Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Towards Real-Time 3D VR Simulation of Stained Glass Windows

Michael Makarenko

April 15, 2024

Supervisor: Dr. John Dingliana

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master in Computer Science (MCS)

Declaration

I, the undersigned, hereby declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Signed

____

Michael Makarenko

April 15, 2024

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Signed

____

Michael Makarenko

April 15, 2024

Acknowledgements

A number of people were crucial in the completion of this work, and I would like to express my gratitude to everyone involved.

Primarily, I thank my mother, Halyna Kushnir, for her unwavering support of all my endeavours, academic and professional.

I sincerely thank my supervisor, Dr. John Dingliana, for his invaluable academic guidance and feedback throughout the course of this dissertation, as well as for the provision of stained glass image files that were used in chapter three.

I would also like to thank my close friend, Adam, for his encouragement and helping me stay motivated.

And finally, a special thanks to Student Universal Support Ireland (SUSI), Trinity Access Programme (TAP), Higher Education Access Route (HEAR), and all the staff of Trinity College Dublin for making my higher education here over the past five years possible.

MICHAEL MAKARENKO

University of Dublin, Trinity College
April 2024

Abstract

Stained glass, a visually stunning craft dating from as early as the late Roman Empire, adorns the windows of both historical and modern architecture. Research across disciplines continues being conducted to maintain and digitise these complex works of art. Concurrently, newly developing 3D and VR technologies offer novel immersive and interactive means to experience objects in virtual environments. This dissertation casts a discerning light on the capacity of these computer technologies to greatly enhance the appreciation and preservation of such artefacts. We define a workflow for making digital recreations from photographs of existing stained glass windows and build a proof-of-concept application in the Unity 3D development engine, deployed to Google Cardboard VR. This prototype is assessed in the extent of its delivery of photorealism and a smooth user experience, running on a high-end Android smartphone released in 2021. Findings show efficient memory utilisation, substandard yet usable frametime performance for VR, and mixed results regarding visual fidelity, with marked differences between the rendered images on the development PC and mobile test device. In this manner, we perform a practical examination of the feasibility of real-time 3D VR for realistic simulations of stained glass windows. We conclude that while accessible mobile hardware does not yet meet the high computational demands of real-time 3D VR stained glass simulation, there is certain potential in more powerful and dedicated VR platforms and future mobile devices. Finally, we propose courses of action to refine and expand upon the procedure and project defined in this body of work, given the prospect for real-world applications in the fields of architectural simulation and preservation of cultural heritage.

Table of Contents

1 Introduction **1**
1.1 Background 1
1.2 Motivation 2
1.3 Research Objectives & Scope 2
1.4 Document Structure 3
2 Related Work **4**
2.1 Computer Graphics Overview 4
2.2 Literature Review 7
2.2.1 Generating Stained Glass Renders 7
2.2.2 Realistic Stained Glass Rendering 9
2.2.3 Digitisation of Stained Glass Windows 11
2.2.4 Applications of Virtual Reality 11
2.3 Summary 13
3 Methodology **14**
3.1 Design 14
3.2 Implementation 15
3.2.1 Setup 15
3.2.2 Scene 16
3.2.3 Shaders 18
3.2.4 From Image to Material 23
3.2.5 Post-Processing and Lighting 27
3.2.6 Interactivity 31
3.2.7 Deployment 40
4 Results **42**
4.1 Visual Fidelity 42
4.2 Performance 45
4.3 Ablation Study 48
5 Conclusion **50**
5.1 Contributions 50
5.2 Limitations & Challenges 50
5.3 Future Work 51
Bibliography **52**

List of Figures

2.1 Meshes of the same object with varying numbers of triangles. Image credit: 3D Digital Recording of Archaeological, Architectural and Artistic Heritage[13] Fig. 2 5

2.2 Visualisations of the effects of normal mapping. Image credits: Normal Mapping by Nikhil Kowshik[14] (top), Learn OpenGL by Joey de Vries[15] (bottom) 5

2.3 High-level visualisation of the graphics pipeline with programmable steps highlighted in blue. Image credit: Learn OpenGL by Joey de Vries[15] 6

2.4 Simplified overview of the relationship between models and shaders sketched in MS Paint 6

2.5 Source input image (left) and rendered output image (right) using Mould's method. Image credit: A Stained Glass Image Filter[16], Fig. 8 7

2.6 Source input image (left), target style input image (centre), and rendered output image (right) using Brooks' method. Image credit: Image-Based Stained Glass[17], Fig. 15 8

2.7 Source input image (left) and rendered output image (right) using the method by Setlur et al. Image credit: Automatic Stained Glass Rendering[18], Fig. 1 8

2.8 Source input image (left) and rendered output image (right) using the method by Seo et al. Image credit: Stained Glass Rendering with Smooth Tile Boundary[19], Fig. 3 8

2.9 Source input image (left) and rendered output image (right) using the method by Doyle et al. Image credit: Painted Stained Glass[20], Fig. 3 and Fig. 6 8

2.10 A sample frame of the source input video (top-left) and three frames of the rendered output video, using the method by Kang et al. Image credit: Video-based Stained Glass[21], Fig. 9 and Fig. 10 9

2.11 Rendered stained glass squares using Shin's method. Image credit: Modelling Stained Glass[22], Fig. 6 9

2.12 Resulting stained glass render with various viewing directions (top row) and light source directions (bottom row) using the method by Kim et al. Image credit: A Realistic Illumination Model for Stained Glass Rendering[23], Fig. 5 10

2.13 Stained glass window relighting results from three different source video input frames using the methods presented by Thanikachalam et al. Image credit: VITRAIL: Acquisition, Modelling, and Rendering of Stained Glass[24], Fig. 19 11

3.1 Module selection window in Unity Hub during Unity Editor install process ... 16

3.2 Sample empty scene view in Unity Editor 16

3.3 Searching for and installing ProBuilder in the Unity Package Manager 17

3.4 Sample bare interior environment created with ProBuilder in Unity 17

3.5 Prototype environment after importing free Unity Asset Store assets. Visible assets include “Lemon Trees” by Numena, “Grass Flowers Pack Free” by ALP, and “NoirMat – Noir Marble Pack Vol. 01” by Noir Project 18

3.6 Wind Shader Graph (WindShader) as implemented in Unity 20

3.7 The shader input parameters we expose to the end user. We configure the UI appearance and minimum/maximum/default values in Shader Graph 20

3.8 Stained Glass Shader Graph (SG_Shader) as implemented in Unity 22

3.9 User-facing shader input parameters for a sample stained glass window 22

3.10 Prototype environment after adding windowpanes with SG_Shader, each using differently configured user input parameters 22

3.11 A thresholding tool’s area of effect on a sample stained glass window image to create an appropriate diffuse texture. Note the imperfect selection on the more intricate parts of the window 23

3.12 Zoomed in before and after the use of a thresholding tool 24

3.13 Using a brush tool to manually clean up the results of a thresholding tool 24

3.14 Poor results in using a basic greyscale filter to generate a normal map 25

3.15 Improved resulting normal map generation using a thresholded image 25

3.16 Image-to-material pipeline (left) and the two resulting materials as previewed in Unity Editor on default spherical meshes (right) 26

3.17 Prototype environment after applying the processed texture and normal map to the SG_Shader materials and creating their respective shadow mask materials 26

3.18 Normal map provides a false sense of depth for the individual panes of stained glass making up the window, observable when light shines at a steep angle (left). Screenshots differ only in sun position 27

3.19 Before (top) and after (bottom) applying bloom, vignette, and ACES tonemapping post-processing. Visible assets include “Street Lamps 2” by SpaceZeta 28

3.20 Before (left) and after (right) applying a generic coloured light cookie texture. Like the distortion texture, it is generic and common to all three windows in the prototype 29

3.21 Attempting to apply one of the window images as a light cookie applies it repeatedly to all light cast by the sun, leading to uncanny and unrealistic results 29

3.22 Before (left) and after (right) generating lightmap data using Unity’s baked indirect global illumination model. All other previously discussed techniques are used in both images. The bluish tint seen on the left image is the result of ambient light as configured in Unity’s lighting settings 30

3.23 Screenshots in GCVR of an unselected (top) and selected (bottom) teleport platform 36

3.24 The primary Unity Editor windows used in an animation workflow 38

3.25 The five teleport platforms, two elevation buttons and three sun buttons 38

3.26 Game View in the Occlusion Mesh render mode provided by MockHMD 40

4.1 One of the three stained glass windows in our prototype (left) and two photographs (centre, right) of stained glass windows by Unknown on their public web blog, highlighting the coloured shadows of stained glass 42

4.2 Comparing our prototype's rendering of stained glass coloured shadows occluded by vegetation shadow (left), and a photograph of the real-world occurrence by Katja Linders on Pinterest (right) 43

4.3 Comparing our prototype's rendering of a stained glass window illuminated by sunlight exhibiting bloom/overexposure (left), and a real-world example photographed by a deleted user on an archived Reddit post (right) 43

4.4 Three GCVR screenshots displaying more pixelated shadows and lower lighting contrast/dynamic range 44

4.5 Investigating the worst frametime spikes in the Unity Profiler window while the application runs on the target device. Unity's Profiler process is revealed to consume over 21% of the frametime of some frames. Note that the GPU and CPU are integrated on mobile chips, so GPU usage is also reported as CPU usage 46

4.6 Frametimes in Unity Profiler are shown to be worse when observing the stained glass windows (top) and better when looking away towards the walls of the interior (bottom), revealing the most performance-intensive parts of our scene to be the windows and what is visible beyond them 46

4.7 The memory section of the built-in Profiler 47

4.8 Comparing two different snapshots of the test device's memory in the Memory Profiler. Note that "Total Allocated" memory as labelled here is defined differently from that seen in Fig. 4.7, can safely exceed available device memory, and is not indicative of application memory requirements 47

List of Tables

3.1 High-level comparison of Unity render pipelines by their key features 15

4.1 Comparing performance of four scenes with varying Unity Objects present or missing. “Full” is the control prototype scene with no removed Objects, “Post & Sun” combines the removals of both “No Exterior” and “No Interior”, and “Empty” removes even the directional light and post-processing present in “Post & Sun”. Data is obtained using Unity Profiler and Unity Memory Profiler at application runtime, with frametimes converted to framerates rounded to the nearest frame per second. Memory usage rounded to the nearest megabyte 49

1 Introduction

This paper's main objective is to explore the possibilities of simulating authentic stained glass window lighting in interactive virtual reality. But before delving into the specifics, this chapter explores the background and motivations that underpin the work conducted here, offering a comprehensive understanding of the context and driving forces behind this study.

1.1 Background

Stained glass windows have been a prominent feature in architectural design revered for their intricate motifs and vibrant colours for centuries, providing not only aesthetic value but also notable cultural and historical significance. The art of creating stained glass windows dates back to ancient civilisation, with "coloured glass windows" described as early as in the 3rd century CE in the early Christian basilicas of Rome[1]. However, it was during the Middle Ages in Europe that stained glass windows arguably reached their peak, adorning the walls of churches and cathedrals with religious and narrative scenes. These windows served not only as mere daylight sources or decorative art pieces, but also the practical purpose of conveying spiritual and cultural stories to the illiterate masses of medieval times.

Today, stained glass windows continue to be a significant element in buildings of all kinds, from the ecclesiastical and traditional to the decorative and modern. However, over time, these windows may deteriorate due to a variety of causes such as weathering, vandalism, and improper maintenance or lack thereof. Such factors inevitably alter the physical and optical properties of the glass, as modelled by Verney-Carron et al. in their paper[2]. The safe cleaning and restoration of these fragile pieces also tends to be rather involved, with ongoing research into novel cleaning methodologies such as that by Maingi et al. fairly recently[3]. Fortunately, the use of computer technology in the fields of art and architecture has greatly enhanced the way we experience and interact with the built environment. We can observe this interweaving with the field of computer science, particularly in areas such as human-computer interaction, architectural simulation, and computer graphics, which shall be explored in this dissertation.

1.2 Motivation

Existing two-dimensional (2D) solutions for digitally recreating and relighting stained glass windows fall short in authentically capturing the elaborate details and colours of the original artwork, as seen by the human eye from various angles in a three-dimensional (3D) environment, due to the inherent limitations of viewing a lone 2D plane. The advent of modern 3D graphics and virtual reality (VR) technologies has opened up a new realm of possibilities for simulating stained glass windows. By providing a more immersive and interactive experience, these technologies offer the potential for a more realistic and detailed depiction of the original artwork, allowing viewers to digitally perceive the windows in a way that was never before possible.

A number of potential practical use cases exist here. One example we can imagine, is providing a virtual tour in VR using not merely static 360° photography, as seen on The Stained Glass Museum's website[4], but with immersive real-time changes in environmental lighting such as the movement of the sun or swaying of trees behind the windows that vastly impacts the appearance and perception of stained glass.

Despite the promises of real-time 3D graphics and VR, their use in the simulation and digitisation of real-world stained glass windows is still a developing and underexplored area. There is a notable lack of research and literature on their application in this specific context. It is therefore necessary to conduct a comprehensive study to assess the full capabilities and limitations of these approaches. In doing so, we can better understand how modern computer technology can be employed in a way that truly captures the beauty and intricacy of stained glass windows, ultimately enriching our appreciation and understanding of these cultural and historical artworks.

1.3 Research Objectives & Scope

The primary objective of this dissertation is to simulate approximate yet high-fidelity stained glass window lighting in real-time 3D using photo image data and deploy the simulation to interactive VR. In the process, we hope to define a workflow that may be used and extended in the future and evaluate the results that this workflow yields by profiling the developed application. In doing so, this research aims to contribute to the expanding body of literature in the field of architectural simulation and

preservation. By examining the potential of real-time 3D VR technologies in this setting, we hope to inspire further advancements and developments in this area.

This work is primarily explorative in nature, and its scope is therefore limited to creating a demonstrable proof-of-concept prototype that achieves the aforementioned objectives, and not a final polished product. We will focus primarily on performant real-time techniques with an eye on accessible, cross-platform, interactive, real-time VR, not limited only to running on powerful new desktop PCs. As such, expensive techniques such as raytracing will not be considered within the scope of this study.

1.4 Document Structure

The subsequent chapter of this thesis, chapter 2, provides necessary contextual information and reviews existing literature that is related to this body of work, presenting a perspective of how this piece fits into the wider field of others related to it. In chapter 3, we will cover the methodology and design of the 3D VR solution developed as part of this research. In chapter 4 thereafter, we shall critically evaluate the results of our work, before concluding the thesis with the closing chapter 5, where we summate the contributions and limitations of this dissertation and propose future work that may be undertaken from here on.

2 Related Work

Before performing a comprehensive review of existing works associated with the subject matter, this chapter provides a concise outline of pertinent computer graphics terminology in order to ensure utmost clarity for the remainder of this dissertation.

2.1 Computer Graphics Overview

Rendering, in this context, is the process of synthesising a 2D image to be displayed on-screen from 3D data[5] by way of a framework known as the computer graphics pipeline or rendering pipeline[6]. This work is done through the use of a computer program known as a renderer, graphics engine, rendering system or similar[7]. The 3D data is known as a scene and the individual 3D objects that compose it are referred to as models. Since models are collections of data, and can be composed of a subset of models, the scene is technically a model itself composed of all 3D objects that exist within and is sometimes referred to as the model[8]. The final image can be either rendered in real-time in a matter of milliseconds or pre-rendered ahead of time, depending on whether or not real-time interactivity is a concern; thus, our focus lies in the former category of the two. In latter sections, we encounter the terms “raytracing” and “photon mapping”; these are global illumination techniques that simulate highly realistic lighting that have until quite recently been constrained to the domain of pre-rendering due to their high computation cost[9]. Practical real-time raytracing currently requires powerful and modern dedicated graphics processing hardware. Global illumination is discussed in a more specific context in section 3.2.5.

3D models are composed of a polygon mesh and, optionally, a material. The mesh describes the geometry of the model by defining vertices, edges, and faces, usually as triangle primitives, as well as normal vectors perpendicular to the individual surfaces of the mesh. The material describes the surface appearance of the model by defining a variety of properties that affect its interaction with light, such as colour. Some material properties may be stored as 2D images called textures or maps, such as the base colour texture or normal map, to have the properties vary at different points on the mesh[10]. The base colour texture, also referred to as the diffuse or albedo texture, describes the unmodified colour of the surface before taking any lighting considerations into account. Colour values here are typically encoded as either RGB

or RGBA, standing for the individual 8-bit Red, Green, Blue, and Alpha values respectively, where Alpha determines not colour but opacity[11]. The normal map is a texture that is used to modify the mesh's normals, which are used in lighting calculations, to give a false sense of depth without increasing geometric complexity in the mesh; normal maps encode normals as RGB values that translate to XYZ vector directions[12]. These textures are applied to the mesh by using texture coordinates, also known as UVs, which are stored in the mesh for each vertex.

