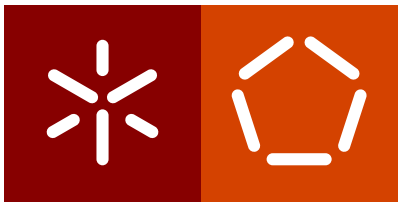
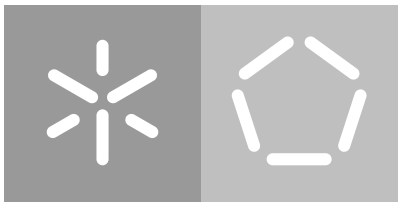


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Manuel da Silva Santos

Mobile Ray-tracing

April 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Manuel da Silva Santos

Mobile Ray-tracing

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Doutor Luís Paulo Peixoto dos Santos

April 2019

ACKNOWLEDGEMENTS

Agradeço a todos que me apoiaram, direta ou indiretamente, neste longo percurso académico. Em especial, ao meu orientador, professor Doutor Luís Paulo Peixoto dos Santos que propôs este tema, deu-me sempre boas ideias e por todo o seu apoio. Também, ao professores dos perfis de Computação Gráfica e de Computação Paralela e Distribuída pela excelente formação que deram.

A todos os meus amigos que sempre me apoiaram neste longo percurso. Em especial, destaco cinco amigos: João Costa, André Pereira, Miguel Esteves, Eduardo Mendes e Miguel Rego que desde que os conheci me ajudaram muito a ultrapassar diversos obstáculos tanto a nível pessoal como na minha formação académica.

Ao grupo Embedded Systems Research Group (ESRG) por me terem acolhido durante uma bolsa de investigação, me tratarem como parte dessa grande família e pelo apoio que sempre me deram.

E por último mas não menos importante, à minha família, em especial aos meus irmãos Pedro Santos e Paulo Santos pelos muitos sacrifícios que fizeram para me apoiar e permitir chegar onde cheguei.

ABSTRACT

The technological advances and the massification of information technologies have allowed a huge and positive proliferation of the number of libraries and APIs. This large offer has made life easier for programmers in general, because they easily find a library, free or commercial, that helps them solve the daily challenges they have at hand.

One area of information technology where libraries are critical is in Computer Graphics, due to the wide range of rendering techniques it offers. One of these techniques is ray tracing. Ray tracing allows to simulate natural electromagnetic phenomena such as the path of light and mechanical phenomena such as the propagation of sound. Similarly, it also allows to simulate technologies developed by men, like Wi-Fi networks. These simulations can have a spectacular realism and accuracy, at the expense of a very high computational cost.

The constant evolution of technology allowed to leverage and massify new areas, such as mobile devices. Devices today are increasingly faster, replacing and often complementing tasks that were previously performed only on computers or on dedicated hardware. However, the number of image rendering libraries available for mobile devices is still very scarce, and no ray tracing based image rendering library has been able to assert itself on these devices. This dissertation aims to explore the possibilities and limitations of using mobile devices to execute rendering algorithms that use ray tracing, such as progressive path tracing. Its main goal is to provide a rendering library for mobile devices based on ray tracing.

RESUMO

Os avanços tecnológicos e a massificação das tecnologias de informação permitiu uma enorme e positiva proliferação do número de bibliotecas e APIs. Esta maior oferta permitiu facilitar a vida dos programadores em geral, porque facilmente encontram uma biblioteca, gratuita ou comercial, que os ajudam a resolver os desafios diários que têm em mãos.

Uma área das tecnologias de informação onde as bibliotecas são fundamentais é na Computação Gráfica, devido à panóplia de métodos de renderização que oferece. Um destes métodos é o ray tracing. O ray tracing permite simular fenómenos eletromagnéticos naturais como os percursos da luz e fenómenos mecânicos como a propagação do som. Da mesma forma também permite simular tecnologias desenvolvidas pelo homem, como por exemplo redes Wi-Fi. Estas simulações podem ter um realismo e precisão impressionantes, porém têm um custo computacional muito elevado.

A constante evolução da tecnologia permitiu alavancar e massificar novas áreas, como os dispositivos móveis. Os dispositivos são hoje cada vez mais rápidos e cada vez mais substituem e/ou complementam tarefas que anteriormente eram apenas realizadas em computadores ou em hardware dedicado. Porém, o número de bibliotecas para renderização de imagens disponíveis para dispositivos móveis é ainda muito reduzido e nenhuma biblioteca de renderização de imagens baseada em ray tracing conseguiu afirmar-se nestes dispositivos. Esta dissertação tem como objetivo explorar possibilidades e limitações da utilização de dispositivos móveis para a execução de algoritmos de renderização que utilizem ray tracing, como por exemplo, o path tracing progressivo. O objetivo principal é disponibilizar uma biblioteca de renderização para dispositivos móveis baseada em ray tracing.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

ACRONYMS

AABB	Axis-Aligned Bounding Box.
ALU	Arithmetic logic unit.
API	Application Programming Interface.
APK	Android Package.
App	Application.
BRDF	Bidirectional Reflectance Distribution Function.
BSD	Berkeley Software Distribution.
BSDF	Bidirectional Scattering Distribution Function.
BSSRDF	Bidirectional Scattering-Surface Reflectance Distribution Function.
BTDF	Bidirectional Transmittance Distribution Function.
BVH	Bounding volume hierarchy.
CG	Computer Graphics.
CISC	Complex instruction set computer.
CPU	Central processing unit.
cstdlib	C Standard Library.
Demo	Demonstration.
DI	Departamento de Informática.
DLP	Data Level Parallelism.
FOSS	Free and open-source software.
GCC	GNU Compiler Collection.
GI	Global Illumination.
GLM	OpenGL Mathematics.
GPL	GNU General Public License.
GPU	Graphics processing unit.
GTest	Google Test.
GUI	Graphical User Interface.

HT	HyperThreading.
IDE	Integrated Development Environment.
ILP	Instruction-level parallelism.
JNI	Java Native Interface.
JVM	Java virtual machine.
k-DOP	k-Discrete Oriented Prototype.
Lib	Library.
Libstdc++	C++ Standard Library.
LLVM	Low Level Virtual Machine.
MEI	Mestrado em Engenharia Informática.
MiEI	Mestrado Integrado em Engenharia Informática.
MIS	Multiple Importance Sampling.
NDK	Native Development Kit.
OBB	Oriented Bounding Box.
OpenGL	Open Graphics Library.
OS	Operating system.
PBRT	Physically Based Rendering: From Theory To Implementation.
PC	Personal Computer.
PDF	Probability Density Function.
Pixel	Picture element.
PRNG	Pseudo Random Number Generator.
RAM	Random-access memory.
RGB	Red Green Blue.
RISC	Reduced instruction set computer.
RNG	Random Number Generator.
RT	Ray Tracing.

SAH	Surface Area Heuristic.
SDK	Software development kit.
SIMD	Single instruction, multiple data.
SMT	Simultaneous multithreading.
SPL	Samples per light.
SPP	Samples per pixel.
stdlib	Standard Library.
STL	Standard Template Library.
Texel	Texture element.
UI	User Interface.
UM	Universidade do Minho.
Voxel	Volume element.

1

INTRODUCTION

1.1 CONTEXT

Programming is like building something with primitive blocks and it can be a very difficult task if we have to program every aspect of the application without some "blocks" already built for us to use. That's why in the programming world there is a whole panoply of free and commercial libraries with APIs that provide a huge quantity of functions for the programmer to use.

In computer graphics, there are many 3D graphics libraries that provide functionalities to render images based in different techniques. Two of the most used render techniques are rasterization and ray tracing. Ray tracing can be used to calculate and simulate the path of particles and waves. It can simulate the propagation of sound, the path of light and even simulate technologies developed by men like the Wi-fi networks. It can render an image by tracing the path of light through pixels in an image plane and simulating the effects of its interactions with virtual objects. It can render much more realistic shadows, reflections and refractions and in an easier way than rasterization. This technique is capable of producing a very high degree of visual realism but at a great computational cost.

Since nowadays we have more and more mobile systems whose computational power increases every year, there is a need for more libraries to help the programmers develop applications for these systems. However, there are not many options to choose from for developing a renderer based on ray tracing. That's why this dissertation focuses on assessing and providing a ray tracer library for mobile systems.

It is important to mention that this dissertation is not focused on rendering techniques other than ray tracing. It is not focused in assessing different integrators (numerical solutions to the rendering equation) and / or assessing different approaches in ray tracing like Packet Traversal. It is also not focused on assessing different quasi random numbers generators neither assessing the performance of different computer systems. It focuses only on providing a library with the basic functionality of a ray tracer, along with some rendering components in order to demonstrate the potential of the library itself.

These rendering algorithms based in ray tracing are very useful because they allow rendering photo-realistic images with a degree of realism much higher than the graphics cards do by default through rasterization. Since the computing power in mobile processors has been increasing, a mobile device with a mid-range processor could run these algorithms in a useful time.

Figure 1.: Illustration of the most common mobile devices - tablet and smartphone (du Net).

1.2 MOTIVATION

Among other factors, the productivity of a programmer depends on what libraries he can use. However, there are almost no rendering libraries based on ray tracing available today for mobile systems like Android, iOS, Windows 10 Mobile, BlackBerry 10, Tizen, Sailfish OS, Symbian and Ubuntu Touch. That said, it is likely that these systems already have enough processing power to render images with fairly complex scenes in an acceptable time by using algorithms based in ray tracing.

It is also important to note that there is not much documentation about the advantages and limitations of executing different rendering algorithms in these devices.

Last but not least, it is important to mention that of all the operating systems available for mobile devices, the one chosen for this ray tracing library was Android due to popularity and portability. More than 75% of the mobile devices use Android, and only 21% use iOS (StatCounter). It also supports a variety of different devices: smart phones, tablet computers, smart TVs, smart watches and even laptop / desktop computers.

Figure 2.: Illustration of a variety of mobile devices compatible with Android (COOLFINESSE (2017)).

1.3 GOALS

The main goal of this dissertation is to assess and demonstrate the advantages of running rendering algorithms in mobile devices.

It is also intended to promote and facilitate the development of applications for mobile systems that use ray tracing techniques, with special emphasis on rendering applications. To do this, a library will be developed to support the fundamental operations of a ray tracing engine. This library will allow the agile development of diverse applications, by using components that invoke the functionality of the library itself.

Additionally, it is intended to supply rendering components at a higher abstraction level, like the camera, scene and the integrator that facilitates further the development of applications.

Finally, it is important to do a demonstration of the rendering application with some interface layer to let the user assess the performance of several functionalities provided by the library.

1.4 Applications of Ray Tracing

The subject of this dissertation is ray tracing because it is a technology widely used in several areas of Computer Graphics. It is a rendering technique that is widely used in the areas of theater and television lighting. It allows set and lighting designers, actors, and directors to develop and visualize complex lighting setups months before a production begins.

Ray tracing is capable of simulating all the light paths in a given environment which is called Global Illumination. Global illumination is a physically correct model, which accurately simulates light behavior in a real physical environment. It is a valuable engineering tool in that allow us to quantitatively analyze the distribution and direction of light and help calculate the radiant heat transfer.

Finally, ray tracing can also be used in Animation. It can be used to add fancy effects such as reflection and shadowing that are often difficult and time consuming for traditional artists to produce. Graphical technology is also capable of rendering photo-realistic images that would be nearly impossible to produce without computerized ray tracing.

1.5 Document Structure

This dissertation is organized in 7 chapters: Introduction (1), State of the Art (2), Software architecture - Library (3), Software architecture - Rendering Components (4), Android Layer (5), Demonstration: Global Illumination (6) and Conclusion & Future work (7).

The first chapter describes the context and motivation behind this work, as well as its goals. Its main purpose is to identify the problem at hand and set up goals that should be accomplished.

The second chapter introduces the main concepts of ray tracing and compares different implementations of ray tracers already available in the world wide web. The reason behind this comparison is to show how many ray tracers are already available to mobile devices and highlight the differences between the features they provide. It also provides some information about the processors of today that helps realize their ability to execute computationally demanding algorithms like ray tracing.

The third chapter explains the proposed approach and explains each module developed in the library.

The fourth chapter describes all the rendering components provided along with the library available for the programmer to use.

The fifth chapter starts with an explanation of some Android specifics, such as the user interface by characterizing its work flow and mentioning some of the benefits and challenges overcame during the development of the application. It also describes some programming decisions made during the development of the ray tracing library.

The sixth chapter summarizes the key results obtained by executing different algorithms with different number of threads and different acceleration structures. It ends with a small comparison of the features provided in this library and an Android ray tracing engine developed by third parties called Android CPU Raytracer (Dahlquist).

Finally, the last chapter ends this dissertation with the conclusions that can be withdrawn from this work and proposes some future work.

Last but not least, the Appendix contains this library API and an example code of how to load a scene from a wavefront obj file. It also contains all the measured execution times in order to let the reader assess better the performance of the developed library.

2

STATE OF THE ART

2.1 RAY TRACING

Ray tracing one frame can be, simultaneously, a computationally demanding task and an embarrassingly parallel task. As tracing rays is a recursive process which aims to calculate the radiance of light of each individual pixel separately. At least one ray is shot per pixel and in a naive approach each ray would be intersected by all the objects in the scene in order to determine which one is the closest primitive intersecting that given ray. To evaluate the light intensity that an object scatters towards the eye, the intensity of the light reaching that object has to be evaluated as well. Ray tracing achieves this by shooting additional secondary rays, because when a ray hits a reflecting or transparent surface, one or more secondary rays are cast from that point, simulating the reflection and refraction effects.

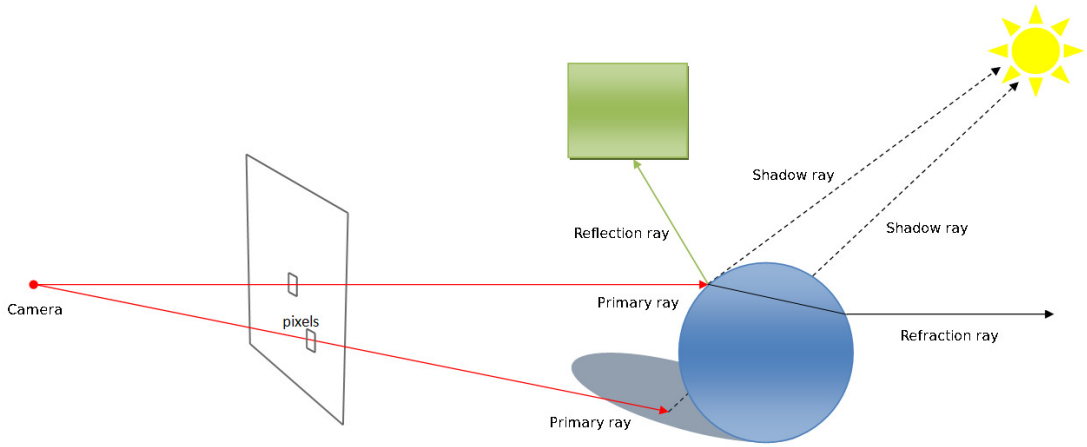


Figure 3.: Illustration of a typical ray tracer algorithm (Galaxie).

A typical image of 1024x1024 pixels tends to cast at least a million primary rays and more rays as shadow, reflection, refraction and secondary rays. As each ray needs to be intersected with the scene geometry, ray tracing algorithms can't provide interactive frame rates with ease.

2.2 TYPICAL CPU FEATURES

Fortunately, the present state of available technology provides affordable machines with multiple CPU cores that can work in parallel and with great performance. This is achieved thanks to Micro-architectural techniques developed inside the processor that are used to exploit ILP, like: the CPU cache, the Instruction pipelining, the Cache prefetching, the Superscalar execution, the Out-of-order execution, the Register renaming, the Speculative execution, the Branch prediction and the SIMD instruction set already available in the current processors.

The cache is a very small and fast multilevel memory inside the processor that temporarily stores the data read from the main memory.

This memory allows reading its content at a very low latency in the order of magnitude of around 1 nanosecond in the first level, 10 nanoseconds at second level and 50 nanoseconds at third level compared to the typical 100 nanoseconds of a DDR main memory. Like it was said, the CPU cache can be around 100x faster than the RAM.

The downside is that it is very small because its production cost are much more expensive than a typical RAM memory. Nowadays, in a common PC, the level 1 has 64kB, the level 2 has 256kB and the level 3 has 8 MB which is very little compared to the typical 16GB provided by the RAM.

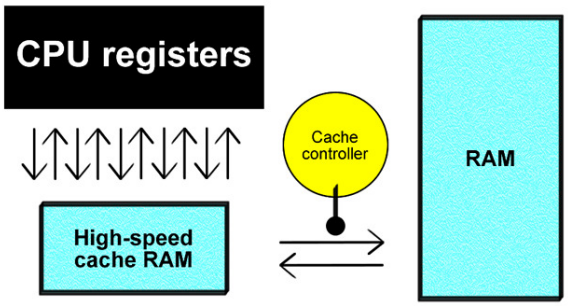


Figure 4.: Illustration of a typical cache memory inside a microprocessor (Karbo and Aps.).

The instruction pipelining is a feature inside a processor which allows to reduce the processor latency by allowing a form of parallelism called instruction-level parallelism within a single processor. Basically, one instruction is divided into stages and it allows the possibility to execute different stages of different instructions simultaneously. In the early days, the Classic RISC pipeline was typically divided into five stages:

- Fetching the instruction
- Decoding the instruction

- Executing the instruction where the arguments can be fetched from registers (1 cycle latency) or from memory (2 cycle latency)
- Memory access where it ensured that writing in memory was always performed in the same stage and allowed to use that value in another instruction before it was written in memory
- Writing back the result value in the register file

Nowadays, the current microprocessors have a pipeline with 8-14 stages and can be more complex than the Classic RISC, but the operating principle is the same. This allows faster CPU throughput, which means the number of instructions that can be executed in a unit of time is greater, than it would otherwise be possible at a given clock rate.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 5.: Illustration of the Classic RISC multilevel CPU pipeline (Dictionaries and Encyclopedias).

The Cache prefetching is, as the name implies, a feature in the processor that makes the processor fetch the data and the instructions from the RAM to the CPU cache before it really needs to read or execute. This allows to reduce the time which the processor spends waiting for the main memory data.

Superscalar execution is when the CPU uses multiple functional units simultaneously for the same type of task. For example, the CPU can fetch, decode and even execute two instructions at the same time.

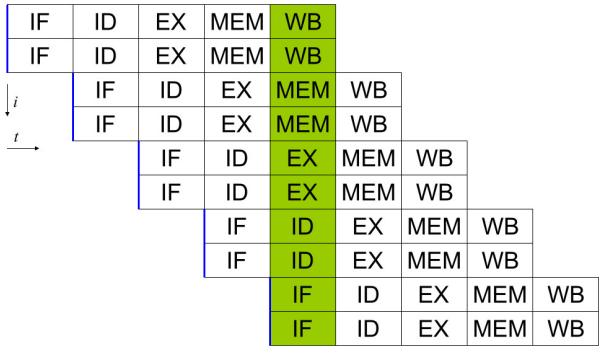


Figure 6.: Illustration of a simple superscalar pipeline (Wikipedia (a)).

Out-of-order execution is the name of the technique that allows the CPU to reorder the instructions that are going to be executed. This allows the CPU to execute the instructions out of order when they don't have a dependency and so it makes use of instruction cycles that would otherwise be wasted. Usually this technique uses the register renaming which is another technique that is used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations.

Speculative execution is the technique where the CPU performs some task that may not be needed. This prevents the CPU from having a delay that would have to be incurred by doing the work after it is known that it is needed.

Branch prediction is the technique where the CPU starts to execute the instructions of a path of a branch before it even knows which branch it needs to take.

Finally, the SIMD instruction set is a set of special instructions that allows the processor to read, write and process larger chunks of data with less instructions. Typically, a 64 bit processor can only work with 64 bits of information at a time during the execution of an instruction. This feature can allow the read or write up to 256 or even 512 bits by using special registers and additional ALUs provided in the processor.

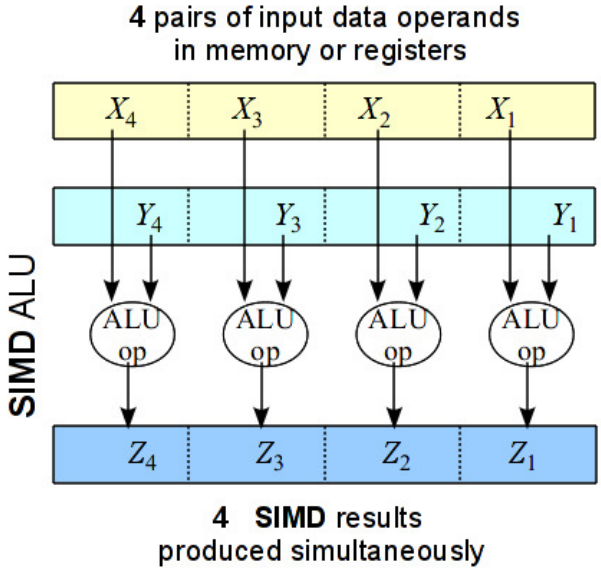


Figure 7.: Illustration of a typical execution of SIMD extension (Bishop).

Nowadays, even mobile devices like smart phones and tablets have multiple CPU cores with features similar to those described above. This opens the possibility to execute more computationally demanding algorithms, like ray tracing, in these devices. It is important to mention that all the optimization techniques described above are or can be transparent to the software programmer, and which only the processor and the compiler have to know they exist.

2.3 KEY FEATURES OF RAY TRACING FOR THIS WORK

Ray tracing is an algorithm that can have a panoply of features or optimizations in order to reduce the time required to trace all the rays and / or to show the results as fast as possible. It also, like any application, can have different types of software licenses and be executed in different platforms.

For this work, the most important features in a ray tracer are: the type of software license, the platform support, interactivity, if it is progressive and the type of rendering components.

2.3.1 *Type of software license*

A software application can have different software licenses, but in order to simplify this dissertation, the software license were grouped into three types: Free, Commercial and Open Source.

The free license means that the user has the right to execute freely the application but cannot copy, modify or distribute the implemented code. This includes the Freeware, Shareware and Freemium types of software licenses.

The commercial license means that the user cannot even execute the application without buying it first and cannot copy, modify or distribute the implemented code. This includes the Proprietary and Trade secret types of software licenses.

The open source license means that the user has the right to execute freely the application and can copy, modify and distribute the source code. This includes the Public-domain software, and the Permissive and Copyleft types of software licenses like the BSD and GPL.

The software license is very important in this work because it lets the user know if he can extend the software or if he can just use the application.

2.3.2 *Platform*

An application can only be executed in the platforms that the developers compiled the code for. So, a ray tracer that can be executed in a desktop may or may not be executed on another platform, like in a mobile device. Usually, the software is specific for a single Operating System. This information is very important for this work because it informs us whether a ray tracer can be executed in a mobile device with the typical Operating System like Android, iOS or Windows Mobile.

2.3.3 *Interactivity*

In ray tracing, interactivity means rendering an image with a very low response time, like a few milliseconds per frame. An interactive ray tracer can render a scene with multiple frames per second.

2.3.4 *Progressive*

A typical ray tracer shows the rendered image only after the whole process is complete. A progressive ray tracer is a ray tracer that updates the color of pixels in the screen as soon as the rays are traced, instead of waiting for the rendering process to complete. This means that an image is rendered quickly with some aliasing or noise and it is progressively improved over time.

2.3.5 *Types of Rendering Components*

A ray tracer can be developed with the different rendering components programmed separately. Some examples of rendering components are: Integrators, Cameras, Scenes, Samplers, Shapes, Lights and Acceleration Structures. In some ray tracers, these rendering components can even be programmable by the user. This is very important because it allows the user to develop his own renderer based on ray tracing without having to develop every feature in the ray tracer.

2.4 RELATED WORK

The possibility of rendering an image with ray tracing was demonstrated in 1980 by Whitted (dos Santos) and since then the number of libraries that provide basic ray tracing functionalities increased greatly. There is a wide range of different ray tracers available today for the programmer to use, yet the majority can only be used with the traditional personal computer hardware (desktop or laptop).

The table 2 shows some applications or frameworks that use ray tracing available today and compares them according to their type of license, platform compatibility, interactivity, if they are progressive and whether they allow development of your own rendering components like the integrator and sampler. Note that some of these ray tracers provide only the engine with the basic ray tracing functions, such as creating rays and intersecting them with geometric primitives, so the rendering components are only programmable if the application that uses the engine supports it. During the research of the available ray tracers, others than the ones presented in the table were found, but they were excluded because the documentation was very poor without explaining the basic functionalities provided or because there was no documentation at all.

Table 2.: Comparison of different applications/frameworks that use ray tracing.

