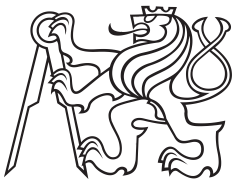
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

# Rendering for virtual reality in high resolution using measured BTF data

Bc. Patrik Schiller

Supervisor: prof. Ing. Vlastimil Havran, Ph.D.
Field of study: Open Informatics
Subfield: Computer Graphics
May 2022

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Schiller** Jméno: **Patrik** Osobní číslo: **474758**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Syntéza obrazu pro virtuální realitu ve vysokém rozlišení s využitím změřených dat odrazivosti**

Název diplomové práce anglicky:

**Rendering for virtual reality in high resolution using measured BTF data**

Pokyny pro vypracování:

Seznamte se s měřenými daty BTF a zběžně prostudujte metody jejich komprimace a dekomprimace se zaměřením na efektivní metody vhodné pro GPU implementaci.
Navrhněte vybraný algoritmus pro dekomprimaci BTF dat tak, aby výsledná implementace syntézy obrazu umožnovala zobrazení s využitím 3D brýlí XTAL 8K firmy VRGINEERS pro sadu jednoduchých 3D objektů s UV parametrizací povrchu.
Navrhněte a vyzkoušejte implementace na grafické kartě s využitím různých programových prostředků (GLSL, CUDA případně OpenCL) a porovnejte je mezi sebou zejména z hlediska rychlosti. Základní algoritmus pro výpočet obrazu při osvětlení bodovým světlem doplňte tak, že pro syntézu obrazu s globálním osvětlováním využívá mapu okolí aproximovanou přiměřeným počtem směrových světel tak, aby zobrazování bylo stále v reálném čase případně bylo progresivní.
Navržené postupy implementujte a otestujte na sadě alespoň pěti testovacích BTF dat a pěti povrchů/tvarů.
Dobrá znalost angličtiny, počítačové grafiky a C++ je nezbytná. Znalost programováni v CUDA na GPU výhodou.

Seznam doporučené literatury:

1) 3D brýle XTAL 8K, https://vrgineers.com/xtal/
2) BTF, https://en.wikipedia.org/wiki/Bidirectional_texture_function
3) Havran, V. and Filip, J. and Myszkowski, K.: 'Bidirectional Texture Function Compression Based on Multi-Level Vector Quantization',
Computer Graphics Forum, Vol 29, No 1, Blackwell Publishing Ltd, ISSN 0167-7055, 2010,
https://dcgi.fel.cvut.cz/home/havravla/btfbase/ .
4) Haindl M, Filip J.: 'Visual Texture', Springer Verlag, ISBN 978-1-4471-4901-9, 2013.

Jméno a pracoviště vedoucí(ho) diplomové práce:

**prof. Ing. Vlastimil Havran, Ph.D. Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

| prof. Ing. Vlastimil Havran, Ph.D.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

# Acknowledgements

I would like to thank my supervisor Prof. Ing. Vlastimil Havran, Ph.D. for providing numerous helpful advice and comments during the work.

Many thanks also go to all those who participated in the usability testing of the application.

Finally, I also thank my family and girlfriend for their support in these difficult and stressful times.


# Declaration

I hereby declare that I am the sole author of this master's thesis and that I have not used any sources other than those listed in the bibliography and identified as references.

In Prague, May 20, 2022

Signature: .....

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 20. května, 2022

# Abstract

The work focuses on the implementation of a real-time renderer targeted for high-resolution Virtual Reality headset XTAL 8K, enabling the examination of various 3D object surface appearances represented by compressed BTF data. The renderer exploits real-time BTF decompression based on the solution presented by Havran et al. in 2010.

The renderer was implemented using three distinct GPU technologies (GLSL, CUDA, and OpenCL), enabling their runtime comparison in terms of performance and rendered image visual quality. An approximation method of image based lighting using an environment map, presented by Pharr et al. in 2004, was implemented to enhance the final visual quality. The method was further extended by a progressive rendering approach enabling BTF evaluation using hundreds of virtual directional lights, while in most cases maintaining 60+ frames per second for the best performing renderer.

A control system consisting of a Wiimote controller was implemented together with GUI to enable intuitive interaction with the application when a VR headset is used.

As a result, an interactive high-resolution VR renderer is available, allowing to examine various BTF materials under various lighting and view conditions.



**Keywords:** BRDF, BTF, VR, XTAL 8K, Real-time PBR, Importance Sampling, Computer Graphics, 3D, DCGI

**Supervisor:** prof. Ing. Vlastimil Havran, Ph.D.



# Abstrakt

Práce se zaměřuje na implementaci real-time rendereru určeného pro náhlavní soupravu virtuální reality XTAL 8K, umožňující zkoumání vizuálních vlastností povrchů 3D objektů reprezentovaných komprimovanými BTF daty. Renderer je založen na dekompresi BTF dat v reálném čase vycházející z řešení, které prezentoval Havran et al. v roce 2010.

Renderer byl implementován s využitím tří různých GPU technologií (GLSL, CUDA a OpenCL), což umožňuje jejich porovnávání za běhu aplikace z hlediska výkonu a kvality výstupního obrazu . Pro zlepšení vizuálního dojmu byla aplikace doplněna o metodu aproximace osvětlení okolím, kterou prezentoval Pharr et al. v roce 2004. Metoda byla dále rozšířena o způsob progresivního vykreslování umožňující výpočet osvětlení pro stovky směrových světel při zachování interaktivní snímkové frekvence. Pro možnost jednoduchého ovládání v rámci virtuální reality bylo implementováno ovládání s využitím ovladače Wiimote, doplněné o grafické rozhraní (GUI), které interakci dále usnadňuje a zpřehledňuje.

Výsledkem je interaktivní aplikace umožňující zkoumání 3D objektů z různorodých materiálů za libovolných světelných podmínek a směru pohledu. Aplikaci je možné využívat jak s náhlavní soupravou XTAL 8K vyznačující se vysokým rozlišením obrazu, tak i s využitím klasického monitoru.



**Klíčová slova:** BRDF, BTF, VR, XTAL 8K, Real-time PBR, Importance Sampling, Počítačová Grafika, 3D, DCGI

**Překlad názvu:** Syntéza obrazu pro virtuální realitu ve vysokém rozlišení s využitím změřených dat odrazivosti

# Contents

# Figures



# Tables

# List of Abbreviations

**AABB** *Axis Aligned Bounding Box.*

**AI** *Artificial Intelligence.*

**API** *Application Programming Interface.*

**BRDF** *Bidirectional Reflectance Distribution Function.*

**BTF** *Bidirectional Texture Function.*

**CDF** *Cumulative Distribution Function.*

**CG** *Computer Graphics.*

**DLSS** *Deep Learning Super Sampling.*

**EM** *Environment Map.*

**FBO** *Framebuffer Object.*

**FPS** *Frames Per Second.*

**FS** *Fragment Shader.*

**GPGPU** *General Purpose Computing on GPU.*

**GPU** *Graphics Processing Unit.*

**GUI** *Graphical User Interface.*

**HDR** *High Dynamic Range.*

**HMD** *Head Mounted Display.*

**IBL** *Image Based Lighting.*

**IPD** *Interpupillary Distance.*

**IS** *Importance Sampling.*

**LDR** *Low Dynamic Range.*

**MC** *Monte Carlo.*

**MLVQ** *Multi-Level Vector Quantization.*

**MRT** *Multiple Render Targets.*

**PCF** *Piecewise-Constant Function.*

**PDF** *Probability Density Function.*

**SIMD** *Single Instruction Multiple Data.*

**SIMT** *Single Instruction Multiple Thread.*

**SVBRDF** *Spatially Varying Bidirectional Reflectance Distribution Function.*

**UI** *User Interface.*

**VR** *Virtual Reality.*

**VS** *Vertex Shader.*

# Chapter 1

# Introduction

Computer graphics today extends to a wide range of various industries, from gaming, cinematography, and multimedia to various types of visualization. Computer visualizations are an essential part to support professional disciplines such as architecture, science or medicine. Thanks to computer graphics, it is possible to display accurate data, which without this power is hard to imagine.

Since the beginning of this discipline, the authors have tried to render 3D objects as accurately as possible, so that ideally it looks indistinguishable from reality. Unfortunately, high image quality is associated with excessive temporal and spatial complexity of the algorithms needed for image rendering. Such algorithms include, in particular, *Ray Tracing*, *Path Tracing*, *Photon Tracing*, and others, which together form methods for computing the global illumination model. Global illumination provides a way of lighting representation that is physically based on the way how light propagates through space and which therefore gives the results closest to reality.

The topic of realistic image synthesis is still the subject of many researches, which try to obtain the best possible image with the least possible memory and consumption demands. Because of the use of computer graphics mainly in interactive applications, such as computer games, modeling or visualization tools, and many more, the demands on time efficiency are the more important. In order to be able to display 3D objects in a good quality while maintaining sufficient rendering times, new approaches are constantly being developed, mostly based on ray-tracing methods, which have proved very useful in rendering a realistic image from the point of view of quality.

To accurately represent 3D models, including the appearance of their surfaces, it is necessary to focus not only on the methods of synthesis, but also on the ways of representing the material itself, which gives the final appearance of the rendered model. Materials are usually defined by many properties (such as roughness, metallicity, refractive index, etc.) along with textures that define the spatial diversity of the material (for example, the pattern of bricks, cloth, concrete and many more.)

The basis of modeling the light properties of materials is the *Bidirectional Reflectance Distribution Function* (BRDF) which describes the properties of reflectivity at a specific point on the surface of the object. As the name implies, this function depends on two directions - the direction of light incident to the surface $\omega_i$ (or also $\omega_l$) and the direction of its reflection $\omega_o$ ($\omega_r$). BRDF can be also interpreted as the probability that a photon incoming from the direction $\omega_i$ will bounce in the outgoing direction $\omega_o$.

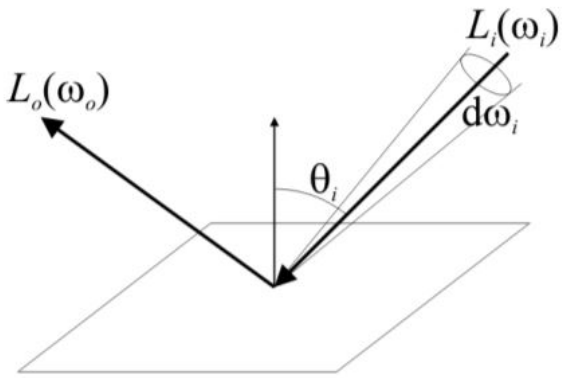



**Figure 1.1:** BRDF illustration

One of the methods suitable for an accurate representation of object materials is the use of *Bidirectional Texture Function* (BTF), which generalises a spatially varying BRDF. The original BRDF parameterization by two direction vectors is extended along the object surface by texturing coordinates (in layman's terms, each texel contains different reflectivity properties - BRDF values). The BTF is therefore dependent not only on different angles of view and the incident light directions, but also on the position on the surface of the object. BTF data are often obtained using a gonioreflectometer, where a planar sample from a given material is scanned in detail in a closed chamber for all selected angles of view and light directions (generally all directions on the unit hemisphere above the illuminated point on the surface). The result of scanning is a relatively large number of textures (images) that capture the properties of the material in different lighting conditions for several view angles. For this reason, BTF is an image-based method of surface representation, where we rely only on the visible properties of the surface and, for example, neglect the surface structure [FH09]. The disadvantage of raw BTF data is its size (several gigabytes per material) and therefore inapplicability for scenes and objects with multiple materials. Fortunately, several research teams have addressed this issue and have come up with a variety of solutions that are both space and time efficient and can be used for real-time image synthesis. One of the compression methods is based on the so-called Multi-Level Vector Quantization of data, which was introduced by Havran et al. in [HFM10], on which this work is based.

With the increasing performance of today's hardware, the term *Virtual Reality* (VR) has become a common standard. Nowadays, it is still considered the best way to experience a computer-generated environment. It can be based on a headset that contains two high-resolution displays (one for each eye), which is also equipped with a number of different sensors to control the movement of the head, eyes, hands, or even the entire figure within the room. However, to be able to perceive 3D space within the generated scene, it is necessary to render a separate image for each eye. As a result, image synthesis for virtual reality is much more demanding than traditional desktop rendering. Therefore, despite the performance of today's hardware, it is still not possible to render an image of similar quality as for classic desktop applications (computer games, etc.) at similar frame rates.

The aim of the work is to conduct research on technologies that enable real-time image synthesis for virtual reality (VR) and subsequent implementation of a real-time VR renderer of 3D models with materials defined using BTF data. Based on the research, suitable technologies and techniques will be selected. Emphasis will also be placed on the use of more similar technologies, which will then be compared in terms of performance - it should be primarily a comparison of `GLSL`, `CUDA`, and `OpenCL`. The result of the work will be an interactive application that can render objects with BTF materials into a virtual reality headset `XTAL 8K` with the highest possible frame rate.

# Chapter 2

# State of the Art

Rendering methods of the modern day can be split into two main categories, which has not changed for the last decades since modern rendering APIs emerged.

The first category represents real-time computer graphics used for entertainment (games), CAD software, and various 3D editors, where performance is the most important. Most of these applications are based on one of the three mainstream graphics APIs represented by `OpenGL`, `Vulkan` and `DirectX`. All three mentioned APIs represent rendering based especially on *Rasterization* methods. The majority of graphics cards (except those used for mass parallel computations) are highly optimized for rasterization (primarily of triangles) and perform very well.

The second category represents the so-called offline rendering, which stands for image synthesis that is not generally performed in real-time. That is given by the complex algorithms and lighting models based on methods of *Raytracing* which is in general much more computationally intense than the rasterization mentioned. However, these renderers can produce photorealistic images, which is convenient for various types of visualization (architecture, design, etc.), movies, or art rendering.

Since 2018 with the introduction of the NVidia `GeForce 20` series, a new rendering approach is available for high-end graphics cards marked as `RTX`. These cards are capable of hardware accelerated raytracing together with rasterization, which means that they contain additional hardware components called `RT-cores` dedicated specifically for raytracing operations (traversal of data acceleration structures, ray-triangle intersections, etc.). However, fully raytraced scenes in high resolution are still not achievable with interactive framerates (in terms of high-quality rendering with various types of effects, which is used, for example in games). Instead, frames computed in low resolution with noise (low number of samples per pixel) are denoised using *Artificial Intelligence* (AI) trained on a specific set of scenes, and subsequently upscaled using a technology called *Deep Learning Super Sampling* (DLSS). AI computations are handled by another type of newly designed GPU components called `Tensor cores`.

That said, real-time hardware raytracing of complex scenes is currently limited mainly to applications that can benefit from AI denoising. That is currently featured only in a subset of modern AAA games because the AI must be trained on the scenes present in the game. However, both RT-cores and Tensor-cores can be used for other types of applications, where the computations can be hardware accelerated using these components - both raytracing and artificial intelligence. These applications include Blender, Maya, OptiX framework, etc.

## 2.1 Rasterization

The computation of a single image within a graphics application can be divided into several parts. In the case of using rasterization methods, it is a so-called rendering pipeline, which is known mainly when using graphical rasterization APIs like `OpenGL`, `DirectX`, or `Vulkan`. It defines the individual logical steps of processing the geometry to be drawn, from transformations to the final rendering on the screen.

Newer versions of the mentioned APIs offer the so-called programmable rendering pipeline (image 2.1), which offers the ability to influence various parts of the rendering process using shaders. Shaders are user-defined functions that run in parallel for a given instance of a problem. The problem can represent, for example, the computation of vertex transformation in the case of the *Vertex Shader* (VS), the computation of the fragment color in the case of the *Fragment Shader* (FS), or other computations in the case of other types of shaders (geometry, tessellation, compute, etc.). Each instance is represented by the same code, only with different data. It follows that the set of instructions is the same, only the data are different - we are talking about the architecture SIMD (*Single Instruction Multiple Data*) or SIMT (*Single Instruction Multiple Thread*). This makes it possible to process large amounts of data in a relatively short time - in parallel.

## 2.2 Raytracing

In the case of global illumination methods (such as raytracing), the process is different, however it still has something in common with rasterization. It is mainly a parallel processing of the resulting pixel color (it is generally claimed that raytracing is the reverse process of rasterization). Scene processing is performed separately for each pixel. For this reason, global illumination methods are much more demanding than rasterization with the growing screen resolution. That is because every ray has to traverse some subset of the scene (when an appropriate data acceleration structure is used), and that applies for all types of rays (primary, secondary, shadow, etc.). Behind every pixel can be many rays traversing the scene.

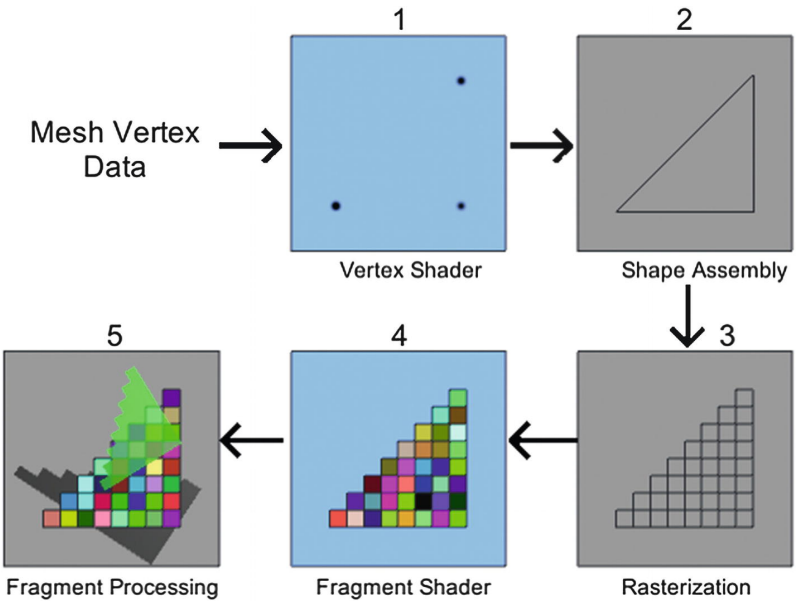



**Figure 2.1:** OpenGL rendering pipeline, rasterization method

The advantage of raytracing is mainly the ability to compute the image based on real-world principles of light propagation through space. Using a `ray` primitive, it is possible to gather more information about the scene than in the rasterization process. Among the additional information, it is possible to evaluate a neighborhood of a shaded point on the surface, which can be used, for example, for indirect lighting, which is essential for photorealistic rendering. In other words, each surface point in the scene can serve as an origin for tracing additional rays that can gather more information needed to calculate the final lighting, color, and shadows.

Many tricks and approximations are used in applications based on rasterization (games, 3D editors, etc.) to simulate the properties of raytracing and to produce high-quality images. However, raytraced images are still considered better and visually closer to reality.

## 2.3 Usage of BTF

One of the methods to represent the properties of heterogeneous materials (spatially varying), which define how light is reflected from the surface, and therefore what color is perceived, is already mentioned *Bidirectional Texture Function* (BTF). It extends previously known *Spatially Varying Bidirectional Reflectance Distribution Function* (SVBRDF) presented by Nicodemus in [Nic65] which has the same parameterization as BTF. However, SVBRDF takes into account only the local properties of a material and is therefore not suitable for the representation of coarse (non-flat) surfaces.

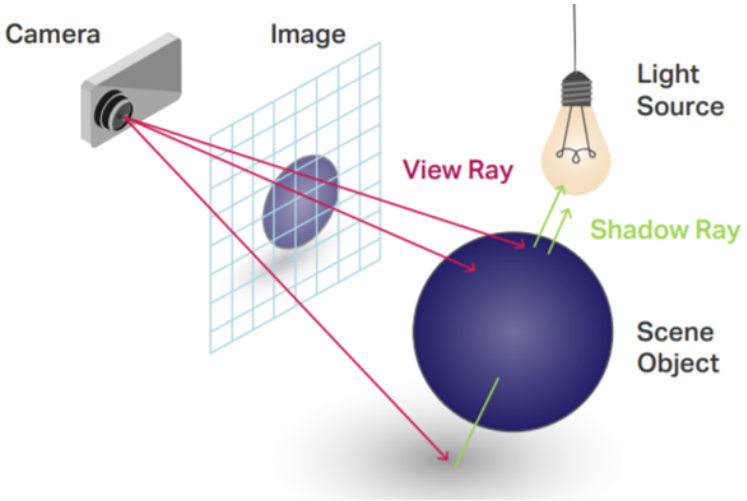



**Figure 2.2:** Raytracing diagram [1]

In contrast, BTF includes nonlocal scattering effects, such as masking and self-shadowing that depend on other parts of the surface than the current shaded point. Because of that, some of the BRDF constraints (presented in chap. 4.1.3) may be violated, and therefore the encoded BRDF values are referred to as *Aparent BRDFs* because they may not be physically correct.

