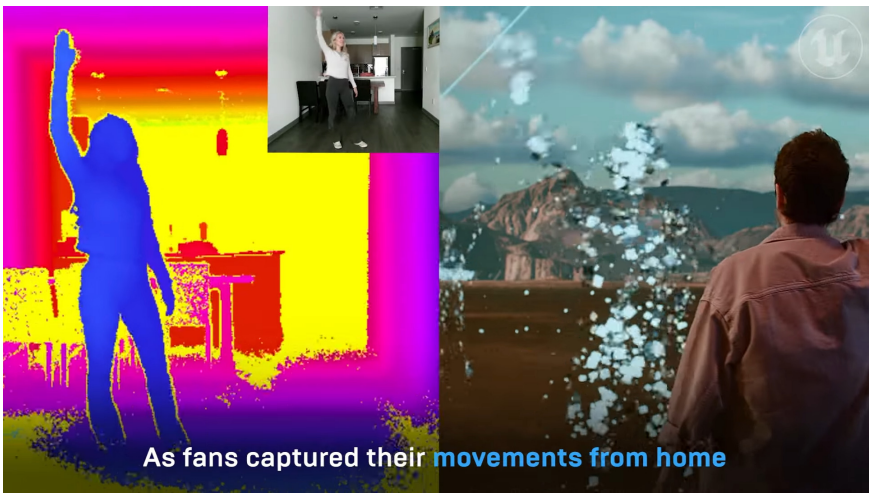

Figure 2.6: To heighten the interactivity even more, the Bastille Virtual "Experience" used the webcam image of people at home to capture their movement and place them into the virtual world (Source: XRToday, 2022).

Green markers list the events that had a higher barrier of entry by needing special hardware. Using VR headsets the visitors of such events were able to interact and immerse themself in the experience. From simple events, like the "Secret Sky Music Festival" (Fig.: 2.5), where the artists were just projected into the virtual world as video, to multiple artists being fully 3D Scanned and animated inside WaveXR (WaveXR, 2022; Fig. 2.6), these Events enabled other forms of experiencing a virtual event than sitting in front of a simple screen.

2.2.3 The Stage Visualizer

Orange markers group together the classic and longest-standing visualization of stages with new approaches. As mentioned in the Introduction these stage visualizers are used for the pre-production and pre-visualization of stages, performances, and architectural spaces. These visualizers include access to databases with real light fixtures as a standard feature as they recreate real stage setups (GmbH, 2023). Additionally, the Unreal Engine can be used to create data for screens and panels on a stage the same way a Media Server would (deadmau5.com, 2021; Greule, 2021). To achieve this, a powerful computer is needed as well as experience with products alike.

In the diagram there is a distinct space between the realistic and the easily accessible experiences. Not much focus has been put into developing assets that enable small artists to work with these new technologies without specialized hardware and software.

Game engines have enabled small game studios and sometimes even sole artists to create

popular and large-scale games that would have taken years to develop with large teams just a decade ago (Bruce, 2022). Game engines, in contrast to the more classic light visualizers, are widely supported, free, and easy to get started with (Noveltech, 2022). Unreal Engine itself created an implementation for the GDTF files (GDTF, 2022; Epic Games, 2023a), but sadly leaving the complex interpretation of the data, creation of the fixture in 3D space, and rendering of beams and special features to the user (Unity Technologies, 2021a). Similarly, multiple user-created assets for the Unity engine exist: (Fok, 2020; igolinin, 2021; Cho, 2018). All of these solutions are made to use the integrated real-time light solutions of the Unity Engine and are not suitable to be run on standalone VR headsets (see 3.5.2), thus leading to the creation of this thesis.

2.3 Scope

While developing the asset to answer the thesis's main question, a lot of supporting work had to be done that is not covered in this thesis. Additionally, some groundbreaking hardware and software limitations were discovered, using up a high percentage of development time. Chapter 6.2 lists many overlooked features, where optimizations would easily be possible but were skipped in favor of focusing on the following topics:

- Rendering of beam geometry: Making a light beam visible in the air using rendering methods that are compatible and performant on standalone, android-based virtual reality headsets.
- Gobo projections: Finding a method to render projections that is compatible and performant, as the removal of the original lights leads to the new beam geometry not shining onto surfaces.

3 Research Design

3.1 Literature research

Books and Official Documentation

The Quest 2, being released in 2020, is a relatively new device. The idea of using mobile hardware architecture for virtual reality, freeing the user of cables and the need to own a powerful computer, is not much older than the Quest itself (Melling, 2018). Game engines, like Unity, are still being optimized for these new devices(arm.com, 2021). Using the most recent version of Unity can be a viable approach to enhance performance (Unity Technologies, 2022a). On the other hand, working with "cutting-edge" products and software comes with limited availability of relevant books and similar literature. In some cases, even the official documentation is found lacking, thus increasing the difficulty of conducting research. Only a handful of books have been released recently covering the newest additions: "Building Quality Shaders for Unity" by Iett, 2022, is a great example of such a book. Being released in 2022 it comprehensively explains and showcases additions to Unity Engine and Shader languages, not found in most other literature. It is by far the most recent book written by a third party on this subject. While it only covers the Shader part of Unity Engine, it holds almost no information on working with shaders in connection to the Quest and Quest 2 headsets. Similarly to other sources, it also contains contradicting information on details about these devices and their (rendering) performance.

An example of this are the recommendations for MSAA on Meta Quest devices: The official documentation by Meta states: "Using MSAA on a PC-based platform comes at a lower cost than using the technique on Meta Quest. [...] It is better to spend GPU time on higher resolution render targets than further increase MSAA." (Meta, 2022b), while a blog post by Meta states almost the exact opposite: "There is extra overhead for 2x/4x MSAA, but on mobile the computational cost is nowhere near as substantial as on PC, relatively. [...] It's almost always preferred to use some level of MSAA rather than keeping your resolution (render-scale) at native." (Ferreira, 2019).

Third Party Tutorials

Tutorials created by third-party authors are a reasonable source of inspiration and examples. On one hand, there is caution to be taken as these are often biased by the background of the author and often do not cite any literature. On the other hand, the authors' diverse backgrounds often lead to information that is hard to find otherwise.
Testing all findings helps relinquish uncertainties.

Open Source Projects

Open Source Projects are projects and programs where the source code can be freely accessed by everyone. While these projects can also be an enormous source of insight, they come with similar tradeoffs as the tutorials. Authors with diverse backgrounds create software and prototypes and share the code they have written. With an open source model, even projects from authors that do not write or speak a common language can benefit from each other as the code is not localized to any language. While comments and documentation can be a hurdle, most programmers use coding conventions (Microsoft, 2022), making anyone able to read the code and make relatively accurate assumptions of its functionality. Two open source projects cited in this thesis by Cho, 2018 and Fok, 2020 are almost entirely commented and described in Japanese and Chinese respectively, only the latter having an English readme file. Again, testing all findings helps relinquish uncertainties.

Paid Assets

In contrast to almost all commercially available programs, assets bought in the Unity Asset Store (accessible at `https://assetstore.unity.com/`) are delivered as uncompiled source code (Unity Technologies, 2023d, 1.5.a). They can be treated almost like open source projects, having major benefits and drawbacks. On one hand, they are strictly regulated to adhere to programming and structuring standards and **must** be documented in English (Unity Technologies, 2023d, pp. 1.1.i, 2.1–2.7). Authors are driven to optimization, improvement, and expansion by the revenue they generate with such assets, creating advanced and well-running code. On the other hand, as the code is bought, its usage often is restricted. The "Asset Store End User License Agreement" (Asset Store EULA) (Unity Technologies, 2023b) regulates the usage of such assets. A fitting explanation on the Unity Engine blog states: "For example, if you think of the Asset Store as a grocery store and the assets as carrots and zucchini, the Asset Store EULA lets you sell a meal that you make from the carrots and zucchini. Your recipe, your preparation, and your presentation are the substantial original work with which vegetables are distributed. The Asset Store EULA, however, does not let you resell individual

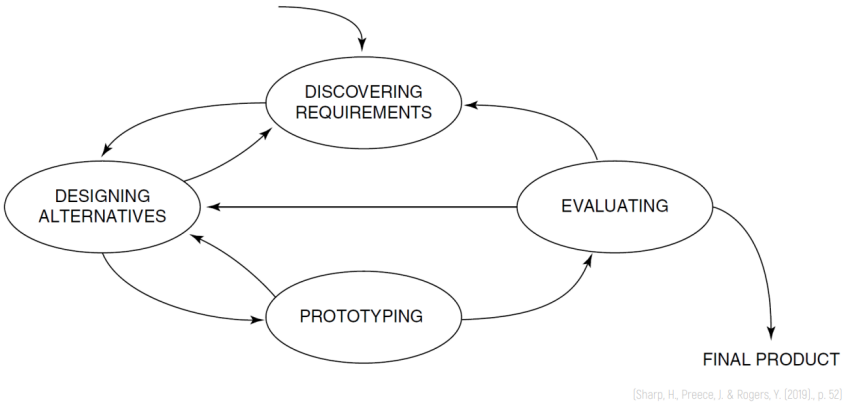

Figure 3.1: Prototyping Flowchart

vegetables outside of a recipe." (Maxine, 2023). Some assets are classified as "Restricted assets", further limiting their usage (Unity Technologies, 2023b, §3.2).

Example Scenes

Example scenes are included in many open source and paid assets or are available to download with the documentation. They are often added to asset or feature releases and are created by the authors of the respective resource. They show best practices and recommended uses of the assets they accompany. They can be used similarly to open source projects, gathering practices and inspiration from other people's works.

3.2 Protoyping

A proven and reliable method of creating software is by prototyping (TenMedia, 2022). Prototyping is especially useful when the path to the goal or the goal itself is not clearly defined, allowing the research to be focused on arising issues and adapting to hurdles and blockages (Joshi, 2021, p. 25). Fig. 3.1 (Preece et al., 2015, p. 52) shows the steps a prototyping approach takes. It is important to preface that the prototyping approach was mainly applied to "Modules" of the asset that was developed in this thesis (3.5 Modules) rather than the whole asset. While the main goal of the asset: "Running an immersive light-visualizer experience on low power virtual reality devices" (1.3) was not changed, the software and tests themselves were adapted and reframed, and the thesis' main goal was also iterated on and refined in rounds of prototypes, as solutions were found and larger hurdles were discovered.

3.3 Tests

Quickly realizing that performance would be a major bottleneck in the development of this asset, performance tests were introduced. Not only providing an answer to the main thesis but the tests could also be used on every change to relinquish uncertainties in research sources and code implemented.

Tests are run by collecting important performance data while the device executes the software and automatically plays through scenes, each with a different focus. Three scenes were created:

- 1: 1000 Lamp Scene
- 2: Concert Scene
- 3: Exhibition Scene

Scene 1 focuses on as many fixtures as possible, measuring even small impacts on performance as the effect is multiplied by all the fixtures in the scene. Scene 2 emulates a medium-sized concert setup. Here we can test our performance in a "real" setup. Scene 3 is a "high fidelity" scene. Here not the amount of lights, but the look and feel (immersion) is important. This scene displays a small exhibition with different items interacting with light.

A conscious decision was made to not collect and record a lot of information. Memory, GPU, and CPU data, conventionally also logged by performance recording tools (Pluberus, 2013), could always be seen live in the OVR Metrics Tool-Tool or the Unity Performance Analyser (see chapter 2.1). Having three defined scenes with no camera movement and perfectly repeatable movement and animation of the lights allowed us to use the timeline (see Fig. 4.1) to pinpoint when certain points in the scene would have vastly different performances. The scene could be analyzed by replaying it exactly and pausing at the right moment, providing all needed data in much higher detail than a recorded "packet" could.

Three approaches were taken in the following sequence, each learning from the previous, as defined in 3.2:

- Collecting data every frame
- Using the OVR Metrics Tool
- Collecting data in 1-second packets

The recorded data was saved and a script was created to display the data as graphs. The graph-generating script also went through multiple prototyping steps which are omitted from this work to keep in scope. The script is available in appendix 3.

3.4 Asset

Creating an asset has a main advantage: It can be developed "standalone" but later be used as a very distinguishable part (module) of a larger program (see 6.3 Real world usage). The research question implies the following requirements:

- "How can an immersive light-visualizer experience, running on low-power virtual reality devices be created?"
  - Software displaying light fixtures
  - Software running on low-power virtual reality devices with, at least, the recommended framerate (2.1)

Further breaking down the points leads to the following requirements and questions already named in 1.3:

- Software that can read files with fixture information.
  - "How can Unity Engine get specifications and models of stage fixtures?"
- Software that simulates a light beam.
- Software that simulates light and Gobo projections.
- Software that can show a 3D Model of the fixtures.
- Software that can receive control signals for fixures.

All previous questions and requirements are to be combined with: **"What are rendering limits and bottlenecks on standalone virtual reality headsets? How can these be subverted?"**, leading to five more or less separated "modules", each running through prototyping and testing. Inspired by unit testing, the modules can be developed mostly independent of each other, allowing faster development and more accurate testing (Khorikov, 2020, p. 150).

3.5 Modules

This Section is split into five subsections, reflecting the list from 3.4. A more detailed list, including proposed module implementations, is added to the end of this thesis.
While this Chapter defines the goals of each module and lists the prototyping and testing steps, each subsection is picked up upon in chapter 4, explaining the details behind each prototyping cycle and showcasing the problems and solutions. Each of the following sections

starts with one of the requirements from the list above in *italic*, stating the goal of the described module.

3.5.1 GDTF Import

"*Software that can read files with fixture information*".

Using the DIN SPEC 15800 (Staffel et al., 2022) as a guideline on how digital twins of real-world fixtures are defined, the asset will be "ingesting" a .gdtf file and creating a 3D representation of said fixture. The software should display the 3D models of the fixtures, and implement further functions e.g. shutter, focus, Gobo-projections, and strobe. The official document of the DIN SPEC provides a well-written structure of a fixture, especially showing how functions are related to each other. The loading of the file is split into three separate tasks:

- 1. Finding and opening the file
- 2. Reading the XML file
- 3. Reading extra files (images and models)

Point 1: Finding and opening the file.
Supplying the file is touched upon in Chapter Real world usage but disregarded here. It is assumed the program has permanent access to the fixture files.

Point 2: Reading the XML file.
As per specification, the XML file holds all technical information of the fixture (Staffel et al., 2022). The structure is defined in a .xsd File (XML Schema file) that is made available by the developers of the specification. Three ways of reading the file were identified:

- Reading selected parameters when needed
- Reading selected parameters into an object
- Reading all parameters into an object

As the reading is only done when the scene is loaded the impact of this process on the app's performance is hard to gauge. Loading times are not in the scope of this work and while one method could use more memory than another, the impact was regarded as miniscule and no tests for this module were designed.

Point 3: Reading extra files (images and models). As the files are used for other modules, reading them is picked up in the following subsections.

3.5.2 Light Beams

"*Software that simulates a light beam*"

Creating a light beam is one of the most important features of the visualizer. Almost all light shows use haze or fog to make the beams of the fixtures visible in the air (Schiavone, 2023). All referenced solutions (Bantin, 2021; Fok, 2020; Cho, 2018) use the build-in Unity lights, which are very limited on standalone virtual reality headsets: Unity Engine on mobile and other low-power devices only supports 4-8 real-time lights at once (Unity Technologies, n.d.). Meta even recommends limiting the usage of real-time lights to **1** per scene (Meta, 2022a). As most stages use many more lights, these assets lose their ability to be run on most of the common standalone VR headsets or other low-power devices.

Bantin already described the switch to "geometry beams" in his thesis outlook (Bantin, 2021, p. 61. Geometry beams are light beams not created by light calculations but by a mesh with a shader. First, buying an asset from the asset store, later switching to a self-written Shader, many tests were run to improve the performance of said light beams:

- Switching to geometry beams
- Optimizing with batching
- Geometry detail: vertices, resolution...
- Shader quality
- Self-written shaders

While being the most impactful (see 5.1.1) this module was also the most complex and had the most setbacks (see 4.3).

3.5.3 Light Projections

"*Software that simulates light and Gobo projections*"

The projection of a circle of light is a very important feature to heighten the realism and immersion of the experience. Additionally, sometimes even more importantly, gobo projections are used to project patterns onto objects. Fig. 3.3 shows the difference a scene can have when it has no projections on the virtual scene, especially if no beams are visible (no fog/haze used). Fig. 3.2 is a typical Gobo pattern: Pictures like this one are supplied with the .gdtf files. This module had the most issues getting anything displayed. Complex

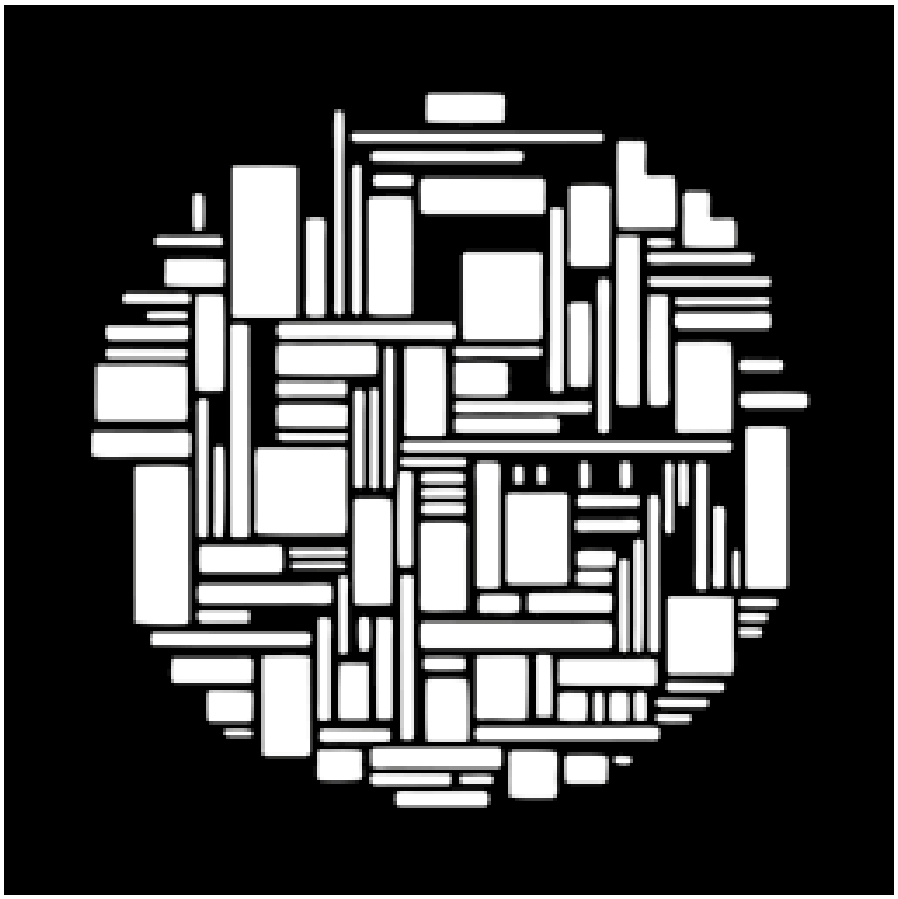

Figure 3.2: A typical file describing a Gobo

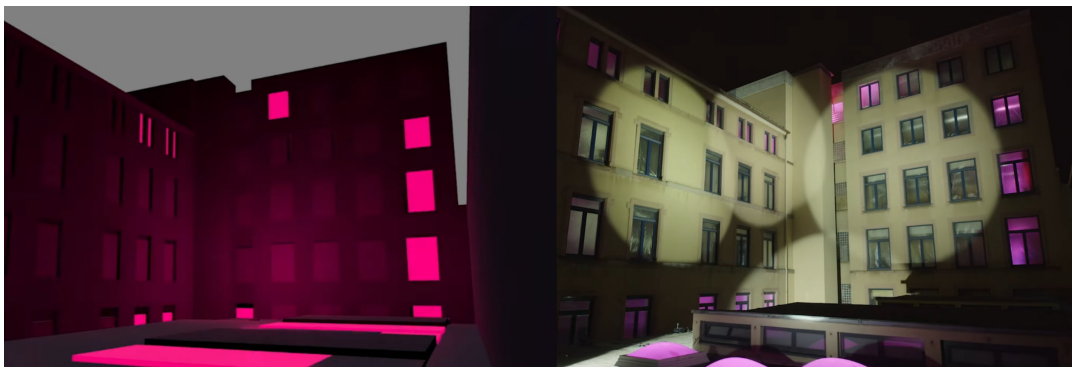

Figure 3.3: A virtual installation without projection (left) and a real installation (right)

matrix calculations and transformations including calculations based on a depth map (see chapter 4.4) are needed to simulate the projection onto surfaces. Analog to the light beams, premade solutions faced many problems, mainly because they were not developed for mobile platforms, especially virtual reality. Chapter 4.4 highlights three of the explored methods of projections and discloses their shortcomings, lastly showing the customization that had to be done to make anything visible on the VR headset:

- Camera projections
- Projection shader in ShaderGraph
- Decal shader asset and its modifications

Unexpectedly, the projection of Gobos had little impact on performance. As only the third method produced tangible and visible results on the VR headset the test is kept simple and only showcases “Gobos” vs “no Gobos”.

3.5.4 3D Models

“*Software that can show a 3D Model of the fixtures*”

This work only looks at the technical aspect of the visibility of fixtures. Discussion, whether or not fixtures ***should*** *be seen, from a Visitors point-of-view, are out of the scope of this work.*

To be able to plan a stage it can be important to display the fixtures as a 3D Model. The model can be used to position the fixture and gauge distances and spacing. The level of detail and complexity in the models affects both the immersion and the performance of the visualization. An option to decide the visibility of the 3D model of each individual fixture is given to the user, enabling fixtures far out of reach of the guests to be made invisible, while

fixtures close to the audience stay visible.
The initial import process for the models was challenging due to the use of an uncommon file format, 3Ds, which is not directly supported by Unity Engine (Unity Technologies, 2022k). Many improvement ideas were moved to the proposed module implementations to focus on the light beams, leaving only three tests.

- PerformanceImpact between:
  - Loaded (Default) Models with a self-written loader.
  - Loading Cubes.
  - Loading nothing.

3.5.5 Art-Net Control

"*Software that can receive control signals for fixures.*"

Controlling the fixtures with a protocol that is used on real stages makes the software usable with real-world show data and controllable with real light consoles. The section and module focus on all the functions needed to control a light fixture, from receiving the data to changing the fixture.

While the list of functions that a gdtf device can have is very long (see Staffel et al., 2022, pp. 54–64), the functions necessary to control most fixtures are only a small subset. Implementation of the following functions was deemed necessary for the minimum viable prototype:

- reading of 8bit and 16 bit channels
- Movement on up to two axis
- Dimmer
- Shutter + Strobo
- Zoom
- Wheels
  - Color Wheels
  - Gobo Wheels
- RGB Values

Performance impact was tested is discussed in 4.6 but deemed negligible in comparison to the beams and projections.

4 Implementation

4.1 Tests

4.1.1 Gathering Data

In the first prototype, capturing performance data inside Unity Engine was quickly made possible with a small script. The script gathers simple data for each frame and stores it in a List. The List `PerfromanceTrackerPackets` (4.1) is stored in RAM until a function is called to save it to Disk, see 4.1.3 Saving data. This function was used extensively in early research of this thesis, capturing huge amounts of Packets with the main information in each: `Time.deltaTime`.