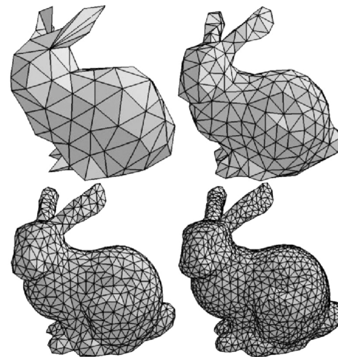

Figure 2.1: Meshes of the same object with varying numbers of triangles. Image credit: 3D Digital Recording of Archaeological, Architectural and Artistic Heritage[13] Fig. 2

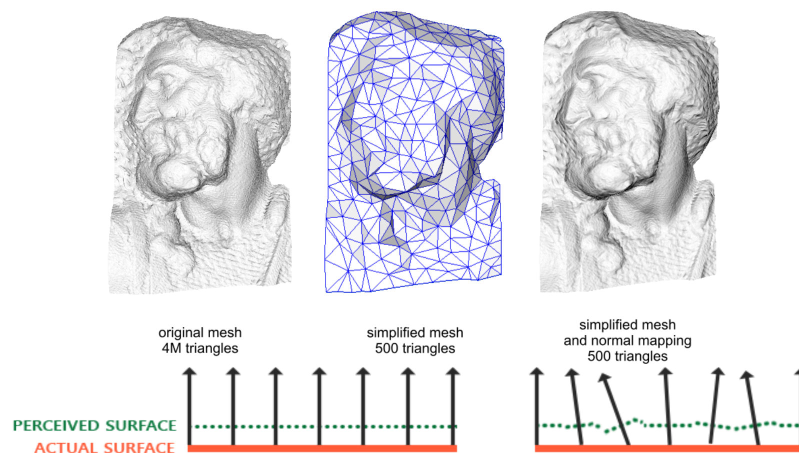

Figure 2.2: Visualisations of the effects of normal mapping. Image credits: Normal Mapping by Nikhil Kowshik[14] (top), Learn OpenGL by Joey de Vries[15] (bottom)

The final appearance of a model is influenced by shaders, programs that run on the Graphics Processing Unit (GPU) of the computer and are the main programmable parts of the rendering pipeline. Compiled shader programs are composed of several individual shader steps, though the two primary ones are the vertex shader, that intuitively operates on the mesh's vertex data, and the fragment shader, which uses rasterised output from the vertex shader and the material properties as its input. Rasterisation is the step in the rendering pipeline that takes place between the vertex

and fragment shaders and converts pure geometric data from the vertex shader into fragments to be passed to the fragment shader. The fragment shader, as its name suggests, calculates the values of fragments, which differ from pixels only in that pixels are the fragments that make it into the final rendered image to be displayed. It is usually in the fragment shader that we find lighting techniques that use a material's properties to calculate its model's final surface appearance in the rendered image[15].

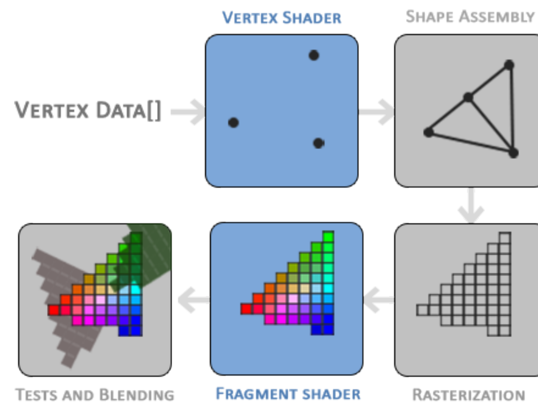

Figure 2.3: High-level visualisation of the graphics pipeline with programmable steps highlighted in blue. Image credit: Learn OpenGL by Joey de Vries[15]

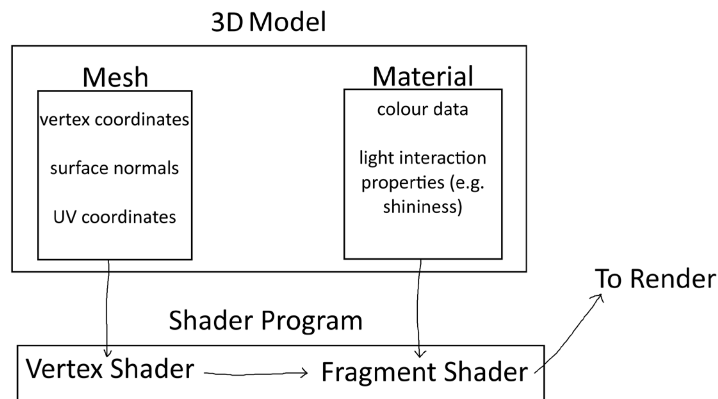

Figure 2.4: Simplified overview of the relationship between models and shaders

Within the graphics pipeline, vertices are transformed between a number of different coordinate spaces defined by differing origin coordinates and ranges that all position coordinates are made relative to; before ending up at final 2D normalised screen coordinate space, where the origin is at the bottom-left of the rendered image and coordinates correspond to on-screen positions. Such intermediate coordinate spaces are used as they are easier to work in for certain operations. The two spaces seen in this paper are object space, the coordinates relative to the object's own local origin, and world space, coordinates which are relative to the global origin of the scene at large[15].

2.2 Literature Review

Armed with relevant computer graphics knowledge, we delve now into a review of literature associated with the aim of our research,

2.2.1 Generating Stained Glass Renders

Until the early 2000s, stained glass windows were not thoroughly explored in computer graphics literature. One of the earliest described rendering techniques in this area was an automated method for transforming an arbitrary image into a stained glass version of that image, as presented by Mould[16]. This stained glass image filter takes a two-dimensional image such as a photo as input and renders a simple stained glass style plane, with associated coloured glass segments and imperfections, as output. Similar approaches exist, such as a method of restyling an image into a 2D texture that plausibly approximates the visual appearance of a specified work of stained glass, with minimal user input, proposed by Brooks[17]; as well as an automated technique that filters input images to create results stylistically similar to modern stained glass artworks described by Setlur et al.[18]; and a smoothed stained glass tile segment generation procedure defined by Seo et al.[19] that, too, renders a stained glass style plane from a source image. A later paper by Doyle et al.[20] presents an approach that claims to directly improve on previous works by offering a better representation of the original input photo while retaining the stained glass style in the final output. Additionally, related to these single image processing works is a rather recent paper by Kang et al.[21] that introduces a method for generating a temporally coherent stained glass animation from a video input in a similarly stylised, two-dimensional fashion.

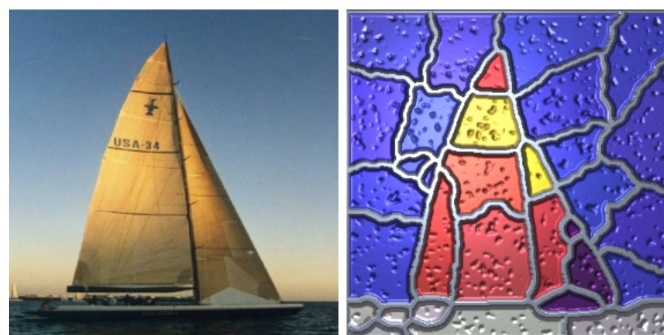

Figure 2.5: Source input image (left) and rendered output image (right) using Mould's method. Image credit: A Stained Glass Image Filter[16], Fig. 8

Figure 2.6: Source input image (left), target style input image (centre), and rendered output image (right) using Brooks' method. Image credit: Image-Based Stained Glass[17], Fig. 15

Figure 2.7: Source input image (left) and rendered output image (right) using the method by Setlur et al. Image credit: Automatic Stained Glass Rendering[18], Fig. 1

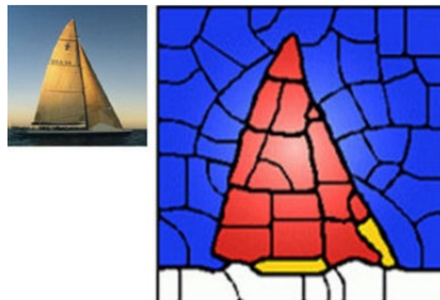

Figure 2.8: Source input image (left) and rendered output image (right) using the method by Seo et al. Image credit: Stained Glass Rendering with Smooth Tile Boundary[19], Fig. 3

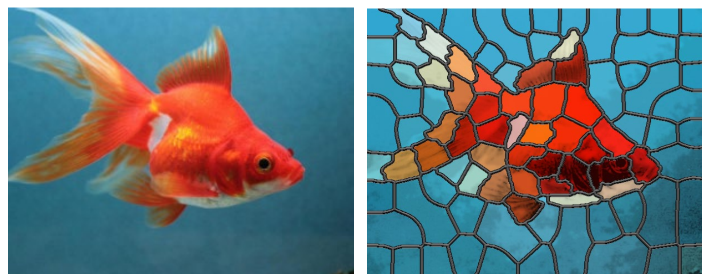

Figure 2.9: Source input image (left) and rendered output image (right) using the method by Doyle et al. Image credit: Painted Stained Glass[20], Fig. 3 and Fig. 6

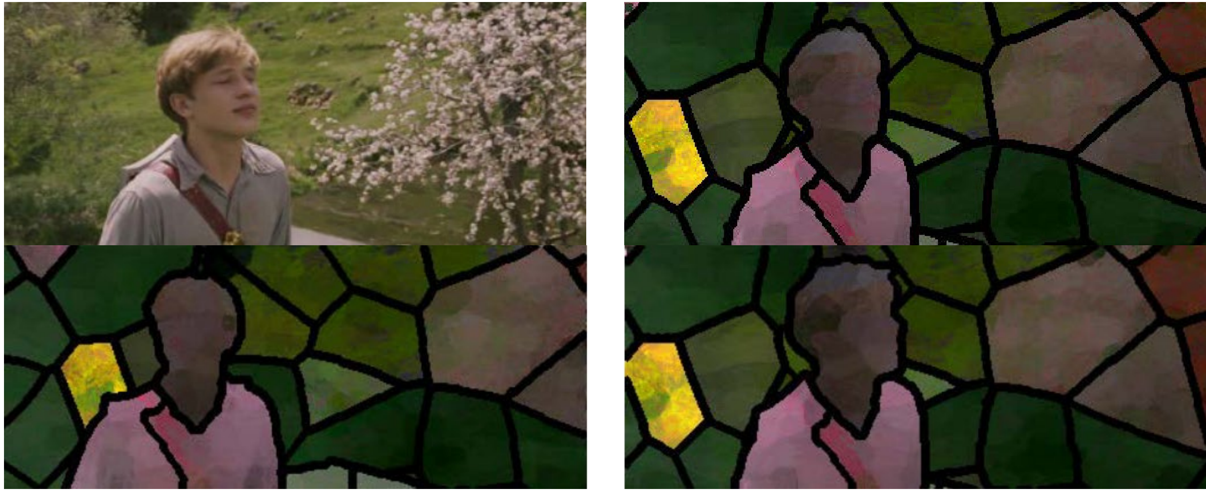

Figure 2.10: A sample frame of the source input video (top-left) and three frames of the rendered output video, using the method by Kang et al. Image credit: Video-based Stained Glass[21], Fig. 9 and Fig. 10

2.2.2 Realistic Stained Glass Rendering

We begin seeing a more three-dimensional approach alongside lighting considerations in the field in works that model the interaction of light with stained glass, such as in the raytracing and photon mapping algorithm developed by Shin[22]. This method focuses on the realistic simulation of light as it passes through coloured transparent surfaces based on the actual chemical and optical characteristics of stained glass, rather than rendering full stained glass windowpanes.

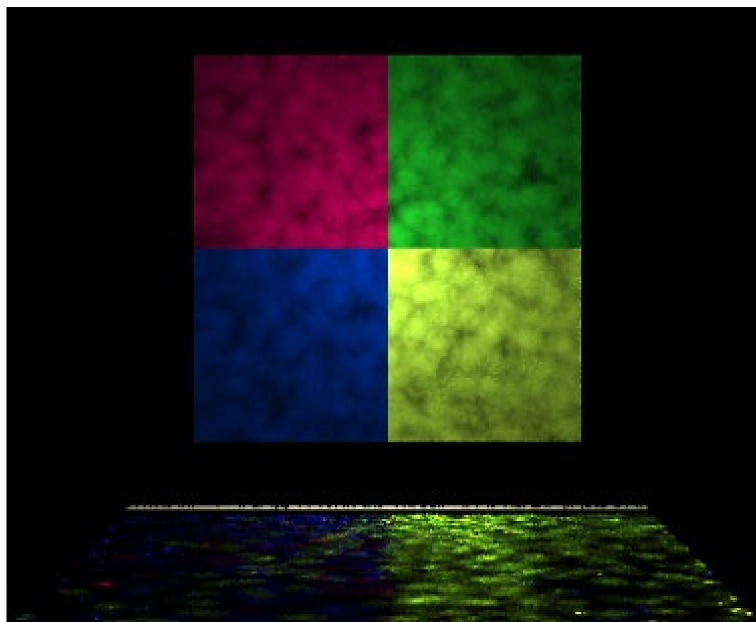

Figure 2.11: Rendered stained glass squares using Shin's method. Image credit: Modelling Stained Glass[22], Fig. 6

Another realistic lighting model has been presented by Kim et al.[23], which concentrates on simulating the appearance of light as it is seen passing through a rendered 2D texture of a real stained glass image. This model is of particular interest as it not only uses a photo of an existing stained glass window but also makes use of performant, real-time rendering techniques unlike aforementioned works.

Figure 2.12: Resulting stained glass render with various viewing directions (top row) and light source directions (bottom row) using the method by Kim et al. Image credit: A Realistic Illumination Model for Stained Glass Rendering[23], Fig. 5

Further, we can observe work done similar to that of Kim et al. in a paper by Thanikachalam et al.[24] and the respective PhD thesis by the first author[25] that proposes methods to perform virtual relighting of acquired stained glass images. The authors provide 2D modelling and rendering techniques that aim to accurately re-create the effects of light transport through stained glass in a physically accurate manner. The resulting solution provides the interactive and dynamic relighting of images of real-world stained glass windows.

Figure 2.13: Stained glass window relighting results from three different source video input frames using the methods presented by Thanikachalam et al. Image credit: VITRAIL: Acquisition, Modelling, and Rendering of Stained Glass[24], Fig. 19

2.2.3 Digitisation of Stained Glass Windows

The VITRAIL paper cited above states as one of its primary contributions the workflow for digitising stained glass windows for use in virtual museums. As this dissertation shares the motivation of digitising these artefacts for a variety of uses, including preservation, it is worth exploring existing works that investigate this practice.

Rahrig et al.[26] demonstrated the high resolution 3D scanning of various stained glass windows with a commercial-grade "structured light scanner" for the purpose of evaluating conservation and restoration measures. Babini et al.[27] present a review of invasive and non-invasive imaging and analysis techniques applied to stained glass windows with their respective potentials and limitations, with a particular focus on the potential of spectral imaging for the purpose of digitisation and analysis over time. This paper led to the same authors to investigate improved strategies for the acquisition of stained glass windows using hyperspectral imaging[28]. In this latter work, the authors provided a detailed methodology for acquiring information on the characteristics of individual stained glass works in-situ at the Swiss National Museum under a variety of lighting conditions.

2.2.4 Applications of Virtual Reality

Beyond preservation of cultural heritage, an additional use case of the digital re-creation of stained glass windows, is for the purpose of building design simulation in the field of construction architecture. The utility of VR as a form of human-computer interaction in this area has seen fairly rapid growth in recent years, and a number of works examine its role here. Lucas[29] discusses VR simulation as a powerful tool in

construction science education and presents a content development framework that can help students develop an understanding of the sequence and components of construction assemblies, evaluated with a pilot test study. This work, with the use of Unity Engine and C# scripting, finds that simulated VR experiences alongside traditional classroom-based delivery of material allowed for enhanced student learning over a lecture-only environment. Additionally, a study by Kim et al.[30] finds that VR simulation was equal or superior to its computer-based counterpart in construction education. Furthermore, a review by Patel et al.[31] on VR in architectural learning finds it to be an "effective educational tool for extremely complicated or conceptual issues that needed visualisation and spatial comprehension" and that the technology "improves students' comprehension and learning performance". Related, is the review by Feng et al.[32] on the use of immersive VR for building evacuation training. The authors concluded that VR is "effective in delivering considerable evacuation knowledge, no matter whether it is multiple knowledge (e.g., best practices) or single knowledge (e.g., spatial skill)".

A paper by Han[33] explores the application of interactive VR in architectural landscape design and the technology's associated advantages over traditional technical graphics. Further research by Shan et al.[34] delves into the use of VR simulation in interior and exterior landscape planning and design alongside modern 3D modelling and computer-aided design software solutions. A review by Ververidis et al.[35] looks at and compares several state-of-the-art VR solutions for the Architecture, Engineering, and Construction (AEC) industry through the lens of interdisciplinary collaboration. This review acknowledges the advantages of VR and finds a need for an open standard combining the best aspects of existing systems due to the significant differences between VR vendors. Two papers by Ehab et al.[36, 37] investigate the potential of VR to enhance public involvement in co-design of architectural projects for public and social spaces. They find that VR technologies can enhance the design process, streamline decision-making, and facilitate participatory urban design by virtue of providing real-time immersive and interactive experiences.

Several abovementioned works also cite accessibility as a primary advantage of VR, such as students visiting virtual construction sites[29] or members of the public contributing to the urban design of public spaces they have not physically visited[37]. We are also interested in the fact that dynamic light variation in images of stained

glass windows improves the memorability of those windows, as in the study performed by Nevin[38]. Additionally, compelling VR simulation a valuable proxy for reality and more useful than image-based stimuli for cognitive science studies, Snow et al. found in their review[39].

2.3 Summary

In the pursuit of simulating coloured transparency such as that of stained glass, numerous approaches have been devised. However, the project developed as part of this research endeavours to go beyond existing works by attempting to perform such lighting simulation in real-time VR. This choice stems from the acknowledged benefits of the medium as reviewed in section 2.2.4 above. The objective is to strike a delicate balance between realism and the demanding performance constraints imposed by real-time rendering. We have discussed related techniques for generating renders in the style of stained glass from diverse image and video inputs in section 2.2.1, though our specific intent lies in simulating authentic windows found in the physical world. In section 2.2.3, we have also examined procedures for the highly detailed digitisation of stained glass, but we provide an alternative means that circumvents the necessity for costly or specialized equipment, relying instead on digital photographs of the desired artefacts.

3 Methodology

In this chapter, we lay out our design considerations and decisions before providing a comprehensive overview of the implementation approach taken.