The bidirectional texture function was first introduced by Dana et al. in 1999 [Dan+99] together with a database of 60 freely accessible BTF samples. It is still considered one of the most advanced and accurate digital representations of visual properties of real-world materials [FH09]. There also exist other surface representations that can model surface appearance, such as bump mapping, parallax mapping, SVBRDFs, etc. but none of them can carry that much surface information as BTF.

**Figure 2.3:** BTF used for visualization of materials used in a car interior [FH09]

[1] `https://gfxspeak.com/2020/09/28/the-levels-tracing/`

Usage of BTF is relevant in terms of photorealistic rendering, where the visual accuracy of the rendered material is essential. Such use cases are in design, architecture, automotive industry, or even in a cultural heritage [KCL18], where the rendered image should match the expected reality. In such a way, it is possible to simulate the appearance of the object without the need to create real-world prototypes. This is convenient, for example, when designing a car interior consisting of various types of leather (figure 2.3), or when designing the interiors or furniture of a building.

# Chapter 3

# Computer Graphics Technologies

Because of the possibility of using more technologies during an image synthesis, the computation can be divided into two logical steps. The first step is to calculate the transformations and the resulting properties of each pixel, or the part of the surface of the object that the pixel represents. That means texturing coordinates, normal, tangent, bitangent (see fig. 3.1) and other properties of the rendered 3D objects needed for shading. Within a rasterization pipeline, this step can be perceived as all operations ending with the rasterization of a given geometric primitive - steps 1-3 in the figure 2.1. The output is a set of fragments, each with its own attributes, whether computed or interpolated.

In the case of raytracing, the first step is represented by finding an intersection of a ray sent from a particular pixel into a scene (or an iteratively traced reflected ray) with an object in the scene closest to the ray origin.

In both cases, the result of the first part of image synthesis is a buffer of fragments, although in the case of raytracing it is a bit more complicated because of the use of recursion.

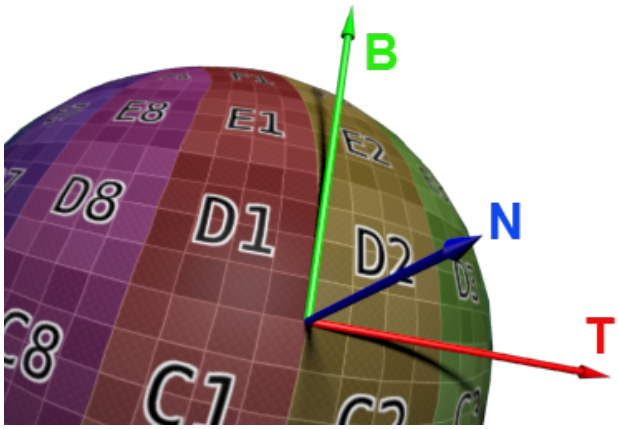



**Figure 3.1:** TBN basis and texturing coordinates illustration[1]

[1] http://rapapa.net/?p=2419

The second phase of image synthesis represents a computation of the final color of the pixel based on the supplied information about the processed part of the object surface (computed in the first phase) - steps 4 and 5 in the figure 2.1. It is in this phase that the BTF data to which this work relates is read and used.

Splitting the computation into two parts has its justification - it is possible to separate the color computation, which is related to the processing of BTF data, from the geometry computations in the scene. As a result, computing a color can be performed using a different technology than the one used for the first part. The implementation of color computation also does not depend on the method used for image synthesis (rasterization or raytracing).

## 3.1 Rendering APIs

APIs for computer graphics programming allow great control over the behavior of the application (renderer), but at the cost of greater programming complexity (it is necessary to write much more code compared to ready-made rendering engines or frameworks). Tens to hundreds of lines of graphics API code can be hidden behind one-line method call in the framework.

### OpenGL

OpenGL is a cross-platform open-standard API for rendering computer graphics on GPUs, developed by the company `Khronos Group`. It is a standardized library that is implemented by most graphics chips manufacturers such as Nvidia, AMD, or Intel. Primarily, OpenGL is used as a rasterization library supplemented by the ability to control some blocks of the rendering pipeline by programmers, using functions called shaders. In general, communication with the GPU using OpenGL is based on the *Client-Server* architecture (or also on the principle of finite / state automaton), where the client is the host application on a CPU and the server represents a GPU processing individual draw commands.

The initial version of OpenGL 1.0 contained only the so-called immediate mode (also available in today's versions of OpenGL), which allows only basic control of OpenGL using commands from the host (CPU) - it is also known as a *fixed rendering pipeline*, where operations directly on the GPU can not be programmed. The data is sent to a GPU by a set of commands specifying the type of uploaded data - mainly its dimensionality (e.g. method *glVertex3f*). Those commands are enclosed by *glBegin* and *glEnd* sequence which defines how the uploaded vertices will be interpreted (by means of a geometric primitive type). All geometric transformations of vertices are controlled using matrices uploaded from the host (managed by OpenGL in matrix stacks). The matrix multiplication is handled by OpenGL in the order given by the particular matrix stack.

Newer versions of OpenGL already contain a programmable pipeline, within which it is possible to use shaders to influence the computation of individual

vertices, fragments, etc. Shaders are written in `GLSL`, which is very similar to `C` language. Recent versions of OpenGL also introduce a new approach to maintain geometry and its data on GPU, using specialized buffers and vertex arrays. Instead of using immediate mode to load the data to a GPU in a vertex-per-vertex fashion, the data is loaded into GPU buffer using a single call. The rendering is then handled by a single draw-call which is associated with the currently bound vertex array (each vertex array can represent some logical set of vertices - e.g., one model). The type of rendered primitives is given by the draw-call itself, so the same buffer can be used for rendering multiple types of primitives (e.g., triangles in the first call, lines in the second call etc.).

Since the release of new graphics cards supporting *RayTracing* (Nvidia RTX 20xx and AMD), it is possible to use shaders in OpenGL (based on the official DXR / DX12), which enables the usage of hardware-implemented raytracing (for example, new types of shaders for BVH tree traversal and computing intersections with primitives - mainly triangles). Raytracing is available for OpenGL after including the *GLSL_NV_ray_tracing* extension for GLSL [NVa].

Although OpenGL is cross-platform technology, it still performs similarly (sometimes better) as DirectX [CP].

### DirectX

DirectX is a set of APIs not only for computer graphics programming but also for sound creation, input processing, etc. [CP]. However, DirectX is not open-source and depends on the platform used (Windows or Playstation). Similar to OpenGL, it implements a programmable rendering pipeline using shaders and it is based on the same principles described in 3.1. Unlike OpenGL, shaders are written in `HLSL` language (instead of GLSL), but both languages are very similar.

DirectX is the official API supporting the new hardware-accelerated ray-tracing (available since the release of Nvidia Turing graphics cards), which is available on Nvidia graphics cards marked as RTX, and some recent cards from AMD. This is specifically an extension of DirectX12 called DXR.

### Vulkan

Vulkan is a cross-platform open-standard computer graphics programming API developed by the Khronos Group, which is considered to be the successor to OpenGL [NVb]. Similar to OpenGL and DirectX, it implements a classic programmable rendering pipeline, shaders can be written in both GLSL and HLSL languages. The Vulkan API allows more control over hardware and commands, unfortunately with this comes more programmer responsibility and API complexity - it is harder to use it. The advantage may be the greater performance in the case of efficient implementation, compared to OpenGL or DirectX.

## 3.2 General Purpose Computing on GPU - GPGPU

There are several solutions for tasks that may not be related to computer graphics and that still require high parallelization. These include scientific computations, artificial intelligence, image recognition, or computer graphics methods that are not based on a standard rasterization rendering pipeline (such as raytracing). Most of the solutions mentioned below allow us to program GPU executable functions that are essentially similar to the already mentioned shaders, but with any number and type of input and output parameters that are not dependent on the inputs and outputs of other blocks in the pipeline. In this case, the functions run in parallel are called *Kernels*.

Parallel computing is based on the so-called SIMD (*Single Instruction Multiple Data*) and SIMT (*Single Instruction Multiple Thread*) architectures, which means that all threads process the same set of instructions on different data. This also indicates that GPU parallelism is not suitable for every kind of problem. The main element of GPGPU processing is a *Kernel*, which represents a "blueprint" of the function which is launched for every processed element (for example, of the array). Kernels are managed in groups (in CUDA called blocks, in OpenCL called work-groups) and these groups form a grid - figure 3.2. Grids can have from 1 up to 3 dimensions, this also applies for all blocks (groups of kernels). One of the main purposes of clustering the kernels into groups is to allow the usage of local memory. Data in local memory are accessible only inside the appropriate group, and the access speeds of a local memory are much bigger than the speeds of a global memory.

Generally speaking, each thread is supposed to process only its subset of the computed data, which can be accessed from a global memory using the thread and block indices. Using these indices, it is possible to compute a unique index that can be used to access data not only in a global memory (see 3.1), but also in other types of GPU memory, such as texture, constant, or local memory. When it is known that the data will be accessed more than once, then the data should be copied to a local memory prior any computations are carried out (it is a programmer responsibility to do so).

```
1 int g_tIdxX = blockDim.x * blockIdx.x + threadIdx.x;
2 int g_tIdxY = blockDim.y * blockIdx.y + threadIdx.y;
3 int g_tIdx = g_tIdxY * (gridDim.x * blockDim.x) + g_tIdxX;
```

**Listing 3.1:** Computing index in global memory (CUDA)

To make it possible to perform some computations on a GPU, it is also necessary to copy the needed data from CPU (Host) to GPU (Device) because in most cases GPU cannot directly access the data on CPU (and it also would not make sense, because the CPU RAM would be a bottleneck). Several types of memory can be used (global, local, constant, texture, etc.), depending on the type and usage of the data.

[2] https://slideplayer.com/slide/16173196/

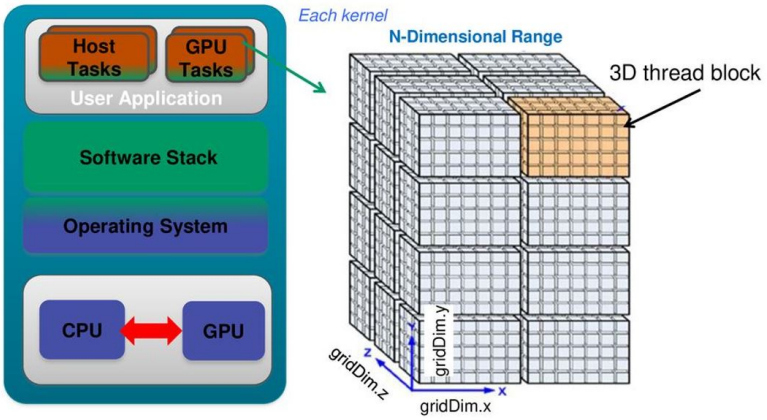



**Figure 3.2:** GPGPU programming model [2]

### 3.2.1 GLSL and HLSL

For some types of computations, it is possible to use traditional graphics shaders available in graphics APIs (OpenGL, DirectX, or Vulkan), especially the fragment shaders. In this way, a reference GPU solution of this work was implemented, which handles the reconstruction of compressed BTF data using chained indexing of several textures, followed by interpolation of the read data.

With some modifications, graphics shaders can be used in a limited mode for general computations (the processed data are passed, for example, in textures). The disadvantage of this approach is the need to manually start a given number of shaders. In the case of a fragment shader, it is necessary to create a primitive, which subsequent rasterization will activate the desired number of fragment shaders. That is not hard to implement, but it is an extra overhead. Not to mention the need to maintain a graphical context and to run vertex shaders and rasterizers that are unrelated to the computation itself. That said, using this approach is appropriate only for graphics applications, where rasterization is needed anyway.

### 3.2.2 Compute Shaders

For the needs of more general computations (e.g., physics or post-processing), graphical APIs contain compute shaders, which more closely correspond to the CUDA or OpenCL kernels. The advantage is that it is only another type of shaders for a particular graphical API and the eventual interconnection of general computations with rendering is clearer and simpler (it is only needed to maintain one context - the graphical one). The disadvantage of compute shaders is their lower performance compared to CUDA or OpenCL. Another disadvantage may be the dependency on the operating system (for example, DirectX with HLSL language is bound to the Windows platform [NV10]).

### 3.2.3 NVidia CUDA

Nvidia CUDA is a framework for general-purpose computing (GPGPU) with a similar programming model as OpenCL and compute shaders. CUDA is characterized by its high efficiency, mainly because of its development closely bound to the development of the hardware for which CUDA is intended (Nvidia GPUs). Another advantage of CUDA is platform independence (Windows, Linux, etc.).

However, the disadvantage of the CUDA framework is the dependency on the device used. CUDA applications can only be run on devices that contain GPUs from Nvidia and where the necessary drivers are installed at the same time. In addition, it is necessary to distinguish individual GPUs according to their *Compute Capability* (CC1.0 - CC8.6 [Q1 2022]), which is mostly determined by the model and age of the GPU used, and which defines the set of implemented functionalities.

### 3.2.4 OpenCL

OpenCL is a GPGPU framework similar to CUDA, but it is independent of both the platform used and the device used. The advantage of OpenCL is therefore the possibility of its use on a wide range of processors, not only the graphics ones (GPU), but also the CPUs. This means that the same code can be run on both Nvidia GPUs and AMD GPUs, and on various CPUs.

Another advantage of OpenCL is that it is supported by a wide range of manufacturers. OpenCL is most supported by AMD with its processors as well as graphics cards, and by Intel. Nvidia also supports OpenCL, but mainly focuses on its own CUDA.

The disadvantage of OpenCL is its lower performance compared to CUDA. This is mainly due to the mentioned device and platform independence, whereas CUDA is developed "tailored" to the target device (Nvidia GPUs) [KDH10].

## 3.3 Rendering Frameworks

Rendering frameworks serve as an abstraction above low-level problems, especially implementation problems of specific graphics APIs (context management, copying data to GPU, etc.), but also as an abstraction above the algorithms used themselves (for example, finding intersections in the case of raytracing, implementation of acceleration structure, etc.). With such a framework, it is possible to write an efficient renderer using less code and knowledge than is needed when using classic graphics APIs and when the programmers have to implement algorithms by themselves.

### NVidia OptiX

Among the most notable rendering frameworks is `NVidia OptiX`[Par+10], which is a rendering framework based on raytracing methods, implementing its own rendering pipeline. It is mainly used for offline photo-realistic image synthesis (for example, in Blender or Autodesk Maya), but thanks to the implementation based on CUDA technology, it can also be used for real-time rendering. OptiX provides a high-level API that abstracts the algorithms needed for raytracing, yet allows the user to control computations within individual rays. As in the case of OpenGL where shaders are used to control the computations on GPU, in the case of OptiX, the behavior can be controlled using functions based on the principle of CUDA kernels. These functions are called programs, they are written in CUDA C (just like CUDA kernels), and define what happens when a ray misses an object in a scene, when an intersection with a bounding box or primitive occurs, or they define how primary rays are generated by a camera. OptiX supports raytracing not only of triangles but also parametric curves (B-Splines, NURBS) and custom primitives defined by AABBs (*Axis Aligned Bounding Box*). Architecture and principles of the OptiX framework are also similar to already mentioned hardware raytracing APIs (DirectX 12 DXR, OpenGL, Vulkan). Functionality is based on *Accelerationn structures*, *Shader binding table*,*Modules* and *Pipeline*.

OptiX is a software renderer (even though it uses GPU), which means that it does not require specialized hardware for the computations, compared to hardware-accelerated raytracing with DXR API (chap. 3.1) which requires RTX capable GPUs. This also means that it can run on any newer type of Nvidia GPU (down to GTX 430 [3]).

## 3.4 Virtual Reality

*Virtual Reality* (VR) is a technology used in many industries, such as medicine, military, building design, entertainment, and more. It enables the user to find himself in a simulated reality represented by an artificially generated environment which can have real-world attributes. The environment should adapt to user actions, such as movement, together with the possibility to interact with it. Virtual reality is often perceived using stereoscopic images, sometimes accompanied by stereoscopic sound and other perceptions like vibrations of a hand-held controller.

Stereoscopic image can be perceived using various devices, one of them being 3D glasses similar to the ones used in a cinema (*shutter glasses*), or using a *VR headset*. VR headset can be represented by a cheap version based on the usage of a regular smartphone as a source of the rendered image, but also by a complex headset, sometimes called *Head Mounted Display* (HMD) that displays stereoscopic images rendered by a computer on two separate

[3] `https://developer.nvidia.com/cuda-gpus#compute`

displays. The main difference between the two types is mainly the image quality and performance.

The disadvantage of virtual reality is the need for a powerful computer (specifically when HMD is used) that can compute relatively high-resolution images (at least 4K) at a reasonable frequency. That is why there has been no greater interest in using VR headsets for a long time, and it is mainly a matter of recent years with the advent of more powerful hardware.

Based on the assignment of this work, the use of VR headset XTAL 8K from the company VRengineers[4](figure 3.3) is planned from the beginning and therefore the research of other devices and solutions does not make sense.

### 3.4.1 Headset XTAL 8K

The headset contains two LCD screens, one with a resolution of 4K (3840 x 2160 pixels) for each eye, as the manufacturer states. In addition, the headset contains many sensors to detect movements of the head, eyes, and hands, and after the installation of the monitoring beacons, the human movement around the room. Virtual reality applications can be controlled by bare hands tracked by headset cameras without the need of using hand-held controllers.

**Figure 3.3:** VR headset XTAL 8K

The headset can be integrated into game engines such as `Unity` or `Unreal engine`, but it is also possible to write your own application that communicates with the headset using the supplied C++ API. This allows sending the computed images within the own application (for example, based on OpenGL library) to individual framebuffers managed by the headset context (left eye (+ detail), right eye (+ detail)). Moreover, it is possible to receive data from available sensors, which allows to control the application behavior.

[4] `https://vrgineers.com/xtal/`

The API documentation is only available to users who are registered to one of the purchased products (it is not possible to access the documentation without purchasing a headset).

Only the data available within the GPU can be loaded into the headset by passing a pointer to an appropriate framebuffer. This means that to upload image data to a headset, it is necessary to use one of the supported graphics APIs (OpenGL, DirectX, and Vulkan). In the case of an image computed on a CPU, or using technologies such as CUDA or OpenCL, it is also necessary to pass the image data using one of the mentioned graphics APIs. Uploading the image data into the context of a given graphics API is needed before it is sent to the headset.

# Chapter 4

# Basics of Global Illumination Methods

To understand how the BTF data used and other implemented functionality work and what they represent, the basics of computer graphics must be covered beforehand.

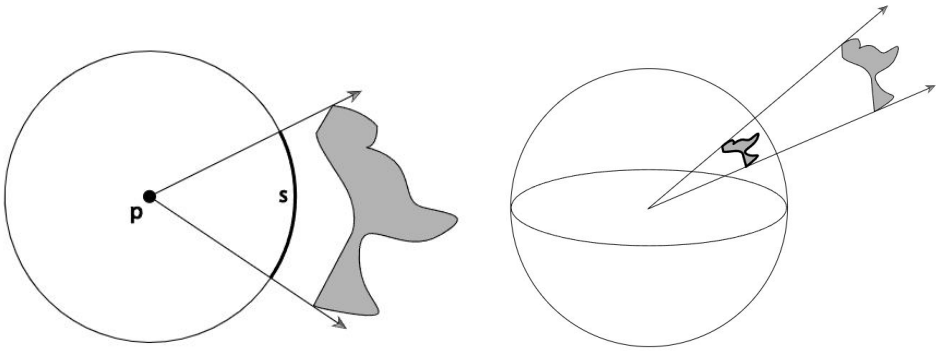
## 4.1 Radiometry

Methods of computer graphics solving how we see observed objects are based on the *Radiometry* discipline which deals with a radiation distribution in space, including a visible light. There also exists a discipline called *Photometry*, which is similar to radiometry but focusing on how (visible) light is perceived by human senses. Because photometry is more subjective (each individual perceives colors differently), the usage of radiometry is more appropriate, mainly because it is objectively measurable. Each physical quantity of radiometry used in computer graphics also has its representative in photometry, only with different units. Also, it is possible to convert between both representations.

### 4.1.1 Solid Angle

Because lots of radiometric calculations in computer graphics are based on (hemi-) sphere integration, the idea of solid angle is used to simplify the integration concept. It represents a non-zero area on a unit sphere, which is defined by some object projection onto it (see figure 4.1b). Similarly to the 2D case with projection onto a unit circle, where length of the arc defined by the projection (**s** in the figure 4.1a) is same as the angle in radians $[rad]$ defining the projection, in 3D the projection onto a unit sphere is defined by a solid angle in steradians $[sr]$ 4.1b.