Codeblock 4.1: Capturing a statistics-packet each Frame

```
1     void Update()
2         {
3             if(!_recording) return;
4             float time = Time.time;
5             PerfromanceTrackerPackets.Add(new PerfromanceFramePacket()
6             {
7                 SecondsSinceStart = time - _recordingStartTime,
8                 FrameSeconds = Time.deltaTime,
9             });
10
11        }
```

This performance data was never analyzed effectively and looking for more information in the hardware itself the second Prototype was created:
As the Unity Engine can only give us little information about the state of the device itself, especially Android (2.1), the OVR Metrics Tool (2.1) is used. Using the OVR Metrics Tool a line of Code can be used to append a string to the csv-file to mark certain moments, for example, the loading of a scene. This was used extensively to mark the beginning and endings of the three scenes used for testing. The Graphs used in the later stages of this chapter as

well as 5 rely heavily on these strings. The .csv-File generated by the OVR Metric Tool has, as per definition of CSV files, the heading for all columns listed:

Codeblock 4.2: All columns of the CSV-File outputted by the OVR Metrics Tool

```
Time Stamp,available_memory_MB,app_pss_MB,battery_level_percentage,
battery_temperature_celcius,battery_current_now_milliamps,
sensor_temperature_celcius,power_current,power_level_state,
power_voltage,power_wattage,cpu_level,gpu_level,cpu_frequency_MHz,
gpu_frequency_MHz,mem_frequency_MHz, minimum_vsyncs,extra_latency_mode,
average_frame_rate,display_refresh_rate,average_prediction_milliseconds,
screen_tear_count,early_frame_count,stale_frame_count,
maximum_rotational_speed_degrees_per_second,foveation_level,
eye_buffer_width,eye_buffer_height,app_gpu_time_microseconds,
timewarp_gpu_time_microseconds,guardian_gpu_time_microseconds,
cpu_utilization_percentage,cpu_utilization_percentage_core0,
cpu_utilization_percentage_core1,cpu_utilization_percentage_core2,
cpu_utilization_percentage_core3,cpu_utilization_percentage_core4,
cpu_utilization_percentage_core5,cpu_utilization_percentage_core6,
cpu_utilization_percentage_core7,gpu_utilization_percentage,
spacewarp_motion_vector_type,spacewarped_frames_per_second,app_vss_MB,
app_rss_MB,app_dalvik_pss_MB,app_private_dirty_MB,app_private_clean_MB,app_uss_MB,
stale_frames_consecutive,screen_fill_rate_left_eye,screen_fill_rate_right_eye,
screen_fill_rate,screen_red_intensity_left_eye,
screen_red_intensity_right_eye,screen_red_intensity,
screen_green_intensity_left_eye,screen_green_intensity_right_eye,
screen_green_intensity,screen_blue_intensity_left_eye,
screen_blue_intensity_right_eye,screen_blue_intensity,
screen_color_intensity_left_eye,screen_color_intensity_right_eye,
screen_color_intensity,screen_brightness,screen_dimming,screen_power_consumption,
hands_input_extra_delay,eyes_input_extra_delay,avg_vertices_per_frame,
avg_fill_percentage,avg_inst_per_frag,avg_inst_per_vert,avg_textures_per_frag,
percent_time_shading_frags,percent_time_shading_verts,percent_time_compute,
percent_vertex_fetch_stall,percent_texture_fetch_stall,percent_texture_l1_miss,
percent_texture_l2_miss,percent_texture_nearest_filtered,
percent_texture_linear_filtered,percent_texture_anisotropic_filtered,
vrshell_average_frame_rate,vrshell_gpu_time_microseconds,
vrshell_and_guardian_gpu_time_microseconds,render_scale,Column1
```

This amount of data is not only overwhelming but also not very useful. A lot of data is written into the head but not collected, as an example frame shows:

Codeblock 4.3: An example Frame-Packet outputted by the OVR Metrics Tool

```
20006,1850,386,99,23,9999,0,395,0,4198,1658,2,4,1651,414,1554,0,0,72,72,48,0,26,0,0,
0,1216,1344,9959,1182,0,19,31,31,39,40,14,13,14,19,76,0,0,10224,439,0,345,36,382,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,100
```

Easily recognizable are the large amounts of zeros at the end of the data frame, which are completely unhelpful. Already touched upon in 3.3, a second argument against the usefulness of this large amount of data is the replayability of the test used in this thesis and the ability to live-monitor with the OVR Metric Tool. Combining this, simple data packets can be saved and interesting timeframes replayed and analyzed live, leading to easier data handling and faster testing. Using this new Insight, the third large prototyping change was made and the self-written script was adapted to the style of the OVR Metrics recording, but only collecting the necessary data to pinpoint situations that should be live-monitored in greater detail (see 4.6). This change also allowed the script to be used on all devices used for testing, not only on devices compatible with the OVR Metrics Tool. Using this self-written script for both the standalone headsets and for computer-based tests made it even more feasible for direct comparisons.

Codeblock 4.6 shows the modified Code for Packet Capture. Packets now only get saved once per Second, calculating the average framerate in between. Additionally, `availableMemoryInMb` saves the total available Memory the System still has available and `Column1` saves a string that can be added from other Scripts (exatly as the OVR Metrics Tool allows).

4.1.2 Running the Tests

Explaining the exact procedure for running the test is neither necessary nor in the Scope of this thesis. Two functions inside the `PerformanceTestRunner.cs` are nonetheless explained as they are relevant to the display and evaluation of the test data. The Function `addStringToCSV(string)`, available in Codeblock 4.4, adds the supplied string to the CSV file, evaluating if the string needs to be sent to the OVR Metric Tool or the self-written `PerformanceTracker`.

Codeblock 4.4: Function addStringToCSV: Adding a Line to the CSV-files “Column1”

```
        private void addStringToCSV(string s)
    {
#if UNITY_ANDROID && !UNITY_EDITOR && DEBUGGINGUSINGOVRMETRICSTOOLSDK
        OVRMetricsToolSDK.Instance.AppendCsvDebugString(s);
#else
        _performanceTracker.AddDebugLine(s);
#endif
        Debug.Log(s);
    }
```

The Function `SceneCountdown`, available in Codeblock 4.5, is responsible for running all the defined tests. Inside its loops, each loop being run for the 3 scenes, the function uses the `addStringToCSV`-Function to add “Loading”, “Start”, “End”, and “Unloading” Tags into the recorded Data. This data is then used in 4.1.4 Displaying Data.

4.1.3 Saving data

A decision needed to be made in which format to save the recorded data: The format should be readable by common data-evaluating software like Excel, and be interpretable by languages such as Python. Zgeb, 2021 provides us with two possibilities. The first, Unity Serialization, can not be read by common programs. Researching further on the second proposed method, JSON Files, expanded our choice to two more: XML and CSV. Before even deciding on a Method to save the data, the collection of the data was moved to the second prototyping step, using the OVR Metrics Tool. The OVR Metrics Tool saves files in the aforementioned CSV files. Adopting this format in the third prototyping step made it possible to reuse the script from 4.1.4, which was originally made for the files generated by the OVR Metrics Tool.

Codeblock 4.5: Function SceneCountdown: Running the defined Tests and putting labels at the right points for evaluation

```
    IEnumerator SceneCountdown()
    {
        [....]
        if (_testMode == TestMode.RunAllTests)
        {
            for (var i = 0; i < _performanceSettings.Count; i++)
            {
                for (var o = 0; o < performanceTestScenes.scenes.Count; o++)
                {
```

```
                    addStringToCSV("Loading␣Scene:␣"
                        + performanceTestScenes.scenes[o] + "␣with:␣"
                        + performanceTestScenes._performanceSettings[i].vlb + "␣at:␣"
                        + Time.time.ToString(new CultureInfo("en-US")));
                    _currentlyRunningPerformanceSettings = i;
                    _currentlyRunningScene = o;
                    performanceTestScenes.LoadScene(o, i,SceneLoaded);
                    addStringToCSV("Countdown␣Start:␣"
                        + Time.time.ToString(new CultureInfo("en-US")));
                    yield return new WaitForSeconds(TestRunTime);
                    addStringToCSV("Countdown␣End:␣"
                        + Time.time.ToString(new CultureInfo("en-US")));
                    performanceTestScenes.UnloadScene();
                    addStringToCSV("Unloading␣Scene:␣" + performanceTestScenes.scenes[o]
                        + "␣with:␣" + performanceTestScenes._performanceSettings[i].vlb
                        + "␣at:␣" + Time.time.ToString(new CultureInfo("en-US")));
                }

            }
            Application.Quit();
        [....]
    }

```

Codeblock 4.6: Modified Function: Capturing a statistics-packet each Second

```
    private void Update()
        {
            if (_recording)
            {
                float time = Time.time;
                _frames += 1;
                if (time› _nextFrameTime)
                {
                    PerfromanceTrackerPackets.Add(new PerfromanceFramePacket()
                    {
                        TimeStamp = (int)Math.Floor(time*1000),
                        AverageFrameRate = ((time - _frameStartTime)*1000)/_frames,
                        availableMemoryInMb =
                                (float)System.GC.GetTotalMemory(false) / 1000000f,
                        Column1 = String.Empty
                    });
                    _nextFrameTime = time + 1f;
                    _frameStartTime = time;
                    _frames = 0;
                }
            }
        }
```

4.1.4 Displaying Data

Creating graphs and other visual aids helps in understanding the data. The script generating these graphs was developed using the same prototyping method as all other features of this thesis. Python was selected as the development language as it has extensive libraries suited for data visualization (Ginsberg, 2023). The script reads the CSV that was created and plots 2 Graphs. CSV´s created by the self-written script as well as the OVR Metrics Tool are compatible, although some versions lack the “Column1” addition to the header (see 2.1), a bug that has been fixed in a later version (Hdouin, 2023). The first graphs generated a simple line, plotting timestamps against the average framerate. For easy orientation, using the Strings in “Colum1”, labels attached to vertical lines were introduced, marking the starts and ends of test runs. (Added to the CSV from the “Testrunner.cs” (described in 4.1.2)). Fig. 4.1 shows the first graph generated. This Graph, while being an excellent example, has some major flaws: It only states the beginning of a Scene load but disregards the actual loading time. The 1000 Lamp Scene takes a long time to load, clearly visible by the huge dip in framerate at about

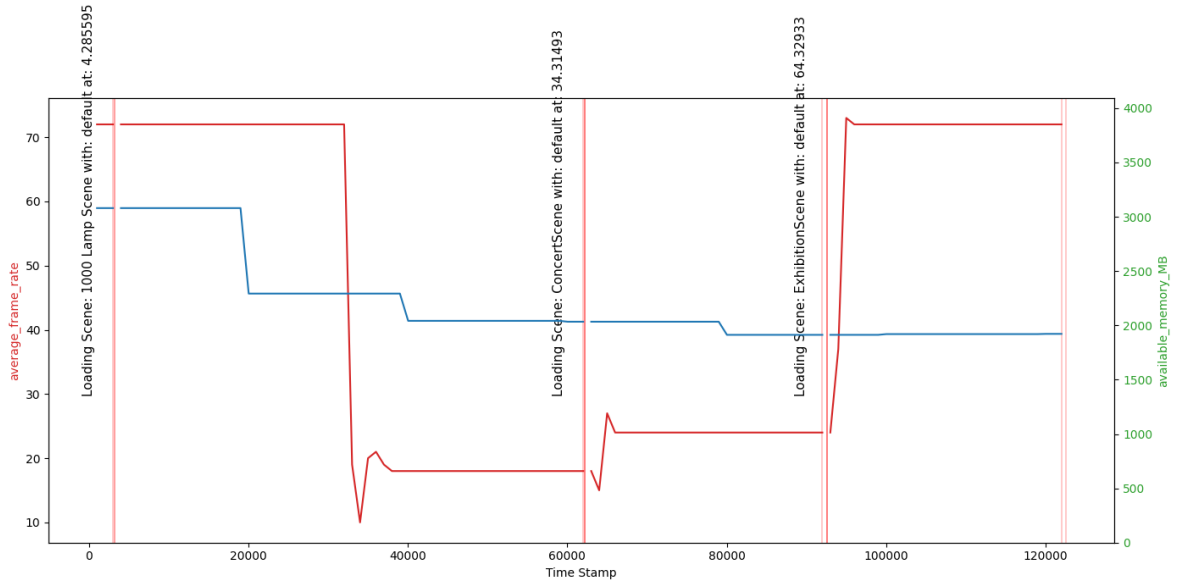

Figure 4.1: First Type of Graph, displaying the framerate in relation to time

Time Stamp 37000. In later iterations of the script, this has been addressed and all graphs used in the evaluation are using the corrected version.

Graph 4.2 shows the relation between different Test Parameters (It only shows Test Parameters "default", the average fps displayed with black outline) and the relation between the different Test Scenes (4.1.5). A bigger picture of the Impact of Parameters can be gained but also how the change in Parameters impacts the Scenes differently. This Graph is generated from the same data as 4.1, inheriting the same flaw: Especially in the first scene, the average is much higher than it should be, as the loading of the scene runs at >70 fps and is also added to the average. Two differently scaled Graphs of this kind are shown in this thesis. When the Test was run on a Quest (1 or 2) the y-axis maximum is at 100 fps. When the tests were run on a Computer the y-axis is scaled to a maximum of 370 fps. To prevent confusion the background of graphs including tests run on a computer is colored orange (like in 5.1).

4.1.5 Test Scenes

Every single one of the tests is run against the same scenes. There were 3 scenes selected as Test-Scenes, each with a different focus.

- 1000 Lamp Scene
- Concert Scene
- Exhibition Scene

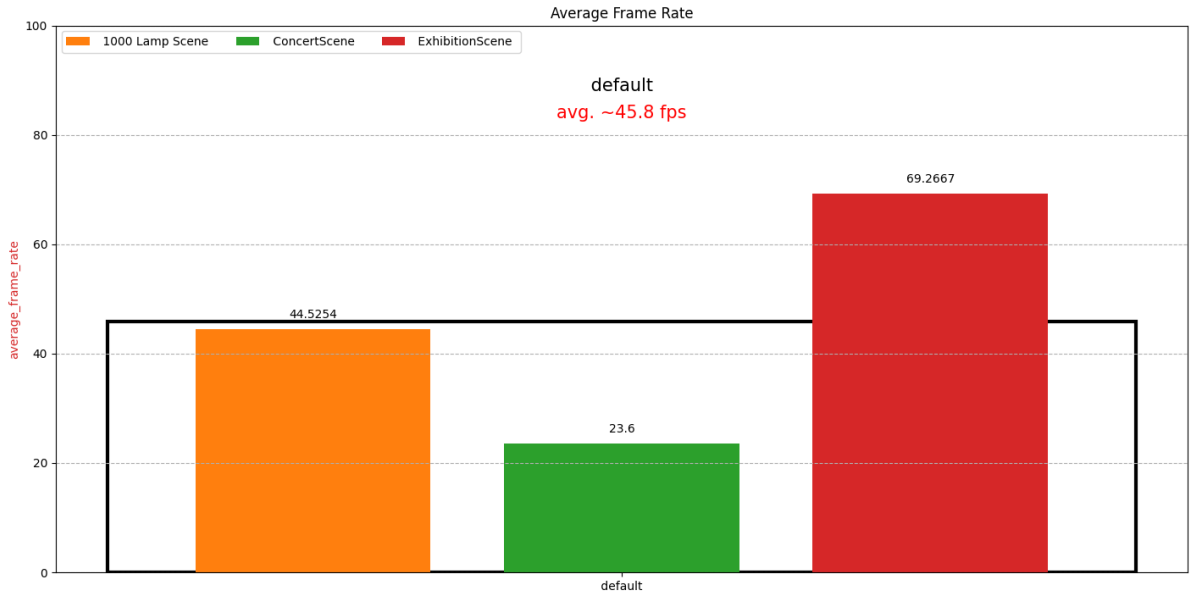

Figure 4.2: Second Type of Graph, giving an Overview of the Scenes and the Test Parameter (here only “default”)

1000 Lamp Scene

The “1000 Lamp Scene”, as the name implies, has about 1000 Lamps. About 600 fixtures have been placed in a grid. The fixtures have been carefully selected from the GDTF library to include many available functions, some being simple, non-moving Fixtures, some being large Washlights or Bars with multiple Emitters, Gobo Wheel, and much more. Table 4.1 is an Overview of the fixtures used and their functions, while Fig. 4.3 shows the Scene with all features enabled.

Amount	Fixture Manufacturer and Name	Notes
66	Arri SkyPanel S30RP	Simple Dimmer, one Channel, no Color
37	SRobe Robin LEDBeam 150 FW RGBA	Small Moving Head with LED Emitter
35	Robin Viva CMY	Large Moving Head with two Gobo Wheels and Color wheel
22	JB-Lighting Sparx 7	Simple Washlight in Mode 1 (Single Beam)
40	Robe Robin Tetra 2	Multi Emitter Bar

Table 4.1: Fixtures used in the “1000 Lamps” Scene

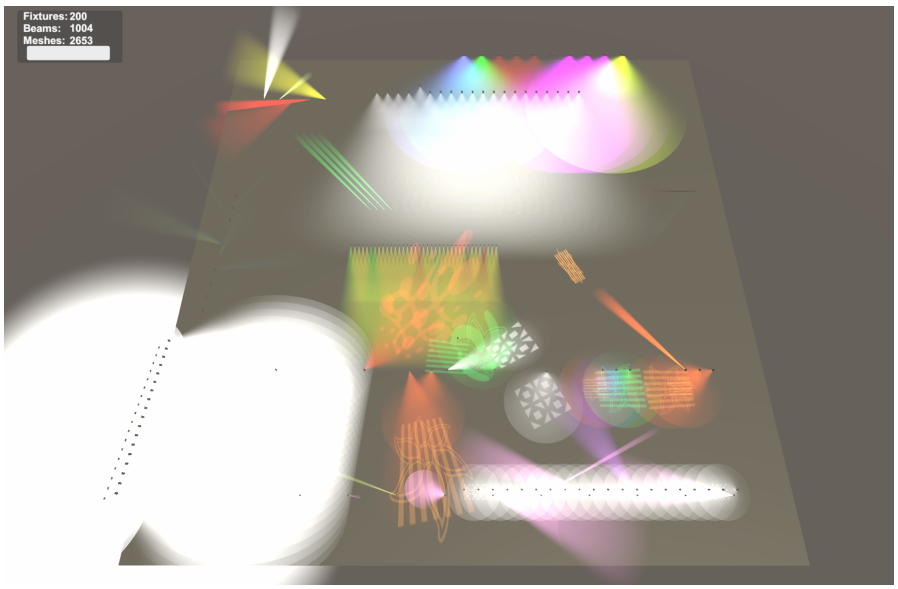

Figure 4.3: The 1000 Lamp Scene with all features enabled. In the top left statistics are displayed. (Source: Screenshot)

Concert Scene

The "Concert Scene" features different fixtures used in real production, as well as video and audio being played, (emulated) ArtnNet Input, and 3D models displaying a stage inside a stadium. See 4.4 for an overview. Table 4.2 lists the used Fixtures. The amount of fixtures might be low for a stage of this size, focusing on playability rather than realism. The Scene is the main benchmark to answer this thesis question (see 5 Evaluation).

Amount	Fixture Manufacturer and Name	Notes
10	Robe Robin LEDBeam 150 FW RGBA	Small Moving Head with LED Emitter
4	Robin Viva CMY	Large Moving Head with Gobos and Color wheel
6	Robe iSpiider	Large WashLight with 19 Emitters

Table 4.2: Fixtures used in the "Concert Scene"

Exhibition

In the "Exhibition Scene" two Robe Viva CMY, one placed on the floor in view of the Camera and one placed outside of view, are the only fixtures imported. Both fixtures are set up to continuously rotate, projecting a gobo onto the floor and wall, and sometimes directly into the camera. A bigger amount of geometry is placed inside the Scene as well using both opaque

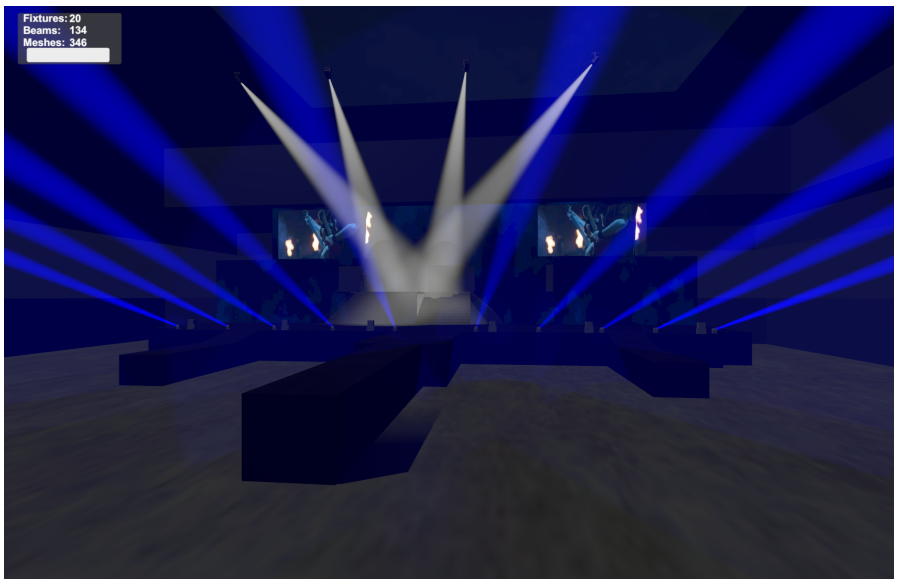

Figure 4.4: The Concert Scene with all features enabled. In the top left statistics are displayed. (Source: Screenshot)

and transparent shaders, showing the interaction of the fixtures with their environment. This scene also suffers most from changes in the detail of the light beam as both fixtures are quite close to the viewer. Fig. 4.5 is a screenshot of the scene.

4.2 GDTF Import

The .gdtf files are defined by DIN Spec. 15800, making it an exceptionally well-documented part of the research for this thesis.

4.2.1 Finding and opening the file

Expecting to be supplied with the location of the GDTF file, the file is just a .zip archive without the file extension (Staffel et al., 2022, p. 8). Using the "zip" library included in .Net the file can be opened and all subsequent files read (Microsoft, 2011). In different scenarios, the file may need to be loaded from an external Server (see 6.2).

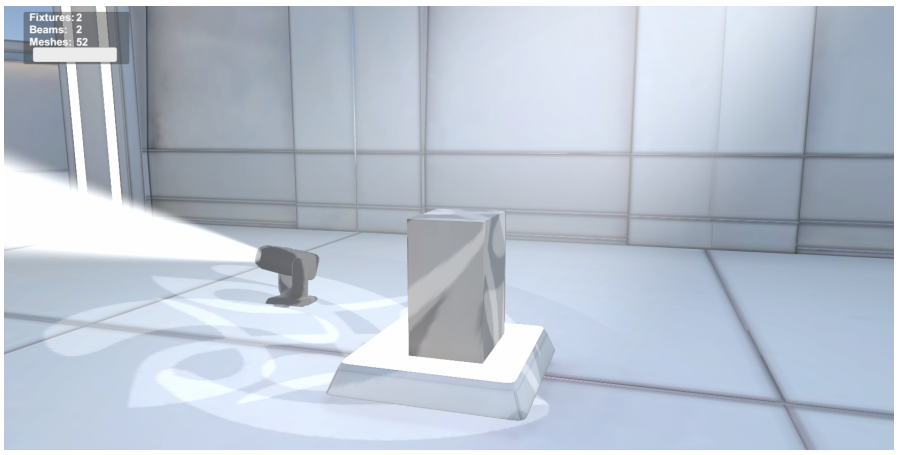

Figure 4.5: The Exhibition Scene with all features enabled. In the top left statistics are displayed. (Source: Screenshot)

4.2.2 Reading the XML file

Almost all information about the fixture is stored in the description.xml file inside the archive, described in the DIN Spec (Staffel et al., 2022, pp. 8–53, 65–90). Three methods of reading the XML were evaluated:

- Reading selected parameters when needed
- Reading selected parameters into an object
- Reading all Parameters into an object

The first option should not be used as we would make large read requests to hard drives every time a small Parameter is needed. Especially because the File is inside a .zip-Archive, parallel operations are not possible (Microsoft, 2011). As we always request more than one parameter when loading/changing a fixture, this method is slow and inefficient. The second option is often used for medium and large-sized .xml files when only parts of the XML are needed. This saves on memory and processing power. The Programmer needs to define which parameters are read and write code for each of them. Bantin uses this approach in his version as well as Fok in his (Bantin, 2021, p.23; Fok, 2020). As the gdtf format is a well-defined and documented specification the third option becomes a possibility: With the help of a .xsd the Windows development tools can create a C-Sharp Object which has all possible Datapoint in its definition (Maddock, 2022). The file it created is about 12000 Lines of Code and would be way too time-consuming for a programmer to create. Reading an XML file into the C-Sharp Code (into the Game engine) makes it possible to access all parts of the XML as a part of the Object, speeding up prototyping of new functions substantially, as all data is already present and correctly formatted. A drawback of this method is that a malformed XML will never be

read by this solution while the other solutions might have a chance of reading it if the malformed part does not need to be accessed. No malformed XML was found during this research.