Product	License[a]	Platform	Mobile	Interactivity	Progressive	Programmable Components
Optix (Nvidia (a))	F	Nvidia GPU	✗	✓	✓	✓
Optix Prime (Nvidia (b))	F	CPU & Nvidia GPU	✗	✓	✓	✓
RenderMan RIS (Pixar)	C	CPU	✗	✓	✓	✓
OctaneRender (Inc)	C	GPU	✗	✓	✓	✗
Embree (Intel	O	Intel CPU	✗	✓	✓	✓
Radeon Rays (AMD)	O	CPU & GPU	✗	✓	✓	✓
PBRT (Matt Pharr)	O	CPU	✗	✗	✗	✗
Visionaray (Zellmann)	O	CPU & Nvidia GPU	✗	✓	✓	✗
YafaRay (Gustavo Pichorim Boiko)	O	CPU	✗	✗	✗	✗
tray_rust (Usher (c))	O	CPU	✗	✗	✗	✗
micro-packet (Usher (a))	O	Intel CPU	✗	✗	✗	✗
tray (Usher (b))	O	CPU	✗	✗	✗	✗
The G3D Innovation Engine (Morgan)	O	CPU	✗	✗	✗	✗
HRay (Kenneth)	O	CPU	✗	✗	✗	✗
Mitsuba (Jakob)	O	CPU	✗	✗	✗	✗
Indigo RT (Limited)	O	CPU & GPU	✗	✓	✗	✗
jsRayTracer (Chedeau)	O	CPU	✗	✓	✓	✗
Android CPU Raytracer (Dahlquist)	O	CPU	✓ (Android)	✓	✗	✗

[a] F - Free, C - Commercial, O - Open source

2.4.1 *Conclusions*

As the table 2 shows, there is a lack of generic ray tracing libraries for the mobile devices. Although there are some closed-source ray tracing demo applications, only one ray tracer already available has some sort of documentation and is compatible with mobile devices, in this case with Android. This ray tracer is open source, uses only the CPU of the device and has a good performance. It only renders spheres and even allows interactions with them during the render process. But, it doesn't support progressive rendering and also does not allow the programmers to use their own rendering components.

This dissertation aims to fix this lack of libraries, by providing one that contains ray tracing basic functionalities and the ability to let the programmer be able to develop his own rendering components like the sampler, integrator, camera, types of lights and even develop his own object loader. It also studies the drawbacks that these mobile devices may have comparing with the average multi-core personal computer hardware. Finally, a small comparison was made with the Android CPU Raytracer (Dahlquist) in order to illustrate the advantages and disadvantages of both. This comparison is important because it enriches all the work done, as it demonstrates if the performance of the provided features are relatively efficient.

3

SOFTWARE ARCHITECTURE - LIBRARY

3.1 APPROACH

Three distinct layers compose the demo application. The ray tracer library is the bottom layer, that can be extended and customized by the middle layer, the rendering components. The top layer is the user interface (UI), which receives and shows the data information to the end-user. The high level architecture of these 3 layers can be seen in the figure 9.

The top layer is the UI which is Operating System (OS), and device dependent. As the library was developed in native code, the UI is divided in 2 sub-layers: Android UI and Java Native Interface (JNI). The Android UI lets the user select which rendering components he wants to use and sends this information to the JNI sub-layer. The main purpose of this demo application is to demonstrate the ray tracing library capabilities. It allows the user to view the rendering image, choose the scene and select the main available options of the rendering components. It allows the selection of shader, number of threads, samples per pixel (SPP) and per light (SPL), it lets to choose the resolution of the rendered image and which acceleration structure to use. Later, the JNI sub-layer configures and sets the bottom 2 layers (rendering components and ray tracer library) with the received values. Regardless of the UI importance, the focus of this dissertation was more on the bottom layers. Aspects like quality of experience and usability of UI were not addressed during the development of this project.

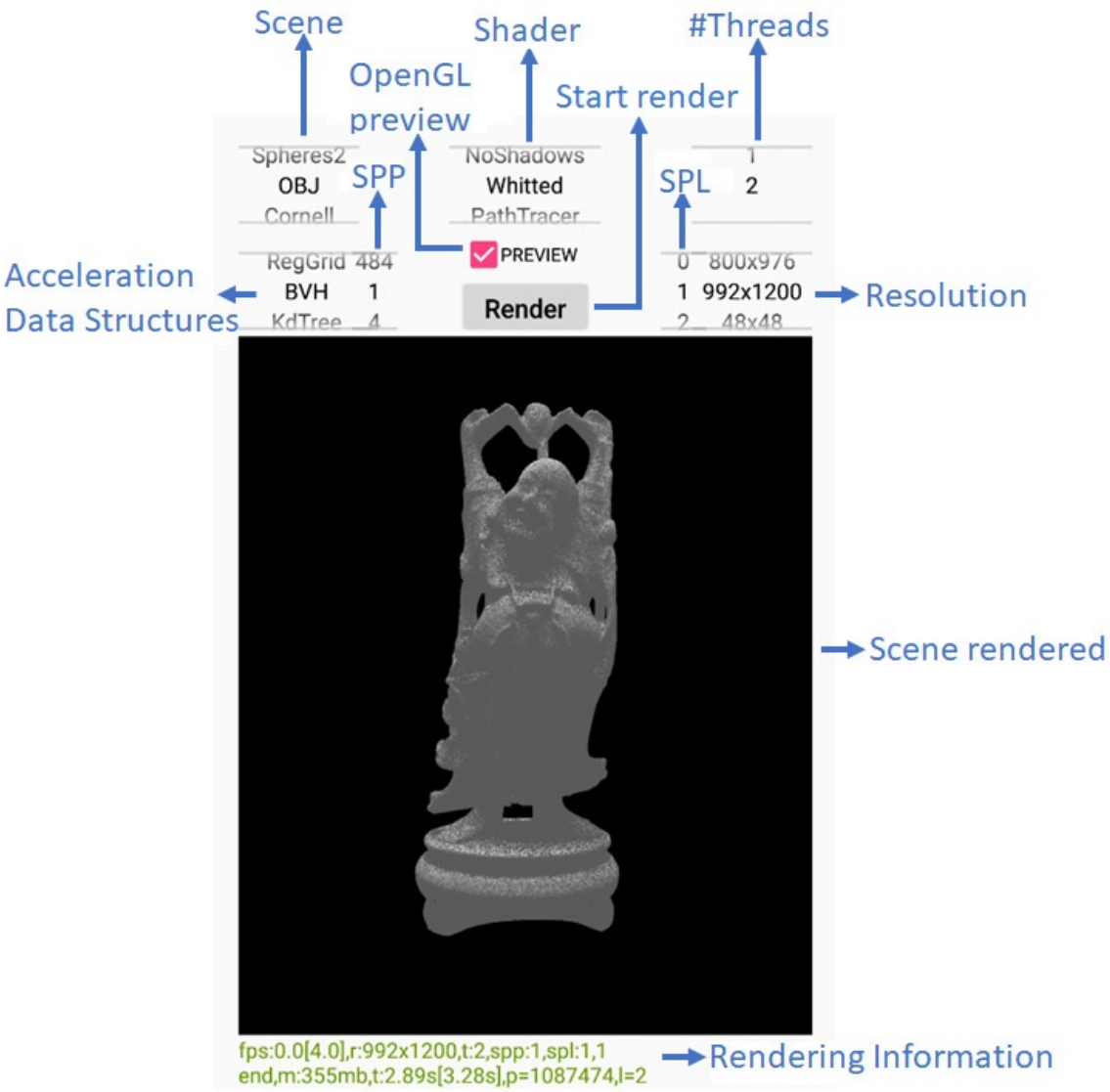


Figure 8.: Illustration of the developed graphical user interface.

The middle layer provides the rendering components, which are abstract concepts about rendering, that use functionalities that the library itself offers to the programmers. Some of these rendering components are: the camera, the light, the sampler, the shader and the object loader. These rendering components are useful for the programmer since it allows them to use features without having the need to know how these were developed. And, of course, this facilitates and accelerates the development of new rendering applications for the programmers. This layer will be described with more detail in the chapter 4.

The bottom layer is the library itself, which contains the business logic of basic features in a renderer, that the rendering components use. These features are the basic functionalities of a ray tracing engine. Those functionalities can be: create vectors and points; create a scene with different shapes like triangles, spheres and planes; create materials; create a

primitive (a shape with a material); cast rays; and even intersect the rays with the primitives in a scene (with or without acceleration structures). More details regarding this layer will be addressed in the section 3.4.

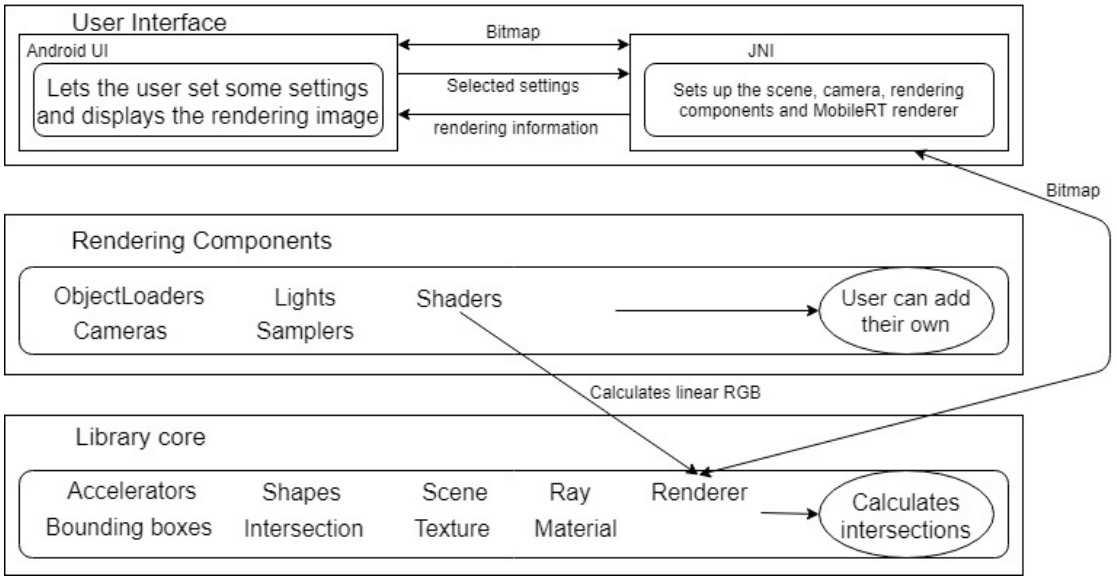


Figure 9.: Illustration of the three layers in the application.

Besides the abstraction layers, there are some important strategic decisions made in order to guide the progress of the development of this library.

For example, the primary rays always have origin in the camera. This decision was made in order to achieve a renderer without dependency of the shader. Another decision made was to develop some acceleration data structures, in order to obtain an efficient ray tracer capable of rendering scenes with some complexity and at useful time. This is important in order to give the user a fast response, saving time and battery of the mobile device. Besides a fast response, this library also allows a progressive rendering, which lets the user see the rendering image while getting more samples per pixel, over time, and gaining detail. Also, as Android devices don't have as much RAM as personal computers, all the floating point member variables used are single precision, in order to use the least memory possible, and allowing to render more complex scenes. Last, but not least, is that the code was developed in a modular way, in this case was programmed in an object oriented paradigm. This allows the programmers to code their own rendering components, like: the shader, camera, sampler, light and the object loader, without having to develop the basic features in the renderer engine as mentioned above.

3.2 OTHER APPROACH

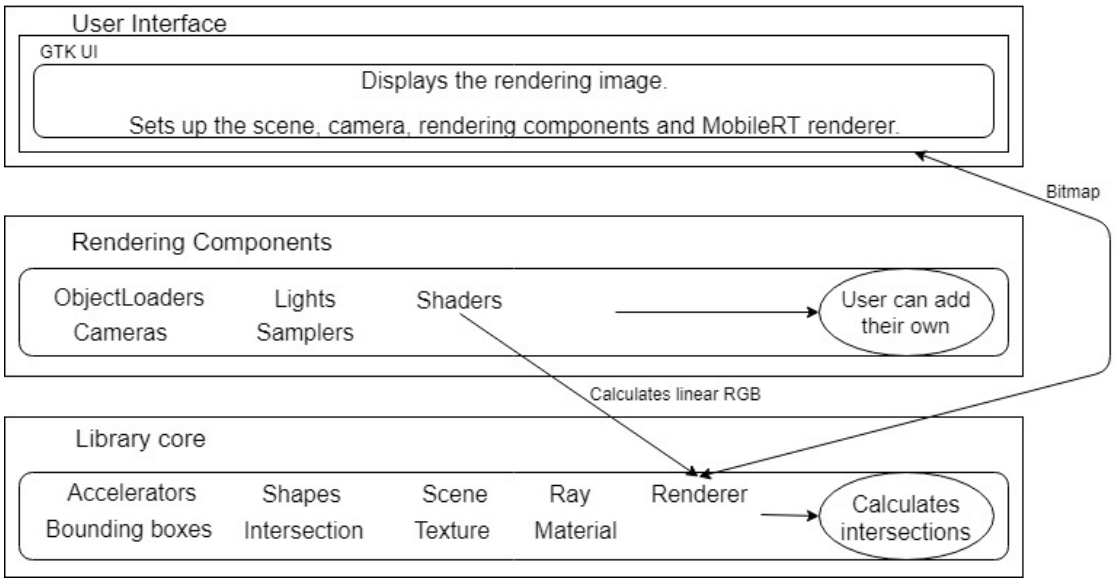


Figure 10.: Illustration of the three layers in Linux native application.

During the development of this dissertation another application was implemented to validate and test the 3 architecture layers. This new approach focused on changing the UI layer by replacing it with native code for creating a Linux desktop application. This consisted in creating a window to show the progressive rendering using the GTK library.

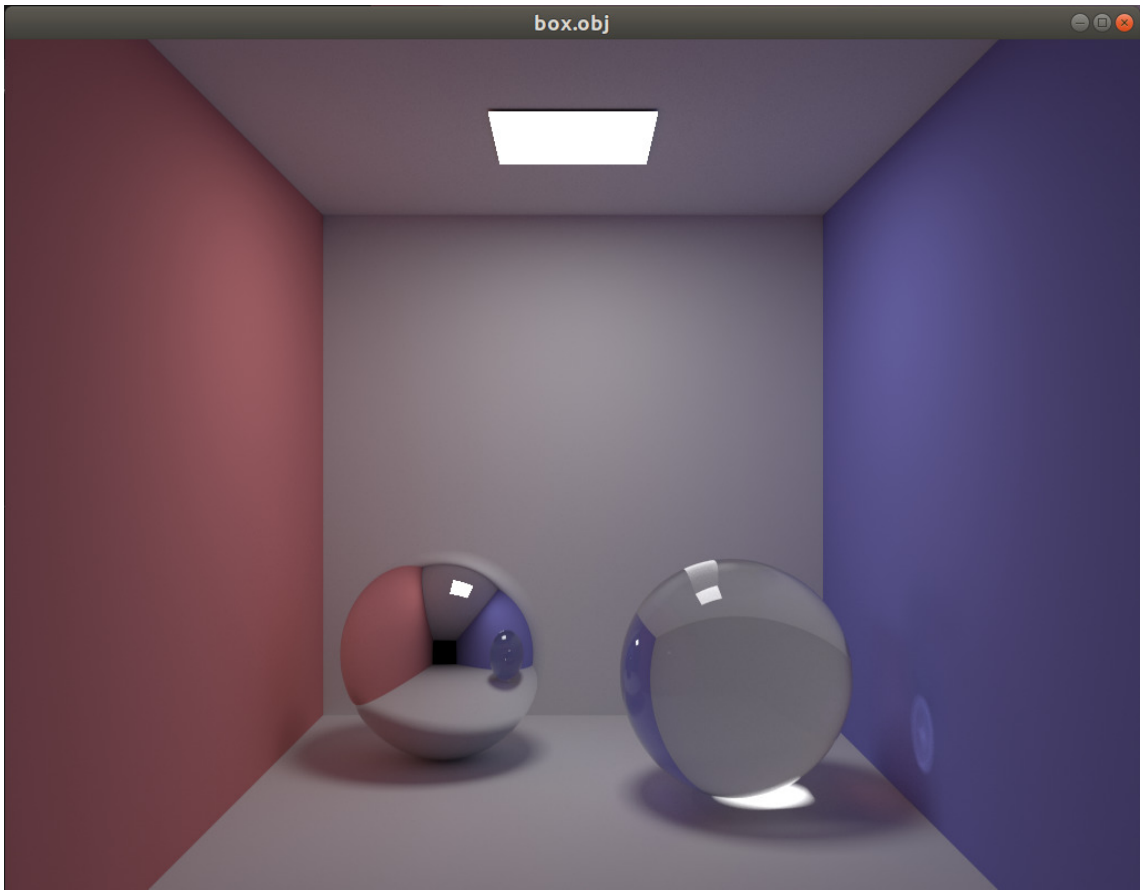


Figure 11.: Linux native application using the MobileRT. The scene rendered is the famous Cornell Box at 1024x768 and with 20000 spp and 1 spl.

This application was very useful during the development of the library because it allowed a more agile debug process. Besides, it also helped to test and validate the code with different C++ compilers namely GNU Compiler Collection (GCC) (team (b)) and Clang from LLVM (Team (a)), and different tools like Valgrind (Developers) and static code analyzers like the Clang tidy tool (Team (b)). It also allowed to do a more complete profile analysis with the Linux perf (Gregg) application, which can be used to measure the time spent on each function. Last, this extra application helped to compare the performance and validate the rendering algorithms implementations with Physically Based Rendering: From Theory To Implementation (PBRT) Pharr et al. (2016). This is a well known reference in the literature which implements a panoply of different state of the art rendering algorithms.

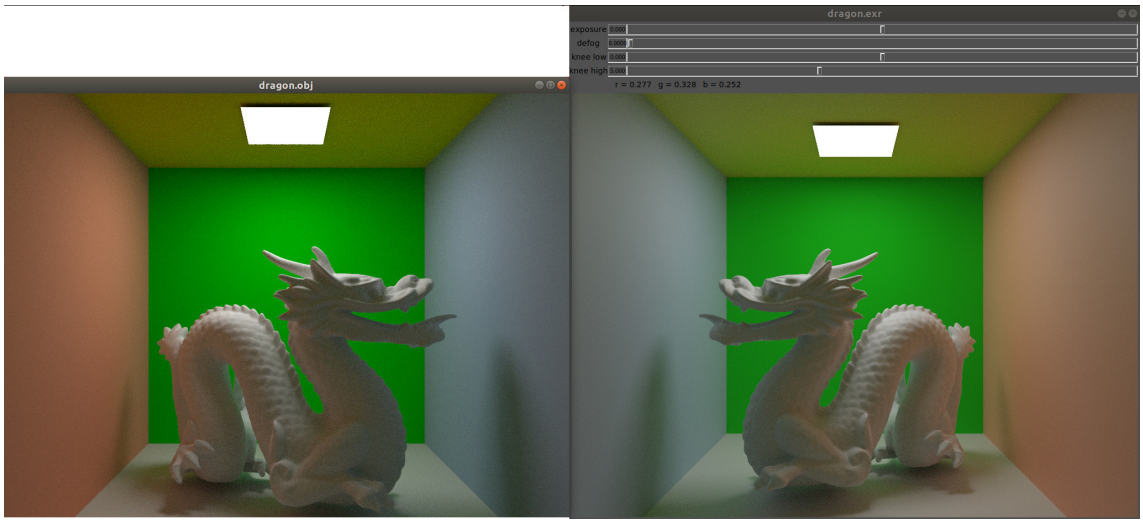


Figure 12.: Comparison of a scene rendered with MobileRT Path Tracer (left) and with PBRT Path Tracer (right).

The figure 12 shows a comparison of a scene with a Dragon model (871306 triangles with diffuse/Lambert material) inside a Cornell Box (12 triangles with diffuse material) rendered with MobileRT and with PBRT Path Tracer algorithms. The image on the left was rendered with MobileRT and took 1 minute and 26.975 seconds to render, while the image on the right was rendered with PBRT-v3 and took 2 minutes and 59.91 seconds to render. This comparison was made in the same hardware, which is described in the table 3, with 1024x768 pixels, with 64 samples per pixel and 1 sample per light, and it was used 8 threads in both images. Both renderers used the BVH acceleration structure in order to reduce the rendering time, which will be described in the section 3.4.8.

3.3 METHODOLOGY

This project started with two main goals in mind. The first goal was to develop a basic android application, capable of including a native library. This native library, was the second goal, and the initial focus was to create a very basic ray tracer engine. This embryonic ray tracer engine, at the time, didn't set out the distinction between the library itself and the rendering components. During the following months, these two goals were developed in parallel in order to bring new features, consolidating the three layers, increasing the compatibility with multiple devices and improve the rendering times.

So, in order to take advantage of most of the mobile CPU resources and give a good performance for the applications, the library and the rendering components were developed using the native programming language C++. This was achieved by using the Native Development Kit (NDK) provided by the Integrated Development Environment

(IDE) Android Studio. This way, the mobile device can provide better performance in computationally demanding applications because it doesn't need to use the Java garbage collector. It also facilitates the porting of existing C/C++ code to Android and promotes developing multi-platforms applications, and allows to bypass the hard limit on the Java heap size alloted for the app (Google (a)).

It is also important to mention that the UI was developed in Java using the traditional Software Development Kit (SDK) provided by the Android Studio, because there is no framework in the NDK that helps the programmer design his own UI natively. Despite that, the performance of the UI is less relevant because it doesn't interfere significantly with the others layers of the application.

3.4 LIBRARY

This section will describe the main functionalities of the bottom layer of the demo application. The figure 13 represents all 3 layers in the application and surrounds with a red rectangle the layer which will be described next.

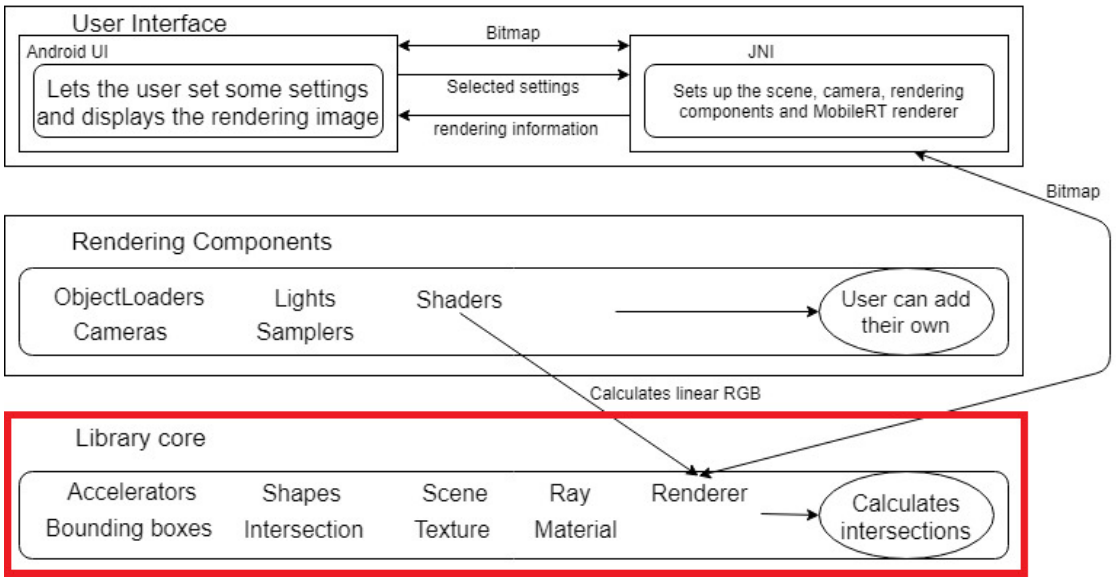


Figure 13.: Illustration of the three layers in the application.

As stated above, this library was implemented in an object-oriented fashion. The most important classes that provide functionalities already developed for the programmer to use are:

- Renderer: class that starts the rendering process and stores the calculated pixels colors in a C style array.

- Scene: class that stores the geometry data in C++ vectors and provides methods to cast rays to the lights in the scene.
- Shapes: set of classes that allows to create triangles, spheres and planes.
- Material: class that stores all four types of material color:
  - Emission light color
  - Diffuse reflection color
  - Specular reflection color
  - Specular refraction color
- Ray: class that represents a ray casted into the scene.
- Utils: source and header files which provide many constant values (ex: TwoPi, Epsilon) and many auxiliary functions available for the user.
- Texture: class that stores a bitmap in which the user can apply as texture to the scene primitives.
- Intersection: class that stores all the important data about an intersection of a ray with a primitive.
- AABB: class that represents a voxel which is aligned in an axis.
- Accelerators: set of classes with structures which reduce the number of ray primitive intersections.