Using the solid angle we can define all points on the unit sphere representing all possible directions centered at point **p** (sphere center). In computer graphics, point $p$ represents shaded point on the surface and points on the sphere defined by solid angles represent all possible directions of incident or reflected light, generally marked as $\vec{\omega}$. Each direction vector $\vec{\omega}$ can be expressed using angles theta ($\theta$, zenith) and phi ($\phi$, azimuth) in terms of spherical coordinates.




**(a) :** Planar angle, $\langle 0, 2\pi \rangle$ radians **(b) :** Solid angle, $\langle 0, 4\pi \rangle$ steradians

**Figure 4.1:** Planar and solid angle [**pbrt**]

### 4.1.2 Radiance

Despite radiometry discipline contains many quantities like *radiant energy* $Q[J]$, *radiant flux* $\Phi[W, Js^{-1}]$ or *irradiance* $E[Wm^{-2}]$, only the most important one will be covered and that is a *radiance* $L[Wsr^{-1}m^{-2}]$.

Radiance is defined as the radiant power (flux density) accepted or emitted by a differential surface $dA$ perpendicular to direction of light $\vec{\omega}$ with respect to unit solid angle $d\vec{\omega}$ (infinitesimally thin cone of incident directions) - figure 4.2. In case of defining radiance with respect to the orthogonal projection of a unit surface $dA$ onto the surface in which lies the currently shaded point $x$, an additional $cos\theta$ term must be added, to compensate change of the projected area based on the angle $\theta$ defined by the surface normal and the radiance direction (also known as *Lambert's law*). In the case of an orthogonally projected unit surface, radiance is defined by a relation 4.1[Žár+05].

$$L(x, \vec{\omega}) = \frac{d^2\Phi}{cos\theta \, dA \, d\vec{\omega}} \tag{4.1}$$

In computer graphics, radiance represents the color of a light ray, often encoded in RGB. Radiance is also the most useful radiometric quantity, because all other quantities can be derived from it.

### 4.1.3 Bidirectional Reflectance Distribution Function

To mathematically represent the reflectivity properties of materials, the so-called *Bidirectional Reflectance Distribution Function* (BRDF) is used. It represents the ratio between the radiance $L_i$ incident to the examined point $x$ on a surface and the radiance $L_o$ reflected from it. In other words, it is the ratio between the *radiance excitance* and *irradiance* for a given shaded point $x$. BRDF can be also viewed as a *Probability Density Function* (PDF) which relates the direction of incoming light($\vec{\omega_i}$) with the direction of its reflection($\vec{\omega_o}$) by probability, that a photon incoming from a direction $\vec{\omega_i}$ will bounce in a direction $\vec{\omega_o}$.

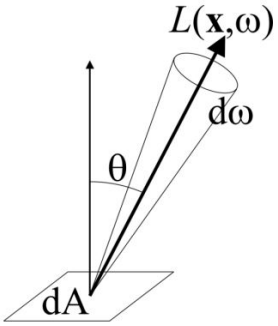



**Figure 4.2:** Radiance accepted/emitted by unit surface $dA$ in direction $\vec{\omega}$ at shaded point $x$ with respect to angle $\theta$ and unit solid angle $d\vec{\omega}$ [RSO]

$$f(x, \vec{\omega_o}, \vec{\omega_i}) = \frac{dL_o(x_o, \vec{\omega_o})}{dL_i(x_i, \vec{\omega_i})(\vec{\omega_i} \cdot \vec{n})\, d\vec{\omega_i}} \tag{4.2}$$

In general, BRDF must respect the energy conservation law, which means that its integral over the unit hemisphere must be less than one: $\int_\Omega f_r(x, \vec{\omega_o}, \vec{\omega_i})\, cos\Theta_i d\vec{\omega_i} < 1, \forall \vec{\omega_i}$ [Žár+05]. It respects the fact that no energy can vanish or emerge, it can only change its type. In other words, the surface cannot reflect more light than it receives. Possibility of the surface being also a light source is in BRDF omitted and handled in the *Rendering Equation* instead.

BRDF is strictly non-negative, $\int_\Omega f_r(x, \vec{\omega_o}, \vec{\omega_i}) \geq 0$, with possibility of its values reaching infinity in some extreme cases.

BRDF also respects Helmholtz principle of reciprocity, which claims that the BRDF values will remain the same when the directions of incident and reflected light are interchanged - i.e. $f_r(x, \vec{\omega_o}, \vec{\omega_i}) = f_r(x, \vec{\omega_i}, \vec{\omega_o})$.

All mentioned properties of BRDF may be violated when an empiric lighting model such as *Phong* is used. These models are suited for real-time rendering mostly in computer games, where the physical nature of light is not that important.

In the process of rendering, BRDF is evaluated only for a subset of directions. Specifically, it is evaluated only for a concrete reflected direction ($\vec{\omega_o}$) given by either a view direction (based on the used camera properties) or by a direction of traced secondary reflections, and all possible directions of incoming light $\vec{\omega_i}$. That is because we are not interested in reflections that cannot be registered by a camera. However, the previously stated is only a mathematical definition. The way how the directions are actually sampled during rendering is discussed in the section 4.2.

### 4.1.4 Rendering Equation

To mathematically describe how much radiance (light) is emitted in the examined direction $\vec{\omega_o}$ for each surface point in the scene, a special integral equation was introduced in 1986 by James Kajiya [Kaj86] . Using this equation it is possible to evaluate total radiance in outgoing direction $\vec{\omega_o}$ for

every point $x$ in the scene, which is also refered as a *Local illumination model*. Whats more, this equation is meant to be applied recursively to model the total radiance equilibrium throughout the scene. Hemisphere integral form of the equation is defined by 4.3.

$$L_o(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + \int_\Omega f(x, \vec{\omega_o}, \vec{\omega_i})\, L_i(x, \vec{\omega_i})\, cos\theta\, d\vec{\omega_i} \tag{4.3}$$

Outgoing radiance at a point $x$ is given by sum of its emissive component $L_e$ (in case the point is part of a light source) and total reflected radiance given by hemisphere integral of all incoming radiance $L_i$ multiplied by BRDF of the given material. Similarly as BRDF, the rendering equation is based on the law of energy conservation.

### 4.1.5 Bidirectional Texture Function

In the case of modeling the reflectivity of non-homogeneous materials, the usage of BRDF is not sufficient, because it describes reflections based only on the directions of incident and reflected light (the same BRDF is applied for all points on the surface). Because of that, BRDF models material appearance only on micro-scale, which omits the geometrical diversity of the surface (bumps, ridges, holes, patterns, etc.).

To be able to model spatially varying reflectivity of the surface, additional parametrization must be added, for example texturing coordinates U,V. This properties are satisfied by *Bidirectional Texture Function* (BTF). In general, this function mathematically describes the reflectivity properties of a material based not only on the incident and reflected light directions, but also on the two-dimensional surface parameterization (position on a surface, given by texturing coordinates). Because of that, BTF models reflectivity on mili-scale, supplementing well-known bump-maps, normal-maps or standard textures. Coarser details are already represented by geometry itself. BTF can be also viewed as a texture, where every texel contains different BRDF.

As it was already stated for BRDF, we are interested only in the radiance reflected towards the camera, therefore the reflected ray $\vec{\omega_o}$ is also marked as a camera view direction $\vec{\omega_v}$. However, material reflectivity must be available for all possible combinations of reflected and incoming light directions, because the camera position and direction are not static in general (camera can move throughout the scene).

Similarly to BRDF, BTF data are tabulated, generally by six-dimensional table where every dimension represents one parameter of BTF. In reality, BTF data are sampled using gonioreflectometer, which produces images (parametrized by texture coordinates) of the material for all desired view and light directions. Because of that, BTF is an image-based method of material representation supplementing already mentioned bump mapping, normal mapping, or texturing, but also, for example, self-occlusion or self-shadowing.

The main downside of this method is its high memory complexity, generally requiring a few gigabytes of images per one material. To make it possible to use BTF in rendering (in particular for real-time rendering), it is needed

to use some compression algorithm, which can significantly reduce memory requirements.

## 4.2 Monte Carlo Sampling

In general, image synthesis is based on the recursive solving of the rendering equation(4.3) for every visible point in the scene, which represents the total emitted radiance from a given point on the surface. The radiance is composed of an emissive $L_e$ component and a reflected $L_o$ component (as stated in chapter 4.1.4). As we can see, the reflected component is given by an integral over a unit hemisphere $\Omega$ which represents all possible directions of the incoming radiance above the illuminated point. To compute an image closest to the ground truth, the whole integral should be evaluated based on some differential solid angle $d\omega$.

However, that is computationally impossible due the large amount of calculations needed to solve even an integral for one illuminated point, not to mention the whole scene and the recursive solving of indirect lighting.

Because of that, integral approximations are used, one of them being *Monte Carlo* (MC) integrator. Monte Carlo numerical integration is based on evaluating the function integral based on random samples of the given function. The more random samples are drawn, the closer the estimation is to the real value of the integral - this is also called convergence. One of the well-known examples of using MC methods is approximation of $\pi$[1] by drawing random samples (with uniform distribution) from a square which has an inscribed circle in it (fig. 4.3).

In computer graphics, Monte Carlo methods are used mainly for direct lighting (light source sampling) and indirect lighting (BRDF sampling). The main problem of the usage of MC in image synthesis is the high variance of integral estimations, when drawing samples from a uniform distribution. The fewer samples are drawn (e.g., per pixel), the more noisy is the rendered image. Moreover, this negative effect is the more noticeable, the more diverse is the area around the shaded point (in terms of incoming radiance).

### 4.2.1 Importance Sampling

Because of the mentioned slower convergence of Monte Carlo integration when drawing random samples with a uniform distribution, the so-called *Importance Sampling* method can be used instead. It is based on the fact that when drawing samples with higher values, the faster the integration will converge to the value of the sampled function integral.

This means that when we want to approximate the illumination of a given point by its environment, it is a good strategy to sample areas with high brightness instead of sampling all directions with a uniform distribution, which also includes dark spots that will hardly contribute to the final reflected

[1] `https://www.youtube.com/watch?v=VJTFfIqO4TU&ab_channel=VincentKnight`

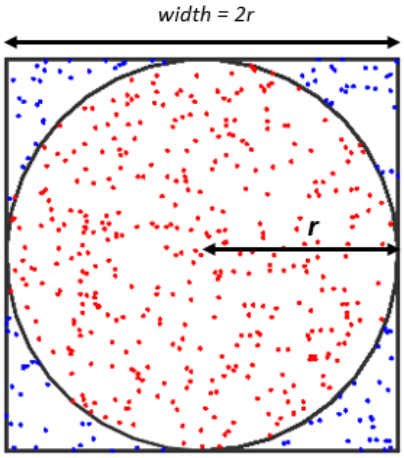
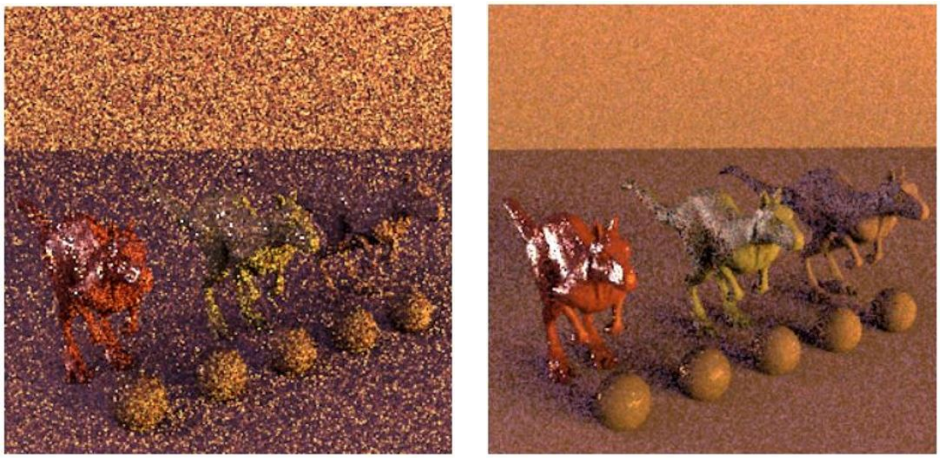



**Figure 4.3:** Monte Carlo $\pi$ estimation simulation

radiance. An example of the difference between uniform and importance sampling of an environment map used for scene illumination can be seen in the figure 4.4



**(a) :** Uniform Monte-Carlo sampling" **(b) :** Importance sampling

**Figure 4.4:** Monte Carlo sampling with uniform distribution vs. Importance sampling of an environment map - 16 samples per pixel [2].

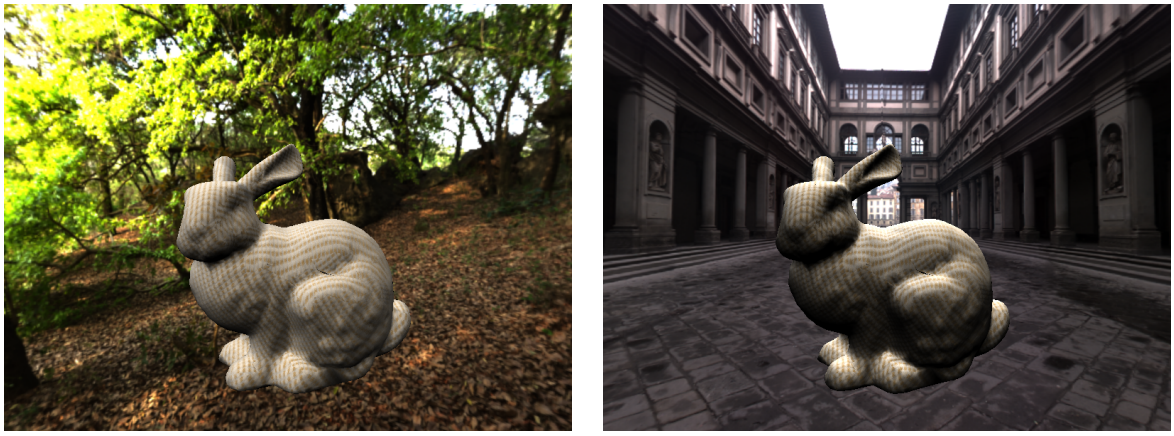
## 4.3 Image Based Lighting

One of the main keys to photorealistic rendering is the choice of the environment and lighting used. Setting up an environment (chapter 4.3.1) enclosing the rendered objects gives the impression that the objects are not simply floating in an empty space. With an appropriate lighting that respects the

[2] `https://graphics.stanford.edu/wikis/cs348b-06/Assignment4`

environment, the impression can even be better. However, manually setting up light sources that correspond to the environment is not that easy, and the illusion can be fairly easily noticed. Moreover, after every change of the environment, the light sources must be reconfigured.

One of the methods to simulate lighting based on the environment is the so-called *Image Based Lighting* (IBL), which can illuminate a given object based on the intensities present in the used image. These images are mostly used also as environment maps (next chapter 4.3.1) and IBL is a natural way to support the final impression. In essence, it projects the light from the environment map onto the object, in the general case acting as one area light source around the whole scene (with various local intensities given by the image).



**(a) :** Environment map `raw013.hdr` "Park" **(b) :** Environment map `raw011.hdr` "City"

**Figure 4.5:** Examples of Image-Based Lighting using an environment map.

Image based lighting is also useful for examination of the behavior of materials under various lighting conditions. Lighting representing the interior of a cathedral, the interior of a car, sky, etc. can be easily setup. The only requirement needed is an appropriate environment map.

### 4.3.1 Environment Map

*Environment Map* (EM) can be viewed as a special type of texture, that is mapped all around the scene, creating a background to improve the overall impression. For example, in computer games are environment maps used for representing the sky, visually filling the empty space where no geometry is present. Environment maps are rendered at infinity because their parameterization is based only on the camera view direction vector, not on the camera position. This means that they are static and they interact only with the camera rotation. This leads to the fact that the usage of EMs can be limited to only represent data, which are visually very far away and do not change based on the camera position - as it was already said, forming a background.

EMs are also used in several graphics programs where the mentioned problem does not matter. For example, when rendering models with materials under various light conditions, IBL using an environment map can simulate real-world lighting conditions of the environment, instead of using some hardcoded light sources. In this case, indoor environments maps are also used.

Environment maps are available in three forms, each with a different representation. All of them are parameterized by a normalized direction vector (for example, a view vector).

The first form is called *Cubemap* and is represented by six separate textures. These textures are mapped onto a virtual box that encapsulates the whole scene at infinity. This approach is mainly used with rasterization APIs like OpenGL, where special samplers were implemented (e.g. `samplerCube`). The direction vector used for sampling is used to pick the correct cube face and to compute the correct UV (texturing) coordinates.

Another form is represented by a single image which parameterization is based on spherical coordinates. Every direction vector can be expressed using two angles, $\theta$ and $\phi$, which can be used as UV coordinates of the supplied image (when normalized by $\pi$). These images must be captured in high quality (at least in 4K), mainly because of the fact that the data are not represented in the same detail through the whole image. For example, the data at the equator are much less detailed compared to the data on poles (where the pole itself is represented by a whole 1 pixel tall slice of the texture). That is caused by non-linear deformation of the image data needed by $\theta, \phi$ parameterization (the deformation can be observed in the figure 7.2).

The last type of parameterization is called *Angular map* which in a simplified form can be viewed as a reflection of an environment in the mirror ball. In this case, UV coordinates are transformed to a unitary disk.

### 4.3.2 Environment Map Importance Sampling

The main problem of standard IBL is the fact that for the most accurate results, the environment map should be sampled for every direction visible from the computed point on the surface. In other words, integral of the whole hemisphere above the illuminated point should be evaluated. As already stated, that is computationally very expensive.

To save rendering time, methods of importance sampling are used (instead of Monte-Carlo sampling), to mainly select environment map texels with high importance (e.g., with high radiance values). To effectively distinguish between texels with light colors and texels representing light sources (direct or indirect light), high dynamic range (HDR) images are used.

One of the possible approaches to effectively sample an environment map was introduced by Pharr et al. in 2004 [PJH17]. The main idea is to generate random variables $(u, v)$ representing image coordinates that favor selecting texels with high brightness over the dark ones.

It is based on transforming 2D joint distribution function sampling into two 1D sampling problems. The input environment map HDR image is first transformed into a piecewise-constant luminance function $p(u, v)$ (PCF) representing the original per-pixel spectral radiance $f(u, v)$. The PCF is then decomposed into separate 1D distribution functions $p(v|u)$, each of them representing the radiance distribution over one column (or row) of the image. For each of the 1D functions, CDF (*Cumulative Distribution Function*) is computed together with its integral. These computed integrals are then used to create a 1D marginal density function $p_u(u)$ over all image columns $u$, for which the CDF is also computed. Both the image PCF and marginal density function $p_u(u)$ are depicted in the figure 4.6.

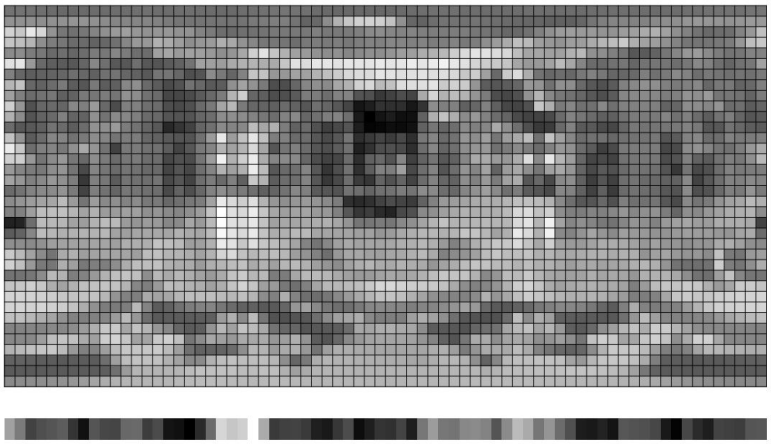


**Figure 4.6:** Piecewise-constant luminance function (top) and its marginal density function $p_v(u)$(bottom) visualization for St. Peter's cathedral environment map - [PJH17]

Sampling of the environment map is based on transforming uniform random variables $\xi_1, \xi_2$ over $[0, 1]^2$ using the appropriate CDFs. First, $\xi_1$ is used to sample CDF value of $p_u(u)$ that results in a new variable $u$. The variable is then used for selection of the concrete column $p(v|u)$ which CDF is subsequently sampled using $\xi_2$, producing a new variable $v$.

