BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PHYSICAL SIMULATION IN VR

FYZIKÁLNÍ SIMULACE VE VR

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **GABRIELA PACÁKOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. JOZEF KOBRTEK**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání bakalářské práce/21242/2017/xpacak01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Pacáková Gabriela**
Obor: Informační technologie
Téma: **Fyzikální simulace ve virtuální realitě**
Physical Simulation in VR
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s rozhraním OpenGL pro tvorbu 3D grafických aplikací a knihovnou OpenVR pro práci s brýlemi virtuální reality.
2. Vyberte vhodnou knihovnu pro modelování zvoleného fyzikálního problému.
3. Navrhněte aplikaci pro vizualizaci zvolené simulace s využitím virtuální reality.
4. Aplikaci naimplementujte
5. Sezbírejte zpětnou vazbu od uživatelů.
6. Vyhodnoťte dosažené výsledky i zpětnou vazbu.

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kobrtek Jozef, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
L.S. 612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

The main aim of this thesis is to introduce the reader to the theory and the implementation of a virtual reality application which uses HTC Vive headset and BulletPhysics engine. It describes the integration of BulletPhysics and OpenVR SDK with OpenGL. The result of this integration is a simple bowling game in VR that uses all the aforementioned resources and provides the user with an entertaining VR experience as well as a deeper understanding of computer graphics and VR principles.

Abstrakt

Hlavním cílem této práce je seznámit čtenáře s teorií a implementací aplikace pro virtuální realitu, která využívá headset HTC Vive a knihovnu BulletPhysics. Popisuje integraci nástrojů BulletPhysics a OpenVR SDK s OpenGL. Výsledkem této integrace je jednoduchá bowlingová hra ve VR, která využívá všechny výše uvedené zdroje a poskytuje uživateli zábavný zážitek z VR, stejně jako hlubší pochopení počítačové grafiky a principů VR.

Keywords

Virtual reality, Physical simulation, OpenGL, Head-mounted display, HTC Vive, OpenVR SDK, BulletPhysics, C++, SDL2, Assimp

Klíčová slova

Virtuální realita, Fyzikální simulace, OpenGL, Headset pro virtuální realitu, HTC Vive, OpenVR SDK, BulletPhysics, C++, SDL2, Assimp

Reference

PACÁKOVÁ, Gabriela. *Physical Simulation in VR.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jozef Kobrtek

Rozšířený abstrakt

Tato práce se zabývá návrhem a implementací jednoduché grafické aplikace, která vizualizuje fyzikální simulaci knihovny BulletPhysics ve virtuální realitě za použití OpenGL jako grafického API. Díky integraci systému virtuální reality HTC Vive dovoluje aplikace uživateli interaktivním a zábavným způsobem ovlivňovat chod simulace. Aplikace modeluje simulaci jako jednoduchou bowlingovou hru, ve které může uživatel vidět a ovlivnit chod simulace při hodu bowlingové koule do kuželek na dráze. Aplikace je implementována v jazyce C ++ na operačním systému Windows, protože z hlediska virtuální reality je pro tento systém poskytnuta zatím nejstabilnější podpora, co se týká knihovny OpenVR a grafických ovladačů.

Aplikace je primárně určena pro používání ve virtuální realitě. Její návrh řeší nejen problémy správné vizualizace fyzikálního světa, modelů a světla, ale i problémy, které vznikají při nesprávném návrhu vizualizace pro virtuální realitu. Práce objasňuje nejčastější problémy spojené s používáním virtuální reality, jako nevolnost, bolest očí a hlavy, a aplikuje popsané postupy správného návrhu vizualizace v implementaci pro eliminování výše zmíněných nežádoucích efektů.

Aplikace používá svůj vlastní jednoduchý game engine. Základní návrh aplikace zahrnuje propojení tří důležitých součástí, a to knihovny BulletPhysics, knihovny OpenVR pro možnost integrace systému HTC Vive a OpenGL, díky kterému je zajištěno vykreslování výsledné scény. Scéna je navržena tak, aby obsahovala základní komponenty pro možnost interakce uživatele, jako kuželky, bowlingové koule a dráhu. Pro zpříjemnění zážitku uživatele z vizualizace jsou v aplikací použity populární efekty počítačové grafiky, jako mapování stínů (angl. Shadow mapping) nebo multisampling pro zaoblení zubatých hran. Práce popisuje principy fungování jednotlivých efektů a popisuje jejich vhodnou implementaci.

Základní okno pro vykreslování je vytvořeno za použití knihovny SDL2. Modely bowlingového sálu, kuželek a koulí vytvořené v programu na modelování Blender jsou za pomoci knihovny Assimp importovány do aplikace, kde jsou následně jejich data vhodně zpracovány a predány OpenGL na vykreslení. Výše zmíněné efekty, jako mapování stínů, jsou vytvořeny za pomoci OpenGL zřetězeného zpracování při vykreslování, které je programovatelné pomocí shaderů (angl. Programmable pipeline).

Velkou část implementace tvoří integrování systému HTC Vive. Tento systém funguje pomocí technologie Lighthouse pro sledování pohybu v prostoru. Tato technologie využívá stanice pro sledování pohybu, které pomocí infra-červeného laseru zaměřují headset a ruční ovladače, a vypočítávají pozici zaměřených bodů v prostoru. Tyto pozice jsou následné vyžádany od systému a zpracovány v aplikaci tak, aby mohly být predány OpenGL pro vykreslení virtuální scény. Tímto způsobem je přenesen reálný pohyb hlavy a rukou uživatele do vykreslené scény v aplikaci. Virtuální realita je známá tím, že využívá binokulární vidění lidských očí, aby vhodným způsobem vizualizovala hloubku vykreslené scény pro uživatele. Tento fakt je následně použit při vizualizaci v OpenGL, kde je obraz vykreslen dvakrát každou sekundu s rozdílným úhlem pohledu pro každé oko.

Interakce s objekty ve virtuální realitě je uskutečněna pomocí použití ovladačů systému HTC Vive do rukou a propojením jejich vlastností s knihovnou BulletPhysics. Na zaměřování objektů s ovladačem ve scéně je využita metoda ray test. Tato metoda spočívá v projekci neviditelného paprsku z grafické reprezentace ovladače směrem do scény. Paprsek má omezenou délku. Při protnutí nějakého objektu tímto paprskem se vrátí výsledek testu s informací o objektu a dále se dá tímto objektem manipulovat za pomoci BulletPhysics constraints, které nastavitelným způsobem omezují a kontrolují pohyb simulovaného objektu.

Pro shrnutí dosažených výsledků bylo provedeno testování na dvanácti uživatelích, které odhalilo nedostatky a zároveň potvrdilo správně použité principy při vývoji pro virtuální realitu. Možnost interakce se simulací uživatelům umožnila mnohem lepší vcítění se do virtuálního světa. Zároveň většina uživatelů odhalila, že během testování aplikace se u nich neprojevily žádné nežádoucí efekty při používání virtuální reality. Náročnost uživatelů se projevila na faktu, že poukazovali na absenci základních herních mechanik pro bowling, jako hodnocení hodu a počítání bodů, což však nebylo hlavním cílem této práce. Grafická vizualizace byla vzhledem k použité technologii hodnocena pozitivně. Základním nedostatkem bylo však podle poloviny uživatelů náročné ovládání interakce s koulí. Druhá polovina uživatelů tvrdila, že náročnost byla přiměřená.

Aplikace v konečném důsledku splňuje základní funkce stanovené zadáním a rozšiřuje vizualizaci o již zmíněné grafické efekty. Implementace však nabízí velký prostor pro zlepšení a budoucí vývoj, například zahrnutí uživatelského rozhraní s využitím principů návrhu pro virtuální realitu, implementaci herní mechaniky bowlingu, nebo možnost několika levelů a modifikace vzhledu některých prvků. Tímto způsobem by se z aplikace mohla stát plnohodnotná hra ve virtuální realitě s vlastním enginem pro virtuální realitu, který by také mohl být jednoduše rozšiřován díky objektově orientovanému návrhu implementace.

Physical Simulation in VR

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Jozef Kobrtek. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Gabriela Pacáková
May 14, 2018

Acknowledgements

I would like to thank my supervisor Ing. Jozef Kobrtek for his help and guidance.

Contents

1 Introduction **2**

2 Theory **4**
2.1 Virtual Reality 4
2.2 The History Of Virtual Reality 4
2.3 Virtual Reality In The 21st Century 6
2.4 Types Of Virtual Reality Applications 7
2.5 Best Practices In VR 9
2.6 HTC Vive 11

3 Application Concept **13**
3.1 OpenGL 13
3.2 BulletPhysics 13
3.3 OpenVR 14
3.4 Other Libraries 15
3.5 Scene Concept 17
3.6 Post-processing Effects 17

4 Implementation **22**
4.1 Application Core 22
4.2 Models 23
4.3 Physics 24
4.4 Light 27
4.5 Post-processing 28
4.6 HTC Vive Integration 30

5 Testing **37**
5.1 Testing Questionnaire 37
5.2 The Results Of Testing 38

6 Conclusion **40**

Bibliography **41**

A DVD Content **42**

Chapter 1

Introduction

The popularity of the term *virtual reality* has risen by a solid amount among computer scientists and technology fans in the recent years. The main cause of this virtual reality trend is the new technologies brought to the market by companies such as Oculus and HTC. These companies developed the first next generation head mounted displays for virtual reality, namely Oculus Rift and HTC Vive, and set the industrial standard among virtual reality hardware. The virtual reality segment of this thesis particularly focuses on the HTC Vive headset.

Having its beginnings at around 1960s, virtual reality has been used in military and astronaut training, robotics, and some forms of entertainment. However back then the developers were widely disadvantaged when it came to computing power and the general lack of appropriate technology, which was not yet developed. There was also a lack of software support. All this lead to a reason why virtual reality did not hit the general public the same way as it did a couple of years back.

As the technology progressed a great deal over the years, developers nowadays do not need to concern themselves with the current state of technology development. Head mounted displays offer low latency and reasonably high resolution which improves every year, stereo sound and some provide motion tracking both for the user's head and for the hands. The bigger concern today is the motion sickness from the experience, although developers and headset manufacturers work very hard to make this issue go away. Another concern might be the still unknown health risks virtual reality carries when used in long-term periods. Due to a still quite high price range, it is possible that it will take a few years to find out.

The aim of this thesis is to guide the reader along the concept and the implementation of an application that supports not just a scene one can look at through the VR lens, but also interact with. It will describe what exactly is necessary to integrate to make an application capable of simulating rigid body movement according to physical laws from the real world, as well as handling user input through motion-tracked HTC Vive controllers and headset. All this will be integrated using C++ as the programming language and OpenGL as the graphical API.

Virtual reality history, concepts and best practices will be discussed in chapter 2. This chapter will be followed by the application concept in chapter 3. The application concept will describe the libraries that are needed for the game to work properly, as well as the concept of the scene that the user will be interacting with (Section 3.5). To make the application more pleasing to the eye of the user, some of the most popular post-processing effects in today's games used in this application will be discussed (Section 3.6).

The implementation description will follow up in chapter 4. The reader will be informed how all the components of the scene, like models and lights, are brought together (Section 4.1) and how post processing is applied to the scene (Section 4.5). Nearing the end of the chapter, it will be described how HTC Vive headset is integrated (section 4.6) and what difference it holds to render a virtual-reality scene in comparison to rendering to a 2D screen. Chapter 5 will describe the testing of the application. Chapter 6 will summarise what was achieved during the realisation of this thesis and what could be done in a potential future development to make the implemented application an interactive game with all the features.

Chapter 2

Theory

This chapter will describe what virtual reality is and it will introduce the reader to the principles of virtual reality design. It will also cover the HTC Vive headset in more detail and describe how it works. In addition to virtual reality, this chapter will cover the basic physic principles of rigid body dynamics and simulation.

2.1 Virtual Reality

In technical terms, virtual reality is the term used to describe a three - dimensional, computer generated environment which can be explored and interacted with by a person [2]. The person inside is immersed into the world on a higher level, their movement and body becoming a full part of the game world [4][6]. This is accomplished through hardware, specifically the head mounted display (abbr. HMD), and the appropriate software. The HMD and the software are set up just the way for them to form the correct amount of sensory stimulation, so the human brain achieves a sense of presence in the virtual world.

2.2 The History Of Virtual Reality

Figure 2.1: Panoramic view of London. Aquatint by Henry Aston Barker, after Robert Barker, 1792

The first indications of virtual reality principles come from late 18th and early 19th century, when panoramic paintings of historical events became very popular (Figure 2.1)[1]. These paintings allowed viewers to see a scene captured by an artist in a full three hundred and sixty degree field of view.

In 1838 the first stereoscope was invented by Sir Charles Wheatstone. It united the two representations of objects in space that are seen by human eyes with a slightly different

[1] https://commons.wikimedia.org/wiki/File:Panorama_of_London_Barker.jpg

(a) A stereoscopic photograph from 1901.

(b) A screenshot of a virtual reality headset projection.

Figure 2.2: A comparison of stereoscopic images.

viewpoint from each eye [3]. This invention essentially lets the user view two photographs with the same content taken with a slightly different angle and allows for them to be presented to each eye with a natural depth perception that human eyes possess. That way a person using the stereoscope felt like they were looking at a real scenery rather than a photograph. Principally, it is the same thing that is used nowadays in virtual reality headsets (Figure 2.2).