Having read the XML file the C-Sharp Object (now stored in RAM) can be used to create a representation inside Unity Engine. Being able to extract basic information about the fixture from the C-Sharp Object a decision needs to be made on which DMX-Mode to load, as, per definition, each mode has its individual structure defined, so it is being treated as a separate device (Staffel et al., 2022). In the test scenes, a loader script gets the dmxMode from a save file, but user input into an interface is also possible. Scenes in Unity are based on game objects and their relationships. The structure of the ‹Geometries›-Node inside the GDTF schema is very similar, so parsing the Schema into Unity game objects is not hard. Fig. 4.6 shows a simple structure of game objects that reflect the structure that is defined in the GDTF schema 4.7.

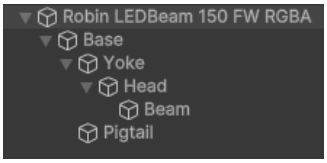

Figure 4.6: Structure of a simple fixture

The main object, highlighted in light gray, is the main “parent” object of the fixture. It holds a script: `GDTFDevice.cs`, holding data like address, managing all attached functions, and negotiating between them (see 4.6). As there are no 3D models or other graphical representations loaded at this point the fixture is invisible in 3D space.

Codeblock 4.7: Geometrie description Example (Position Data has been cut to reduce clutter)

```
1     ‹Geometry Model="Base" Name="Base" Position=[....]›
2         ‹Axis Model="Yoke" Name="Yoke" Position=[....]›
3             ‹Axis Model="Head" Name="Head" Position=[....]›
4                 ‹Beam BeamAngle="42.400000" BeamRadius="0.069500" BeamTy
5                 ColorRenderingIndex="75" ColorTemperature="8000.000000"
6                 FieldAngle="60.000000" LampType="LED" LuminousFlux="1796
7                 Model="Beam" Name="Beam" Position=[....] PowerConsumptio
8                 RectangleRatio="1.777700" ThrowRatio="1.000000"/›
9             ‹/Axis›
10        ‹/Axis›
11        ‹Geometry Model="Pigtail" Name="Pigtail" Position=[....]/›
12    ‹/Geometry›
```

4.2.3 Parsing the position matrix

Codeblock 4.7 omits the position data. Example data of such parameter looks like this: `"{1.000000,0.000000,0.000000,-0.000043} {0.000000,1.000000,0.000000,0.000000} {0.000000,0.000000,1.000000,-0.025000} {0,0,0,1}"`
Formating it in a more readable 4x4 Matrix:

$$Position = \begin{bmatrix} 1.000000 & 0.000000 & 0.000000 & -0.000043 \\ 0.000000 & 1.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 1.000000 & -0.025000 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These matrices are called “Spatial Transformation Matrices” or simpler TRS (Transform, Rotate, Scale) and can describe a translation, rotation, scale, and even shear in one matrix (BrainVoyager, n.d.). After parsing the values into a Untiy `Matrix4x4` the position, rotation, and scaling data could theoretically simply be read (Unity Technologies, 2022f), but because Unity uses a right-handed coordinate system with y as the “up” axis, and the GDTF files are defined with a left-handed coordinate system with z as the “up” axis, the TRS needs to be mirrored and rotated. As the 3D files face a similar issue the exact math is described more in chapter 4.5. The data returned after the matrix is transformed is then used to position the parts of the fixture in the relative position to each other.

4.2.4 Reading extra files (images and models)

The .zip Archive also includes images, symbols, and 3D models (Staffel et al., 2022, pp. 8–11, 29, 31). For the 3D representation (4.5) or the GoboTextures (4.4) these files need to be accessed. A Function supplying a file was programmed using the aforementioned “.zip” library, providing access to the needed files. Further processing of the files is touched upon in the respective chapters.

4.3 Light Beams

The Structure shown in Fig. 4.6 includes a `Beam` Object, defining where in the structure a Beam should be placed. As a second option, locating the Beam with a Geometry Reference is available to manufacturers and makers of GDTF descriptions (Staffel et al., 2022, pp. 38–39). It is most often used with Fixtures that have multiple light emitters, defining them once and referencing the definition, yielding a similar structure to 4.7 without defining the numerous details of the same Emitter multiple times (Staffel et al., 2022, pp. 34–36).

Figure 4.7: Structure of a fixture with multiple emitters

4.3.1 Switching to Geometry Beams

Switching to Geometry Beams is the first step to reduce the performance impact of the Assert. Instead of attaching a *Spot Light*, as done by Bantin and Wason (Bantin, 2021; Fok, 2020), a "fake" beam is attached to the "Beam" object. A already by Bantin mentioned and colloquially recommended solution is the "Volumetrics Light Beams" Asset available in the Asset Store (Bantin, 2021, p. 61). It is a paid Asset and promises "The perfect, easy and cheap way to simulate density, depth and volume for your spotlights and flashlights, even on Mobile! It greatly improves the lighting of your scenes by automatically and efficiently generating truly volumetric procedural beams of light to render high-quality light shafts effects." with a "Super FAST and low memory footprint: doesn't require any post-process, command buffers, nor compute shaders: works great even on low-performance platforms such as Mobiles and WebGL" (Tech Salad, 2023).

Adding a Volumetric Light Beam (VLB) to the Lamp Objects is quickly done and positioned with the same function as the other Objects (see 4.2.3). The Beam description also includes data about `BeamAngle`, `BeamRadius` and `FieldAngle` being used to set fitting parameters on the VLB: `Spot Angle` and `Light Source Radius`. The maximum brightness of a beam is available from the gdtf as `LuminousFlux`. The Beam inside Unity Engine has two parameters that need to be adjusted accordingly to this value: The `intensity` of the beam, essentially how transparent it is, and the `Light Range Max Distance`, controlling how far the beam reaches. Because a physically accurate representation was not deemed necessary for the answer to this thesis question, a simple calculation was set up, using one "brightness" value set on the GDTFDevice (see 4.6.3) to set both the `intensity` and the `Light Range Max Distance`.

While the Solution looks very promising, it was quickly apparent that there was room to optimize. Interestingly some settings led to the expected/promised performance improvements while others made almost no difference.

4.3.2 Optimizing with Batching

Batching is a simple solution to render many objects more efficiently. Multiple Styles of Batching exists (Saladgamer, 2021):

GPU Instancing is the most common form of batching. Under normal conditions, a draw call (a call to the GPU to render the attached mesh) needs to be done for every object, which is especially taxing on the CPU, as it has to prepare this data and then transfer it to the GPU-Memory. With GPU Instancing all Objects that use the same mesh, material, and shader can be pulled together into one draw call, telling the GPU to render the Mesh multiple times

without needing to transfer the data multiple times. Controversly GPU Instancing saves on CPU performance, as the CPU needs to spend less time transferring the data again and again to the GPU-Memory (Unity Technologies, 2023e). This Method only works if many objects are using the same mesh, material, and shader, as the Beams do.

SRP Batcher The Scriptable Render Pipeline Batcher (SRP Batcher) is a rather modern function of Unity Engine introduced together with the Scriptable Render Pipelines (Lever, 2022, p. 11). It uses a different approach to reduce time spent with draw calls: It reduces changes made to the GPU Memory. By keeping Material and their used Shaders in the Memory (in so-called CBUFFERs) the CPU only needs to send the mesh and the properties of each object that uses this Shader/Material and saves sending all Material data. This works particularly well if the Scene only uses a small amount of shaders. This also means the SRP Batcher can be, in contrast to GPU Instancing, more easily used on Complex Scenes. Materials need to share a Shader, called Material Variants, but they can use different Textures and Mesh (Unity Technologies, 2022g; Lever, 2022, p. 120).

Three test runs were conducted and chapter 5.1.1 dives into the lack of improvements that were determined with the Tests.

4.3.3 Geometry Detail: vertices, resolution...

Because optimizing the draw calls had little effect on the performance on the VR Headset the second prototype focused on the simplest and most recommended form of optimization: Reducing the number of vertices (Ferreira, 2019). The VLB-Asset has a simple setting to change the mesh that all beams use. Table 4.3 lists the Settings used for testing, while Fig. 4.8 shows a visual comparison between the settings.

Quality Setting	**No. of Sides**	**No. of Segements**	**Mesh Statistics**	
Low	6	1	50 vertices, 60 triangles	
Medium	16	1	130 vertices, 160 triangles	
High (Default)	24	5	386 vertices, 624 triangles	
Test Cylinder	16	1	32 vertices, 32 triangles	

Table 4.3: Comparison of VLB-Beam Mesh details

Performance Improvements were noticeable but not substantial, detailed in Chapter 5.1.1.

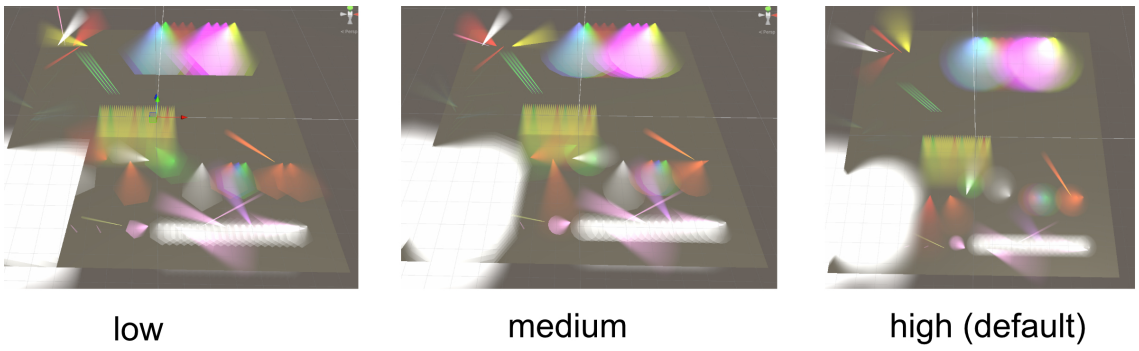

Figure 4.8: Comparison between the Quality Settings of the VLB

4.3.4 Self written Shaders

A further deep dive into the inner workings of the VLB Asset revealed a complex system of shaders to make many of the included features possible (Tech Salad, 2023). Understanding the inner workings of the shaders seemed key to tackle further performance improvements so a decision was made to research how a shader is written.

To write a shader one needs to either use ShaderLanguage (see 2.1) or the Shader Graph (see 2.1). Before development of the shader started the following specifications were defined:

- Inputs:
  - Color of the Beam
  - Length of the Beam
  - Angle of the Beam
  - Intensity of the Beam
  - *Falloff Speed*
- Simple Mesh with a low amount of vertices
- Mesh deforms (Angle and Lenght)
- Color and transparency changes (Color and Intensity)
- *The Intensity falls off toward the End of the Beam*

The *italic* parameters are later removed (see page 44). A Shader using the Nodes of the Shader Graph, and a shader using Shader language were created and compared. This explanation follows the written Shader; The full Shadergraph, with annotations, is added to the end of this document as appendix 1. To write a shader one first defines if the shader renders a transparent or an opaque object. The shader itself is split into two parts: The vertex shader and the pixel or fragment shader. The vertex shader is run on every verticie of a mesh, its expected return value being the position of the verticie "on the screen" (in *Screenspace*). The fragment shader uses this information to run its calculation of every pixel the object covers on the screen, yielding the color the pixel of the screen should have (Varcholik, 2014, pp. 54–56).

Vertex Shader

Codeblock 4.8: Vertex Shader

```
1    v2f vert(appdata v)
2            {
3                 v2f o;
4
5                UNITY_SETUP_INSTANCE_ID(v);
6                UNITY_INITIALIZE_OUTPUT(v2f, o);
7                UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
8                o.objPos = v.vertex;
9
10               v.vertex.y += 1;
11               v.vertex.y *= _falloffLength*0.5f;
12               v.vertex.x *= tan(radians(_Angle))*(v.vertex.y);
13               v.vertex.z *= tan(radians(_Angle))*(v.vertex.y);
14
15
16
17               o.vertex = UnityObjectToClipPos(v.vertex);
18
19               return o;
20           }
```

Using codeblock 4.8 the vertex shader is explained in the following section. In the vertex shader, the default action is just one function: `UnityObjectToClipPos`. It uses the position data of the verticie and returns the position in screenspace (4.8 line 17). The exact math is not relevant to the performance and is out of the scope of this work. Fig. 4.10 shows the mesh used for the beams. Because we want to deform our mesh we run the calculations on the

vertice position data before calculating the screen space position:

In the first step, the mesh is deformed to reflect more of a beam: Having a small point of origin and spreading out with increasing distance. While it would have been possible to make a more "beamlike" mesh from the beginning (shown in 4.9), the decision to use a cylinder was inspired by the VLB Asset, as it includes the parameter "light source radius" to widen the origin-point, useful when the emitter is a large lens or wash light (Tech Salad, 2022; Greule, 2021, pp. 151–170). In the simple shader written for testing this feature is not included; The mesh is a cylinder nonetheless.

A conscious decision was made with the original mesh: The cylinder has a height of 2 and a radius of 0.5. As its origin is exactly in the middle (at height 1), the first step is to move it up 1 unit, putting the center of the bottom circle of the mesh at the coordinate (0,0,0). In line 9 one unit gets added to the y-position of the vertice, using the += operator, and in line 10 the y-position gets multiplied with the `\_falloffflength` parameter, essentially stretching the cylinder in upwards direction, making the beam reach (or shine) further (Swiniarski, 2022). As the cylinder is already 2 high, the `_falloflength` gets halved before multiplication, making its value a 1:1 presentation of how many units the beam reaches.

The operation in lines 12 and 13 gives the beam its "beam shape". It is split into two parts:

$$v.vertex.z* = \tan(radians(\_Angle)) * v.vertex.y$$

With this calculation the z (and x) position of each verticie is adjusted depending on the height by multiplying the position of the verticie (essentially the radius of the beam at the height of the verticie) with the height of the beam (the blue part of the calculation), creating a simple beam shape. Because the shape of a beam, when looked at from the side, can be argued as two right-angle triangles, simple trigonmic functions can be applied. The tangent of the angle at the bottom of the Beam equates to the ratio between the height and the radius (see Fig. 4.9). As the height is just `v.vertex.y` and already used in the function and the Angle is input by the user or a script, the red part of the equation simply moves the larger part of the cone in and out, depending on the input on the `\_Angle` parameter.

Fragment Shader

Codeblock 4.9: Fragent Shader

```
1     fixed4 frag(v2f i) : SV_Target
2             {
3                 UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i);
```

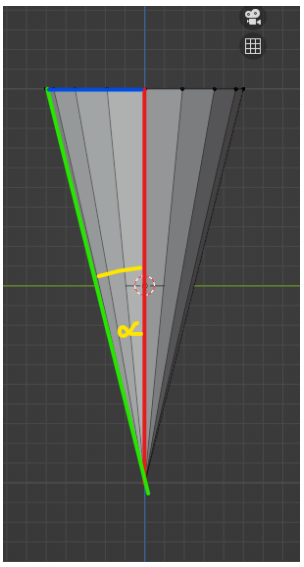

Figure 4.9: Visualizing how the tan of α can give us the radius (blue) when the height (red) is known

```
4
5                   _Color.w = (1-(i.objPos.y/_fallofflength))*_FalloffSpeed;
6                   return _Color;
7               }
```

The Fragments shader, also called the Pixel shader, is a lot leaner than the Vertex shader. It is run on every pixel the beam occupies and receives the position of the pixel inside the `i` element. Using this data together with the `\_fallofflength` and `\_Falloffspeed`, the opacity, saved in the `_Color.w`-value, is adjusted. Then the color is returned.

During the process of narrowing down the exact impact of different parts of the shader code, a modification was made to the fragments shader. The only calculation done inside the fragment shader was removed, stripping the beam of its fading with distance, improving performance (detailed in chapter 5.1.1).

4.4 Light Projections

Generating projections onto a scene requires information on the geometry of the scene. Unity Engine enables shaders to access information on the scene in the form of a Depth Map.

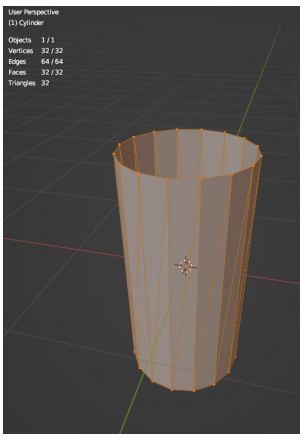

Figure 4.10: Mesh of the cylinder used for the Beam shader

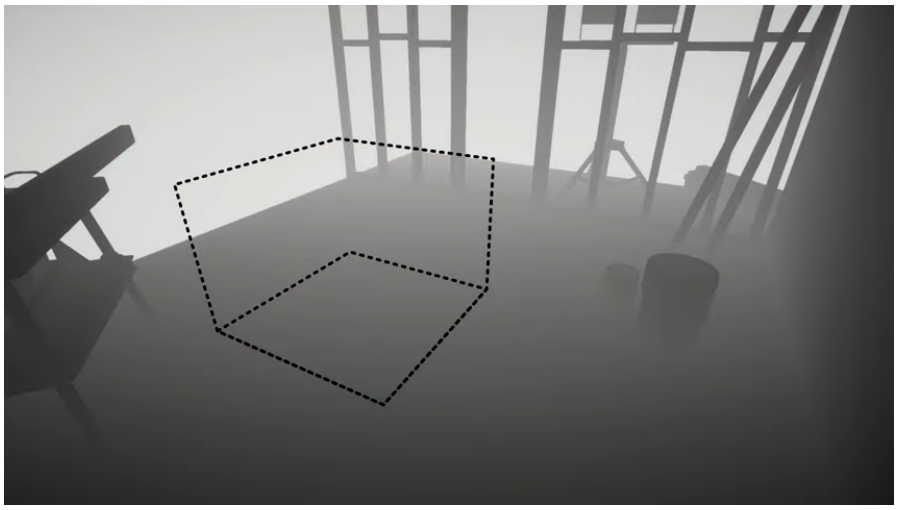

Figure 4.11: Depth Map with transparent (not in Depth Map included) Cube marked. (Source: Ilett, 2021)

Depth Map

A Depth Map, sometimes called a Depth Texture or Depth Buffer, is a texture generated by the Engine including depth information about the image (Fig 4.11). It colors each pixel by the distance of the first opaque object from the camera, black (close) to white (far) (Iett, 2022, p. 209). This Depth Map is created before any transparent objects are rendered, leading them to be not visible in the Depth Map. There is no clear data about the performance impact of creating and using the depth texture. Unity's Documentation only states: "Note that generating the texture incurs a performance cost." (Unity Technologies, 2020). Other Parts of the documentation state: "Depth textures can come directly from the actual depth buffer, or be rendered in a separate pass, depending on the rendering path used and the hardware.

Typically when using the Deferred Shading rendering path, the depth textures come "for free" since they are a product of the G-buffer rendering anyway." (Unity Technologies, 2021b). When using URP the creation of the texture needs to be enabled manually, regardless of using the deferred or forward renderer(Iett, 2022, p. 207).

Inspired by the steps Unity Engine takes to calculate shadows, a first prototype was developed imitating the process: A camera, placed at the origin of the projection (the light source) renders a Depth Map. This Depth Map is then used to determine the size of the cookie that would be projected onto the surface: The further away something is the larger the cookie would be. After generating the texture and the Depth Map from the point of view of the light, some matrix multiplications can be used to render the texture onto the scene from the player's camera point of view (Lague, 2021, 24:20-25:33; Unity Technologies, 2019c). While working on this thesis no tangible results could be achieved using this method. The complex matrix multiplications never yielded the expected result, and a second method was prototyped:

4.4.1 Projection Shader

Researching into the calculations of projections a different approach to projections was discovered: Using a mesh (a cube in this case) and the Depth Map, the intersection could be calculated by:
First, calculating the position of each point on the depth map the cube covers (see Fig. 4.11). This is done with a matrix multiplication called the "*InverseViewProjection*", literally going *Inverse* from the *View Projection* (the depth map) to the world space (Jeremiah, 2011; Iett, 2022, p. 38). **Second**, checking if the Point is inside the Cube: A second matrix multiplication transforms the coordinates from world space to the local space of the cube. As the Cube is of known size ((0.5,0.5,0,5) to (-0.5,-0.5,-0.5) in the example), it is easy to determine if the point is inside the Cube (see Fig. 4.12; Ilett, 2021, 6:28). If it is not the pixel gets discarded and no further calculations on it are made (Iett, 2022, p. 298). At this point, the shader returns all the pixels where the cube intersects with any opaque object. In the **third** step we add 0.5 to the local coordinates of the cube (to get their range between 0 and 1) and use the x and z coordinates of the pixels (in the local space of the cube) to sample a texture, leading to the texture being projected onto the surface from the y direction.

The ShaderGraph implementation of Ilett works great, even in VR. It is attached as attachment 2. As it is made for decals, e.g. stickers or spraypaint, some modifications had to be done to make it work as a light projection:

- The cube needs to be expanded to at least encompass the light beam.
- The size of the projection needs to change depending on the distance from the light source.

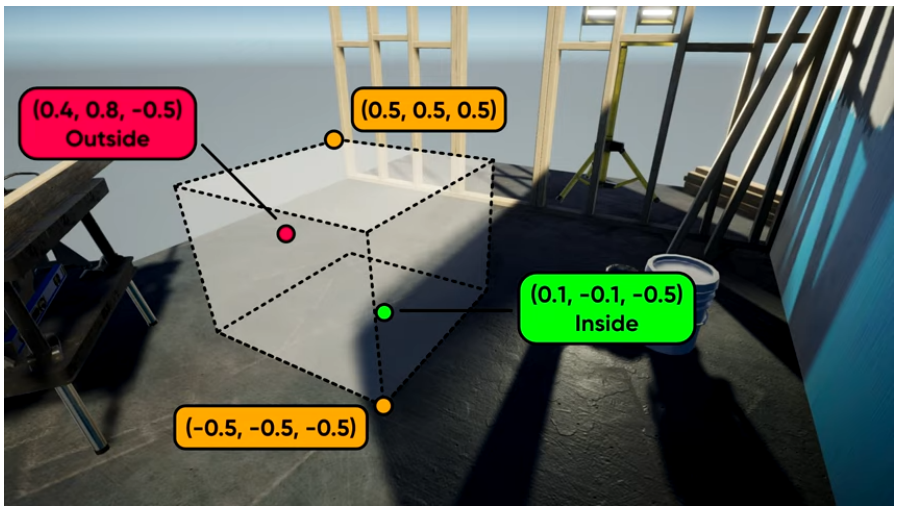

Figure 4.12: Two Points (red and green) sampled from the depth map and transformed to local space of the cube. (Source: Ilett, 2021)

Implementing the first point led to two further discoveries: First, because of the changed size of the Cube, almost every calculation had to be touched, making the ShaderGraph take on the literal form of "spaghetti Code" (Abbes et al., 2011, pp. 181–190). Second, when the camera entered the Cube itself, the projection disappeared. Especially the later point could only be fixed by switching from the Shader Graph to writing Shader Code due to a bug in the Version of ShaderGraph that was worked with at this point in development (still working with Unity Version 2021.3.12f and Shadergraph version 12.0.0 (Unity Technologies, 2021d, p. 12.0.0). Ilett based his work on a written shader by Colin Leung (Leung, 2022). While shaders made with Shader Graph are automatically compatible with virtual reality, written shaders are not: The Shader by Leung is either only rendered on one eye, or directly in the face of the viewer, seemingly ignoring all depth information (the latter in *Single-pass Instanced* the first in *Multi-pass* rendering modes (see 2.1)). Following a guide by Unity and fixing a couple of issues related to using different helper functions (Shader include functions), the shader was made compatible with the VR Headsets (Unity Technologies, 2021e; St-Laurent, 2004, pp. 17, 343; Unity Technologies, 2022h). The main changes to scale the projection will be explained using Codeblock 4.10: Line 2 calculates a Distance-Factor by using the z-value of the cubes' height and because the cubes' coordinates are already corrected to be in the -0.5 to 0.5 range again after it was scaled, adding 0.5f creates a range between 0 and 1. Because the UV coordinates (the coordinates placing the texture) react opposite to scaling (large numbers scale the object down), the factor is the reciprocal of that 0 to 1 value. In line 10 the uv map is then multiplied with that factor after all pixels not needed are clipped from calculations in line 6 (as described in Fig. 4.12). Because of running from 0 to 1, the factor scales the uv toward the 0,0 corner, resulting in the texture getting smaller toward a corner. Line 11 uses the ScaleFactor to add the right amount back to the UV coordinates, centering the projection

again.