That said, sampling of the two CDFs produces new random variables $(u, v)$ representing image coordinates that favor areas of the environment map with high brightness. An additional probability of selecting the sample must be computed to adequately scale the radiance sampled from the environment map under $u, v$.

Given the algorithm mentioned, values $(u, v)$ in range of $[0, 1]^2$ are produced, which can be directly used as texture coordinates to retrieve the appropriate radiance values from the environment map image. Together with $(u, v)$ values the direction vector of environment map sample is produced, which is needed for following shading purposes (e.g. direction to the sampled incident light $L_i$).

# Chapter 5

# Bidirectional Texture Function Compression

As already mentioned, raw BTF data are not suitable for rendering (not to mention real-time rendering) due to their enormous size, meaning several gigabytes per material. Instead, data compression must be performed to lower the memory requirements before the data is used for rendering.

The main requirements for compression are to maintain visual quality along with good compression ratios. An additional requirement can represent the time complexity needed for BTF data decompression and reconstruction during rendering, which is crucial for real-time rendering. The importance of the individual requirements mentioned depends on the particular use case. Photorealistic offline rendering can, for example, favor visual accuracy over the time needed for BTF decompression.

In terms of BTF data compression, various algorithms are available that can be split into three categories. The first category is based on linear basis decomposition using SVD (*singular value decomposition* of a matrix) or *vector quantization*, enabling to separate the compression per each BTF parameter.

The second category uses analytical reflection models to represent the BTF data, for example, by using per-pixel *Lafortune reflectance lobes* parameterized by both the view and illumination directions [HFM10].

The last category of BTF compression methods is based on probabilistic BTF modeling using *Markov random fields* for texture synthesis of BTF subset images to approximate the rough structure of materials. These methods achieve high compression ratios for arbitrary BTF resolution, however the visual quality of some highly non-Lambertian materials is compromised (e.g. metals).

A decent comparison of BTF data compression methods available in 2009 is part of a survey paper [FH09].

## 5.1 BTFbase

For the purposes of this work, which result should be a real-time VR renderer, the `BTFbase` solution presented by Havran et al. in 2010 [HFM10] was selected as the one with very good compression ratios in the range of $1 : 233 - 1 : 2040$ while maintaining relatively low computational requirements on BTF reconstruction during rendering. In addition, the visual quality of materials is negligibly affected by the compression.

The only disadvantage of the solution is the relatively high compression time for each material, however, the compression process is handled only once per material lifetime and is not the subject of the application implemented, and therefore does not affect the rendering itself.

### 5.1.1 BTF Compression Using MLVQ

BTF data compression is based on *Multi-Level Vector Quantization* (MLVQ) [HFM10]. It represents a way how several similar values (vectors) can be expressed by one common vector and the corresponding indices and weights in the lookup table. In the case of compression of BTF data, the insertion of a new record (its encoding) always takes place on the basis of parameterization by individual angles of view directions $\vec{\omega}_o$ ($\theta_o$, $\phi_o$) and incident light directions $\vec{\omega}_i$ ($\alpha$, $\beta$). Each record in the table is represented by its row-index and the value of the corresponding parameter (forming "2D lookup coordinates").

Insertion is performed from the lowest level (table $P_1$, given by figure 5.1). For the fixed position $[x, y]$ in the texture and fixed angles $\theta_o$, $\phi_o$ and $\alpha$ a vector of colors is created which components are parametrized by angle $\beta$. In other words, for each sampled angle $\beta$ representing horizontal angle of illumination direction $\vec{\omega}_i$, representative color is stored into currently constructed vector on a specific position given by the angle $\beta$.

The resulting vector is normalized and compared with the records already stored in the table $P_1$. If a similar vector already exists in $P_1$ (based on similarity metrics), its index and scale are returned, which can be used to obtain the original vector in the process of compressed data reconstruction. Otherwise, when no similar vector is found, the created vector is inserted into the table ($P_1$) as a new record and its index and scale are returned.

In both cases, the returned index and scale data are meant to be stored in the table one level above, in this case table $P_2$, where the parameterization according to the angle $\alpha$ takes place. These returned values are gradually forming a vector to be stored in table $P_2$ the same way as the data are stored in $P_1$.

The same process is also repeated for tables $P_3$ and $P_4$, while indices and scales representing records in $P_4$ are stored in the final table $P_6$ parameterized by texturing coordinates. The whole process of multi-level vector quantization is depicted in figure 5.1.

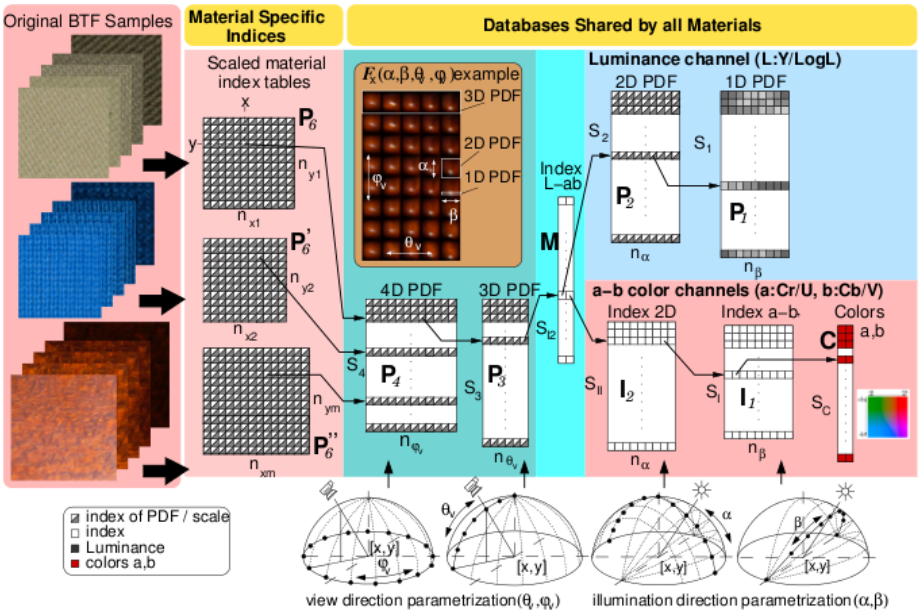



**Figure 5.1:** BTFbase data compression diagram [HFM10]

The individual cells (records) of tables $P_6$, $P_4$, $P_3$ and $P_2$ contain an index and a scale represented by two-colored squares (gray-white) in the figure 5.1. When two vectors are compared, both of them are unfolded down to the records of table $P_1$ by substituting the stored indices by the corresponding vectors in a targeted (lower level) table. This idea for vectors in $P_4$ is illustrated by relation 5.1.

$$\begin{aligned}\vec{v}_{P_4} &= [{}_1\vec{v}_{P_3}, {}_2\vec{v}_{P_3}, \ldots, {}_n\vec{v}_{P_3}] \\ &= [{}_1[{}_1\vec{v}_{P_2}, {}_2\vec{v}_{P_2}, \ldots, {}_n\vec{v}_{P_2}], {}_2[{}_1\vec{v}_{P_2}, {}_2\vec{v}_{P_2}, \ldots, {}_n\vec{v}_{P_2}], \ldots, {}_n[{}_1\vec{v}_{P_2}, {}_2\vec{v}_{P_2}, \ldots, {}_n\vec{v}_{P_2}]]\end{aligned} \tag{5.1}$$

Based on the previously stated, it means that tables $P_6$, $P_4$, $P_3$ and $P_2$ contain only indices and scales used purely for chained indexing of the data and the data itself are stored only in tables $P_1$ and $I_1$ ($C$).

To even more improve the space efficiency (compression ratio), the color compression is separated into the luminance component $Y$ represented by the table $P_1$ and into chrominance $UV$ components represented by tables $I_1$ and $C$. This makes it possible to express the same color with differing lightness using a single chrominance pair of U,V values and several luminance values. Because of that, an additional table $M$ is used to link components from both tables $P_2$ and $I_2$ to represent the encoded color. Individual angles used for parameterization are visualized at the bottom of the already mentioned figure 5.1.

The algorithm also distinguishes between *Low Dynamic Range* (LDR) and *High Dynamic Range* (HDR) materials. In the case of LDR, *YCbCr* color model is used, whereas *LogLUV* color model is used for HDR materials.

### 5.1.2 BTF Decompression and Rendering

Reading and reconstruction of BTF data is based on chained indexing of individual tables and subsequent interpolation of the read data. Reading takes place from the top level table $P_6$, representing the six-dimensional BTF function (dependent on texturing coordinates, direction of view, and direction of light incidence). Table $P_6$ is parameterized by UV texturing coordinates $[x, y]$ and contains the indices of the records in table $P_4$ along with the scales. The scales allow the reconstruction of the original values from the read values representing the normalized data produced by the vector quantization process. In each level, two adjacent values are read, between which it is interpolated. Reading one texture color (BRDF value) requires a reading of 63 values within the tables (1x $P_6$, 2x $P_4$, 4x $P_3$, 8x $M$ (merging table of tables $P_2$ and $I_2$), 8x $P_2$, 16x $P_1$, 8x $I_2$, 16x $I1$) (not counting the scaling factors).

Because the reconstruction of colors is based on a linear interpolation of adjacent values, appropriate view and light direction dependent scales and offsets are also stored. Using this data it is possible to select suitable candidates (by means of texture offsets) to perform linear interpolation using the corresponding weight $\alpha$ in a way given by $c = \alpha \cdot c_1 + (1 - \alpha) \cdot c_2$.

# Chapter 6

# Analysis and Design of the Application

The chapter is devoted to a brief discussion of the requirements for the application. Next, the design of particular parts of the application is introduced together with the approaches and methods needed.

## 6.1 Functional Requirements

- **FR1**: The application provides three distinct renderers (GLSL, CUDA, and OpenCL), it is possible to switch between them in run-time.
- **FR2**: The best-performing renderer can compute two 4K images (one for each eye), which are subsequently displayed using the XTAL 8K VR headset, all in 60+ FPS.
- **FR3**: The application can be controlled using a VR/console controller, a generic API is ready for future addition of a new controller.
- **FR4**: It is possible to switch between different BTF materials and rendered 3D objects, also a texture resolution can be changed in run-time.
- **FR5**: Rotation of the rendered object, camera (+zoom) and environment map is possible in run-time.
- **FR6**: Image Based Lighting approximation by hundreds of virtual directional lights is available. The approximation is handled on application startup, the sampled virtual directional lights are evaluated progressively during the rendering. Lighting using a single point light source is available when IBL is disabled.

## 6.2 Main Design Ideas

Because one of the main goals of the work is to provide the rendered image via VR headset (specifically XTAL 8K), the appropriate API of the VR headset must be integrated into the application. In the case of integrating the API into an already (partly) implemented renderer, it could cause a problem representing the need to reimplement various parts of the application to fit the needs and constraints given by the API. Therefore, a decision was made to base the implementation on the OpenGL demo officially provided by VRengineers company. In this way, it was guaranteed that the API will work already from the beginning of the development.

In terms of the need to provide a comparison between different GPU technologies (namely, GLSL, CUDA, and OpenCL), the application must be designed in the way that it will be possible to switch between different renderers without the need to re-launch the application (all renderers will be accessible at the same time). In the optimal case, the output of all renderers should be the same image. That means that the majority of parameters should be shared between all renders (e.g., transformation of cameras, transformation of 3D objects, currently used materials, etc.).

Optimally, most of the GPU data should be shared to avoid wasting resources (mainly GPU memory). For example, textures should be loaded to the GPU only once and then shared among the renderers.

Finally, an appropriate control system should be designed to provide a decent amount of control over the application while using the headset, which automatically implies that the usage of a keyboard and mouse is unacceptable. Instead, the usage of some controller should be introduced together with meaningful functionality mapping to the particular controls. Additionally, an on-screen GUI (*Graphical User Interface*) could be used to extend the control over more parameters of the application.

## 6.3 Renderer

As it was already mentioned, the algorithm implemented will use several GPU technologies. The most intuitive one is `GLSL`, which stands for the shading language of the OpenGL graphics API. In the case of OpenGL renderer, all work can be performed in a single pass, including the BTF data reconstruction.

For CUDA and OpenCL renderers, it is not that straightforward. The main problem is that it is not possible to perform rasterization using these technologies without implementing a custom rasterizer in the given technology (which does not make sense). Instead, the advantages of OpenGL (alternatively DirectX or Vulkan) will be used to perform rasterization in the preferably most efficient manner, and its results will be used for color computing using CUDA and OpenCL. This implies that the rendering process must be split into two passes.

The first pass handled by OpenGL processes the geometry and produces the so-called G-buffer containing per-fragment attributes (normals, tex. coordinates, tangents, etc. - see Fig. 3.1). The second pass implemented in CUDA and OpenCL will perform shading using the data from the first pass, including the BTF data reconstruction. This method is known as deferred rendering, which is discussed in more detail in the chapter 6.3.3.

### 6.3.1 Offscreen Rendering

The rendering process can be used not only to compute the final images directly displayed on a screen but also to compute a data used in subsequent rendering phases. This method, known as *Offscreen Rendering*, is a key feature required by already mentioned deferred rendering, but also can be utilized for generating various types of textures, post-processing, etc.

In the case of OpenGL and standard single-pass rendering, the default framebuffer is bound to accumulate the computed image, which is displayed directly on the screen. Binding of the default frame buffer (in OpenGL having index 0) is not necessary until some other frame buffer is bound and used. The use of a framebuffer other than the default one is already considered as off-screen rendering because the data are not directly sent to the display and typically are further processed.

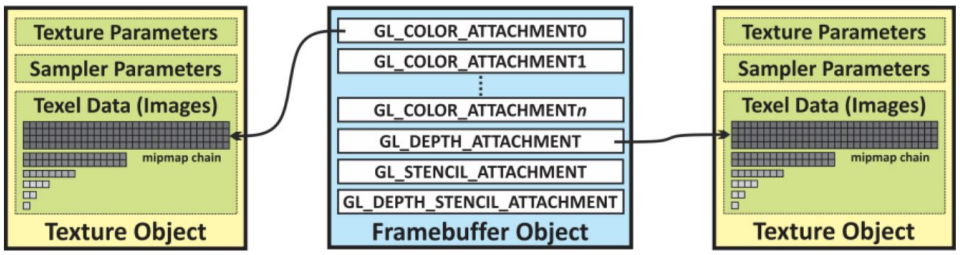



**Figure 6.1:** Framebuffer object layout in OpenGL 4.5 [FA18]

The rendered data can be written to a single frame buffer at a time. However, each framebuffer can have multiple defined layers, also called attachments or images [khrb]. Each of these images can be accessed in the fragment shader and used as an output channel for a given type of data. Every custom-defined framebuffer can have an optional count of attachments (the upper bound is platform and device dependent) used for storing color information (GL_COLOR_ATTACHMENT), an optional attachment for storing depth information (GL_DEPTH_ATTACHMENT), and one optional attachment for storing stencil information (GL_STENCIL_ATTACHMENT) - see figure 6.1. The method of using multiple output color textures is known as *Multiple Render Targets* (MRT) described in the following chapter.

Color attachments can be represented by textures, both 2D and 1D, and by renderbuffers. Renderbuffer can be defined as a special type of texture without any texturing features such as mip-mapping and filtering, which can save some resources when the mentioned features are not needed.

### 6.3.2 Multiple Render Targets

When there is a need to output more than one color buffer during a single render call, the *Multiple Render Targets* (MRT) method is used. An appropriate off-screen framebuffer must be set up and the corresponding output layout must be defined in the fragment shader. Client-side indices of output textures and renderbuffers are defined by GL_COLOR_ATTACHMENT{$X$} enumerators (maximum value of $X$ is system and hardware dependent and can be querried out), while GLSL output indices are set by layout qualifiers. MRT method is mainly used for deferred rendering.

### 6.3.3 Deferred Rendering

Deferred rendering is a multi-pass rendering method that aims to reduce fragment shader (FS) calls. In conventional forward rendering, many fragments (~candidates for the final pixels) are produced, and in many cases there is a chance of more fragments representing the same pixel. Because the fragment shader is invoked for every fragment, it also means that there are many FS calls that are wasted. Based on the fact that the majority of fragment shaders deal with a lot of computations, this can be a problem.

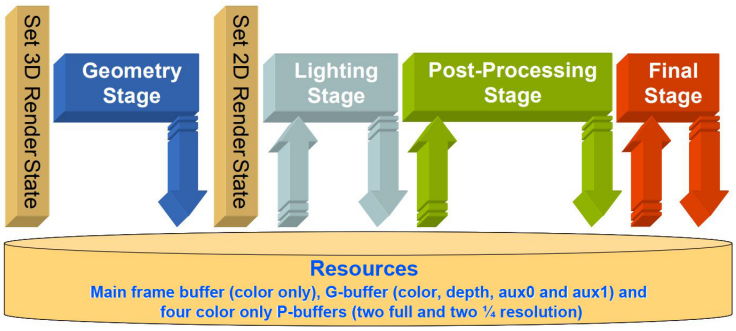



**Figure 6.2:** Architecture of deferred rendering in general [Tha11]

Deferred rendering solves this problem by separating the rasterization with depth testing from the expensive color computation into two passes. In the first pass, the scene is rendered into auxiliary G-buffer textures based on the off-screen rendering method (chap. 6.3.1), saving the necessary fragment attributes needed for calculations in the second pass. This process is handled by a standard vertex shader, but a very simple fragment shader, which in many cases only encodes the mentioned fragment attributes into textures. At this stage, a depth test is also performed to discard unneeded fragments. It still means that there will be many wasted FS calls, but the complexity of the first-pass FS compared to the second-pass FS is completely different.

In the second pass, there will be only as many FS calls as the resolution of the display has pixels, meaning that for each pixel only one FS will be executed. Instead of reading the necessary attributes from shader input variables (e.g., *in vec3 v_Normal*) as would be done in forward rendering, the attributes are read from the mentioned auxiliary G-buffer textures, filled

in the first pass, instead. The following operations are the same as in the original forward rendering pipeline.

Deferred rendering is also a method on how to use the rasterization library to produce the needed fragments, which can be subsequently used by different technologies, for example, by the mentioned CUDA and OpenCL, to compute the final image.

## 6.4 BTF Materials

Multiple loaded materials are also required to provide the user with the ability to observe 3D objects under different conditions, one of them representing different properties of the materials used. Each material is represented by indices needed for the chained BTF reconstruction, as mentioned in the chapter 5.1.2, supplemented by scales and colors data. These compressed BTF data can be loaded and converted to textures using the `BTFtools` module accessible in the original BTFbase implementation. It is the most effective way to use BTF data in GLSL. These textures can also be shared with the CUDA and OpenCL renderers without the need to load them separately (and probably using a different type of memory than textures).

Furthermore, view direction and light direction dependent data are needed for BTF reconstruction. These data are already encoded in textures supplied with the BTFbase implementation, which are meant to be used as faces of a cubemap texture. These data define how to interpolate the data during BTF reconstruction (BTF reconstruction is based on chained linear interpolation). Fortunately, these data are the same for all materials and must be loaded only once.

All materials will be loaded on the application startup, and switching between them will be handled only by re-mapping appropriate index and scale textures. This approach is more memory intensive, however, loading the materials one by one while the application is running could cause performance problems and stutters.

## 6.5 Environment Map Approximation

Because the per-frame importance sampling of an environment map as described in 4.3.2 is expected to be computationally intensive and insufficient for real-time rendering, an approximation alternative is proposed.

Instead of generating many random samples every frame, a fixed count of environment map samples will be computed in the preprocess phase during the application initialization, approximating the environment map by a set of virtual directional lights used in the same way for every computed frame.

## 6.6 Progressive Rendering

In the case of lighting (IBL, chap. 4.3.2) based on the mentioned approximation of an environment map by many directional lights, as given by the requirements, it is not possible to evaluate the fragment shader (mainly BTF reconstruction) for all the lights in a single frame. From personal experience, it is possible to evaluate 3 to 10 lights per frame[1] while maintaining stable at least 60 *Frames Per Second* (FPS).

To make it possible to compute the final color using all available directional lights, the so-called *progressive rendering* approach must be exploited. It is based on splitting the calculations into many frames. It is known mostly from 3D software, where photorealistic renderers are implemented. While the image is being rendered, partial results are gradually displayed, replacing the previous ones. As a result, finer images with less noise are replacing the old noisy ones until the end, when the final rendered image is available.

This approach can also be used in limited scenarios for real-time rendering, in case the scene is static. Then the progressive rendering process will occur only after some interaction is performed (camera is moved, object is moved, etc.). The main advantage of this approach is that the application is still interactive having stable FPS and the image is fully computed under one second after the last interaction was made (in case of having 60 FPS, 5 lights per frame, and under 300 directional lights in total).