The invention of computers and new technology allowed to make attempts at bringing virtual reality closer to where it is now. The first virtual reality system is considered to be *The Sensorama* by Morton Heilig in 1957 (Figure 2.3a). The Sensorama featured not just the stereo sound and the stereoscopic display, but also a vibration chair, a smell generator and fans. This was followed by another of his inventions in 1960 called *The Telesphere Mask*, which was the first true head-mounted display (Figure 2.3b). It provided stereo sound and stereoscopic display, but was not capable of motion tracking. The biggest breakthrough yet came with the invention of *The Sword of Damocles* in 1968 [6]. This virtual reality system was the first one to provide an input from a computer rather than a camera. The graphics consisted of simple wireframe objects and the headset provided motion tracking in addition to stereoscopic display. It was very heavy and thus it needed to be suspended from the ceiling using a mechanical arm (Figure 2.3c).

For the next two decades virtual reality provided mainly simulations for flight and military training and in medical industry. In the 1990s it was brought into the entertainment industry. It was now possible to publicly use virtual reality machines in arcades, even though the technology was still unavailable for private ownership in households. These virtual reality systems provided an immerse experience with displays that had a latency

(a) The Sensorama.

(b) The Telesphere Mask.

(c) The Sword of Damocles.

Figure 2.3

of less than 50 milliseconds and some of them were also connected together for multi - player gaming. At that time, two headsets made an appearance. One was the *Sega VR*, which was announced in 1993 but never released due to testers developing headaches and motion sickness. The other one was *The Nintendo Virtual Boy* released in 1995. Due to its negative reception from public, it was discontinued a year after. Since then, virtual reality has not seen a proper comeback until the second decade of the 21st century, mainly because the systems were still lacking the lightweight technology that would keep the pace with the creative vision.

2.3 Virtual Reality In The 21st Century

With the sudden development of lightweight mobile systems and smartphones with high density displays that are capable of projecting 3D graphics to their screens, virtual reality hardware became a lot easier to develop too. The lightness of these systems allowed to overcome many obstacles such as the weight of the head-mounted displays. In 2010 the first prototype of the next generation virtual reality headset, the Oculus Rift (Figure 2.4b) was announced by the company named Oculus VR. It featured a rotational head tracking

(a) Google Cardboard.

(b) Oculus Rift.

(c) HTC Vive.

Figure 2.4

and ninety degree field of view, which was never seen before. With the price range of the HMD's that was, and still is, quite high, to make the virtual reality more public, companies like Google released lenses in headsets made from cardboard[2] (Figure 2.4a) with the possibility to plug in a smartphone with the appropriate software to experience the depth of the stereoscopic computer generated scenes. In 2015 Valve Corporation and HTC released the HTC Vive headset (Figure 2.4c) running on the Steam VR platform. It was the first headset that featured positional motion tracking for the head and the hands of the user in a defined area in addition to 1K displays for both eyes and fresnel lenses. As of now virtual reality is becoming a large part of the industry with companies dedicating more and more teams towards VR software development. The hardware is being actively improved with HTC and PlaystationVR announcing wireless headsets in the near future.

2.4 Types Of Virtual Reality Applications

The market for virtual reality software is growing and it can already be observed that there are various types of applications from which some provide the option for user input and some do not.

Non-interactive, or *passive* VR applications include 360 degree movies or computer generated scenes, very often used for educational purposes [4]. When using a non interactive application, the user does not have control over the scene and is unable to provide any input or influence the application flow. It can prove to be a good source of entertainment nevertheless, but the impossibility of interaction breaks the sense of presence of the user in the virtual reality world.

On the other hand, *interactive* applications provide enough distractions for the user's sense of presence and, as a study suggests, even the pain intensity is lowered while interacting with the VR world [7]. It is therefore entirely possible that virtual reality will have a positive effect during therapies and it might be used more frequently in medicine. Virtual reality systems these days provide many possibilities on how to interact with a scene.

2.4.1 Interaction In Virtual Reality

With the first new generation HMDs interaction in a VR application was established still using mouse and keyboard or a non-tracked hand held controller. Another interaction could be achieved by firing an invisible horizontal ray from the user's viewpoint, letting the user choose an item by looking at it. This is also known as raycasting. However the true breakthrough and more complete immersion came only with the hand held controllers tracked in 3D space in addition to the headset. HTC Vive and later Oculus developed their own controller that the user would hold in hand. The controller would track the real hand's position and provide tracking data to project the tracked hand into the VR scene, allowing to interact with the objects and the scene itself. Companies like Cyberglove Systems[3] developed a hand worn tracking device that can track each finger separately, and a fully featured API.

There are two main views that can be used or combined when designing user interaction with hand tracking, namely the *egocentric* and the *exocentric* view [6].

[2] https://vr.google.com/cardboard/
[3] http://www.cyberglovesystems.com/

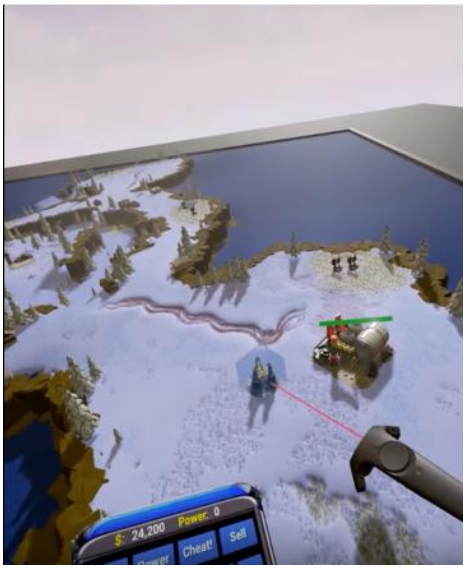


Figure 2.5: An example of exocentric interaction in an egocentric view.

Egocentric view is very closely related to *proprioception*[4]. This view is a first person view of the world, which provides egocentric interactions within this particular environment, for example standing inside a room in virtual reality filled with objects and interacting with the objects.

Exocentric view is a view where a person manipulates a model of an environment outside of it. For example being situated above the VR world and interacting with objects inside it like in strategy games.

In VR these two views are often combined and the user can apply exocentric interactions while in egocentric view (Figure 2.5) and vice versa.

2.4.2 Health Concerns

Many negative health effects were reported for some users while using VR including nausea, headaches, eye strain or physical injury. The causes of these side-effects can be an incorrect calibration, high latency or incorrectly placed objects in space that is motion captured.

The Importance Of Low Latency

Latency is the time the system takes to respond to the user's request. In VR terms, latency is the time it takes for the motion tracking system to process user tracking data to the time the display in the headset responds with the correct projection of user's pose [6]. It is very important to make this time as short as possible, so that the user does not notice the scene lagging behind. Same applies when tracked controllers are used. If they lag behind in the virtual world, it causes conflict with user's senses and the experience can quickly become nauseous and headaches can occur.

Incorrect latency is the biggest cause of VR motion sickness and when developing VR systems it needs to be profoundly understood. Many headset manufacturers therefore recommend a minimum of 90 frames per second when rendering a VR scene, as it lowers input

[4]The physical sense of the body, its pose and motion.

lag greatly. However there can be times when the fps drops bellow 90, and the manufacturers thought of that. Oculus possesses a technology called *Asynchronous SpaceWarp*[5] (*Asynchronous Reprojection* in HTC headsets), which interpolates additional frames when the application performance drops.

2.5 Best Practices In VR

When designing a VR application there exist some recommended concepts that one should follow to make the best possible VR experience. It is important to realise that many of these concepts exist for a reason and that usually is to get rid of motion sickness inducing effects that the user would not mind when projected to the 2D flat screen.

2.5.1 Camera Views

As stereoscopy is used in virtual reality headsets to apply realistic depth to an image it is important to realise that the cameras rendering an image must posses the same field of view as the user's eyes and the viewpoints for both cameras have to be located at appropriate distance from each other. In other words, the inter-camera distance needs to be the same as the interpupillary distance. Fortunately, the SDKs for the headsets usually take care of the projection and view matrices and they can be requested from the device.

It is very common in non-VR applications, especially ones with a first-person view to occasionally take control of the user's view and point it in some direction grabbing the user's attention and pointing it towards some commotion in the scene. In VR however this principle will not have a very pleasant effect. On the contrary, it can induce motion sickness quickly, because controlling the view of the user for them in VR causes sensory input to become asynchronous. The user's head would be still, not turning around in any direction but the projection in the headset would be suggesting otherwise. Same principle applies to cinematic cameras. It is also important to avoid effects such as depth of field or motion blur, as they can heavily affect what the user can actually see while wearing a headset and blurring image intensely while moving the head around can also be nauseous[1].

2.5.2 User Interface

Graphical user interface, also known as GUI or a HUD (from *Heads Up Display*) is very common in 3D applications and games. It consists of information that is important for the user, such as a minimap, remaining health or an aiming crosshair. When projected on a flat screen, the HUD is implemented usually as a 2D overlay over the 3D scene. However when incorporating user interface into virtual reality, this concept becomes unusable. As the image is rendered twice from different viewpoints, it would be difficult to unite the 2D overlay in such a way that it would not appear distorted due to binocular disparity. In addition it is inefficient to place such elements in user's peripheral vision. That would force the user to constantly switch focus on the edges of the image which would cause eye strain and possible headaches.

The solution to this is to place the user interface and visual clues inside the VR environment [4][6][8], as it is depicted in figure 2.6. In the figure it is visible that the previously 2D user interface projected on top of the 3D scene is now a part of the scene. If the user turns their head around, the interface stays in place as any other 3D object in the scene

[5] https://developer.oculus.com/blog/asynchronous-spacewarp/

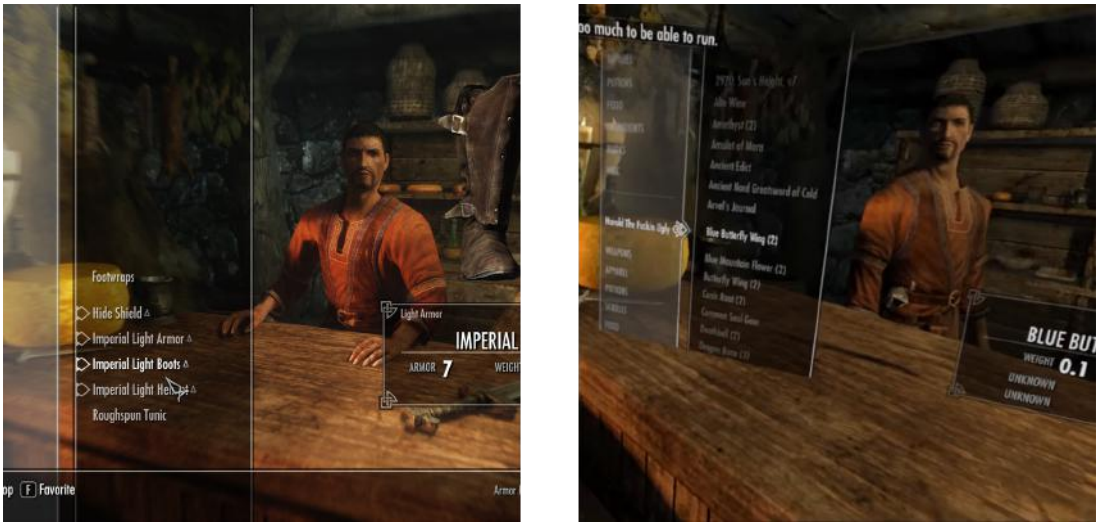


Figure 2.6: A comparison of non-VR (left) and VR (right) interface in The Elder Scrolls V: Skyrim.

and does not follow user's view with zero latency. This helps to keep user's sense of depth and immersion and prevents all sorts of aforementioned problems.

2.5.3 VR Environment

Drawing the environment in VR also consists of some recommended practices. The VR world should be scaled 1:1 to the real world, especially with games motion tracking the player while standing up. All the movement while the headset is used is supposed to be as close to the real movement as possible. For example if a user turns their head 90 degrees, the world should also rotate by 90 degrees. In traditional 3D applications, going up or down the stairs is not a problem at all. However it is impossible to just walk up the stairs or heightened terrain in VR and teleporting the player up the terrain could be really sickness inducing. This should therefore be avoided and replaced by using lifts or other means if possible.

Drawing a user avatar can also increase immersion [4]. An avatar is the representation of the user's body in VR world. It makes a big difference when the user looks down and sees their own legs or when the motion tracked hand controllers help to estimate the position of user's shoulders and elbows.

2.5.4 Anti-aliasing

Anti-aliasing, discussed in detail in sub-section 3.6.4 is an important effect that should always be employed in virtual reality scenes. Anti-aliasing artefacts also called *staircases* or *jagged edges* are even more visible in virtual reality due to constantly moving viewpoint [6]. It will never be possible to fully eliminate them, but sampling techniques help to reduce these artefacts. The downside to this is that some techniques require a lot of performance and thus the latency might be increased when the framerate drops.