Codeblock 4.10: Main changes to the Leung's decal shader

```
1         //This calculation needs to be done before the clip function, as the clip
      for some unknown reason interferes with the calculation.
2         float distanceScaleFactor = 1/(decalSpaceScenePos.z+0.5f);
3
4         // if ZWrite is Off, clip() is fast enough on mobile, because it won't write
       the DepthBuffer, so no GPU pipeline stall(confirmed by ARM staff).
5         //clips everything outside the cube && not on a surface;
6         clip(0.5f-abs(decalSpaceScenePos));
7
8         //Sample the decal texture
9         float2 uv = decalSpaceUV.xy * _MainTex_ST.xy + _MainTex_ST.zw;
10        uv *= distanceScaleFactor; //scaling it down by zoom level
11        uv += 0.5f-(0.5f*distanceScaleFactor); //centering it in unitys cube mesh
```

In the described state the shader still retains its original texture: A demo sticker included in the shader (Leung, 2022). The loading and changing of textures is described in chapter 4.6.6 Wheels.
After the test where run and evaluated further optimization was not considered, shifting focus to other parts of this research.
A single feature was later added, allowing color mixing of the Gobo texture to emulate colored light shining through the Gobo, but was not considered to impact the performance, meaning tests were run. This change is reflected in the attached script but will not be described in the scope of this work (see appendix 1).

4.5 3D Models

The .gdtf-Files include, by definition, a 3D model of the fixture (Staffel et al., 2022). The creators of fixtures are requested to supply 3 types of 3D Models of their fixture: "Low", "Default" and "High". It is also stated that *"All models of a device combined should not exceed a maximum vertices count of 1200 for the default meshlevel of detail"* (Staffel et al., 2022, 27–28, Table 33).

None of the tested GDTF files (for the full list see attachment 4) included any other 3D model than the "Default". Additionally, many Fixtures did not keep to the 1200 vertices, sometimes exceeding the count tenfold.

While this Module includes a lot more code than the Shaders described in the sections before this one, the description is intentionally kept short, only diving into the specialties the GDTF description holds. For a more detailed view of the implementation, review the source code (https://github.com/DarkTJ/Unity-GDTF), the DIN SPEC (Staffel et al., 2022), and the main sources for details about the 3DS file format: Autodesk Ltd., 1996; Battista, 2016; Pitts and Bourke, 1996.

Five steps are done to get a 3D representation of the fixture from the file into the 3D space of Unity:

- 1. Reading the information from the xml-file
- 2. Reading 3DS and GLTF files to create a Mesh
- 3. Scaling and transforming the Mesh
- 4. 3D representation of the Emitter
- 5. 3D description of the Beam

4.5.1 Reading the information from the xml-file

How the file is opened, read, and parsed is already described in 4.2 GDTF Import. Fig. 4.7 shows a simple fixture: This Module uses the `Model` parameter from that description to determine which Model to load. The XML file also includes a `Models` node (see Codeblock 4.11), in which each of the Models the fixture uses are described in more detail. Each `Model` has a `Name`, a `Length, Width and Height`, and either a `primitiveType` (when a primitive geometry is used) or a `File`, holding a filename (Staffel et al., 2022, pp. 27–28). This filename is subsequently used to load the right model.

Codeblock 4.11: Example Models Node from a GDTF-file

```
1 ‹Models›
2       ‹Model File="yoke" Height="0.187000" Length="0.571000" Name="Yoke"
    PrimitiveType="Undefined" Width="0.102000"/›
3       ‹Model File="" Height="0.020000" Length="0.065000" Name="Beam" PrimitiveType="
    Cylinder" Width="0.065000"/›
4       ‹Model File="base" Height="0.100428" Length="0.456001" Name="BASE"
    PrimitiveType="Undefined" Width="0.184000"/›
5       ‹Model File="head" Height="0.101394" Length="0.505000" Name="HEAD"
    PrimitiveType="Undefined" Width="0.102000"/›
6 ‹/Models›
```

4.5.2 Reading 3DS and GLTF files to create a Mesh

When work on this Asset began the GDTF Spec only allowed one file type for 3D Models: 3DS. As already stated in Research Design, the 3DS format is quite old and niche, not being supported by Unity Engine (Unity Technologies, 2022k). Finding an archived Asset, made for Unity 5.3.4 (released in 2016), which had the simplest form of 3DS file import, helped get the development kickstarted. (Battista, 2016). "The 3ds file format is made up of chunks. They describe what information is to follow and what it is made up of, its ID and the location of the next block. IF you don't understand a chunk you can quite simply skip it." (Pitts and Bourke, 1996). The loader created to read the files does exactly that: Only reads the chunks understood and needed, even leaving out material definitions, only returning a simple mesh and color. Because meshes inside 3DS files, analog to the Spatial Transformation Matrices (see 4.2.3), use the left-handed coordinate system with z as the "up" axis, while Unity Engine uses a right-handed coordinate system with y as the "up" axis, the object is mirrored and rotated with the following code:

Codeblock 4.12: Rotating and mirroring a 3DS mesh to correctly import it into Unity Engine. The ability for multiple meshes has been omitted here to reduce complexity.

```
1    Quaternion rotationQuarterion = Quaternion.Euler(270,0,180);
2    //Fix the vertices for the LeftHanded Coordinate System
3    for (int verti = 0; verti < verticesModel.Length; verti++) //Each vertice of the
      mesh
4    {
5        verticesModel[verti] = rotationQuarterion * verticesModel[verti]; //gets
    first rotated around the center by x=270° and z=180° (defined in line 1)
6        verticesModel[verti].x *= -1;  //and then mirrored on the x axis
7    }
8    //Flipping the normal on the triangles, to turn it "inside out" because of the
    mirror
9    for (int tri = 0; tri < trianglesModel.Length; tri++)
10   {
11       int store;
12       tri++; //the first coordinate can stay as is
13       store = trianglesModel[tri];
14       trianglesModel[tri] = trianglesModel[tri + 1]; //the second gets the value
    of the third
15       tri++;
16       trianglesModel[tri] = store; //the third gets the value of the second
17   }
```

The Mesh can then be returned and handled further.

In the newest iteration of the GDTF DIN SPEC (from Feb. 2022) a second allowed file format was introduced: .gltf (Staffel et al., 2022, p. 28). The GLTF format is a much more known and supported 3D file format: "Khronos promotes glTF as the JPEG of 3D. They believe that the format is so useful that it will become as ubiquitous as the JPEG format for 2D images. The glTF format works well for Augmented Reality (AR) and Virtual Reality (VR) because it supports both motion and animation." (Schechter, 2020). Already having worked with GLTF files for the development of the XREvent Creator (see chapter 6.3), a solution for loading these files was already in place and could quickly be adapted to return the Mesh the same way the 3DS-loader did (Brigsted, 2022).

The returned Mesh is then used to create a game object, which is named according to the `Name`-parameter, and, because the XML file has `Length, Width and Height` as parameters and not scale-factors, the bounding box needs to be determined first (stored as `ModelSize` and by dividing the GDTFSize with the ModelSize the scale factors are calculated (see Codeblock 4.13). Lastly, the game object is parented to its respective Object.

Codeblock 4.13: Scaling the 3D Object by using the bounding box

```
1     //Set scale
2     Vector3 GDTFSize = new Vector3(modelToLoad.Length, modelToLoad.Height,
      modelToLoad.Width); //getting the length/height and width
3     Bounds ModelBounds = new Bounds();
4     foreach (var rend in threeDModel.GetComponentsInChildren<Renderer>())
5     {
6         ModelBounds.Encapsulate(rend.bounds); //all loaded meshes get measured into
      one bound
7     }
8     Vector3 ModelSize = ModelBounds.size;
9     threeDModel.transform.localScale = new Vector3(GDTFSize[0] / ModelSize[0],
      GDTFSize[1] / ModelSize[1], GDTFSize[2] / ModelSize[2]); // the size-factor is
      calculated and applied
10    threeDModel.transform.SetParent(parent.transform,false); //the object is
      parented to its respective place in the structure of game objects, getting its
      correct position.
```

4.5.3 The Lightsource

The Lightsource or Emitter is described inside a GDTF definition as "Beam". Using its description a 3D representation can be created using a special shader emulating a Lightsource, adding realism/immersion, especially when looking directly at a light source from inside the

light beam (see Fig. 4.13). Sadly not all tested devices defined their emitter with a 3D model, leaving some lights without this representation.

4.5.4 Loading Cubes

Because the performance took a big hit when all models were loaded (see 5.1.3) a quick second prototype was built: Instead of loading the 3D Model from a file or a primitive, a cube is spawned. This cube is then scaled and positioned using the same method and data as the real object, leading to a surprisingly accurate representation of the fixture (see Fig. 4.14) but with substantially fewer vertices in the scene.

4.6 Art-Net Control

Following the proposed attributes from chapter 3.5.5, this chapter dives into each attribute, explaining the (stylistic) function and covering hurdles in implementing it. All functions described here are considered support functions. As they are necessary to get anything out of the fixtures they do not get their own section in chapter 5 Evaluation, instead their impact, if there is any, is discussed in this chapter.

All functions that a fixture can perform are described in the XML file included in the GDTF. The `‹DMXMode›`-node holds all used channels, the geometry they influence, and the dmx channel they use. Depending on what the Channel does, more information is supplied (as detailed in the following sections) (Staffel et al., 2022, pp. 43–47, 54–64).

- reading of 8bit and 16 bit channels
- Movement on up to two axis
- Dimmer
- Shutter + Strobo
- Zoom
- Wheels
  - Color Wheels
  - Gobo Wheels
- RGB Values

4.6.1 Reading of 8bit and 16 bit channels

The first step to controlling a fixture is to collect the data that holds the dmx signal. A pre-made asset is used to receive Art-Net Packets, allowing modern light-control consoles or desktop software to "talk" to the virtual fixtures (James, 2015; Cho, 2018). These Art-Net Packts get stored inside a `DMXController` inside the scene as a dmxData array. It also notifies all fixtures in the scene of the new packet and the devices copy their respective part of the dmxData array into the `GDTFDevice`. This `GDTFDevice`-script is attached to the main object of each fixture, coordinating all other attributes. It holds a list of all attributes attached to the fixture and the channel or channels on which they receive their data. Codeblock 4.14 for example holds a Channel with `Offset="9"`, meaning it receives its data on the Channel of the fixture plus 9. The two Attributes, Pan and Tilt, in Codeblock 4.15 both hold two values

in the `Offset`-parameter, meaning they are 16-bit Channels. For each DmxChannel a script is written which receives the data and does the expected action (explained in the following sections). As new data arrives, it is distributed to each script: one channel gets sent as a byte value between 0 and 255, and 16-bit values as an unsigned integer between 0 and 65535.

Codeblock 4.14: Descripton of a Dimmer inside the XML

```
1    <DMXChannel DMXBreak="1" Geometry="Head" Highlight="255/1" InitialFunction="
     Head_Dimmer.Dimmer.Dimmer 1" Offset="9">
2        <LogicalChannel Attribute="Dimmer" DMXChangeTimeLimit="0.000000" Master="
     Grand" MibFade="0.000000" Snap="No">
3            <ChannelFunction Attribute="Dimmer" DMXFrom="0/1" Default="0/1" Name="
     Dimmer 1" OriginalAttribute="" PhysicalFrom="0.000000" PhysicalTo="1.000000"
     RealAcceleration="0.000000" RealFade="0.000000">
4                <ChannelSet DMXFrom="0/1" Name="Closed" WheelSlotIndex="0"/>
5                <ChannelSet DMXFrom="1/1" Name="" WheelSlotIndex="0"/>
6                <ChannelSet DMXFrom="255/1" Name="Open" WheelSlotIndex="0"/>
7            </ChannelFunction>
8        </LogicalChannel>
9    </DMXChannel>
```

Codeblock 4.15: Descripton of Pan and Tilt inside the XML

```
1
2 <DMXChannel DMXBreak="1" Geometry="Yoke" Highlight="None" InitialFunction="Yoke_Pan.
    Pan.Pan 1" Offset="1,2">
3     <LogicalChannel Attribute="Pan" DMXChangeTimeLimit="0.000000" Master="None"
    MibFade="0.000000" Snap="No">
4         <ChannelFunction Attribute="Pan" DMXFrom="0/2" Default="32768/2" Name="Pan 1
    " OriginalAttribute="" PhysicalFrom="270.000000" PhysicalTo="-270.000000"
    RealAcceleration="0.000000" RealFade="2.500000">
5             <ChannelSet DMXFrom="0/2" Name="" WheelSlotIndex="0"/>
6             <ChannelSet DMXFrom="32768/2" Name="Home" WheelSlotIndex="0"/>
7             <ChannelSet DMXFrom="32769/2" Name="" WheelSlotIndex="0"/>
8         </ChannelFunction>
9     </LogicalChannel>
10 </DMXChannel>
11 <DMXChannel DMXBreak="1" Geometry="Head" Highlight="None" InitialFunction="Head_Tilt
    .Tilt.Tilt 1" Offset="3,4">
12     <LogicalChannel Attribute="Tilt" DMXChangeTimeLimit="0.000000" Master="None"
    MibFade="0.000000" Snap="No">
13         <ChannelFunction Attribute="Tilt" DMXFrom="0/2" Default="32768/2" Name="Tilt
     1" OriginalAttribute="" PhysicalFrom="-135.000000" PhysicalTo="135.000000"
    RealAcceleration="0.000000" RealFade="0.800000">
14             <ChannelSet DMXFrom="0/2" Name="" WheelSlotIndex="0"/>
15             <ChannelSet DMXFrom="32768/2" Name="Home" WheelSlotIndex="0"/>
16             <ChannelSet DMXFrom="32769/2" Name="" WheelSlotIndex="0"/>
17         </ChannelFunction>
18     </LogicalChannel>
19 </DMXChannel>
```

4.6.2 Movement

Because each ‹Axis›-node has a ChannelFunction describing its maximum and minimum rotation (PhysicalFrom and PhysicalTo, see Codeblock 4.15), the DMX Value can be used to calculate the exact rotation the axis, respectively the game object, using the following formula:

$$rotation = PhysicalFrom + (PhycsicalTo - PhysicalFrom) * (DMXvalue/DMXMaxValue)$$

4.6.3 Dimmer

The dimmer script uses the received DMX value to calculate a factor of how much of the maximum power the lamp should output. Because the dimmer is not the only attribute influencing the brightness of the fixture, the calculated value is sent to the `GDTFDevice`-script. It waits for all attributes to send their changes and then updates the beams.

4.6.4 Zoom

Zoom influences two values: First, the angle of the beam is calculated exactly the same as the rotation of the axis is in 4.6.2. Second, the Zoom influences the brightness of the beam, as the light is either focused or spread out. The DIN SPEC does not define a calculation to be used for this. After a short research into the math behind lenses and focused light the exact calculation was declared out of scope of this work. A simple calculation was implemented, simply calculating a factor between the default value of the zoom, the current value of the zoom, and the `PhysicalTo` and `PhysicalFrom` values. Similar to the dimmer this factor is sent to the `GDTFDevice`-script.

4.6.5 Shutter + Strobo

The strobo and shutter functions are batched as they are closely related. In real fixtures the shutter is a mechanical solution to block the beam, when moving it in and out of the beam quickly it can also be used as a strobo. Modern LED fixtures often use a "software shutter" essentially just turning the emitter on and off, as LEDs do not have any afterglow or startup time. As the shutter can influence the brightness of the beam without changing the dimmer, it also just sends its factor to the `GDTFDevice`-script. The description of the shutter inside the XML includes all the functions the shutter can do: The most simple operation "Shutter closed" and "Shutter open" are a simple multiplication of the beam intensity with 0 and 1 respectively. All other functions are time-dependent, meaning they need to be calculated every frame. In Unity, the current timestamp can be received in each frame with the `Time.time()` function, similar to `Time.deltaTime` (see chapter 2.1). This value is used in different calculations to determine the brightness factor of the Shutter Attribute:

In the following equations, `w` holds the period (`DutyCycle`) of the function (in seconds), and `x` holds the `Time.time()`-value (in seconds).

A Strobo is a simple square wave, turning off and on following a set frequency. It is generated by making a boolean operation on a sinus function, essentially returning 1 when the sinus is above zero and 0 when below:

$$y = 0 < \sin\left(x \cdot \frac{\pi}{0.5 \cdot w}\right)$$

A Pulse differentiates itself from the Strobe by fading. Three kinds of pulse patterns are available:

Fig. 4.15 shows the first pattern: Simply called `Pulse`, rising and falling linearly. This triangle wave can be implemented using the modulo operator:

$$y = \frac{1}{(0.5 \cdot w)} \cdot |0.5 \cdot w - \mathrm{mod}\,(x, w)|$$

The second pattern, Fig. 4.16, simplifies the formula. The modulo operator generates a saw wave on its own, just being modified by w to create the desired Duty Cycle.

$$y = \frac{1}{(w)} \cdot \mathrm{mod}\,(x, w)$$

Fig. 4.17 shows the third pattern where the pulse only fades when lowering the brightness. As it is just the second method mirrored, the formula is simply subtracted from 1:

$$y = 1 - \frac{1}{(w)} \cdot \mathrm{mod}\,(x, w)$$

4.6.6 Wheels

In real fixtures colored foils and gobo patterns are installed inside wheels. They are called wheels as they physically turn inside the fixture: A color wheel is defined with RGB values and physical rotation values, while a gobo wheel additionally saves a gobo texture (see Fig. 3.2 for a reference). It is technically not discouraged to define a colored gobo by setting a color value, **Annex E** "Wheel Slot Image Definition" defines: "Colored gobos (3) shall use an RGB approximation. The RGB approximation shall be calculated on the basis of Pure White is the CCT of the fixture light source and the ICC color profile embedded within the PNG. (See ISO 15076-1:2010) The default shall be sRGB" (Staffel et al., 2022, p. 97). When receiving a DMX value it is converted from the DMX range, 1 to 256, to a physical rotation value ,0° to 360°. The color or gobo texture at that rotation value is then either given to the `GDTF-Device` or directly to the projector to be displayed. Not deemed necessary to implement in the scope of this work is the transition between the different wheel positions. In a real fixture, the transition takes time and can be visible as the projection/color changes from one to the next. The current implementation instantaneously snaps the color or gobo from the previous to the desired location.

4.6.7 Loading a Texture2D

Loading the textures seems straightforward at first.

But there is a fatal pitfall in this. Unity saves each texture loaded in the limited Random Access Memory (RAM) of the device. Spawning 100x of the same lamp, each having two gobo wheels with 8 textures, degrades the performance on the VR headset substantially. Here a more complex solution needed to be implemented: The concept is simple: When loading the same fixture multiple times, instead of loading all images for each fixture from a file, use the one already in memory. A simple Dictionary of all the textures that have been loaded already was created. As a key inside the Dictionary, we first used the name of the fixture and the name of the file, later switching to just the file name as it was discovered that even different fixtures of the same manufacturer sometimes used the same gobo, further reducing memory usage. Additionally, a built-in function from Unity Engine was used to compress the textures, making them more suitable to be used on the VR headset that have very limited memory (Unity Technologies, 2022i; Williams, 2015).

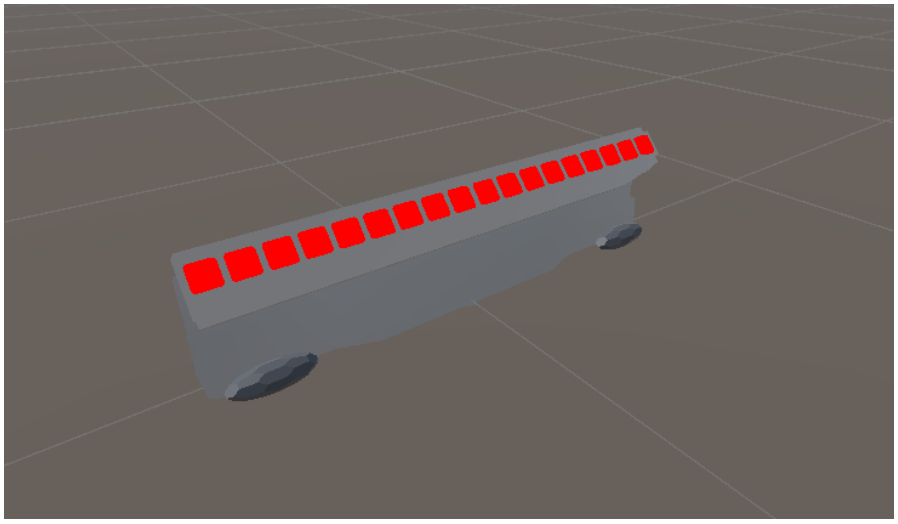

Figure 4.13: A fixture (Robe Robin Tetra 2) defining its emitters in 3D, here colored red. (Source: Screenshot inside Unity Engine)

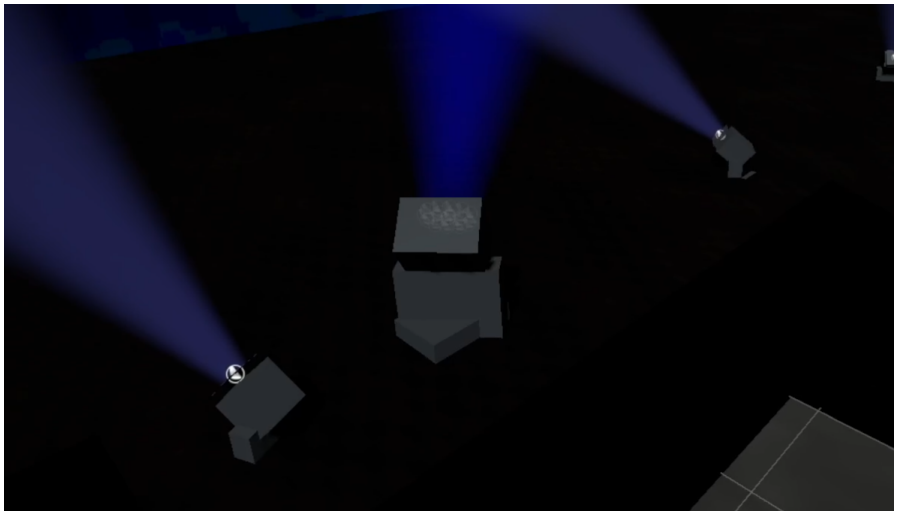

Figure 4.14: Cubes instead of 3D Models are used.