3.1 Design

The first stage of any software project is choosing a toolset. As this will affect the rest of the project, making a well-informed decision is of utmost importance. Unity Engine is an all-in-one, real-time 3D development engine by Unity Technologies[40]. It was the engine of choice in a previously discussed study by Lucas[29], is free and well-documented[41], and has a thriving asset and plugin ecosystem[42]. It is a popular engine of choice across a variety of industries such as AEC and automotive transportation & manufacturing; with applications developed using it being downloaded over three billion times per month in 2019 on over 1.5 billion unique devices across more than 20 platforms[43]. It is also the leader in AR/VR development, with Unity's internal estimates of its use in the sphere ranging from 60%[44] to as high as 95%[45]. Its main competitor is Unreal Engine by Epic Games[46]. While Unreal is popular with large development studios; Unity supports a wider array of platforms, enjoys larger userbase and market share, and is considered to be more user-friendly and accessible[47]. We can therefore consider this Unity Engine to be a suitable choice for the purposes of creating a performant, interactive, real-time VR prototype application. Unity provides a choice of three different rendering pipelines:

- The legacy Built-in Pipeline focuses on ease-of-use and compatibility at the cost of customisability.
- The High Definition Render Pipeline (HDRP) targets new, high-end devices and provides advanced graphics capabilities at the cost of performance, similar to Unreal Engine.
- The Universal Render Pipeline (URP) is optimised for performance to reliably provide scalable, modern graphics to a range of platforms from web and mobile to PC and VR while lacking support for newer performance-intensive techniques such as real-time raytracing.

The latter two are based on Unity's newer Scriptable Render Pipeline (SRP) that provides developers more direct control in the C# programming language. Of the three, URP is the most suitable choice for our VR prototype as it is modern, extensible, and optimised for all VR platforms, including mobile and untethered VR[48].

	Built-in	URP	HDRP
Customisable (SRP)	✗	✓	✓
VR-Friendly	✓	✓	✗
Advanced Graphics	✗	✗	✓

Table 3.1: High-level comparison of Unity render pipelines by their key features

3.2 Implementation

The following section describes the general steps for creating the Unity prototype project scene provided in a public GitHub repository[i].

3.2.1 Setup

First, it is necessary to download the Unity Hub application from the official Unity website[ii] and install it. This application is used for Unity licence and project management and Unity Editor installs. With an active Unity account and Personal or Educational licence, we can select a version of Unity Editor to install. The prototype was developed on Windows 11 using the latest recommended version, which as of this point in development was 2022.3.17f1. It should be noted that Unity versions are generally backwards compatible but not forwards compatible (i.e. using a newer version of Unity Editor with an older project is supported, but not vice versa). Before the installation begins, we are presented with a choice of modules to include in the install. The modules included for the development of the prototype in this dissertation were "Android Build Support" and its dependencies, as well as "Windows Build Support (IL2CPP)".

[i] https://github.com/Zugidor/VR-Stained-Glass
[ii] https://unity.com/download

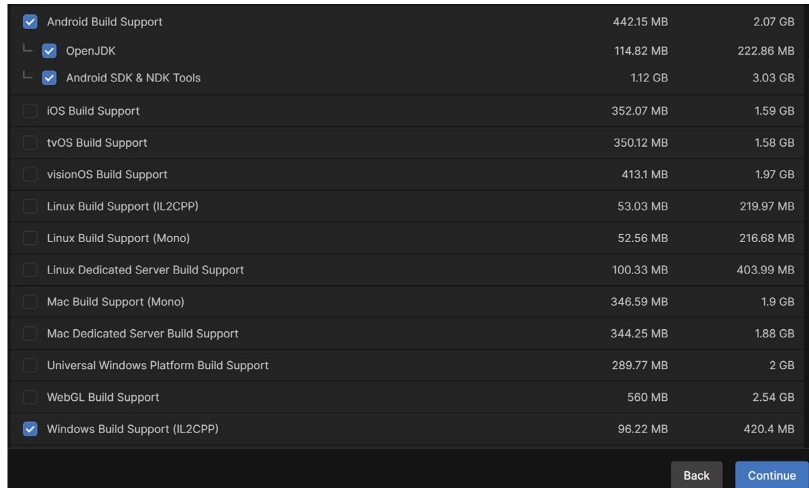

Figure 3.1: Module selection window in Unity Hub during Unity Editor install process

After the installation is complete, a new project is created using the "3D cross-platform (URP)" template. As VR compatibility may be added to an existing project, we concentrate first on developing the prototype scene before preparing the project for VR deployment. After the project is created, it is opened in a Unity Editor window. We follow the optional but recommended step of configuring a code editor such as Visual Studio Code to work with Unity[49].

3.2.2 Scene

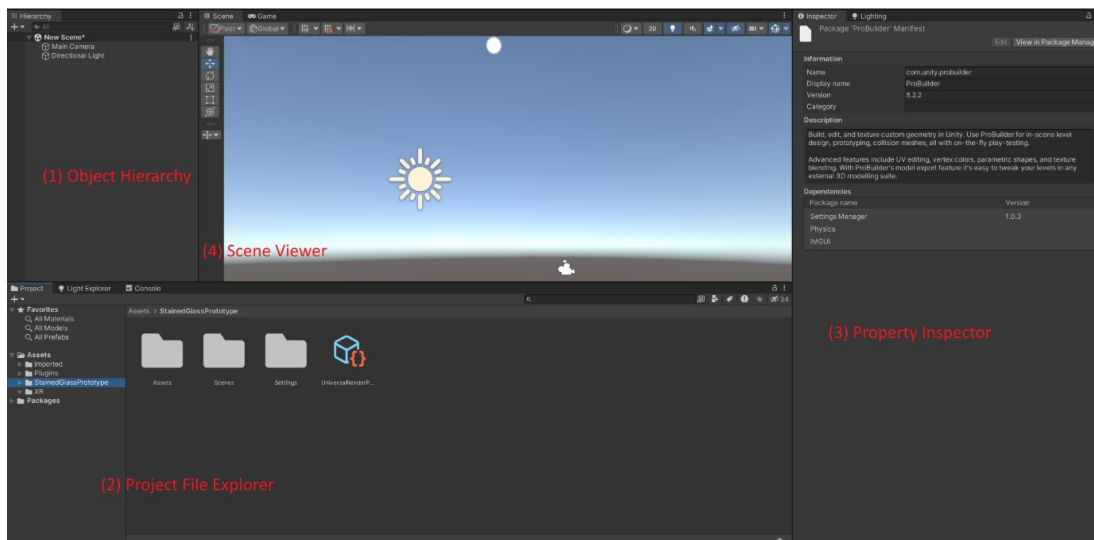

Figure 3.2: Sample empty scene view in Unity Editor

In the Unity Editor, we observe four primary windows of interest on-screen similar to Fig. 3.2 above:

1. The Object Hierarchy window that lists all items associated with the currently open scene. New objects are added here with the right-click menu.

2. The Project File Explorer window, which allows the user to navigate the currently opened project's files and folders. New files can be created with the right-click menu.
3. The Property Inspector window, where all the properties of the last selected item are displayed and can be edited.
4. The Scene Viewer window which displays a configurable 3D render of the current scene and allows direct selection and manipulation of visible objects.

To create an interior environment in the scene, we must install the ProBuilder package by opening the Unity Package Manager window found in the top toolbar menu under "Window > Package Manager" and searching for "ProBuilder" in the Unity Registry packages. Referring to the official Unity Learn tutorial on using ProBuilder for prototyping[50] provides us the requisite knowledge to create a rudimentary room with cutouts for windows. Afterwards, we furnish the scene with third-party assets such as textures and models freely available on the Unity Asset Store. All assets used in the final prototype project are provided in numerically labelled folders in the "Imported" folder and listed in a sources.txt file in the top-level "Assets" folder in the project files.

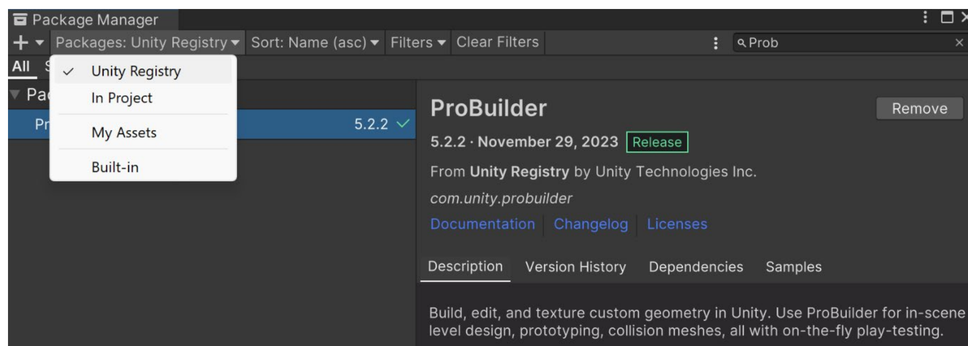

Figure 3.3: Searching for and installing ProBuilder in the Unity Package Manager

Figure 3.4: Sample bare interior environment created with ProBuilder in Unity

Figure 3.5: Prototype environment after importing free Unity Asset Store assets. Visible assets include “Lemon Trees” by Numena[iii], “Grass Flowers Pack Free” by ALP[iv], and “NoirMat – Noir Marble Pack Vol. 01” by Noir Project[v]

3.2.3 Shaders

In Unity, shader programs are contained in Unity Shader objects (instances of the Shader class), while Unity materials contain references to Shader objects[48]. Unity shaders can be developed using either the ShaderLab declarative language and High-Level Shader Language (HLSL), or the Shader Graph[51] node-based visual scripting tool. In Shader Graph, each “node” represents a constant, variable, function, or a mathematical or logical operation; with the exception of the vertex and fragment “master nodes” that represent vertex shader and fragment shader outputs of the shader program, respectively. Shader Graph supports custom nodes programmed in HLSL, and Shader Graphs can be converted into ShaderLab/HLSL code. Shader Graphs nodes can be grouped, and notes can be added as comments to improve a graph’s readability. For the purposes of creating a proof-of-concept prototype, Shader Graph meets our requirements while saving development time.

As per Nevin[38], we would like to see variation in the light and shadow passing through the windows for the purposes of memorability. We can achieve this by creating a shader program that simulates wind by manipulating the vertices of the tree models. The official Unity YouTube channel provides a tutorial video for making a simple wind shader[vi], which we base our shader off of. The final wind shader used in the prototype in Shader Graph can be seen in Fig. 3.6. Moreover, the shader can be described in pseudocode, as in listing 1.

[iii] https://assetstore.unity.com/packages/3d/vegetation/trees/lemon-trees-200372
[iv] https://assetstore.unity.com/packages/2d/textures-materials/nature/grass-flowers-pack-free-138810
[v] https://assetstore.unity.com/packages/2d/textures-materials/noirmat-marble-pack-vol-01-128318
[vi] https://youtu.be/ZsoqrHHtg4I

```
Vertex WindShaderV(Vector2 WindDirection, Float WindStrength,
 Float WindSpeed, Float Flexibility){

  // strength to cyclically vary with time
  Float Strength = WindStrength * Time.SineTime();
  // original object space coords i.e. relative to self
  Vector3 ObjPos = Position(Space=Object);
  // vector.rgba == vector.xyzw
  Float PosY = ObjPos.G;
  // distort mesh from top
  Strength *= PosY;
  // bending function (x+1)^4 - (x+1)^2
  Strength = (Strength+1) * (Strength+1);
  Strength = (Strength * Strength) - Strength;
  // direction is 2D (looking from top)
  Vector2 Pos2d = WindDirection * Strength;
  // swap Y with Z (up/down to front/back)
  Vector3 Pos3d = Vector3(R=Pos2d.R, G=0, B=Pos2d.G);
  // original world space coords i.e. relative to entire scene
  Vector3 WrlPos = Position(Space=World);
  // new world space coords
  WrlPos += Pos3d;
  Vector3 NewPos = Transform(Input=WrlPos, From="World", To="Object",
   Type="Position");

  // non-uniform movement i.e. adding false sense of randomness to wind
  // speed to constantly vary with time
  Float Speed = WindSpeed * Time.Time();
  // convert speed to UV (XY) coords offset by speed value
  Vector2 VarOffset = TilingAndOffset(Offset=Vector2(Speed));
  // repeating noisy texture using offset by speed value as coords such that
   it moves with time at a constant speed, scaled by flexibility value to
   simulate a more or less flexible tree
  Float RandVal = GradientNoise(UV=VarOffset, Scale=Flexibility,
   Type="Deterministic");
  // simple a+(b-a)*t linear interpolation between full new position and
   original position according to randomly varying val
  Vector3 FinalPos = Lerp(A=NewPos, B=ObjPos, T=RandVal);
  Vertex Vert = Vertex(Position=FinalPos);
  return Vert;

}
Fragment WindShaderF(Texture2D Diffuse, Texture2D Normal){

  Vector4 Diff = SampleTexture2D(Texture=Diffuse, Type="Default",
   Space="Tangent");
  Vector4 Norm = SampleTexture2D(Texture=Normal, Type="Normal",
   Space="Tangent");
  // dim colour by factor of 4 so the tree doesn't look too bright through
   the window
  Fragment Frag = Fragment(BaseColor=Diff.RGB/4, Alpha=Diff.A, Smoothness=0,
   Normal=Norm);
  return Frag;

}
```

Listing 1: Pseudocode translation of the wind shader with explanatory comments

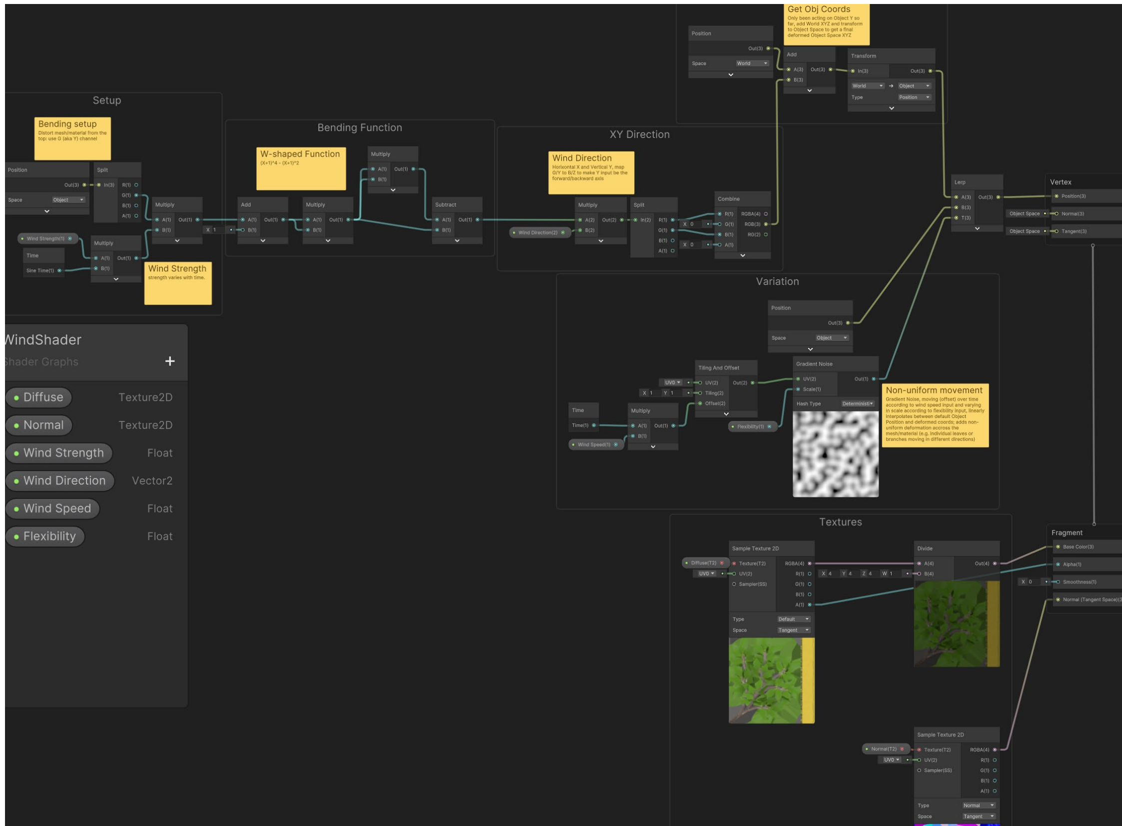

Figure 3.6: Wind Shader Graph (WindShader) as implemented in Unity

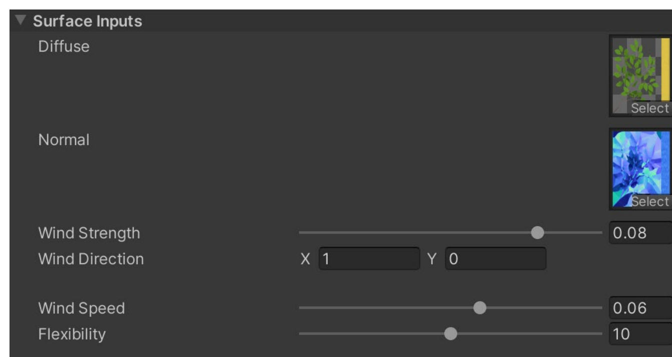

Figure 3.7: The shader input parameters we expose to the end user. We configure the UI appearance and minimum/maximum/default values in Shader Graph