Implementation of this approach in the case of the renderer using the rasterization pipeline should be based on composing two framebuffers, representing the result of the previous frame and the result of the current frame. The current frame represents the calculations performed by the fragment shader for a given subset of lights (e.g. 5), which is then composed with the previous frame to produce a new displayed frame. The frame displayed is also saved for the next iteration. Adequate normalization must be performed to scale the color in the desired color range.

## 6.7 Rendering for Virtual Reality

One of the limitations of the XTAL headset API is the need to pass image data using one of the available graphics APIs (OpenGL, DirectX, or Vulkan). This means that it is currently not possible to load data directly from the CPU (such as an array) or from the CUDA context into the headset. However, since OpenGL will be used anyway, at least for rasterization for each of the renderers, it is not perceived as a problem.

Workflow of rendering into a headset will consist of acquiring camera transformation matrices followed by rendering the scene for each eye separately. Rendered data will be sent to the headset as a pointer to a corresponding

[1] The performance is heavily dependent on the radiance distribution in the environment map used, on the average number of light sources evaluated for each pixel, and on the BTF material used.

OpenGL texture, which is required by the XTAL API. All mentioned steps will be wrapped by the render loop responsible for the per-frame rendering operations.

## 6.8 User Interface

The application user interface must be designed so that it provides the user with an intuitive and robust control over the application while using the XTAL 8K VR headset.

The main problem of using a VR headset is the fact that the user cannot see the controller used. This leads to the need for a simple and memorable control system that can be used without the need to see the interaction performed on the control device. That can be achieved by using some of the available console or VR controllers with an intuitive mapping to the application attributes and functions.

In addition, the control system should be implemented in such a way that it will enable another type of controller to be integrated in the future without the need to know the implementation details of the application itself. By this, the application will not be bound to a specific controller that may be deprecated and not available in a few years.

Due to the high number of variable parameters of the application, additional *Graphical User Interface* (GUI) should be implemented to simplify the control over the application.

### 6.8.1 Controls

The controller itself should provide only the minimum of buttons needed mainly for interactions that would not make sense in terms of a GUI. These interactions mainly represent all kinds of rotations, including the camera, model, and environment. The rest of the interactions should be implemented in a GUI, whereas some of the important ones could also be mapped to controller buttons, serving as shortcuts.

Among the controllers available for the application implemented can be, for example, considered the "oldschool" `Wiimote`[2] controller from Nintendo, and a `HTC Vive`[3] controller corresponding to the HTC Vive VR headeset. The main difference between the controllers lies in the targeted use-case.

In the case of Wiimote, the controller is dedicated to interact with a game console (mainly the Nintendo one), which is connected to a television or PC monitor, therefore the controller is meant to be still visible.

However, in the case of the HTC Vive, the controller is designed to be primarily used in virtual reality, where in principle the controller cannot be seen due to the use of a VR headset.

[2] `https://commons.wikimedia.org/wiki/File:Wiimote.png`
[3] `https://www.vive.com/eu/accessory/controller2018/`

**(a) :** Wiimote controller[1] **(b) :** HTC Vive controller[2]

**Figure 6.3:** Discussed controllers comparison.

An additional desktop control system should also be implemented for cases when VR is not used. This involves substituting the camera motion performed by a head, when a headset is used, by a mouse. Other interactions can be mapped to a keyboard, including free camera motion throughout the scene and changing application parameters.

### 6.8.2 GUI

*Graphical User Interface* should provide two types of functionality. Firstly, the mentioned menu can be displayed and used to change the parameters of the application. The second, also a very important functionality, provides the user with information about the interaction performed with the controller, eventually providing some notifications from the application. Without the provided information, controller interactions could be confusing for the user.

# Chapter 7

# Implementation

The implementation was based on the knowledge gained from previous work done during the SVP subject, where various demo applications were implemented to solve possible problems beforehand. For example, a software renderer (written in pure C++) based on the original BTFbase implementation written in old OpenGL and GLSL, was implemented. In addition, a demo application was implemented that combined the CUDA framework with the VRG API to examine possible approaches of displaying a rendered images using CUDA in a VR headset. The last demo application implemented a combined software rasterizer with a kernel ("fragment shader") written in CUDA that handled the BTF chained indexing and color reconstruction. Three most important demo applications were also used for performance comparisons as a part of the performance testing (chap 8.2.6).

## 7.1 Structure of the Application

Eventhough class names of all three renderers evoke the fact that they implement the requisites of a standard renderer, in reality, some of the functionality is handled from the outer scope.

Initialization, render loop, and communication with the VR headset are handled by the core class `App` of the VRG framework used, defined in a `gldemo::core` namespace. The majority of rendering is implemented by the class `BTFRenderer` which combines the common functionality of all three renderers and acts as a core class of the application.

CUDA and OpenCL specific implementation is part of a separate class `CUDA_Renderer`, `OCL_Renderer` respectively. CUDA shading kernel is defined in a separate file *FragmentShader.cu* as a function `__global__ void FragmentShader` defined in a `Kernels` namespace. OpenCL shading kernel is defined in a file *clFragmentShader.cl* as a function `__kernel void main`.

## 7.2 OpenGL Renderer

OpenGL renderer was implemented as the first one, so that it could serve as a reference solution during the implementation of the CUDA and OpenCL renderers. Most of the functionality was derived from the original GLSL 1.0 implementation accessible on the BTFbase webpage[1]. The original implementation was also using an experimental extension that allows the usage of a programmable shader pipeline, despite the usage of OpenGL 1.0 which, in general, does not support shaders. Because of that, most of the work was focused on reimplementing the BTF renderer using a modern OpenGL 4.5.

Because a decent graphics framework (further denoted as a VRG framework) was also bundled with the VRengineers OpenGL demo application, the decision was made to use the demo application as a foundation of the subsequent implementation. This way was also guaranteed that the VR API will be functional from the beginning of the implementation.

The mentioned reimplementation consisted mainly of replacing the old OpenGL 1.0 immediate mode rendering pipeline with a modern OpenGL 4.5 rendering pipeline, which in the end led to writing a new renderer from scratch using some functionality from the mentioned VRG framework. New methods had to be implemented, for example, to allow the initialization of Vertex arrays (VAO) together with Vertex (VBO) and Index buffers (EBO), which are required by modern OpenGL. In addition, vertex attributes and shader uniform variables handling were revised.

### 7.2.1 GLSL

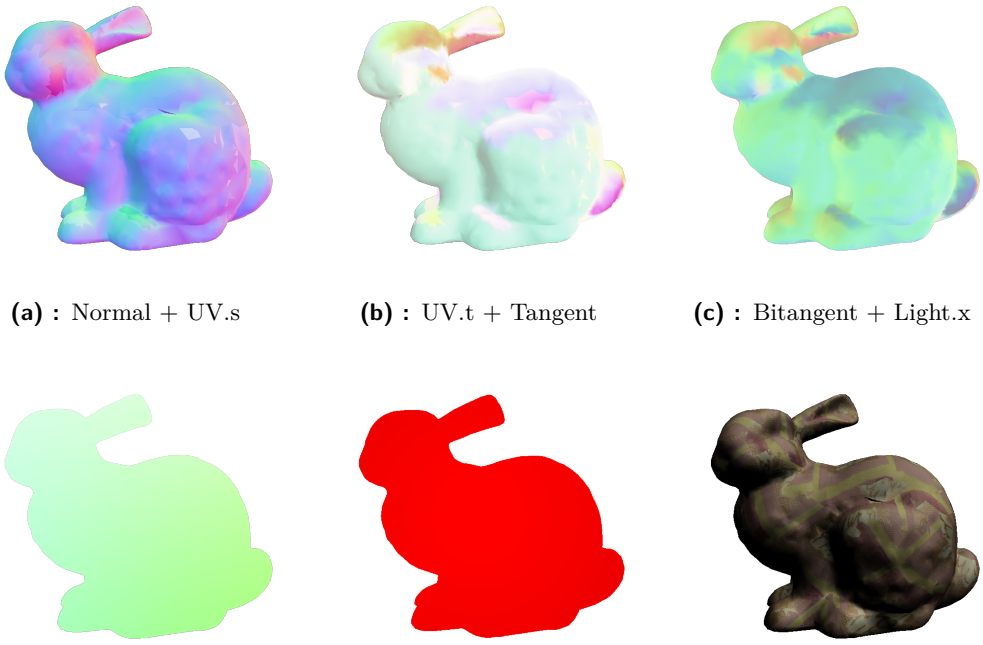
The least modified part of the original code is the GLSL fragment shader which handles the BTF data reconstruction and rendering. Only a few modifications needed for the OpenGL version conversion were made, mainly consisting of designing a new attribute and uniform variables layout. Other changes would involve changing the BTF reconstruction algorithm itself, which did not make sense.

Some minor changes were also made to enable progressive rendering, which is described in the chapter 7.7, and also to enable shading using multiple light sources, which in this case are used to approximate the method of image based lighting (chap: 4.3).

## 7.3 Deferred Rendering

To split the rendering between two distinct technologies, a deferred rendering method was introduced in chapter 6.3.3. The first pass is handled by OpenGL using the same vertex shader used also by the full OpenGL renderer described in the previous chapter. This ensures that all scene transformations will be the same among all renderers.

[1] `https://dcgi.fel.cvut.cz/home/havravla/btfbase/`



**(a) :** Normal + UV.s **(b) :** UV.t + Tangent **(c) :** Bitangent + Light.x

**(d) :** Light.yz + Camera.xy **(e) :** Camera.z **(f) :** Result after shading

**Figure 7.1:** G-buffer renderbuffers rendered by OpenGL and used in CUDA and OpenCL renderers by means of deferred shading (a-e) + the result of rendering (f)

The following fragment shader encodes the shading attributes to the G-buffer based on the method of MRT (*Multiple Render Targets*) which results are depicted in the figure 7.1. G-buffer was designed as an off-screen framebuffer consisting of five color attachments, each containing different data (table 7.1). Color attachments were implemented using renderbuffers instead of standard textures, because there was no need for data filtering or mip-mapping. G-buffer renderbuffers were also mapped into the CUDA and OpenCL contexts to enable their usage in the following shading process, which is similar to the GLSL shader used in the case of OpenGL renderer.

Before the data are stored in the color channels of the corresponding renderbuffer, an appropriate values rescaling must be performed, due to the fact that colors are defined in the range of $\langle 0, 1 \rangle$ (alternatively $\langle 0, 255 \rangle$). Because all the stored data are normalized vectors, the easier the operation is.

| Texture | Format | Stored data |
|---|---|---|
| NUV | RGBA32F | N.x, N.y, N.z, UV.s |
| UVT | RGBA32F | UV.t, T.x, T.y, T.z |
| BL | RGBA32F | B.x, B.y, B.z, L.x |
| LC | RGBA32F | L.y, L.z, C.x, C.y |
| C | R32F | C.z |

**Table 7.1:** Renderbuffer layout of G-buffer used between 1st and 2nd rendering pass of CUDA and OpenCL renderers. Visualization in figure

## 7.4 CUDA Renderer

In contrast to OpenGL, where only pixels representing the shaded object are processed by the fragment shader, in the case of CUDA (and OpenCL), kernels are launched for all pixels of the screen according to the MRT. Knowing that, each frame can be processed by a fixed number of launched kernels. Grid resolution is precomputed during the initialization in such a way that every block has $64^2$ threads to satisfy the size of a warp in which threads are executed by CUDA. The number of blocks in a given dimension is, therefore, given by the upper bound of division of the screen size dimension by the square root of the block size (listing 7.1).

Shading is handled by a single kernel named `FragmentShader`, and the BTF data reconstruction process is similar to the original BTFbase decompression algorithm. Firstly, the needed geometric information is read from the supplied G-buffer textures and rescaled back to the range of $\langle -1, 1 \rangle$. The following process separates pixels that contain surface information from pixels representing the background.

Background pixels are set to have a constant background color, or an environment map texture is applied to the background pixels when it is supplied.

The foreground pixels (representing the rendered object) are then processed in the same way as in the GLSL implementation. For each supplied light source, an incident angle to the surface is computed, which is then used for filtering of unlit pixels. For lighted pixels, the color is computed in a way that is based on the rendering equation 4.1.4.

Lastly, the computed color is saved in the output texture represented by the CUDA surface (chap. 7.4.3).

```
1 float blockWidth = sqrt(THREADS_PER_BLOCK);
2 float gridX = ceil(screenX / blockWidth);
3 float gridY = ceil(screenY / blockWidth);
```

**Listing 7.1:** Computing index in the global memory (CUDA)

[2]The number of threads per block can changed using the configuration file, however, the number should be one of 16, 64, 256, or 1024.

### 7.4.1 Constant Memory

To represent a global data used by all threads, which serve as uniform variables compared to GLSL implementation, constant memory was used. It is a special type of read-only global memory, which accessed data are cached within each multiprocessor [Slo21a]. The contents of the mentioned constant memory are similar to GLSL uniforms memory and are also updated with the same frequency. For clarity, data loaded into constant memory are encapsulated in a struct named `Uniforms`.

### 7.4.2 CUDA-OpenGL Interoperation

When using CUDA in graphics-related computations, in many cases, frequent data interchange (e.g., each frame) between the graphics API and CUDA is needed. For example, when CUDA generated geometry is rendered by a graphics API or in the case of using CUDA for the rendered image post-processing.

However, because the standard process of executing kernels is based on copying data from the host to GPU, performing the computation, and subsequently copying the results back to the host (CPU), for rendering purposes, this approach cannot be used.

The main problem is represented by frequent copy operations of the data between CPU and GPU, which acts as a major bottleneck. Although it is possible to render an image on GPU, then copy it to CPU and afterwards display it, for example, using the *glDrawPixels* method, as it was discovered during the implementation, it is heavily inefficient. For every frame, two data copy operations are performed between the host and device (copy the computed image from GPU (CUDA) to CPU and then display the image using OpenGL, which means copying the data back to the GPU).

Instead of this approach, a specially designed extension called *cuda_gl_interop* was used, which provides a way to share memory between OpenGL and CUDA contexts within a GPU. Before rendering the frame using CUDA, an appropriate OpenGL buffer is mapped to the CUDA context and then written to during the computation. After the frame is computed, the OpenGL buffer is unmapped and can be subsequently used by OpenGL, for example, to display it on the screen. The easiest approach is to use the method *glDrawPixels*.

However, during the implementation, performance problems caused by copying computed data using the glReadPixels function were discovered. After minor research, the method of direct rendering into an offscreen framebuffer was selected as the best performing. This change involved usage of writeable textures in CUDA accessible as surfaces (chap. 7.4.3).

In that case, the color attachment (e.g. `GL_COLOR_ATTACHMENT0`) of the appropriate framebuffer is mapped instead of a regular OpenGL buffer. After the frame is computed, displaying it is only about unmapping the color attachment and copying it to the main framebuffer within the GPU using the method *glBlitFramebuffer*.

### 7.4.3 Textures and Surfaces

Although the use of global memory would probably be more efficient instead of using many textures (in a general case), in the case of maintaining three different renderers is the usage of shared textures (GPU memory) probably the most efficient approach (in terms of memory requirements).

Because the mentioned rendering into an offscreen framebuffer was implemented, not only read-able textures had to be utilized, but also a texture that can be written to. Standard read-only textures in terms of CUDA can be represented by texture objects, however read-write textures must be represented as surface memory objects. Its structure is more similar to OpenGL renderbuffer, which serves only as a memory and does not provide any texturing features (interpolation, mip-mapping etc.).

## 7.5 OpenCL Renderer

The OpenCL renderer represented by `OCLRenderer` class was implemented in a manner similar to the CUDA renderer. One of the main differences between these two technologies is the method of representing the device code. CUDA-specific files with *.cu* extension are directly linked into the project (the project must have also linked additional CUDA dependencies). Each *.cu* file can contain multiple kernels and both host and device methods. Additionally, the same implementation of one method can be executed by both host and device (CPU and GPU) in the case when the method is introduced by ___*host*___ and ___*device*___ directives at the same. This advantage was exploited during the implementation of custom mathematical functions used on both sides. In contrast, OpenCL kernels are more like GLSL shaders, defined in separate *.cl* files which are loaded, parsed, compiled, and linked into OpenCL programs in runtime.

One of the observed benefits of OpenCL is the availability of CG-related mathematical operations, which are accessible, for example, in GLSL. This made the implementation easier, because no custom mathematic functions were needed as opposed to CUDA, where a small helper module was implemented for the needs of the shading kernel.

OpenCL kernel launching is driven by a command queue that accommodates various commands for data manipulation, device synchronization, and launching kernels itself. The queue guarantees that the commands are executed in a given order. Loading of host data into a device kernel is handled separately for each input parameter of that kernel by the `clSetKernelArg` method. It is not required to set the arguments for every kernel invocation, only when the loaded data change. This also reminds of *uniform* variables in GLSL.

For that reason, the struct `CL_Uniforms` was used in a similar way as in CUDA implementation, to encapsulate the majority of shading parameters. In the case of OpenCL, definition of passed structures must be handled with additional care about struct packing (code snippet 7.2), when using

parameters that do not satisfy the byte alignment (e.g. three component vectors).

Unfortunately, pointers to image memory (e.g. `image2d_t`) cannot be stored in a passed structure and must be passed as kernel parameters instead [AMD11]. In the case of using many textures representing the G-buffer together with textures containing BTF data, this leads to the kernel having many input parameters (29).

```
1 typedef struct __attribute__ ((packed)) CL_Uniforms
2 {
3     // some data (possibly non-aligned)...
4 } CL_Uniforms;
```

**Listing 7.2:** Example of OpenCL kernel struct packing

### 7.5.1 OpenCL-OpenGL Interoperation

Sharing device data between OpenGL and OpenCL is available in a way similar to *CUDA-GL interoperation* (chap. 7.4.2). However, compared to CUDA, in the case when OpenCL is used, OpenGL resources must be mapped (acquired) and unmapped (released) in each frame. More specifically, OpenGL resources used in OpenCL must be unmapped from the OpenCL context prior to its use back in OpenGL, and vice versa [khra]. In the case of CUDA, OpenGL resources are mapped and unmapped only on application startup and termination respectively, together with the case when a used material is switched for another.

Acquisition of OpenGL resources is handled by commands stored in a command queue preceding the kernel call command. Following the kernel invocation, OpenGL resources are released by appropriate commands stored in the command queue - the main idea is depicted in code snippet 7.3.

```
1  // i) Acquire OpenGL resources
2  clEnqueueAcquireGLObjects(Queue, TexCnt, &TexRef, 0, NULL, NULL)
3  /* Some more OpenGL resources acquisition */
4
5  // ii) Launch the kernel
6  clEnqueueNDRangeKernel(
7   Queue, KernelObj, DimsCnt, NULL, GlobalWorkSize, LocalWorkSize,
        0, NULL, NULL
8  )
9
10 // iii) Release used OpenGL resources
11 clEnqueueReleaseGLObjects(Queue, TexCnt, &TexRef, 0, NULL, NULL)
12 /* Rest of OpenGL resources release */
```

**Listing 7.3:** OpenGL resources sharing with OpenCL - queue commands pattern

## 7.6 Environment Map Approximation

All environment map related functionality is part of a class `EnvMap`, while the importance sampling implementation was derived from [PJH17]. The topic was briefly discussed in the chapter 4.3.2.

An additional structure `Distribution1D` was implemented that represents the radiance distribution over a 1D image slice, in 4.3.2 referenced as $p(v|u)$ (alternatively as $p_u(u)$ for column integrals). The mentioned struct provides a method to sample the distribution using uniform random variable $\xi_1$ or $\xi_2$ to get transformed variables $u, v$.

These variables can then be used to construct a direction vector $\omega_i$ representing the incoming light $L_i$ from the given sample. In addition, the $u, v$ variables are used directly as texture coordinates to get the color of the spectral radiance of the sample.

An example of samples drawn using uniform random variables $\xi_1, \xi_2$, and variables resulting from importance sampling $u, v$, is depicted in the figure 7.2. It can be seen that the red samples (importance sampling) are cumulated around the area with the highest radiance values (expressed in HDR), probably representing the Sun, whereas green uniform samples are spread all over the image also covering the irrelevant dark spots.