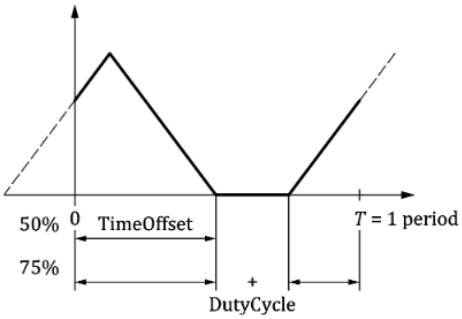

Figure 4.15: A triangle wave describing a pulse (Source: Staffel et al., 2022, p. 98)

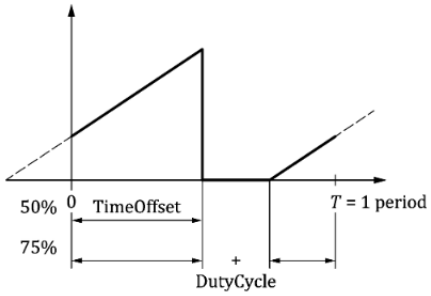

Figure 4.16: A saw wave describing a “PulseSawOpen” (Source: Staffel et al., 2022, p. 100)

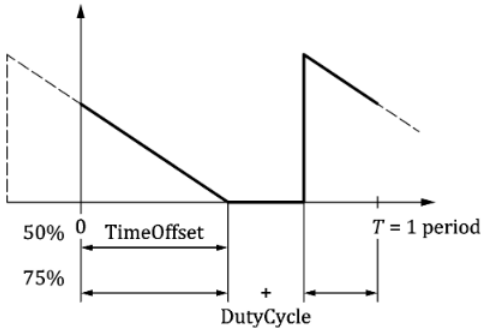

Figure 4.17: A saw wave describing a “PulseSawClose” (Source: Staffel et al., 2022, p. 99)

5 Evaluation

5.1 Test-Results

The following sections first go through each module's test results, contextualizing and analyzing each change and the resulting performance impact, and later compare the results to each other.

5.1.1 Light Beams

Optimizing with Batching

Fig. 5.1 shows the results instancing has when measuring on a computer: A huge improvement in framerate can be seen, especially in the first scene, as the CPU saves magnitudes of work. Even the simpler second scene, with a smaller amount of beams, improves by a significant amount (around 22%), as the CPU is freed of draw calls. Only the third scene's performance is indifferent to the settings because of its small amount of beams. The SRP Batcher performs worse than the "old school" GPU Instancing, as the scenes are rather simple and the beams all use the same mesh and material, an ideal setup for GPU Instancing (Unity Technologies, 2022g).

Fig. 5.2 shows the same tests, run on the VR headset. Contrary to the expectation no performance gain could be observed. The average framerate even drops, especially visible in the "Concert Scene". In subsequent research, multiple recommendations were found not to use GPU Instancing on the VR headsets. (Saladgamer, 2021; Meta, 2022a). Searching for further information hinted at the Tile-Based GPU architecture and its implications (Ferreira, 2019;). All further discovered sources dive very deep into low-level memory and GPU management which can not be covered in the scope of this work (Molnar, 1994; Arm Limited, 2021). Because the performance of this part of the research could be substantial (as fig. 5.1 shows) it is emphasized in chapter 6.2 Outlook.

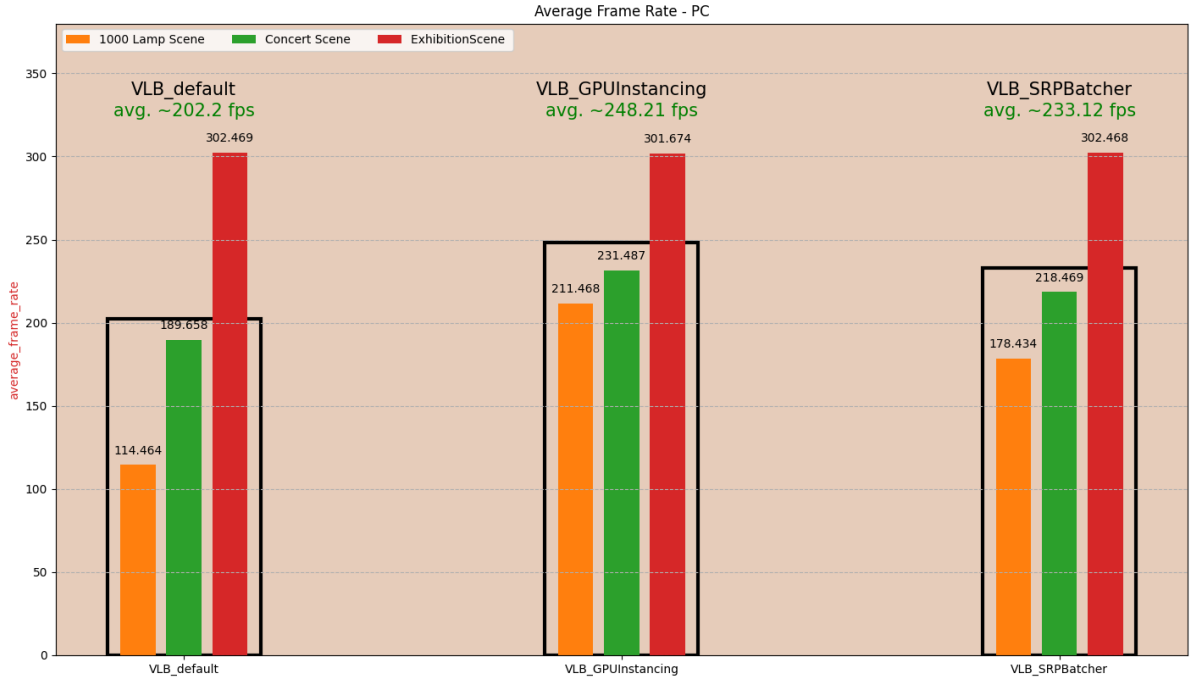

Figure 5.1: Comparing different render modes for VLB-Beams on PC

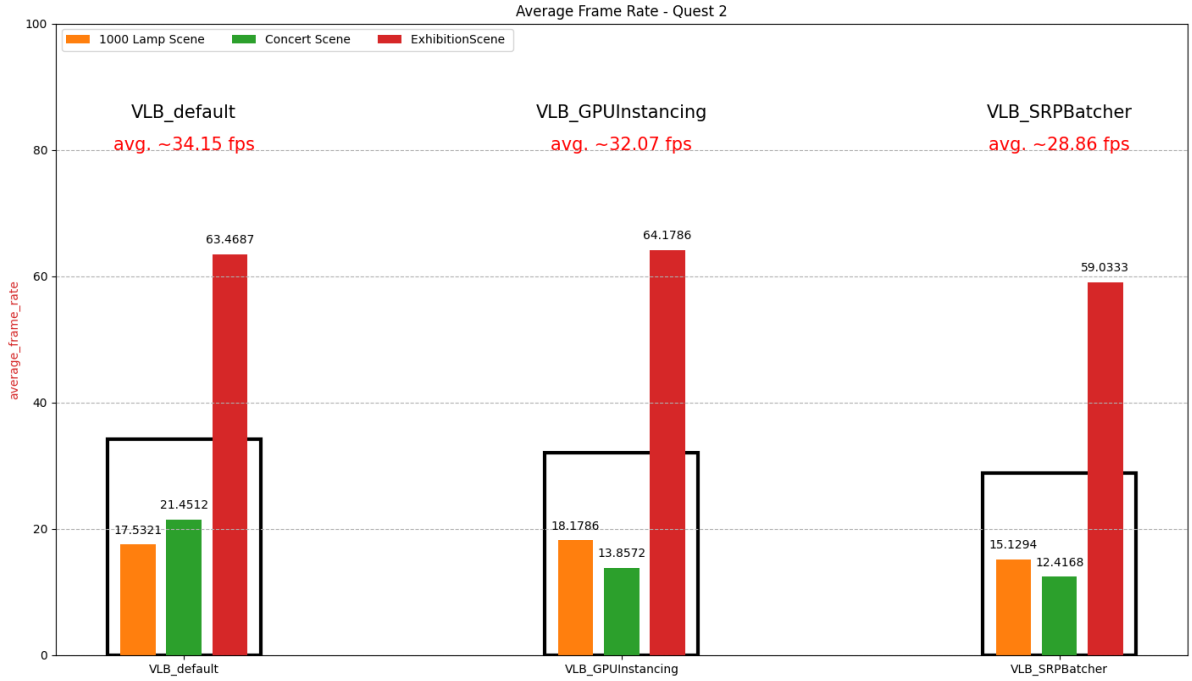

Figure 5.2: Comparing different render modes for VLB-Beams in VR

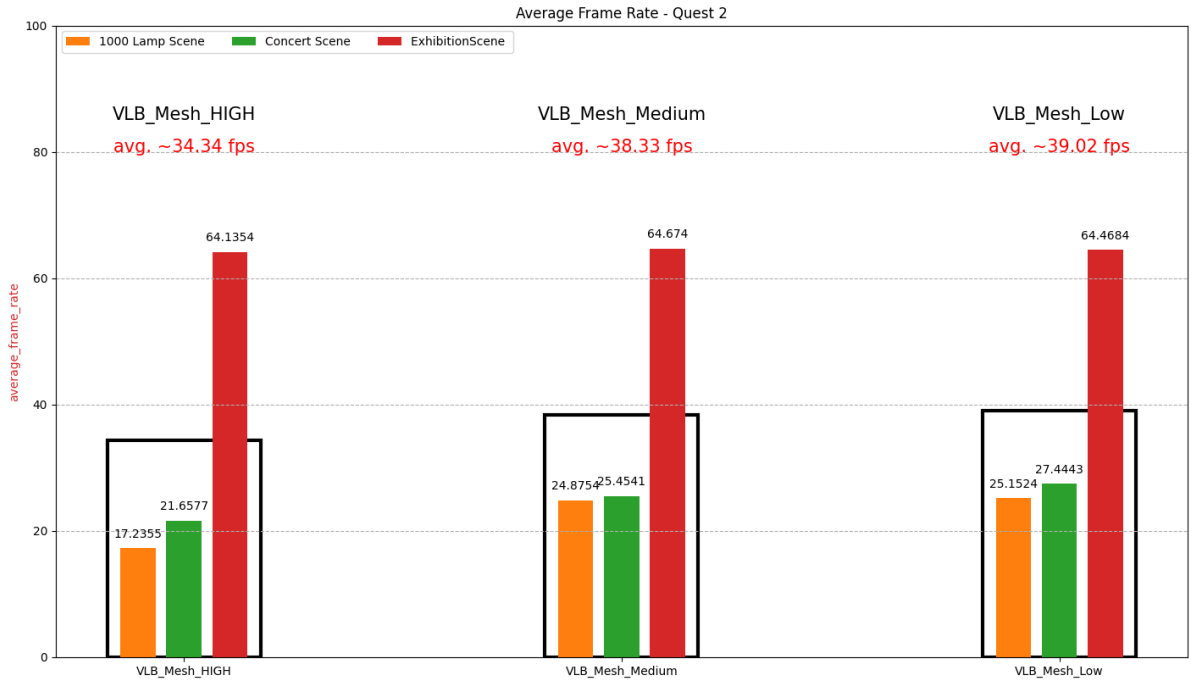

Figure 5.3: Comparing Mesh-Quality settings

Geometry Optimizations

Fig. 5.3 shows the improvements the reduction in vertices has on the performance of the application. Similar to the batching there is no real change in performance in the "Exhibition scene" as there are only 2 beams. The other scenes show minimal performance improvement.

Writing a Shader

Fig. 5.4 shows not only the difference between the VLB, the self-written shader, and the ShaderGraph Shader, but it also reveals the impact of the Fragment shader. The *alternative Versions* of the Shaders have no calculations done in the Fragment shader, noticeably improving performance. As the Vertex shader runs only on the vertices and our mesh has a small number of vertices, calculations done in the Vertex shader are not heavy on performance. But a Beam on low settings (from table 4.3), covering 1/4th of a 1920x1080 pixel screen, has to run through 50 vertex shader passes, but, as it is covering $(1920x1080)/4 = 518400$ pixels, the Fragment shader pass runs 518400 times. While opaque objects get blocked by other opaque objects, the transparent beams never block each other, making the performance gain especially visible in the 1000 Lamps scene as many beams are overlapping each other.
As the performance of this module rose with every prototype, the average fps of the first

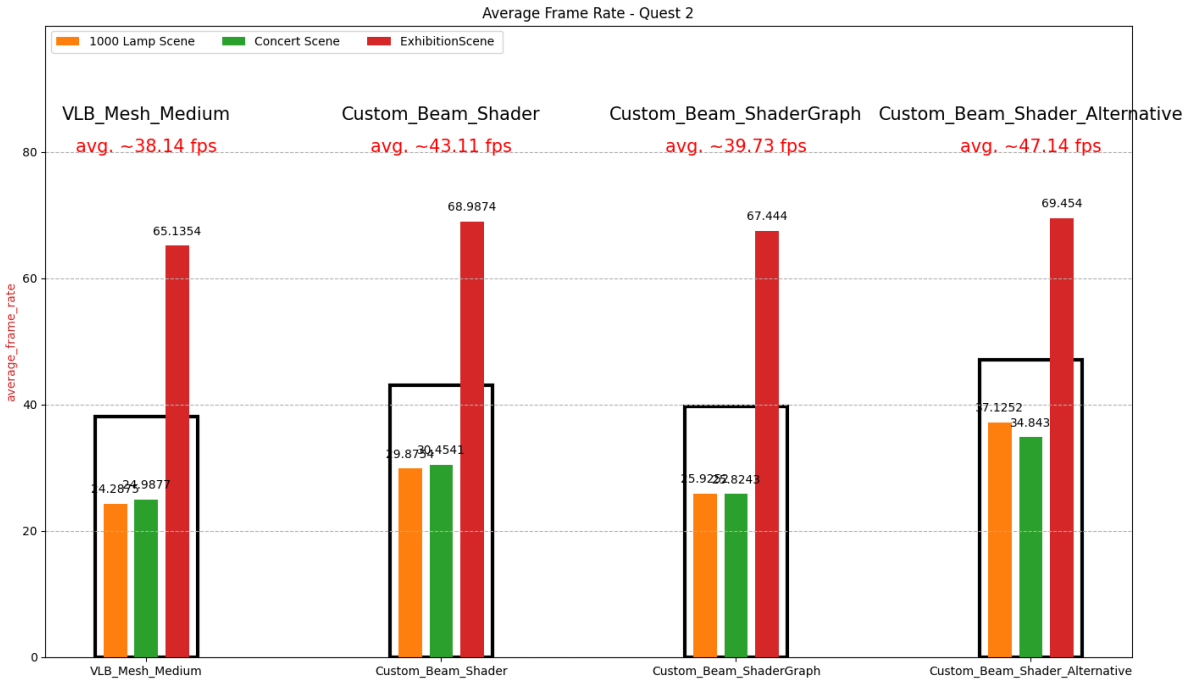

Figure 5.4: Comparing VLB with the self written Shader

two scenes is still very low. Very obvious is the very similar performance of both scenes when using the very lean, alternative self-written shader, even though the beam count is very different (1004 vs. 134). Using different analytics tools, no hint could be found where the processing power is wasted. Very time-intensive analysis with for example trace analysis (Google, 2023), and a more in-depth understanding of the hardware will be needed to understand the issues, falling out of the scope of this thesis (see chapter 6 Conclusion).

5.1.2 Gobo Projections

While developing the first prototype, a concern quickly arose: The decal shader, made to project small and medium-sized decals onto objects heavily uses the fragment shader pass. When these decals only fill a small amount of screenspace, this is not an issue. Bt the Gobo "Boxes" (Fig. 5.6) fill a large amount of screen space, so covering a large number of pixels. This leads to many calculations that need to be done per frame.

Fig. 5.5 shows the measurements taken with the Gobos, surprisingly debunking the concern. The Projections only have little impact on performance, ignorable in comparison with the impact of the beams.
The only hint was found inside the modified decal shader stating: "if ZWrite is Off, clip() is fast enough on mobile, because it won't write the DepthBuffer, so no GPU pipeline stall(confirmed

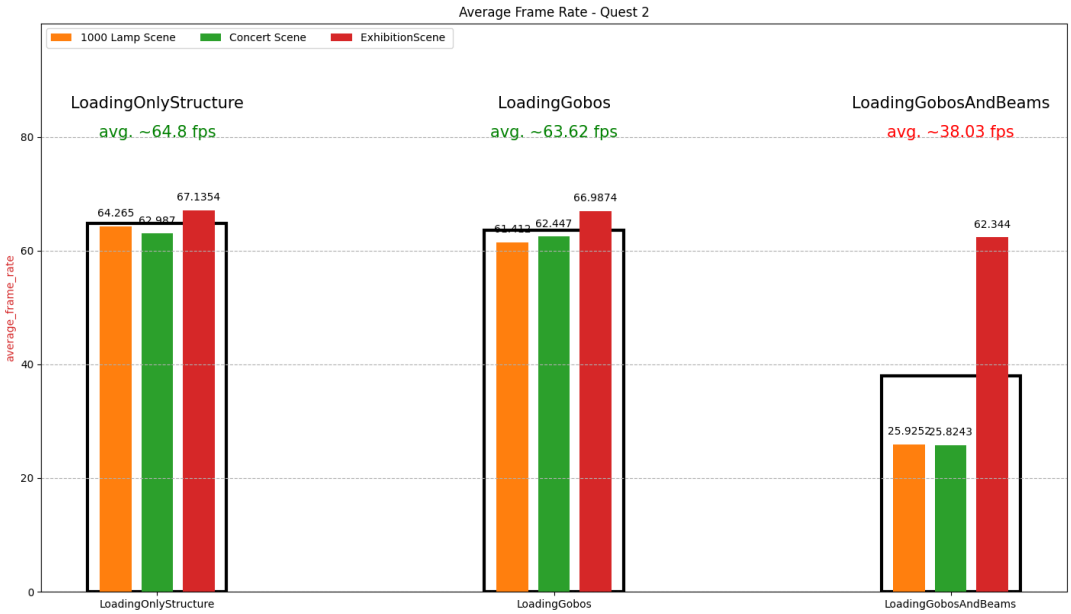

Figure 5.5: Comparing Impact of Gobos against Beams

by ARM staff)" (Leung, 2022). No further sources or details could be found regarding this very curios statement directly mentioning mobile hardware.

5.1.3 3D Models

Fig. 5.7 shows the Impact the loading and display of 3D Models has on the performance. The tests are run with no beams or projections loaded.

These results are, in contrast to the other ones, very much expected, very clearly showing that rendering more vertices leads to fewer frames per second. As fig. 4.14 already showed, the fixtures, even with as few vertices as possible stay very recognizable. This module satisfies its requirements. When using a high number of fixtures or fixtures that are fare away, one should disable the 3D models. Fixtures that are medium to close to the viewer can be just displayed with cubes and single fixtures or fixtures that are very close can be displayed with their supplied 3D models.

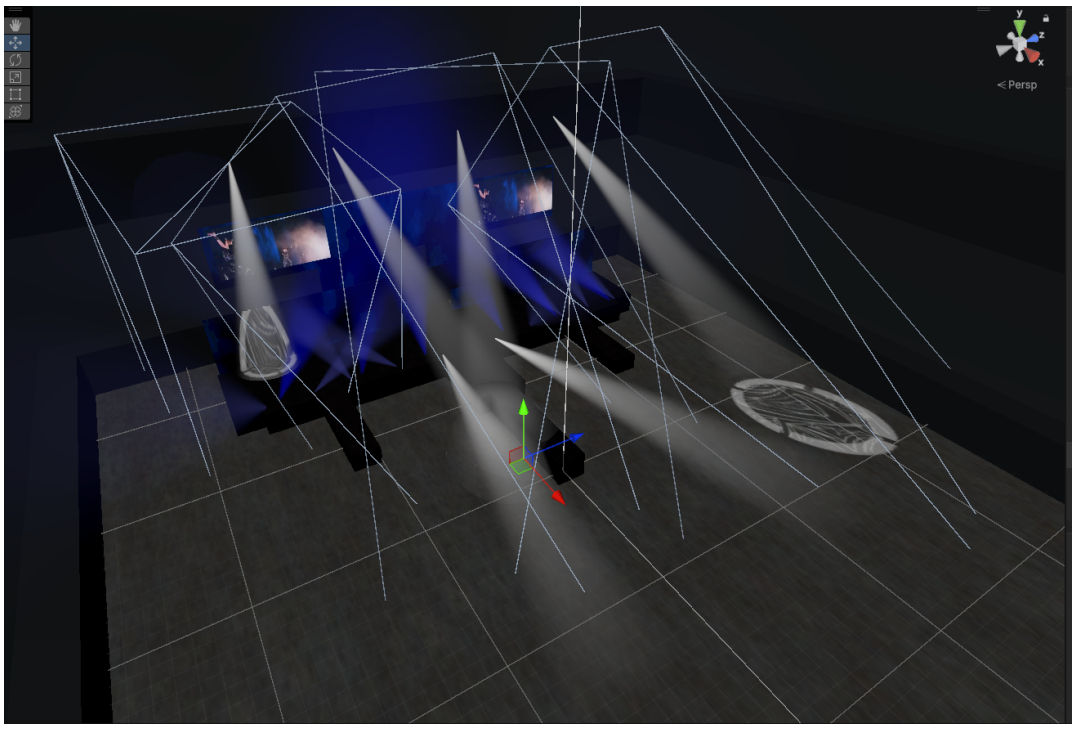

Figure 5.6: Four Fixtures with Gobo-Projection-Boxes made visible. Two Projecting a Gobo, two just the light circle.

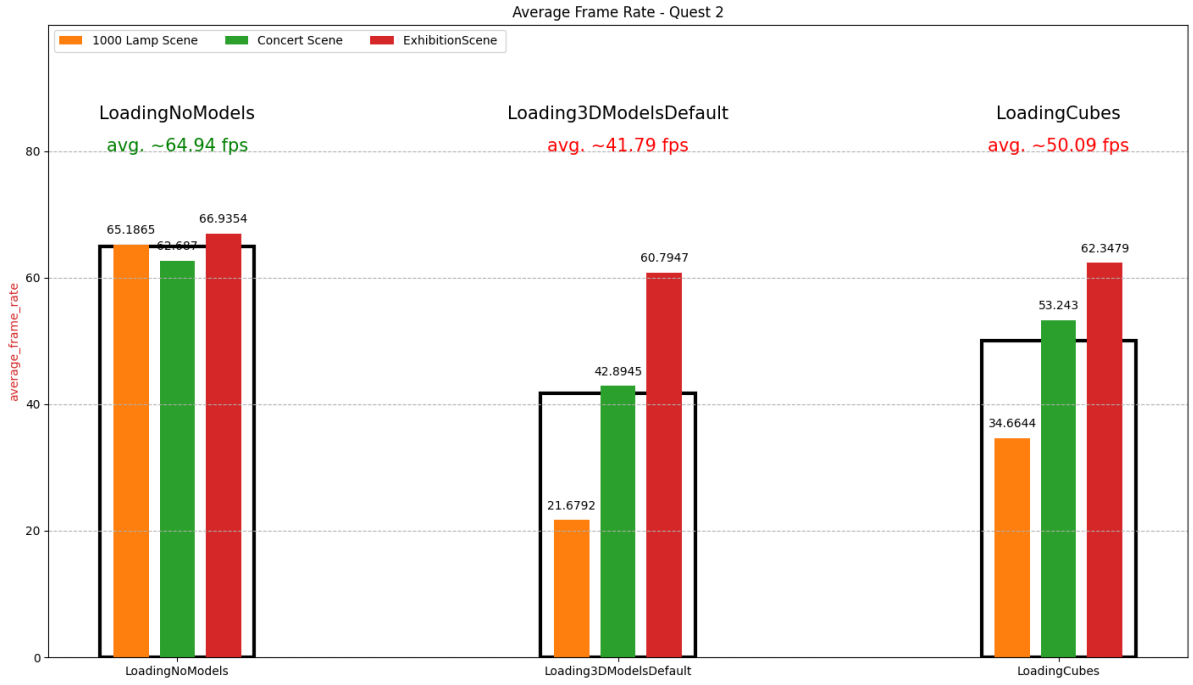

Figure 5.7: Comparing Impact of 3D Models and Cubes

6 Conclusion

Two patterns are recognizable throughout the work on this thesis:

1. Very different optimization results from PC and VR headset.
While it is expected that the VR headset runs on a much lower framerate than a high power Computer the results of optimization have a very different impact on the VR headset and the computer. As the internal architecture of these standalone headsets is vastly different from traditional computers for whom most tutorials and documentation are written, performance improvements visible on the computer have very different impacts on the VR headsets.
2. Unexplained bottlenecks on the VR headsets.
As discussed in detail in chapter 5.1.1, even after removing almost all features from the light beams, the fps would bottleneck at a certain point, well below the recommended value, hinting at a deeper limit of the hardware, not covered in any sources.