The second, and most important, shader that we require is the stained glass shader. We approximate the heterogeneity and imperfections in stained glass windows with an additional generic normal map, using it as a “distortion texture”, alongside the usual diffuse texture and normal map inputs. Desired effects include variables to control the appearance of objects behind the glass such as blur, brightness, and transparency. As the windows are static, we only implement a fragment shader, leaving the vertices unmodified.

```
Fragment SG_ShaderF(Texture2D Diff, Texture2D DisTex, Texture2D Norm,
 Float Met, Float Smooth, Float Dist, Float Transpar, Float Bright,
 Float AmbOcc, Float Blur, Float Thick){

  // distortion
  Vector2 Distort = SampleTexture2D(Texture=DisTex, Type="Normal",
   Space="Tangent").RG * Dist;
  // make effect more subtle
  Distort = Distort / 40;
  // distort original 2D normalised screen coordinate
  Distort = ScreenPosition(Mode="Default").RG + Distort;
  // colour value behind transparent object at distorted coordinate
  Vector3 Distorted = SceneColor(UV=Distort);

  // Box Blur
  Vector2 Blr1, Blr2, Blr3, Blr4, Blr5, Blr6, Blr7, Blr8, Pos;
  Pos = ScreenPosition(Mode="Default");
  Blr1 = TilingAndOffset(UV=Pos, Offset=Vector2(Blur, 0));
  Blr2 = TilingAndOffset(UV=Pos, Offset=Vector2(Blur, Blur));
  Blr3 = TilingAndOffset(UV=Pos, Offset=Vector2(0, Blur));
  Blr4 = TilingAndOffset(UV=Pos, Offset=Vector2(-Blur, 0));
  Blr5 = TilingAndOffset(UV=Pos, Offset=Vector2(-Blur, -Blur));
  Blr6 = TilingAndOffset(UV=Pos, Offset=Vector2(0, Blur));
  Blr7 = TilingAndOffset(UV=Pos, Offset=Vector2(-Blur, Blur));
  Blr8 = TilingAndOffset(UV=Pos, Offset=Vector2(Blur, -Blur));
  Vector3 Blurred = (SceneColor(UV=Blr1) + SceneColor(UV=Blr2) +
   SceneColor(UV=Blr3) + SceneColor(UV=Blr4) + SceneColor(UV=Blr5) +
   SceneColor(UV=Blr6) + SceneColor(UV=Blr7) + SceneColor(UV=Blr8) +
   Distorted) / 9;

  // LERP between Base and max(Base,Blend) to provide end user with controls
   for both transparency and a sense of thickness where a "thicker" glass
   only lets higher colour values (like sunlight) through
  Vector3 Glass = Blend(Base=Vector3(Thick), Blend=Blurred,
   Opacity=Transpar, Mode="Lighten");

  Vector3 DiffTex = SampleTexture2D(Texture=Diff, Type="Default",
   Space="Tangent").RGB * Glass;
  // tint for a warmer, more natural white balance and user controllable
   stained glass pigment brightness.
  Vector3 Brightness = Vector3(R=Bright, G=(Bright / 1.2),
   B=((Bright / 1.2) / 1.2)) * DiffTex;
  Vector3 NormTex = SampleTexture2D(Texture=Norm, Type="Normal",
   Space="Tangent").RGB
  Fragment Frag = Fragment(BaseColor=DiffTex, Metallic=Metal,
   Smoothness=Smooth, AmbientOcclusion=AmbOcc, Alpha=1, Normal=NormTex,
   Emission=Brightness);
  return Frag;

}
```

Listing 2: Pseudocode translation of stained glass shader with explanatory comments

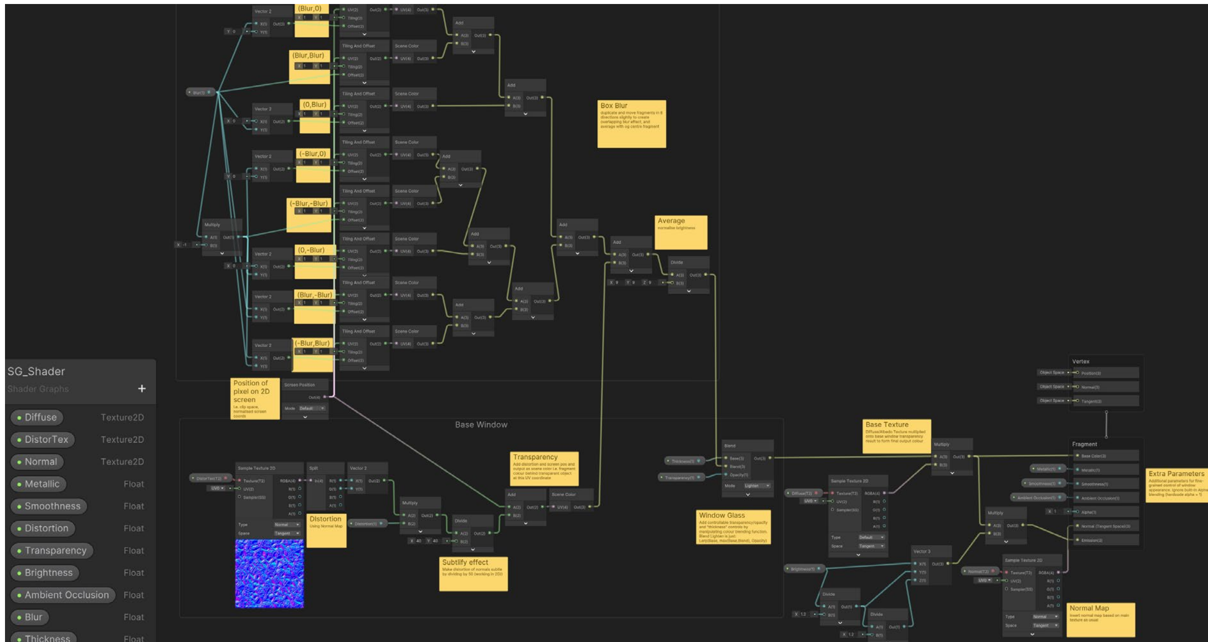

Figure 3.8: Stained Glass Shader Graph (SG_Shader) as implemented in Unity

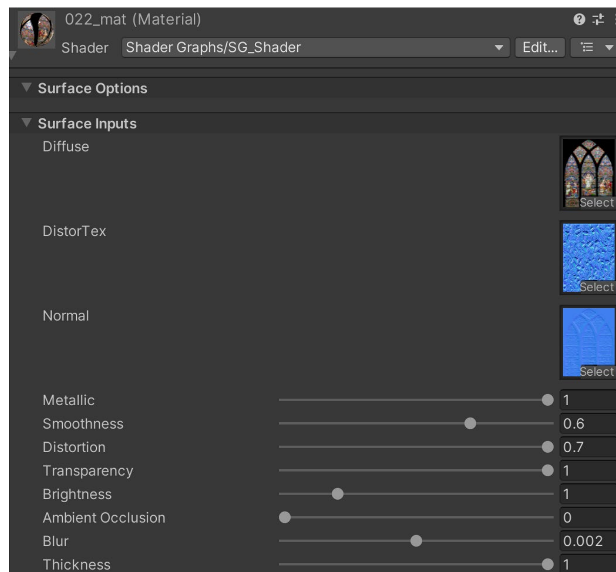

Figure 3.9: User-facing shader input parameters for a sample stained glass window

Figure 3.10: Prototype environment after adding windowpanes with SG_Shader, each using differently configured user input parameters

3.2.4 From Image to Material

This project used digital photographs of existing stained glass windows collected by colleagues at Trinity College as part of a 2022 – 2023 project[52]. However, neither these nor any other unprocessed photo images, can be used as-is with the above SG_Shader. We must first prepare an appropriate texture image, normal map, and shadow mask using an image editing program such as Adobe Photoshop[53] or Paint.NET[54].

The main diffuse texture that determines the colour of the light passing through the windowpane is created by blacking out all elements of the image that are not the stained glass panes themselves, such as walls, sills, seams, lead cames, and any objects partially or completely obscuring the window. This is necessary as any non-black areas, such as dark brown, will be rendered as transparent rather than opaque, due to multiplication by a non-zero value. Only black areas of the texture, with RGB values of exactly zero, will be rendered as opaque, letting no light through. This work can be done with a combination of thresholding tools like "magic wand" or "bucket fill" and manual paint brushing where thresholding fails to adequately black out sections of the image, due to the inherent limitations of working with raster images.

Figure 3.11: A thresholding tool's area of effect on a sample stained glass window image to create an appropriate diffuse texture. Note the imperfect selection on the more intricate parts of the window

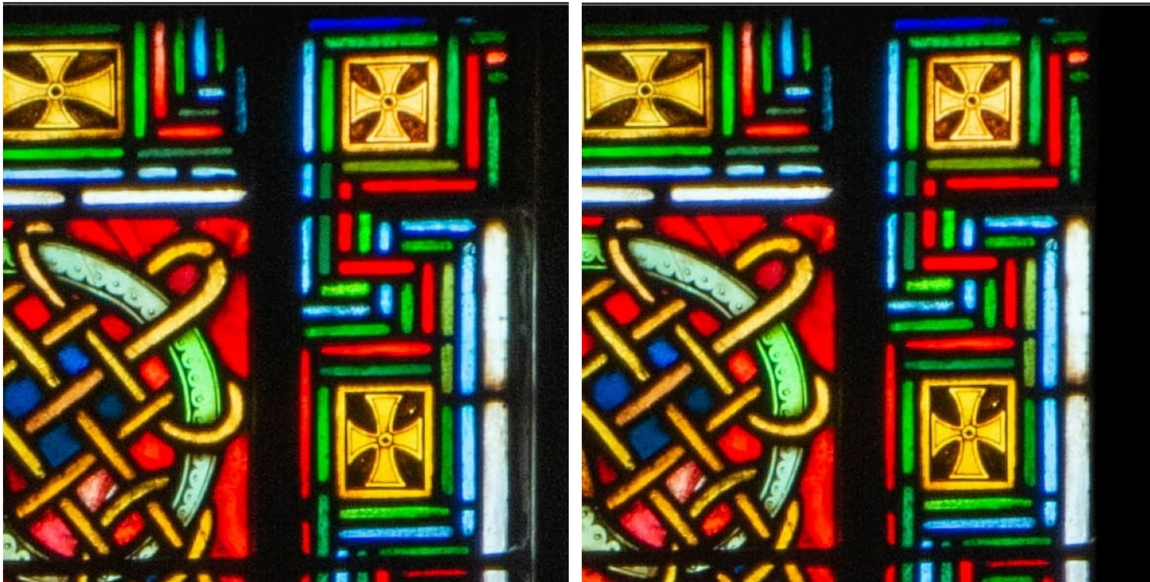

Figure 3.12: Zoomed in before and after the use of a thresholding tool

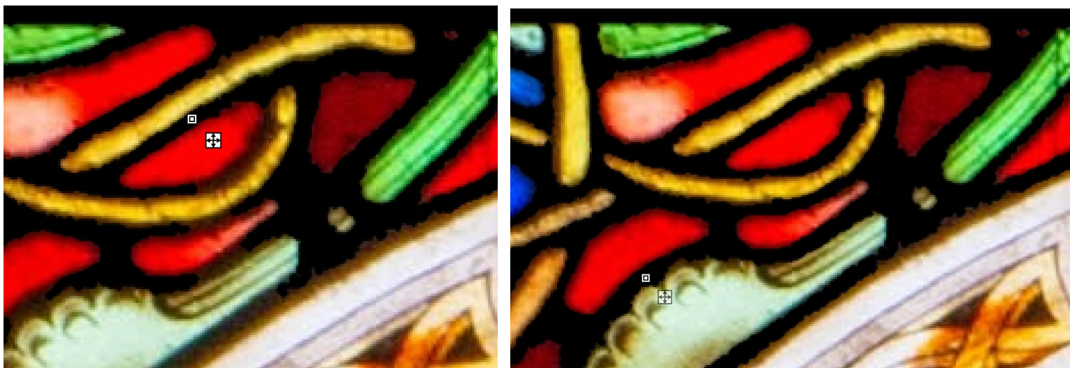

Figure 3.13: Using a brush tool to manually clean up the results of a thresholding tool

After completing this step, we must generate a normal map to provide a sense of depth to the window. We must create a greyscale image where all the glass panes are coloured white with the maximum RGB values of 255 and all else is black with RGB values of zero. This can be achieved more quickly than the previous step, as we have already coloured all opaque areas black, so we use the image editor's relevant thresholding tool to colour all non-black areas white. Any unsatisfactory parts can be cleaned up again by manually brushing white or black where necessary. This processed black-and-white image can then be used to generate a normal map with a tool such as the one found in Photoshop 2023. It is important to note that simply applying a greyscale filter leads to improper normal map generation and unsatisfactory results.

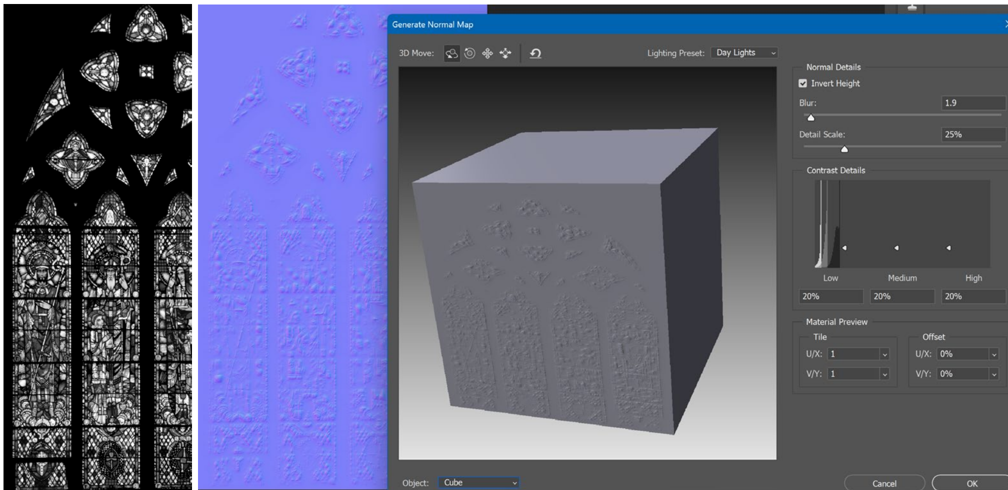

Figure 3.14: Poor results in using a basic greyscale filter to generate a normal map

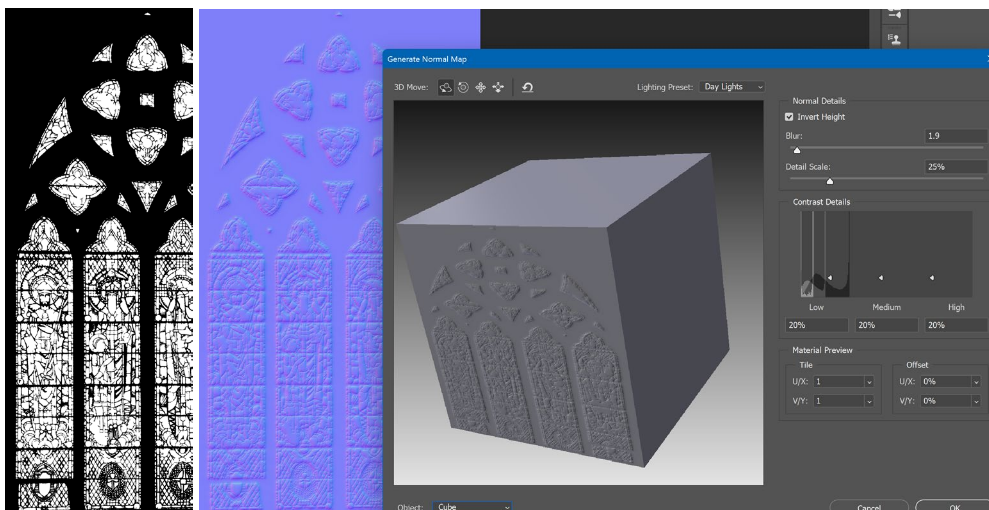

Figure 3.15: Improved resulting normal map generation using a thresholded image

Finally, a shadow mask is created by simply marking all white areas as transparent, which is usually done in image editors by selecting and deleting the desired areas. To soften the appearance of the shadows and reduce the amount of light allowed through the shadow mask, we apply a Gaussian blur filter. In paint.NET, a radius value of 15.0 provides an acceptable result. It is particularly important to save this mask as a PNG or similar image file format that stores alpha channel information, unlike JPEG. The amount of time and effort required depends on the desired level of detail and quality in the resulting material. Working on three windows, the average time required to create a finished texture, normal map, and shadow mask was found to be approximately one hour per window.

We create a material in Unity by using the right-click context menu in the project file explorer window labelled in Fig. 3.2 and select our SG_Shader as the

material's shader. The normal map and base colour texture are used as inputs to an SG_Shader material as seen in Fig. 3.9, while the shadow mask is used on another material, using the default URP Lit shader, applied to a different 2D plane mesh, which we set to render "shadows only", aligned with the window mesh in order to cast the shadows cast by the opaque parts of the window. It is important to ensure at this point that our normal maps are marked as such and not "Default" under the "Texture Type" property in the Unity Property Inspector window for each normal map.

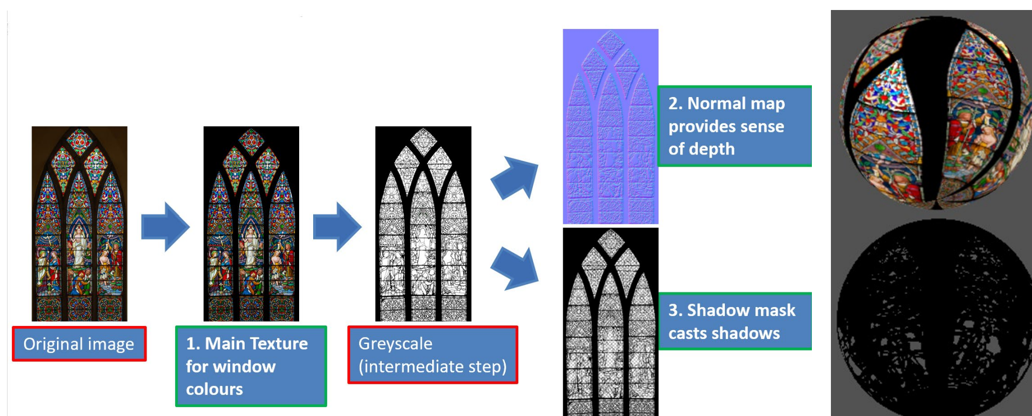

Figure 3.16: Image-to-material pipeline (left) and the two resulting materials as previewed in Unity Editor on default spherical meshes (right)

Figure 3.17: Prototype environment after applying the processed texture and normal map to the SG_Shader materials and creating their respective shadow mask materials

Figure 3.18: Normal map provides a false sense of depth for the individual panes of stained glass making up the window, observable when light shines at a steep angle (left). Screenshots differ only in sun position

3.2.5 Post-Processing and Lighting

Post-processing refers to the application of effects and filters to the entire image after the frame has been rendered to stylise or improve the realism of the image by simulating physical camera and film properties[48]. Unity's URP includes an integrated implementation of post-processing effects using the volume framework. Bloom and vignette are listed as some of the most common and performant effects. For VR, it is recommended that we use the vignette effect and avoid lens distortion, chromatic aberration, and motion blur[55]. The Bloom effect creates fringes of light extending from the borders of bright areas in an image, creating the illusion of extremely bright light overwhelming the camera. Vignetting is the term for the darkening towards the edges of an image compared to the centre, drawing focus to the centre of the image. We can use tonemapping to remap the colour values of the image to ACES colour space so as to approximate the appearance of a photo-realistic high dynamic range image with a wide contrast between the darkest and brightest parts of the image. Most post-processing effects in URP are configured in a special "volume" object. In this prototype, we use a global volume which takes effect throughout the entire scene[48, 55].