2.6 HTC Vive

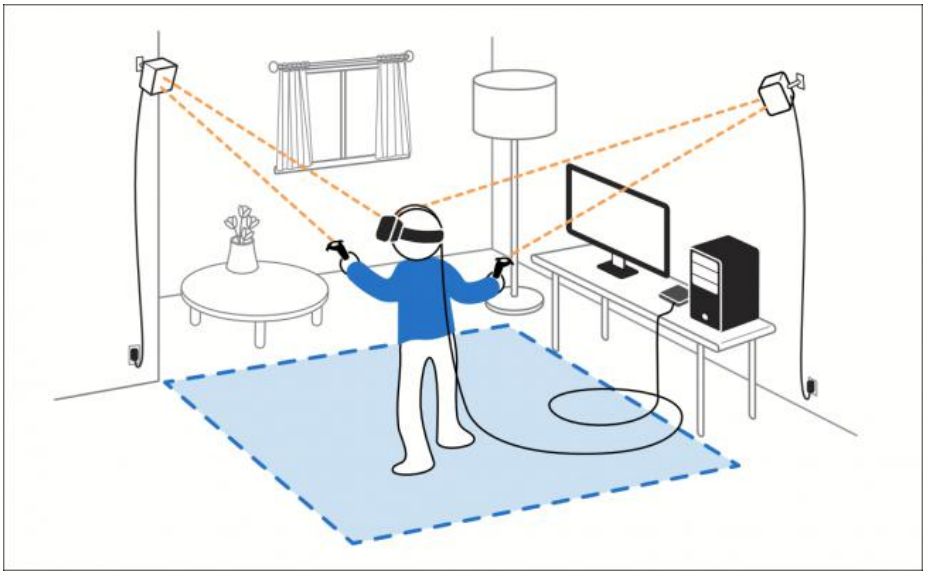


Figure 2.7: A drawing depicting Lighthouse tracking technology.

HTC Vive[6] is a virtual reality headset developed by HTC and Valve that will be used to access the virtual reality element of this project. It is powered by SteamVR, a Valve platform for VR content. It uses a room motion tracking technology called Lighthouse to communicate with the HMD and its controllers in 3D space by emitting infra-red laser pulses (Figure 2.7). Vive is able to gather motion tracking data in an area of 4 x 4 meters which can be expanded adding more base stations. The whole Vive system consists of the following components and technologies:

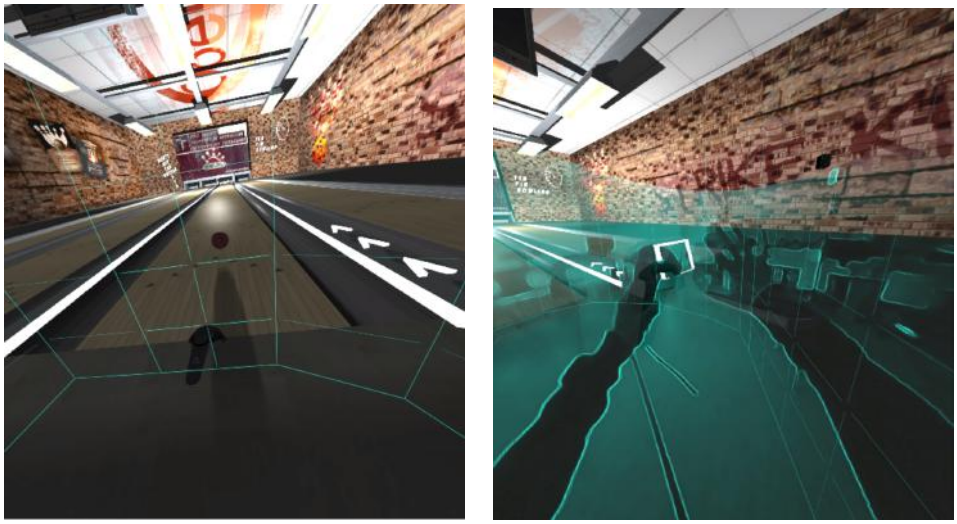
- **Vive headset**: It communicates with the base stations to transfer tracking data of user's head position and rotation. In addition, the headset features a front faced camera that can capture the scene's depth and warn the user of any obstacles by projecting the camera output to the headset's screen with the help of *Vive Chaperone* technology which will be mentioned later.

- **Vive Controllers**: They enable the user to interact with the VR scene, including the GUI and objects in it. The controllers provide tracking data about their position, which is basically the position of user's hands. This adds a great deal of immersion as the user needs to move around the room to reach objects or adjust the view.

- **Vive Base Stations**: They send requests and receive responses with tracking data to tracked devices allowing the application to process the data accordingly. The base station consists of two laser emitters that spin at a speed of sixty times per second, and a laser beacon. The laser beacon emits a synchronisation pulse and one of the two emitters sweep a laser beam across the room. The tracked devices posses receptors with photo sensors that can recognise the sync pulse and the beam. When the receptor catches the sync pulse, it counts until a photo sensor is hit by the laser beam. Lighthouse calculates *where* and *when* that photo sensor was hit to

[6] https://www.vive.com/

find the exact position of that receptor in relation to the base station. When there are multiple receptors a 3D pose in space is formed relative to the base station. Adding more base stations improves tracking range.



(a) (b)

- **Vive Chaperone System**: This is a system that draws boundaries around the user's tracked space inside the running application (Figure 2.8a). This way the user is aware if they approach the end of the tracked area without crossing a boundary. As mentioned earlier, the headset consists of a front facing camera that can be used for different purposes. It can be turned on while still wearing headset, and the user can view the real surroundings in addition to the VR scene, but it can also be configured so that when the user exits the tracking area it will project the depth of the real surrounding to the VR application (Figure 2.8b).

Chapter 3

Application Concept

This chapter will inform the reader about the details of the libraries that are used in the final solution. In addition, it will describe how the scene will look like and what exactly will be needed to render it. It will also offer a closer look to how the post - processing works and details towards some of the most popular effects.

3.1 OpenGL

OpenGL[1] *(Open Graphics Library)* is a cross platform and language independent graphical API and an industrial standard used in many graphical applications. It provides access to the graphics processing unit (GPU) and allows the programmer to render scenes with hardware acceleration. This API was first released in 1992 by SGI and has been maintained by Khronos Group since 2006. For it to remain cross platform, it has been made a purely rendering API not responsible for handling sound or application windows events. Other libraries must be incorporated to the application to do so. From version 2.0 onward, OpenGL consists of a programmable rendering pipeline which uses shaders that allow to customise many rendering effects like lights, shadows or post-processing. This way colour, textures, position and other properties of a pixel in the final rendering can be altered by the programmer with ease. Due to the fact that this application integrates virtual reality, only OpenGL 4.5 onward is supported. As an addition, *GLEW (OpenGL Extension Wrangler Library)* is used in this application. It determines, which OpenGL extensions are supported and loads OpenGL function pointers from the graphic driver on the target platform.

3.2 BulletPhysics

BulletPhysics[2] is an open source physics engine written in C++ and available for personal and commercial use under the zLib[3] license. It is used in many popular 3D applications such as Blender, in games and in movies for visual effects simulation. Bullet provides the simulation of the dynamics for the rigid bodies and collision detection in the application. The core of the library is divided to sub-modules, like collision detection, rigid body dynamics, soft body dynamics etc. This allows the programmer to integrate only the mod-

[1] https://www.khronos.org/opengl/
[2] https://github.com/bulletphysics/bullet3
[3] https://www.zlib.net/zlib_license.html

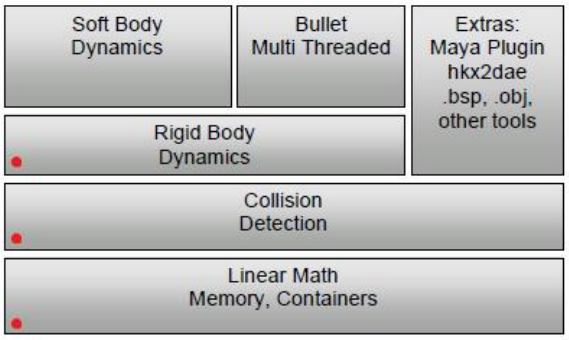


Figure 3.1: BulletPhysics modules. The ones used in this application are marked with a red dot.

ules they need in the current application (Figure 3.1). The engine is written in C++ and contains a module implementing mathematical structures like matrices and vectors that deal with position and rotation of the rigid bodies in the world space. This way Bullet is very easy to integrate. The engine itself does not have a renderer, but it can be integrated with any rendering API. Each rigid body simulates collision and acts on the physical forces. The rigid bodies have a world transform that can be obtained by the programmer and connected to the rendered shape on the side of the graphics API. The transform is a matrix providing information about the position, rotation and scaling of the rigid body. It can be easily converted to a matrix supported by OpenGL. Another great advantage of Bullet is that the default coordinate system it uses is the same as in OpenGL, a right handed system with X to the right, Y up and -Z forward. This way complicated transformations do not need to be applied to synchronise the positions of the objects in the invisible physical world with the ones in the visible rendered world.

3.3 OpenVR

OpenVR[4] API is a library, which supports access to the VR hardware without the need to rely on specific vendor drivers. This library will be used in the application to access the tracking information for the HTC Vive headset and controllers and to render the final scene to the headset's screen. The only requirement is that SteamVR[5] needs to be installed to use OpenVR code, otherwise the head mounted display will not initialise in the application. OpenVR also provides „hello vr world" examples for rendering APIs including OpenGL and Direct3D which makes it easy to integrate into an existing project. OpenVR is written in C++ and developed by Valve Corporation.

This API can be broken into six interfaces, each providing access to different parts of the headset functionality:

- **IVRSystem**: This is the main interface of OpenVR. It provides access to tracking data, distortion functions, system events and controller states. The pointer to this system can be retrieved using the `VR_Init()` function.
- **IVRChaperone**: This interface provides access to Vive Chaperone system.

[4] https://github.com/ValveSoftware/openvr
[5] https://steamcommunity.com/steamvr

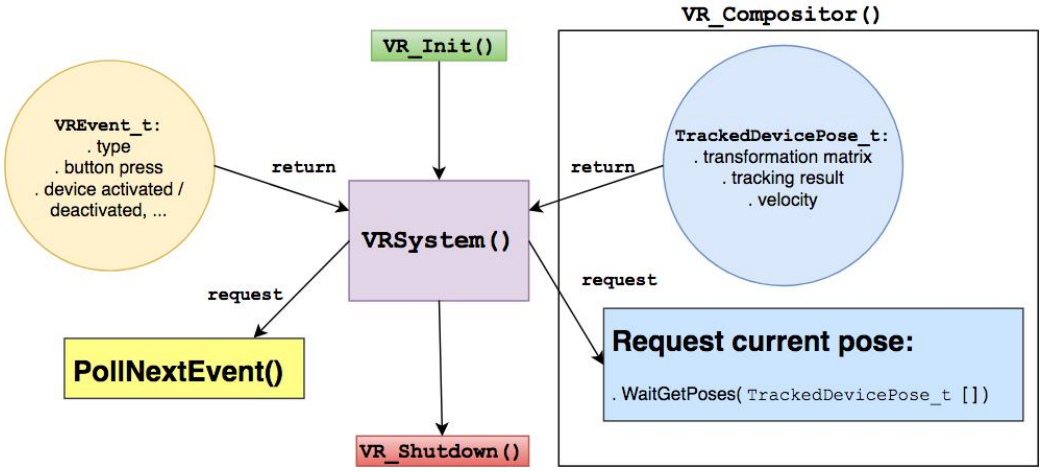


Figure 3.2: Essential VR calls to access the HMD system data.

- **IVRCompositor**: Provides access to the VR compositor sub-system. The compositor takes care of projecting the final distorted image to the headset's screen. It also calculates poses needed to render the images for both eyes.
- **IVROverlay**: Manages the 2D overlay rendered over the 3D scene. Overlays usually serve as a user interface for VR applications. The most common overlay is the VR dashboard.
- **IVRRenderModels**: Provides access to 3D models for tracked devices such as the HTC Vive controllers. The model of a controller can then reflect the position of the physical controller inside the VR world.
- **IVRScreenshots**: Provides a way to take and share screenshots across the dashboard or Steam platform.

It is very easy to request projection, view or pose matrices for the tracked hardware and supply it to the rendering part of the application. The VR compositor, which the rendered scene needs to be supplied to takes care of the appropriate distortion and the headset can even recommend at what resolution to render the scene for both eyes. On the other hand everything from system event polling, tracked device detection and initialisation of VR scene components to cleanup and shutdown is up to the programmer. Figure 3.2 displays the connection between components of OpenVR API that are needed for basic event processing and accessing motion tracking data.

3.4 Other Libraries

SDL2[6] *(Simple DirectMedia Layer 2)* is a cross platform library that provides OpenGL or Direct3D applications low level access to keyboard, mouse, audio and a lot more. In this application, the library handles window creation, OpenGL context[7] creation and registration of keyboard and mouse events when using the application. SDL is written in C and therefore it provides native integration into C++. OpenGL itself is not capable of loading

[6] https://www.libsdl.org/
[7] https://www.khronos.org/opengl/wiki/OpenGL_Context

images that serve as textures, but SDL can handle image loading as well. Plain SDL can load *.bmp* images only, but there is an extension named *SDL2_image*. This extension allows to load other image formats such as *.png, .jpg, etc.* This application uses the extension to load *.png* textures and supply the loaded data to OpenGL.

Assimp (Open Asset Import Library) is a portable library written in C++ that can load 3D model files into one bigger structure, the `aiScene`, which contains smaller structures for model meshes. The `aiScene` consists of a node tree starting with a root node. It can be recursively processed allowing an easy access to vertices, indices, texture coordinates and other properties that need to be supplied to OpenGL pipeline to render objects. Assimp is well documented and thus not difficult to integrate into an application. It supports many 3D model formats including *.obj + .mtl, .3ds* and *.dae*. Detailed integration is discussed in chapter 4, section 4.2.

GLM (OpenGL Mathematics) is a cross platform header library written in C++ that handles all sorts of mathematical operations when it comes to 3D transformations in OpenGL. It is based on the GLSL language[8] specifications, therefore fully compatible with the OpenGL rendering pipeline. GLM offers functions dealing with matrix rotations and translations, scaling, vector and matrix multiplications as well as quaternion operations.