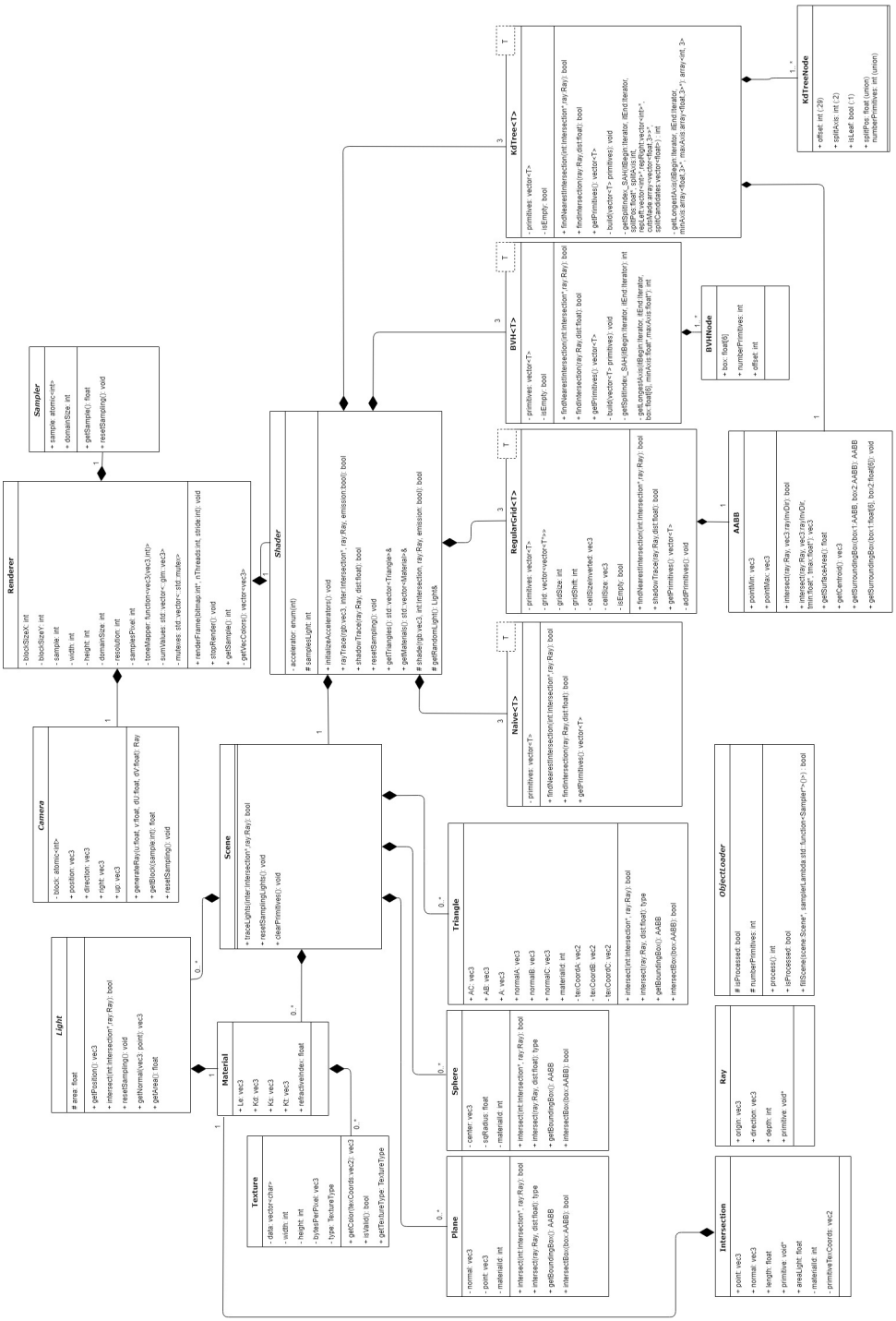


Figure 14.: Class diagram of the library.

The figure 14 shows the class diagram of the developed ray tracer library. Its worth to note that was followed two different approaches in different components of the library.

For example, the three shapes implemented (Plane, Sphere, Triangle) were developed in this layer. This means that a developer is restricted to only use these three types of shape and can't add other types. This restriction was necessary in this library because, as the intersect method in a shape can be called many times for each casted ray depending on the complexity of the geometry, it has a big impact in the overall performance of the application. So the strategy followed in this library was to implement each shape as an independent class (i.e., without inheritance) and making each acceleration structure C++ templates. This way, the Shader class has the acceleration structures as members, where it is used an acceleration structure for each type of shape.

On the other hand, the Camera, the Sampler, the Light, the Shader and the Objectloader are all abstract classes where the programmer needs to develop his own classes through inheritance. This allows more flexibility of how the programmer wants to render a particular scene. Making it possible, for example, to render a scene by simulating the paths of light or by simulating the propagation of sound.

3.4.1 *Third parties dependencies*

Before describing each functionality provided in each class developed, it is important to mention that this library uses 3 other libraries developed in C++ by third parties. Those libraries are:

- C++ Standard Library (group of C++ enthusiasts): used in order to save the data in popular containers like the array and the vector. It also provides some useful utilities like the unique pointers in the Dynamic memory management utilities, the String and the Atomic operations used in the Samplers.
- OpenGL Mathematics (Creation): used to create 3d points and vectors, perform geometry calculations and store pixels and primitive's colors. Before using this library, it was implemented an internal mathematics library and its performance was not so good, compared with GLM library.
- Google Test (Google (b)): used to create some unit tests and to mock some classes. This is important to have a safe and maintainable code.

All these libraries are Android compatible and reliable in terms of performance and maintenance.

Besides being dependent on these 3 libraries, the Android demo app also depends on OpenGL ES 2.0 (Inc. (b)) from Android SDK and depends on CMake (kitware) to compile the application.

The OpenGL ES 2.0 is necessary to let the application rasterize one frame of the scene so the user can view a raw preview of the scene without having to wait a long time to render it with the ray tracer. For example, in the Conference scene, the OpenGL in the Samsung Galaxy Fresh Duos can rasterize one frame in less than 17ms compared to 500ms on the NoShadows shader. This is done by copying the scene data from native code to the OpenGL of Java Virtual Machine (JVM) with the help of Java Native Interface (JNI).

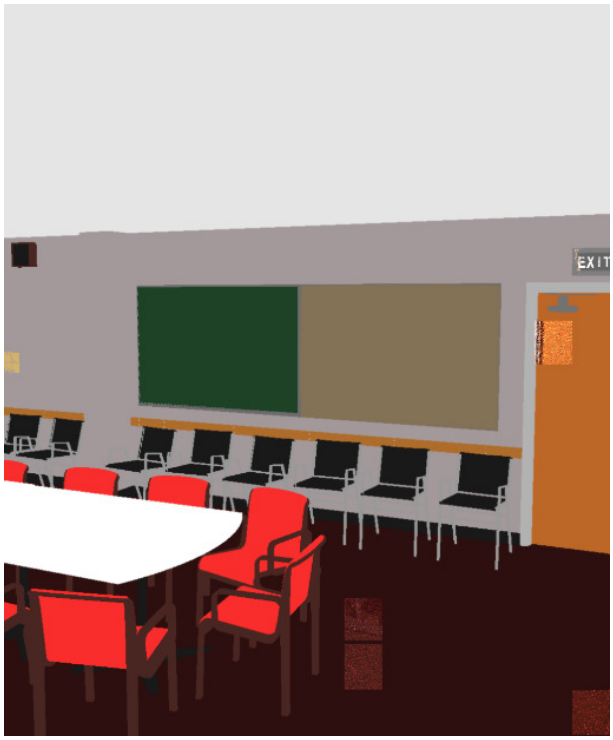


Figure 15.: The Conference scene rendered with OpenGL ES 2.0 before ray tracing the scene.

The CMake was used so that the application code could be compiled on any operating system. As it is a cross-platform family of tools designed to build, test and package software developed for a variety of operating systems like Windows, Linux, Mac OS X, and even FreeBSD. The main goal of CMake is to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of the programmer's choice. This facilitates the integration of third party libraries that use CMake to compile into our own project.

The NDK toolchain provides a C++ compiler compatible with features up to C++ 17, so developers can use the latest features provided in the language in order to code more efficiently and with better and new functionalities.

Last, but not least, it is important to mention that all the code of this application, including the ray tracer library, the rendering components and the graphical user interface

(GUI) are available on GitHub (Inc. (a)), which is a web-based hosting service for version control using Git. It is 100% open source and can be accessed on: https://github.com/PTPuscas/MobileRayTracer.

3.4.2 *Renderer*

The Renderer is the closest class to the application that starts the rendering process. This class provides 4 main methods:

```
1 void renderFrame(uint32_t *bitmap, int32_t numThreads, uint32_t stride) noexcept;
2 void stopRender() noexcept;
3 uint32_t getSample() noexcept;
4 std::vector<::glm::vec3> getVecColors() noexcept;
```

Listing 3.1: Main methods in Renderer

The *renderFrame* method starts the rendering process and writes the calculated light luminance of each pixel in the array parameter *bitmap*. This method also allows to choose the number of threads that will render the image into the bitmap, and needs the *stride* of that bitmap. The image plane is divided into 256 tiles of pixels and each thread asks the Sampler in the Renderer for the index of a tile and starts to render it completely. This allows the user to view the scene being rendered in different locations.

Figure 16.: Illustration of tiling the image plane.

The *stopRender* method stops the rendering process without resetting the pixels' colors already calculated.

The *getSample* method is used in order to get the current sample per pixel which the threads are rendering. This is useful to get an estimate of the remaining time to finish the rendering process.

Last, the *getVecColors* is a method to get the sum of the obtained colors of each sample in each pixel. This allows the developers to convert the obtained image to a custom format, for example store in 8 bit image or store in a 16 bit image.

3.4.3 *Scene*

The Scene is the class that handles the process of aggregating all the lights, primitives and the respective materials. The user should fill the C++ vectors inside this class with all the geometry data of the scene. Then, it is necessary to move the scene into a Shader, where it will build the desired acceleration structure and moves the primitives into that structure. This class provides 3 main methods:

```
1 bool traceLights(Intersection *intersection, const Ray &ray) const noexcept;
2 void resetSampling() noexcept;
3 void clearPrimitives() noexcept;
```

Listing 3.2: Main methods in Scene

The *traceLights*, as the name implies, is a method that tries to intersect a ray with all the light sources in the scene. It returns a boolean indicating whether that ray intersected a light or not. If a light was intersected, it writes the intersection data inside the intersected parameter.

The *resetSampling* method serves to reset all the samplers in the lights, so in the next run, these samplers will restart the sequence.

The *clearPrimitives* method serves to clear all the vectors with primitives.

3.4.4 *Ray*

The Ray is a very simple class, as it doesn't provide any particular method. It only stores the essential information of a ray, consisting of, origin, direction, depth and a pointer to the primitive from where it was casted. In order to cast a ray, it is needed, at least, an origin and a direction. The depth counts the number of bounces a ray performs in its path. To simulate a bounce, the programmer has to create a new ray, with its new direction and origin, and has to increment the depth counter. Lastly, the pointer serves to avoid shadow acne, which happens when the ray intersects the same primitive where it was casted from because of the limited precision of floating point operations. So, in summary, this class provides:

```
1 const glm::vec3 origin;
2 const glm::vec3 direction;
3 const int32_t depth;
4 const void *const primitive;
```

Listing 3.3: Main member variables in Ray

3.4.5 *Intersection*

The Intersection is another class which doesn't provide any particular method. It serves to pass the essential information of an intersection to the shade method in the shaders. This information is: the point of the intersection, the normal, the material of the intersected primitive, the distance of the intersection from the origin of the ray, a pointer to intersected primitive and the area of the intersected light. The area of the intersected light only have a value greater than zero if it intersected a light, and serves to let the developer calculate the solid angle of the intersected light. It also provides some hidden information, like the

material id and the texture coordinates, for the Shader to get the corresponding intersected material and deliver it to the shade method. So, the information it provides are:

```
glm::vec3 point;
glm::vec3 normal;
Material material;
float length;
const void *primitive;
float areaLight;
int materialId;
glm::vec2 primitiveTexCoords;
```

Listing 3.4: Main member variables in Intersection

3.4.6 *Material*

Lastly, the Material is another simple class that just serves to store the different components of the color of a material. In order to simulate reality, a material is composed of: light emission, diffuse reflection, specular reflection, and specular refraction with the respective refractive index. The light emission is the color which that material emits by itself without any light source around. The diffuse reflection represents the percentage of color which is reflected in all directions equally. The specular reflection is the percentage of color which is reflected in only one direction, like a mirror. And finally, the specular refraction represents the percentage of color which is refracted in only one direction, which depends on the refractive index. For example, in order to simulate a glass, the respective material should have zero light emission, a little bit of diffuse reflection, and an higher specular reflection and specular refraction colors, with a refractive index of around 1.5. This means that, the glass does not emit light, it's mostly transparent, it only reflects most of the light in one direction and refracts most of it in another direction.

This class also provides a C++ vector with textures in order to let the user load a 3D scene with multiple textures. Each texture have a type, like "DIFFUSE", "SPECULAR", "EMISSIVE" and "NORMAL". Each type of texture replaces the color of the corresponding component in the material. So, for example, if a user wants to put a light source with a texture in a scene, it is necessary to add the texture into this vector and the type of that texture should be "EMISSIVE". If this is done properly, then when a ray intersects that light, the rayTrace method in Shader will take care of replacing the light emission color by the one provided in the texture, using the corresponding calculated texture coordinates.

In summary, this class provides the following information:

```
glm::vec3 Le; //light emission
```

```
2 glm::vec3 Kd; //diffuse color
3 glm::vec3 Ks; //specular reflection color
4 glm::vec3 Kt; //specular refraction color
5 float refractiveIndex;
6 std::vector<Texture> textures;
```

Listing 3.5: Main methods in Scene

3.4.7 *Shapes*

In order to make possible to generate scenes with all kind of objects, it was developed 3 types of shapes: plane, sphere and triangle.

One way to allow this would be to use the process of inheritance available in C++. Which means, for example, create a class called Shape and create sub classes like Triangle, Plane or Sphere that are derived from that class. And each of those classes would implement different intersect methods. This way, it would be possible for the programmer to develop his own custom shapes as new rendering components. But of course, the benefit of this flexibility also brings a performance loss as the application will have at runtime to figure out which of the derived classes is required to call, because calling a virtual function requires a v-table lookup.

The other way to do it is by developing each shape class without inheritance and being each class independent of each other. Then, in order to call the intersection method of each class, it is required to call them separately. This means that it is required to have 3 C++ vectors, 1 for each type of shape. This way it is possible to avoid the performance loss from the v-table lookup at runtime. It provides better performance but it also restricts the possibility of adding new shapes as rendering components. As the mobile devices are not that powerful and limited in resources, the performance and battery life are big concerns. This ray tracer was implemented this way, because the intersect method can be called millions of times while rendering a complex scene. Also, as nowadays most of the scenes are made of triangles and these can be rearranged in order to make custom shapes, the user can always do that in any time.

So, all shapes provide 4 important methods:

```
1 bool intersect(Intersection *intersection, Ray ray) const noexcept;
2 bool intersect(Ray ray, float dist) const noexcept;
3 bool intersectBox(AABB box) const noexcept;
4 AABB getBoundingBox() const noexcept;
```

Listing 3.6: Main methods in Shape

The first *intersect* method determines whether a ray intersects the shape and returns a boolean indicating that. If the ray intersects the primitive, the corresponding intersection data is saved in the intersection parameter.

The second *intersect* is similar to the previous one but it just checks whether the ray intersected or not the shape, without having to calculate the intersection point. This method is used in the shadow trace method because it doesn't need the data about the intersection, as it just checks if a primitive is in a shade or not.

The *intersectBox* method checks if an AABB parameter contains the shape. This is useful to determine in which AABB the shape belongs to in the Regular Grid accelerator.

Finally, the *getBoundingBox* method, as the name implies returns the smallest AABB voxel that surrounds the shape. This is used in the building of the acceleration structures because it is simpler to intersect an AABB than a shape. Also, by combining these voxels, it allows to discard multiple shapes in one intersection call if multiple shapes are inside a larger AABB.

Beside sharing these methods, all shapes have a member variable called materialId, which is the index of the material of that shape. If all shapes have the index to the corresponding material, it allows to setup scenes with an unique material for multiple shapes. This is important because it allows a reduction in memory usage when different shapes have the same material.

Plane

The plane is an essential primitive shape because it allows the user to build indoor scenes with just 6 primitives (floor, walls, and ceiling). It can also be used for a simple background of a scene.

The construction of a plane requires just an arbitrary point in the plane and the normal of that plane. So, the intersect method implemented has the following algorithm taken from belthaczar:

Let

$$intersectionPoint = rayOrg + rayDir * dist \tag{1}$$

$$\overrightarrow{n} \cdot (q - x) = 0 \tag{2}$$

$$\begin{aligned} \overrightarrow{n} \cdot (q - intersectionPoint) &= 0 \\ \overrightarrow{n} \cdot (q - rayOrg - \overrightarrow{rayDir} * dist) &= 0 \\ \overrightarrow{n} \cdot (q - rayOrg) &= (\overrightarrow{n} \cdot \overrightarrow{rayDir}) * dist \\ dist &= [\overrightarrow{n} \cdot (q - rayOrg)] / (\overrightarrow{n} \cdot \overrightarrow{rayDir}) \end{aligned} \tag{3}$$

where

$\overrightarrow{n}$	: normal of the plane
q	: known point on the plane
x	: any point on the plane
$rayOrg$	: origin of the ray
$rayDir$	: direction of the ray
$dist$	: distance of the ray
$intersectionPoint$	: intersection point of the ray in the plane

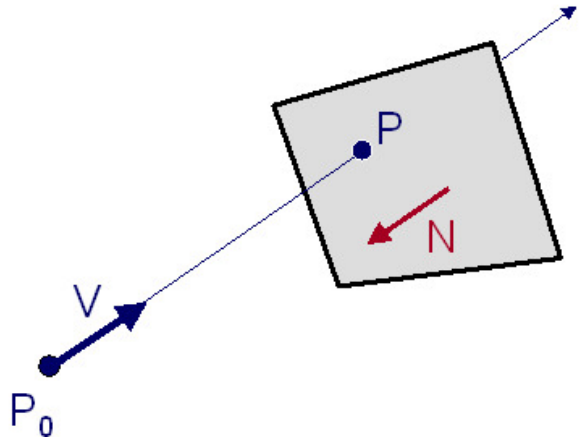


Figure 17.: Illustration of a ray intersecting a plane (Vink).

So, in order to intersect a ray with a plane, simply replace the arbitrary point in the plane equation with the ray equation.

Sphere

The sphere is another important primitive shape that allows the user to build some common objects with a shape of a ball. This shape is provided even if it's possible to imitate a sphere with a bunch of triangles, as intersecting a ray with a single sphere is always faster than intersecting it with hundreds or thousands of triangles.

The construction of a sphere requires just the point in the center and the radius of the sphere. It also should be noted that a ray can intersect a sphere at two points and therefore it is necessary to determine the closest intersection point to the origin of the ray. The intersection algorithm used in this library was written in an algebraic form, and is as following (Feminella):

Let

$$intersectionPoint = rayOrg + \overrightarrow{rayDir} * dist \tag{4}$$

$$(X - Cx)^2 + (Y - Cy)^2 + (Z - Cz)^2 - R^2 = 0 \tag{5}$$

$$\begin{aligned} &(rayOrg + \overrightarrow{rayDir} * dist - Cx)^2 \\ &+(rayOrg + \overrightarrow{rayDir} * dist - Cy)^2 \\ &+(rayOrg + \overrightarrow{rayDir} * dist - Cz)^2 - R^2 = 0 \\ &<=> a * dist^2 + b * dist + c = 0 \\ &<=> dist = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \end{aligned} \tag{6}$$

where

X, Y, Z	: are all the points in the sphere
C	: center point of the sphere
a	: $\|\overrightarrow{rayDir}\|^2$
b	: $2 * \overrightarrow{rayOrgToCenter} \cdot \overrightarrow{rayDir}$
c	: $\|\overrightarrow{rayOrgToCenter}\|^2 - R^2$
$\overrightarrow{rayOrgToCenter}$	: vector from the origin point of the ray to the center of the sphere
R	: radius of the sphere
$rayOrg$	: origin point of the ray
$\overrightarrow{rayDir}$	: direction of the ray
$dist$	: distance of the ray
$intersectionPoint$	: intersection point of the ray in the sphere

So, in order to intersect a ray with a sphere, it is only necessary to replace the arbitrary point of the sphere equation with the ray equation and solve the quadratic equation in order of the distance.

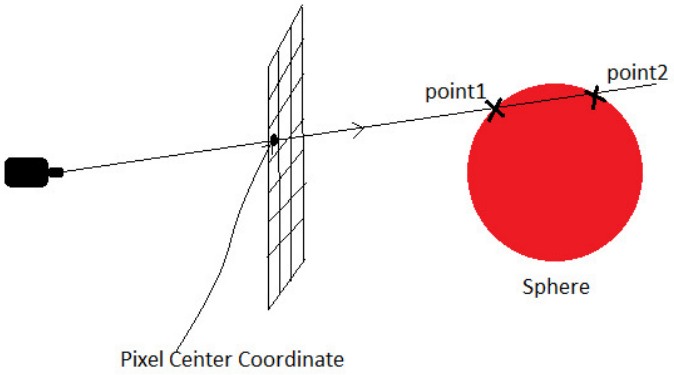


Figure 18.: Illustration of a ray intersecting a sphere in two points (Chirag).

Triangle

Finally, the last implemented shape is obviously the triangle, because, as it is the simplest primitive with an area, it allows to build many different object shapes. The implemented Ray-Triangle Intersection algorithm was the Möller-Trumbore (Moller and Trumbore (1997)) algorithm, which was named after its inventors Tomas Möller and Ben Trumbore.

The construction of a triangle requires 3 points $[A, B, C]$, 2 vectors $[AB, AC]$ and the normals of each vertex in the triangle. This is needed in order to execute the Möller-Trumbore algorithm for the intersection of a ray with a triangle. The Möller–Trumbore ray-triangle intersection algorithm is a fast method for calculating the intersection of a ray and a triangle in 3 dimensions without the need to precompute the plane equation containing the triangle. So, in order to intersect a ray with a triangle, it is just required to store 1 triangle vertex and 2 triangle vectors, making a total 9 floating point values, or 36 bytes.

As the intersection normal is important for calculating the incident radiance at each point, it is also required to store the normal of each vertex, making the total of $36 + 36 = 72$ bytes. Finally, as this shape allows the construction of many complex scenes, it was also added the possibility to have different texture coordinates for each vertex. Making the total of 6 glm::vec3 plus 3 glm::vec2 plus 1 integer for the index of a material, or 100 bytes. This class ends up containing a lot more data than the others shapes, but it is required if we want to render scenes with visual appealing properties.

Lastly, the implemented intersect method for the triangle has the following algorithm:
Let

$$intersectionPoint = rayOrg + \overrightarrow{rayDir} * dist \tag{7}$$

$$dist = (\overrightarrow{AC} \cdot ((rayOrg - A) \times \overrightarrow{AB}))/(\overrightarrow{AB} \cdot (\overrightarrow{rayDir} \times \overrightarrow{AC})) \tag{8}$$

where

A, B, C	: points of the triangle
AB	: vector of the triangle from point A to B
AC	: vector of the triangle from point A to C
$rayOrg$	: origin point of the ray
$rayDir$	: direction of the ray
$dist$	: distance of the ray
$intersectionPoint$	: intersection point of the ray in the triangle

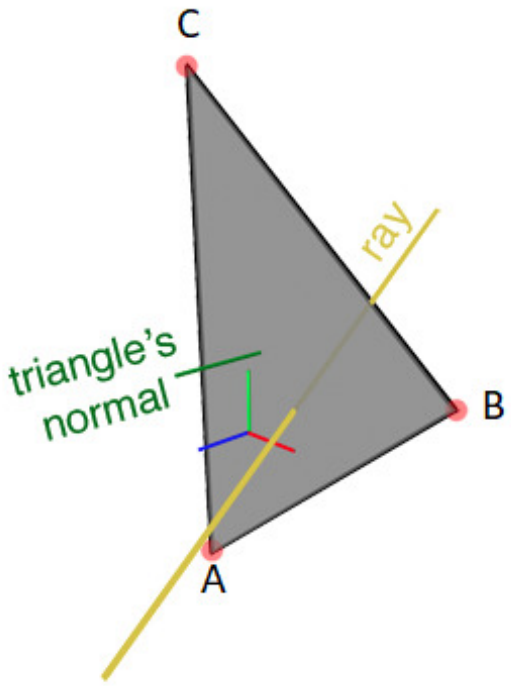


Figure 19.: Illustration of a ray intersecting a triangle (Scratchapixel).