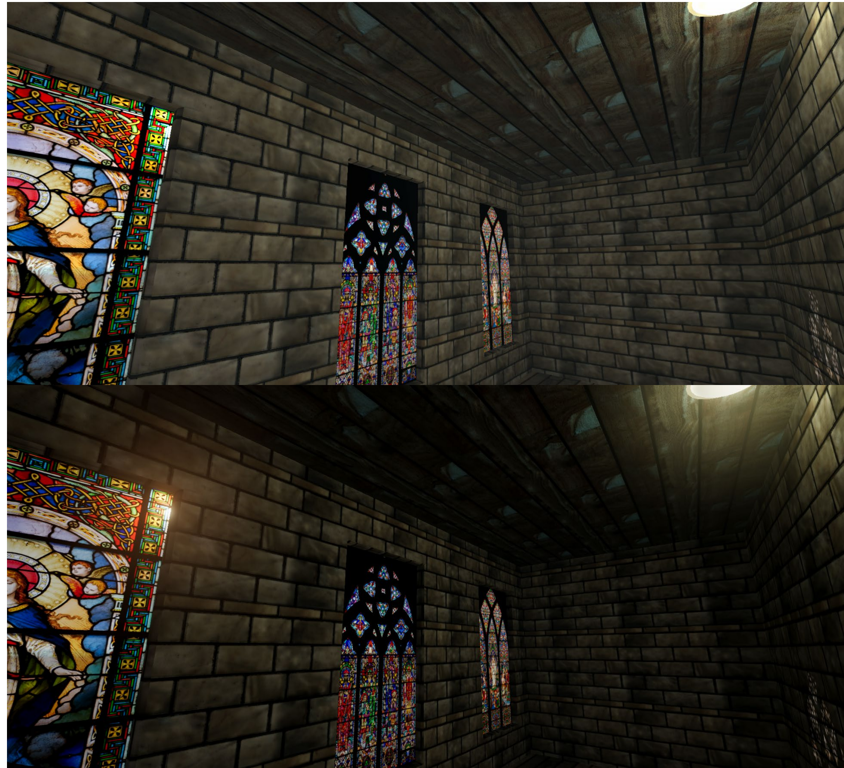

Figure 3.19: Before (top) and after (bottom) applying bloom, vignette, and ACES tonemapping post-processing. Visible assets include "Street Lamps 2" by SpaceZeta[vii]

While we have previously ruled out the use of real-time raytracing as too expensive, we still desire to have some approximation of coloured shadows or caustics cast by light passing through the stained glass windows. This can be done by using a coloured Unity light cookie texture, which is a mask attached to a light object to create a shadow with a specific shape or colour, changing the appearance and intensity of the light. Light cookies are an efficient way of simulating complex lighting effects with minimal to no runtime performance impact[48]. The light cookie used for this prototype was originally sourced from Jojo's Textures[viii], provided free for personal use, and edited with a variety of desaturation and blurring filters to achieve a plausible appearance. The edited texture is made seamless using the free online tool IMGonline[ix]. The light cookie is attached to the directional light object representing the sun, and as such individual textures could not be cast as on a per-window basis.

[vii] https://assetstore.unity.com/packages/3d/props/exterior/street-lamps-2-260395
[viii] https://jojotextures.blogspot.com/2016/12/stained-glass-seamless-textures-1.html
[ix] https://www.imgonline.com.ua/eng/make-seamless-texture.php

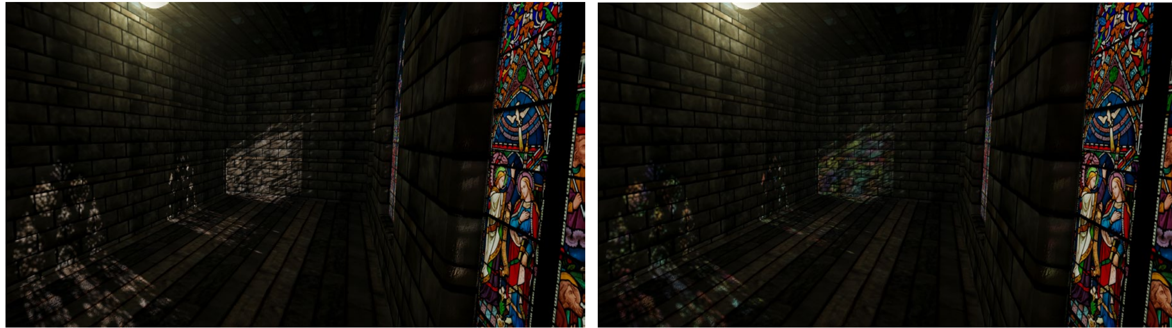

Figure 3.20: Before (left) and after (right) applying a generic coloured light cookie texture. Like the distortion texture, it is generic and common to all three windows in the prototype

Figure 3.21: Attempting to apply one of the window images as a light cookie applies it repeatedly to all light cast by the sun, leading to uncanny and unrealistic results

Unity provides wide array of lighting options to approximate how light behaves in the real world[48]. In this prototype, we use a point light source for the interior lamp, and a directional light source for the sun. Since the point light is static and does not move, we can mark it as such and allow Unity to "bake" its light data into a texture called a lightmap so that it does not need to be re-calculated every frame, improving performance. We do not have this luxury with the directional light, as we must recalculate its interaction shadows of the windows when the sun moves, and of the shifting tree shadows even if the sun itself does not move. This is all regarding direct light, which is defined as light that hits a surface at most once before being registered by the camera. Indirect light, such as ambient light from the sun or light that bounces more than once inside a room, is commonly baked in all aspects of a scene. Modelling both direct and indirect lighting to provide realistic results is known as global illumination, which Unity has a wealth of lighting configuration settings for. This

prototype uses "Baked Indirect" which provides a balance of performance and visual fidelity. Lightmaps are baked using Unity's progressive GPU lightmapper to generate the required lightmap textures. Unity chooses which scene objects to bake lighting for based on whether they are marked "static" or not. The only object in our scene that is expected to move currently is the directional light, so we can safely mark all other objects static in the top-right corner of the property inspector for each object.

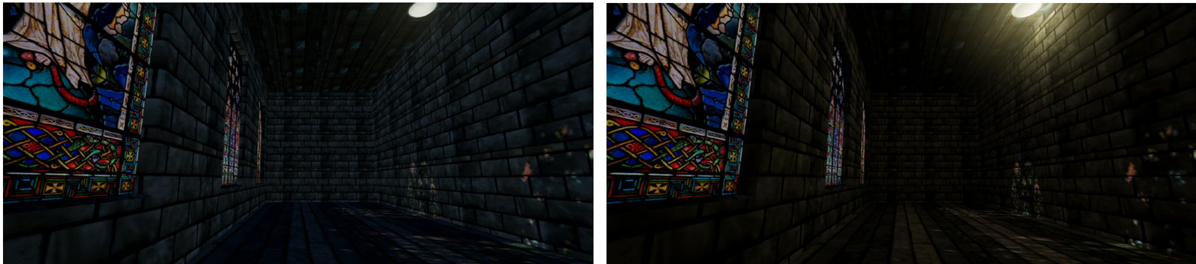

Figure 3.22: Before (left) and after (right) generating lightmap data using Unity's baked indirect global illumination model. All other previously discussed techniques are used in both images. The bluish tint seen on the left image is the result of ambient light as configured in Unity's lighting settings

Unity's URP exposes advanced graphics options to us in the .asset files inside the Settings folder[55]. We use the "High Fidelity_PipelineAsset.asset" file by default. These setting's changes are reflected instantly in the Scene View, so we can adjust them to achieve the balance between performance and visual fidelity that we desire. The most relevant settings here for our prototype are:

- Opaque Texture: Must be enabled for transparency to function in our windows.
- Opaque Downsampling: A selection of filters for anything seen through our windows that effectively blur the trees in addition to the blur implemented in SG_Shader.
- HDR: Must be enabled as we are using the Tonemapping and Bloom post-processing filters.
- Render Scale: Should be kept at 1 unless the target platform shows unsatisfactory performance, in which case this can be lowered to reduce the rendering resolution of the final image.

- Anti-aliasing: Hardware anti-aliasing should be disabled as it is not compatible on all platforms and can be performance intensive to use on top of software anti-aliasing like FXAA.
- Shadows: For our scene's size, a Max Distance of 25 seems ideal. A cascade count of 2 with the split at 12.5 metres and last border at 0 provides an acceptable level of visual quality. Depth and Normal Bias are 1 by default and may need to be reduced if light is found bleeding through the corners of the walls or windows where it should not. We enable high quality soft shadows for more realistic results.
- Main Light – Shadow Resolution: We can safely increase this to the maximum value of 4096.

3.2.6 Interactivity

In addition to the standard Scene View we have been working with so far and that all previous figures have been screengrabbed from, Unity Editor provides a "Game View" which displays how the final, built application will look and run[48]. As such, it requires a properly configured camera object to see anything other than a blank screen, and custom input handling in order to provide any amount of end user interaction. Our camera object uses the default built-in settings, except that post-processing must be explicitly enabled in order for the effects to be rendered in Game View. We also enable Fast Approximate Anti-Aliasing (FXAA) in the camera object, a post-processing effect, which is the recommended anti-aliasing setting in Unity when optimising for performance[55]. Anti-aliasing is a set of techniques used to smooth out and reduce aliasing, the jagged polygon edges of objects or thin lines in raster images[56]. Software-based anti-aliasing is one of the few post-processing effects applied per camera object rather than in volume objects. As we are targeting VR deployment, we must right-click our camera object and select the "XR > Convert Main Camera to XR Rig" option. Unity automatically creates two parent objects for our camera object. The immediate parent is a simple empty Camera Offset object that serves to place the camera a certain distance higher relative to the top-level XR Rig parent object. The XR Rig is initialised with a CameraOffset.cs input helper script, which is given the height of the aforementioned Camera Offset object. This simulates the VR headset being above the main body's centre of mass. The Camera itself is provided a Tracked Pose Driver component to provide motion control functionality.

Four C# scripts compose the interactive elements of this prototype, written with the assistance of the Unity scripting documentation[57]. Unity provides the UnityEngine namespace through which the vast majority of Unity scripting is done. Scripts are attached as components of a Unity object and can communicate with sibling components of the same object. Unity scripts are often defined as classes derived from the UnityEngine.MonoBehaviour base class, which provides lifecycle methods such as Start(), called once when a script is enabled and used for initialisation, and Update(), called every frame and where most core script functionality is defined.

CamController.cs uses simple WASD key input handling using the UnityEngine.Input class to create a movement direction vector based on the transform forward and right vectors of the object's Transform component. The Transform component contains position, rotation, and scale information of the object in world space. The movement vector is multiplied by the delta time between the previous and current frames, to generate consistent movement independent of framerate, before being supplied to the Move() method of the CharacterController component. A CharacterController is necessary as it calculates movement and collision detection for the object it is attached to, so we mark it as a dependency for Unity to automatically create one whenever the CamController script is attached to an object. For ease of testing, we add an Esc key handler to quit Game View and have the left shift key double our movement speed. When rotating the camera, we capture the mouse when the right mouse button is held by hiding the cursor and locking its position for convenience. The movement of the mouse is still recorded by Unity in this state. We get this mouse movement from Unity's Input class; multiply it by the delta time as before; multiply by twenty so that the camera rotation is not too slow; and add the horizontal and vertical values to the Transform's yaw and pitch, respectively. We unhide and unlock the cursor's position when the right mouse button is released, revealing the cursor to be in the same position as before rotating the camera and not somewhere on the edge of the screen. A user-editable member field can be exposed in the Unity Editor's UI by declaring the field public, as is done with the base movement speed float in this script. By attaching CamController.cs to the parent XR Rig object, which is effectively the "body" of the user, we can move and look around in Game View.

```
using UnityEngine;

[RequireComponent(typeof(CharacterController))]

public class CamController : MonoBehaviour
{
  CharacterController charCon;
  public float speed = 1.5f;
  float pitch, yaw, roll;
  Vector3 MovementInput()
  {
    Vector3 direction = Vector3.zero;
    Vector3 forward = transform.forward;
    Vector3 right = transform.right;
    if (Input.GetKey(KeyCode.W))
    {
      direction += forward;
    }
    //...
    direction.y = 0;
    return direction;
  }
  void Start()
  {
    charCon = GetComponent<CharacterController>();
    pitch = transform.eulerAngles.x;
    yaw = transform.eulerAngles.y;
    roll = transform.eulerAngles.z;
  }
  void Update()
  {
    // Press Escape to quit
    //...
    Vector3 moveDirection = speed * MovementInput();
    // Press Shift to sprint
    //...
    charCon.Move(moveDirection * Time.deltaTime);
    if (Input.GetMouseButton(1))
    {
      Cursor.visible = false;
      Cursor.lockState = CursorLockMode.Locked;
      Vector2 mouseMovement = 20 * Time.deltaTime *
       new Vector2(Input.GetAxis("Mouse X"), -Input.GetAxis("Mouse Y"));
      yaw += mouseMovement.x;
      pitch += mouseMovement.y;
      transform.eulerAngles = new Vector3(pitch, yaw, roll);
    }
    else
    {
      Cursor.visible = true;
      Cursor.lockState = CursorLockMode.None;
    }
  }
}
```

Listing 3: Contents of CamController.cs with most comments and the more repetitive and simple code removed, but otherwise unmodified

For the purposes of evaluating the prototype in VR, we will use Google Cardboard VR (GCVR). By testing on a mobile device, we can ensure that the prototype is performant and should run at a high framerate without issue on any other VR platform. This also makes testing cheap and accessible, as no specialised nor expensive hardware is required, only a relatively modern Android phone. Even a GCVR headset is technically optional, as GCVR apps will still run with no headset without issue. As such, we require a teleporting method of movement using the GCVR pointer. We consult the GCVR Unity plugin documentation[58] and install the relevant plugin as per the official quickstart guide[59]. Next, we add the CardboardReticle object from the plugin files as a child of our camera object and attach the CardboardStartup.cs script from the GCVR sample project files to the top-level parent object of all interactive objects, which is the Room object in our prototype. Finally, we must add a new layer in Unity, that we call "Interactive" in the prototype, and select it as the Reticle Interaction Layer Mask in CardboardReticle. We can now add objects to the scene that the CardboardReticle can select and interact with in GCVR by marking them with the "Interactive" layer. Since the CardboardReticle casts a ray to determine if an object is looked at, we must ensure that interactive objects have properly configured static collider[48] components.

To create teleportation platforms that will allow us to change positions around the scene, we add five simple cube objects to the scene, and write a script that will provide the actual teleportation functionality in GCVR. We define special public methods in GCVR interaction scripts which are called by GCVR when specific conditions are met, as in listing 4 below. We then attach this TeleportPlatform.cs script to each of the five cube objects we wish to use as teleporters. Note that the XR Rig is not a direct parent, child, nor sibling object to these platforms, so we make it a public field for the user to initialise with the CharacterController they wish to be targeted by the teleportation script.

```
/*
  @author: Michael Makarenko (Zugidor)
  @date: 19 March 2024
*/

using UnityEngine;

public class TeleportPlatform : MonoBehaviour
{
  // The CharacterController component (movement) of the XR Rig to teleport
(move)
  public CharacterController rig;
  Renderer platform;
  Color ogColour;
  void Start()
  {
    // Get the renderer of the platform
    platform = GetComponent<Renderer>();
    // Store the original base map colour of the platform
    ogColour = platform.material.color;
  }
  private void TeleportXRRig()
  {
    // Teleport the Rig to this platform
    rig.Move(new Vector3(transform.position.x, rig.transform.position.y,
transform.position.z) - rig.transform.position);
  }

  // OnPointer methods called by CardboardReticle

  public void OnPointerEnter()
  {
    // When the platform is looked at, change its color
    platform.material.color = Color.red;
  }
  public void OnPointerExit()
  {
    // When the platform is no longer looked at, change it back
    platform.material.color = ogColour;
  }
  public void OnPointerClick()
  {
    // When active platform is clicked, teleport the XR Rig
    TeleportXRRig();
  }
}
```

Listing 4: Unmodified contents of TeleportPlatform.cs, comments provide all necessary explanation of the code

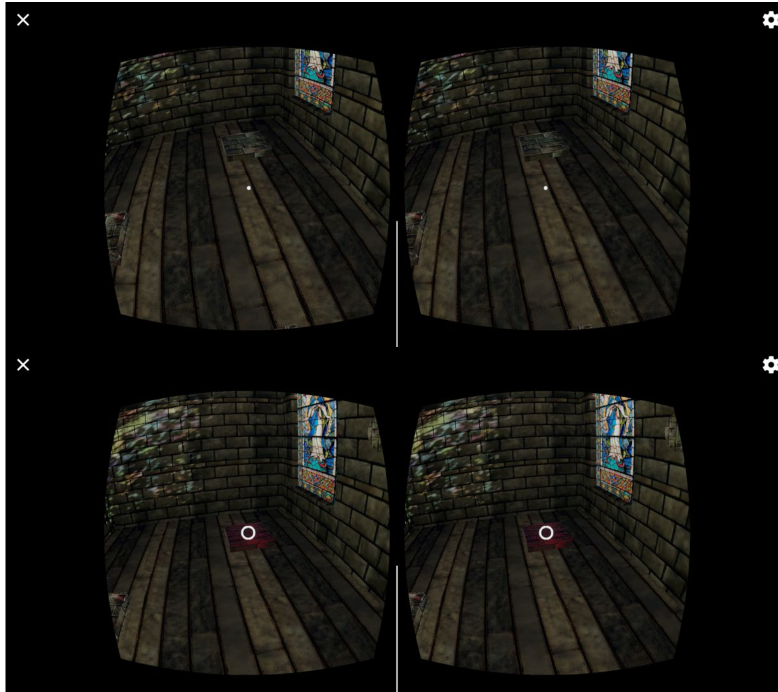

Figure 3.23: Screenshots in GCVR of an unselected (top) and selected (bottom) teleport platform