[8] https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

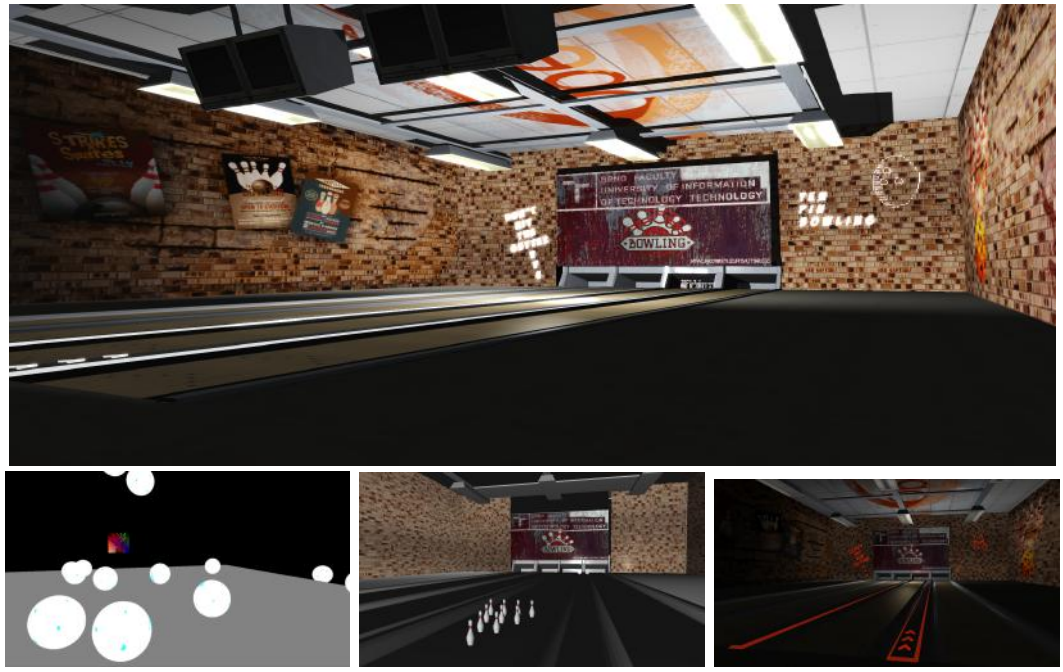


Figure 3.3: Four iterations of the scene. The top one is how the final application looks like.

3.5 Scene Concept

The application demonstrates the physical simulation interactively during a simple bowling game. This means that it needs to be designed in a manner so that the user can interact with a bowling ball using a HTC Vive controller and then throw it towards the bowling pins. This action demonstrates the rigid body collision and movement according to physical forces applied to the ball and to the pins if they are hit.

The scene therefore needs to contain a bowling alley with a bowling ball stand filled with balls, and ten bowling pins ready to be knocked over. That is just for the model part. Lights are very essential, as the user needs to see the simulation properly. It is also important to make the scene look pleasing to the user's eyes, so different textures need to be created and applied to the models, the models need to be adjusted to contain interesting elements and some post processing effects such as shadows and anti-aliasing are essential during the final rendering. The final scene went inevitably through a few iterations depending on which components of the application core were currently implemented. Figure 3.3 shows the progress of the scene concept.

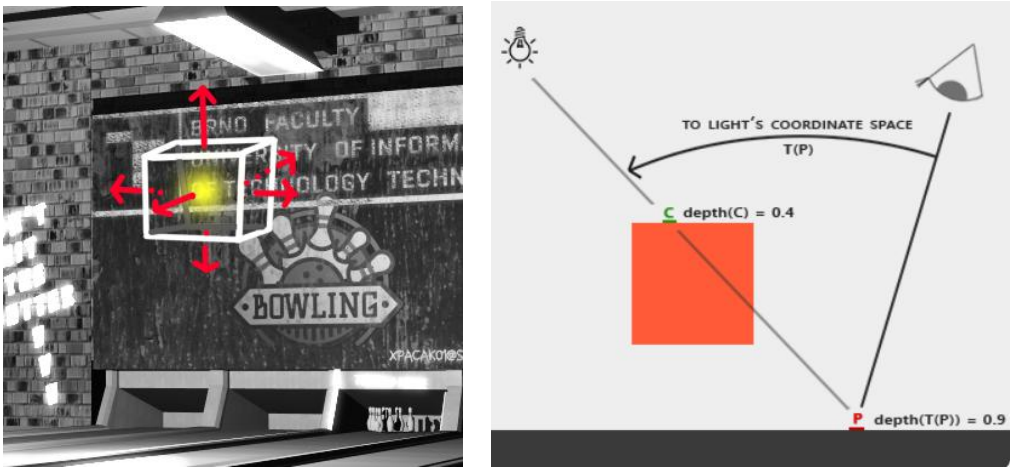
3.6 Post-processing Effects

Post-processing is very commonly used in 3D rendering and graphics. Every new game on the market supports a variety of post-processing effects. These effects can add additional depth and smoothness to the image, more realistic lighting or shadow projection. Some effects, not only those used in this application, require multiple render passes. This means that the image is first rendered into an off-screen framebuffer object, an FBO[9], post-processing shaders are applied and then it is rendered either to the screen or to an-

[9] https://www.khronos.org/opengl/wiki/Framebuffer_Object

other FBO. It is common for the post-processing to be implemented in OpenGL fragment shader (for Direct3D API it is pixel shader), but some effects can also be implemented using vertex shader and geometry shader. Following sections will discuss the principles of post-processing effects used in this solution.

3.6.1 Shadow Mapping



(a) A sketch of the point light in the application with the depth cube around it.

(b) The principles of shadow test.

Figure 3.4

Shadows are created when there is an absence of light due to the light source being occluded by some object. They are an important part of any lit scene as they add a great sense of realism. If the application is supposed to work in VR, where scene realism holds a great value, it is recommended to implement shadows. There are a few rendering techniques that can produce shadows, but this subsection is dedicated especially to the shadow mapping technique.

The idea behind shadow mapping is to render the scene's depth from the light's point of view [10][11]. The depth map, which holds values of the distance of a surface from a certain viewpoint, in this case the light's viewpoint, will be stored in an off screen FBO. It needs to be noted, that the shape of the depth map depends on the type of the light that is used in the scene. In this application, point light is used to lighten the scene. As the point light is a light that shines in six directions, the depth map will contain the scene rendered from six directions as well, and thus it is going to create a cube (Figure 3.4a). This technique is also called *omni-directional shadow mapping*. After the depth map is rendered, a shadow test is performed in the fragment shader to decide which fragments are in light and which are in shadow.

Figure 3.4b depicts how shadow testing works. Let us assume one wants to know if a fragment at point P is in shadow. To work it out, one must transform P using T into light space coordinates. The point P's z coordinate holds the depth value, which corresponds to 0.9 in the figure. During the depth map indexing however, it was found that the closest depth on the trajectory from the light to P is at point C. This means, that there is no way for point P to be lit as it is occluded by C, therefore P must be in shadow.

3.6.2 High Dynamic Range Rendering

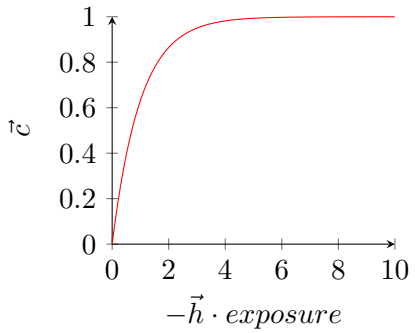
When it comes to rendering lit areas, the contrast of the darkness and the lightness of the scene is captured in a certain range, which is called a dynamic range. In a usual scenario, the range of the colour values of the pixels is clamped between zero and one. This however has a negative impact on the final look of the scene. When it comes to human eyes, the luminance and radiance could be captured in values ranging from zero to thousands, whereas the same scene when rendered, would have the values only from 0 to 1.

Figure 3.5: A comparison of HDR and LDR scene.

To overcome this undesired effect, *high dynamic range (HDR)*[10] rendering was developed (Figure 3.5). In low dynamic range (LDR), the colour values in the buffers are expressed as 8 bit integers. In HDR the colours are expressed in floating point numbers, either 16, or 32 bit, which allocates a lot more memory for storing the values and a high range of luminosity and radiance per pixel can be stored [11]. All the operations when rendering an HDR image are performed using floating point arithmetic.

Not many display devices are capable of showing images in HDR. This is why after rendering an HDR scene, *tone-mapping* needs to be performed. Tone-mapping is a non-linear process where the pixels are mapped to values from 0 to 1, but the range of the luminance is not lost. There are many tone-mapping algorithms available, such as the exposure tone-mapping[10] [10]. It can be seen in algorithm 3.1, where $\vec{c}$ is the clamped colour value, $\vec{h}$ is the HDR colour value, and *exposure* is a parameter that can be adjusted according to what type of scene is rendered. According to the graph curve, it can be seen that the luminance range will be preserved enough for the darker colours whereas lighter colours will be gradually limited.

$$\vec{c} = \vec{1.0} - e^{-\vec{h} \cdot exposure} \tag{3.1}$$



[10] https://learnopengl.com/Advanced-Lighting/HDR

3.6.3 Bloom

Bloom, sometimes also referred to as *glow*, is a very efficient post-processing shader effect when it comes to bright areas of a scene. In a rendered scene the light from light sources or from emissive materials[11] doesn't „bleed“ around its natural borders. In real world scenes however, this bleeding light effect happens quite often, as the lens that the image is captured by can never focus perfectly, especially with intensely lit scenes.

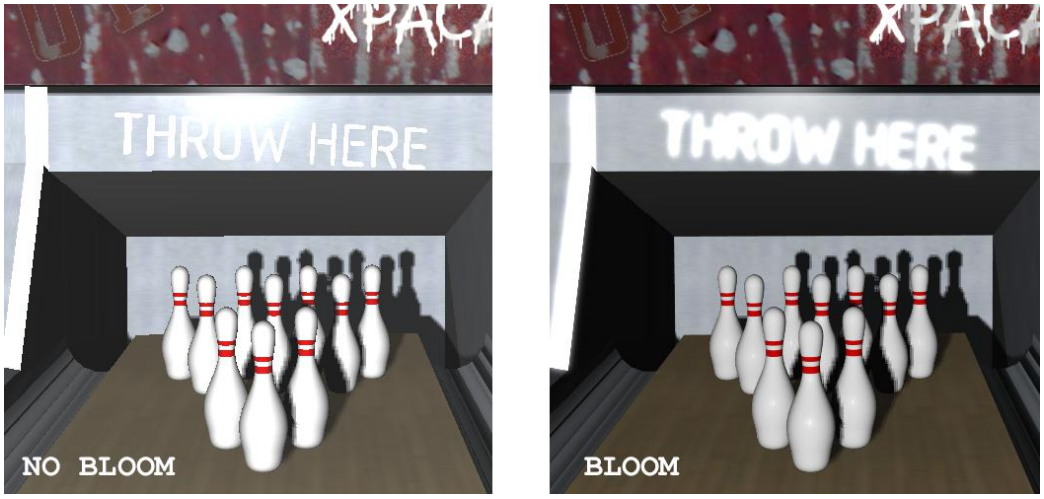


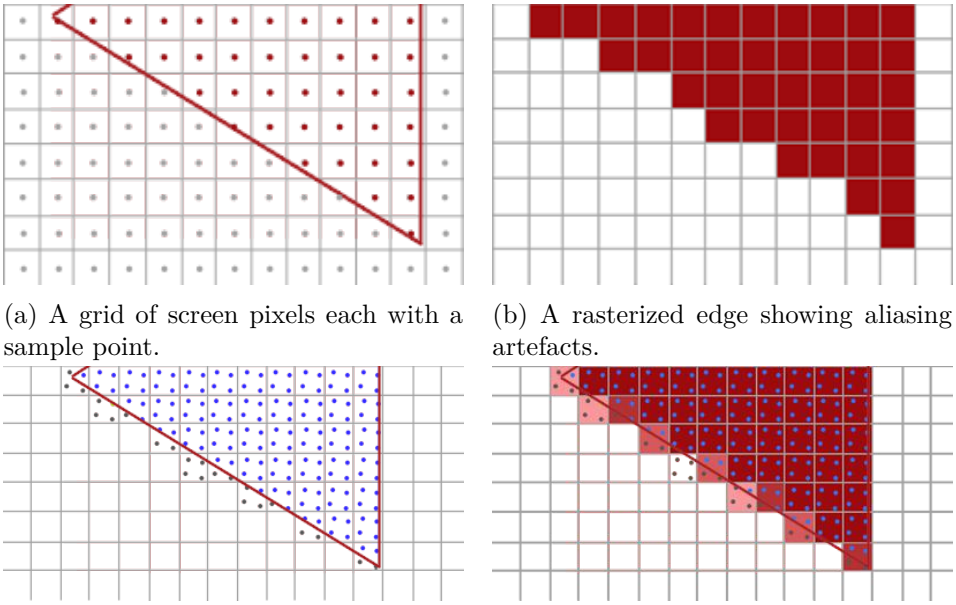
Figure 3.6: In the application, bloom is making the „throw here“ sign glow.

This effect is especially effective in combination with HDR and the terms bloom and HDR are sometimes misused or used interchangeably, even though these are very different effects for very different purposes. In many scenes it is very hard to demonstrate the HDR effect to a viewer without applying bloom. The basis of bloom effect lies in extracting the bright areas of the scene using a threshold filter, blurring these bright areas and applying them over the non-blurred scene. The result can be seen in figure 3.6. One of the efficient ways to blur these bright areas is to use Gaussian blur [10]. The reason bloom benefits from HDR so much is that in HDR the brightness threshold can exceed the value of 1.0 thus offering a much bigger range for adjusting the threshold filter. The details of the implementation will be discussed in chapter 4.