3.4.8 *Acceleration Data Structures*

As previously stated, the rendering algorithms based on ray tracing can be very computationally demanding. Because, naively, for a scene with N primitives, it is necessary

to try to intersect every ray casted into the scene with all of the N primitives, as this algorithm has a complexity of O(N) in the Big O notation. So, for a scene composed with thousands or even millions of primitives, this task can be very time consuming and can waste a lot of battery power on the mobile device. Luckily, there are already known techniques, called acceleration data structures, which help to accelerate this process by reducing the number of intersections. Some of these structures can reduce the complexity of finding the nearest intersection of a ray from O(N) to $O(\log_2(N))$. This means, for example, for a scene with 1048576 primitives, it is possible to cast a ray and find the nearest intersection by just intersecting it with only 20 primitives.

There are 2 types of approaches that acceleration structures can have:

- Subdivision of Space: the 3D space of the scene is divided into smaller portions of space in which, each voxel can be uniform or irregular:
  - Regular Grids
  - Octrees
  - Kd-trees
- Subdivision of Objects: the 3D primitives are aggregated in groups inside voxels:
  - Bounding Volume Hierarchy (BVH)

The idea in Subdivision of Space is allowing the intersections to start with the nearest primitives of the ray and is only intersected with the further ones, if the closest ones are not in the same direction of the ray. Besides trying to intersect the ray with the nearest primitives first, it also makes it possible to reject simultaneously multiple primitives.

In contrast, structures based on Subdivision of objects only allow the rapid and simultaneous rejection of groups of primitives, because it doesn't take into account the direction of the ray.

The developed library provides 3 types of structures: the Regular Grid, the Kd-tree, and the Bounding Volume Hierarchy (BVH).

As the performance is a concern in this project, all of these developed data structures are class templates, instead of using the traditional class hierarchy. Also, as mentioned before, the shapes are classes without any hierarchy shared. This allows to build different acceleration data structures for each type of shape. So, for example, it is possible to put all the triangles of the scene in a BVH, all the spheres in a Regular Grid and all the planes in a Kd-tree. Or even, put all types of primitives in 3 Regular Grids with different sizes. It therefore allows a variety of combinations.

Finally, it is important to mention that all these data structures are built basing on Axis Aligned Bounding Box (AABB), which are simple volume elements (voxels) that can cover, one or more primitives from the 3D scene. An AABB is a voxel, like the name implies,

that is aligned to the axis of the scene. This type of voxels are great to use in acceleration structures because it is very fast to test whether or not a ray intersects it (Barnes) and it only needs to store in memory 2 points of that voxel.

All the acceleration structures provided in the MobileRT library supply the following methods:

```
1 bool findNearestIntersection(Intersection *intersection, Ray ray) const noexcept;
2 bool findIntersection(Ray ray, float dist) const noexcept;
3 std::vector<T> getPrimitives() const noexcept;
```

Listing 3.7: Main methods in Acceleration Structures

The method *findNearestIntersection* receives a pointer to an intersection and a ray, and finds the nearest intersection of that casted ray by using the correspondent acceleration structure. It will return a boolean telling if it found an intersection and all the data of this intersection is passed to the *intersection* parameter.

The second method is similar to the first one, but it will return on the first intersection found and it doesn't need to pass the intersection data to the parameter. This method is useful when casting shadow rays, as these rays only need to know if they didn't intersect anything between their origin and a light.

The last method is a method to just get the primitives. It is useful when the developer wants to access the geometry and do something with it, like, for example, pass it to the OpenGL in order to rasterize one frame.

Regular Grid

A Regular Grid is an acceleration structure where the scene space is subdivided into equal voxels. Each voxel can have inside one or more primitives of the scene, if a ray intersects that voxel, then it tries to intersect the primitives which it contains. The order of testing each voxel is from the nearest of the ray into the furthest in the direction of that ray.

This type of structure is very fast to build but its traversal is poorly efficient due to poor distribution of primitives by voxels. It can not resolve efficiently the "teapot in the stadium problem", because in sparse scenes, the casted ray will always be checked against many empty cells. Or even worse, as most of the scene's primitives can be inside of just a few voxels. Also, one primitive may be contained in different voxels, which can make the ray intersect it multiple times.

The figure 20 shows the process of finding the nearest intersection of a ray in a scene. The process starts by choosing the nearest voxel of the ray. Then, it tries to intersect the ray with all the primitives inside. If it intersects a primitive, then that intersection is the nearest and the algorithm stops. Otherwise, it starts to stepping through the grid in the same direction as the ray, and checking each cell's primitives.

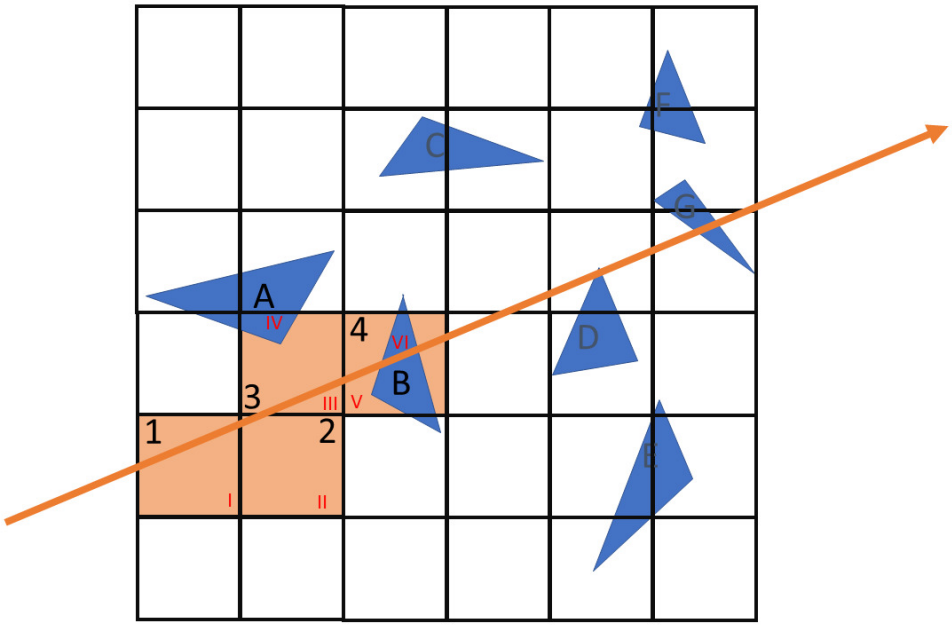


Figure 20.: Illustration of traversing a Regular Grid acceleration structure.
The roman letters indicate the navigation order in the grid.

The developed ray tracer library provides a Regular Grid as an acceleration structure and its trace operation is as shown in the figure 20.

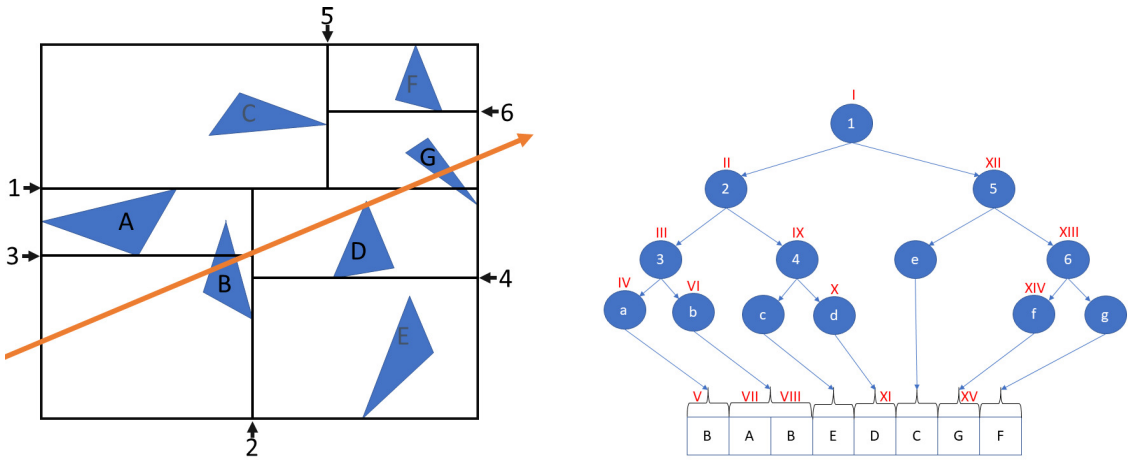
Kd-tree

This library also provides a Kd-tree structure. This structure divides the scene recursively through various cuts made by planes. The calculation of these cutting planes is given by the Surface Area Heuristic (SAH) algorithm which is a very popular heuristic commonly used in ray tracers.

The root of the tree corresponds to an AABB which surrounds the whole scene. Then, each interior node represents the cuts by those planes that recursively subdivide space perpendicular to a coordinate axis. And finally each leaf node stores the indexes to all the primitives overlapping the corresponding voxel.

It is important to mention that, like the Regular Grid, this data structure can contain repeated primitives, since when splitting the scene through a plane, it can cut a primitive in the middle and so making it belong to both sides of the plane.

The figure 21 illustrates the whole process of finding the nearest intersection of the ray, in the same example as before.



(a) Scene divided by Kd-tree

(b) Traversing the Kd-tree

Figure 21.: Illustration of traversing of a Kd-tree acceleration structure.

The process starts by intersecting the ray with an AABB which covers all the primitives of the scene. If, it doesn't intersect that AABB, then the algorithm stops and it didn't find any intersection. If, it does intersect, then starts to step through the binary tree structure, as shown in the right figure. The roman letters indicate the order of visit at each node of the tree.

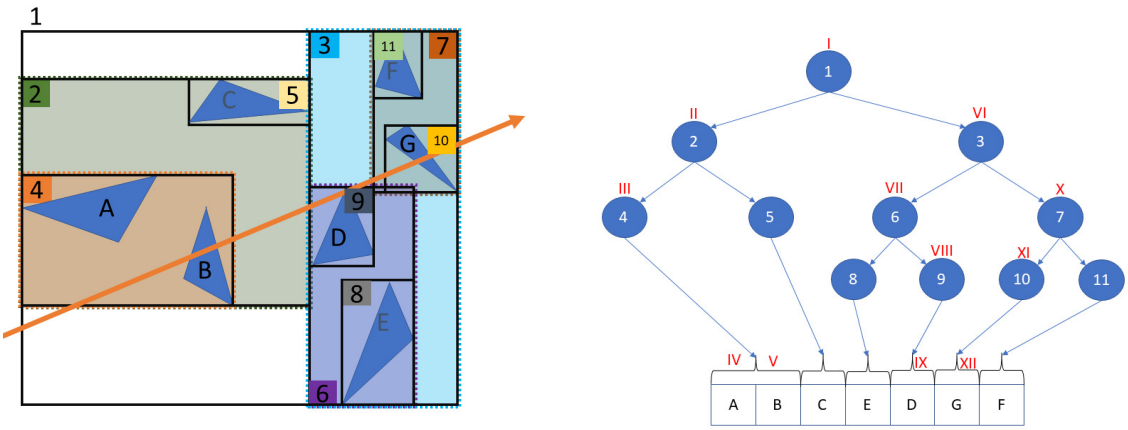
In the literature, usually, this data structure is the one which reduces the most the rendering time for scenes with a huge amount of primitives (Vinkler et al.). Because it splits the scene in many irregular voxels and by using the ray's origin and direction, it can start by intersecting the ray with the nearest primitives and then discard the furthest ones. However, the current state of this data structure in the developed library is not yet on pair with the current Bounding Volume Hierarchy implementation in MobileRT, as it will be seen in chapter 6.

Bounding Volume Hierarchy

Lastly, the Bounding Volume Hierarchy (BVH) is another acceleration structure provided in the developed library. This structure is different from the Regular Grid and Kd-tree because the space is not subdivided into many voxels. The primitives are themselves grouped in voxels, allowing to build a structure where each voxel contains, for sure, one or more primitives inside.

The figure 22 shows an example of a BVH structure of the previous scene. As it can be seen, each primitive is covered by an AABB, and then two or more AABBs are covered by one larger AABB. In this library, this is built in a top down manner, where all the primitives in the scene are grouped into one AABB. Then all the primitives in that AABB are divided in two AABB where some of them will be covered by one AABB and the rest of them by the

other. As the Kd-tree, the calculation of those subdivisions is performed with the Surface Area Heuristic (SAH) algorithm which is a very popular heuristic commonly used in ray tracers (Lahodiuk).



(a) Scene divided by BVH

(b) Traversing the BVH

Figure 22.: Illustration of traversing a Bounding Volume Hierarchy acceleration structure.

This structure has an advantage and a disadvantage over the other two acceleration structures. The vector of primitives does not contain repetitive elements. This is great because it doesn't waste memory with repeated primitives, or with pointers to primitives. The downside is that, while visiting the tree structure, the ray intersection must always be checked to both children nodes.

Surface Area Heuristic

As it was said above, the Kd-tree and the BVH in this library use the Surface Area Heuristic (SAH) in order to split the scene. The algorithm used is very simple, it just needs to calculate the minimum area surface of the sum of the left and the right sub-trees, as following:

$$SAH_{optimal} = min(S_L * N_L + S_R * N_R) \tag{9}$$

where

N_L = number of primitives in left subtree
N_R = number of primitives in right subtree
S_L = surface area of left subtree
S_R = surface area of right subtree

The figure 23 shows a very simple example of how the SAH algorithm works.

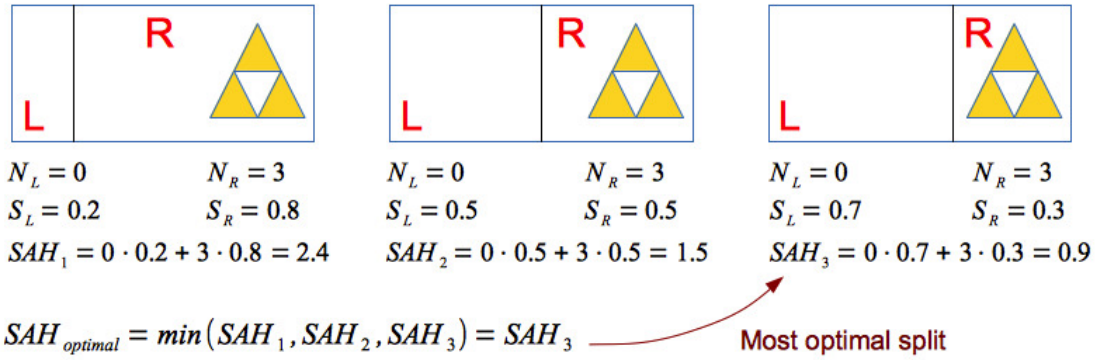


Figure 23.: Illustration of how Surface Area Heuristic works in Bounding Volume Hierarchy acceleration structure (Lahodiuk).

3.4.9 *Texture*

In Computer Graphics, for a scene to get a fair bit of realism, it needs to have many primitives with many vertices. This takes up a considerable amount of extra overhead, since each casted ray needs to intersect many more primitives. What artists and programmers generally prefer is to use a texture (de Vries). A texture is a 2D image used to add detail to an object without needing to add extra vertices to it.

The developed library also provides support for applying textures in the models of the scene. The figure 24 illustrates a scene rendered with this library, which contains many textures applied to the geometry. It is noteworthy that the rendered scene is the San Miguel model released with the PBRT book. This scene contains almost 10 million triangles and uses 287 different materials with textures, making it use the total of 3.6 GB of RAM memory, with the laptop W230SS referenced in the table 3.

The Texture class of this library is very simple and just stores the data information of the bitmap in a C++ vector along with the width, height and the number of bytes per pixel. It is important to mention that this library supports various types of textures like: "NOT_VALID", "DIFFUSE", "SPECULAR", "EMISSIVE" and "NORMAL". Each type means that, if the material has a texture, the correspondent component color of the material will be replaced in the library before calling the shade method of the developed shader by the programmer. For example, if a material have a color in the diffuse component and has a texture of type "DIFFUSE", then when a ray intersects a primitive with that material, the library will replace the diffuse color of that material by the texture "DIFFUSE" using the texture coordinates calculated at the intersection. The same applies to the "SPECULAR" and "EMISSIVE" types of textures. The "NORMAL" type of texture means that the calculated intersection normal will be replaced by a value read on a texture. And

finally, the "NOT_VALID" means that the texture was not loaded properly, i.e., couldn't open the texture file or the texture file was not a bitmap with data of 1 or 3 bytes per pixel.

The main methods provided for the programmer are:

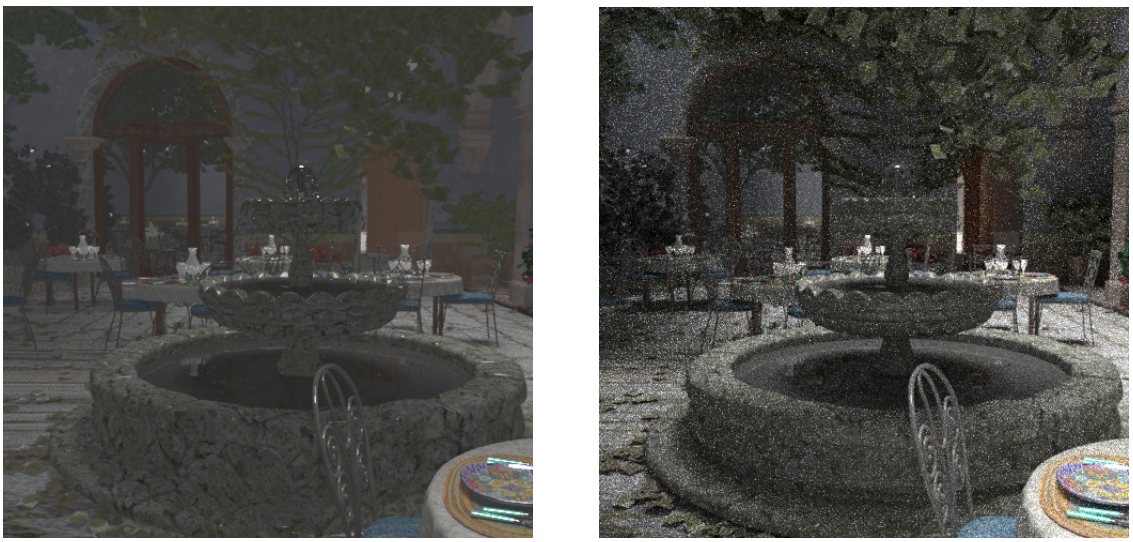
```
1 glm::vec3 getColor(const glm::vec2 &texCoords) const noexcept;
2 bool isValid () const noexcept;
3 TextureType getTextureType () const noexcept;
```

Listing 3.8: Main methods in Texture

The method *getColor* calculates the color of the bitmap for a specific texture coordinates. The texture coordinate needs to be in the range of 0 to 1. This method only works with bitmaps where the information for each pixel is contained in 1 or 3 bytes.

The method *isValid* checks whether the texture is valid, i.e., it was loaded properly from the file into the texture data.

Last, the method *getTextureType* returns the type of the texture loaded. This is useful for the class Shader, in order to replace the intersected material color by the correspondent texture.



(a) Whitted w/ 100 spp & 1 spl at 496x496 (b) Path Tracing w/ 100 spp & 1 spl at 496x496

Figure 24.: Illustration of San Miguel scene with many textures used.

3.4.10 *Utilities*

Besides all these classes, this library also provides a header file with a panoply of auxiliary methods and constants for the programmer.

A constant is a value that cannot be altered by the program during normal execution (Wikipedia (b)). Usually, the programmer gives a name to the constant which describes what it intends to mean, so by reading the code anyone can understand. This library provides the following constants, and with the respective values set:

RayLengthMin: The minimum length that a ray must travel ($1e-5$)

RayLengthMax: The maximum length that a ray can travel (maximum value of float from C++ stdlib)

Epsilon: The smallest difference between two floating-point numbers (machine epsilon)

Pi: The pi value (3.14 ...)

TwoPi: The 2*pi value

QuarterPi: The pi/4 value

ShadowBias: A small bias to be applied to a ray origin in order to avoid shadow acne ($1e-4$)

RayDepthMin: The minimum depth that a ray must travel (1)

RayDepthMax: The maximum depth that a ray can travel (6)

NumberOfBlocks: The number of blocks which the image plane will be divided (256)

NumberOfAxis: The number of axis in which the scene is composed (3)

StackSizeBVH: The size of the stack used in the $findNearestIntersection$ and $findIntersection$ methods in the BVH accelerator

StackSizeKDtree: The size of the stack used in the $findNearestIntersection$ and $findIntersection$ methods in the KdTree accelerator

Besides the constants, this library also provides many helper functions that aim to aid the programmer's needs. The provided functions are then:

LOG: Is a variadic function which converts all the arguments into a single string and prints it in the console (works with the Standard output and on Android)

createPointerVector: Receives a vector and creates another vector with pointers from the received vector

equals: Checks if 2 floating-point or glm::vec3 values are equal

roundDownToMultipleOf: Receives 2 integer values and calculates the nearest number of the first argument which is multiple of the second argument

roundUpToPowerOf2: Calculates the nearest power of 2 value for an arbitrary number

usedBitsCounter: Counts the number of bits set to 1 in an unsigned value

haltonSequence: Gives the n^{th} halton sequence value for an arbitrary index and base

toneMap: Receives a glm::vec3 value which represents the sum of radiance of all samples per pixel, receives the number of samples taken and the gamma value to apply, and calculates the average of a vec3 rgb values with the number of samples used, and applies a gamma correction to it

convertVec3ToIntColor: Receives a glm::vec3 value which represents the average color of a pixel (between 0 and 1) and converts it to an integer (between 0 and 255)

balanceHeuristic: Calculates the balance heuristic of 2 PDFs (essential to perform MIS)

powerHeuristic: Calculates the power heuristic of 2 PDFs (essential to perform MIS)

getCosineWeightedHemisphereSample: Generates a random point in a cosine weighted hemisphere and rotates it towards the received direction

fresnelEquation: Calculates the Fresnel equation, i.e., for an incident and normal vectors, calculates the percentage of reflected and refracted light

createTextureFromFile: Receives a path to a bitmap file and the type of texture it represents, and creates a Texture with the data provided in that file

degToRad: Converts an angle in degrees to radians

radToDeg: Converts an angle in radians to degrees

sumBox: Receives 2 boxes of type AABB and calculates a box which surrounds both boxes

getBounds: Receives a vector of primitives and returns a box which surrounds all those primitives

getArcTan: Calculates the arctangent of an angle in radians

4

SOFTWARE ARCHITECTURE - RENDERING COMPONENTS

As stated before, the developed library allows some flexibility for the programmer. It allows to develop customized cameras, lights, samplers, shaders and objectloaders. This is important to let the programmer specify how he wants to load a scene and how he wants to render it. Making it possible, for example, to setup a renderer to render a photo realistic image or to simulate a wifi network.

So, in order to show the functionalities provided by the library, it was also developed a few Rendering Components. It was developed 2 types of camera, 2 types of light, 4 samplers, 5 shaders and 1 object loader.

As the implemented ray tracer was programmed in an objected oriented fashion, each Rendering Component was developed separately, in a different class. This allows the user to develop his own Rendering Components without having to develop the ray tracer engine.

This chapter will describe the main functionalities of the middle layer of the demo application. The figure 25 represents all 3 layers in the application and surrounds with a red rectangle the layer which will be described next.

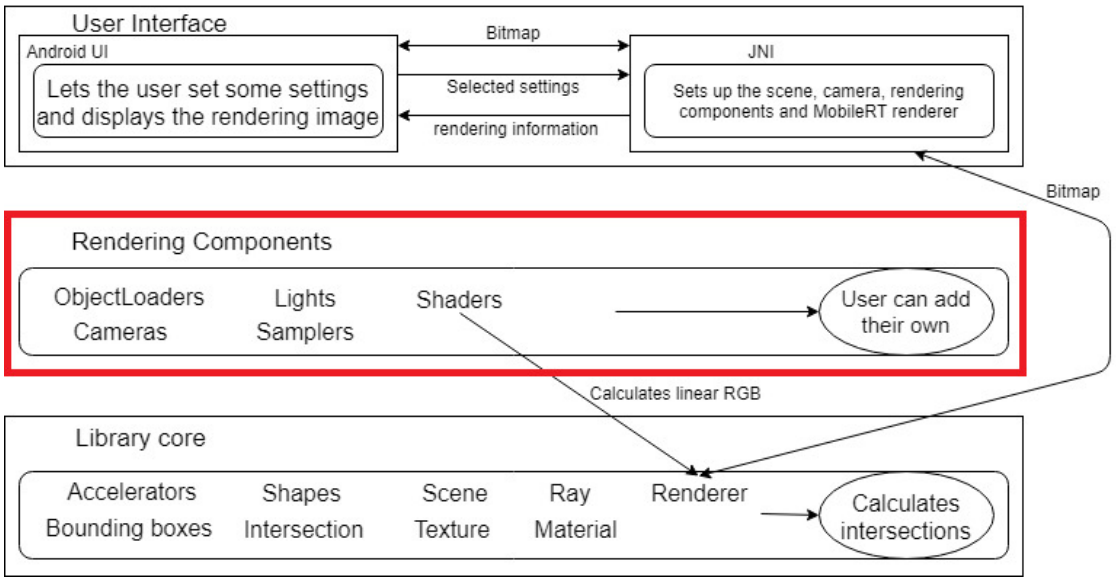


Figure 25.: Illustration of the three layers in the application.