The thesis's main question, "How can an immersive light-visualizer experience, running on low-power virtual reality devices be created?", can only be answered partially: With this thesis, a lot of the foundational work has been implemented to make data of fixtures available, display models and structure, and control fixtures with professional hardware. Due to Unity Engines cross-device compatability, running this asset on powerful hardware is possible and shows the usability of the asset. To answer the question fully, further research is needed focusing on rendering limitations and exploring the exact implications of, for example, the Tile-based GPU architecture (Molnar, 1994).

6.1 Evaluating the Research Design

(Too) Fast Prototyping? Prototyping pushed the progress of this research. In the preceding chapters, it was demonstrated how evaluating requirements, adapting to issues, and changing the focus of development can push forward progress, where changes, the documentation, and the documentation of those changes are varied and doubtful. In the later stages of development, the progress and code that was created in the earlier cycles often was added upon and expanded. This was often made difficult by the quick, but not "clean" Code written in earlier prototype rounds, leading to the slowdown of progress (Martin and Coplien, 2009).

As additions required large changes, parts could not be reused but needed to be rewritten and new bugs were introduced, keeping the scripts and shader running was often more work than implementing new features or changes. Applying principles, for example, SOLID (Akritidis, 2023) would have made earlier development slower but could have sped up later research. Nonetheless, without this shortfall, a more complete answer to the thesis question would probably not have been found, as only this flexible method allowed each of the modules to quickly make enough progress to even discover the bottleneck.

6.2 Outlook

While the goal of this research could not be adequately reached, the Asset still has immense potential. In chapter 4.4 the optimization for Tile-based GPUs was already mentioned. Getting, for example, instancing to work, could be a big step toward better framerates on the standalone VR headsets. A deeper understanding of the limitations discovered in chapter 5.1.1 could also be able to make big performance gains possible. Additionally, many functions and ideas are collected in appendix 2, many of which are described in Staffel et al., 2022 and could be implemented with little effort.

Possible performance improvements with newer versions of Unity Engine and newer versions of the used Headsets have already been announced or released: A new Headsets from Meta (Quest 3) is already available, while the XR Headset made by Apple is to be released in Q1 of 2024 (Apple, 2024). While these headsets will certainly have more computing power than their predecessors, optimization will also be made easier. Features like eye-tracking, enabling foveated rendering (Meta, 2023), and potentially more comprehensive documentation.

6.3 Real world usage

As part of the “XRevent” project, a frontend for artists is created and szenes can be shared and played together via the World Wide Web. 3D Models can be uploaded and placed, sky and light can be changed and fixtures can be placed and configured (see Fig. 6.1 and 6.2). Some GDTF files are supplied on the server or more can be uploaded by the artist, making the fixture available to all users. Multiplayer with voice-chat is available making it a social experience.

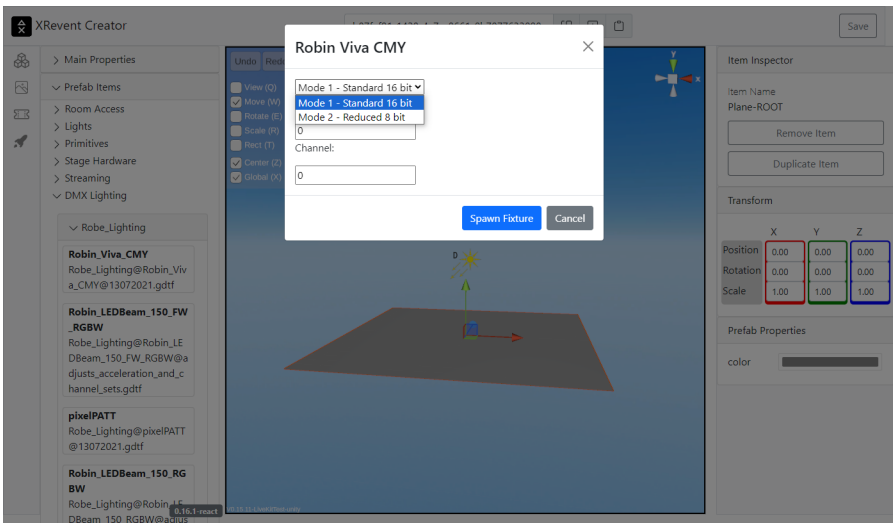

Figure 6.1: After loading a fixture, the user of the XRevent Creator can input all necessary details

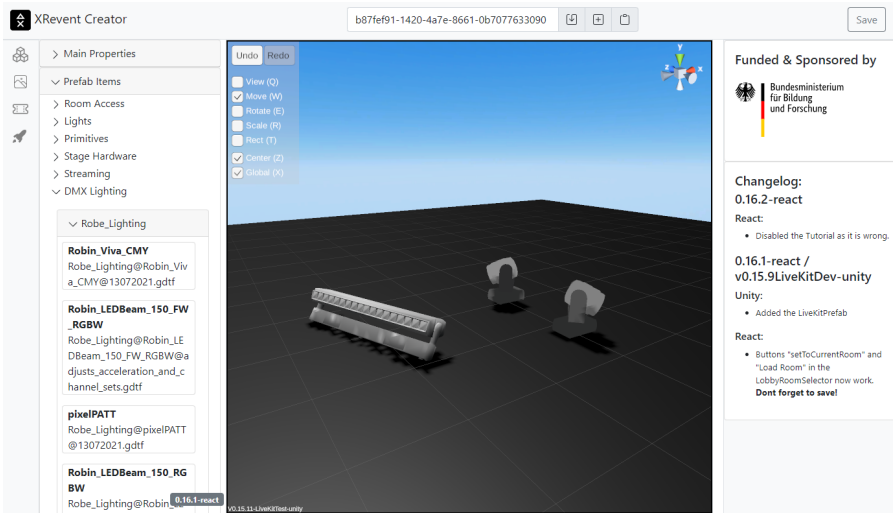

Figure 6.2: Three fixtures placed inside the virtual world of the XRevent Creator

Bibliography

Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension.

Akritidis, G. (2023, November). Solid: How to use it, why and when - c-sharp and unity development.

Apple. (2024). Apple vision pro - apple. Retrieved January 10, 2024, from https://www.apple.com/apple-vision-pro/

Arm Limited. (2021). Tile-based rendering. Retrieved September 20, 2023, from https://developer.arm.com/documentation/102662/0100/Tile-based-GPUs

arm.com. (2021, February). Optimizing unity games for arm – arm®. Retrieved September 19, 2023, from https://www.arm.com/resources/unity

Autodesk Ltd. (1996, May). 3d studio file format (3ds), document revision 0.1. https://www.graphicon.ru/oldgr/courses/cg2000s/files/3dsmli.html

Bantin, L. (2021, May). Realisierung eines lighting visualizers unter verwendung der 3d-engine unity.

Battista, F. D. (2016, May). 3ds loader runtime importer | tools | unity asset store. Retrieved February 4, 2024, from https://assetstore.unity.com/packages/tools/3ds-loader-runtime-importer-62536

BrainVoyager. (n.d.). Spatial transformation matrices. Retrieved March 15, 2023, from https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html

Brigsted, T. (2022, December). Github - siccity/gltfutility: Simple gltf importer for unity. Retrieved February 4, 2024, from https://github.com/Siccity/GLTFUtility

Bruce, G. (2022, April). Us: Charting the rise of indie video games.

Cho, S. (2018, November). Github - sugi-cho/artnet.unity: Artnet library for unity(c-sharp). based on an artnet library for c-sharp and vb.net developers. based on the acn project codebase.

Dasch, T. (2019). Pc rendering techniques to avoid when developing for mobile vr. Retrieved September 19, 2023, from https://developer.oculus.com/blog/pc-rendering-techniques-to-avoid-when-developing-for-mobile-vr/

deadmau5.com. (2021). Deadmau5: Exploring music's new digital reality | unreal engine. Retrieved February 4, 2024, from https://deadmau5.com/exploring-musics-new-digital-reality-unreal-engine/

Epic Games. (2022). Dmx overview | unreal engine 4.27 documentation.

Epic Games. (2023a). Dmx gdtf support in unreal engine | unreal engine 5.1 documentation. Retrieved February 4, 2024, from https://docs.unrealengine.com/5.1/en-US/dmx-gdtf-support-in-unreal-engine/

Epic Games. (2023b). Sony music's 'digital madison beer' sets the virtual concert world on fire | unreal engine - youtube. Retrieved February 4, 2024, from https://www.youtube.com/watch?v=-4ZXuaHRx30

Ferreira, C. (2019). How to optimize your oculus quest app w/ renderdoc: Quest hardware and software offerings. Retrieved September 19, 2023, from https://developer.oculus.com/blog/how-to-optimize-your-oculus-quest-app-w-renderdoc-quest-hardware-and-software-offerings/

Fok, W. (2020, August). Github - wason-fok/unity-dmx-fixture-library: Use artnet protocol to control fixtures in unity that are parsed from the gdtf library.

GDTF. (2022, November). Gdtf wiki. https://gdtf-share.com/help/en/help/gdtf_builder/key_dataformat.html

Gilbert, N. (2023). 74 virtual reality statistics you must know in 2023: Adoption, usage & market share - financesonline.com. Retrieved September 7, 2023, from https://financesonline.com/virtual-reality-statistics/

Ginsberg, C. (2023, September). Top 5 programming languages for data analysts | classes near me blog.

GmbH, S. (2023). Overview - syncronorm gmbh.

Google. (2023). Perfetto - system profiling, app tracing and trace analysis - perfetto tracing docs. Retrieved February 5, 2024, from https://perfetto.dev/docs/

Greule, R. (2021). Licht und beleuchtung im medienbereich. In *Licht und beleuchtung im medienbereich*.

GuinnessWorldRecords. (2023). World record: Largest music concert in a video game. *Guinness World Records*. Retrieved June 25, 2023, from https://www.guinnessworldrecords.com/world-records/563742-largest-music-concert-in-a-videogame

Hdouin, S. (2023, February). Update v5.3.0 : New ui, fixes for v51 and ovr metrics tool integration - quest games optimizer by anagan79.

Iett, D. (2022). *Building quality shaders for unity®*. Apress Berkeley, CA.

igolinin. (2021, June). Github - igolinin/dmxtools: Set of tools to simplify working with virtual and practical lights in unity3d. support for artnet in and usb dmx out. Retrieved September 19, 2023, from %5Curl%7Bhttps://github.com/igolinin/DMXtools%7D

Ilett, D. (2021, September). Decals & stickers in unity shader graph and urp. Retrieved February 4, 2024, from https://www.youtube.com/watch?v=f7iO9ernEmM

James, M. (2015, May). Github - mikecodesdotnet/artnet.net: An artnet library for c-sharp and vb.net developers. based on the acn project codebase.

Jeremiah. (2011, July). Understanding the view matrix | 3d game engine programming. Retrieved February 4, 2024, from https://www.3dgep.com/understanding-the-view-matrix/

Joshi, S. G. (2021, April). Prototyping in research - in5000 – qualitative research methods.

Khorikov, V. (2020). *Unit testing principles, practices, and patterns*. Manning. https://books.google.de/books?id=rDszEAAAQBAJ

Lague, S. (2021, November). I tried creating a game using real-world geographic data. Retrieved February 4, 2024, from https://www.youtube.com/watch?v=sLqXFF8mlEU

Leung, C. (2022, July). Github - colinleung-nilocat/unityurpunlitscreenspacedecalshader: Unity unlit screen space decal shader for urp. Retrieved February 4, 2024, from https://github.com/ColinLeung-NiloCat/UnityURPUnlitScreenSpaceDecalShader

Lever, N. (2022, April). Universal renderpipeline for advanced unity creators.

Maddock, C. (2022, July). Xml schema definition tool (xsd.exe) - .net | microsoft learn.

Martin, R. C., & Coplien, J. O. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

Maxine. (2023, March). Can i use assets from the asset store in my commercial game? – unity.

Melling, J. (2018). Vr world cup 2018 first step to widespread take-up - graphics, gaming, and vr blog - arm community blogs - arm community. Retrieved September 19, 2023, from %5Curl%7Bhttps://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/2018-world-cup-in-vr-could-be-a-useful-first-step-towards-widespread-takeup%7D

Meta. (2019). Monitor performance with ovr metrics tool: Unity | oculus developers.

Meta. (2022a). Best practices for rift and android | oculus developers. Retrieved September 19, 2023, from https://developer.oculus.com/documentation/unity/unity-best-practices-intro/

Meta. (2022b). Multisample anti-aliasing analysis for meta quest | oculus developers.

Meta. (2023, January). Eye tracked foveated rendering | oculus developers. Retrieved December 10, 2023, from https://developer.oculus.com/documentation/unity/unity-eye-tracked-foveated-rendering/

Microsoft. (2011, September). Zipfile klasse (system.io.compression) | microsoft learn.

Microsoft. (2022). .net documentation c-sharp coding conventions - c-sharp | microsoft learn. Retrieved February 4, 2024, from https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Milgram, P., & Kishino, F. (1994). A taxonomy of mixed reality visual displays. *IEICE TRANSACTIONS on Information and Systems*, *77*(12), 1321–1329.

Molnar, S. (1994). A sorting classification of parallel rendering. Retrieved September 19, 2023, from https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15869-f11/www/readings/molnar94_sorting.pdf

Moore, E. (2022). Designing vr games worth playing: 6 key considerations. Retrieved July 1, 2023, from https://www.toptal.com/designers/virtual-reality/designing-vr-games

Noveltech. (2022, July). Game engine fees & pricing in 2020 - unity, unreal engine, gamemaker studio, godot | noveltech.

Pitts, J., & Bourke, P. (1996, June). 3d-studio file format (.3ds) from autodesk ltd, document revision 0.91. http://paulbourke.net/dataformats/3ds/

Pluberus. (2013, January). Msi afterburner guide - performance recording. Retrieved February 4, 2024, from https://docs.google.com/document/d/15mq4aMUVxc7Vm6Iz2XyfU807o7dPdHYv9KiG35y edit?pli=1

Preece, J., Rogers, Y., & Sharp, H. (2015). *Interaction design: Beyond human-computer interaction* (4th ed.). Wiley.

Saladgamer. (2021, April). Optimization guidelines - volumetric light beam documentation.

Schechter, S. (2020, May). Gltf resources & tools | marxent. Retrieved February 4, 2024, from https://www.marxentlabs.com/gltf-files/#:~:text=What's%20a%20glTF%20file%3F, Microsoft%20Word%20or%20PowerPoint%20files.

Schiavone, L. (2023, December). Fog and haze basics – x-laser.

Skarbez, R., Smith, M., & Whitton, M. C. (2021). Revisiting milgram and kishino's reality-virtuality continuum. https://www.frontiersin.org/articles/10.3389/frvir.2021.647997.

Staffel, M., Berhorst, R., Krude, G., Müller, D., Valchář, J., Vanek, P., Powell, J., & Greenlee, R. (2022). Dinspec15800 d and ics35.240.99;97.200.10.

St-Laurent, S. (2004). *Shaders for game programmers and artists*. Thomson Course Technology PTR.

Swiniarski, S. (2022, November). C-sharp operators codecademy.

Tech Salad. (2022, May). Volumetric light beam sd - volumetric light beam documentation.

Tech Salad. (2023, December). Volumetric light beam | vfx shaders | unity asset store.

TenMedia. (2022). Software prototyping.

Unity Technologies. (n.d.). Unity - Manual: Render pipeline feature comparison. Retrieved March 15, 2023, from https://docs.unity3d.com/Manual/render-pipelines-feature-comparison.html

Unity Technologies. (2019a). Unity - Manual: Writing shaders overview. Retrieved March 15, 2023, from https://docs.unity3d.com/Manual/SL-ShadingLanguage.html

Unity Technologies. (2019b). Unity - scripting api: Time.deltatime. Retrieved January 2, 2024, from https://docs.unity3d.com/ScriptReference/Time-deltaTime.html

Unity Technologies. (2019c, August). Unity - manual: Shadows. Retrieved February 4, 2024, from https://docs.unity3d.com/550/Documentation/Manual/ShadowOverview.html

Unity Technologies. (2020). Unity - scripting api: Camera.depthtexturemode. Retrieved September 19, 2023, from https://docs.unity3d.com/ScriptReference/Camera-depthTextureMode.html

Unity Technologies. (2021a). Dmx gdtf support in unreal engine | unreal engine 5.1 documentation.

Unity Technologies. (2021b). Unity - manual: Cameras and depth textures. Retrieved September 19, 2023, from https://docs.unity3d.com/Manual/SL-CameraDepthTexture.html

Unity Technologies. (2021c). Unity - manual: Compute shaders. Retrieved February 4, 2024, from https://docs.unity3d.com/Manual/class-ComputeShader.html
Unity Technologies. (2021d, January). Changelog | shader graph | 17.0.2. Retrieved February 4, 2024, from https://docs.unity3d.com/Packages/com.unity.shadergraph@17.0/changelog/CHANGELOG.html
Unity Technologies. (2021e, April). Unity - manual: Single-pass instanced rendering and custom shaders. Retrieved February 4, 2024, from https://docs.unity3d.com/2021.3/Documentation/Manual/SinglePassInstancing.html
Unity Technologies. (2022a). Rendering and visual effects roadmap | unity. Retrieved September 19, 2023, from %5Curl%7Bhttps://unity.com/roadmap/unity-platform/rendering-visual-effects?utm_source=demand-gen&utm_medium=pdf&utm_campaign=empowering-creative-teams&utm_content=advanced-visual-effects-ebook%7D
Unity Technologies. (2022b). Unity platform rendering visual effects. Retrieved March 15, 2023, from https://portal.productboard.com/unity/1-unity-platform-rendering-visual-effects/tabs/3-universal-pipeline
Unity Technologies. (2022c, January). About shader graph | shader graph | 17.0.2. Retrieved February 3, 2024, from https://docs.unity3d.com/Packages/com.unity.shadergraph@17.0/manual/index.html
Unity Technologies. (2022d, January). Complex lit shader | universal rp | 13.1.9. Retrieved February 5, 2024, from https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@13.1/manual/shader-complex-lit.html
Unity Technologies. (2022e, January). Simple lit shader | universal rp | 13.1.9. Retrieved February 5, 2024, from %5Curl%7Bhttps://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@13.1/manual/simple-lit-shader.html%7D
Unity Technologies. (2022f, January). Unity - scripting api: Matrix4x4. Retrieved February 4, 2024, from https://docs.unity3d.com/ScriptReference/Matrix4x4.html
Unity Technologies. (2022g, March). Unity - manual: Scriptable render pipeline batcher.
Unity Technologies. (2022h, April). Unity - manual: Built-in shader include files. Retrieved February 4, 2024, from https://docs.unity3d.com/Manual/SL-BuiltinIncludes.html
Unity Technologies. (2022i, April). Unity - scripting api: Texture2d. Retrieved February 5, 2024, from https://docs.unity3d.com/ScriptReference/Texture2D.html
Unity Technologies. (2022j, July). Unity - manual: Stereo rendering. Retrieved February 5, 2024, from https://docs.unity3d.com/Manual/SinglePassStereoRendering.html
Unity Technologies. (2022k, August). Unity - manual: Model file formats.
Unity Technologies. (2023a). Unity - manual: Hlsl in unity. Retrieved February 4, 2024, from https://docs.unity3d.com/Manual/SL-ShaderPrograms.html
Unity Technologies. (2023b, January). Asset store terms of service and eula.
Unity Technologies. (2023c, January). Getting started with ray tracing | high definition rp | 14.0.10. Retrieved February 5, 2024, from https://docs.unity3d.com/Packages/

com.unity.render-pipelines.high-definition@14.0/manual/Ray-Tracing-Getting-Started.html
Unity Technologies. (2023d, June). Unity asset store submission guidelines - asset store.
Unity Technologies. (2023e, July). Unity - manual: Gpu instancing.
UTA. (2021a). *Forever changed:covid-19's lasting impact onthe entertainment industry* (tech. rep.). UTA,sightX,commune. Retrieved June 12, 2023, from https://unitedtalent.app.box.com/s/j4ijgxd8169jpgkxji8eqxnyqc7lszy2
UTA. (2021b). *Virtual + reality:the future of digital& live entertainment in apost-pandemic world* (tech. rep.). UTA,sightX,commune. Retrieved June 11, 2023, from https://unitedtalent.app.box.com/s/fcaha4xzcbvqtvcs3q9e03esmvhqaab1
Varcholik, P. (2014). *Real-time 3d rendering with directx and hlsl.* Addison-Wesley Professional.
Verttermann, T. (2021, June). Welche framerate für youtube? | 25p oder 30p? | motionside pictures®.
VPLT. (11, 2023). Home | gdtf share.
W. Robbins, R. (2014). Computer graphics and digital visual effects. Retrieved May 5, 2023, from https://scholarworks.calstate.edu/downloads/n296wz676
WaveXR. (2022, April). Past waves - wave.
Williams, S. (2015, March). How can i programmatically load a texture into an image the same way the unity editor does? - stack overflow. Retrieved February 5, 2024, from https://stackoverflow.com/questions/29034892/how-can-i-programmatically-load-a-texture-into-an-image-the-same-way-the-unity-e
XRToday. (2022). Into the innerverse: Inside bastille's first virtual concert. Retrieved June 5, 2023, from https://www.unrealengine.com/en-US/spotlights/into-the-innerverse-inside-bastille-s-first-virtual-concert
XRToday. (2023). Virtual reality statistics to know in 2023. Retrieved June 5, 2023, from https://www.xrtoday.com/virtual-reality/virtual-reality-statistics-to-know-in-2023/
Zgeb, B. (2021, February). Persistent data: How to save your game states and settings | unity blog.