3.6.4 Multisampling

Multisampling, also abbreviated *MSAA* as in *multisampled anti-aliasing* is just one of many methods to achieve anti-aliasing in a rendered scene [11]. Anti-aliasing is a technique that helps to smooth out jagged edges of curved surfaces in an image. When the OpenGL rasterizer takes vertices and transforms them into fragments, it has to determine the screen coordinates of every fragment since it all depends on the resolution of the rendered scene. In figure 3.7a[12] a grid of screen pixels is displayed with a sample point in the centre. The pixels that have their sample point covered by the inside of the shape will be rasterized and the others will be left out. Hence the aliasing artefacts visible in figure 3.7b, where the shape is already rasterized.

[11] Emissive material is a type of material which emits light, but does not contribute to scene's lighting.
[12] Source: https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing



(a) A grid of screen pixels each with a sample point.

(b) A rasterized edge showing aliasing artefacts.

(c) Multiple samples per pixel.

(d) Multisampled edges filled with average colour.

Figure 3.7

Multisampling takes care of this issue by adding multiple sample points in each pixel (Figure 3.7c). The more samples added, the more precise the rasterization. It would seem at first that the fragment shader needs to be run more than once per each rasterized pixel, but the opposite is true. When the shader is run, all sub-samples of the pixel will have stored a certain colour. At the edges where some samples belong to the rasterized shape and some do not, an average colour is computed from the sub-samples and applied to the pixel (Figure 3.7d). This way some colours at the edges might seem to be blending with the rest of environment when viewed from a distance. Nowadays graphic processing units (GPU's) usually support a sub-sample count of 2, 4, 8 and 16.

Chapter 4

Implementation

This chapter will describe how the core features of the application are implemented. It will contain a detailed view of the integration of various libraries such as BulletPhysics and OpenVR as well as a description of some of the most important principles used in the rendering of the final scene, encountered problems and their solutions.

4.1 Application Core

Every application that renders in real time including this one can be divided into two large sub-parts, that is the initialisation of the scene and then the rendering loop. In addition, these kinds of applications can have various rendering states, for example the starting menu, the pause menu, the level scene, and these states can differ when it comes to initialisation and rendering. A person with object oriented programming skills could already figure out that each state will have its own class, but every state class will inherit from a common interface for all states.

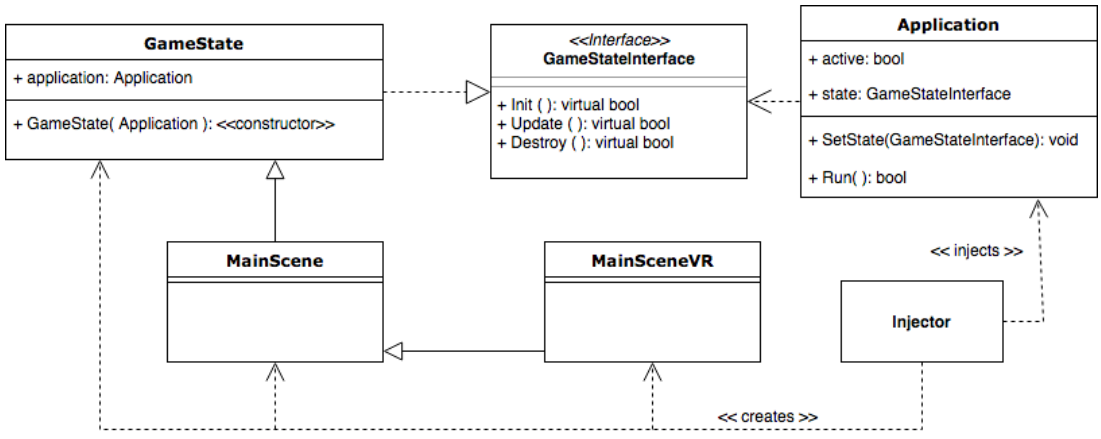


Figure 4.1: A class diagram of the application core showing dependency injection.

This is why `GameStateInterface` was implemented. It consists of virtual functions for initialisation, per-frame update and cleanup. The interface is implemented by the `GameState` class and `MainScene`, the parent class of `MainSceneVR`, inherits from it. Figure 4.1 depicts the relationship between the most essential components of the application. It can also be

noticed that *dependency injection*[1] design pattern is used in the solution. The particular example as to why it was used is that the `Application` class can be dependent on different game states with different implementations without the need to change any code in this class. These states can also then be injected during runtime.

`MainScene`, obvious from its name, is a class container for all the objects needed while initialising and rendering the scene. It houses shaders, models, references to physics world and data needed for post-processing. Calling `Init()` causes to build the scene by creating all the objects needed during per-frame rendering. Afterwards `Update()` is called in a loop in the application core. This method manages the rendering of the objects per frame and steps the physical simulation. It also calculates the time needed to render one frame, as it needs to be supplied to the physical world to properly synchronise to the frame rate.

`MainSceneVR` is an extension of the `MainScene` from which it inherits. The purpose of this class is to handle the virtual reality side of the application, from OpenVR initialisation through event polling and rendering the VR scene. It consists of methods both for the initial setup and those called on a per-frame basis. More about these is mentioned in section 4.6.

4.2 Models

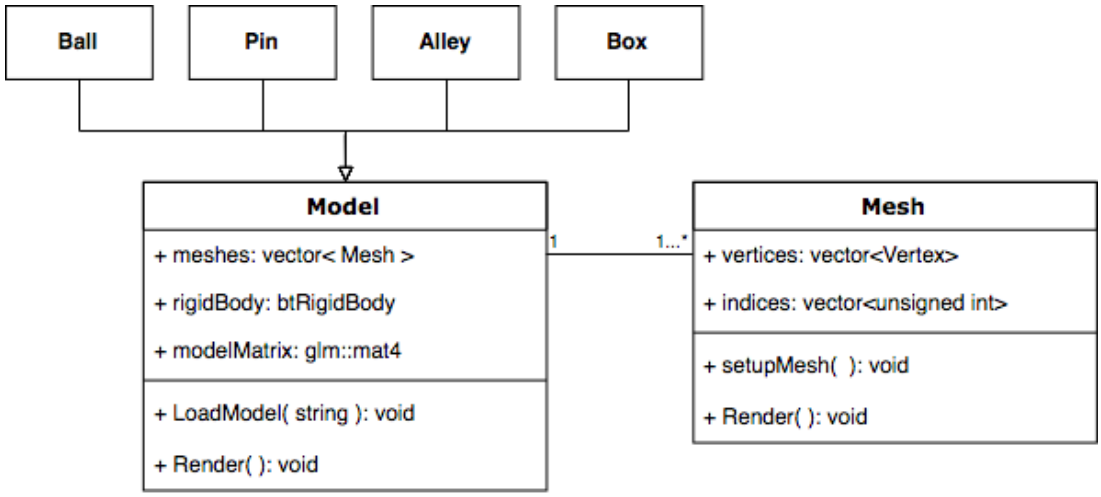


Figure 4.2: A diagram showing a relationship between Model and Mesh class.

Game models were created in Blender and exported as .obj model files. An .obj file unfortunately cannot be supplied to OpenGL directly. Vertices, indices and other elements need to be extracted from the .obj file and supplied to the API to render the model. `Model` and `Mesh` class integrate the Assimp library and process all the data and upload them to OpenGL buffer objects.

`Mesh` is a certain part of a model. Some models only have one mesh, some can have more than one. Each mesh processes the corresponding Assimp `aiMesh` on construction. It extracts vertices, indices, texture coordinates, normals and material colours and saves it into appropriate vectors of data structures. It also prepares the OpenGL buffer objects for rendering in the `setupMesh()` method and loads the textures of the mesh calling

[1] https://msdn.microsoft.com/en-us/library/hh323705(v=vs.100).aspx

`loadMaterialTextures()`. Mesh essentially processes and stores everything needed to render a model in OpenGL.

`Model` consists of one or more meshes. It imports the scene using `Assimp::Importer` object and recursively processes the nodes. In figure 4.2 it is displayed that a few classes inherit from model. `Model` serves as a storage for meshes during initialisation of the game. The reference to the rigid body for physical simulation is assigned only if a class that inherits properties from `Model` is realised. Each entity that can be found in the scene has its own class. This is because the objects have different collision shapes and different properties are needed for their rigid bodies in physical simulation. As an example, there are ten pins in the scene. Each pin looks the same, thus rendering data such as vertices and material colour can be stored in one place in the memory and shared amongst all pins, but each pin needs its own reference to a rigid body for the physical simulation. That is because the world position of each pin is dependent on the position of their own rigid body.

4.3 Physics

As it was stated earlier in section 3.2, Bullet does not have nor require a rendering API to simulate the physics. Therefore the task at hand when integrating physics into a game or application is to visualise the simulation. To make a connection between OpenGL rendered models and BulletPhysics simulated objects, it is important to understand how rigid bodies work. In the previous section it was mentioned that each model contains a reference to a rigid body, the `btRigidBody`.

A `btRigidBody` object is used to simulate rigid bodies with six degrees of freedom[2] in the game. Each rigid body object is created supplying a `btRigidBodyConstructionInfo` structure. It contains mass, inertia tensor, motion state and collision shape.

Figure 4.3: The approximation of a pin collision shape is a cone.

The collision shape supplies the shape of the rigid body. As an example, a rigid body of the bowling ball will have a collision shape of a sphere. When the rigid body of the ball is initialised, it will act on physical forces just like a sphere in reality. This might lead to a question of whether more complex models will also have complex collision shapes. It depends entirely on the model and the scene. However the more complex the collision shape, the more resources it takes to render a frame. It is not that important when there are only several objects in a scene but with increased amount the simulation can struck

[2] https://en.wikipedia.org/wiki/Six_degrees_of_freedom

down performance. In this application for example, the collision shape of the bowling pin is a simple cone (Figure 4.3). It is an entirely sufficient approximation of the shape.

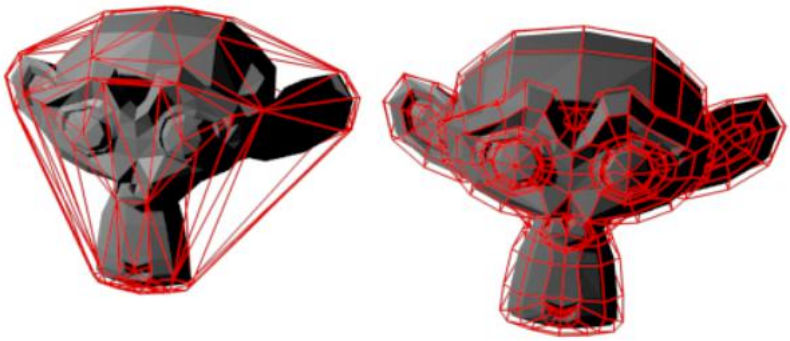


Figure 4.4: The convex collision shape (left) and the concave collision shape (right).

Sometimes however, it is needed for the collision shape to be exactly the shape of the exported model. Bullet provides creating custom collision shapes, or combining simple shapes to generate a compound shape [5].

The basic collision shapes such as sphere, cone, cylinder or box are convex shapes[3], as none of their interior angles are bigger than 180 degrees. Bullet also provides the construction of a convex hull shape that creates the convex approximation of a collision shape. Such approximation can be seen in figure 4.4 on the left. It is quite efficient and does not take up a lot of resources, but the collision would not be as precise as in the real world.

The other type of 2D and 3D shape is the concave shape[4] as seen in figure 4.4 on the right. The bowling lane edges and the gutters have a concave collision shape, so that the ball falls into them when it approaches the edge of the lane. A concave shape is constructed using the `btBvhTriangleMeshShape`, where the programmer needs to supply the vertices of the meshes that OpenGL uses to Bullet to generate a triangle mesh collision shape. Such shapes are recommended to be used only with non-moving (static) rigid bodies, otherwise they can negatively affect the performance of the simulation. In the game, the `Alley` class uses the `btBvhTriangleMeshShape`. The rigid bodies are initialised with each entity in `Entities.h` calling the `pInit()` method.

4.3.1 Rigid Bodies In Bullet

There are three types of rigid bodies in BulletPhysics, namely *dynamic, static and kinematic* rigid bodies. All of them serve different purposes and it is important to choose the right type for the simulation. During the construction of the rigid body, the already mentioned `btRigidBodyConstructionInfo` structure is passed to the `btRigidBody` constructor. One of the properties, the inertia tensor can be calculated from a collision shape with a non zero mass by calling `calculateLocalInertia()`.

Dynamic rigid body is a rigid body that can collide with other objects and acts on the forces applied to it. If a force is applied on this kind of rigid body, it will simulate the movement according to that force. The behaviour of the body depends on several parameters, one of which is mass that has to be a positive non-zero value. In this particular

[3] https://www.mathopenref.com/polygonconvex.html
[4] https://www.mathopenref.com/polygonconcave.html

implementation dynamic bodies are used to simulate bowling balls and bowling pins. Bullet allows to set different parameters for these during initialisation, such as the aforementioned mass, friction, damping and restitution. Some of these properties can be changed dynamically during simulation as well.

Static rigid bodies on the other hand do not act on any kind of force applied to them. They still simulate collision and possess properties as dynamic bodies, such as friction, but they do not move at all during simulation. Static bodies are created when the mass of the rigid body is set to zero. In the implementation they are used to simulate the environment of the level, such as walls and bowling lanes. Basically anything that other objects could collide with.