In order to allow the user to view the windows from different points of elevation, we add cylinders to a wall of the scene to function as buttons to lift the previously added platforms. We write an ElevateButton.cs script to execute this desired functionality and attach it to each of the two buttons in the prototype scene. As before, we make the CharacterController public as we must elevate the rig with the platforms. We also declare the parent object of all the platforms public to easily select them all at once for the position transform. The rig's position is transformed with Move() as before, and the platforms are elevated by assigning a new position 0.8 units higher in the Y direction, as the cubes in our prototype scene were initially placed 0.8 units into the ground. Sometimes, the platform's collider may interfere with lowering the rig, so we temporarily disable the platform's colliders in the grounding method. State is synced between the buttons by updating a boolean value across all instances of ElevateButton.cs every time any of the buttons is activated. We also have a boolean to indicate that a button has been pressed, in order to prevent the button from activating several times a second as OnPointerClick() gets called by GCVR every frame.

```
public class ElevateButton : MonoBehaviour{
  public CharacterController rig;
  public GameObject elevatorParent;
  bool grounded = true;
  bool pressed = false;
  float rigElevatedY, rigGroundedY, elevatorElevatedY, elevatorGroundedY;
  void Start(){
    rigGroundedY = rig.transform.position.y;
    rigElevatedY = rigGroundedY + 0.8f;
    elevatorGroundedY = elevatorParent.transform.GetChild(0).position.y;
    elevatorElevatedY = elevatorGroundedY + 0.8f;
  }
  private void Elevate(){
    rig.Move(new Vector3(rig.transform.position.x, rigElevatedY,
     rig.transform.position.z) - rig.transform.position);
    for (int i=0; i<elevatorParent.transform.childCount; i++){
      elevatorParent.transform.GetChild(i).position =
       new Vector3(elevatorParent.transform.GetChild(i).position.x,
       elevatorElevatedY, elevatorParent.transform.GetChild(i).position.z);
    }
    grounded = false;
  }
  private void Ground(){
    for (int i=0; i<elevatorParent.transform.childCount; i++){
      elevatorParent.transform.GetChild(i).position =
       new Vector3(elevatorParent.transform.GetChild(i).position.x,
       elevatorGroundedY, elevatorParent.transform.GetChild(i).position.z);
      elevatorParent.transform.GetChild(i)
       .GetComponent<BoxCollider>().enabled = false;
    }
    rig.Move(new Vector3(rig.transform.position.x, rigGroundedY,
     rig.transform.position.z) - rig.transform.position);
    grounded = true;
    for (int i=0; i<elevatorParent.transform.childCount; i++){
      elevatorParent.transform.GetChild(i)
       .GetComponent<BoxCollider>().enabled = true;
    }
  }
  public void OnPointerExit(){
    pressed = false;
  }
  public void OnPointerClick(){
    if (!pressed){
      if (grounded)
        Elevate();
      else
        Ground();
      transform.parent.GetChild((transform.GetSiblingIndex() + 1) % 2)
       .GetComponent<ElevateButton>().grounded = grounded;
      pressed = true;
    }
  }
}
```

Listing 5: The contents of ElevateButton.cs, with all comments and previously seen code removed, such as highlighting the button red, and most whitespace trimmed

The final interactive element present in the prototype scene is three identical buttons placed above each window which activates or deactivates the movement of the directional light, or sun, around the scene. The sun's animation of orbiting the scene is implemented using Unity's animation system. The directional light is manually rotated, and key positions are saved as keyframes in an animation clip. This clip is then included in a state machine called an animator controller[48]. We define an empty node with no animation as default and conditional state transitions to and from the SunRotation animation clip based on the SunButtonPress parameter. This is a "trigger" type parameter, which is effectively a boolean that is false by default and resets to false whenever a state transition conditional on the trigger is executed. When SunButtonPress is triggered during Empty, the animation clip begins, and the sun begins rotating around the scene. Triggering SunButtonPress in this state interrupts the clip, restoring the sun to its original position, transitioning to the red exit node which automatically loops the state machine back to the entry node and leads back to the default transition to the Empty node. In SunButton.cs, we declare the sun object public for the user to supply the relevant directional light that has an Animator component attached to it. We use this component's SetTrigger() method to activate state transitions in the animator controller. State is managed by the animator controller and SunButtonPress trigger, so we do not sync a boolean between buttons.

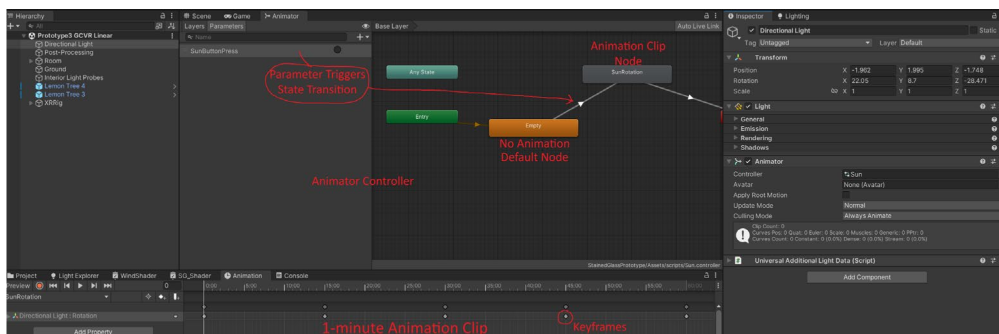

Figure 3.24: The primary Unity Editor windows used in an animation workflow

Figure 3.25: The five teleport platforms, two elevation buttons and three sun buttons

```
/*
  @author: Michael Makarenko (Zugidor)
  @date: 20 March 2024
*/

using UnityEngine;

public class SunButton : MonoBehaviour
{
  // The Sun directional light to rotate (about y-axis only, continuously
    until pressed again)
  public GameObject sun;
  Animator sunAnim;
  Renderer button;
  Color ogColour;
  bool pressed = false;
  void Start()
  {
    // Get the renderer of the button
    button = GetComponent<Renderer>();
    // Store the original base map colour of the button
    ogColour = button.material.color;
    // Get the animator of the sun
    sunAnim = sun.GetComponent<Animator>();
  }
  // OnPointer methods called by CardboardReticle
  public void OnPointerEnter()
  {
    // Change the button's colour to indicate it's being looked at
    button.material.color = Color.red;
  }
  public void OnPointerExit()
  {
    // Change the button's colour back to its original colour
    button.material.color = ogColour;
    // Reset pressed to false
    pressed = false;
  }
  public void OnPointerClick()
  {
    // If the button is pressed, toggle animation
    if (!pressed)
    {
      sunAnim.SetTrigger("SunButtonPress");
      pressed = true;
    }
  }
}
```

Listing 6: Contents of SunButton.cs almost entirely unmodified except for the removal of some whitespace. Comments provide all necessary explanation of the code

3.2.7 Deployment

We can use the MockHMD XR Unity package to simulate the stereo rendering and occlusion mesh of a VR headset[60]. This plugin is often used to assist in development for VR without a VR headset. "Initialize XR on Startup" and "Mock HMD Loader" must be enabled in "Project Settings > XR Plug-in Management > Windows, Mac, Linux Settings". We can now click the play button near the top-centre of the Unity Editor to enter Game View, which will use the "Both Eyes" render mode by default, displaying two offset frames for each eye side by side. Selecting the drop-down menu allows us to select the "Occlusion Mesh" render mode, as well as either the left or right eye individually if desired.

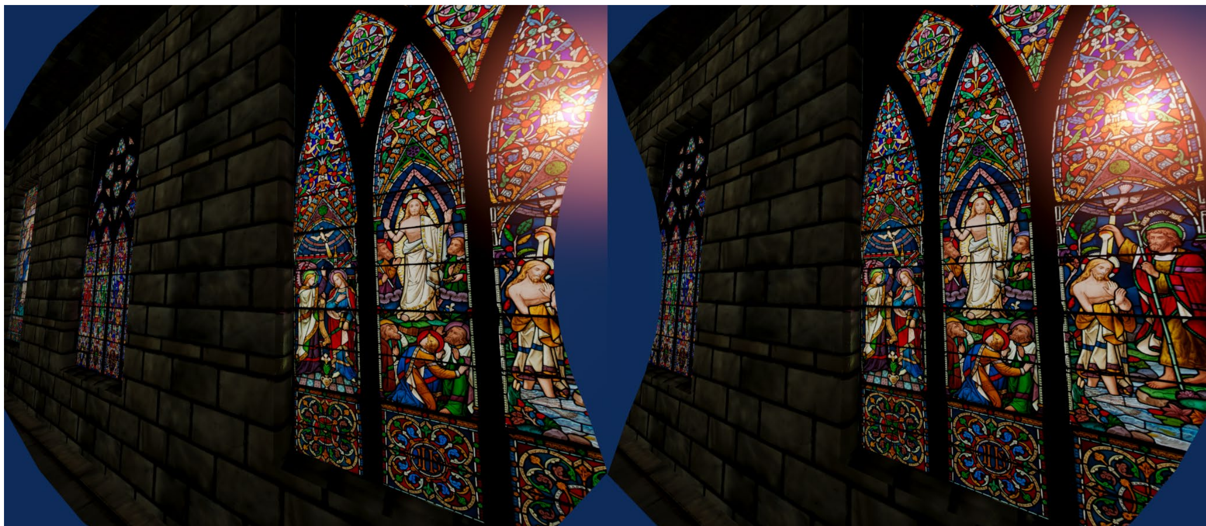

Figure 3.26: Game View in the Occlusion Mesh render mode provided by MockHMD

We refer back to the GCVR quickstart guide[59] to configure our build settings correctly for deploying to Android. In "File > Build Settings" we select Android and click "Switch Platform" and, with the "Prototype3 GCVR" scene open, we click "Add Open Scenes". Since our prototype is composed of a single scene that is never changed to another, we ensure only the one desired scene is added and enabled here. Below are the specific settings in "Project Settings > Player > Android Settings" used in this prototype that differ from or are not mentioned in the quickstart guide.

- Resolution and Presentation – Default Orientation: The guide leaves the choice between Landscape Left or Right up to the reader; we use Landscape Left i.e. rotating the phone anti-clockwise from portrait.
- Other Settings – Rendering:

    - We disable Auto Graphics API and manually ensure that only OpenGLES3 is present in the Graphics APIs list.
    - To limit support to only modern Android mobile GPUs that are most likely to run the VR app without issue, we enable “Require ES3.2”.
    - Texture compression format should be ASTC, which is newer and more efficient than ETC2.
    - Ensure that “Allow HDR Display Output” and “Use HDR Display Output” are disabled, as they negatively impact performance and result in an extremely dark interior scene.
- Other Settings – Identification: If “Target API Level” is set to “Automatic: Highest Installed”, this must be changed and manually set to “Android 13.0 (API Level 33)” or higher, as the former setting does not install and use API Level 33 or higher by default.
- Other Settings – Configuration: Again, to support only modern phones and avoid having to deal with technical issues on older or low-end mobile devices, we only enable “ARM64” under “Target Architectures” and leave “ARMv7” disabled.

All other settings should follow as specified in the aforementioned guide or left untouched at their defaults. We can now connect our Android test device to the computer via USB cable and click “Build And Run” at the bottom of the Build Settings window. Depending on the device, the application may not look as desired, and certain settings will need to be altered. Throughout the development of the prototype, several final changes were made to improve the final GCVR output. We adjusted the range, intensity, and indirect multiplier properties of the point light of the interior ceiling lamp; tweaked the stained glass window material properties, particularly the brightness, thickness, and transparency inputs; and corrected the post-processing bloom intensity. While we are here, we ensure that “High Quality Filtering” is disabled in bloom settings, as this is recommended to minimise the negative performance impact on mobile devices[55].

4 Results

Below, we present and evaluate our resulting application in terms of realism and real-time performance as we developed the project with the aims of pushing the former as much as possible within the constraints of the latter.

4.1 Visual Fidelity

To convey moving aspects of the prototype, such as the trees swaying in the wind or the sun moving across the sky, an illustrative video was recorded and can be found in the aforementioned GitHub repository associated with this prototype. This video demonstrates visuals both on desktop in Game View with MockHMD's Occlusion Mesh and in GCVR on an Asus ROG Phone 5 with a screen resolution of 1080×2448 pixels, running Android 13 using the process and settings described in this thesis. We compare our rendered results with photographs of stained glass lighting phenomena as below.

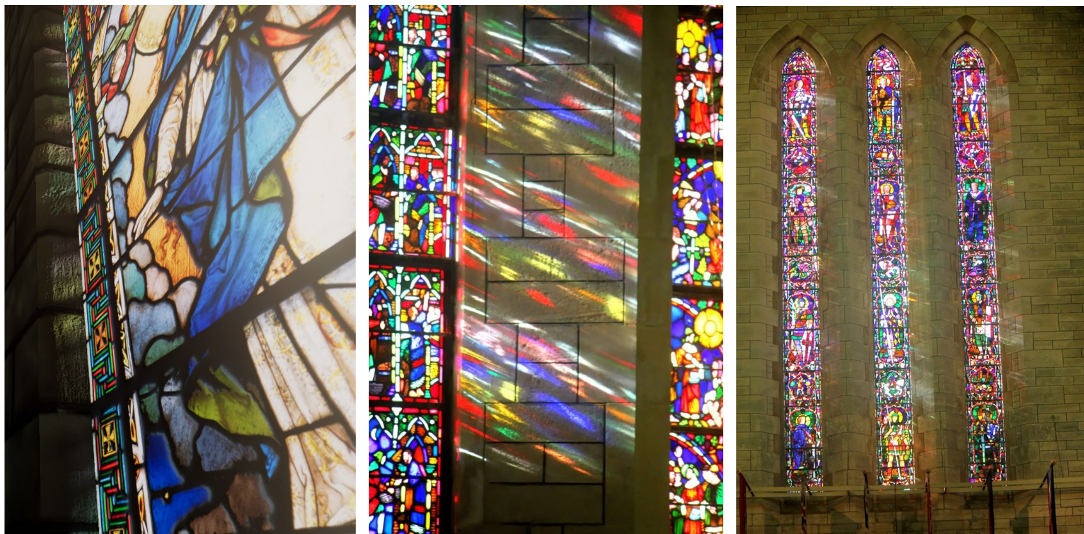

Figure 4.1: One of the three stained glass windows in our prototype (left) and two photographs (centre, right) of stained glass windows by Unknown on their public web blog[x], highlighting the coloured shadows of stained glass

[x] https://lookingforsearching.blogspot.com/2014/02/stainglass-shadows.html

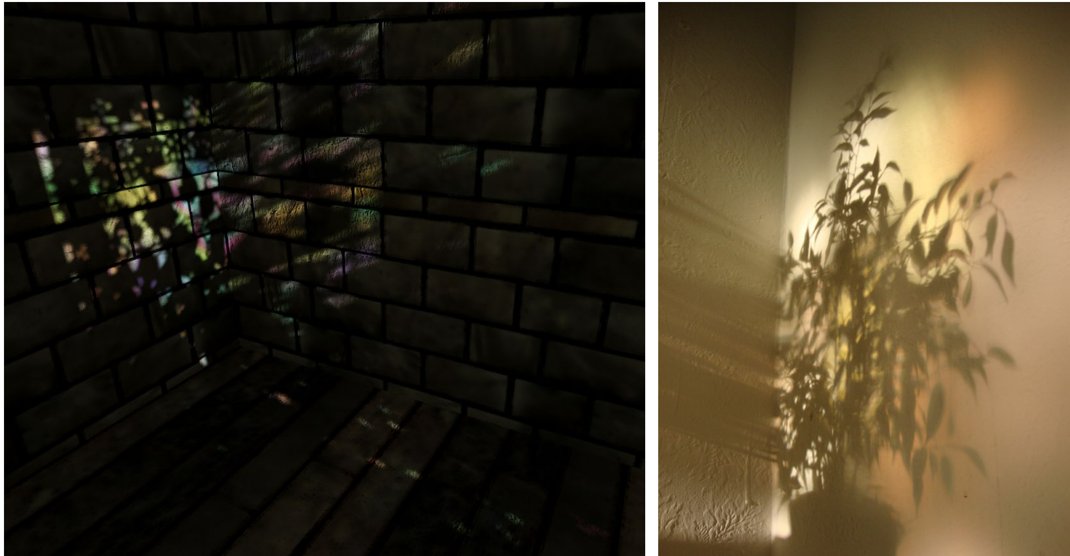

Figure 4.2: Comparing our prototype's rendering of stained glass coloured shadows occluded by vegetation shadow (left), and a photograph of the real-world occurrence by Katja Linders on Pinterest[xi] (right)

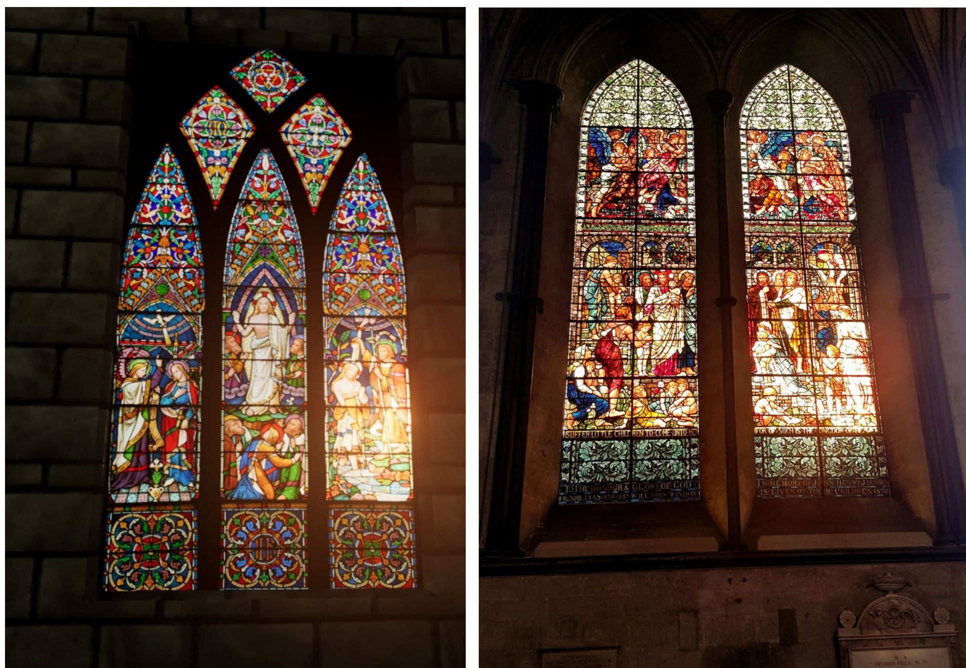

Figure 4.3: Comparing our prototype's rendering of a stained glass window illuminated by sunlight exhibiting bloom/overexposure (left), and a real-world example photographed by a deleted user on an archived Reddit post[xii] (right)

[xi] https://www.pinterest.com/pin/300474606360173969/
[xii] https://www.reddit.com/r/Catholicism/comments/agxpae/

We obtain these results in mere minutes by adjusting the material settings exposed to the user and made easily accessible in the Unity Editor UI by the SG_Shader, and by moving the tree models or directional light as needed, preparing the prototype scene to be quickly built and deployed with the desired appearance. Unfortunately, the visuals of the application in GCVR are visibly downgraded compared to the results seen in Unity's Scene or Game View. We can observe lower resolution shadows and lighting with a visibly lower contrast between light and shadow, as illustrated in Fig. 4.4 below.