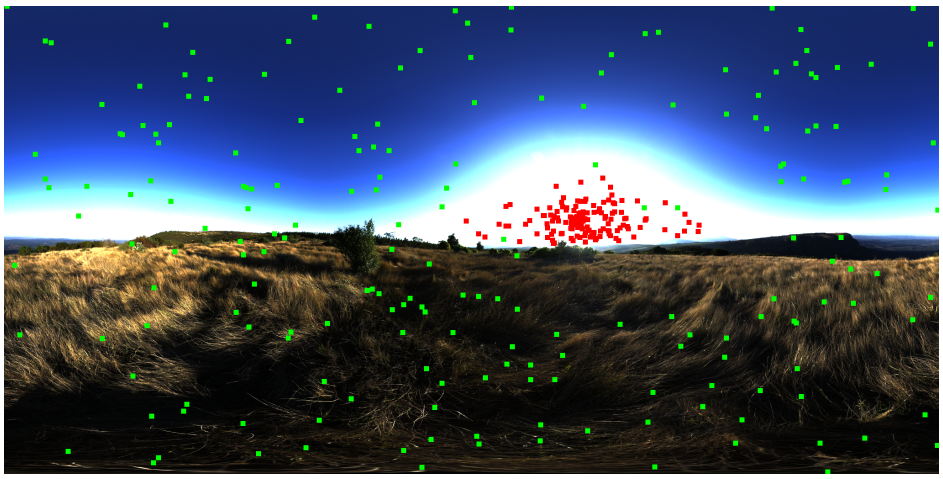


**Figure 7.2:** Visualization of original pseudo-random samples (green) and samples based on importance sampling (red)

The resulting samples are then converted into a set of directional lights represented by a simple struct `DirectionalLight`. Each instance contains the sampled radiance color from the environment map, direction of the light $L_i$, and its sample probability. All these data are computed during the initialization of the application and are sent to the appropriate shader as a set of uniform variables during rendering.

## 7.7 Progressive Rendering

This method requires an additional processing of the computed images. Because all the computed data are stored into an off-screen framebuffer, instead of the main framebuffer directly displayed on the screen because of the needs of VRG API, the main idea was straightforward. Load both framebuffer color textures into some kernel, perform the composition, and display it on screen. The main requirement was the re-usability of the same functionality for all three renderers. Implementation of this functionality separately for each renderer would also be possible but ineffective (against DRY[3] principle).

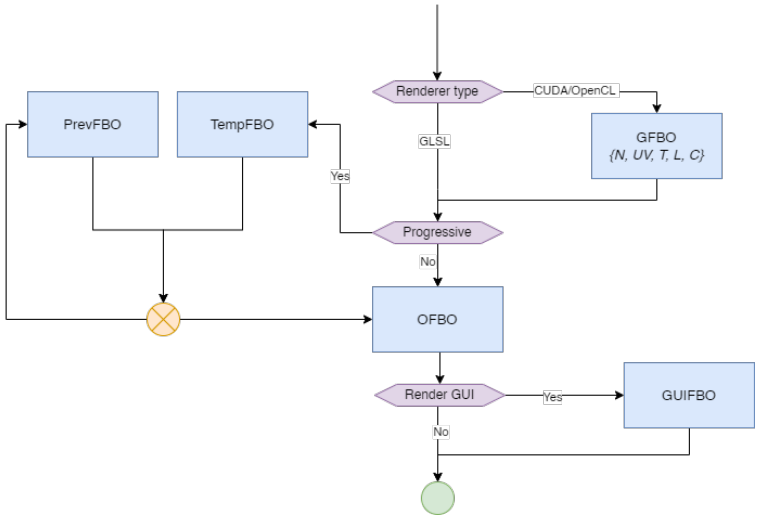



**Figure 7.3:** Diagram of used framebuffer objects

The first approach was about using CUDA kernel and only two framebuffers - the main offscreen FBO and the FBO carrying new computed data for a given subset of lights. This involved using read-write CUDA surface (7.4.3) representing the main offscreen FBO, and a standard CUDA texture representing the second "auxiliary" FBO. Simple kernel was implemented to handle the composition. Unfortunately, this solution suffered from heavy stutters (not FPS drops, FPS seemed to be the same), which made it useless.

Subsequent investigation of the problem was carried out involving reimplementation of the kernel, switching between CUDA surfaces and textures, adding a third texture (to prevent reading and writing to the same texture / surface) and many more. None of the actions helped and the implementation was scrapped. The performance issues were probably caused by multiple re-mapping of the framebuffers between OpenGL and CUDA.

[3]Don't Repeat Yourself

**(a) :** 1 virtual directional light **(b) :** 5 virtual directional lights **(c) :** 200 virtual directional lights

**Figure 7.4:** Comparison of lighting using 1 directional light, 5 directional light, and 200 progressively rendered lights

The second approach involved using OpenGL, which was the right way. Implementation was based on the usage of a simple fragment shader (however, the usage of a compute shader would probably be slightly more effective) and re-used vertex shader for environment map texturing. The idea was to launch fragment shaders for all pixels on the screen, where each FS invocation handled the composition of one pixel. Because the usage of read/write texture probably would not be that stable and would require a special *NV_texture_barrier*[4] extension, the solution with three textures (FBOs) was used instead. Moreover, to save some computation time, `texelFetch` method for texture reading was used. Comparison of progressive rendering using 200 virtual directional lights is depicted in the figure B.1.

## 7.8 XTAL VR Headset Integration

As it was already stated (chapter 6), the core of the application is based on the VRG framework (OpenGL demo application from VRengineers based on their custom rendering framework) which was modified to serve the needed purposes. This ensured from the beginning that the integration with the VR headset will work and there will not be any problems with integrating it later.

### 7.8.1 Rendering to a Headset

To render a stereoscopic image (respectively, a pair of images), an eye-specific transformation must be applied to the scene for each eye, to ensure that the stereoscopic frame displayed in the headset will form a 3D illusion. These transformations represent both projection and view matrices which reflect the current settings (pupillary distance, FOV, etc.) of the headset together

[4] https://stackoverflow.com/questions/11410292/opengl-read-and-write-to-the-same-texture

with the headset position in the virtual scene. The matrices are accessible via the VRG API. The acquisition of the appropriate transform matrices is illustrated in the code snippet 7.4.

```
1  // Get headset (head mounted display) world position
2  const auto hmdPose = m_vrgHmd->GetPose(VRG_POSE_HMD);
3  // VRG framework method to get transl. & rot. matrix from pose
4  const auto hmdTransform = math::PoseToMatrix(hmdPose);
5  // Get eye-specific pose
6  const auto cameraPoseLeft = m_vrgHmd->GetPose(
       VRG_POSE_CAMERA_LEFT);
7  // Compose view matrix based on the hmd pose and eye pose
8  glm::mat4 ViewL = glm::inverse(hmdTransform * math::PoseToMatrix(
       cameraPoseLeft));
9  // VRG framework method to get projection matrix from FOV
10 glm::mat4 ProjL = math::FovToMatrix(fovLeft, NEAR, FAR);
```

**Listing 7.4:** Retrieving left eye camera transform matrices

Standard desktop applications render frames to the default framebuffer to display the data on the screen. In the case of XTAL VR headset, data are passed as a pointer to a texture (attachment) of off-screen framebuffer (more on this in the chapter 6.3.1) using the supplied VRG API. Loading frame data into the headset is controlled by a specific sequence of VRG API commands defining which data belong to the currently processed frame. The sequence is depicted in the code snippet 7.5.

```
1  auto frame = m_vrgHmd->BeginFrame();     // New frame init
2  m_BTFrenderer->renderScene(RIGHT_EYE);
3  GLuint colorR = m_BTFrenderer->getOFBO(RIGHT_EYE)->
       GetColorTexture();
4  frame.SubmitCoordinates(VRG_LAYER_RIGHT, 0, 1, 1, 0);
5  frame.SubmitLayer(VRG_LAYER_RIGHT, &colorR);
6
7  /* Render and send data for the left eye */
8
9  frame.End();     // Send the new frame to the headset
```

**Listing 7.5:** Sequence of rendering new frame and sending it to the headset

## 7.9 Control System

Initial version of the application control system was based on the use of keyboard and mouse inputs. This enabled the mapping of the newly implemented functionality to one of the many available keys without the need to figure out how to map the feature to a combination of buttons on the controller to be used. However, this method is not applicable when the user is wearing a headset, which makes the usage of a keyboard nearly impossible.

[5]The image represents contents of a debug framebuffer displayed on a monitor when a VR version of the application is used. It represents only a side-by-side visualization of framebuffers for both eyes, causing the images in the visualization to be deformed and not match the originals. Indeed, the data for each eye are sent to the headset separately.

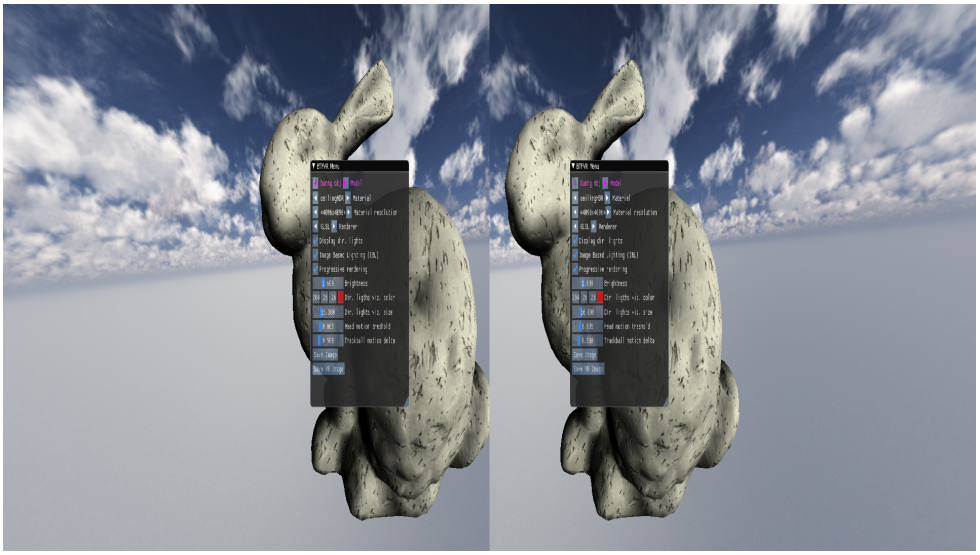


**Figure 7.5:** Example of a stereoscopic image including the rendered menu, which is displayed using the XTAL 8K headset [5].

That said, VR or console specific controller had to be used to make the control over the application easier and more intuitive without the need to see what button is pressed (as opposed to the mentioned keyboard).

Integration of a console controller was performed after all the rendering functionality was implemented to eliminate possible problems with needed functionality re-mapping. Due to the vast amount of editable parameters, usage of the controller alone was considered as insufficient, and an accompanying graphical user interface was added.

All implementation related to the control system and functionality for changing the application state is defined in the files *ControlsApi.h* and *ControlsApi.cpp*. Application state is represented by the structure `Globals` which is accessible within the `BTFRenderer` class and which is also passed to the controls API methods (chap. 7.9.2).

### 7.9.1 Wiimote Controller

The Wiimote controller, depicted in the figure 7.6, was implemented as the first interaction device used to interact with the VR application (and in the end the only controller implemented). Among its advantages can be considered a relatively low price while having a decent amount of functionality proven by many years of its presence on the gaming consoles scene.

Communication with the controller was implemented using the `Wiiuse` library, which simplifies handling of the controller events and setting its state (e.g., turning on LEDs or rumbling). The controller was integrated into the application using the API discussed in the following chapter.

[6] `https://www.walmart.ca/en/ip/Wii-Remote-Controller-Nunchuk-Plus-Motion-Nunchuck-Control-Joystick-Game-silicone-cover-wrist-strap-u/4RT6ZRZT1QXH`

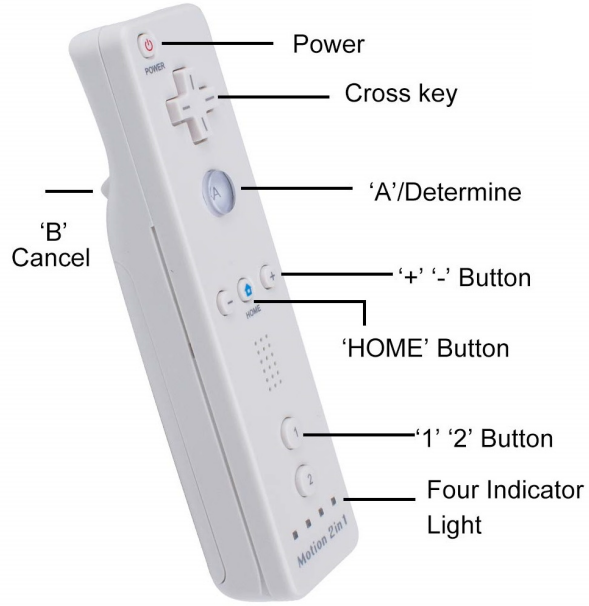



**Figure 7.6:** Wii remote controller[6]

Final version of the button binding to the application functionality is depicted in the diagram A.1.

### 7.9.2 Generic Controls API

Due to the possibility of using different input devices in the future, a generic control API was implemented to enable the integration of various types of controllers, each of them possibly having a different behavior, without the need to know the implementation details of the renderer itself. Instead, state of the renderer can be controlled using a set of static functions which names and parameters indicate their functionality.

### 7.9.3 Graphical User Interface

Graphical user interface was implemented using the `ImGui` library which provides a simple approach to creating and rendering custom GUI elements. Content of the GUI is handled by the library, including the management of geometry and its rendering. In the case of displaying the GUI over an image resulting from progressive rendering, an additional framebuffer `GUIFBO` was used, as depicted in the diagram 7.3, because the original framebuffer containing the rendered data is subsequently used in the next progressive rendering iteration. An example of a rendered menu can be seen in the figure 7.7.

Rendering the GUI in VR is a bit more challenging. The main problem is the need to render the GUI stereoscopically correct, which means perspectively transformed for each eye by its projection matrix.

However, that was not achievable using only the ImGui library, because the library is designed for 2D rendering only. The final solution is thus based on an additional offscreen rendering. The GUI is rendered into an auxiliary FBO texture (`UIFBO`), which is subsequently mapped onto a perspectively transformed quad defined in view-space coordinates to cover the desired view area. The result can be seen in the figure 7.5.


**Figure 7.7:** Application menu implemented using ImGui, displayed on top of the rendered image

### 7.9.4 Application Configuration

To enable the application to be configured without the need to recompile the project, support for a configuration file was added. The name of the file is fully customizable (together with its extension) as long as the corresponding file name is supplied as a program launch parameter. The current name of the configuration file is $config.env$.

The configuration file can contain two types of records (lines). Commented lines starting with # are not parsed and can serve as a source of information. Each non-commented line can represent one of the application attributes and is defined by $key = value$ pair. Supported data types are $String$, $Float$, $Integer$, $Boolean$ and $Vector$. Vectors are treated as having three float components, however, integer values can be used in the configuration file too. The vector values are separated by a dash $(-)$. The boolean attributes can have values $\{y, true, 1\}$ that are evaluated as $True$, other values are evaluated as $False$. An example of supported syntax is depicted in a snippet 7.6.

The implementation of the environment (application configuration) is represented by the `Environment` class, which instance is subsequently used to populate global variables on the application startup or restart. The class provides methods (based on the read data type) to retrieve the desired attribute using the same key as is used in the configuration file. When an error occurs during parsing the configuration file, default values are used instead. The same applies when the configuration file was not found.

```
1  # Line commented out
2  key_vector=100.0-200.0-155.0
3  key_string=string
4  key_float=245.12
5  key_bool=true
```

**Listing 7.6:** Configuration file syntax.

## 7.10 Summary of Used Technologies

### STBimage 2.22

Library used for image loading in the texture creation process and saving of the rendered images in a file.
`https://github.com/nothings/stb`

### Dear ImGui 1.76

Library used to create a graphical interface rendered on top of the rendered image.
`https://github.com/ocornut/imgui`

### Wiiuse 0.15.5

Simple library implementing API for communication with Nintendo Wii controllers, used for the integration of the Wiimote controller.
`https://github.com/wiiuse/wiiuse`

### GLM 4.2

OpenGL math library used mainly for transformation matrices and their multiplication.
`https://github.com/g-truc/glm`

### VRG (hmd lib) 4.3

API used for communication with the VR headset. The API is accessible only to headset owners.

### VRG rendering framework

Set of classes forming an OpenGL rendering engine/framework provided by VRengineers as a part of the OpenGL VR demo. Subset of the classes was used as a base for the application implemented. All classes are part of the `vrg` directory and are accessible via the main `gldemo::` namespace.

### CUDA 11.4.136

GPGPU framework for mass-parallel computing on GPUs, used for the implementation of one of the shading kernels.

### OpenCL 3.0

GPGPU framework for mass-parallel computing on GPUs and CPUs, used for implementation of one of the shading kernels.

### OpenGL 4.6.0

Graphics API used for rasterization, rendering and displaying the rendered content.

### GLFW 3.3

Windowing API used for creating OpenGL rendering context and handling input events
`https://github.com/glfw/glfw`

### Dolphin 4.0.8325

Software used for emulation of various controllers on PC including the Wii controller used.
`https://github.com/dolphin-emu/dolphin`

# Chapter 8

# Results and Testing

Chapter deals with the implemented application verification, consisting of two parts, the *User Testing* and the *Performance Testing*. First, the principles of user testing are briefly described, followed by the description of the test strategy that includes the choice of users tested and the test scenario. Lastly, the results of the user testing are shown together with some advice given by the tested users.

In the second part, the performance testing strategy is discussed together with the hardware used. Subsequently, the results of the performance tests are discussed along with possible improvements that could be implemented to enhance the overall performance.

## 8.1 User Testing

User testing (also usability testing) is an essential part of software development, which is crucial for any kind of applications having an *User Interface* (UI). Every application should be subject to multiple user testing iterations to guarantee that the application will be used as intended and no confusion is present.

One of the examples of user testing practices is described in a book *Don't make me think* [Kru06] written by Steve Krug, which is devoted to human-computer interaction principles. The whole user interface testing topic is described in a chapter *9: Usability testing on 10 cents a day* accompanied by a fictional test scenario serving as an illustration of how to efficiently test the user interface of an application.

One of the main principles of proper usability verification of the UI is the iterative testing performed already during the application development. In such a way, eventual problems are discovered relatively early, and it is easy to fix them. Ideally, the first tests should be carried out even before the development starts and in its early phases. This type of tests is also known as *Group testing* and consists mainly of brainstorming about the UI in a group of various people, such as developers, testers, future users, etc, each of them having a different point of view based on their experiences. By this approach, many ideas are presented and the best ones are chosen for subsequent development.

| Tester | Sex | Age | Experience with VR | Occupancy |
|---|---|---|---|---|
| Tester A | Male | 52 | No | IT businessman |
| Tester B | Female | 21 | No | Student of Architecture |
| Tester C | Male | 24 | Yes | Student of Medicine |
| Tester D | Male | 24 | No | IT freelancer |
| Tester E | Male | 25 | Yes | Student of CTU FEE (HCI) |

**Table 8.1:** Information about tested users.

When the application is near completion, *Usability tests* are performed. These tests are based on the behavior examination of given individuals, therefore each test is performed with only one user at a time. Users tested are told to solve a given set of tasks within the application, and the test conductors record possible problems with the usability of the application. These results form a valid scale of the usability of the application and are used for subsequent improvements of the application.

### 8.1.1 Testing Strategy

Because the UI was implemented in the last weeks of the application development while under time pressure, group tests were omitted. Therefore, the user tests were focused only on the final *Usability* part performed with a group of users described in the table 8.1. A set of interview questions was prepared that served as a baseline for the testing. Each user was asked an initial subset of questions (1.X in 8.1.2) investigating their experience with VR together with their expectations of the application tested based on a brief introduction to the topic of BTF rendering.

The Wiimote controller and the XTAL VR headset were then introduced, and some time was dedicated to free exploration of the VR application by the user. Based on the user experience, a few more questions (2.X in 8.1.2) were asked to capture the first impressions about the control system, GUI, and the application usability.

The desktop version of the application was subsequently launched, and the user was given an additional time for another free exploration. Lastly, questions about the desktop version were asked and the user was given the opportunity to give constructive feedback, such as suggestions for possible improvements, by the last subset of questions 3.X and 4.X in 8.1.2.

### 8.1.2 Questions and General Answers

This chapter is devoted to the questions asked during the usability testing together with the answers that capture the general opinion of the users tested.

#### 1.1) Do you have any previous experience with Virtual Reality? If so, which types of controllers have you tried?

Three of the tested users had no prior experience with virtual reality. Tester C had experienced virtual reality as a form of educational tool in medicine dedicated to a virtual human body examination. Tester E had an experience with a VR headset `Oculus Quest I` in terms of exhibition in `Institute of Intermedia (IIM)`[1], and also in terms of entertainment represented by a VR game `BeatSaber`[2]. Additionally, tester E had previous experience with the Wiimote controller (chap: 7.9.1).