Kinematic rigid bodies are usually used to control a player character body. This kind of body can move and collide with other dynamic bodies, but does not simulate impact when other bodies hit it. It is a special case of a rigid body and it is not necessarily needed in current implementation, as it could cause unwanted collisions if the player wanted to reach for an object using HTC controllers.

4.3.2 Visualisation with OpenGL

It is important to understand that the rendering API such as OpenGL is just a tool for visualisation of the physical world that can exist on its own without the user's knowledge. Each rigid body has its position in world space. Whether the body currently moves or not, during each render call, a position of this rigid body can be saved from its world `btTransform` by calling `getWorldTransform`. The `btTransform` is essentially a 4x4 matrix that can be easily converted to a `glm::mat4` matrix used in the rendering core of the application. A conversion function, among other utility functions, is implemented in the `BulletUtils` static class.

Once the transform of a rigid body is converted and saved as a `glm::mat4` matrix, it can be passed to the OpenGL shader as the model matrix. The rendered representation of the rigid body is therefore synchronised with the physical world.

4.3.3 Stepping The Simulation

Another important aspect of visualising Bullet with OpenGL is stepping the simulation[5] with correct parameters. The `stepSimulation(timeStep)` method needs to be called every frame. Let it be said that the application needs to run at 60 frames per second (fps). From the simulation point of view one would assume that the `timeStep` should be then set at `1 / 60`. This will work, but if and only if the machine running the simulation is powerful enough to render at 60 fps no matter how many operations it needs to perform during one render call. In majority of scenarios this however will not work. The framerate is likely to drop if post-processing is applied or if there are many objects in the game and shaders and the simulation become really busy. It might also become problematic, when the application is run on another machine which is less powerful. The problem can be spotted by the user easily. The simulation would start running extremely slow, but if the fps was measured, it would show the correct values.

The solution to this problem is fairly simple. At the beginning of each render call, in this case in the `Update()` method of the `MainScene` class, a timer logic is set up to calculate the delta time between each frame (Algorithm 1). The SDL2 library handling the applica-

[5] http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Stepping_The_World

tion window has a function `SDL_GetTicks()` that counts milliseconds since initialisation. This way a time difference, the delta time, between each frame can be calculated and passed to Bullet as the `timeStep`.

Algorithm 1 Calculation of the delta time for simulation steps.

```
1  deltaThen = 0; // during initialisation
2  bool Update()
3  {
4      deltaNow = SDL_GetTicks(); // milliseconds
5      // calculate deltaTime and convert to seconds
6      deltaTime = (deltaNow - deltaThen) / 1000.0;
7
8      //steps the simulation with deltaTime and max 5 sub-steps
9      dynamicWorld->stepSimulation(deltaTime, 5);
10     // perform frame rendering
11     // ...
12     deltaThen = deltaNow;
13     return true;
14 }
```

$$timeStep < maxSubSteps \cdot fixedTimeStep \tag{4.1}$$

Bullet also performs movement interpolation (i. e. estimation) if the maximum number of sub-steps passed to the simulation is greater than one. During the simulation it is important to satisfy inequality 4.1, where the *fixedTimeStep* is set by Bullet to `1 / 60`. If the inequality is not satisfied the simulation is losing time from the mathematical point of view. In this implementation, the simulation needs to run at least at 90 fps, optimally at 120 fps because of the virtual reality element. That is why the maximum number of sub-steps of the simulation has been set to five, which proves to be enough during runtime.

4.4 Light

There are three types of light that approximate the lighting in real world, the point, directional and spot light. These lights can also be implemented using various reflection models that can add intensity, reflection and other properties, such as Phong or Blinn-Phong [10]. Which type of light should be used always depends on the scene and which parts the light should illuminate.

Directional light is usually used to simulate sunlight, as it has an infinite distance from the scene and infinite size. The illumination of the objects depends entirely on its direction. As the bowling lane is a closed area, this type of light is not suitable to be used in the project.

Spot light is a light that can illuminate a certain area in the scene. It has a fixed distance and origin, and the light spreads from it forming a cone over the illuminated area.

Last type of light is the *point* light. It is a light that has an origin in one point and shines in all directions. This is the primary type of light that illuminates the bowling lane in the application.

$$\mathbf{x} = \frac{1.0}{K_c + K_l \cdot d + K_q \cdot d^2} \tag{4.2}$$

The `Light` class is implemented to hold all the required information about the light source later passed to the shader, such as its colour and position. This implementation uses the *Phong reflection model*[9][10] and the light therefore takes advantage of the ambient, diffuse and specular colours of the materials and their shininess factor. The light implementation also employs *attenuation*, otherwise referred to as the intensity of the light. The attenuation is responsible for the scattering of the light across space. It is implemented using equation 4.2, which is very common in computer graphics. Parameter d is the distance from the light to the pixel and constant, linear and quadratic parameters (K_c, K_l, K_q respectively) are chosen according to the type of scene.

4.5 Post-processing

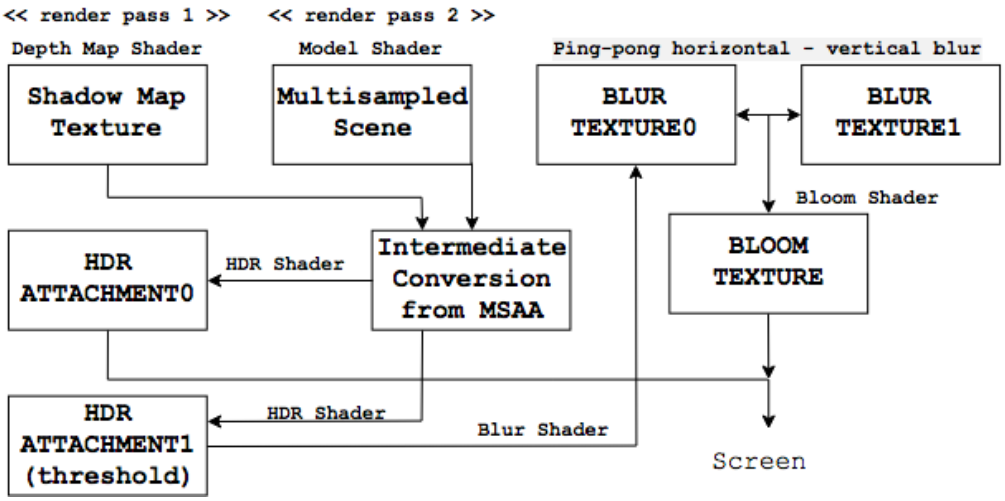


Figure 4.5: A diagram showing how the final post-processed scene is created.

The implementation of post-processing effects relies heavily on OpenGL framebuffer objects. The scene needs two rendering passes and additional shader processing of the rendered scene to get the final result. Figure 4.5 depicts the order of post-processing effects application. It also shows which shader is used in which phase to produce the final scene.

First rendering pass is realised when the scene is rendered from the light's perspective to construct the depth map cube for shadow mapping. To make this rendering pass faster, no textures, materials or colours are applied to the models, because to find out the depth of the scene, these elements are not needed. The depth map is saved into the dedicated off-screen framebuffer with a cube map texture attachment. This texture is later going to be applied over the scene.

The second render pass renders the models and the light with all the textures and materials into a framebuffer with multisampled texture attachment with a number of samples of choice. This causes the OpenGL rasterizer to perform multisampling while rendering the scene. However it is not possible to process the image from a multisampled texture

directly without converting it to a 2D texture. This is done by copying the multisampled framebuffer contents to an intermediate framebuffer with a 2D texture attachment. It is achieved by binding the multisampled framebuffer for reading, then binding the intermediate framebuffer for drawing and calling `glBlitFramebuffer()`. The intermediate framebuffer will contain the multisampled frame of the scene in a 2D texture attachment. The shadow test is performed at this stage and the depth map is applied to the scene as well.

Afterwards, the shader for HDR is used to convert the scene to an HDR scene. Tone-mapping and gamma correction[6] is applied and the result is rendered to a quad situated in the normalised device space coordinates[7]. This quad is rendered by calling `RenderQuad()` whenever it is needed to save the post-processed image to another framebuffer. The HDR framebuffer contains two attached textures. This is because the shader for HDR post-processing contains two outputs. This technique is also referred to as *multiple render targets*. The first output of the shader is a fully rendered scene in HDR colours. The second output however performs thresholding. It only renders the pixels that have a certain brightness over some numerical threshold value that can be adjusted. The pixels bellow the limit are discarded. Algorithm 2 shows how threshold brightness is calculated. It is a dot product of the current fragment with the RGB values of 0.2125, 0.7154 and 0.0721 respectively.

Algorithm 2 Threshold filter in GLSL shading language used in this implementation.

```
1 float brightness = dot(FragColor.rgb, vec3(0.2125, 0.7154, 0.0721));
2 if(brightness > THRESHOLD_VALUE)
3         ThresholdOutput = vec4(FragColor.rgb, 1.0);
```

The bright parts of the scene are needed to apply bloom to the final image. The thresholded scene needs to be blurred to create a glowing effect and then applied over the first render target of the HDR shader, the fully rendered scene. There are different ways of blurring the scene. In this implementation, the Gaussian blur was used in a combination with two framebuffers for „ping-pong“ switching between blur iterations. The image is blurred in ten iterations, five horizontally and five vertically. In the first iteration, a horizontal blur is applied and the image is stored to one of the blurring framebuffers. Then a vertical blur is applied to the image and the image is stored to another framebuffer for blurring. These framebuffers switch the image and blur it according to the number of iterations. The final image is applied over the previously rendered scene creating a bloom effect.

The blur shader contains pre-defined Gaussian weights according to which the blur is applied in each iteration. Algorithm 3 shows how the resulting colour is achieved. The texel from the image that is being blurred is retrieved with a slight offset in the texture coordinates and is multiplied by the corresponding Gaussian weight.

The `PostProcessing.h` file contains classes implementing various framebuffers with texture and depth attachments according to what type of texture attachments were needed. All the shaders are stored inside the C++ code in the `ShaderStrings.h` file.

[6] https://learnopengl.com/Advanced-Lighting/Gamma-Correction

[7] The coordinates of the virtual display device, lower left corner corresponds to (0,0) and upper right to (1,1)

Algorithm 3 A part of the GLSL blur shader.

```
uniform bool horizonal;
// pre-defined Gaussian weights
uniform float weight[5] =
    float[] (0.227027, 0.1945946, 0.1216216, 0.054054, 0.016216);
// size of single texel
vec2 tex_offset = 1.0 / textureSize(image, 0);
// current fragment
vec3 result = texture(image, TexCoords).rgb * weight[0];
if(horizontal)
{
    for(int i = 1; i < 5; ++i)
    {
        result +=
            texture(image, TexCoords + vec2(tex_offset.x * i, 0.0)).rgb *
                weight[i];
        result +=
            texture(image, TexCoords - vec2(tex_offset.x * i, 0.0)).rgb *
                weight[i];
    }
}
else // ...vertical blur
```

4.6 HTC Vive Integration

Integrating a virtual reality system into an application brings its own challenges and complications. The `hellovr_opengl`[8] example in the OpenVR library provides the basics on how to access tracking data and how to draw a scene including the Vive controller models to the headset's screen. The example is very basic and does not provide any clues on how to initiate interaction with objects. Object picking will be discussed in section 4.6.3. However it is two thousand lines long in a single `.cpp` file, so to actually understand how the VR system works, it is better to analyse the code and draw a diagram of how to incorporate the system into this application and note down which functions to call at what time during runtime.

4.6.1 Initialisation process

The application needs to be fully initialised before it can start rendering. Figure 4.6 describes the order of functions that need to be called to initialise every component that is needed. The detailed description of these functions is listed bellow:

1. **Obtaining the VR system pointer.** When initialising the VR system by calling `VR_Init()`, the VR system pointer is returned. This pointer is needed to access tracking data, events and other parts of the initialised VR system.

2. **Initialising components.** The function `InitializeComponents()` takes care of the initialisation of render models and the Vive Compositor. Render model initialisation requests the appropriate interface from the VR system and the Vive Compositor is initialised by calling `VRCompositor()`.

[8] https://github.com/ValveSoftware/openvr/tree/master/samples/hellovr_opengl

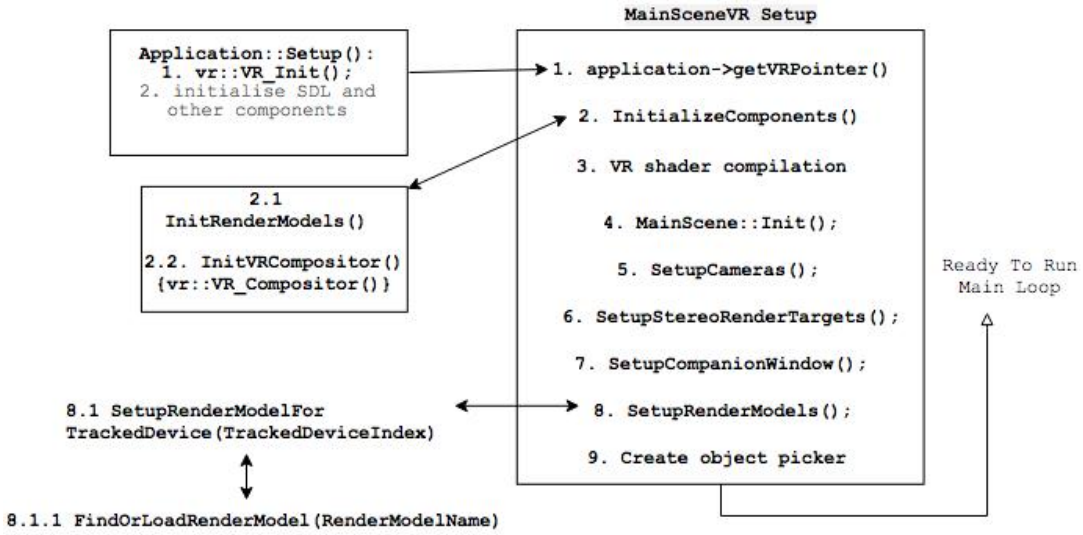


Figure 4.6: The initialisation order of the application.