Appendix

Codeblock 1: Appendix: The self written Shader

```
Shader "Shader␣Graphs/GDTF_Beam_ShaderGraph"
{
    Properties
    {
        [MainColor] _Color ("Color", Color) = (1.000000,0.000000,0.000000,1.000000)
        _FalloffSpeed ("FalloffSpeed", Range(0.100000,3.000000)) = 0.200000
        _Angle ("Angle", Range(5.000000,75.000000)) = 5.000000
        _fallofflength ("fallofflength", Range(0.100000,100.000000)) = 1.000000
        _intensity ("intensity", Range(0.000000,200.000000)) = 1.000000
    }
    SubShader
    {
        Tags
        {
            "QUEUE"="Transparent" "RenderType"="Transparent" "DisableBatching"="
    False" "RenderPipeline"="UniversalPipeline" "UniversalMaterialType"="Unlit"
        }
        Pass
        {
            Name "Universal␣Forward"
            Tags
            {
                "QUEUE"="Transparent" "RenderType"="Transparent" "DisableBatching"="
    False" "RenderPipeline"="UniversalPipeline" "UniversalMaterialType"="Unlit"
            }
            ZTest Less
            ZWrite Off
            Cull Off
            Blend SrcAlpha One

            CGPROGRAM
            #pragma vertex vert
```

```
31            #pragma fragment frag
32
33            float4 _Color;
34            float _fallofflength;
35            float _Angle;
36            float _FalloffSpeed;
37
38            #include "UnityCG.cginc"
39
40
41            struct appdata
42            {
43                float4 vertex : POSITION;
44
45                UNITY_VERTEX_INPUT_INSTANCE_ID
46            };
47
48            struct v2f
49            {
50                float4 vertex : SV_POSITION;
51                float4 objPos : TEXCOORD0;
52
53                UNITY_VERTEX_INPUT_INSTANCE_ID
54                UNITY_VERTEX_OUTPUT_STEREO
55            };
56
57            v2f vert(appdata v)
58            {
59                v2f o;
60
61                UNITY_SETUP_INSTANCE_ID(v);
62                UNITY_INITIALIZE_OUTPUT(v2f, o);
63                UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
64                o.objPos = v.vertex;
65
66                v.vertex.x *= tan(radians(_Angle))*(_fallofflength*0.5f)*(v.vertex.y
      +1);
67                v.vertex.z *= tan(radians(_Angle))*(_fallofflength*0.5f)*(v.vertex.y
      +1);
68
69                v.vertex.y += 1;
70
```

```
                v.vertex.y *= _fallofflength*0.5f;

                o.vertex = UnityObjectToClipPos(v.vertex);

                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i);

                _Color.w = (1-(i.objPos.y/_fallofflength))*_FalloffSpeed;
                return _Color;
            }
            ENDCG
        }
    }
}
```

Codeblock 2: and proposed [ ] features]Appendix: Implemented [X] and proposed [ ] features

```
GDTF Import:
        [x] Reading all parameter into one object
        [ ] Paramter (not a comprehensive list off all available parameters):
                [x] Dimmer
                [ ] Emitter Details
                [x] Axis
                [x] Zoom
                [x] Shutter
                [x] Strobo
                [x] Wheels
                [x] Gobo Projections
                [ ] Frost
                [ ] Focus
                [ ] Prisma
                [x] Color Wheels
                [x] Color Mixing
        [x] Beams
                [x] Volumetric Light Beams
                [x] Simple, mesh based, Shader
                [ ] Better Instancing on Tiled GPUs
                [ ] Fog
```

```
                [ ] Occlusion
                [ ] Optimizing multi emitter fixtures
        [ ] Dynamic Shader Loading
        [x] 3D Models
                [x] Reading the 3ds and GLTF files
                [x] Using just Cubes
                [ ] culling
                [ ] automatic analisies of meshes
                [ ] optimisation of meshes
                [ ] Materials
                [ ] GPU Instancing
        [x] Projections
                [x] Simple Projections
                [ ] Occlusion
                [ ] Rotation
                [ ] Multi Gobo Blending
                [ ] Frustum Optimization
                [ ] ?Shader Optimization?
        [x] DMX Calculations
                [x] ArtNet recieve
                [ ] Memory allocation and GC optimization
                [ ] sACN
                [ ] Built-in DMX Controller
        [ ] Multiplayer
        [ ]
```

Codeblock 3: Appendix: Python Code to generate the Graphs

```
import math
import os

import numpy as np
import pandas
import pandas as pd
from matplotlib import pyplot as plt


def Average(lst):
    return sum(lst) / len(lst)


# function to add value labels
```

```
15 def addlabels(axis, x, y, texts, colors):
16     if type(colors) is list:
17         if type(y) is list:
18             for i in range(len(x)):
19                 axis.text(i, y[i] + 4, texts[i], ha='center', color=colors[i],
    fontsize=15)
20         else:
21             for i in range(len(x)):
22                 axis.text(i, y, texts[i], ha='center', color=colors[i], fontsize=15)
23     else:
24         if type(y) is list:
25             for i in range(len(x)):
26                 axis.text(i, y[i] + 4, texts[i], ha='center', color=colors, fontsize
    =15)
27         else:
28             for i in range(len(x)):
29                 axis.text(i, y, texts[i], ha='center', color=colors, fontsize=15)
30
31
32 # function to filter a string array by include and exclude.
33 # exclude is hard filter! No item containing exclude will be in the array, even if
    it contains include.
34 # (eg: include = "Loading", exclude = "LoadingScreen" -> "LoadingScreen" will not be
     in the array)
35 def filterStringArray(array, include, exclude):
36     arrayCopy = array.copy()
37     array = []
38     for arrayItem in arrayCopy:
39         if include not in arrayItem:
40             continue
41         if exclude in arrayItem:
42             continue
43         array.append(arrayItem)
44     return array
45
46
47 ### Main ###
48 # find all relevant files in the directory
49 files = os.listdir(os.path.dirname(os.path.realpath(__file__)))
50 files = filterStringArray(files, ".csv", ".meta")
51 # sort the files by their name -> the files are named after the run they belong to
52 files.sort()
```

```
print(files)
# read the csv files
columns = ["Time Stamp", "available_memory_MB", "average_frame_rate", "Column1"]
dtypes = {"Time Stamp": int, "available_memory_MB": float, "average_frame_rate":
    float, "Column1": str}
dataFrames = []
for file in files:
    dataFrames.append(pd.read_csv(file, dtype=dtypes, usecols=columns))

# set up the plot 1 (average frame rate and available memory against time stamp) we
    generate a plot for each run
plt.rcParams["figure.figsize"] = [14.00, 8.00]
plt.rcParams["figure.autolayout"] = True
textPlacements = [-4000, -2000, 2000, 4000]
textPlacementIndex = 0

for df in dataFrames:
    fig, ax1 = plt.subplots()
    color = 'tab:red'
    ax1.set_xlabel('Time Stamp')
    ax1.set_ylabel('average_frame_rate', color=color)
    ax1.plot(df['Time Stamp'], df['average_frame_rate'], color=color)

    ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis
    color = 'tab:blue'
    ax2.set_ylabel('available_memory_MB', color=color)
    ax2.set_ylim([0, 1024])
    ax2.plot(df['Time Stamp'], df['available_memory_MB'])
    ax2.tick_params(axis='y', labelcolor=color)
    texts = []
    LoadingTimes = {}
    LoadedTimes = {}
    UnloadingTimes = {}

    # draw a vertical line when there is a string in Column1
    for i in range(len(df['Time Stamp'])):
        if pandas.isnull(df['Column1'][i]) is False:
            ax1.axvline(x=df['Time Stamp'][i], color='r', linestyle='-', linewidth
    =0.3)

            text: str = df['Column1'][i]

```

```
92            if text.__contains__("Loading") is True:
93                ax1.text(df['Time␣Stamp'][i] + textPlacements[textPlacementIndex] -
    2000, 10, df['Column1'][i],
94                         rotation=90,
95                         fontsize=11)
96
97                texts.append(df['Column1'][i])
98
99                if LoadingTimes.__contains__(df['Column1'][i]) is False:
100                   LoadingTimes[df['Column1'][i]] = df['Time␣Stamp'][i]
101
102           elif text.__contains__("Loaded") is True:
103               if LoadedTimes.__contains__(df['Column1'][i]) is False:
104                   LoadedTimes[df['Column1'][i]] = df['Time␣Stamp'][i]
105                   ax1.text(df['Time␣Stamp'][i] + textPlacements[textPlacementIndex
    ] + 6000, 10, df['Column1'][i],
106                            rotation=90,
107                            fontsize=11)
108
109           elif text.__contains__("Unloading") is True:
110               if UnloadingTimes.__contains__(df['Column1'][i]) is False:
111                   UnloadingTimes[df['Column1'][i]] = df['Time␣Stamp'][i]
112
113   """         textPlacementIndex += 1
114           if textPlacementIndex == 4:
115               textPlacementIndex = 0"""
116
117   plt.show()
118
119   ## Create a second plot where the x-axis ist the run und the y-axis is the
    average frame rate of that run
120
121   longTexts = []
122   AverageFrameRate = []
123   AverageFrameRateLong = []
124   # get the run from the texts
125   for i in range(len(texts)):
126       longTexts.append(texts[i].split(":")[1])
127       longTexts[i] = longTexts[i][:-4] + ":" + texts[i].split(":")[2]
128       longTexts[i] = longTexts[i][:-2]
129
130       texts[i] = texts[i].split(":")[2]
```

```
131         texts[i] = texts[i][:-2]
132 
133     # remove duplicates from texts
134     tempcopy = texts.copy()
135     texts = []
136     for item in tempcopy:
137         if item not in texts:
138             texts.append(item)
139 
140     # reformate the Text in the Dictionary
141     test = list(LoadingTimes.keys())
142     for key in test:
143         newkey = key.split(":")[1]
144         newkey = newkey[:-4] + ":" + (key.split(":")[2])
145         newkey = newkey[:-2]
146         LoadingTimes[newkey] = LoadingTimes.pop(key)
147 
148     test = list(LoadedTimes.keys())
149     for key in test:
150         newkey = key.split(":")[1]
151         newkey = newkey[:-4] + ":" + key.split(":")[2]
152         newkey = newkey[:-2]
153         LoadedTimes[newkey] = LoadedTimes.pop(key)
154 
155     test = list(UnloadingTimes.keys())
156     for key in test:
157         newkey = key.split(":")[1]
158         newkey = newkey[:-4] + ":" + key.split(":")[2]
159         newkey = newkey[:-2]
160         UnloadingTimes[newkey] = UnloadingTimes.pop(key)
161 
162     # Calculating the average frame rate for each run
163     for i in range(len(longTexts)):
164         currentAsString: str = longTexts[i]
165         # find the startTimestamp of the current run
166         startTimestamp = LoadedTimes[currentAsString]
167         # find the endTimestamp of the current run
168         endTimestamp = UnloadingTimes[currentAsString]
169 
170         Timestamps: list = list(df['Time␣Stamp'])
171         startIndex = Timestamps.index(startTimestamp)
172         endIndex = Timestamps.index(endTimestamp)
```

```
        calc = []
        for j in (df['Time Stamp'][startIndex + 3:endIndex]):
            calc.append(df['average_frame_rate'][df['Time Stamp'] == j].values[0])
            calc = [x for x in calc if not math.isnan(x)]  # remove nan values
        AverageFrameRateLong.append(Average(calc))

    # calculate the average of all runs for a performancesetting
    for i in range(0, len(longTexts), 3):
        calc = [AverageFrameRateLong[i], AverageFrameRateLong[i + 1],
    AverageFrameRateLong[i + 2]]
        AverageFrameRate.append(Average(calc))

    print(texts)
    print(AverageFrameRate)
    print(longTexts)
    print(AverageFrameRateLong)

    # set up the groups
    x = np.arange(len(texts))  # the label locations
    multiplier = 0

    # on ax1 the average of all three runs is shown
    fig, ax1 = plt.subplots(layout='constrained')
    # enable grid
    ax1.grid(True, which='both', axis='y', linestyle='--')

    # draw the Thicker Bars behind the thinner ones
    width = 0.35
    offset = 0
    rects = ax1.bar(x + offset, AverageFrameRate, width)

    color = []
    for i in range(len(rects)):
        rects[i].set_color('white')
        rects[i].set_linewidth(3)
        rects[i].set_edgecolor('black')
        c: color = 'green'
        c2: color = 'red'
        if rects[i].get_height() > 60:
            color.append(c)
        else:
            color.append(c2)
```

```
    AverageFrameRateRounded = []
    for i in range(len(AverageFrameRate)):
        AverageFrameRateRounded.append(round(AverageFrameRate[i], 2))
        AverageFrameRateRounded[i] = "avg.␣~" + str(AverageFrameRateRounded[i]) + "␣
    fps"

    addlabels(ax1, x + 0.3, 80, AverageFrameRateRounded, color)
    addlabels(ax1, x + 0.3, 85, texts, "black")

    # ax1.bar_label(rects, padding=15, color=color, fontsize=15)
    multiplier = -1
    width = 0.08  # the width of the bars
    spacing = 0.025
    # setup each of the three bars
    for i in range(3):
        tempListAvgFramerates = []
        for j in range(len(texts)):
            tempListAvgFramerates.append(AverageFrameRateLong[i + 3 * j])
        offset = (width + spacing) * multiplier
        rects = ax1.bar(x + offset, tempListAvgFramerates, width, label=longTexts[i
    ].split(":")[0])
        ax1.bar_label(rects, padding=7)
        multiplier += 1

    color = 'tab:red'
    ax1.set_ylabel('average_frame_rate', color=color)
    ax1.set_title('Average␣Frame␣Rate')
    ax1.set_xticks(x, texts)
    ax1.legend(loc='upper␣left', ncols=3)
    ax1.set_ylim(0, 100)

    plt.show()
```