4.1 SHADERS

A shader is the most important Rendering Component because it describes how to calculate the result of casting a ray in a certain point from the camera. This result can be described as a color, or can have a different meaning that the programmer wants.

All of the following shaders were implemented in order to get an image, so each ray casted from a camera returns a color of that pixel. Basically, a shader describes how the Rendering Equation is approximated. The Rendering Equation describes how the total radiance reflected by any point x of a surface in a direction ωo is calculated. The rendering equation may be written in the form:

$$L(x \rightarrow \omega_o) = L_e(x \rightarrow \omega_o) + \int_{\Omega_s} f_r(x, \omega_i \leftrightarrow \omega_o) L(x \leftarrow \omega_i) \cos(\overrightarrow{N_x}, \omega_i)\, d\omega_i \tag{10}$$

where

$L_e(x \rightarrow \omega_o)$: is the emitted spectral radiance in direction ω_o
$f_r(x, \omega_i \leftrightarrow \omega_o)$: is the proportion of light reflected from ω_i to ω_o at position x, also known as bidirectional reflectance distribution function (BRDF)
$L(x \leftarrow \omega_i)$: is the spectral radiance coming inward toward x from direction ω_i
$\cos(\overrightarrow{N_x}, \omega_i)$: is the weakening factor of outward radiance due to incident angle

The figure 26 illustrates the different components of the rendering equation.

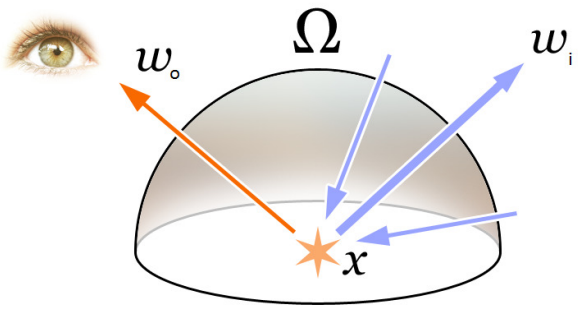


Figure 26.: Illustration of the rendering equation describing the total amount of light emitted from a point x along a particular viewing direction and given a function for incoming light and a BRDF (Wikipedia (g)).

In summary, a shader, in this context, tries to solve the rendering equation with a numerical solution.

So, it was developed 5 shaders: NoShadows, Whitted, PathTracer, DepthMap and DiffuseMaterial.

The PathTracer is the only shader which tries to solve the rendering equation taking into account mathematical and physical formulas that describe reality. While the NoShadows and Whitted are simpler approximations to solve the rendering equation more efficiently.

Finally the DepthMap and the DiffuseMaterial are shaders whose purpose is only to help debug the library code.

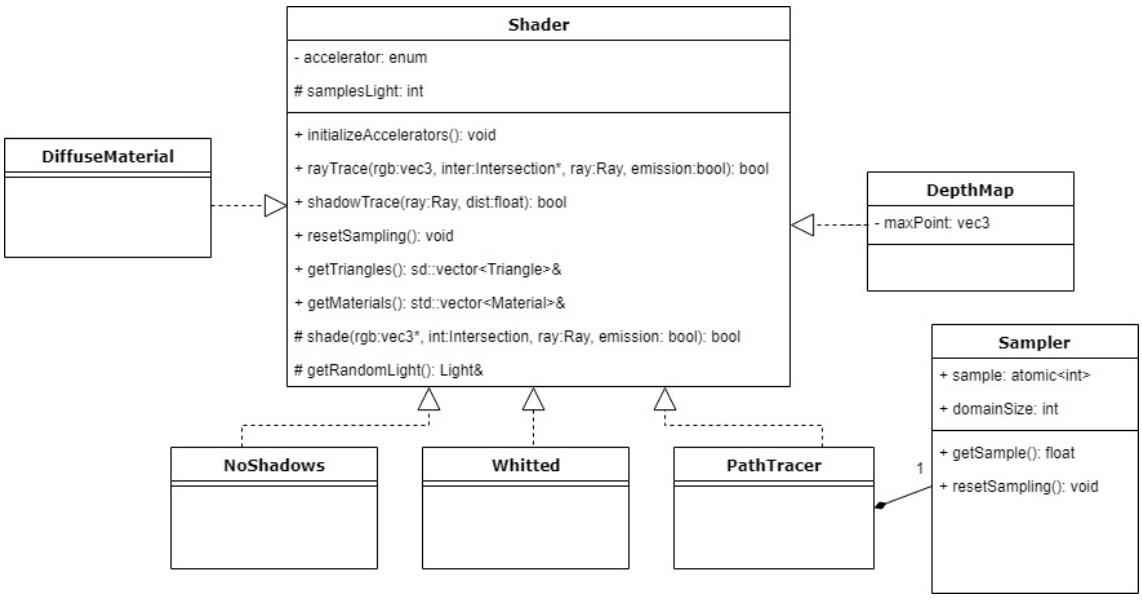


Figure 27.: Class diagram of the Shaders.

To write a shader for this library, the user must implement the *shade* method. This method receives a parameter called *rgb* which is a pointer to where the user must

write the calculated radiance for a pixel. For that calculation, the method also receives the *intersection*, *ray* and *emission* parameters. The *intersection* parameter has all the information about the intersection and the intersected material, while the *ray* parameter contains just the information about the casted ray. Finally, the *emission* parameter serves to specify if the caller wants to add the emission from the light or not, which is useful when developing a recursive shader. It is up to the user to implement their shading algorithm in order to paint the pixels.

Besides that method, the class Shader provides 3 other important methods that all shaders can have access to:

```
1 virtual bool shade(glm::vec3 *rgb, Intersection intersection, Ray ray, bool emission) noexcept = 0;
2 bool rayTrace(glm::vec3 *rgb, Intersection *intersection, Ray ray, bool emission) const noexcept;
3 bool shadowTrace(Ray ray, float dist) const noexcept;
4 Light &getRandomLight () const noexcept;
```

Listing 4.1: Main methods in Shader

The *rayTrace* is a method that calls the $findNearestIntersection$ method of the selected acceleration structure. Its purpose is to find the nearest intersection of the parameter *ray*. After calculating the nearest intersection, it gets the corresponding intersected material and calls back the *shade* method, allowing the possibility to implement some complex shaders where a pixel may need information of other primitives instead of the one it covers.

The *shadowTrace* method is a little different than the former because it just returns whether a given ray with a given maximum distance intersects any primitive in the scene or not. This method is useful, for example, when casting rays to the lights in order to check if a point in the scene is in shade or not.

Lastly, the *getRandomLight* method returns a random light from the scene. To select the light, it uses a random value generated by a C++ generator called Mersenne Twister which uses an uniform distribution. This method is important because it allows the implemented shader to randomly sample a light instead of having to sample all the lights, and thus having a better performance for scenes with many lights.

4.1.1 *DepthMap*

DepthMap is a very simple shader that calculates an image or image channel that contains information relating to the distance of the surfaces of scene objects from a viewpoint (Wikipedia (c)). In other words, it builds an image where each pixel represents the distance from the camera to the corresponding intersected primitive. Where, the brightest pixels represents closer primitives and the darkest pixels represents the farthest. It is useful to

various things, like simulating fog, smoke or large volumes of water; or even to create Shadow Maps.

The algorithm is as much as simple as:

Algorithm 1: Algorithm of DepthMap Shader.

input : `furthestPoint`, `ray`, `intersection`
output : `outputRGB`

1 `maxDist` ←distance from `ray.origin` to `furthestPoint`
2 `invDist` ← `maxDist` − `intersection.length`
3 `depth` ← normalize `invDist` by dividing it by `maxDist`
4 `outputRGB` ← [`depth`, `depth`, `depth`]

where

furthestPoint : is the furthest 3D point that the user set as parameter in the DepthMap constructor

ray : contains some information about the casted ray, like its origin, direction and depth

intersection : contains some information about the intersection, like its point, normal, distance from ray origin and material of the intersected primitive

outputRGB : the output parameter to write the pixel's RGB color

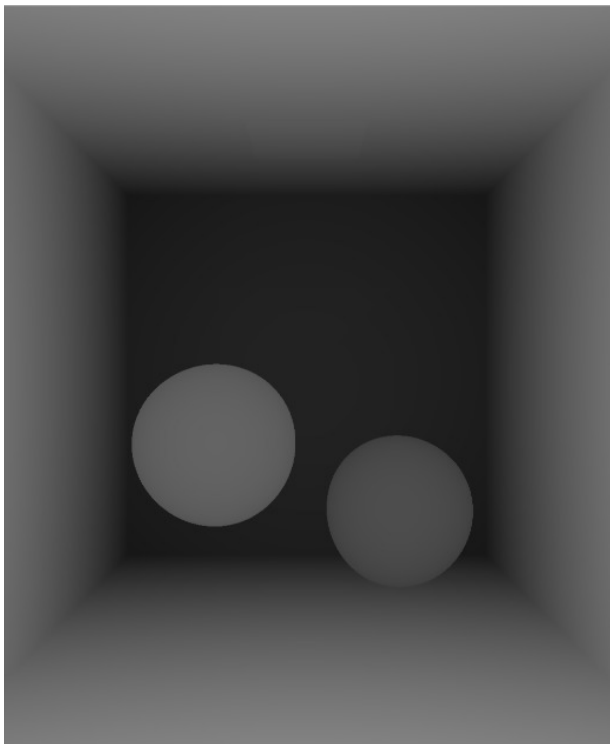


Figure 28.: DepthMap shader.

4.1.2 *DiffuseMaterial*

The DiffuseMaterial is another very simple shader that only outputs the material's color of the intersected primitive. It starts by checking whether the material has diffuse color, and if it doesn't then tries the other 3 colors. This can be useful to debug a ray tracer because it renders the scene, including all types of materials, very fast. And the algorithm is very simple as illustrated below:

Algorithm 2: Algorithm of DiffuseMaterial Shader.

```
   input  : ray, intersection
   output : outputRGB
1  if intersected material is diffuse then
2  |  outputRGB ← kD
3  else if intersected material is specular reflective then
4  |  outputRGB ← kS
5  else if intersected material is specular refractive then
6  |  outputRGB ← kT
7  else if intersected material is a light source then
8  |  outputRGB ← Le
```

where

ray : contains some information about the casted ray, like its origin, direction and depth

intersection : contains some information about the intersection, like its point, normal, distance from ray origin and material of the intersected primitive

outputRGB : the output parameter to write the pixel's RGB color

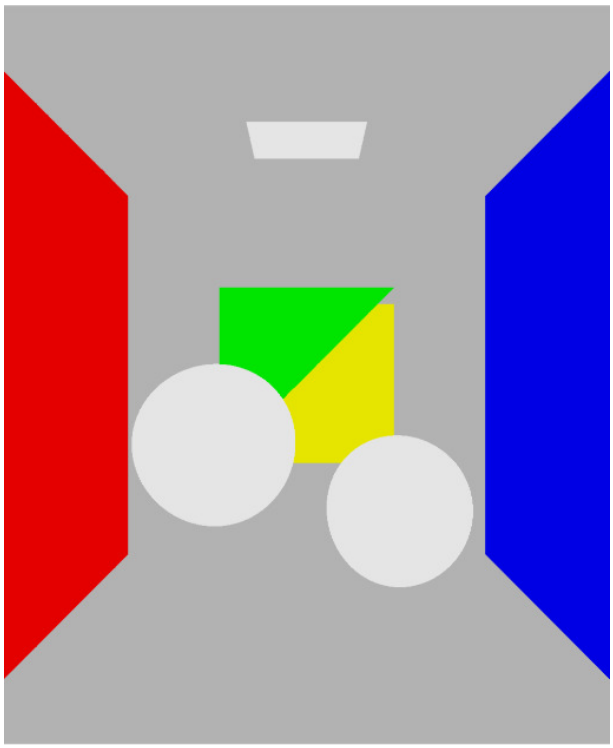


Figure 29.: DiffuseMaterial shader.

4.1.3 *NoShadows*

NoShadows is a simple shader because, as the name implies, it does not synthesize the shadows and it only simulates the direct lighting. It simulates direct lighting in primitives with diffuse surfaces and it does not take into account the light reflected by the other primitives. The indirect lighting is approximated with a fixed ambient color of about 10% of the diffuse color of the primitives. This can be useful in order to let the user render a scene and check if a primitive is directly illuminated or not, without having to wait too much time.

So, the developed algorithm is just as following:

Algorithm 3: Algorithm of NoShadows Shader.

```
input  : ray, intersection
output : outputRGB
if intersected material is a light source then
    outputRGB ← kD
else if intersected material is diffuse then
    foreach sampleLight do
        Choose a random light
        Choose a random position in area light
        Calculate vector to light
        if intersected surface is facing towards the light then
            cosNL ← vectorToLight · intersectionNormal
            outputRGB ← outputRGB + (lightRadiance × cosNL)
    outputRGB ← outputRGB/kD
    outputRGB ← outputRGB/#samplesLight
outputRGB ← outputRGB + kD × 0.1 // ambient light
```

where

ray : contains some information about the casted ray, like its origin, direction and depth

intersection : contains some information about the intersection, like its point, normal, distance from ray origin and material of the intersected primitive

outputRGB : the output parameter to write the pixel's RGB color

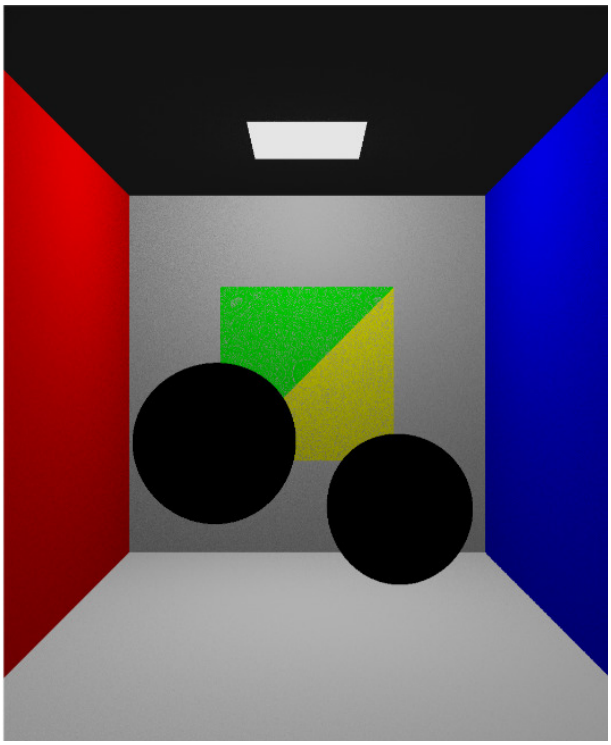


Figure 30.: NoShadows shader.

4.1.4 *Whitted*

As the name of this shader implies, Whitted is the algorithm presented by John Turner Whitted in the 1980s (dos Santos). Unlike the previous shader, it simulates indirect lighting, but only on specular reflective surfaces and on specular transmission surfaces.

This algorithm, in each intersection point on diffuse surfaces, samples a random point in a random light with shadow rays, in order to synthesize shadows without causing the banding effect.

It also recursively performs ray casting in order to do transmission and reflection on specular surfaces.

The direction of the reflected light in specular surfaces are given by the next formula (Inc. (b)):

$$\overrightarrow{reflectionDir} = \overrightarrow{I} - 2 \times (\overrightarrow{N} \cdot \overrightarrow{I}) \times \overrightarrow{N} \tag{11}$$

where

$\overrightarrow{I}$: is the incident vector at the intersected primitive
$\overrightarrow{N}$: is the normal vector of the intersected primitive

Finally, the refracted light in a transmission surface is calculated by using the refractive index of the intersected primitive, the incident direction and the intersection normal. So, the specular refraction is given by the formula 13 (Inc. (b)):

$$k = 1 - eta \times eta \times (1 - (\overrightarrow{N} \cdot \overrightarrow{I}) \times (\overrightarrow{N} \cdot \overrightarrow{I})) \tag{12}$$

$$R = \begin{cases} eta \times \overrightarrow{I} - (eta \times (\overrightarrow{N} \cdot \overrightarrow{I}) + \sqrt{k}) \times \overrightarrow{N}, & \text{if } k \geq 0 \\ [0,0,0], & \text{otherwise} \end{cases} \tag{13}$$

where

$\overrightarrow{I}$: is the incident vector at the intersected primitive
$\overrightarrow{N}$: is the normal vector of the intersected primitive
eta : is the ratio of indices of refraction

To simulate shadows on diffuse surfaces, plus reflective and refractive surfaces, the algorithm was divided into each case. So it makes it possible to simulate refractive surfaces, reflective surfaces and even surfaces with both components, and with shadows.

And so, the whole developed Whitted algorithm is as following:

Algorithm 4: Algorithm of Whitted Shader.

```
input  : ray, intersection
output : outputRGB
if Ray didn't reached maximum depth then
    if intersected material is a light source then
        if Ray hits the light on the front then
            outputRGB ← Le
        else
            outputRGB ← 0
    else
        if intersected material is diffuse then
            foreach sampleLight do
                Choose a random light
                Choose a random position in area light
                L⃗ ← Calculate vector to light
                if intersected surface is facing towards the light then
                    Cast shadow ray
                    if primitive is not in shadow then
                        theta ← ∠(N⃗, L⃗)
                        lightSample ← (kD × radLight × cos(theta)) ÷ #samplesLight
                        outputRGB ← outputRGB + lightSample
        destIOR ← refractive index of intersected material
        sourceIOR ← refractive index of previous material
        kr ← calculate fresnel equation
        kt ← 1 − kr
        if intersected material is specular reflective then
            R⃗ ← Calculate reflection direction
            Cast a specular ray
            radIncident ← rayTrace(specularRay)
            ∠theta ← ∠(N⃗, R⃗)
            specularSample ← kS × radIncident × cos(theta)
            outputRGB ← outputRGB + kr × specularSample
        if intersected material is specular refractive then
            eta ← sourceIOR/destIOR
            R⃗ ← Calculate refraction direction
            Cast a specular ray
            radIncident ← rayTrace(transmissionRay)
            ∠theta ← R⃗ · N⃗
            specularSample ← kT × radIncident × cos(theta)
            outputRGB ← outputRGB + kt × specularSample
        outputRGB ← outputRGB + kD × 0.1// ambient light
```

where

ray : contains some information about the casted ray, like its origin, direction and depth

intersection : contains some information about the intersection, like its point, normal, distance from ray origin and material of the intersected primitive

outputRGB : the output parameter to write the pixel's RGB color

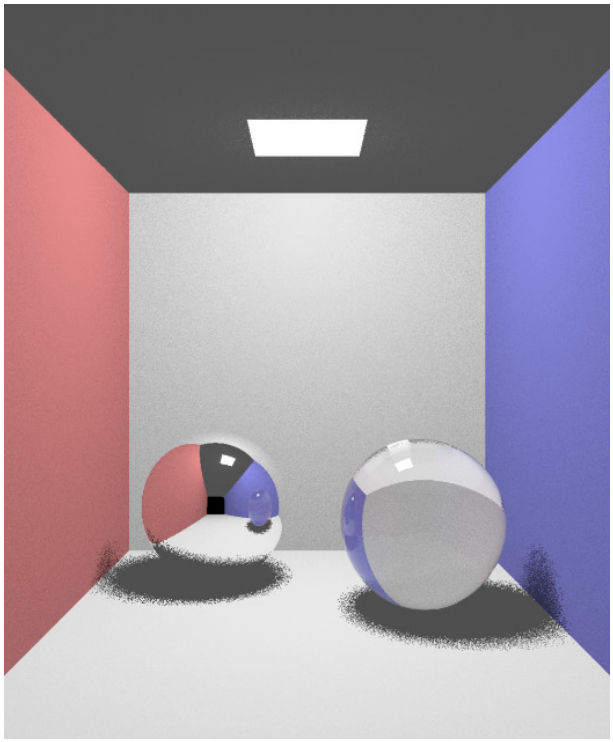


Figure 31.: Cornell box rendered with Whitted shader.

4.1.5 *PathTracer*

The last developed shader is the canonical Path Tracer, which is a Monte Carlo method of rendering images from 3D scenes, such that the global illumination is faithful to reality. This renderer algorithm can synthesize very realistic scenes, but it takes much longer rendering times than the previous shaders. Unlike the previous shaders, this one fully simulates both direct and indirect lighting, so the ambient light has no place in this algorithm, and also only supports lights with an area. It simulates the light reflected on diffuse and specular surfaces, as well as the light refracted on transparent surfaces.

The light reflected on diffuse surfaces is divided in 2 parts: direct lighting and indirect lighting. The direct lighting is calculated using the next event estimation, which is a

technique of importance sampling, that can make the Monte Carlo integration converge much faster, i.e., with fewer samples.

The next event estimation algorithm is as follows:

Algorithm 5: Algorithm of next event estimation

foreach *sampleLight* **do**
  Choose a random light (PRNG uniformly distributed)
  Choose a random position in area light (PRNG uniformly distributed)
  Calculate vector to light
  $\theta \leftarrow \angle(\overrightarrow{N}, \overrightarrow{L})$
  if *intersected surface is facing towards the light* **then**
    Cast shadow ray
    if *primitive is not in shadow* **then**
      $BRDF_{lambert} \leftarrow \texttt{kD} \div \pi$
      $\texttt{solidAngle} \leftarrow (\texttt{areaLight} \times (\overrightarrow{Nl} \cdot \overrightarrow{-L})) \div \texttt{lightDistance}^2$
      $PDF_{light} \leftarrow 1 \div \texttt{solidAngle}$
      $\texttt{sample} \leftarrow (BRDF_{lambert} \times Le \times \cos(\theta)) \div PDF_{light}$
      $\texttt{Ld} += \texttt{sample} \times \#lights \div \#samples$

On the other hand, the indirect light sampling is performed, by random sampling the hemisphere of a surface. The random sampler used was the cosine weighted hemisphere, as this algorithm, can make the Path Tracer converge faster:

Algorithm 6: Algorithm of Lambert BRDF sampling.

Calculate new direction with cosine weighted hemisphere sampling
Transform vector from tangent coordinates to world coordinates
$\texttt{Li} \leftarrow$ Cast secondary ray
$BRDF_{lambert} \leftarrow \texttt{kD} \div \pi$
$\theta \leftarrow \angle(\overrightarrow{N}, \overrightarrow{secondaryRay})$
$PDF_{hemisphere} \leftarrow \cos\theta \div \pi$
$\texttt{sample} \leftarrow (BRDF_{lambert} \times Li \times \cos(\theta)) \div PDF_{hemisphere}$
$\texttt{LiD} += \texttt{sample}$

To improve the Path Tracer even further, it was implemented a russian roulette scheme, where the probability to continue is given by the color of the intersected material. And the russian roulette algorithm is as simple as:

Algorithm 7: Algorithm of russian roulette.

$\texttt{randomNumber} \leftarrow$ random number between 0 and 1
$\texttt{Pcontinue} \leftarrow$ material color
if $randomNumber \geq Pcontinue$ **then**
  return

Lastly, the reflected and refracted light in a specular surface are calculated in a similar way to the Whitted algorithm presented previously.