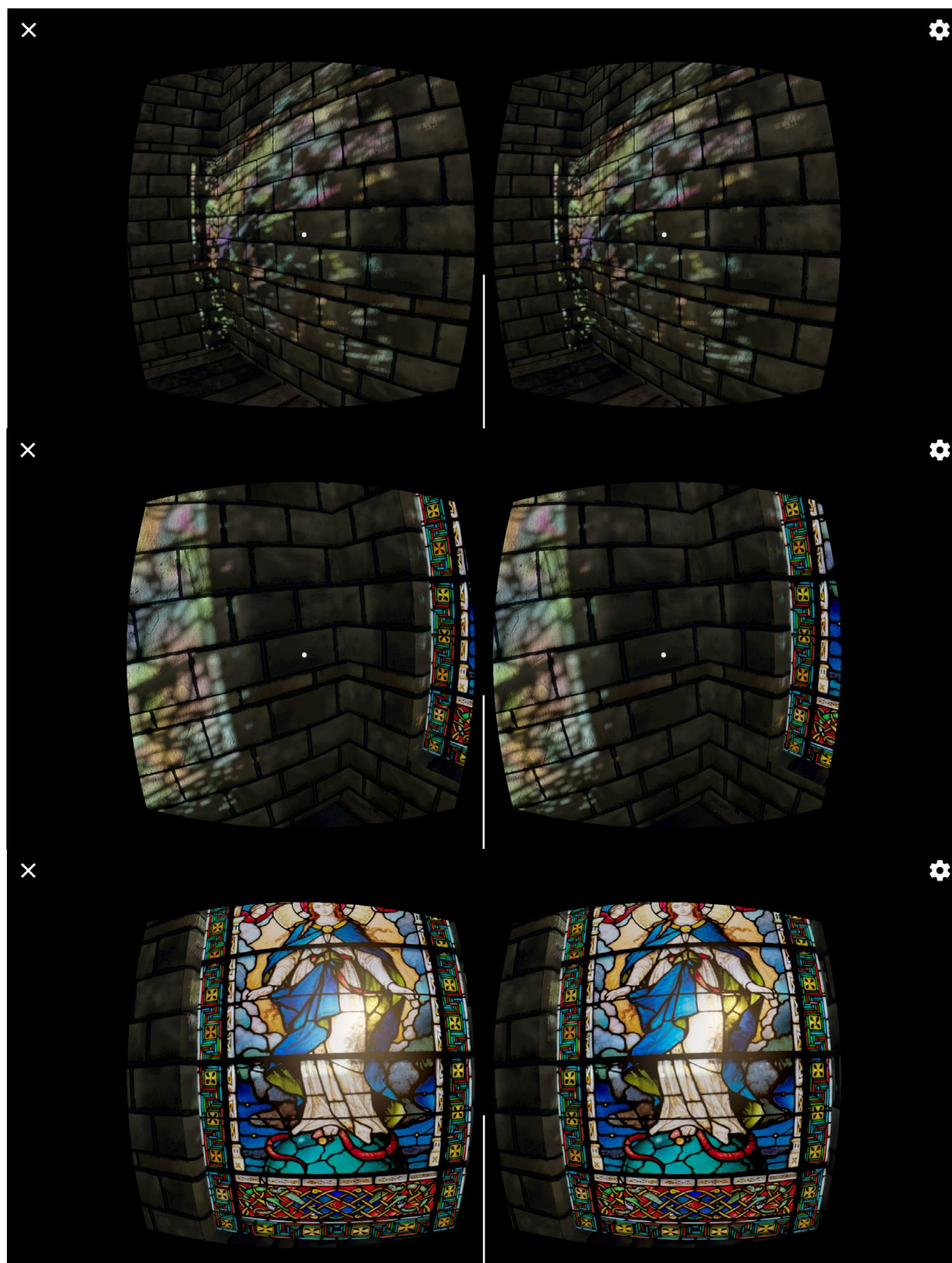

Figure 4.4: Three GCVR screenshots displaying more pixelated shadows and lower lighting contrast/dynamic range

4.2 Performance

To measure our prototype's runtime performance, we can make use of Unity's profiling tools[48]. In the Build Settings window, we must enable "Development Build" and "Autoconnect Profiler" before clicking "Build And Run" with the target device connected by USB cable. When the application launches on the target device, the Profiler window should open in Unity, which can also be manually opened by navigating to "Window > Analysis > Profiler". Here, as long as the circular red "Record Profiling" button is enabled, we can observe the most recent 300 – 2000 frames depending on the respective setting in Unity Preferences. The profiler records frametimes in milliseconds, that is, how long it takes to display a given frame; we will be converting to frames per second (FPS) by dividing 1000 by the frametime, as FPS is the more common and intuitive metric. While 24 FPS is the standard for film, 30 FPS is commonly considered the minimum acceptable framerate in real-time rendering applications, with 60 FPS or higher being ideal for delivering a smooth end user experience on a traditional display. Most standard modern displays such as computer monitors and television screens have a refresh rate, which is the number of times a new image is rendered by the display every second, of 60 Hertz. A high framerate is made doubly important given that we are targeting VR, as a low framerate may result in motion sickness. 90 FPS is considered the ideal standard for smooth VR experiences, and the majority of VR headsets on the market have a refresh rate of 90 Hertz. Framerates as low as 60 or 72 are generally deemed acceptable but suboptimal[61]. Most common mobile applications consume 130 – 500 megabytes of memory during use[62]. This gives us the necessary context to understand whether our prototype application's resource usage and performance are satisfactory.

Our test device is a capable and relatively modern 2021 phone equipped with a Qualcomm Snapdragon 888 chip and 16 gigabytes of memory. The Unity profiler does not calculate average, maximum, nor minimum frametimes for us. We recorded an average framerate of approximately 59 FPS, with a maximum of 165 and a minimum of 34. The framerate often hovers in the 55 – 65 range with occasional spikes to beyond 70 and rare troughs down to the 40s. The worst offending falls in framerate are explained by inspecting the profiler and finding that the profiler process itself is taking up substantial amounts of frametime at certain moments, as seen in Fig. 4.5.

We can therefore confidently assume that the minimum FPS during regular use is higher and likely in the 40s.

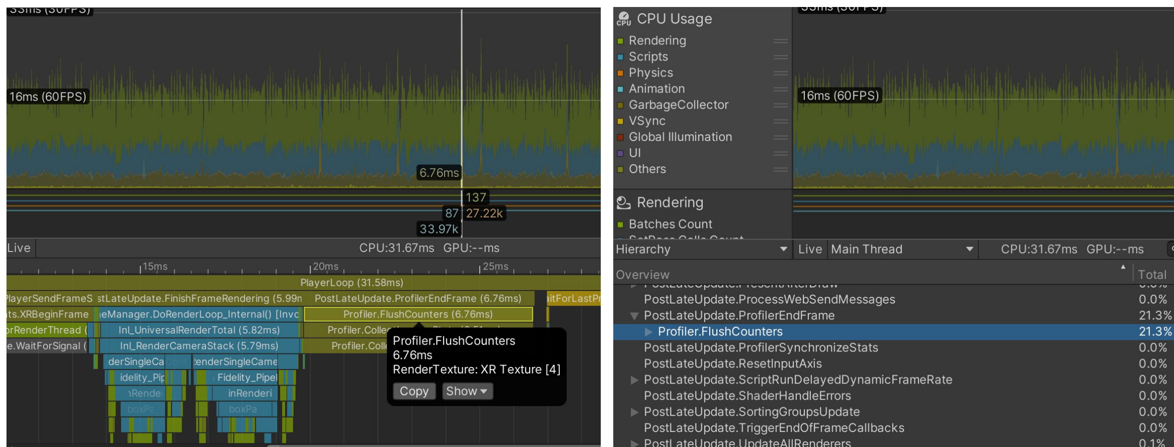

Figure 4.5: Investigating the worst frametime spikes in the Unity Profiler window while the application runs on the target device. Unity's Profiler process is revealed to consume over 21% of the frametime of some frames. Note that the GPU and CPU are integrated on mobile chips, so GPU usage is also reported as CPU usage

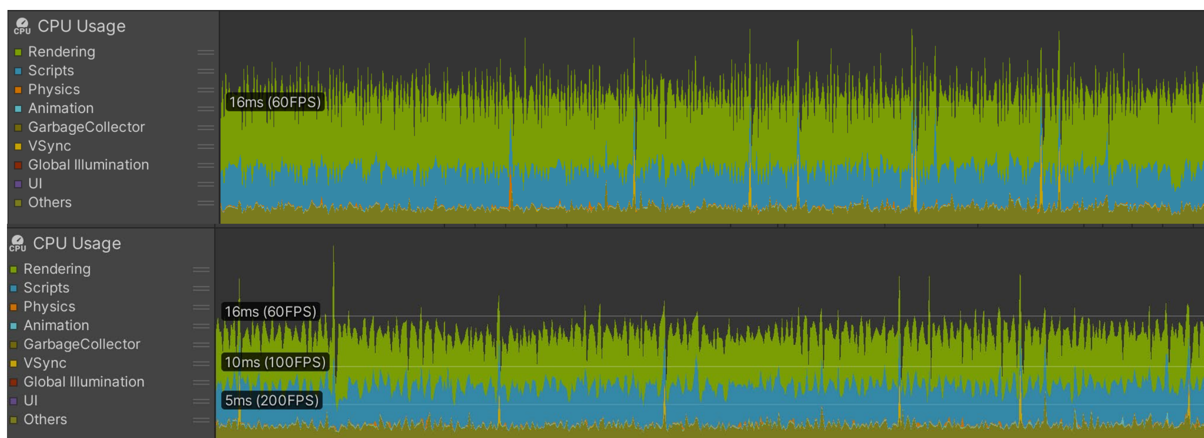

Figure 4.6: Frametimes in Unity Profiler are shown to be worse when observing the stained glass windows (top) and better when looking away towards the walls of the interior (bottom), revealing the most performance-intensive parts of our scene to be the windows and what is visible beyond them

With regard to memory utilisation, we can use both the regular Profiler window and the dedicated Memory Profiler[48] to examine information on the contents of the application's working memory. Where the Profiler shows memory usage on a per-frame basis, the Memory Profiler allows us to take snapshots of the target device's working memory for more thorough analysis. It should be noted that taking a memory snapshot can temporarily reduce framerate to below 15 FPS due to

the slow copying of target device memory contents to the computer, which may take approximately 200 milliseconds; we can safely disregard this outlier. Total application memory usage is found to range between approximately 400 megabytes and 450 megabytes. This variation is ascribed to memory management factors out of our control, as Unity Object memory usage is constant at 163.3 megabytes between snapshots of memory.

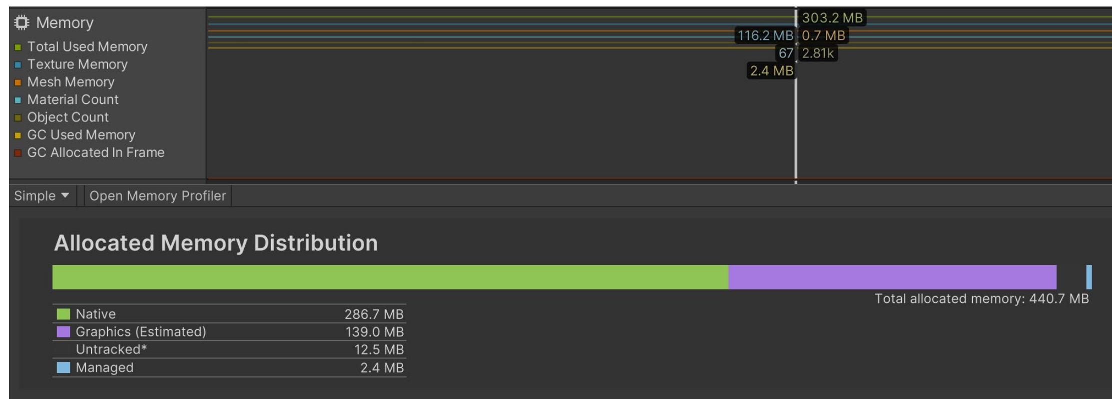

Figure 4.7: The memory section of the built-in Profiler

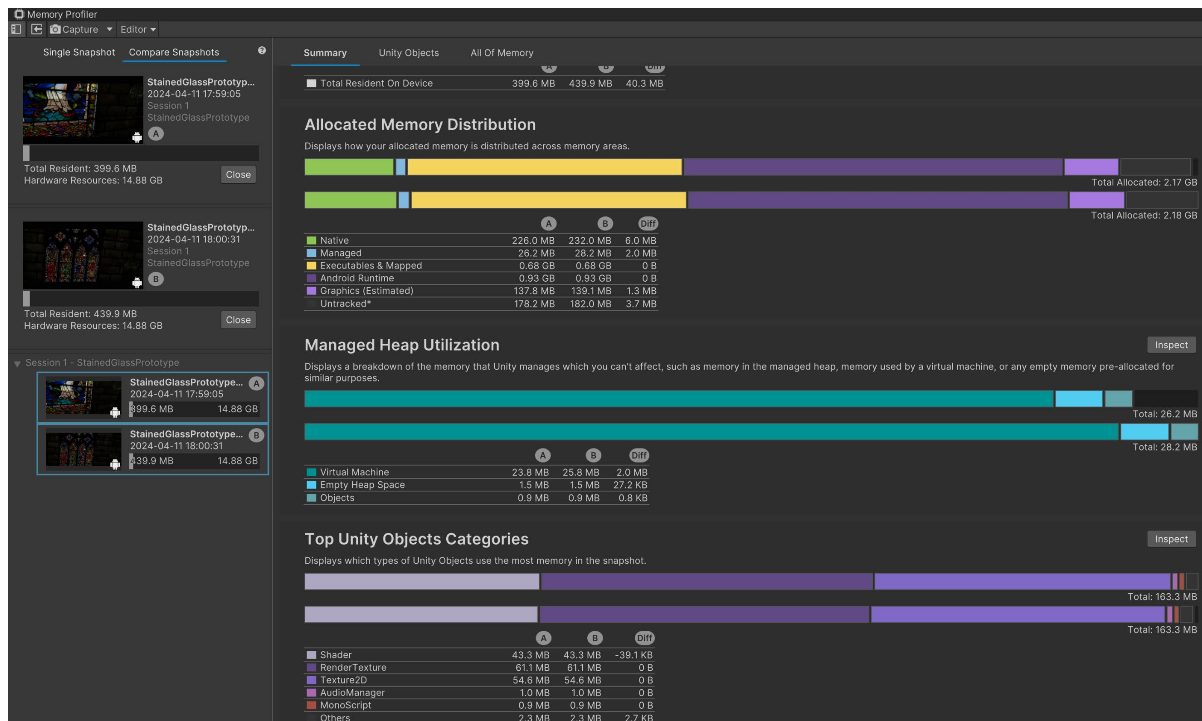

Figure 4.8: Comparing two different snapshots of the test device's memory in the Memory Profiler. Note that "Total Allocated" memory as labelled here is defined differently from that seen in Fig. 4.7, can safely exceed available device memory, and is not indicative of application memory requirements

4.3 Ablation Study

To investigate the performance impact of the exterior objects, the grass textured ground plane and two trees equipped with the Wind Shader, we perform a profiling run on a build with these objects deleted from a duplicate scene. We observe an uplift in performance and so decide to also examine the impact of the interior and all visible objects as a whole. To this end, we create three additional duplicate scenes, one with only the interior composed of the room, windows, and ceiling lamp deleted; another with all objects deleted save for the XR Rig, Directional Light, and Post-Processing Volume, which are present in all scenes except for the last; the emptiest scene contains only the XR Rig and the default skybox illuminated by static ambient light. We provide the observed results for direct comparison in table 2 below. From these results, we can extrapolate:

- The interior is more resource intensive than exterior elements with regards to both graphics and memory.
- The primary Unity Objects composing the scene consume relatively little memory, at approximately 17% of total application memory.
- Framerate maxima vary wildly and are not particularly reliable performance indicators.
- Post-Processing and a Directional Light in an otherwise empty scene have little to no performance overhead at all, with average FPS and memory usage falling within a margin of error of ±3%.
- A Unity application deployed to GCVR appears to be performance intensive by default. As the test device has a refresh rate of 144 Hertz, we ensured that Vertical Sync, a feature that caps the maximum framerate to the refresh rate of the device's screen, was off during this ablation study. Despite this, we only observed average framerates slightly above 200 FPS in an empty scene devoid of detail on a modern and powerful mobile device.