3. **VR shader compilation.** There are three shaders needed to draw the VR scene properly in addition to other shaders that handle the rendering of the main scene. The shader for render models takes care of rendering the models of tracked devices, such as controllers, in the scene at their current position. Another shader is the companion window shader that is responsible for rendering the output to the companion window on the screen. The last shader is the controller shader that can be used to draw the axes of the controllers.

4. **Main Scene initialisation.** Model loading and scene setup is situated in the parent `MainScene` class. Therefore to load the shaders, lights, post-processing and the models for the scene, the `Init()` method of the `MainScene` has to be called.

5. **Setting up cameras.** When rendering in VR, two cameras need to be set up instead of one, because the scene needs to be rendered from both eyes, which have a slightly different viewpoint. When rendering the scene, it also has to be rendered twice per frame. Setting up the cameras consists of requesting the projection and the eye pose matrix of both eyes supplying the appropriate eye index. These matrices will be needed during rendering when the view and the projection matrix will need to be supplied to OpenGL.

6. **Setting up render targets.** Since the application needs to render two images per frame, one for each eye, it needs to have two render targets to store these images and then supply them to the VR Compositor for distortion. This is done by setting up four framebuffers. Two for each eye, one of them with a multisampled texture attachment and one with a 2D texture to convert the multisampled texture to. The need to convert from a multisampled texture to a normal 2D texture is described in section 4.5. As to why multisampling, or anti-aliasing in general, is important in a VR scene is mentioned in section 2.5.

7. **Setting up companion window.** The companion window is a window on the computer's screen projecting the images of both eyes next to each other. This way for

example an observer can look at what the user wearing the headset is doing in the VR world. The companion window can also serve as a quick debugging tool so the programmer does not need to wear the headset every time they need to test a small change in code. The window is set up by initialising OpenGL buffers for the position and the texture coordinates of a quad rendered in normalised device space. The images of the scene from both eyes are bound to the quad just before each rendered frame.

8. **Setting up render models.** The application needs to detect which devices of the Vive system are connected and active during initialisation. Then it has to load a model for that device from the SteamVR platform and prepare it to be ready to render into the scene during the main loop. The `bVRRenderModel` class was implemented to hold information for each new render model, such as its vertices and textures and to initialise OpenGL buffers so these models can be rendered. Each unique render model is saved into an array of render models. When `FindOrLoadRenderModel()` is called, it then searches the array of loaded models and loads a new one only if it has not been already loaded. The function `SetupRenderModelForTrackedDevice()` is called during the application initialisation, but also afterwards per-frame, when the application is detecting whether the `VREvent_TrackedDeviceActivated` event happened in the system.

9. **Creating Object Picker.** `ObjectPickerVR` is a class handling the interaction with the objects using HTC Vive controllers. It is closely tied to Bullet, as the object picking with a VR controller requires to perform *raycasting*[9]. It will be mentioned in depth in a later section.

4.6.2 Rendering process

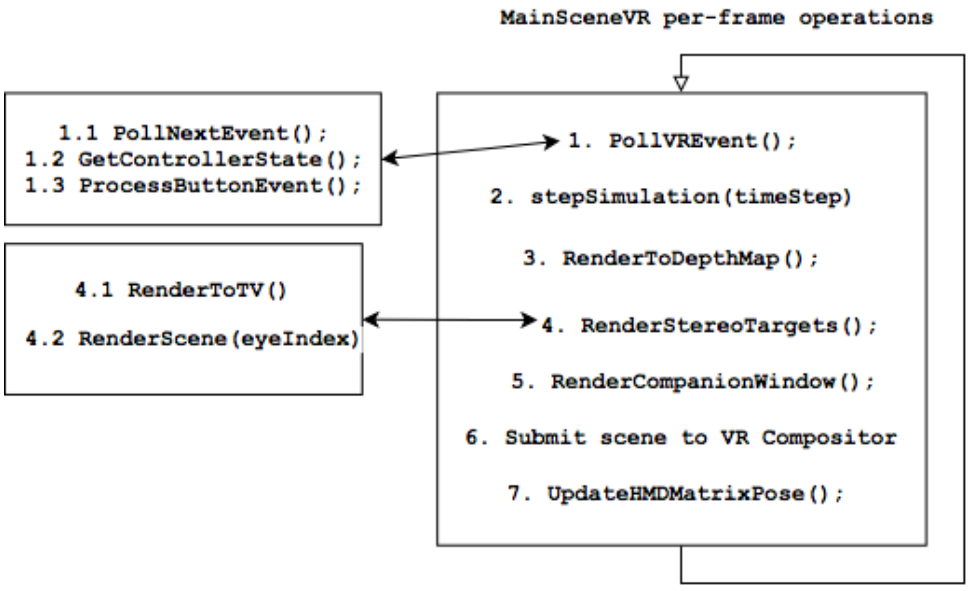


Figure 4.7: The initialisation order of the application.

[9] http://bulletphysics.org/mediawiki-1.5.8/index.php/Using_RayTest

The rendering process takes care of drawing every frame to the screen and to the HTC Vive HMD display applying transformations from the VR Compositor. Figure 4.7 describes the order of rendering operations. The detailed description of these methods is written bellow:

1. **Processing VR System Events.** The `PollVREvent()` method checks for events that happened in the VR system, like new device activation or deactivation. Afterwards, the `vrSystemPointer->GetControllerState()` is called to get the state of attached controllers. Button events are processed in the `ProcessButtonEvent()`, that checks which button was pressed and what action should the application take. Trigger button controls object picking.

2. **Stepping the simulation.** This method is called through the `btDynamicsWorld` pointer that holds the data about the physics world. Detailed description about this method is in sub-section 4.3.3.

3. **Rendering the scene's depth.** This method is responsible for rendering the depth of the scene needed for the shadow test. As it takes its own render pass to render from light's point of view, it is separated from the rest of the scene rendering.

4. **Rendering stereo targets.** The `RenderStereoTargets()` method first takes an additional render pass with a camera that is being situated close to the bowling pins to render the zoomed view to the TV screen above the bowling lane. This kind of an effect would be a part of the 2D GUI overlay in non-VR applications. The reason for this is just to inform the user how many pins they hit, as the long bowling lane prevents from seeing it properly.

   Afterwards the appropriate framebuffers are bound and the scene is rendered two more times. Once from the left eye and once from the right eye into multisampled texture attachments. These are then converted to 2D textures, ready to be submitted to the VR compositor.

5. **Rendering the companion window.** Just before the images are submitted to the VR Compositor, they are projected to the companion window as two textures bound next to each other on a quad rendered in normalised device coordinates.

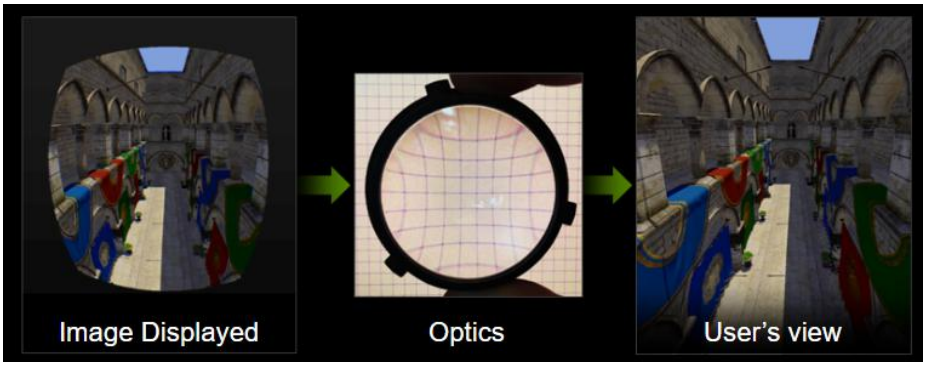


Figure 4.8: The distortion the image goes through (barrel and pincushion distortion respectively) before the user's eyes process it.

6. **Submitting to the VR Compositor.** This is not a difficult task once the scene is rendered. The `vr::Texture_t` is constructed with appropriate parameters for both eyes. The first required parameter is the `void` pointer of the `uintptr_t` (unsigned int pointer) of the OpenGL texture ID. Additional two parameters are pre-defined values of the `openvr.h` file, the `vr::TextureType_OpenGL` and the `vr::ColorSpace_Gamma`. Once the texture is in a VR system format, it is ready to be submitted to the Compositor by calling `vr::VRCompositor()->Submit(vr::Texture_t)`. The Compositor applies appropriate distortion to the image, so when the user views it through the Vive lens it fills the user's field of view (Figure 4.8).

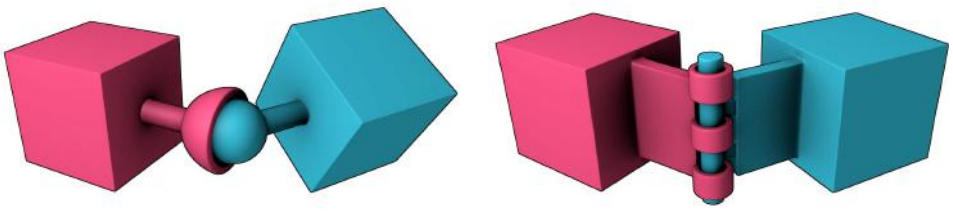
7. **Updating HMD pose matrices.** It is a general knowledge in computer graphics that the *projection*, *view* and *model* matrix needs to be supplied when rendering a scene. However in VR, usually the algorithm 4 applies. The *eye* matrix is the matrix that provides stereo disparity as both eyes have a slightly different view. It can be obtained from the system by calling `GetEyeToHeadTransform()`. This method returns a matrix, whose *inverse* is the matrix needed in the algorithm 4. The application already possesses the appropriate eye projection and eye pose matrices, as they were saved during camera setup.

Algorithm 4 The model-view-projection matrix in VR.

```
1 // matrix = projection * eye * view * model
2 mvpMatrix = eye_projection * eye_pose * HMD_pose * model;
```

Calling `vr::VRCompositor()->WaitGetPoses()` with appropriate parameters updates the HMD poses. This will save poses in an array of tracked device poses data structures, the `vr::TrackedDevicePose_t` structures. Looping over the pose data structures, the HMD pose of the device will be saved in the `mDeviceToAbsoluteTracking` property. The matrix needs to be inverted before it can be treated as the *view* matrix in the application.

4.6.3 Interacting With The Scene



(a) The ball-socket constraint. (b) The hinge constraint.

The basic principle behind the object interaction is the appropriate use of BulletPhysics *constraints*[10]. When a constraint is applied to a rigid body, it makes the rigid body behave according to certain rules, limiting its range of motion relative to some point[5]. There are different types of constraints, some of them allow the body to move in all six degrees

[10] http://bulletphysics.org/mediawiki-1.5.8/index.php/Constraints

of freedom, and some limit their movement only along a certain axis like the hinge constraint depicted in figure 4.9b. Properties of every constraint can be adjusted so it behaves according to need.

When picking up the object, the point-to-point, `btPoint2Point` constraint is used. It is also alternatively named the *ball-socket* constraint (Figure 4.9a). When the bowling ball is picked up, it is attached to the Vive controller with a point-to-point constraint, allowing it to be moved around when moving the controller in user's hand.

The rigid body needs to be targeted appropriately by the user and Bullet needs to recognise it in order to pick it up. This is achieved by performing a *ray test*. Ray test is a method where an invisible ray is cast from an origin in a certain direction. The ray's distance can be adjusted. Whenever the ray hits a body, it returns it as the result of the ray test. Then it becomes possible to perform further actions using the body that has been hit. This method can be easily incorporated to the VR scene.

Ray casting is performed with the origin at the Vive controller axes origin and the ray is shot in the direction of -Z axis, as that is the forward direction in this application. Looking at figure 4.10 it is relatively easy to extract the coordinates of the origin and the direction for the ray test from the Vive controller pose matrix. When creating the direction vector for the ray casting, the Z coordinates have to be saved with a negative sign.

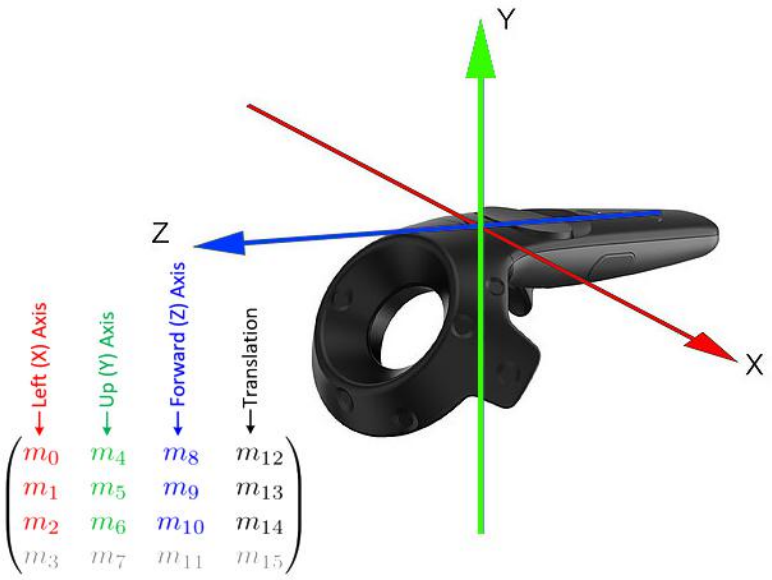


Figure 4.10: The visualisation of controller axes needed for ray testing with the description of the controller's pose matrix indices.