#### 1.2) What do you expect from a VR application in general? What functionality and behavior should it implement?

Most testers mentioned an artificially generated environment within which the user can move and interact with. The application should provide a decent degree of immersion in the virtual environment.

#### 1.3) Based on the information given about the work and application implemented, what are your expectations in terms of functionality and features?

Testers mentioned that the application should allow the user to freely change parameters of the scene, including the camera, object, and lights motion, the material used and its resolution, alternatively to change the light intensity. Based on the interactions performed, an arbitrary 3D model with arbitrary material can be observed under various lighting and view conditions.

#### 2.1) What impressed you and what confused you about the application?

Impressions of individual testers differed. Some of them were impressed by the rendered environment, whereas some of them liked the most the visual appearance of the rendered objects and materials used. Additionally, a progressive rendering seemed interesting to the testers.

Most of the confusion shared between the testers was about the control system based on many possible interactions, which was difficult to recall for the first time.

[1] `https://www.iim.cz/en/about-us/`
[2] `https://beatsaber.com/`

Some testers also noted that the Wiimote controller does not have a good button layout, mainly the bottom buttons seemed to be too far. One tester also mentioned a blurred GUI. However, the overall impression of the application was positive.

### 2.2) Do you prefer an interaction using a GUI over an interaction using the controller alone?

The general opinion was that both variants of the interaction should be used. GUI menu can support new users who do not have sufficient experience with the designed control system, mainly with the various shortcuts or button combinations (combos). On the contrary, more experienced users may see shortcuts as a better and faster type of interaction than using the menu.

### 2.3) Do you find interaction using the Wiimote controller intuitive enough? (Both in terms of mapping to application functionality and in terms of physical attributes of the controller)

Functionality binding to the controller seemed intuitive for all the users tested, except for the mentioned button combos, which were harder to remember. However, the opinion about the Wiimote controller itself differed.

Tester A was for a reduction of the buttons used in one controller. In particular, the individual buttons `1` and `2` at the bottom of the controller were considered the most problematic. Additional functionality should instead be delegated to an eventual second controller designed to be held in the second hand (alternatively, a gamepad-style controller held in both hands could be used). The main point was that only one hand is currently being occupied, while the second one is not used at all. The tester A also lacked a usage of accelerometers that could extend the list of mapped functionality without the need to use more buttons.

Testers B, C, and E were quite satisfied with the controller except for the confusion with button combinations already mentioned. However, this could be resolved with an additional help window and brief information in a `Readme` file. Tester E also suggested some ideas on redesigning the combo button layout (more on this in the chapter 8.1.3.).

Tester D considered the behavior of the Wiimote controller arrow buttons (cross) strange and disruptive, mainly when interacting with the menu. The problem was in the need of pressing the buttons exactly in the desired direction, otherwise the controller also performed interaction in the perpendicular direction. In this way, unwanted interactions were sometimes performed, which in the case of menu was unpleasant.

### 3.1) Which type of application did you find more attractive? Why?

All tested users stated that the VR version of the application was more exciting, mainly because it is much more immersive than just looking at a monitor. The observed object also seems more plastic, and the details of the

material are more noticeable when VR is used. Additionally, tester D noted that when using a VR headset, the user can fully focus on the scene without being disrupted by the environment, as can be the case when looking at a monitor on a desk.

#### 3.2) Which type of application was easier to interact with?

Users A, B, and C found the desktop controls to be more intuitive, mainly because of their daily experience with a keyboard and a mouse. In contrast, testers D and E found the interaction using a controller to be better, despite the downsides mentioned.

#### 4.1) Did the application meet your pre-launch expectations?

All tested users were satisfied with the application, the behavior was similar or better than expected. Some testers would welcome the possibility to change the environment map used in a run-time. Testers A, B, C, and E would also like to be able to choose from a wider range of different types of 3D objects.

#### 4.2) Which part of the application / control system do you consider as the worst?

Tester A considered the worst hardly reachable buttons `1` and `2` on the Wiimote controller. The mentioned buttons should be used only in rare situations. Tester B was not satisfied with the "overcomplicated" control system. Tester C considered the blurry GUI mentioned as the worst. Tester D was not satisfied with the menu interaction using the Wiimote controller due to the mentioned problem with arrow keys (mentioned in the answer 2.3). Tester E found the rotation of the environment map confusing and non-intuitive.

#### 4.3) What would you improve in the application?

The answers are discussed in the following chapter.

### 8.1.3 User Advice

The main problem of the application, reported by the testers, seemed to be the control system, particularly the mapping of advanced functionality to the Wiimote controller. However, the ideas, how to redesign the button combinations used for the advanced functionality mentioned, differed from user to user, meaning that the opinion is relatively subjective and may not be welcomed by others.

The most significant and interesting changes proposed testers A and E. The main idea of the tester A was to distribute the control over the application evenly among both hands (similarly as a keyboard and mouse are also used by both hands). That could be achieved either by introducing a second controller of preferably the same type, or by using a game pad dedicated for both

hands. On the contrary, tester E stated that one controller is enough and proposed the usage of buttons `1` and `2` instead. These buttons were supposed to be pressed (held) by the second hand, while the main hand performs the secondary (advanced) interactions.

Nevertheless, all tested users agreed that some type of information about the control system should be available, at least in the form of a *Readme* file.

The tester A also lacked the usage of an accelerometer which is a part of the Wiimote controller and which could be used for rotation interactions. However, tester E, who already had experience with the Wiimote controller, stated that the accelerometer of the particular controller is not precise and therefore is useless.

One of the minor ideas proposed by the tester E was about the possibility to add a variable option to invert the *Camera zoom* interaction mapped to the buttons `A` and `B` (which is also considered as subjective).

Majority of the tested users would also welcome the possibility to change an environment map in a run-time.

## 8.2 Performance Testing

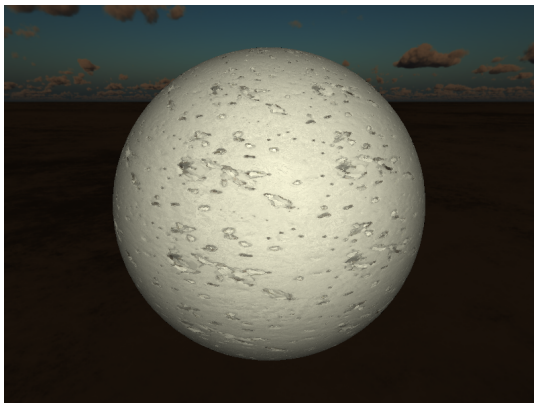
Various performance tests were carried out to verify the overall performance of the application. The tests were devoted to the comparison of individual technologies used for the rendering (GLSL, CUDA, and OpenCL), the comparison of desktop and VR versions of the application, the comparison of various types of lighting used, and also the comparison of the new GPU implementations with the initial CPU implementation.

### 8.2.1 Tests Setup

A specific test environment was prepared beforehand, consisting of manually set scene parameters to ensure that each test was performed under the same conditions, and to represent a form of the worst-case scenario. The tests were performed in such a way that the geometry covered a large part of the screen (to increase the number of pixels where BTF must be evaluated) together with all lights directed to the visible part of the objects surface so that all computed fragments were enlightened. This approach can be considered as the worst-case scenario[3], when all available light sources contribute to the final color, and therefore BTF decompression must be evaluated for each fragment as many times as is the count of available lights (except for few edge cases).

Because of that, an environment map `env020.hdr` ("*Desert*"), that has the majority of radiance present in one small area, was used to concentrate all approximation directional lights into one spot (see fig. 8.1b).

[3]Worst case scenario for the average observation of the object in a reasonable scale. Complete worst-case scenario would be represented by a close-up detail covering the whole screen, however, that is not expected to be performed by users that frequently.



**(a) :** Rendered view during the performance tests

**(b) :** Virtual directional lights sampled behind the camera (purple dots) from the approximated environment map.

**Figure 8.1:** Tested scene setup, consisting of a manually selected view direction, camera zoom, and environment map together with its rotation.

| Name | Alias | #Vertices | #Faces |
|---|---|---|---|
| koule.obj | Sphere | 3122 | 6240 |
| bunny.obj | Bunny | 5063 | 10122 |
| hippo.obj | Hippo | 1810 | 3616 |

**Table 8.2:** Information about 3D models used for performance tests.

All test results were acquired as an average of three measurements over a 5-second period. Performance was mainly measured in *Frames Per Second* (FPS), representing the overall application performance, recognized by the majority of computer users and which is easily understandable.

Additionally, the render time for all 200 virtual directional lights was measured in case of progressive rendering. That represents the total time needed to approximate the environment map IBL by 200 directional lights while maintaining the application interactive, having preferably 60 FPS or more.

In terms of GPGPU computations (CUDA and OpenCL), the block size was set to 16x16 threads in a 2D grid configuration, making 256 threads per block.

Rendering to the XTAL 8K VR headset was exploited, whenever it made sense, to demonstrate the performance that can be expected from the application when using a high-resolution VR headset.

Tests were performed for 5 various BTF materials in terms of their size and type (HDR, LDR). In this way, all possible kinds of problems could be found. Used materials together with their description are depicted in the figure 8.2. Information about the used 3D models is available in the table 8.2.

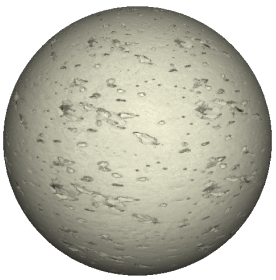
**(a) : walkwayHDR**
42.4MB



**(b) : pulli**
30.95MB



**(c) : corduroy**
17.6MB



**(d) : ceilingHDR**
11.75MB

**(e) : pinktileHDR**
5.6MB

**Figure 8.2:** BTF materials used during the performance testing. All materials were used in a resolution of 2048x2048 pixels.

### 8.2.2 Used Hardware and Software

All measurements were acquired using the following hardware provided by CTU FEE in VR laboratory.

- CPU: Intel i9-10900X, 10 cores / 20 threads, @3.7GHz
- GPU: RTX 2080Ti, 11GB VRAM
- RAM: 128 GB DDR4
- Disk: Seagate Capacity 6TB, 7200 rpm
- VR Engineers XTAL 8K headset
- OS: Windows 10 Education Edition

### 8.2.3 GLSL, CUDA, and OpenCL Performance Comparison

First group of tests was dedicated to the comparison of the used GPU technologies, by means of overall performance in FPS, while rendering to the XTAL headset. It can be seen that both CUDA and OpenCL struggle with the HDR materials, however, in the case of `corduroy` material, the performance of CUDA compared to GLSL is similar or even better.

The lower performance of rendering using GPGPU technologies (table 8.3) can be caused by many factors. One of the problems can be the fact that the GLSL rendering is handled as a single-pass process, whereas CUDA and OpenCL need two passes for the implemented deferred shading. Additional work is also needed for OpenGL textures and renderbuffers sharing and for the needed FBO content copy operations (atleast two copy operations, when progressive rendering and GUI are disabled - see fig. 7.3).

OpenCL specific problem is also the need to explicitly acquire and release OpenGL resources in every rendered frame (chap. 7.5.1), which was found as a major bottleneck.

| **BTF Material** *2048x2048 px* | Render time [*Frames Per Second* (FPS)] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Sphere** | | | **Bunny** | | | **Hippo** | | |
| | *GLSL* | *CUDA* | *OpenCL* | *GLSL* | *CUDA* | *OpenCL* | *GLSL* | *CUDA* | *OpenCL* |
| walkwayHDR | 79.3 | 34.3 | 18.7 | 81.9 | 38.9 | 20.9 | 77.7 | 36.9 | 19.2 |
| pulli | 119.1 | 55.0 | 23.9 | 116.6 | 60.0 | 24.5 | 115.4 | 57.9 | 23.5 |
| corduroy | 55.1 | 51.5 | 23.1 | 52.3 | 55.0 | 23.3 | 47.2 | 54.1 | 22.6 |
| ceilingHDR | 91.5 | 31.1 | 19.4 | 92.4 | 37.9 | 20.8 | 87.3 | 36.5 | 19.0 |
| pinktileHDR | 147.1 | 34.8 | 19.7 | 135.3 | 38.2 | 21.5 | 136.2 | 36.7 | 19.1 |

**Table 8.3:** GLSL, CUDA and OpenCL performance comparison in *Frames Per Second* (FPS), in 2x 4K resolution used in XTAL headset and 5 directional lights environment map approximation.

### 8.2.4 Virtual Reality and Desktop Performance Comparison

The following set of performance tests focused on a comparison of rendering to a VR headset with rendering to a standard 4K display. Measurements were performed only with the best performing technology, which is considered GLSL (given by the results in the table 8.3). Even though the VR headset has exactly double the resolution of the 4K display (having one 4K display per each eye), the results in the table 8.4 do not match that.

That is caused by the way the VR headset uses the rendered framebuffer and how the per-eye transformation matrices are defined, making the peripheral area of the stereoscopic image bigger. Based on that, the objects are rendered from a greater distance than in the case of rendering to a monitor, leading to the object covering a smaller area on the screen and the need of less fragment shaders to be evaluated.

| **BTF Material** *2048x2048 px* | Render time [*Frames Per Second* (FPS)] | | | | | |
|---|---|---|---|---|---|---|
| | **VR XTAL 2x 4K** | | | **Desktop 4K** | | |
| | *Sphere* | *Bunny* | *Hippo* | *Sphere* | *Bunny* | *Hippo* |
| walkwayHDR | 79.6 | 82.0 | 77.7 | 131.7 | 147.3 | 135.9 |
| pulli | 119.1 | 116.6 | 115.4 | 208.2 | 223.6 | 217.1 |
| corduroy | 55.1 | 52.3 | 47.2 | 91.0 | 96.2 | 86.0 |
| ceilingHDR | 91.5 | 92.4 | 87.3 | 156.3 | 173.4 | 160.9 |
| pinktileHDR | 147.1 | 135.3 | 136.2 | 253.2 | 250.8 | 246.3 |

**Table 8.4:** VR and desktop renderers performance comparison in 2x 4K resolution used in XTAL headset and 4K resolution of the monitor used, and 5 directional lights environment map approximation.

### 8.2.5 Various Types of Lighting Performance Comparison

Third group of tests was devoted to performance behavior when various types of lighting are used. The tests were performed only for GLSL and CUDA renderers using a `Bunny` 3D model. Namely, the performance of illumination using one point light was compared to the illumination using 5 virtual directional lights approximating the environment map IBL, together with the progressive scene illumination using 200 of the mentioned directional light sources.

Despite the fact that the progressive rendering using 200 directional light sources is a version of a *5 Directional lights* rendering extended into more frames, it performed worse in terms of FPS (table 8.5). The main cause of the lower performance is probably an additional overhead represented by image data transfers between auxiliary FBOs, and by the additional shader used for composing the temporary frames (based on the FBO diagram depicted in the figure 7.3).

However, the frame drops under 60 FPS can be compensated by lowering the number of light sources processed in a single frame, adequately increasing the total time needed for computing IBL using all 200 lights.

| **BTF Material** *2048x2048 px* | Render time [*Frames Per Second* (FPS)] | | | | | |
|---|---|---|---|---|---|---|
| | **1 Point light** | | **5 Dir. lights** | | **200 Dir. lights (PR)** | |
| | *GLSL* | *CUDA* | *GLSL* | *CUDA* | *GLSL* | *CUDA* |
| walkwayHDR | 341.6 | 113.8 | 82.0 | 38.9 | 61.9 | 34.4 |
| pulli | 450.7 | 150.2 | 116.7 | 60.0 | 92.3 | 54.9 |
| corduroy | 239.5 | 141.1 | 52.3 | 55.0 | 45.0 | 50.3 |
| ceilingHDR | 373.2 | 114.5 | 92.4 | 37.9 | 73.2 | 35.1 |
| pinktileHDR | 482.6 | 116.1 | 135.3 | 38.2 | 93.9 | 35.5 |

**Table 8.5:** Performance comparison for a single point light, together with 5 directional lights and 200 directional lights approximating the environment map used. 200 directional lights are rendered progressively, 5 lights in each frame, to remain the application interactive. Tested in VR (2x 4K ressolution) for bunny 3D model.

### 8.2.6 CPU and GPU Performance Comparison

The last subset of tests was dedicated to the comparison of newly implemented GPU solutions with CPU based solutions implemented initially. The initial implementation consists of a full CPU offline renderer[A] (both rasterization and shading are handled by a single-thread CPU program), a CPU & CUDA combined offline renderer[B] (CPU rasterization, CUDA shading and BTF evaluation), and an experimental real time CPU & CUDA & OpenGL renderer[C], having the same functionality as the previously mentioned one, except the content rendered is displayed on screen using OpenGL, making the renderer real-time and interactive.

Newly implemented renderers are GLSL[D], CUDA[E], and OpenCL[F], which are subject to this work.

In contrast to the majority of previous performance measurements, these measurements were performed at a desktop resolution of only 800x600 pixels as is the original resolution of the CPU renderers. Performance tests were also carried out only for the `Sphere` 3D object.

In addition, the timings for renderers A and B were measured in milliseconds, subsequently recomputed to FPS.

Based on the results in the table 8.6 it can be seen that the renderer B performs significantly worse that the renderer C, even though both renderers are nearly identical. The reason why renderer B (and also renderer A) performs worse than the similar renderer C is the fact that the measured time also takes into account the time needed for saving the rendered image into a file. However, in the case of the renderer C, the image data remain on the GPU (OpenGL) instead, to be subsequently displayed on a screen.

| **BTF Material** *2048x2048 px* | Render time [*Frames Per Second* (FPS)] | | | | | |
|---|---|---|---|---|---|---|
| | ***CPU***[A] | ***CPU & CUDA***[B] | ***CPU & CUDA & OpenGL***[C] | ***GLSL***[D] | ***CUDA***[E] | ***OpenCL***[F] |
| walkwayHDR | 1.8 | 9.8 | 16.4 | 2177.8 | 1047.7 | 267.0 |
| pulli | 2.2 | 10.5 | 16.5 | 2840.0 | 1307.7 | 291.3 |
| corduroy | 2.0 | 9.1 | 17.6 | 1622.0 | 1197.3 | 272.7 |
| ceilingHDR | 1.8 | 9.3 | 17.4 | 2411.0 | 1064.3 | 281.0 |
| pinktileHDR | 1.9 | 9.5 | 17.4 | 3961.3 | 1109.3 | 297.5 |

**Table 8.6:** CPU and GPU performance comparison. Tested on a desktop with a resolution of 800x600 pixels. `CPU` - rasterization and shading handled by CPU; `CPU&CUDA` - rasterization handled by CPU, shading handled by CUDA kernel; `CPU&CUDA&OpenGL` - same as CPU&CUDA only the rendered content is displayed on a screen using OpenGL (no saving to a file); `GLSL`, `CUDA`, and `OpenCL` - newly implemented renderers.

### 8.2.7 Unresolved Problems and Possible Solutions

This chapter is dedicated to the discussion on previously known and newly discovered performance problems and their possible solutions.

#### OpenCL-OpenGL Resources Sharing

One of the major previously known performance problems is related only to the OpenCL renderer. It is caused by the need to acquire and release all OpenGL resources in each render-loop iteration (already mentioned in the chapter 7.5.1), while the (un-)mapping process acts as a costly operation. The problem was encountered already during the development and after some research it was considered as a GPU driver-related problem, as stated on the `StackOverflow` forum [4].

The problem is more noticable, due to a large number of OpenGL textures being mapped one by one into the OpenCL context (cubemap faces, BTF data textures, deferred rendering G-buffer textures, output framebuffer texture, and environment map texture).

Possible solution minimizing the performance issues with the (un-)mapping of OpenGL resources is the use of `GL_TEXTURE_2D_ARRAY`[5] or `GL_TEXTURE_3D`[6] texture formats, where only one reference per the whole stack needs to be passed. Both mentioned texture types are supported by OpenCL[7] and also by CUDA[8].

#### GPU Explicit Synchronization

The overall performance can be also limited by the currently used explicit GPU synchronization (e.g. `glFinish`), needed mainly when uploading the rendered image into the XTAL VR headset. Without the explicit synchronization, various artifacts and image flickering were observed in the headset.

#### Types of GPGPU Memory Used

As already stated, the performance of GPGPU calculations can be affected by the OpenGL resources sharing. However, another type of performance issues can be caused by an inappropriate type of memory used. For example, when using a lot of texture memory, the performance can be lower[9] due to a cache utilization, than when using global or local memory instead.