So, the algorithm developed in Path Tracer Shader is as following:

Algorithm 8: Algorithm of Path Tracer Shader.

input : `russianRoulette`, `ray`, `intersection`, `lightEmission`
output: `outputRGB`

if *intersected material is a light source* **then**
  if *`lightEmission` == true* **then**
    `outputRGB` $\leftarrow$ `Le`
  else
    `outputRGB` $\leftarrow 0$
  return
`randomNumber` $\leftarrow$ random number between 0 and 1
`Pcontinue` $\leftarrow$ material color
if *`randomNumber` $\geq$ `Pcontinue`* **then**
  return
if *Ray reached maximum depth* **then**
  return
if *intersected material is diffuse* **then**
  foreach *sampleLight* **do**
    Choose a random light (PRNG uniformly distributed)
    Choose a random position in area light (PRNG uniformly distributed)
    Calculate vector to light
    $\theta \leftarrow \angle(\overrightarrow{N}, \overrightarrow{L})$
    if *intersected surface is facing towards the light* **then**
      Cast shadow ray
      if *primitive is not in shadow* **then**
        $BRDF_{lambert} \leftarrow$ `kD` $\div \pi$
        `solidAngle` $\leftarrow$ (`areaLight` $\times (\overrightarrow{Nl} \cdot \overrightarrow{-L})$) $\div$ `lightDistance`^{2}
        $PDF_{light} \leftarrow 1 \div$ `solidAngle`
        `sample` $\leftarrow (BRDF_{lambert} \times Le \times \cos(\theta)) \div PDF_{light}$
        `Ld` $+=$ `sample` $\times$ *#lights* $\div$ *#samples*
  end

```
34 if intersected material is diffuse then
35     Calculate new direction with cosine weighted hemisphere sampling
36     Transform vector from tangent coordinates to world coordinates
37     Li ← Cast secondary ray
38     BRDF_lambert ← kD ÷ π
39     θ ← ∠(N⃗, secondaryRay⃗)
40     PDF_hemisphere ← cos θ ÷ π
41     sample ← (BRDF_lambert × Li × cos(θ)) ÷ PDF_hemisphere
42     LiD+ = sample
43 if intersected material is only specular reflective then
44     castReflection ← true
45 if intersected material is only specular refractive then
46     castRefraction ← true
47 if intersected material is both specular reflective and refractive then
48     kr ← fresnelEquation
49     randomNumber ← random number between 0 and 1
50     if randomNumber > kr then
51         castRefraction ← true
52     else
53         castReflection ← true
54 if castReflection == true then
55     Calculate reflection direction
56     Cast a specular ray
57     Li ← Cast specularRay ray
58     LiS ← Li × kS
59 if castRefraction == true then
60     eta ← 1/refractiveIndex
61     cosDir ← rayDir⃗ · intersectionNormal⃗
62     if intersected surface is facing the same direction as the ray then
63         Invert eta
64         Invert normal
65     Calculate refraction direction
66     Li ← Cast specularRay ray
67     LiT ← Li × kT
68 outputRGB ← outputRGB + Ld + LiD + LiS + LiT
```

where

russianRoulette : a MobileRT Sampler used to randomly determine when to stop the ray bounces

ray : contains some information about the casted ray, like its origin, direction and depth

intersection : contains some information about the intersection, like its point, normal, distance from ray origin and material of the intersected primitive

outputRGB : the output parameter to write the pixel's RGB color

(a) Path Tracer algorithm with light sampling (took 694.18s)

(b) Path Tracer algorithm without light sampling (took 421.67s)

Figure 32.: Cornell box with Buddha model rendered with Path Tracer shader at 992x1200.

The figures 32 illustrates a scene, with a Buddha model inside a Cornell box composed by 1087463 triangles and rendered with this PathTracer shader. It was rendered with the Android emulator executed by the computer with specifications referenced in the table 3. It was used 202 samples per pixel in both images and both used the BVH acceleration structure with 2 threads. On the left image, it was used light sampling while the right image was rendered without it. As it can be seen, the rendered image with light sampling have less noise than without those extra samples, but also led to an increase in the execution time. This is expected, as those extra samples have higher importance but also requires extra CPU processing time.

Figure 33.: Illustration of a scene rendered with PathTracer at 496x496 with 10000 spp and 1 spl.

The figure 33 illustrates an image rendered with this PathTracer shader. This particular scene has 12940 triangles plus two area lights. As it can be seen, this shader is capable of simulating Global Illumination. Also notable is the caustic projected onto the red wall from the light passing through the wine glass. However, in order to get this quality of image with path tracing, it took 56 minutes of rendering time, even with the BVH accelerator and with 8 threads. Because, it was rendered with 496x496 pixels and with 10000 samples per pixel. All of this was executed by the laptop W230SS, which has the next specifications:

Table 3.: Laptop device specifications.

Device	CPU	Cache(L1/L2/L3)	RAM
Clevo W230SS	4xCore i7-4710MQ w/ HT @ 2.5GHz	i32KB + d32KB/256KB/6MB	16GB

Unfortunately, a typical Android mobile device has worse specifications than this, so it would take even more time to render. The chapter 6 shows some rendering times that some Android devices took to render with both Whitted and Path Tracer shaders.

4.2 SAMPLERS

There were implemented four samplers: Constant, Stratified, HaltonSequence and MersenneTwister.

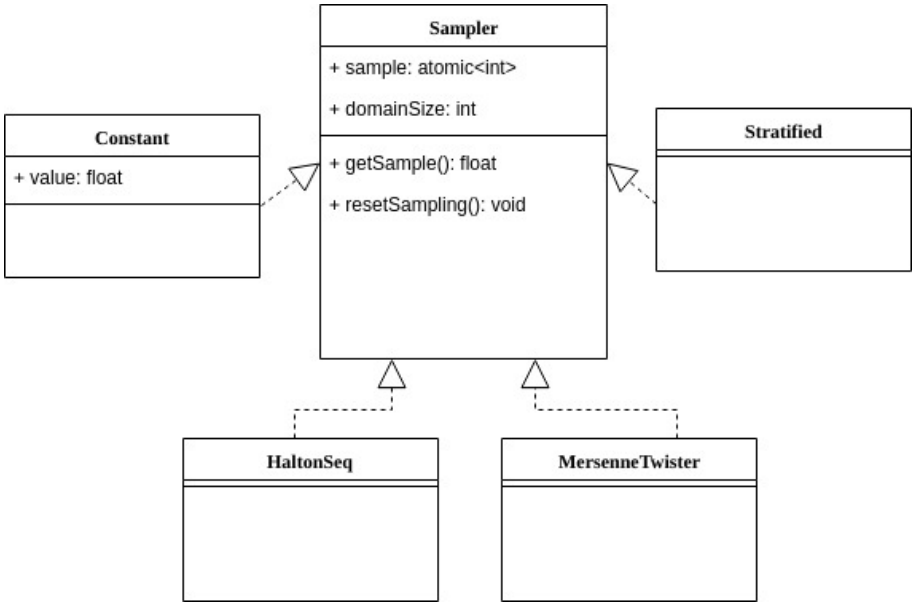


Figure 34.: Class diagram of the Samplers.

All samplers just provide two methods for the user to use:

```
1 virtual float getSample() noexcept = 0;
2 void resetSampling() noexcept;
```

Listing 4.2: Main methods in Sampler

The *getSample* generates a random number between 0 and 1. Each implemented Sampler must ensure this interval and have at its disposal an atomic variable *sampleCounter* to ensure non-repetition of samples between different threads.

The *resetSampling* method serves to restart the sequence.

4.2.1 *Constant*

This sampler is the simplest because it always returns the same number passed to the constructor.

This is used when the user only needs one sample per pixel and wants all the samples to be in the middle of each pixel.

4.2.2 *Stratified*

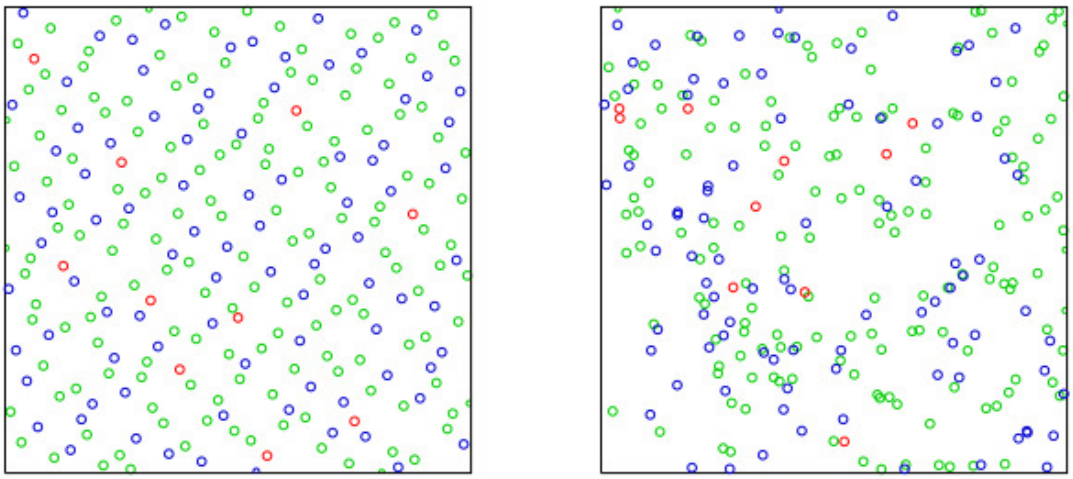
This sampler makes each sample at equal distance $(1/domainSize)$ and in ascending order. For example, for a domain size of 4, the samples taken are going to be: 0, 0.25, 0.5 and 0.75.

This is useful for taking samples when supersampling without any noise, because each sample is not randomly chosen.

4.2.3 *HaltonSequence*

As the name implies, this sampler generates the Halton sequence.

Halton sequence is a quasi random number sequence which is a deterministic sequence with low discrepancy. These sequences are usually good for Rendering Algorithms like Path Tracing because it can make the rendering equation converge faster, that is, with fewer samples.



(a) Halton sequence in a 2D image plane (Wikipedia (d)).

(b) Pseudorandom sequence in a 2D image plane (Wikipedia (d)).

Figure 35.: Difference between quasi random sequence and pseudo random sequence.

The Halton sequence algorithm is very simple as shown in algorithm 9 and it was taken from Wikipedia (d).

Algorithm 9: Algorithm of Halton Sequence.

input : `index`, `base`
output : `result`

```
1 f ← 1
2 result ← 0
3 while index > 0 do
4     f ← f ÷ base
5     result ← result + f × (index mod base)
6     index ← ⌊index ÷ base⌋
```

where

`index` : the desired index in the sequence
`base` : the base of the domain
`result` : the generated random number

4.2.4 *MersenneTwister*

This sampler is just a wrapper to call the MersenneTwister algorithm from the standard C++ library. This generator is a pseudo random number generator (PRNG), which is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. Although the PRNG-generated sequence is not truly random, because it is completely determined by an initial value, called the PRNG's seed, it was used a std::random_device for the generator seed because it produces non-deterministic random numbers, when supported. These sequences are typically used in simulations that use Monte Carlo methods like, for example, the Path Tracing. It is a good PRNG because it produces uniformly distributed numbers, it doesn't repeat the same sequence, it does not exhibit correlation between successive numbers and is one of the fastest PRNG available.

4.3 LIGHTS

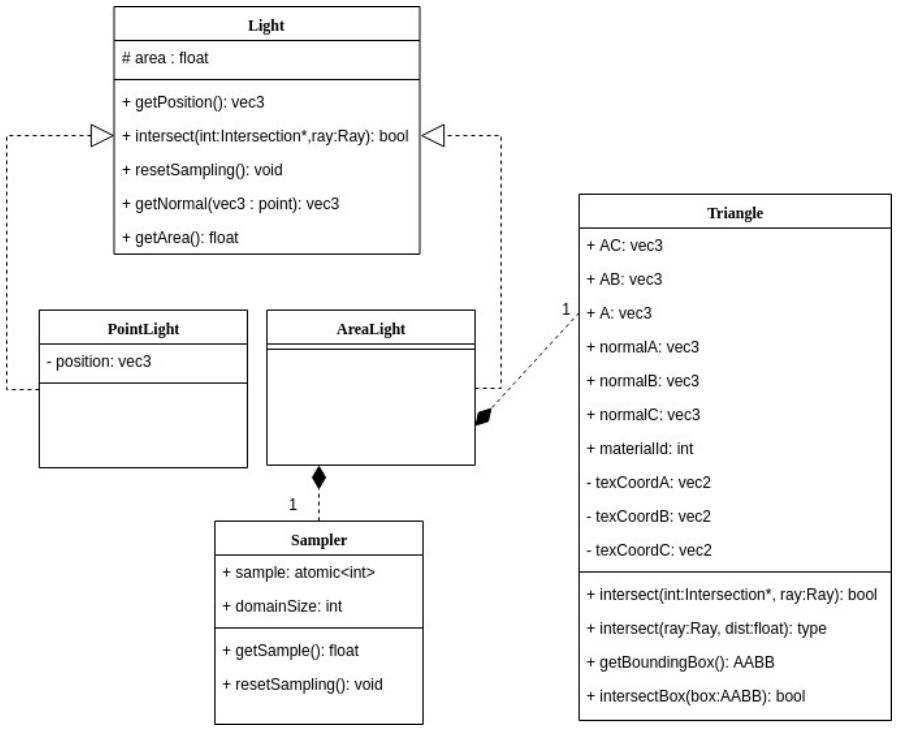


Figure 36.: Class diagram of the Lights.

There were implemented two types of light sources: point light and area light. These light sources provide the user five main methods:

```
1 glm::vec3 getPosition() noexcept = 0;
2 virtual bool intersect(Intersection *intersection, const Ray &ray) const noexcept = 0;
3 float getArea() const noexcept;
4 virtual glm::vec3 getNormal (const glm::vec3 &point) const noexcept = 0;
5 virtual void resetSampling() noexcept = 0;
```

Listing 4.3: Main methods in Light

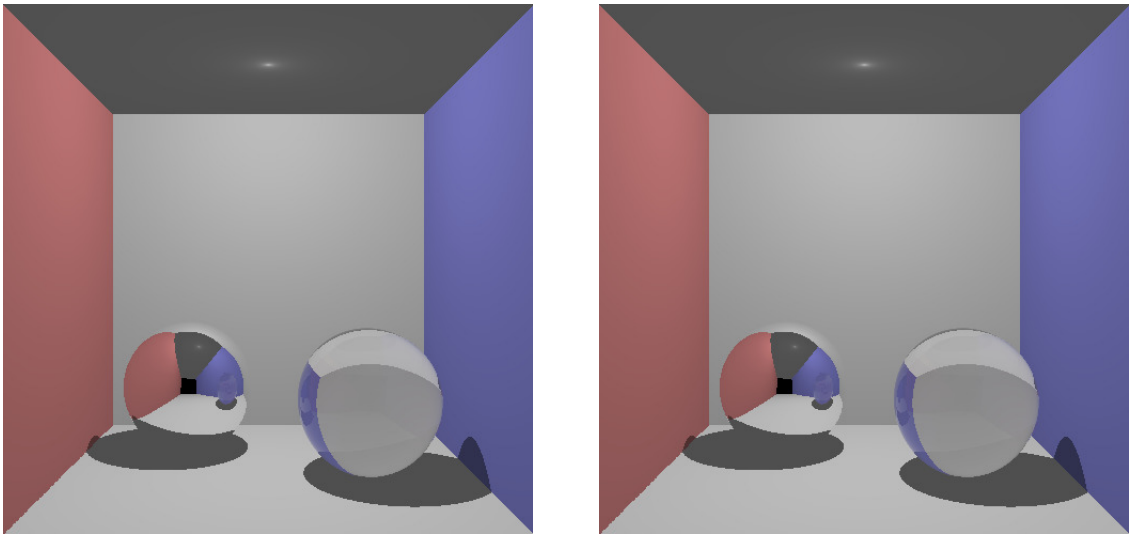
The *getPosition* method just returns the position of the light source and the *intersect* method determines whether a given *ray* as a parameter intersects this light source and, if it intersects, writes the result to the *intersection* parameter.

The *getArea* method, as the name implies, returns the area of the light. This is useful when the shader algorithm needs the area of the light in order to project it against the field of view in each intersection point. And the *getNormal* method, returns the normal direction of the light at that position point passed as parameter. Like the previous method, this is useful for shaders which needs to project the area of the light against the field of view in

each intersection point. Last, the *resetSampling* is a method to restart the random sequence of the samplers used in the light. This method only has meaning in the AreaLight because it uses a Sampler in order to get a random position within the light.

4.3.1 *Point light*

The Point light is the simplest form of a light because, as the name implies, it is just a point of light which emits light in all directions at once. This type of light is useful to render fast shadows because it just simulates hard shadows, in that each shadow is only constituted with an umbra, which is the fully dark shadow.



(a) Hard shadows with 1 sample per light.

(b) Hard shadows with 1000 samples per light.

Figure 37.: Hard shadows.

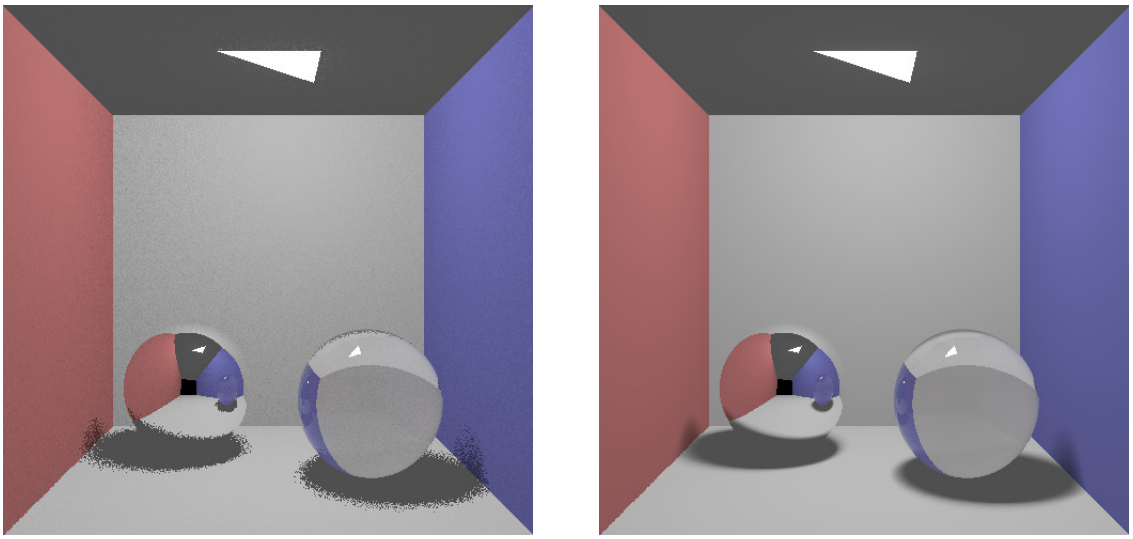
Therefore, the *getPosition* method is very simple because it only returns the position of the light source determined in its constructor.

And the *intersect* method is also quite simple because the ray parameter never intersects the point light. This is because the probability of a ray intersecting a single 3D point in the world is practically null.

The *getArea* method obviously always returns zero, because a point has no area. And the *getNormal* returns an invalid normal, i.e., a vector direction with zero length.

4.3.2 *Area light*

The Area light is another type of light source where the light comes from an area. For this application, the Area light has a shape of a triangle. This shape is intended as the triangle is the simplest shape which allows to construct more complex shapes. This allows the simulation of soft shadows as illustrated in the figure 38.



(a) Soft shadows with 1 sample per light. (b) Hard shadows with 1000 sample per light.

Figure 38.: Soft shadows.

With barycentric coordinates, it is very easy to generate a random point in the triangle. A triangle with three points: A, B and C. We get vectors AB and AC, being:

```
AB=[Bx−Ax,By−Ay,Bz−Az] and AC=[Cx−Ax,Cy−Ay,Cz−Az].
```

Listing 4.4: Vectors AB and AC in a triangle

These vectors tell us how to get from point A to the other two points in the triangle, by telling us what direction to go and how far. So, with barycentric coordinates R=1/3, S=1/3 and T=1/3, we get the point in the center of the triangle. To generate a random point in the triangle, we have to generate two random numbers between 0 and 1 (R and S). Then we have to make sure we stay inside the triangle by checking if they are larger than one:

```
if (R + S >= 1) {
  R = 1 − R
  S = 1 − S
}
```

Listing 4.5: Algorithm of Area light

Finally we can obtain a random point in the triangle by starting at point A, then getting a random percentage along vector AB and a random percentage along vector AC:

```
1 RandomPointPosition = A + R*AB + S*AC
```

Listing 4.6: Algorithm of Area light

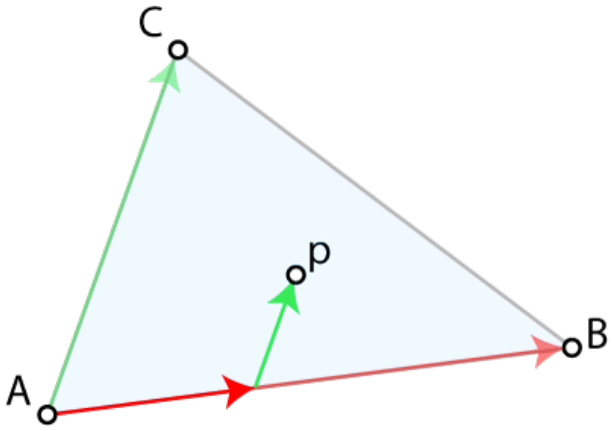


Figure 39.: Calculation of point P by using barycentric coordinates starting at point A and adding a vector AB and a vector AC (Jacobson).

So, the *getPosition* algorithm is as following:

Algorithm 10: Algorithm of getPosition in Area light.

input :
output : `randomPosition`

1 `R` ← generate random number in a range [0,1)
2 `S` ← generate random number in a range [0,1)
3 **if** *random numbers get outside of triangle* **then**
4 | Invert generated numbers
5 `randomPosition` ← $\texttt{triangle}.A + \texttt{R} \times \texttt{triangle}.AB + S \times \texttt{triangle}.AC$

where

`randomPosition` : the generated random position in the area light

And the *intersect* method just calls the intersection method of a ray with the triangle presented in the previous section, and if it does intersect then it updates the material and the area in the intersection parameter.

Algorithm 11: Algorithm of intersect in Area light.

input : `ray`, `intersection`
output : `intersection`

```
1 Call intersect method of the triangle
2 if intersected then
3 |   Update intersection's material
4 |   Update intersection's area light
```

where

`ray` : contains some information about the casted ray, like its origin, direction and depth

`intersection` : contains some information about the intersection, like its point, normal, distance from ray origin and material of the intersected primitive

The *getArea* method returns the area of the triangle, which is calculated at the construction of the AreaLight. And the *getNormal* method calculates the normal of that point by using the method described by joojaa.

4.4 CAMERAS

The Camera describes how the process of rendering will start, because, in this ray tracer engine, all the primary rays will be casted from the camera. The programmer can set the camera anywhere in the scene can direct it wherever he wants, so making it possible to render the same scene from different perspectives.

Only two types of cameras were implemented for the demonstration: perspective and orthographic cameras.

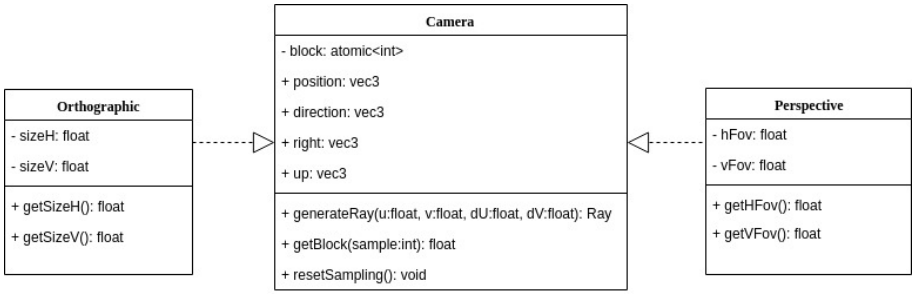


Figure 40.: Class diagram of the Cameras.

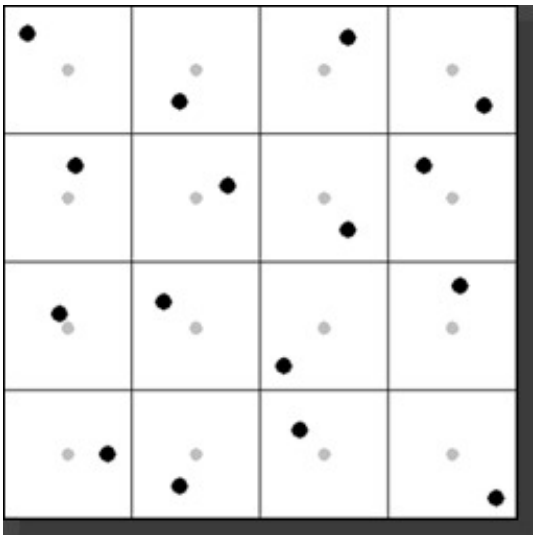
The camera only provides one method for the user to cast a ray from the camera position in direction to the image plane, and two more auxiliary methods:

```
1 virtual Ray generateRay(float u, float v, float deviationU, float deviationV) const noexcept = 0;
2 float getBlock(int sample) noexcept;
3 void resetSampling() noexcept;
```

Listing 4.7: Main methods in Camera

The method *getBlock* is used by the Renderer in order to select a random tile in the image plane. So, when a thread finishes rendering a tile, it will ask the Camera another random tile. And the method *resetSampling* is used to restart the random sequence used in the method *getBlock*.

Finally, the method *generateRay* needs four parameters to create a ray. The u and v are used to choose the targeted pixel in the image plane. Being u the inverse of the index of the pixel in its line, that is, $x/width$, and v the inverse of the index of the pixel in its column, that is, $y/height$. In order to allow the reduction of aliasing in the generated images of the scene, the camera also accepts two extra parameters *deviationU* and *deviationV* which are variances inside a pixel. The *deviationU* is a horizontal variance of the pixel, that is, $[-0.5 * pixelWidth, 0.5 * pixelWidth]$ and *deviationV* the variance in the vertical of the pixel $[-0.5 * pixelHeight, 0.5 * pixelHeight]$. This technique is called jittering, as shown in the left figure 41, and allows to reduce the aliasing effect by introducing noise into the output image. Although the final image gets some noise, this effect turns out to be visually more appealing than aliasing because it is an effect without patterns that are easily detectable by the human eye. It is important to reduce the aliasing effect because, it is an effect that the human eyes can detect very fast because the generated image will have quite regular patterns, as shown in the right figure 41.