	Framerate (Frames Per Second)			Memory Usage (Megabytes)	
	minimum	average	maximum	minimum	maximum
Full	34	59	165	400	450
No Exterior	45	77	120	386	432
No Interior	56	86	156	365	403
Post & Sun	91	212	329	335	373
Empty	84	216	281	338	373

Table 4.1: Comparing performance of four scenes with varying Unity Objects present or missing. "Full" is the control prototype scene with no removed Objects, "Post & Sun" combines the removals of both "No Exterior" and "No Interior", and "Empty" removes even the directional light and post-processing present in "Post & Sun". Data is obtained using Unity Profiler and Unity Memory Profiler at application runtime, with frametimes converted to framerates rounded to the nearest frame per second. Memory usage rounded to the nearest megabyte

5 Conclusion

We conclude this study by laying out the contributions of this project, summarising its limitations, and proposing potential work to extend the design presented here.

5.1 Contributions

In this dissertation, we have documented a novel approach to digitising stained glass windows by rendering them in real-time within a three-dimensional environment viewable in virtual reality, with digital photographs as the sole prerequisite input. We reviewed existing literature related to the topic at hand and framed the context within which this work has been conducted. In the process of developing this workflow, we performed an investigation of the viability of real-time 3D graphics and interactive VR technologies for near-photorealistic simulations of stained glass windows. A proof-of-concept prototype application has been built as part of this research using the Unity 3D development engine and deployed to Google Cardboard VR on an Android mobile device. The Unity project that the application was built from has been made available on a public GitHub repository and is accompanied by a demonstration video. The workflow defined in this thesis has been evaluated by analysing the resulting application's frametime performance and memory usage on a high-end smartphone released in 2021.

5.2 Limitations & Challenges

As we have worked with optimisations for mobile VR in mind, we had to compromise on realistic yet performance-intensive graphics techniques such as real-time raytracing and volumetric lighting. Indeed, as noted in section 3.1, our choice of SRP in Unity was informed by this consideration, and some such techniques, such as raytracing, are natively supported only in HDRP. We describe in section 4.2 how profiling this application revealed suboptimal, albeit usable, framerate performance and satisfactory memory usage on the target device. Visually, the scene is shown to be impressive within Unity Editor on a PC, but significantly less so in mobile VR. This may put the viability of mobile VR for accessible real-time simulation of stained glass into question due to the limitations of modern mobile hardware for the time being. A limitation of this study is also the fact that we only tested one device; chipsets that are more powerful than that of the test device already exist as of the time of writing.

Additionally, there exist a vast array of non-smartphone based VR devices, both tethered and untethered, that may provide an improved experience. Within the application, stained glass windows are modelled as rectangular planes in 3D space placed in simple rectangular cutouts in the interior room, resulting in jarring, flat black areas where there should be three-dimensional extruded sills or frames. Beyond the application itself, the image-to-material pipeline described in section 3.2.4 is undermined by the length of time, approximately one hour, necessary to process a single image into the required three textures to create stained glass window and shadow mask materials.

5.3 Future Work

We hope for the above challenges to be understood as an encouragement to conduct further examination and improvement on the work presented here, with the aim of contributing to the preservation and public viewing of stained glass artworks by way of digital architectural simulation.

We suggest investigation into a method to resolve the issue of tedious manual editing of images to effectively automate the process and allow for the mass-conversion photographs into high quality stained glass material textures and associated shadow masks, perhaps with the use of computer vision techniques. This challenge may also be tackled by switching from the use of digital photographs to detailed scans informing of the physical material properties of individual stained glass windows. We additionally note the possibility of using photogrammetry or emerging AI technologies to create high-quality 3D models of sills and frames for individual non-rectangular stained glass windows.

It may be worthwhile to attempt to apply the high-level workflow and principles from chapter 3 in a different render pipeline such as HDRP or even a wholly separate development engine such as Unreal Engine with a visual fidelity-first rather than performance-first focus. This idea naturally lends itself to deploying and testing on high-end VR platforms that provide superior experience to mobile VR, albeit at the cost of mass public accessibility. A more conservative approach may be to refine the Unity prototype presented here and test deploying to a variety of target VR devices, making use of Unity's strength of wide platform cross-compatibility.

Bibliography

[1] N. H. J. Westlake, *A History of Design in Painted Glass*. United Kingdom: J. Parker and Company, 1881.

[2] A. Verney-Carron *et al.*, "Alteration of medieval stained glass windows in atmospheric medium: review and simplified alteration model," *npj Materials Degradation,* vol. 7, no. 1, 2023, Art no. 49, doi: 10.1038/s41529-023-00367-0.

[3] E. M. Maingi *et al.*, "Challenges in laser cleaning of cultural heritage stained glass," *Journal of Physics: Conference Series,* vol. 2204, no. 1, 2022, Art no. 012079, doi: 10.1088/1742-6596/2204/1/012079.

[4] "The Stained Glass Museum Virtual Tour." The Stained Glass Museum. https://stainedglassmuseum.com/virtualtour.php (accessed 2024).

[5] M. Rouse. "What is Rendering? - Definition from Techopedia." Techopedia. https://www.techopedia.com/definition/9163/rendering (accessed 2024).

[6] G. Leach. "Lecture: Graphics pipeline and animation." RMIT University Australia. https://web.archive.org/web/20171207095603/http://goanna.cs.rmit.edu.au/~gl/teaching/rtr%263dgp/notes/pipeline.html (accessed 2024).

[7] "What is a Rendering Engine?" AR Visual. https://arvisual.eu/dictionary/rendering-engine/ (accessed 2024).

[8] Y.-C. Tian and D. C. Levy, Eds. *Handbook of Real-Time Computing*. Springer Nature Singapore, p. 734.

[9] H. W. Jensen, "A practical guide to global illumination using ray tracing and photon mapping," presented at the ACM SIGGRAPH 2004 Course Notes, Los Angeles, CA, 2004.

[10] "What is a 3D material?" Foundry Visionmongers Limited. https://www.foundry.com/insights/design/3d-materials-explained (accessed 2024).

[11] "What is 3D texturing?" Adobe Inc. https://www.adobe.com/products/substance3d/discover/3d-texturing.html (accessed 2024).

[12] "What is normal mapping?" Adobe Inc. https://www.adobe.com/products/substance3d/discover/normal-mapping.html (accessed 2024).

[13] P. Novaković, M. Hornak, M. Zachar, and N. Joncic, *3D Digital Recording of Archaeological, Architectural and Artistic Heritage*. University of Ljubljana, Faculty of Arts, 2017.

[14] N. Kowshik. "Normal Mapping." California Polytechnic State University, San Luis Obispo. https://users.csc.calpoly.edu/~zwood/teaching/csc473/finalw10/nkowshik/ (accessed 2024).

[15] J. d. Vries, *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion.* Kendall & Welling, 2020.

[16] D. Mould, "A Stained Glass Image Filter," in *Eurographics Workshop on Rendering*, P. Dutre, F. Suykens, P. H. Christensen, and D. Cohen-Or, Eds., 2003: The Eurographics Association, in 14th Eurographics Symposium on Rendering, doi: 10.2312/EGWR/EGWR03/020-025.

[17] S. Brooks, "Image-Based Stained Glass," *IEEE Transactions on Visualization and Computer Graphics,* vol. 12, no. 6, pp. 1547-1558, 2006, doi: 10.1109/TVCG.2006.97.

[18] V. Setlur and S. Wilkinson, "Automatic Stained Glass Rendering," in *Advances in Computer Graphics*, Hangzhou, T. Nishita, Q. Peng, and H.-P. Seidel, Eds., 2006: Springer-Verlag Berlin Heidelberg, pp. 682-691, doi: 10.1007/11784203_66.

[19] S. Seo, H. Lee, H. Nah, and K. Yoon, "Stained Glass Rendering with Smooth Tile Boundary," in *Computational Science – ICCS 2007*, Beijing, Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds., 2007: Springer-Verlag Berlin Heidelberg, pp. 162-165, doi: 10.1007/978-3-540-72586-2_23.

[20] L. Doyle and D. Mould, "Painted Stained Glass," in *Computational Aesthetics*, A. Forbes and L. Bartram, Eds., 2016: The Eurographics Association, in Workshop on Computational Aesthetics, doi: 10.2312/exp.20161058.

[21] D. Kang, T. Lee, Y.-H. Shin, and S. Seo, "Video-based Stained Glass," *KSII Transactions on Internet and Information Systems,* vol. 16, no. 7, pp. 2345-2358, 2022, doi: 10.3837/tiis.2022.07.012.

[22] J. Shin, "Modeling Stained Glass," Swarthmore College, 2005. [Online]. Available: https://www.swarthmore.edu/sites/default/files/assets/documents/engineering/js_report_final.pdf

[23] J.-A. Kim, S. Ming, and D. Kim, "A Realistic Illumination Model for Stained Glass Rendering," in *ICVR*, Beijing, R. Shumaker, Ed., 2007: Springer-Verlag Berlin Heidelberg, pp. 80-87, doi: 10.1007/978-3-540-73335-5_9.

[24] N. Thanikachalam, L. Baboulaz, P. Prandoni, S. Trümpler, S. Wolf, and M. Vetterli, "VITRAIL: Acquisition, Modeling, and Rendering of Stained Glass," *IEEE Transactions on Image Processing,* vol. 25, no. 10, pp. 4475-4488, 2016, doi: 10.1109/TIP.2016.2585041.

[25] N. Thanikachalam, "Image Based Relighting of Cultural Artifacts," Docteur és Sciences, Faculté Informatique et Communications, École Polytechnique Fédérale de Lausanne, Laboratoire de Communications Audiovisuelles, 6990, 2016. [Online]. Available: https://infoscience.epfl.ch/record/218529

[26] M. Rahrig and M. Torge, "3D Inspection of the Restoration and Conservation of Stained Glass Windows Using High Resolution Structured Light Scanning," *Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci.,* vol. XLII-2/W15, pp. 965-972, 2019, doi: 10.5194/isprs-archives-XLII-2-W15-965-2019.

[27] A. Babini, S. George, T. Lombardo, and J. Y. Hardeberg, "Potential and Challenges of Spectral Imaging for Documentation and Analysis of Stained-Glass Windows," *Proc. IS&T London Imaging Meeting 2020: Future Colour Imaging,* vol. 1, pp. 109-113, 2020, doi: 10.2352/issn.2694-118X.2020.LIM-27.

[28] A. Babini, T. Lombardo, K. Schmidt-Ott, S. George, and J. Y. Hardeberg, "Acquisition strategies for in-situ hyperspectral imaging of stained-glass windows: case studies from the Swiss National Museum," *Heritage Science,* vol. 11, no. 1, 2023, Art no. 74, doi: 10.1186/s40494-023-00923-6.

[29] J. Lucas, "Rapid development of Virtual Reality based construction sequence simulations: a case study," *Journal of Information Technology in Construction (ITcon),* vol. 25, pp. 72-86, 2020, Art no. 4, doi: 10.36680/j.itcon.2020.004.

[30] S. Kim, Z. Rybkowski, and H. D. Jeong, "Developing and Testing Computer- and Virtual Reality-Based Target Value Design Simulations," in *Proceedings of the 31st Annual Conference of the International Group for Lean Construction*

(IGLC31), Lille, France, 2023, Lille, France, pp. 629-638, doi: 10.24928/2023/0194.

[31] P. Patel and S. Khan, "Review on Virtual Reality for the Advancement of Architectural Learning," in *2023 IEEE Renewable Energy and Sustainable E-Mobility Conference (RESEM)*, Bhopal, India 2023: IEEE, doi: 10.1109/RESEM57584.2023.10236123.

[32] Z. Feng, V. A. González, R. Amor, R. Lovreglio, and G. Cabrera-Guerrero, "Immersive virtual reality serious games for evacuation training and research: A systematic literature review," *Computers & Education,* vol. 127, pp. 252-266, 2018, doi: 10.1016/j.compedu.2018.09.002.

[33] W. Han, "Research on the Application of Virtual Reality Technology in the Integrated Design of Architectural Landscape," in *2021 Global Reliability and Prognostics and Health Management (PHM-Nanjing)*, Nanjing, China, 2021: IEEE, doi: 10.1109/PHM-Nanjing52125.2021.9613094.

[34] P. Shan and W. Sun, "Auxiliary use and detail optimization of computer VR technology in landscape design," *Arabian Journal of Geosciences,* vol. 14, no. 9, p. 798, 2021, doi: 10.1007/s12517-021-07131-1.

[35] D. Ververidis, S. Nikolopoulos, and I. Kompatsiaris, "A Review of Collaborative Virtual Reality Systems for the Architecture, Engineering, and Construction Industry," *Architecture,* vol. 2, no. 3, pp. 476-496, 2022, doi: 10.3390/architecture2030027.

[36] A. Ehab, G. Burnett, and T. Heath, "Enhancing Public Engagement in Architectural Design: A Comparative Analysis of Advanced Virtual Reality Approaches in Building Information Modeling and Gamification Techniques," *Buildings,* vol. 13, no. 5, 2023, Art no. 1262, doi: 10.3390/buildings13051262.

[37] A. Ehab and T. Heath, "Exploring Immersive Co-Design: Comparing Human Interaction in Real and Virtual Elevated Urban Spaces in London," *Sustainability,* vol. 15, no. 12, 2023, Art no. 9184, doi: 10.3390/su15129184.

[38] K. Nevin, "The Role of Light Variation in the Attending to and Memorisation of Stained-Glass Windows: An Eye Tracking and Behavioural Study.," M.Sc., School of Psychology, Trinity College Dublin, 2024. [Online]. Available: http://hdl.handle.net/2262/104878

[39] J. C. Snow and J. C. Culham, "The Treachery of Images: How Realism Influences Brain and Behavior," *Trends in Cognitive Sciences,* vol. 25, no. 6, pp. 506-519, 2021, doi: 10.1016/j.tics.2021.02.008.

[40] *Unity Engine*. Unity Technologies Inc. [Online]. Available: https://unity.com/products/unity-engine

[41] "Unity Documentation." Unity Technologies Inc. https://docs.unity.com/ (accessed 2024).

[42] "Unity Asset Store." Unity Technologies Inc. https://assetstore.unity.com/ (accessed 2024).

[43] United States Securities and Exchange Commission. (2020). *Form S-1 Initial Public Offering Registration Statement By Unity Software Inc.* [Online] Available: https://www.sec.gov/Archives/edgar/data/1810806/000119312520227862/d908875ds1.htm

[44] G. Nichols. "Google's DeepMind teams with leading 3D game dev platform Unity." ZDNET. https://www.zdnet.com/article/googles-deepmind-teams-with-leading-3d-game-dev-platform-unity/ (accessed 2024).

[45] J. Gaudiosi. "Why Valve's Partnership With Unity Is Important to Virtual Reality." FORTUNE. https://fortune.com/2016/02/11/valves-partners-with-unity/ (accessed 2024).

[46] *Unreal Engine*. Epic Games, Inc. [Online]. Available: https://www.unrealengine.com/

[47] R. Johns. "Unity vs Unreal." Hackr.io. https://hackr.io/blog/unity-vs-unreal-engine (accessed 2024).

[48] "Unity Manual." Unity Technologies Inc. https://docs.unity3d.com/Manual (accessed 2024).

[49] "Speed up debugging with Microsoft Visual Studio Code." Unity Technologies Inc. https://unity.com/how-to/debugging-with-microsoft-visual-studio-code (accessed 2024).

[50] "Enhance your prototype with ProBuilder." Unity Technologies Inc. https://learn.unity.com/tutorial/enhance-your-prototype-with-probuilder (accessed 2024).

[51] "Unity Shader Graph Package Manual." Unity Technologies Inc. https://docs.unity3d.com/Packages/com.unity.shadergraph@17.0/manual/index.html (accessed 2024).

[52] D. Shepherd. "Art, Light, and Awe: Uncovering the Mysteries of Stained Glass." Templeton Religion Trust. https://templetonreligiontrust.org/explore/art-light-and-awe-uncovering-the-mysteries-of-stained-glass/ (accessed 2024).

[53] *Adobe Photoshop*. Adobe Inc. [Online]. Available: https://www.adobe.com/ie/products/photoshop.html

[54] *Paint.NET*. dotPDN LLC. [Online]. Available: https://www.getpaint.net/

[55] "Unity Universal Render Pipeline Package Manual." Unity Technologies Inc. https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@17.0/manual/ (accessed 2024).

[56] E. Reinhard, "Sampling, Reconstruction, Aliasing, and Anti-Aliasing," in *Encyclopedia of Color Science and Technology*, R. Shamey Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, pp. 1-9.

[57] "Unity Scripting Reference." Unity Technologies Inc. https://docs.unity3d.com/ScriptReference/index.html (accessed 2024).

[58] "API Reference for Google Cardboard XR Plugin for Unity." Google. https://developers.google.com/cardboard/reference/unity (accessed 2024).

[59] "Quickstart for Google Cardboard for Unity." Google LLC. https://developers.google.com/cardboard/develop/unity/quickstart (accessed 2024).

[60] "Unity Mock HMD XR Package Manual." Unity Technologies Inc. https://docs.unity3d.com/Packages/com.unity.xr.mock-hmd@1.0/manual/index.html (accessed 2024).

[61] S. Butler. "How Important Are Refresh Rates In VR?" How-To Geek. https://www.howtogeek.com/758894/how-important-are-refresh-rates-in-vr/ (accessed 2024).

[62] Jovan. "How Much RAM Does A Smartphone Need Today?" KommandoTech. https://kommandotech.com/guides/how-much-ram-does-a-smartphone-need/ (accessed 2024).