Once the origin and the direction is established the ray-test needs to be called at an appropriate time. That means mapping the function to a certain button event. The button scheme for Vive controllers is depicted in figure 4.11. Objects can be picked up holding down the controller trigger button. Whenever the trigger is pressed down, the `PickBody()` method is called from the `ObjectPickingVR class`. This method performs the ray test by calling the `rayTest()` method from the BulletPhysics world pointer. It saves the result in the supplied `ClosestRayResultCallback`.

If the ray has hit, the body that was hit will be stored in the callback structure. Another small step before applying constraint to it is figuring out whether it is a dynamic rigid body.

Figure 4.11: HTC Vive controller scheme.

Any other type of rigid body has to be ignored. If this is true, the point-to-point constraint can finally be created and applied to the rigid body as seen in algorithm 5.

Algorithm 5 Creation of the picking constraint.

```
1 // the coordinates where the ray had hit
2 btVector3 pickPos = RayCallback.m_hitPointWorld;
3 // upcast the object that was hit to a rigid body
4 btRigidBody *body =
      (btRigidBody*)btRigidBody::upcast(RayCallback.m_collisionObject);
5 // calculate anchor point of the constraint
6 btVector3 localPivot = body->getCenterOfMassTransform().inverse() * pickPos;
7 btPoint2PointConstraint* p2p = new btPoint2PointConstraint(*body, localPivot);
8 dynamicWorld->addConstraint(p2p, true);
```

If the controller trigger is being held down, the `movePickedBody()` method is called every frame. This method needs the updated origin and direction of the Vive controller to move the body with the controller around in the scene by manipulating the constraint.

When the controller trigger is released, the rigid body constraint is removed. The rigid body acts on physical forces normally even when the constraint is applied to it. If the user mimics throwing the ball and releases the controller trigger at the right moment, the ball will simulate the throw and roll on the bowling lane towards the pins.

Chapter 5

Testing

The most famous testing questionnaire about virtual reality applications in context with simulation sickness is the Kennedy Simulation Sickness Questionaire (SSQ) [6]. This questionnaire results in four scores, the total score and the disorientation, oculomotor and nausea scores. However, as this application does not include all the aspects needed to perform such an advanced test, a small questionnaire was prepared for the participants. Questions ranged from asking about the most common VR side-effects to the opinion about the appearance of the application.

5.1 Testing Questionnaire

VR Bowling is an implementation of a bachelor's thesis that was created using C++, OpenGL, physics simulation library called BulletPhysics and OpenVR API to integrate the HTC Vive VR system. After testing the application, please answer the following questions:

1. Was this the first interactive application in VR that you have tested?
2. Have you detected any unpleasant feelings during or straight after the testing, such as nausea, feeling lightheaded, eye strain or other?
3. From a scale of one to ten, rate the visual aspect (appearance) of the application. (1 - worst, 10 - best)
4. From a scale of one to ten, rate how much were you immersed into the VR scene. (1 - not at all, 10 - totally)
5. Did the fact that you were able to interact with the objects help you to immerse into the virtual reality scene more?
6. Was there any noticeable application latency or any framerate drops? Did the scene reflect your movements immediately?
7. How would you rate the difficulty of grabbing and throwing the ball? (suitable, too difficult, too easy)
8. Did you find the heads up TV screen with the zoom to the pins incorporated into the VR environment helpful when you wanted to find out how many pins have you hit?

9. Optional: Is there anything else that you would suggest for a better VR experience?

5.2 The Results Of Testing

The application was tested by twelve users in total. Some of the questions were only informative, others were asked to provide room for the opinion of the user. The results prove that the application is designed appropriately following some of the best VR practices, as none of the participants, whether it was their first time using virtual reality or not, expressed any type of negative side-effects.

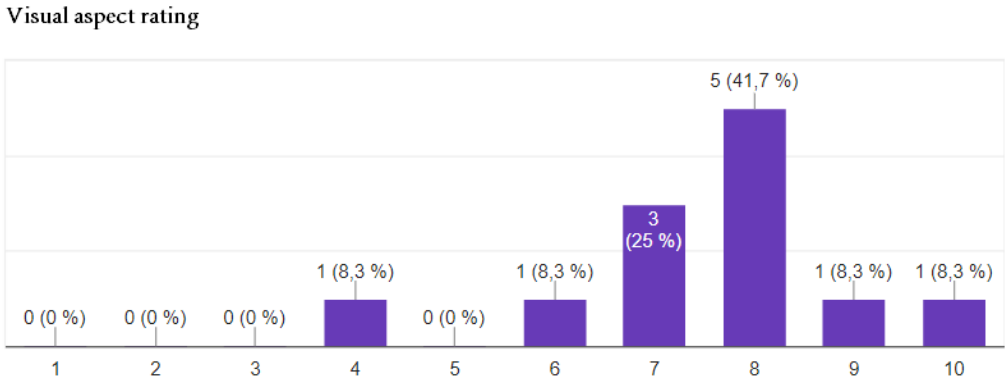


Figure 5.1: The results of question 3.

In question 3 users were asked about the visual aspect of the application, such as adequate lighting or pleasant level design. The results can be seen in figure 5.1, with an average value of 7.5 out of 10.

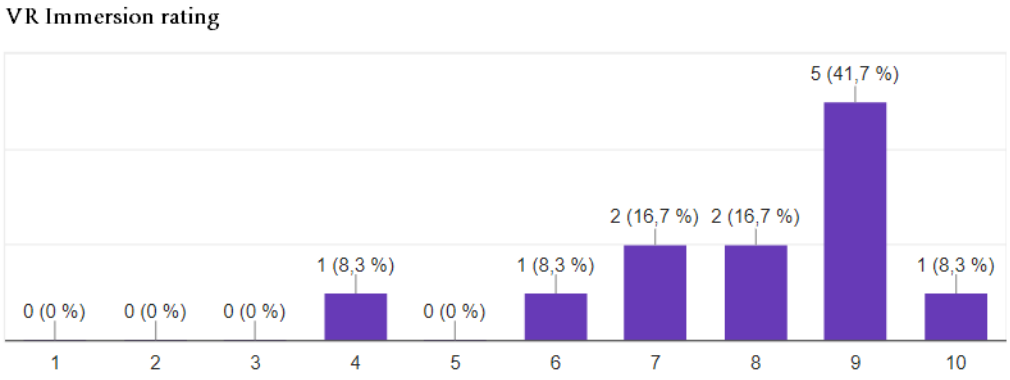


Figure 5.2: The results of question 4.

The results of question 4 in figure 5.2 show that most of the participants were truly immersed. During the testing however it was revealed by some that they were experiencing a little discomfort, because they feared they would hit a real life object or trip over the HTC Vive chord that leads to the headset. Question 5 was answered with a 100 % yes by all the users, meaning that the interaction using motion tracked controllers truly helps to immerse into the VR scene even more.

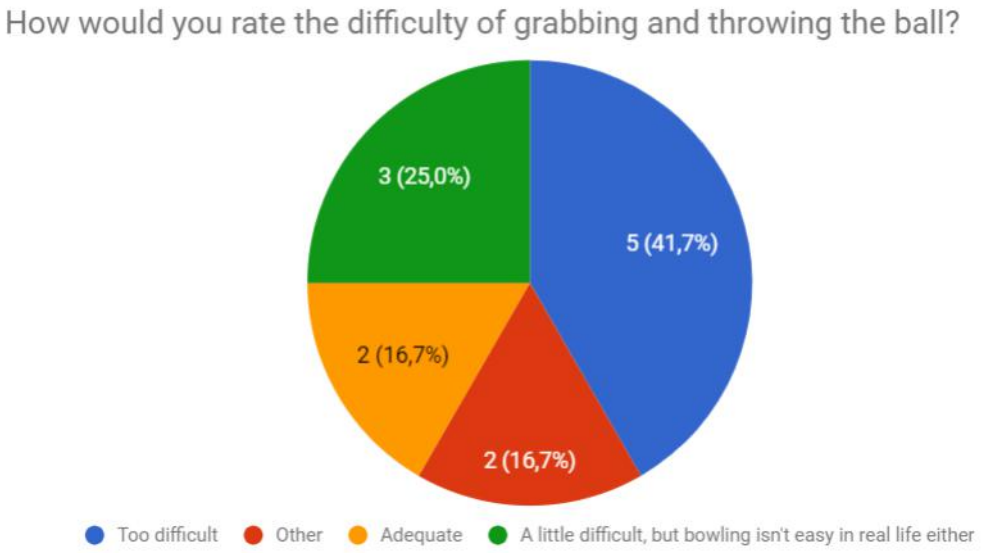


Figure 5.3: The results of question 7.

The application did not show any signs of lower latency or lag, as none of the participants noticed anything out of order. Question 7 however shows that the interaction mechanism of grabbing the objects with a controller could use some improvements. Looking at figure 5.3, it is apparent that almost half of the participants regarded throwing the ball to hit the pins as too difficult. Some participants responded with a custom answer saying that grabbing the ball with the controller is a little immersion breaking, as when compared to real life, the bowling ball has to be held differently than a controller.

The bowling alley is quite long and it is difficult to see what is happening on the other end. The heads up zoom of the pins incorporated into the VR environment as a TV screen was proved to be a good idea, as it helped all of the participants to see how many pins they hit, if any. The last question, which was optional was merely informative. The answers showed that the participants would welcome some additions, such as the bowling game mechanics, like score counting or various different levels, and audio.

Chapter 6

Conclusion

The aim of this thesis was to appropriately demonstrate an interactive physical simulation in virtual reality. This was achieved by creating a very simple virtual reality bowling game. To visualise the simulation of rigid bodies it was important to integrate OpenGL and BulletPhysics and create a visually pleasant representation of an environment for the user. To help achieve this, some of the most popular post-processing effects were used during the visualisation. Another important step was to integrate the HTC Vive virtual reality system using OpenVR, and use BulletPhysics in such a way that the user interaction with simulated rigid bodies is possible using Vive hand-held controllers.

At the beginning of this thesis the reader was informed about the basic methodology in connection with virtual reality also regarding virtual reality application design principles and unwanted side-effects when using virtual reality. Afterwards the application concept chapter followed. It was about the basic scene concept, integrated libraries and the workings of some of the post-processing effects that were used in the application. That was followed by the explanation of the implementation from window creation to multiple render passes and by how the HTC Vive system and BulletPhysics were integrated into the application.

The last part consisted of the user feedback questionnaire and its results. The user feedback proved that this implementation was quite successful when it comes to common virtual reality problems, and no participant experienced any serious negative side effect during testing. However the results also show what the application is lacking and thus provide a lot of room for improvement.

During a future development, this application could become a full scale virtual reality game incorporating bowling game mechanics such as score counting, virtual reality graphical user interface and audio. Due to the object oriented design, it is relatively easy to build upon the project and add additional features, potentially making this implementation a small custom made virtual reality game engine.

Bibliography

[1] Unreal Engine VR Best Practices. [Online; visited on 29.04.2018]. Retrieved from: https://docs.unrealengine.com/en-us/Platforms/VR/ContentSetup

[2] *What is Virtual Reality*. Virtual Reality Society. [Online; visited on 10.04.2018]. Retrieved from: https://www.vrs.org.uk/virtual-reality/what-is-virtual-reality.html

[3] Brewster, D.: *The Stereoscope; its History, Theory, and Construction, with its Application to the fine and useful Arts and to Education: With fifty wood Engravings*. John Murray. 1856.

[4] Craig, A.; Sherman, W.; Will, J.: *Developing Virtual Reality Applications: Foundations of Effective Design*. Foundations of Effective Design Series. Elsevier Science. 2009. ISBN 9780080959085.

[5] Dickinson, C.: *Learning Game Physics with Bullet Physics and OpenGL*. Packt Publishing. 2013. ISBN 9781783281886.

[6] Jerald, J.: *The VR Book: Human-Centered Design for Virtual Reality*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool. 2016. ISBN 978-1-97000-112-9.

[7] Jose, G.-M.; Olga, G.-M.; Katia, C.-H.: Interactive and Passive Virtual Reality Distraction: Effects on Presence and Pain Intensity. *Studies in Health Technology and Informatics*. vol. 167, no. Annual Review of Cybertherapy and Telemedicine 2011. 2011: page 69–73. ISSN 0926-9630. doi:10.3233/978-1-60750-766-6-69.

[8] Linowes, J.: *Unity Virtual Reality Projects*. Packt Publishing. 2015. ISBN 9781785286803.

[9] Phong, B. T.: Illumination for Computer Generated Pictures. *Commun. ACM*. vol. 18, no. 6. June 1975: pp. 311–317. ISSN 0001-0782. doi:10.1145/360825.360839.

[10] Sellers, G.; Wright, R. S.; Haemel, N.: *OpenGL Superbible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional. 7 edition. 2015. ISBN 0672337479, 9780672337475.

[11] Shreiner, D.; Sellers, G.; Kessenich, J. M.; et al.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional. 8 edition. 2013. ISBN 0321773039, 9780321773036.

Appendix A

DVD Content

- The LaTeX source code of the bachelor's thesis text
- The bachelor's thesis *.pdf* file
- The source code of the application including all dependencies
- The compiled *.exe* binary file of the application
- Video showing interaction with the VR scene