[4] `https://stackoverflow.com/questions/17899514/opencl-opengl-interop-performance`

[5] `https://www.khronos.org/opengl/wiki/Array_Texture`

[6] `https://www.khronos.org/opengl/wiki/3D_Texture`

[7] `https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/clCreateFromGLTexture.html`

[8] `https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OPENGL.html#group__CUDART__OPENGL_1g80d12187ae7590807c7676697d9fe03d`

[9] The lower performance can relate to only some subset of graphics cards, mainly the newer ones.

### Low FPS in VR

In some cases the VR application can suffer from low FPS, in rare cases reaching bottoms of 30 FPS. However, as observed in the XTAL 8K VR headset, the image appears smooth even when facing performance issues. This is probably feature of the headset, when some image warping and interpolation is very likely performed.

### Multiple Threads

The whole application is currently implemented as single-threaded. The overall performance should not be affected, because no extra per-frame operations are computed, except for a few transformation matrices and occasional uniform variables binding. However, two problems were experienced during the develpoment and testing.

The first problem is bound to the variable time needed for initialization of the Wiimote controller, which can reach up to 10 seconds. When compared to the rest of the application initialization time, the total initialization time can be more than doubled, which may be annoying.

The second problem is bound to polling the events from the Wiimote controller, which is limited to approximately 60-65Hz, limiting the FPS to the mentioned frequency.

Both problems mentioned could be resolved by moving the Wiimote-specific implementation into another thread.

# Chapter 9
# Conclusion

Chapter is devoted to a summary of contributions to the work and to a subsequent proposal for future work.

## 9.1 Summary

After a brief research of available GPU technologies and the study of the global illumination principles together with BTF data compression algorithm presented in [HFM10], a robust solution was proposed to enable high-resolution image synthesis of 3D objects with compressed BTF materials delivering sufficient frame rates even when a *Virtual Reality* headset `XTAL 8K` is used, all together implemented using three distinct GPU technologies. Various types of lighting were also incorporated to enhance the final visual quality. Additionally, a decent user interface was introduced to enable a high degree of control over the application, especially when VR headset is used. Lastly, an adequate usability and performance testing was carried out to evaluate the overall quality of the application implemented, which delivered relatively satisfactory results.

Based on the work assignment requiring to exploit the usage of various GPU technologies, three distinct renderers handling the compressed BTF rendering were implemented. The first one represents a standard single-pass renderer implemented using only a modern graphics API OpenGL 4.6. The second renderer represents a two-pass deferred shading approach split between OpenGL used for rasterization, and CUDA GPGPU technology used for shading and the BTF evaluation. In a similar manner, OpenCL renderer was implemented, delivering a slower but multi-platform alternative to the CUDA mentioned.

Subsequently, an approximation method of *Image Based Lighting* (IBL) using an environment map was implemented based on [PJH17] solution. The method was extended by a progressive rendering approach enabling to evaluate many more light sources than it is possible in a single frame while maintaining attractive FPS.

A special care was devoted to the design of *User Interface* (UI), which is crucial for the usability of the application implemented, mainly in terms of VR.

Following, user testing and performance testing were carried out to evaluate the overall quality of the application implemented. The results of user testing have shown that the UI is designed well, only a few changes were proposed by the testers in terms of mapping of advanced functionality to the Wiimote controller currently used.

The performance testing results shown that the best performing GPU technology is GLSL. However, these results may be distorted by the specific requirement where all three renderers must be available at the same time to enable real-time comparison, leading to the need of sharing the majority of GPU resources used among the renderers, which can cause performance bottlenecks

## 9.2 Future Work

Although the current implementation required a lot of time and effort, there are still topics that are not part of the final version of the application. One of them is the OptiX renderer which research and implementation was held for nearly two weeks. Unfortunately, because of time pressure, this part of the implementation was abandoned and left as a possibility for future extensions of the application.

Current version of the application officially supports rendering of only one 3D model at a time, however, more complex scenes could be rendered consisting of more objects together with various materials. Also, a modern-day geometry loader like `Assimp` could be integrated and used instead of the old Wavefront .obj parser used currently. Related to that, also a decent scene-graph should be incorporated to support more complex scenes. It is already part of the VRG framework, but minor implementation changes are needed to make the scene graph usable.

The implementation provides a generic controls API enabling relatively easy integration of new control devices into the application. Currently, keyboard & mouse controls, together with Wiimote controller are implemented, however additional control devices could be added, such as `HTC Vive` proposed by some testers during the usability testing.

Last but not least, the CUDA and OpenCL implementation should be revised to find possible bottlenecks and errors caused by the insufficient experience with the technology used during the implementation.

# Bibliography


[Nic65] Fred E. Nicodemus. "Directional Reflectance and Emissivity of an Opaque Surface". In: *Appl. Opt.* 4.7 (July 1965), pp. 767–775. DOI: 10.1364/AO.4.000767. URL: http://opg.optica.org/ao/abstract.cfm?URI=ao-4-7-767.

[Kaj86] James T. Kajiya. "The Rendering Equation". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902.

[Dee+88] Michael Deering et al. "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics". In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, pp. 21–30. ISBN: 0897912756. DOI: 10.1145/54852.378468. URL: https://doi.org/10.1145/54852.378468.

[Guo98] Baining Guo. "Progressive Radiance Evaluation Using Directional Coherence Maps". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 255–266. ISBN: 0897919998. DOI: 10.1145/280814.280888. URL: https://doi.org/10.1145/280814.280888.

[Dan+99] Kristin J. Dana et al. "Reflectance and Texture of Real-World Surfaces". In: *ACM Trans. Graph.* 18.1 (Jan. 1999), pp. 1–34. ISSN: 0730-0301. DOI: 10.1145/300776.300778. URL: https://doi.org/10.1145/300776.300778.

[Žár+05] Jiří Žára et al. *Moderní počítačová grafika*. cze. 2. Praha: Computer Press, 2005. ISBN: 80-251-0454-0.

[Kru06] Steve Krug. "Don't Make Me Think". Czech. In: trans. by Jan Škvařil. 2nd. Computer Press, Brno CZ, 2006, pp. 113–135. ISBN: 80-251-1291-8.

[FH09] Jiři Filip and Michal Haindl. "Bidirectional Texture Function Modeling: A State of the Art Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.11 (2009), pp. 1921–1940. DOI: 10.1109/TPAMI.2008.246. URL: http://library.utia.cas.cz/separaty/2009/RO/filip-bidirectional%20texture%20function%20modeling%20state%20of%20the%20art%20survey.pdf.

[NV10] *DirectX11 Compute Shaders Vs CUDA*. [Accessed 27-December-2021]. 2010. URL: https://forums.developer.nvidia.com/t/directx11-%20compute-shaders-vs-cuda/16127.

[HFM10] V. Havran, J. Filip, and K. Myszkowski. "Bidirectional Texture Function Compression Based on Multi-Level Vector Quantization". In: *Computer Graphics Forum* 29.1 (2010), pp. 175–190. DOI: https://doi.org/10.1111/j.1467-8659.2009.01585.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01585.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01585.x.

[KDH10] Kamran Karimi, Neil G. Dickson, and Firas Hamze. *A Performance Comparison of CUDA and OpenCL*. [Accessed 27-December-2021]. 2010. DOI: 10.48550/ARXIV.1005.2581. URL: https://arxiv.org/vc/arxiv/papers/1005/1005.2581v1.pdf.

[Par+10] Steven G. Parker et al. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM Transactions on Graphics* (Aug. 2010).

[Oos11] Jeremiah van Oosten. *OpenGL Interoperability with CUDA*. https://www.3dgep.com/opengl-interoperability-with-cuda/. [Accessed 8-January-2022]. Dec. 2011.

[Tha11] Jonathan Thaler. "Deferred Rendering". In: (Feb. 2011). URL: https://www.researchgate.net/profile/Jonathan%5C_Thaler2/publication/323357208_Deferred_Rendering/links/5a8fce31aca272140560aaad/Deferred-Rendering.pdf.

[AMD11] *Using a pointer/image in a struct, array of images*. [Accessed 9-March-2022]. Jan. 2011. URL: https://community.amd.com/t5/archives-discussions/using-a-pointer-image-in-a-struct-array-of-images/td-p/50897.

[PJH17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Infinite Area Lights". In: *Physically Based Rendering: From Theory to Implementation*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. Chap. 14.2.4, pp. 845–850. ISBN: 978-0-12-800645-0.

[FA18] Petr Felkel and David Ambrož. *Rendering Into Texture*. [Accessed 28-April-2022]. Oct. 2018. URL: https://cent.felk.cvut.cz/courses/PGR2/seminars.html.

[KCL18] Yong Hwi Kim, Junho Choi, and Kwan H. Lee. "An efficient method for specular-enhanced BTF compression". In: *Computers Graphics* 75 (2018), pp. 1–10. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2018.06.001. URL: https://www.sciencedirect.com/science/article/pii/S0097849318300918.

[Slo21a] Jaroslav Sloup. *CUDA - types of memory and their usage*. https://cent.felk.cvut.cz/courses/GPU/2021/lectures/03/lecture_3.pdf. [Accessed 5-January-2022]. Oct. 2021.

[Slo21b] Jaroslav Sloup. *OPENGL COMPUTE SHADERS, CUDA OpenGL interoperation*. https://cent.felk.cvut.cz/courses/GPU/2021/lectures/08/lecture_8.pdf. [Accessed 5-January-2022]. Nov. 2021.

[Kal+22] Simon Kallweit et al. *The Falcor Rendering Framework*. Mar. 2022. URL: https://github.com/NVIDIAGameWorks/Falcor.

[khra] *clEnqueueReleaseGLObjects(3) Manual Page*. [Accessed 12-May-2022]. URL: https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/clEnqueueReleaseGLObjects.html.

[khrb] *Framebuffer object*. [Accessed 28-April-2022]. URL: https://www.khronos.org/opengl/wiki/Framebuffer_Object.

[Intela] *Intel® Embree - High Performance Ray Tracing*. [Accessed 4-January-2022]. URL: https://www.youtube.com/watch?v=5C-IYDMn9p4&ab_channel=IntelSoftware.

[Intelb] *Intel® oneAPI Base Toolkit System Requirements*. [Accessed 4-January-2022]. URL: https://www.intel.com/content/www/us/en/developer/articles/system-requirements/intel-oneapi-base-toolkit-system-requirements.html.

[CP] *OpenGL vs DirectX*. [Accessed 4-January-2022]. URL: https://www.cprogramming.com/tutorial/openglvsdirectx.html.

[NVa] *Turing Extensions for Vulkan and OpenGL*. [Accessed 3-January-2022]. URL: https://developer.nvidia.com/vulkan-turing.

[NVb] *What Is Vulkan?* [Accessed 4-January-2022]. URL: https://developer.nvidia.com/vulkan.

# Appendix A
# User Manual

## A.1 Wiimote Controls

Wiimote controls are mainly dedicated for the VR version of the application. Functionality binding is depicted in the diagram A.1.

## A.2 Desktop Controls

The desktop version of the application enables the interaction using a keyboard with a mouse, together with the Wimmote controller already discussed in the previous section. To enable interaction using a mouse, the *Mouse Mode* must be actived by a single right-click into the scene. This way the mouse cursor gets locked and it is possible to use the mouse within the application. To disable mouse mode, `Esc` key must be pressed or a menu must be displayed using a `M` key.

In terms of camera motion, it can be split into two categories. The first category represents *Orbit Controls*, which is similar to how the Wiimote controls work, and it utilizes mouse buttons. The second category is implemented in a way how movement, mainly in computer games, is implemented, which is based on the well-known `W,A,S,D` combination.

However, it is required to use only one type of interaction through the whole application run-time, otherwise the camera stops working properly and the application must be reset (run-time reset using key `R`, or hard reset).

In both cases, looking around with the camera is available by moving the mouse, when the mentioned *Mouse Mode* is active.

### a) Orbit Controls

Orbit movement controls require to move with the mouse while one of the mouse buttons is pressed. The motion is mapped to spherical coordinates defining the rotation/position.

- **Left Mouse Button** - camera orbits around the object.
- **Middle Mouse Button** - object rotates around its center.

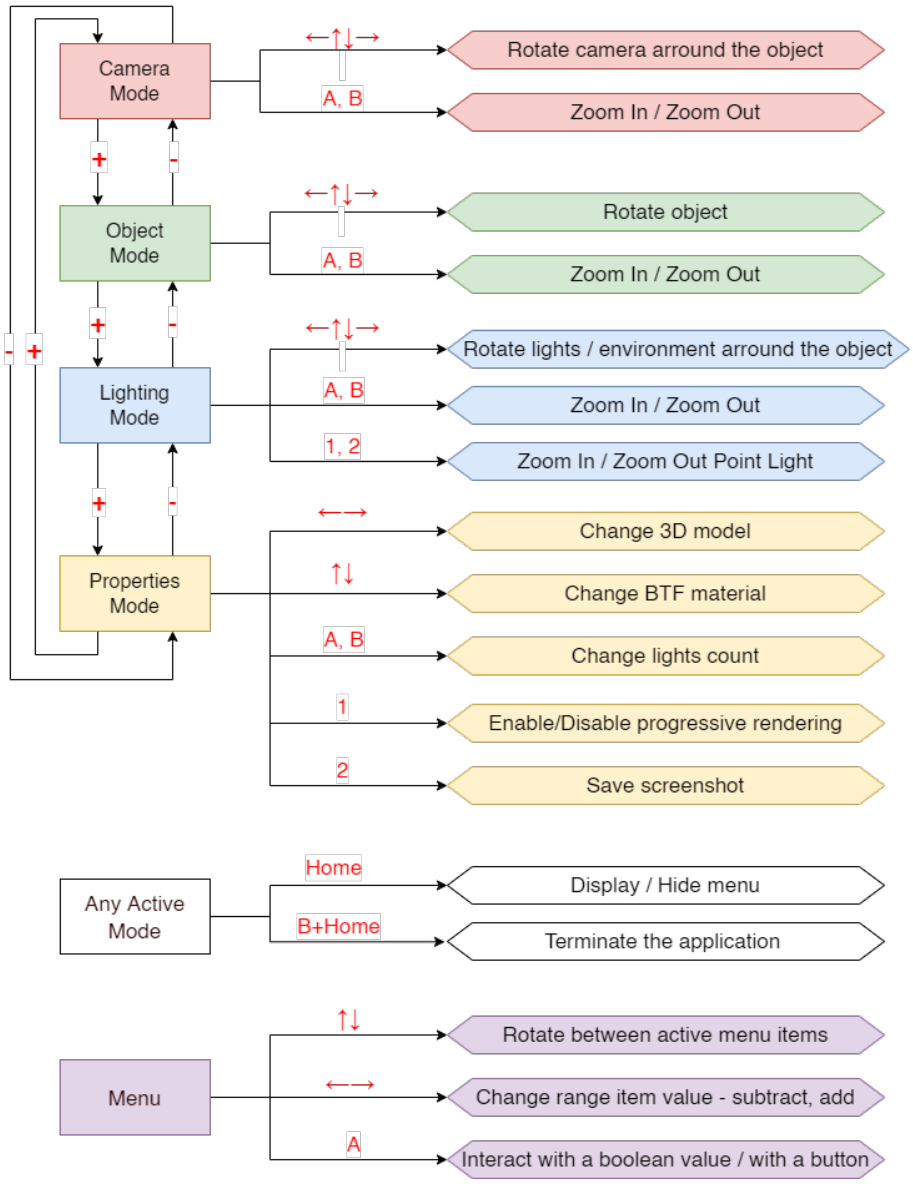



**Figure A.1:** Wiimote controls functionality mapping diagram.

- **Right Mouse Button** - environment or point light rotates around the object.
- **Mouse Wheel Rotation** - zoom in / zoom out.

### b) First-person Controls

First person controls enable free motion throughout the scene.

- **W**, **A**, **S**, **D** - move forward, left, backward, right.
- **Space** - move up.
- **Left shift** - move down.

### Control Over Parameters of the Application

Key bindings of the application parameters.

- **M** - display/hide menu.
- **Left** & **Right arrows** - change BTF material.
- **8**, **5** - next / previous 3D object.
- **Up** & **Down arrows** - change the number of lights evaluated in a single frame.
- **I** - initialize *Interpupillary Distance* (IPD) measurement and calibration of the XTAL 8K VR headset.
- **1**, **2**, **3** - use GLSL/CUDA/OpenCL renderer.
- **4**, **6** - lower / increase rendering resolution.
- **+**,**-** - increase / lower resolution of the currently used material.
- **P** - enable / disable progressive rendering.
- **N** - enable / disable *Image Based Lighting* (IBL).
- **U** - save standard screenshot.
- **O** - save VR screenshot (when VR headset used).
- **Esc** - terminates the application.

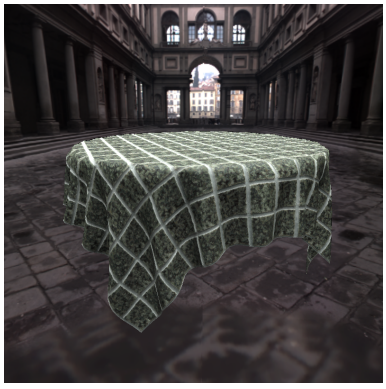
# Appendix B

## Image Gallery

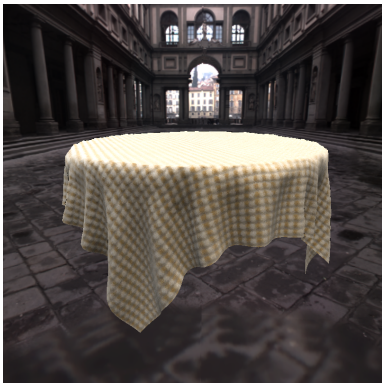
**(a) :** Corduroy, 1 Point light

**(b) :** CeilingHDR, 1 Point light

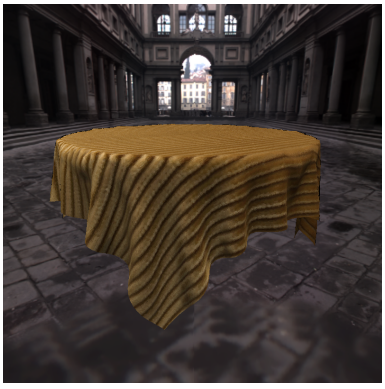
**(c) :** Corduroy, 200 Dir. lights



**(d) :** Impalla, 200 Dir. lights



**(e) :** Proposte, 200 Dir. lights



**(f) :** Corduroy, 200 Dir. lights

**(g) :** Pulli, 1 Point light

**(h) :** Pulli, 200 Dir. lights

**(i) :** CeilingHDR, 200 Dir. lights

**Figure B.1:** Examples of materials under various lighting and view conditions.

# Appendix C

## Configuration File Example

```
#### static data - should not be modified ####
btf-cubemap-folder=data/btf/cubemaps

#### customizable data ####
use-vr=true
use-vr-native-res=false
vr-fb-width=3840
vr-fb-height=2160

fb-width=1920
fb-height=1080

fb-min-scale-factor=-5.0
fb-max-scale-factor=3.0

## Models loading paths and params ##
model=data/obj/koule.obj
model=data/obj/bunny.obj
model=data/obj/hippo.obj

base-model-scale=1.0
base-model-offset=0.0-0.0-0.0

## BTF materials loading ##
btf-material=data/btf/materials/impalla
btf-material=data/btf/materials/ceilingHDR
btf-material=data/btf/materials/proposte

## Environment map loading ##
# Cathedral
envmap-path=data/envMaps/raw023.hdr
envmap-exposure=125

## CUDA Settings ##
cu-threads-per-block=256

## OpenCL Settings ##
ocl-threads-per-block=256
ocl-kernels-path=data/kernels/opencl

## Other parameters ##
bg-color=25-25-30
desktop-cam-init-pos=0.0,0.0,3.5
screenshots-folder=screenshots
```

**Listing 1:** Configuration File Example

# Appendix D

## Contents of Attached CD

Visual Studio project is available in the folder `project`. Windows x64 executable is available in the folder `windows`. PDF file of the thesis together with LaTeX source code is available in the folder `doc`. Some demo images are available in the folder `img`. Instructions (*README.txt*), how to launch the application are supplied with the executable in the `windows` folder.

Both executable in `windows` folder and the project in `project` folder require the same `data` sub-folder, which is due to the size constraints split into many zip files.

```
DP/
├── project/
│   └── data/
│       ├── btf/
│       │   ├── cubemaps/
│       │   └── materials/
│       ├── envMaps/
│       ├── kernels/
│       ├── obj/
│       ├── shaders/
│       └── textures/
├── windows/
│   └── data/
│       ├── btf/
│       │   ├── cubemaps/
│       │   └── materials/
│       ├── envMaps/
│       ├── kernels/
│       ├── obj/
│       ├── shaders/
│       └── textures/
├── doc/
│   ├── pdf/
│   └── latex/
└── img/
```