(a) Illustration of how sampling the plane image with jittering works in order to avoid aliasing (Waters).

(b) Aliasing effect (Science and Engineering).

Figure 41.

Sampling with jittering is, as previously stated, a good technique to avoid aliasing in the image. The figure 42 shows an example of stratified sampling with 1 and 256 samples per pixel. As can be seen in the figure on the left, there is a noticeably aliasing in the image, and we can't even perceive the correct positions of the black squares on the background. Of course, with 256 samples per pixel, the aliasing effect is greatly reduced, but also, the execution time is linearly increased, which is a downside to the user experience.

(a) Stratified sampling with 1 sample per pixel.

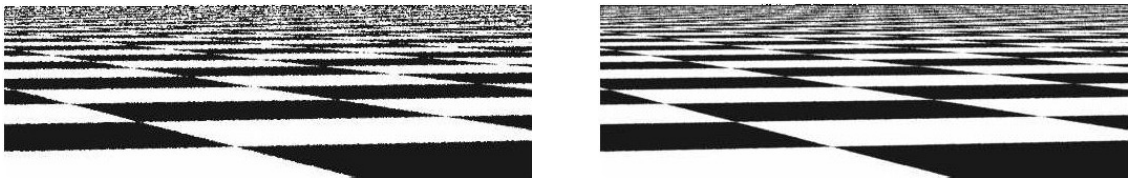
(b) Stratified sampling with 256 samples per pixel.

Figure 42.: Stratified sampling.

Sampling with jittering is, obviously not perfect, as shown in the figure 43. But, it produces more visually appealing images, although it introduces some noise. Even, with just 1 sample per pixel, the generated image is less "jaggy" in the background. And, with 4 samples per pixel, the noise is greatly reduced and its quality is visually comparable to the stratified sampling with 256 samples per pixel. This means that by using jittering it

is possible to obtain images pleasing to the human eye with fewer samples per pixel, and consequently in less execution time.

The developed Renderer class applies some deviation to the rays casted from the camera, where the programmer can develop his own method of deviation (i.e. jittering, stratified, etc.) by using a Sampler of his choice.



(a) Jittered sampling with 1 sample per pixel.

(b) Jittered sampling with 4 samples per pixel.

Figure 43.: Jittered sampling.

4.4.1 *Perspective Camera*

This type of camera is the most used in renderers because it uses perspective projection. With it, it can simulate images being seen by the human eyes, which means, it simulates the depth of the objects, and produces 2D images with a 3D projection.

To obtain an image plane with perspective, it is necessary to have a Field of View. In order to accept any resolution of the image plane, we have to divide the field of view in two parts: horizontal and vertical. This way, we can obtain the aspect ratio of the image plane we want.

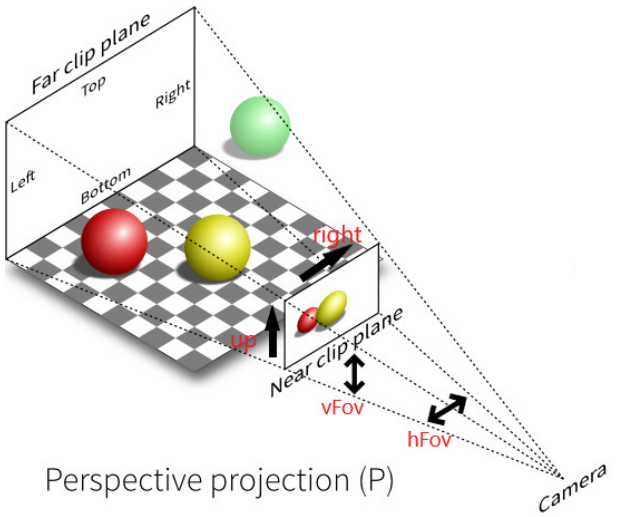


Figure 44.: Perspective camera with hFov and vFov (Schim).

The algorithm to generate a ray from the perspective camera is very simple. By knowing the horizontal Field of View and vertical Field of View in radians, and with u and v as parameters, it is possible to calculate the direction of that ray.

It starts to calculate the distance to go through the right vector and the up vector of the camera. That can be done with the arctangent of each Field of View of the camera (horizontal and vertical) and multiplying it with $u - 0.5$ in the right vector and with $0.5 - v$ in the up vector. This makes sure that we go through every pixel, starting with the pixel from the top left corner to the bottom right corner of the image plane. Then the destination point of the ray is just the sum:
$destinationPoint = cameraPosition + cameraDirection + rightVector + upVector$, and so its direction is just:
$rayDirection = destinationPoint - cameraPosition$. The origin of the ray is the position of the camera, because in a perspective camera, all rays come from the same point: the point where the camera is located.

Algorithm 12: Algorithm of generateRay in Perspective Camera.

input : u, v, deviationU, deviationV
output : ray

1 $\texttt{rightFactor} \leftarrow arctan(\texttt{hFov} \times (\texttt{u} - 0.5)) + \texttt{deviationU}$
2 $\overrightarrow{rightVector} \leftarrow \texttt{right} \times \texttt{rightFactor}$
3 $\texttt{upFactor} \leftarrow arctan(\texttt{vFov} \times (0.5 - \texttt{v})) + \texttt{deviationV}$
4 $\overrightarrow{upVector} \leftarrow \overrightarrow{up} \times \texttt{upFactor}$
5 $\texttt{destinationPoint} \leftarrow \texttt{cameraPos} + \overrightarrow{cameraDir} + \overrightarrow{rightVector} + \overrightarrow{upVector}$
6 $\overrightarrow{rayDir} \leftarrow \texttt{destinationPoint} - \texttt{cameraPos}$
7 $\texttt{ray} \leftarrow Ray(normalize(\overrightarrow{rayDir}), \texttt{cameraPos})$

where

u	: relative index of pixel in a line
v	: relative index of pixel in a column
deviationU	: horizontal deviation inside a pixel
deviationV	: vertical deviation inside a pixel

4.4.2 *Orthographic Camera*

In this projection mode, an object's size in the rendered image stays constant regardless of its distance from the camera. This can be useful for rendering 2D scenes and UI elements, amongst other things.

The orthographic camera removes the sense of perspective by drawing the image plane without simulating the depth of the objects, making all the objects looking flat. This is achieved by inverting the logic of the perspective camera. Instead of calculating the

direction and always having the same origin, in the orthographic camera, the direction is always the same for all rays, and the origin of the ray varies.

It starts to calculate the distance to go through the right vector and up vector of the camera, like the perspective camera. But, instead of using the Field of View, we now use the $sizeH$ and $sizeV$, which are the horizontal and vertical sizes of the image plane. Then, to make sure that we go through all pixels from top left pixel to the bottom right, we need to use the u and v values from the parameters. Then, the origin of the ray is: $rayOrigin = cameraPosition + rightVector + upVector$.

Algorithm 13: Algorithm of generateRay in Orthographic Camera.

input : `u, v, deviationU, deviationV`
output : `ray`

1 $\overrightarrow{hDeviation} \leftarrow ((\texttt{u} - 0.5) \times \texttt{sizeH}) \times \overrightarrow{right}$ `// horizontal deviation in image plane`
2 $\overrightarrow{horizontalDeviation} \leftarrow \overrightarrow{hDeviation} + \overrightarrow{right} \times \texttt{deviationU}$ `// add horizontal jittering`
3 $\overrightarrow{vDeviation} \leftarrow ((0.5 - \texttt{v}) \times \texttt{sizeV}) \times \overrightarrow{up}$ `// vertical deviation in image plane`
4 $\overrightarrow{verticalDeviation} \leftarrow \overrightarrow{vDeviation} + \overrightarrow{up} \times \texttt{deviationV}$ `// add vertical jittering`
5 $\texttt{rayOrigin} \leftarrow \texttt{cameraPos} + \overrightarrow{horizontalDeviation} + \overrightarrow{verticalDeviation}$
6 $\texttt{ray} \leftarrow Ray(\overrightarrow{cameraDir}, \texttt{rayOrigin})$

where

`u`	: relative index of pixel in a line
`v`	: relative index of pixel in a column
`deviationU`	: horizontal deviation inside a pixel
`deviationV`	: vertical deviation inside a pixel

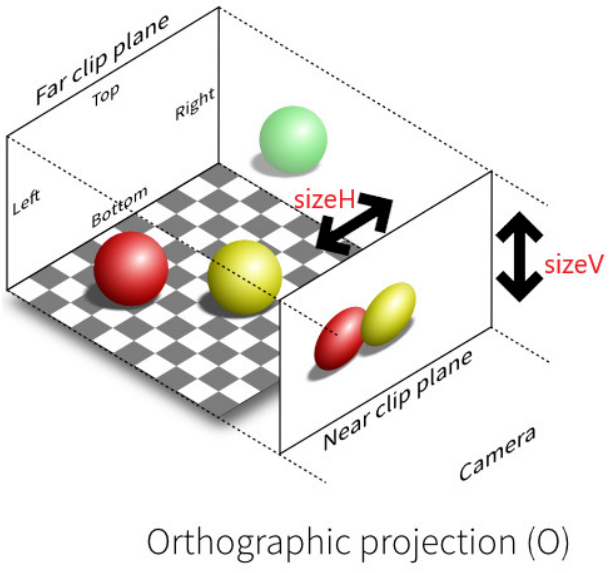


Figure 45.: Orthographic camera with sizeH and sizeV (Schim).

4.5 OBJECT LOADERS

This library also allows the programmer to develop his / her own Object Loaders. An Object Loader is, as the name implies, a class that allows the ray tracer to import meshes from different Model file formats. There are many different types of file formats for storing meshes: 3ds, FBX, Wavefront .obj file, COLLADA, SketchUp, AutoCAD DXF, etc.

All these file formats allows the user to store the positions of many triangles in a file, which together form the geometry of the scene.

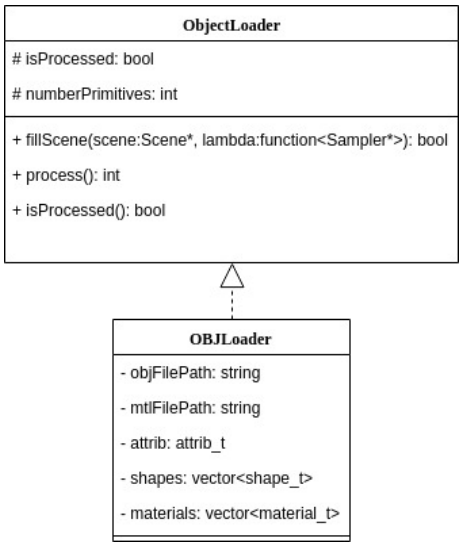


Figure 46.: Class diagram of the ObjectLoaders.

Every ObjectLoader developed by a programmer must implement two methods:

```
1 virtual int32_t process() noexcept = 0;
2 virtual bool fillScene(Scene *scene, std::function<std::unique_ptr<Sampler> ()> samplerLambda)
3   noexcept = 0;
4 bool isProcessed() const noexcept;
```

Listing 4.8: Main methods in ObjectLoader

The method *process* serves to let the object read the geometry file and fill the internal structures which the programmer must implement. Later, when the process of caching the data from the file geometry is done, the programmer should set the member variable *isProcessed* to true, so the method *isProcessed* returns the proper result. Finally, the method *fillScene* is where it should pass all the cached geometry to the scene. Note that this method also needs to receive a function which returns a pointer to a Sampler, also known as, factory. This way, in case the geometry file also supports lights, it allows to set the lights in the Scene with the same type of Sampler. The process of reading a geometry file and

pass the data to the Scene was separated in two steps, because the programmer can always use a third party library to parse a geometry file. And usually, those libraries require the programmer to setup some temporary structures.

This library only provides one Object Loader called OBJLoader that allows loading the scene geometry from Wavefront obj files, which was achieved using a third party library called "tinyobjloader" (syoyo). This type of Model file format is a good choice because, besides being simple, it is open and has been adopted by many 3D graphics application vendors, like, 3ds Max and Blender.

5

ANDROID LAYER

This chapter will describe the main functionalities of the top layer of the demo application, which is the UI layer. The figure 47 represents all three layers in the application and surrounds with a red rectangle the layer which will be described next.

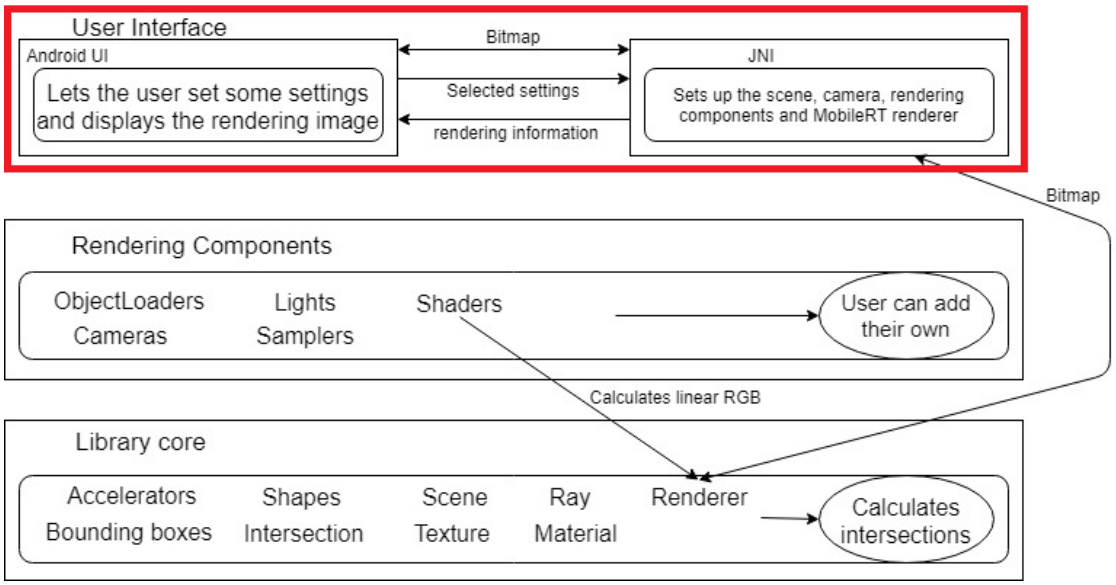


Figure 47.: Illustration of the three layers in the application.

5.1 ANDROID SPECIFICS

A typical Android application is programmed in Java programming language (Google (a)). It usually needs to communicate with an UI which is typically designed with the Android Studio Layout Editor and converted to Java. The code is compiled with Android SDK along with any data, assets and resource files used into an Android package (APK).

A big advantage for the programmers in Java is that they don't need to manually manage the memory. This prevents many code issues because of bad memory management, at the cost of some performance hit, as the Java Garbage Collector of the Java Virtual Machine

(JVM) uses some usually unknown algorithm to determine when to free the unused memory. There is also the limitation of the available memory for the application's heap imposed by the JVM (Google (a)). So, the only way of developing a ray tracer engine without these limitations is by programming it natively by using the Android Native Development Kit (NDK). Unfortunately, the NDK tool-chain doesn't provide any GUI libraries like Qt, GTKmm, or wxWidgets, that facilitates the development of a GUI in native code.

So, to obtain an optimal performance by default, in a mobile device, the ray tracer library and the rendering components were programmed in C++ by using the NDK while the UI was programmed in Java by using the SDK. The UI calls the methods provided by the ray tracer using the Java Native Interface (JNI).

The JNI is a programming standard that allows Java code running on a JVM to call native applications and libraries (programs specific to a hardware platform and operating system) written in other languages such as C, C++ and assembly.

Unfortunately, the use of JNI incurs some considerable overhead and performance loss under some circumstances. That's why it is not advised to use JNI to call native functions repeatedly. But, in this case, as the ray tracer can be a very computing demanding application, the time spent rendering the scene can be seconds, minutes or even hours depending on the shader algorithm and the number of samples used. So the number of JNI calls while the ray tracer is rendering a scene has been kept to a minimum, to only display some rendering information like, the current sample per pixel and the state of the ray tracer, to check if it is still running or if it finished. These methods are called every 250 ms so it updates the rendering text in the Android TextView.

5.2 USER INTERFACE

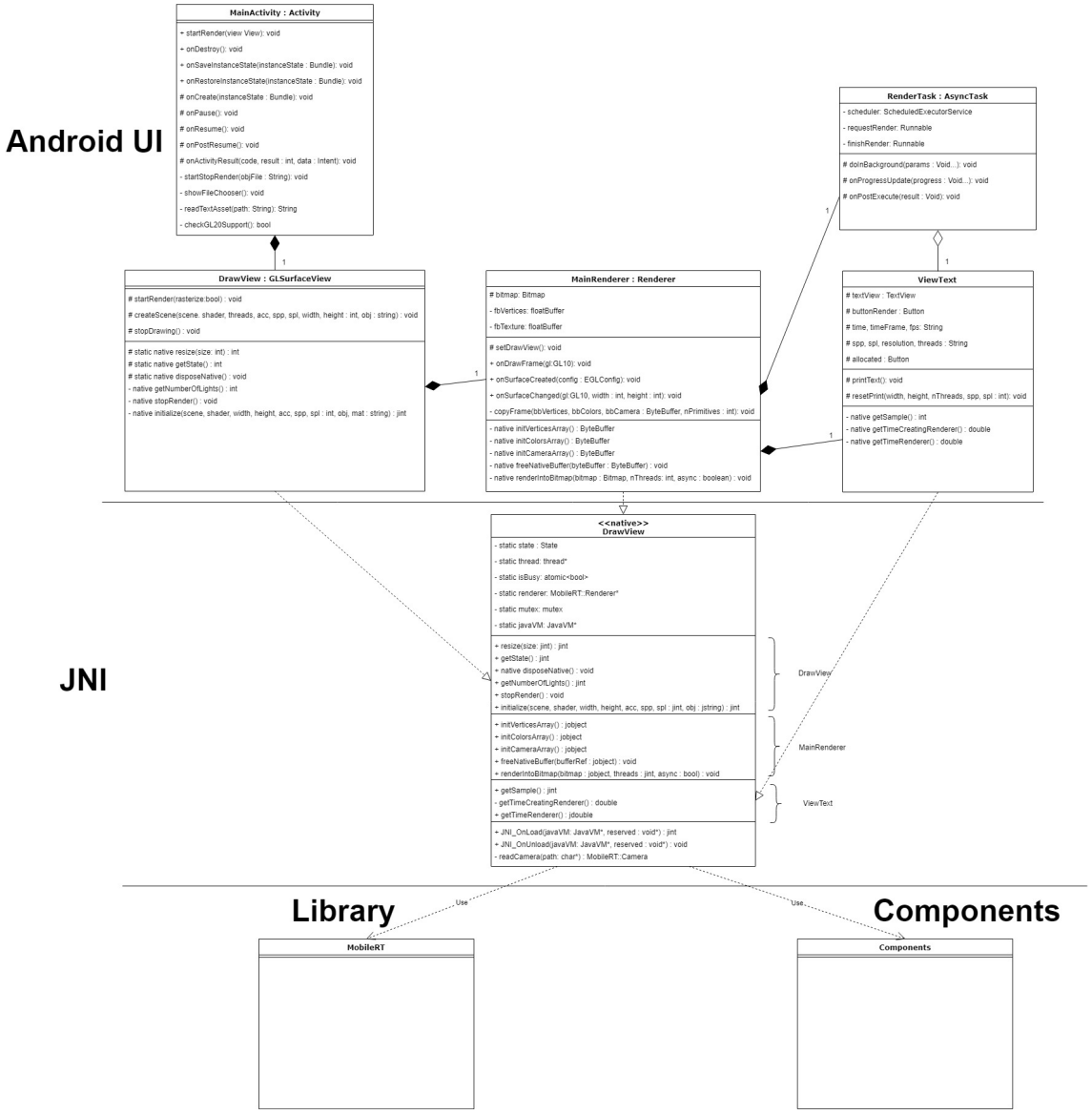


Figure 48.: Illustration of the class diagram in the UI.

As said previously, the UI in this demo application is divided in two sub layers: the Android UI and the JNI. The figure 48 illustrates how the various layers of the application are structured. The classes DrawView, MainRenderer and ViewText from the UI have access to many methods which a C++ header file provides through JNI. It was necessary to split the provided methods in three classes in order to avoid having a class object shared across many objects on the Android UI side and thus provoking a memory leak on the Java garbage collector.

The JNI sub layer provides several features of a ray tracing engine to the Android sub layer. It allows to setup predefined rendering components, as well as, setup some predefined scenes or even load scenes from OBJ geometry files. Among those methods, the ones provided to DrawView class are:

INITIALIZE sets the rendering components (samplers, shader, camera, lights and objectloader), fills the scene geometry and setups the MobileRT renderer

STOPRENDER stops the rendering process and sets the state machine value to stopped

GETSTATE pass to JVM the Renderer's current state (running, idle, stopped, finished)

DISPOSENATIVE clears the memory used by the MobileRT renderer and joins the thread and deletes it

RESIZE converts the width or height to a value which is multiple of the number of tiles in the renderer (ex: if it was set 16x16=256 tiles, then calculates the nearest number which is multiple of 16)

GETNUMBEROFLIGHTS pass to JVM the number of lights in the scene (for debug purposes)

And the ones provided to MainRenderer are:

INITVERTICESARRAY copy the scene's triangles' positions to a ByteBuffer where Java can read and pass to the OpenGL thread

INITCOLORSARRAY copy the scene's triangles' colors to a ByteBuffer where Java can read and pass to the OpenGL thread

INITCAMERAARRAY copy the camera's positions to a ByteBuffer where Java can read and pass to the OpenGL thread to set an equal camera

FREENATIVEBUFFER free the memory of the buffer argument

RENDERINTOBITMAP starts the rendering process and let the user choose whether in the current thread or in another separately

Finally, the provided native methods for the ViewText are:

GETSAMPLE pass to JVM the number of samples per pixel already rendered (for debug purposes)

GETTIMECREATINGRENDERER pass to JVM the time spent creating the acceleration structure in the Renderer (for debug purposes)

GETTIMERENDERER pass to JVM the time spent rendering the scene (for debug purposes)

Besides those methods for the Java UI, three additional methods were developed. Two of them are useful to setup and clear the JavaVM environment in the native code:

JNI_ONLOAD setups the JavaVM pointer in order to be accessed in any native method

JNI_ONUNLOAD destroys the JavaVM environment and clears the pointer in order to free the memory used

The last developed method is one which setups a Perspective camera from a file:

READCAMERA creates a perspective camera by using the data parsed from a ".cam" file

Figure 49 shows a simplification of the interaction between the Android UI sub layer and the JNI sub layer. The interaction between these two sub layers is kept to a minimum while the ray tracer engine is still rendering the scene. It's worth to note that both sub layers keep a pointer to the bitmap, in order to let the ray tracer engine write the calculated pixel's colors in the bitmap and at the same time let the OpenGL thread update the GPU with that information.

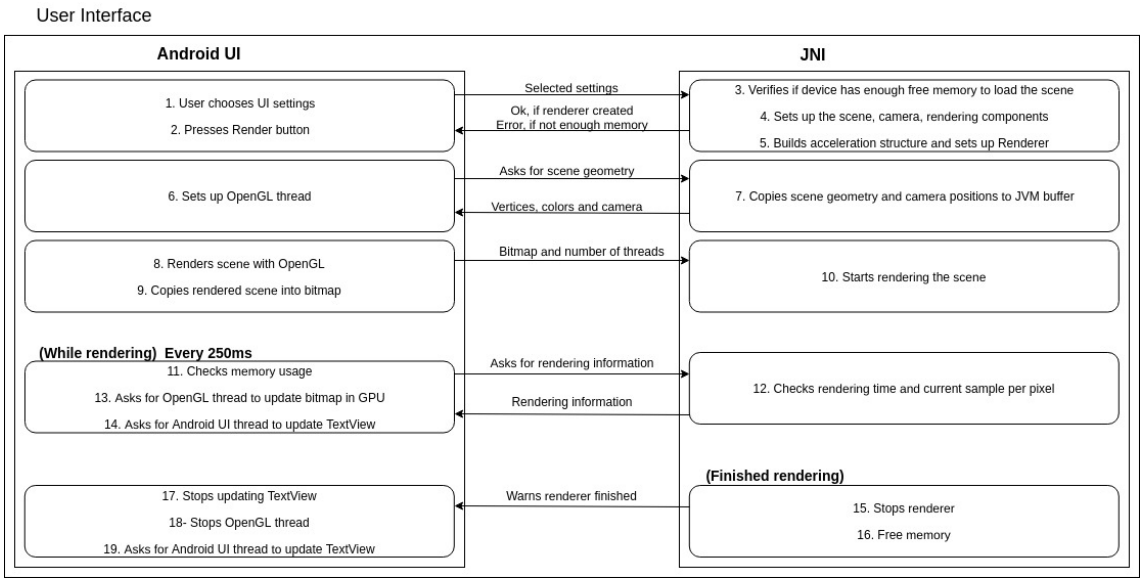


Figure 49.: Illustration of interaction between Android UI and JNI sub layers.