Codeblock 4: Appendix:List of all tested GDTF files

```
    ADJ@Mega_TriPar_Profile_Plus@24122021.gdtf
    ADJ@Quad_Phase@24122021.gdtf
    American_DJ@32_HEX_Panel_IP@1.6.6_out_for_real_world_testing_-
    _updated_Virtual_Dim_Zone1.gdtf
    American_DJ@Jolt_300@v1.0.gdtf
    American_DJ@Mega_TRI_Bar@v1.0.0.gdtf
    ANDRE_GELEIA_LUZ@PAR_LED_RGBW_3W@PAR_LED_3W_RGBW_8_CANAIS_DMX_rev.gdtf
```

```
    ARRI@L5C@DMX_v3.5_13_Jan_2021.gdtf
    ARRI@Orbiter@DMX_v4.5_29_Dec_Reduced_Channel_Modes.gdtf
    ARRI@Orbiter@DMX_v4.5_29_Dec_Standard_and_ECC.gdtf
    ARRI@SkyPanel_S120C@DMX_v4.4_13_Jan_2021.gdtf
    ARRI@SkyPanel_S30RP@DMX_v4.4_13_Jan_2021.gdtf
    Astera_LED_Technology@AX3@V1.2.gdtf
    Astera_LED_Technology@AX5@V1.2.gdtf
    Astera_LED_Technology@AX5_TriplePAR@Astera_LED_Technology_AX5_TriplePAR.gdtf
    Astera_LED_Technology@Helios@V1.2.gdtf
    Astera_LED_Technology@Titan@V1.2.gdtf
    Ayrton@Arcaline_2_3G_1M@V1.21_UUID_changed.gdtf
    Ayrton@CosmoPix_R@V2.3_FonctionName.gdtf
    Ayrton@Ghibli@V2.12_-_Soft_V2XX_-_BladeCorrection.gdtf
    Ayrton@MagicBlade_FX@V2.11_-_3DReplaced.gdtf
    Ayrton@MagicBlade_R@V2.3_FonctionName.gdtf
    Basic_LED_PAR@Basic_LED_PAR@Basic_LED_PAR.gdtf
    Basic_Moving_Head@Basic_Moving_Head@Basic_Moving_Head.gdtf
    BlenderDMX@LED_PAR_64_RGBW@v0.3.gdtf
    BlenderDMX@Moving_Beam@v0.3.gdtf
    Blue_Show@Wash_RGBW_Zoom_19x15w@test25ch.gdtf
    B_Sound@Led_Blinder_2x100@2.150.gdtf
    Cameo@PIXBAR_650_CPRO@v0.3_fixed_30Ch_Strobe_-_removed_physics.gdtf
    Cameo@Q-Spot_15_W@1.0_Release.gdtf
    Cameo@ZENIT_W600@Original_Body_withDmx_Modes_and_defusor.gdtf
    Cameo@ZENIT_W600@Original_Body_with_full_Dmx_Modesv1.10.gdtf
    Cameo@ZENIT_Z120_G2@Beta_v.2.0.gdtf
    Cameo@ZENIT_Z120_G2_-_Outrigged@Beta_v.2.0.gdtf
    Chauvet_DJ@EVE-F50Z@Chauvet_DJ_EVE-F50Z.gdtf
    Chauvet_Professional@Maverick_Silens_2_Profile@Betav.1.8.gdtf
    Chauvet_Professional@Ovation_CYC_1_FC@Feb.
    _2021_based_on_Quick_Reference_Guide_Rev.4.gdtf
    China@32x18_RGBWAUV_18ch@0002.gdtf
    China_LED_RGBW_7CH@LED_PAR_RGBW@0001.gdtf
    Chroma-Q@Studio_Force_II_12@Version_1.4.gdtf
    CKC_Lighting@Aurora_Profile_V8@Aurora_Profile_V8_Standard.gdtf
    CLAYPAKY@ArollaProfileHP@ClayPaky_Official_File_Fw_Ver_1.8.gdtf
    CLAYPAKY@Arolla_Profile_HP@V1.0.gdtf
    CLAYPAKY@Arolla_Profile_MP@ClayPaky_Official_File_Fw_Ver_1.3.gdtf
    CLAYPAKY@Midi-B@V1_1.gdtf
    CLAYPAKY@Midi-B@V1_4.0.gdtf
    CLAYPAKY@MIni-B@ClayPaky_Official_File_Fw_Ver_2.6.gdtf
    CLAYPAKY@MiniXtylos@ClayPaky_Official_File.gdtf
```

```
    CLAYPAKY@Mini_Xtylos_HPE@ClayPaky_Official_File.gdtf
    CLAYPAKY@SharpyPlusAqua@ClayPaky_Official_File_Fw_Ver_1.4.gdtf
    CLAYPAKY@Sharpy_Plus@ClayPaky_Official_File_Fw_Ver_2.5.gdtf
    CLAYPAKY@XTYLOS@v1.1.gdtf
    CLAYPAKY@XtylosAqua@ClayPaky_Official_File_Fw_Ver_1.4.gdtf
    Clay_Paky@Sharpy_Plus@V.1.1_by_StefanoBigoloni.
    com_Standard_no_ch15_gobo_2_rotation_fine_it_is_in_progress.gdtf
    CLF_Lighting@Aorun@v1.5.gdtf
    CLF_Lighting@LEDBar_Pro@v1.0.gdtf
    CLF_Lighting@Odin@v1.3.gdtf
    Contest@Thunder_80@Ver._1.0.gdtf
    DTS_Lighting@Chrome_1_Bracket@REVISION_1.gdtf
    DTS_Lighting@Fos_100_Solo_FC@final.gdtf
    Ehrgeiz@Cobalt_Spot_Plus_5R@1.27.gdtf
    ELATION@Cuepix_Blinder_WW2@Cuepix_Blinder_WW2_Rev.1.gdtf
    ELATION@FUZE_PAR_Z120_IP@Beta_v.1.4.gdtf
    ELATION@KL_Fresnel_8_FC@1.1
    _Out_for_real_world_testing_added_CMY_and_CMY_Extended.gdtf
    ELATION@Smarty_Hybrid@1.3.2._out_for_real_world_testing_-_updated_highlight.gdtf
    Elation_Professional@DARTZ_360@v1.5.gdtf
    Elation_Professional@Proteus_Maximus@V1.gdtf
    Erik_Nelson_Entertainment@Diamond_Pro_Venue_460@V1.1.gdtf
    Erik_Nelson_Entertainment@Diamond_Pro_Venue_600@V2.1.gdtf
    Eurolite@LED_Bar_RGB_252_10@24122021.gdtf
    Eurolite@LED_PIX-144@0008.gdtf
    Eurolite@THA-100F@Release_1.0.gdtf
    Expolite@Tour-Cyc_540_RGBW@1.3.0_finished_29Ch_Mode_-_out_for_real_life_testing.
    gdtf
    Expolite@Tour-LED_50_XCR_invisible_beam@1.5.4_tested_in_real_life.gdtf
    Expolite@Tour-LED_Pro_28_CM_plus_W_Zoom_MKII@2.2._out_for_real_world_testing_-
    _fixed_Channel_Sets.gdtf
    Expolite@TourLED_Power_4CM-W@1.5_Release.gdtf
    Expolite@Tour_Blinder_400@Final.gdtf
    FINEART@FINE_1000L_BSWF@20210830.gdtf
    FINEART@FINE_1400ZL_PERF@20220118.gdtf
    FINEART@FINE_1800_PERF@20210826.gdtf
    FINEART@FINE_360ZL_BEAM@20210312.gdtf
    Flash_Butrym@BSW_LED_200@0.0.0.11.gdtf
    FOS_Technologies@Scorpio_BSW@2021-12-23.gdtf
    FUSION_by_GLP@FUSION_Stick_FS20@Beta_v.1.3.gdtf
    Futurelight@DMH-80@V2.8.gdtf
    GDTF_Template_test@Axis_Test@axis.gdtf
```

```
    Generic@12_Light@v1.0.gdtf
    Generic@4-lite_Blinder@1.2.3._out_for_real_world_testing_-_added_an_new_image.
    gdtf
    Generic@4_Cell_Blinder@4CellBlinderRev0.1.gdtf
    Generic@PAR_64_MFL_(CP62)@1.1.2._out_for_real_world_testing_-_added_emitter.gdtf
    Generic@PAR_64_NSP_(CP61)@1.1.2._out_for_real_world_testing_-added_emitter.gdtf
    Generic@PAR_64_VNSP_(CP60)@1.1.2._Out_for_real_world_testing_-_eded_emitter.gdtf
    GLP@JDC-1@FINISHED.gdtf
    GLP@JDC-1@Release_Candidate.gdtf
    i.Shine@3218_RGB@VER3.gdtf
    ICON_Germany@2_Fach_Blinder_COB_2x100W_WW@V2.0.gdtf
    JB-Lighting@JBLED_A7@1.7.0_updated_3D_Geometry_-
    _todo_check_shutter_in_gMA3_and_real_life.gdtf
    JB-Lighting@P12_Profile_HC@V_1.17.gdtf
    JB-Lighting@P12_Profile_HP@V_1.18.gdtf
    JB-Lighting@P12_Profile_WW@V_1.15.gdtf
    JB-Lighting@P12_Spot_HC@V_1.16.gdtf
    JB-Lighting@P12_Spot_HP@V_1.17.gdtf
    JB-Lighting@P12_Spot_WW@V_1.16 - Kopie.gdtf
    JB-Lighting@P12_Spot_WW@V_1.16.gdtf
    JB-Lighting@P12_Wash_HC@V_1.15.gdtf
    JB-Lighting@P12_Wash_HP@V_1.15.gdtf
    JB-Lighting@P12_Wash_WW@V_1.15.gdtf
    JB-Lighting@P18_MK2_Profile_HC@V_1.06.gdtf
    JB-Lighting@P18_MK2_Profile_HP@V_1.07.gdtf
    JB-Lighting@P18_MK2_Profile_WW@V_1.06.gdtf
    JB-Lighting@P18_MK2_Wash_HC@V_1.06.gdtf
    JB-Lighting@P18_MK2_Wash_HP@V_1.06.gdtf
    JB-Lighting@P18_MK2_Wash_WW@V_1.06.gdtf
    JB-Lighting@P18_Profile_HC@V_1.04.gdtf
    JB-Lighting@P18_Profile_HP@V_1.05.gdtf
    JB-Lighting@P18_Profile_WW@V_1.04.gdtf
    JB-Lighting@P18_Wash_HC@V_1.04.gdtf
    JB-Lighting@P18_Wash_HP@V_1.04.gdtf
    JB-Lighting@P18_Wash_WW@V_1.04.gdtf
    JB-Lighting@P7@V_1.19.gdtf
    JB-Lighting@P9_Beamspot_HP@V_1.05.gdtf
    JB-Lighting@Sparx18@V_1.37.gdtf
    JB-Lighting@Sparx18@V_1.38.gdtf
    JB-Lighting@Sparx30@V_1.02.gdtf
    JB-Lighting@Sparx_7@0.9_added_Mode_2_3.gdtf
    JB-Lighting@Varyscan_P3@Beta_v.1.13.gdtf
```

```
    LightGO@LED_UMBRELLA_145@1.05.gdtf
    LightGO@LED_UMBRELLA_145@2.013.gdtf
    Lightmaxx@Multi_Color_Spot_LED_RGBA@VER.2.gdtf
    Litecraft@AT10_Plus@v1.3.gdtf
    Luxibel@B_Narrow@1.2_Release.gdtf
    Marslite@4_x_Mini_LED_Moving_Head_RGBW_Bar@V.1.1_by_StefanoBigoloni.
    com_Multi_mode.gdtf
    Marslite@LED_BAR_12_PIX_RGB@V.1.1_Full_Mode_36_CH_Single_Pixel_RGB.gdtf
    Martin@Mac_Aura_XB@Updated_to_GDTF_1.1.gdtf
    Martin_Professional@ERA_300_Profile@20210326.gdtf
    Martin_Professional@ERA_600_Performance@20210513.gdtf
    Martin_Professional@ERA_600_Profile@20210513.gdtf
    Martin_Professional@ERA_800_Performance@20210513.gdtf
    Martin_Professional@ERA_800_Profile@20210513.gdtf
    Martin_Professional@Mac_2k_profile_II@Roughly_working.gdtf
    Martin_Professional@MAC_Aura_PXL@20211212.gdtf
    Martin_Professional@MAC_Ultra_Performance@20211214.gdtf
    Martin_Professional@MAC_Ultra_Wash@20211213.gdtf
    Martin_Professional@MAC_Viper_Performance@Rev_1.5.gdtf
    Martin_Professional@P3_Global@20210406.gdtf
    Martin_Professional@P3_Motion@20210406.gdtf
    Martin_Professional@RUSH_PAR_4_UV@Rev.0.gdtf
    Martin_Professional@VC-Grid_15_16x16@20210126.gdtf
    Martin_Professional@VC-Grid_25_8x8@20210125.gdtf
    Martin_Professional@VC-Grid_30_8x8@20210123.gdtf
    Martin_Professional@VC-Grid_60_4x4@20210122.gdtf
    Martin_Professional@VC-Grid_60_8x8@20210122.gdtf
    Martin_Professional@VC-Strip_15_16x1@20210126.gdtf
    Martin_Professional@VC-Strip_15_32x1@20210126.gdtf
    Martin_Professional@VC-Strip_25_16x1@20210125.gdtf
    Martin_Professional@VC-Strip_25_8x1@20210125.gdtf
    Martin_Professional@VC-Strip_30_16x1@20210123.gdtf
    Martin_Professional@VC-Strip_30_8x1@20210123.gdtf
    Martin_Professional@VC-Strip_60_4x1@20210122.gdtf
    Martin_Professional@VC-Strip_60_8x1@20210122.gdtf
    Martin_Professional@VDO_Atomic_Bold@20211206NoMeas.gdtf
    Martin_Professional@VDO_Atomic_Dot_CLD@20211209.gdtf
    Martin_Professional@VDO_Atomic_Dot_WRM@20211209.gdtf
    Martin_Professional@VDO_Dotron@20210407.gdtf
    Martin_Professional@VDO_Fatron_20_1000mm@20210127.gdtf
    Martin_Professional@VDO_Fatron_20_320mm@20201222.gdtf
    Martin_Professional@VDO_Sceptron_10_1000mm@20201223.gdtf
```

```
    Martin_Professional@VDO_Sceptron_10_320mm@20201223.gdtf
    Martin_Professional@VDO_Sceptron_20_1000mm@20201223.gdtf
    Martin_Professional@VDO_Sceptron_20_320mm@20201223.gdtf
    Martin_Professional@VDO_Sceptron_40_1000mm@20201223.gdtf
    Martin_Professional@VDO_Sceptron_40_320mm@20201223.gdtf
    Niethammer@HPZ115D@HPZ115D_V1.3.gdtf
    OXO_Light@Pixyline_18_fcw@1.8.7_out_for_real_world_testing_-_finished_36Ch_Mode.
    gdtf
    PRG@Icon_Stage@Ext_Mode_Only.gdtf
    Prizma_Lighting@PPL-P60@PPL-P60-1.gdtf
    Prolights@ArcPod15Q@Rev_1.4.gdtf
    Prolights@ArcPod27Q@Rev_0.8.gdtf
    Prolights@ArcPod48Q@Rev_1.6.gdtf
    Prolights@ArcPod96Q@Rev_1.31.gdtf
    Prolights@Aria700Profile@Rev_1.22_-_gobo_image_updated.gdtf
    Prolights@AstraBeam260IP@Rev_0.20_-
    _Real_fade_values_added_fix_focus_channel_sets.gdtf
    Prolights@AstraWash19PIX@Rev_0.65.gdtf
    Prolights@DisplayCobTRWDFC_60@Rev_1.02.gdtf
    Prolights@DisplayCobWW_30@Rev_0.4.gdtf
    Prolights@EclCTPlusXY26@Rev_0.1.gdtf
    Prolights@EclCyclorama050@Rev_1.9_-_emitters_CIE_xyY_updated.gdtf
    Prolights@EclCyclorama100@Rev_1.8_-_emitters_CIE_xyY_updated.gdtf
    Prolights@EclDisplayDAT2700Profile@Rev_1.2.gdtf
    Prolights@EclDisplayDAT2700Wash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayDAT2700Wash25-50@Rev_0.1.gdtf
    Prolights@EclDisplayDAT3000Profile@Rev_1.2.gdtf
    Prolights@EclDisplayDAT3000Wash15-30@Rev_0.1.gdtf
    Prolights@EclDisplayDAT3000Wash25-50@Rev_0.2.gdtf
    Prolights@EclDisplayDAT4000Profile@Rev_1.2.gdtf
    Prolights@EclDisplayDAT4000Wash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayDAT4000Wash25-50@Rev_0.1.gdtf
    Prolights@EclDisplayDAT5600Profile@Rev_1.2.gdtf
    Prolights@EclDisplayDAT5600Wash15-30@Rev_0.1.gdtf
    Prolights@EclDisplayDAT5600Wash25-50@Rev_0.1.gdtf
    Prolights@EclDisplayDATFCProfile@Rev_1.4.gdtf
    Prolights@EclDisplayDATFCWash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayDATFCWash25-50@Rev_0.2.gdtf
    Prolights@EclDisplayUN2700Profile@Rev_1.5.gdtf
    Prolights@EclDisplayUN2700Wash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayUN2700Wash25-50@Rev_0.2.gdtf
    Prolights@EclDisplayUN3000Profile@Rev_1.5.gdtf
```

```
    Prolights@EclDisplayUN3000Wash15-30@Rev_0.3.gdtf
    Prolights@EclDisplayUN3000Wash25-50@Rev_0.2.gdtf
    Prolights@EclDisplayUN4000Profile@Rev_1.4.gdtf
    Prolights@EclDisplayUN4000Wash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayUN4000Wash25-50@Rev_0.2.gdtf
    Prolights@EclDisplayUN5600Profile@Rev_1.3.gdtf
    Prolights@EclDisplayUN5600Wash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayUN5600Wash25-50@Rev_0.2.gdtf
    Prolights@EclDisplayUNFCProfile@Rev_1.2_-_Shutter_default_value_changed.gdtf
    Prolights@EclDisplayUNFCWash15-30@Rev_0.2.gdtf
    Prolights@EclDisplayUNFCWash25-50@Rev_0.2.gdtf
    Prolights@EclFresnel2KDY@Rev_1.1_-
    _physical_description_added_default_values_fixed.gdtf
    Prolights@EclFresnel2KTU@Rev_1.2_-
    _physical_description_added_default_values_fixed.gdtf
    Prolights@EclFresnel2KTW@Rev_1.93_-_emitters_CIE_xyY_updated_physical_desc.
    _added_default_value_fix_CTC_ch_fix.gdtf
    Prolights@EclFresnelDY@Rev_1.1_-_physical_description_added_default_values_fixed
    .gdtf
    Prolights@EclFresnelJDY@Rev_1.1_-
    _physical_description_added_default_values_fixed.gdtf
    Prolights@EclFresnelJTU@Rev_1.1_-
    _physical_description_added_default_values_fixed.gdtf
    Prolights@EclFresnelJTW@Rev_1.1.gdtf
    Prolights@EclFresnelTU@Rev_1.1_-_physical_description_added_default_values_fixed
    .gdtf
    Prolights@ECLFresnelTW@Rev_1.5_CIE_emitters_fixed.gdtf
    Prolights@EclFS_19@Rev_1.2_-_All_Modes_-_CTC_channels_fixed.gdtf
    Prolights@EclFS_26@Rev_1.2_-_All_Modes_-_CTC_channels_fixed.gdtf
    Prolights@EclFS_36@Rev_1.2_-_All_Modes_-_CTC_channels_fixed.gdtf
    Prolights@EclFS_50@Rev_1.2_-_All_Modes_-_CTC_channels_fixed.gdtf
    Prolights@EclGalleryProfileFC@Rev_0.4.gdtf
    Prolights@EclMiniFRFC@Rev_1.2_-_all_modes.gdtf
    Prolights@EclPanelTWC@Rev_1.47_-_FW_1.5_-_2.5.gdtf
    Prolights@EclPanelTWC@Rev_1.49_-_FW_3.0.gdtf
    Prolights@EclParDY_24@Rev_1.1_-_physical_description_added_default_values_fixed.
    gdtf
    Prolights@EclParFC_24@Rev_1.3_-
    _physical_values_added_default_values_fixed_CTC_channels_fix.gdtf
    Prolights@EclParTU_24@Rev_1.1_-_physical_description_added_default_values_fixed.
    gdtf
    Prolights@EclProfileCTPlusRGB19@Rev_1.2_-_physical_values_added_-_def.
```

```
    _values_fixed.gdtf
    Prolights@EclProfileCTPlusRGB26@Rev_1.3_-_physical_values_added_-
    _default_values_fixed.gdtf
    Prolights@EclProfileCTPlusRGB36@Rev_1.2_-__physical_values_added_-_def.
    _values_fixed.gdtf
    Prolights@EclProfileCTPlusRGB50@Rev_1.3_-__shutter_default_values_fixed.gdtf
    Prolights@EclProfileFWDY_19@Rev_1.3_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWDY_26@Rev_1.3_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWDY_36@Rev_1.2_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWDY_50@Rev_1.3_-_RDM_personality_ID_added.gdtf
    Prolights@EclProfileFWTU_19@Rev_1.1_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWTU_26@Rev_1.2_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWTU_36@Rev_1.2_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWTU_50@Rev_1.2_-
    _RDM_personality_ID_added_physical_description_added_default_value_fixed.gdtf
    Prolights@EclProfileFWVW_19@Rev_1.0.gdtf
    Prolights@EclProfileFWVW_26@Rev_0.2.gdtf
    Prolights@EclProfileFWVW_36@Rev_1.1_-_Emitters_updated.gdtf
    Prolights@EclProfileFWVW_50@Rev_1.0.gdtf
    Prolights@Lumipix15IP@Rev_1.0.gdtf
    Prolights@MiniEclWDDY_19@Rev_0.2.gdtf
    Prolights@MiniEclWDDY_26@Rev_0.2.gdtf
    Prolights@MiniEclWDDY_36@Rev_0.2.gdtf
    Prolights@MiniEclWDDY_50@Rev_0.2.gdtf
    Prolights@MiniEclWDTU_19@Rev_0.1.gdtf
    Prolights@MiniEclWDTU_26@Rev_0.1.gdtf
    Prolights@MiniEclWDTU_36@Rev_0.1.gdtf
    Prolights@MiniEclWDTU_50@Rev_0.1.gdtf
    Prolights@PixieWash@Rev_0.6_-_Shutter_default_value_fixed.gdtf
    Prolights@Polar3000@Rev_0.4.gdtf
    Prolights@RA2000ProfileHB@Rev_1.38_-_real_fade_value_added.gdtf
    Prolights@StudioCobDY_15@Rev_0.1.gdtf
    Prolights@StudioCobDY_30@Rev_0.1.gdtf
    Prolights@StudioCobDY_60@Rev_0.2.gdtf
    Prolights@StudioCobFC-30@Rev_1.1_-_Pigtail_added_Phisycal_values_added.gdtf
    Prolights@StudioCobPlusDY2_19@Rev_1.2.gdtf
```

```
    Prolights@StudioCobPlusDY2_37@Rev_1.3.gdtf
    Prolights@StudioCobPlusDY2_54@Rev_1.3.gdtf
    Prolights@StudioCobPlusTW_20@Rev_1.0.gdtf
    Prolights@StudioCobPlusTW_36@Rev_1.0.gdtf
    Prolights@StudioCobPlusTW_53@Rev_1.0.gdtf
    Prolights@StudioCobTU_15@Rev_0.1.gdtf
    Prolights@StudioCobTU_30@Rev_0.1.gdtf
    Prolights@StudioCobTU_60@Rev_0.1.gdtf
    Prolights@Sunrise2IP@Rev_1.1_-_Firmware_v._1.1.gdtf
    Prolights@TabledC@Rev_1.02.gdtf
    Pro_light@London_HAZE_1500_Pro@Finale_-_Ch_fog_as_a_dimmer_attribute.gdtf
    Robe_Lighting@pixelPATT@13072021.gdtf
    Robe_Lighting@Robin_300X_LEDWash@adjusts_channel_sets.gdtf
    Robe_Lighting@Robin_300_LEDWash@adjusts_channel_sets.gdtf
    Robe_Lighting@Robin_600X_LEDWash@adjusts_channel_sets.gdtf
    Robe_Lighting@Robin_800X_LEDWash@adjusts_channel_sets.gdtf
    Robe_Lighting@Robin_iPointe@11062021.gdtf
    Robe_Lighting@Robin_iSpiider@2023-04-06
    __Pattern_geometry_improvement__Flower_dimmer_revision.gdtf
    Robe_Lighting@Robin_LEDBeam_150_FW_RGBA@adjusts_acceleration_and_channel_sets.
    gdtf
    Robe_Lighting@Robin_LEDBeam_150_FW_RGBW@adjusts_acceleration_and_channel_sets.
    gdtf
    Robe_Lighting@Robin_LEDBeam_150_RGBA@adjusts_acceleration_and_channel_sets.gdtf
    Robe_Lighting@Robin_LEDBeam_150_RGBW@adjusts_acceleration_and_channel_sets.gdtf
    Robe_Lighting@ROBIN_LEDBeam_350_FW@13072021.gdtf
    Robe_Lighting@Robin_MegaPointe@13072021.gdtf
    Robe_Lighting@Robin_Pointe@04062021.gdtf
    Robe_Lighting@Robin_Tetra1@04062021.gdtf
    Robe_Lighting@Robin_Tetra2@04062021.gdtf
    Robe_Lighting@Robin_Viva_CMY@2022-11-24__SVG_thumbnail_added.gdtf
    Roxx@Show_Daylight_Medium@TW_A_29062021.gdtf
    Roxx@Show_Daylight_Narrow@TW_A_29062021.gdtf
    Roxx@Show_Daylight_Wide@TW_A_29062021.gdtf
    Roxx@Show_FC_Medium@TW_A_29062021.gdtf
    Roxx@Show_FC_Narrow@TW_A_29062021.gdtf
    Roxx@Show_FC_Wide@TW_A_29062021.gdtf
    Roxx@Show_Tungsten_Medium@TW_A_29062021.gdtf
    Roxx@Show_Tungsten_Narrow@TW_A_29062021.gdtf
    Roxx@Show_Tungsten_Wide@TW_A_29062021.gdtf
    Roxx@Show_TWplus_Medium@TW_A_25062021.gdtf
    Roxx@Show_TWplus_Narrow@TW_A_25062021.gdtf
```

```
    Roxx@Show_TWplus_Wide@TW_A_29062021.gdtf
    Selecon@Pacific_23-50@1.2._Out_for_real_world_testing_-_added_Picture.gdtf
    set@Ladder_Lights@SteveG.gdtf
    set@LightCage@SteveGiovanazzi.gdtf
    set@Scaffold@SteveG.gdtf
    SGM_Light@G-1_Beam@Betav.1.3.gdtf
    SGM_Light@Q-2@Release_1.3.gdtf
    SGM_Light@Q-7@Release_1.0.gdtf
    SGM_Light@Q-8@Rev_D.gdtf
    SHEHDS@LEDSpot90W_GDTF@0.0.1.4.gdtf
    SHEHDS@LED_FLAT_PAR_18x18W_RGBWAUV@v1.2.gdtf
    Silver_Star@Neptune_FX_Wash@v1.1.gdtf
    Starway@Megakolor@V1.0.gdtf
    Template@Basic_LED_PAR@v0.1.gdtf
    Terbly@ELV-G4-RGBW@Edited_by_Luke_Chikkala.gdtf
    Terbly@GLW1940@Offical.gdtf
    Terbly@GLW760@Offical.gdtf
    Terbly@GLZ100@Offical.gdtf
    Terbly@GLZ100X@Offical.gdtf
    Terbly@GLZ300_IP@Offical.gdtf
    Terbly@SL1200P@Offical.gdtf
    Terbly@U300@Offical.gdtf
    TipTop@RGBWAUV_18x18@VER.2.gdtf
    Varytec@EasyMove_150@24122021.gdtf
    WEINAS@BSW360@BSW_360.gdtf
    Yuexin_Lighting@YX-MH60@initial_commit.gdtf
```

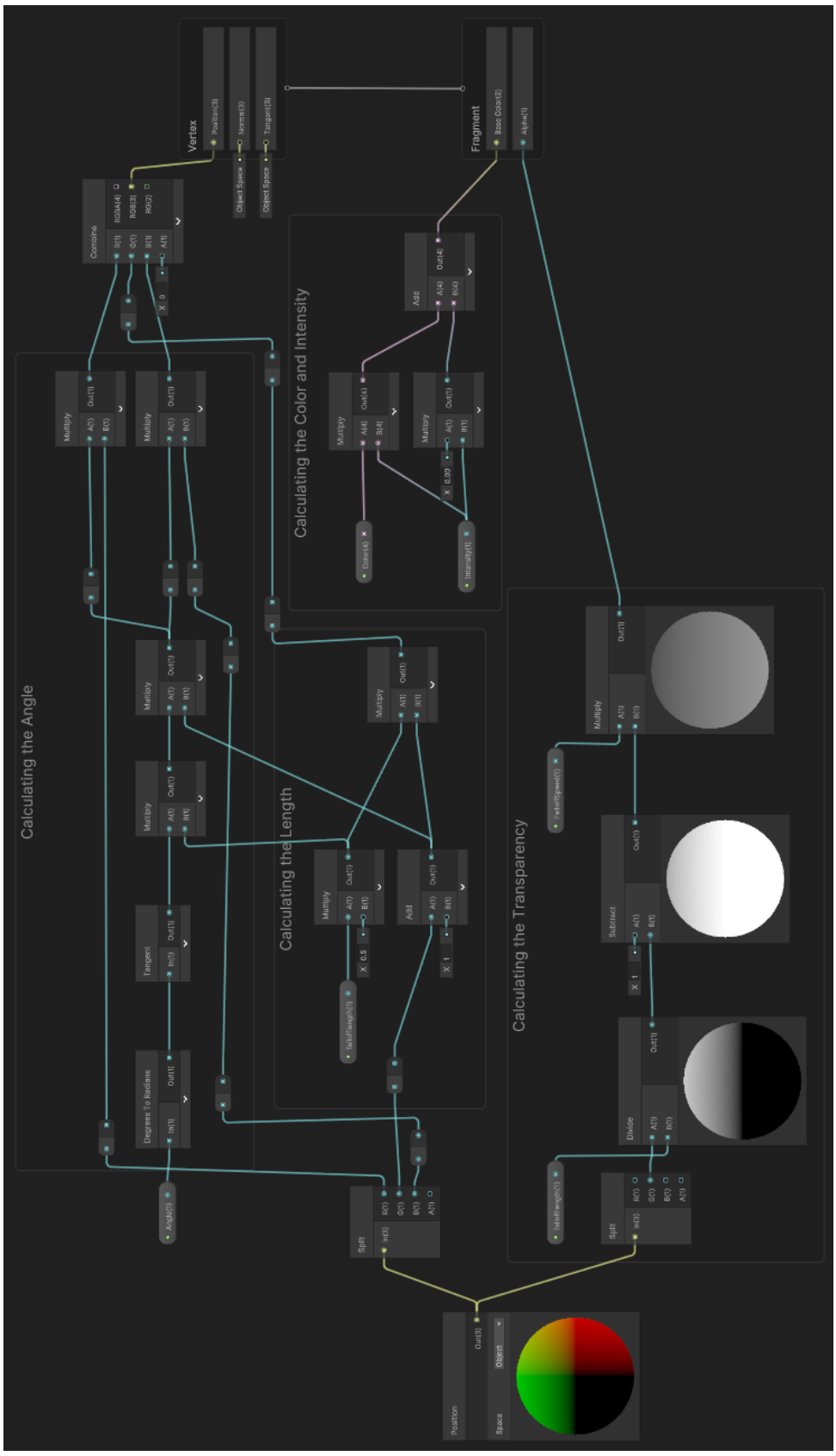

Figure 1: Appendix: Shader Graph made to resemble the VLB Beams

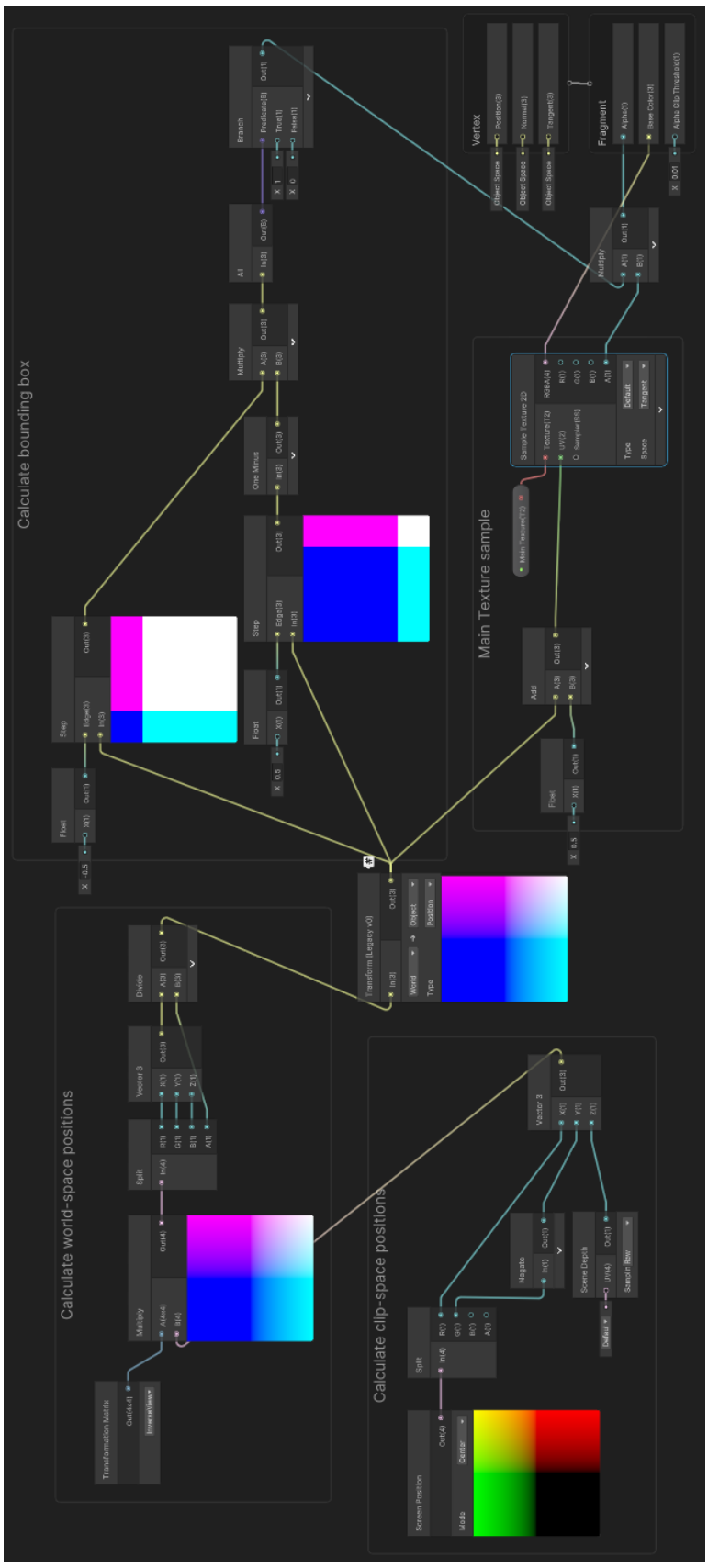

Figure 2: Appendix: ShaderGraph made by Ilett (Ilett, 2021)

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel

Visualizing Stage-Light Fixtures on Standalone Virtual Reality Headsets using the Unity Engine

selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Hamburg, 07. Februar 2024