The Android UI is programmed in Java and only the UI thread can refresh it. The UI thread is the main thread of execution in an Android application. It is this thread which is in charge of executing the code of the lifecycle callbacks in the Activity class.

For this project, there are only two goals for the UI: should be as simple as possible and should be able to allow progressive ray tracing, which means, refreshing the image while it is being rendered.

In order to not get stuttering frame rate in an Android application, it is required to leave the UI thread as free as possible. It is not supposed to execute computationally demanding code with it as that will delay the refreshment of the UI.

In order to allow progressive ray tracing, it was used a pool of one thread called Render Task thread that every 250 ms wakes up the thread and updates the strings used in the TextView. These strings contain the information for the user, such as the rendering time, number of threads used, etc. Before going to sleep, it publishes the progress on the UI thread and requests the GL rendering thread to render a frame. Then when the UI thread wakes up, it concatenates all those strings into one and updates the TextView with it. Finally when the GL rendering thread wakes up, it renders a fullscreen square made by two triangles and applies a texture with a bitmap on them. That bitmap is where the ray tracer library is writing the rendered scene.

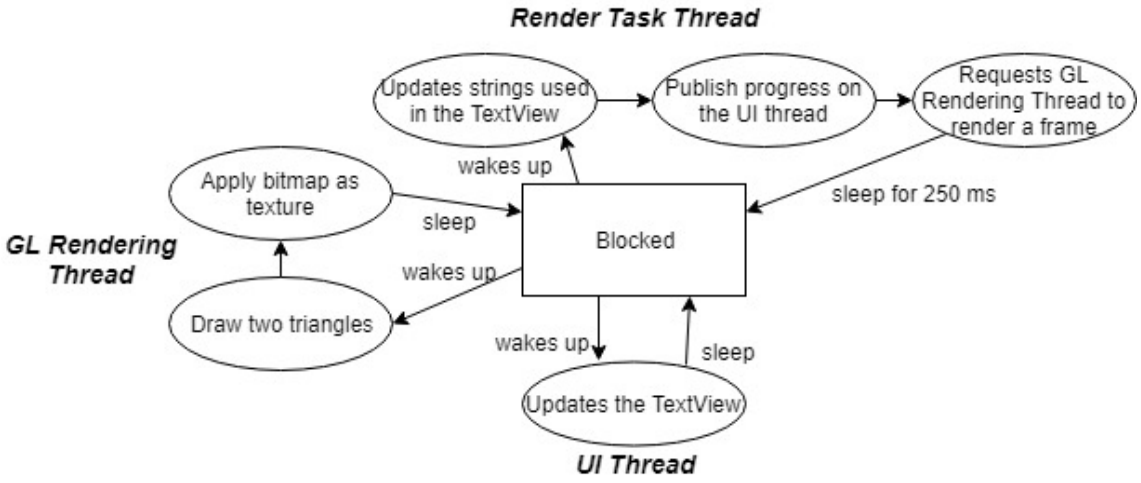


Figure 50.: Execution flow of UI thread, Render Task thread and GL rendering thread.

5.3 PROGRAMMING DECISIONS

It was important to properly split the application into three layers of abstraction, in order to let the programmer use the developed ray tracer library on any Operating System. The bottom layers (library and rendering components) and the JNI sub layer in the UI were compiled as three separated dynamic libraries, as this was the only way to let the JVM execute native code.

5.3.1 *Android benefits*

As mentioned above, this dissertation focuses only on ray tracing in the Android OS. This is intentional because, as mentioned in chapter 1, it is the OS for mobile devices with the most market share.

Developing apps in Android is cheaper than in other platforms. The company provides its SDK for free, so all the costs are destinated to the app testing and deploying. That means that the developers don't have to make a big investment in that part. Investing less on the app development means that they will have a higher return on investment (ROI) and their project will be more profitable.

Software developers, commonly learn to develop in Java, and it is easier for them to adapt to that programming language for mobile app development. Android is built mostly in Java, so its adaptation becomes faster and easier.

5.3.2 *Android challenges*

Developing applications for mobile devices has different challenges compared with the traditional personal computer (PC) hardware.

The Android UI has some particularities like only the UI thread can modify the Android UI components, like a view, button, number pickers, etc. So, it is easy to fall in the trap of letting the UI thread run everything in the code, and that will make the app unresponsive and very slow.

The size of RAM available for executing applications is typically smaller. In the 2010s the most expensive mobile devices had only about 512MB of RAM, and nowadays most of them have more than 4GB. But, still, comparing them with PCs which typically have 16GB it is a big reduction. This can make a big impact on the maximum number of primitives that a scene can have in the ray tracer library.

And the CPU microarchitecture is generally simpler and with smaller computational power. Also the OS is shaped for the mobile world, making a lot of restrictions in the performance of the applications in order to save battery. The amount of main memory available for the applications can also be affected by the OS.

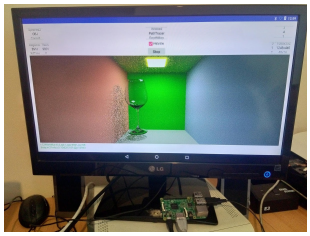
Other challenges are related with the communication mechanism between the SDK and the NDK, because two different languages environments need to communicate in runtime (GUI in Java and the library in C++). This involves learning how to use the JNI, so that the native code can send and receive information from JVM. And, as stated before, this can make a performance hit in the overall application.

Finally, by default, an Android app cannot access external storage like an SD card nor can be executed while the device is locked. So, in order to let this demo app to read geometry

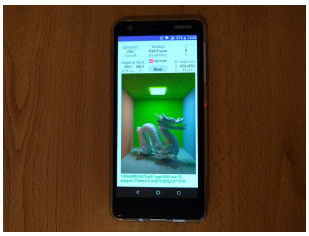
files from the SD card and to be able to render a scene while the mobile device is locked, it is necessary to explicit set those settings in the Android manifest.

5.3.3 *Compatibility*

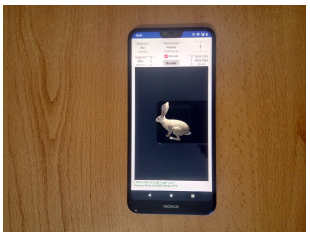
As the portability of the developed library was a concern from the start, this library in addition to being Android compatible, it is also possible to compile it for others devices with different Operating Systems (OS). So, it is possible to run it in several Android devices, like: a smart phone, a tablet computer, a smart TV, or even a smart watch; and also use it in a PC with a Linux or Windows OS. During the development of this dissertation, test applications were developed in order to achieve a good compatibility with different devices and different operating systems. The highlights of this work can be seen in the figures 51, that shows various devices running an application with the ray tracing library.



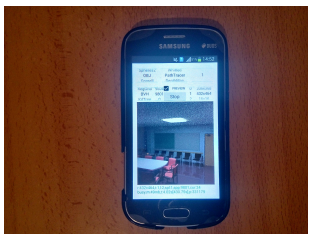
(a) Demo application being run in Raspberry Pi 3 with Android 7.1.1.



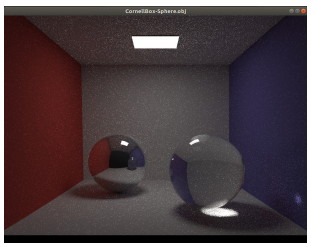
(b) Demo application being run in Nokia 3.1 with Android 8.1.



(c) Demo application being run in Nokia 7.1 with Android 9.0.



(d) Demo application being run in Samsung Galaxy Fresh with Android 4.1



(e) Demo application being run in a laptop with Ubuntu 18.04

Figure 51.: Demo application being run in various devices.

6

DEMONSTRATION: GLOBAL ILLUMINATION

In order to assess the performance of the developed library, 3 scenes were selected to measure the rendering time in 3 different devices. In all scenes, it was placed 2 area lights in shape of a triangle in order to visualize more realistic shadows and light phenomena like the caustics. It was always used the same compiler, clang, and the same compilation flags, being the most relevant: -O3, -flto=full and -ftree-vectorize (team (a)). These flags were set in all translation units, even in the third party libraries used, like the glm and the tinyobjloader.

The scenes were carefully chosen for their size, structure and appearance. It was chosen one large scene with some hundred of thousands triangles, one moderate with few tens of thousands of triangles and one small scene with just a couple of thousands of triangles.

So, it was tested a large scene, called Conference Room, which is a scene with 331179 triangles as illustrated in figure 52. This scene consists mainly of diffuse materials, except the floor which is specular, so it reflects the incident light in only one direction. The size of the Regular Grid used in this scene was 128x128x128 in all the measurements, as this size showed the best results for the grid.

(a) Rendered with Whitted (113.128s)

(b) Rendered with Path Tracer (474.65s)

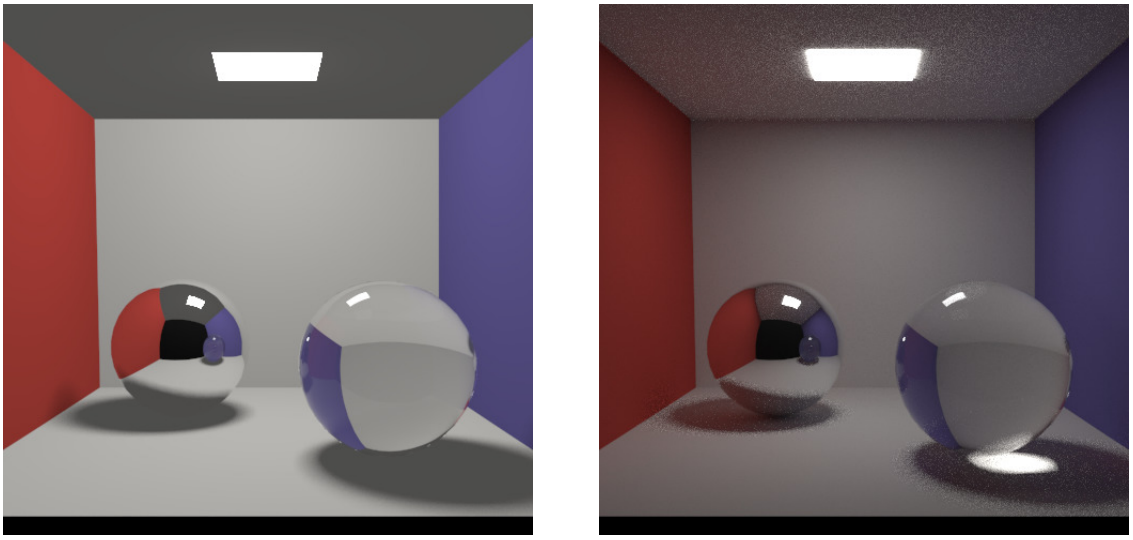
Figure 52.: Conference Room, rendered with MobileRT (BVH) w/ 1000spp at 496x496 (on Ubuntu 18.04 with W230SS).

The medium scene used for testing was the model of Porsche 911 GT2 inside a box, as illustrated in figure 53. This scene consists of 22023 triangles and the Porsche has a diffuse texture applied to it except in the tires and in the glass. The tires are made of diffuse materials with a tiny bit of specularity. The glass are, of course, specular which reflects and refracts the light with the refractive index of 1.5, and a little bit diffuse in order to mimic real life glass. Finally, the size of the Regular Grid used in this scene was 64x64x64 in all the measurements.

(a) Rendered with Whitted (251.165s) (b) Rendered with Path Tracer (1054.59s)

Figure 53.: Porsche 911 GT2, rendered with MobileRT (BVH) w/ 1000spp at 496x496 (on Ubuntu 18.04 with W230SS).

Finally, the small scene used for testing was the Cornell Box scene, as illustrated in figure 54. This scene consists of only 2186 triangles, arranged in the form of 2 spheres inside a box. The left sphere is a perfect mirror and the right sphere is made of glass. The size of the Regular Grid used in this scene was 64x64x64 in all the measurements as the previous scene.



(a) Rendered with Whitted (130.147s) (b) Rendered with Path Tracer (274.251s)

Figure 54.: Cornell Box, rendered with MobileRT (BVH) w/ 1000spp at 496x496 (on Ubuntu 18.04 with W230SS).

The models of the Conference Room and the Cornell Box were downloaded from http://casual-effects.com/data/index.html (McGuire (2017)). While the model of the Porsche was downloaded from https://free3d.com/3d-model/porsche-911-gt-43465.html (Free3D).

Like it was said before, the developed demo application was tested in 3 different devices. The devices used in this experiment were 3 smartphones, each from different generation:

- Samsung Galaxy Fresh Duos GT-S7392
- Nokia 3.1
- Nokia 7.1

These devices were chosen to test the developed library due to their different specifications from one another. For example, the Samsung Galaxy Fresh Duos GT-S7392 is a device with a single core low end CPU. This device is identical to the common low end smartphones available today at the market. The Nokia 3.1 is a device with a mid end CPU and is identical to most of the mid range smartphones available. Finally, the Nokia 7.1 is a device with a high end CPU which has specifications near the high end smartphones available last couple of years.

The table 4 shows in more detail the most important specifications of each device.

Table 4.: Android devices specifications.

Device	CPU	RAM	View resolution	Android OS
Samsung Galaxy Fresh	1xARM Cortex A9@1GHz	512MB	432x464	4.1
Nokia 3.1	8xARM Cortex-A53@1.5GHz	2GB	656x880	8.1
Nokia 7.1	8xQualcomm Snapdragon@1.8GHz	3GB	992x1536	9.0

6.1 RESULTS OBTAINED

This section shows the median of the measured times to render each scene presented above with the Whitted shader in each device. In all these measurements with the Whitted shader, the 3 scenes were rendered with 1 sample per pixel and 1 sample per light. The goal is to assess the performance of the developed acceleration structures in those devices.

After those measurements, it was also measured a scene rendered with the Path Tracer shader. But this time, the goal is to assess the performance of the developed library with PBRT. And all the measurements were made with 64 samples per pixel in a scene with many more triangles than the previous ones.

6.1.1 *Whitted Shader*

Samsung Galaxy Fresh Duos GT-S7392

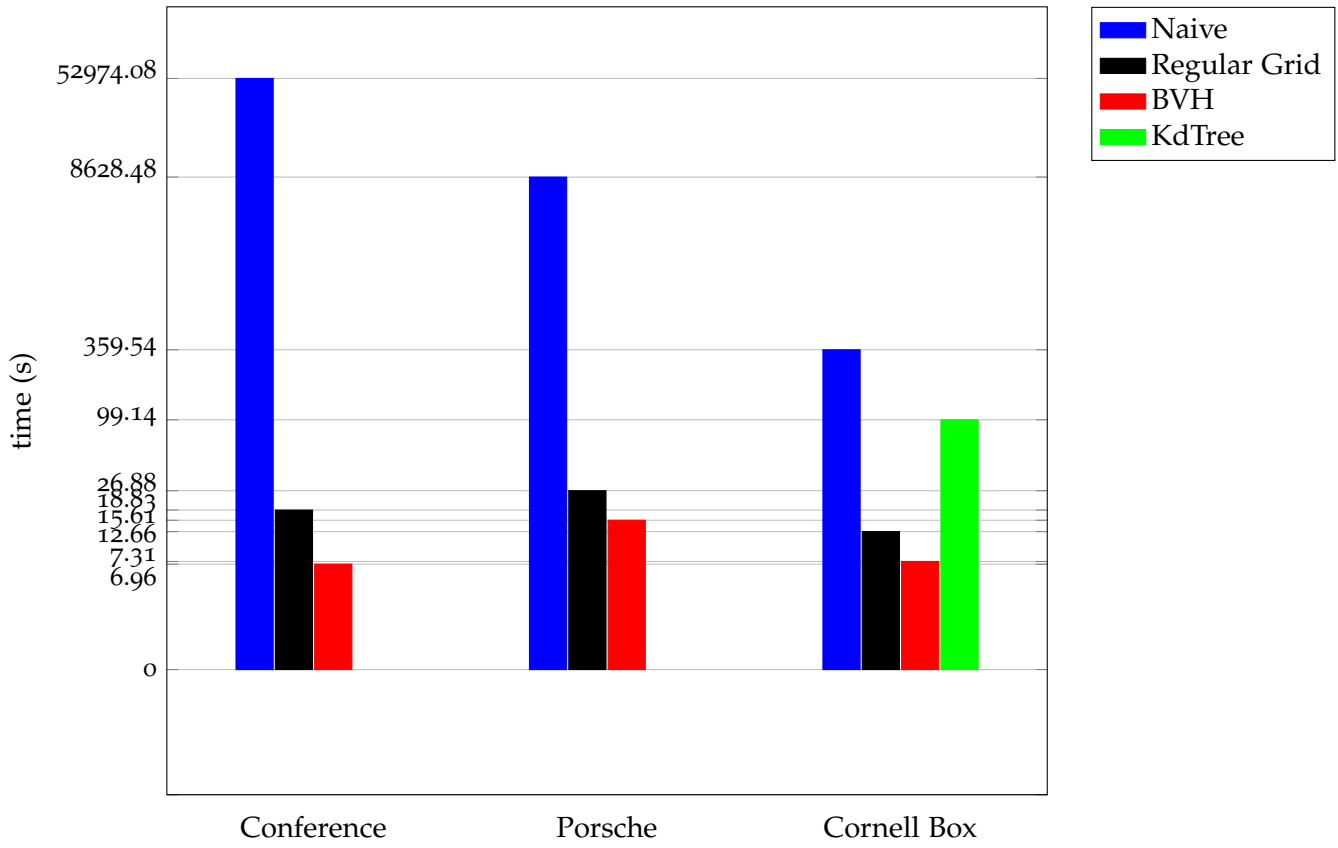


Figure 55.: Rendering times with Samsung smartphone.

The figure 55 and the table 5 show the measured rendering times in the Samsung Galaxy Fresh device. It is important to mention that the scale of the Y axis (execution time) is logarithmic, so the proportion of the graph can be misleading. Also, the resolution rendered with this device was 432x464, which is the resolution of the View where the Android UI draws the bitmap passed to the ray tracer.

Like it was expected, the rendering times with the Naive implementation is much longer than those with the acceleration structures. The Naive version took 52974.08 seconds to render the Conference scene while the Regular Grid just took 18.83 seconds which is 2812 times faster. And the BVH was even better, as it rendered the scene in just 6.965 seconds. Note however, unfortunately, the Kd-tree acceleration structure took more than 5 minutes to build in all cases, so it was only possible to let the Kd-tree building process finish in the Cornell Box scene because it has far fewer primitives.

On the other side of the graph, the Cornell Box scene, which has only 2186 triangles, took 359.54 seconds to render naively. And, with the Regular Grid and BVH, it just took 12.665 and 7.315 seconds respectively. However, the Kd-tree only could reduce the rendering time to 99.14 seconds. Unfortunately, it didn't perform as expected because it should have a rendering time lesser than the Regular Grid. This may happened because of the nature of the Kd-tree, where splitting an axis in a position where it cuts a primitive in half, makes a copy of that primitive for each child node in the Kd-tree structure.

As it could be seen, the data acceleration structures can make a huge impact in the rendering time, even in a low end mobile smartphone like the Samsung Galaxy Fresh. For this low end mobile device, the structure that caused the greatest rendering time reduction was the BVH.

Table 5.: Rendering times with Samsung smartphone.

Scene	**Accelerator**	**#Threads**	**Time (s)**	**Speedup**
Conference	Naive	1	52974.0	1.0
Conference	Reg Grid	1	18.8	2812.0
Conference	BVH	1	6.9	7605.0
Conference	Kd-tree	1	didn't finish	N/A
Porsche	Naive	1	8628.4	1.0
Porsche	Reg Grid	1	26.8	321.0
Porsche	BVH	1	15.6	552.0
Porsche	Kd-tree	1	didn't finish	N/A
Cornell Box	Naive	1	359.5	1.0
Cornell Box	Reg Grid	1	12.6	28.0
Cornell Box	BVH	1	7.3	49.0
Cornell Box	Kd-tree	1	99.1	3.6

The acceleration structures have greatly reduced the rendering time of the scene, but this improvement in speed has a cost. These structures have to be constructed before starting to render the scene and they may need much more memory than the Naive version.

As the figure 56 shows, the BVH needs 4.035 seconds to build the structure for the Conference scene. This means that, the total execution time is the time spent rendering the scene plus the time building this structure, making the total time of around 11 seconds ($6.965 + 4.035 = 11$). This is still far less than the time rendering naively. The same applies to the Regular Grid but a little bit slower by taking the total time of 24.73 seconds ($18.835 + 5.895 = 24.73$).

The other 2 scenes were much faster to build the data structures as they had less primitives. For example, the Cornell Box scene, just took 0.18 seconds to build the Regular Grid and 0.02 seconds for the BVH. Building these structures in just those times, means that they are almost free speedups as the user just needs to wait less than 1 second. The

same applies for the Porsche scene as the building times were also less than a second. Note however that the obtained execution times are not capable of rendering a scene in realtime, where the scene is rendered in each frame. The building of the acceleration structures and the rendering process take more time than the typical 16.67 ms available between frames in order to get 60 frames per second.

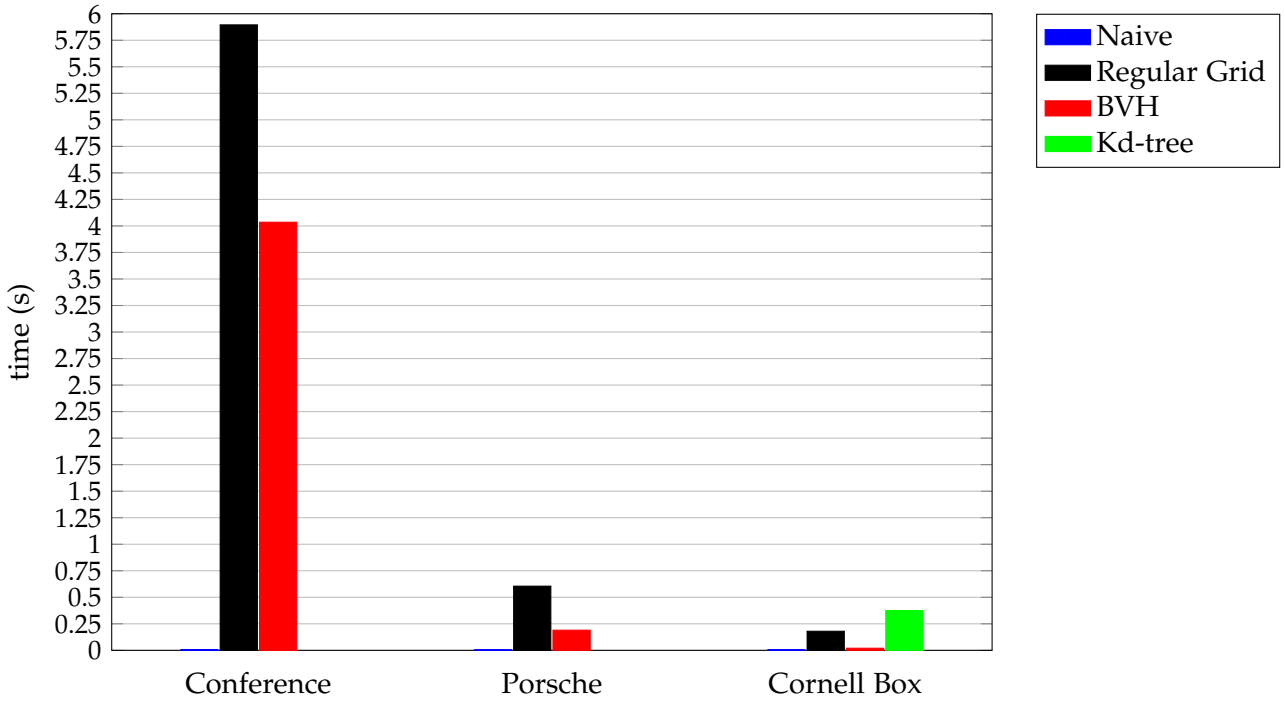


Figure 56.: Build times with Samsung smartphone.

As said before, these acceleration structures need more memory than the Naive version.

The figure 57 shows the memory consumption of the demo application while rendering with and without the accelerators. As it can be seen, all the acceleration structures have a memory consumption a little greater than the Naive version.

In the Conference scene, the Naive version consumes around 40 MB, while the Regular Grid and BVH consume 87 and 49 MB respectively. This means that the Regular Grid use 47 MB more than the Naive version, while the BVH only use extra 9 MB. This seems a lot, but extra 9 MB is just 22.5 % more memory than the Naive version, while more 47 MB is around 117.5 % more memory used. Paying extra 22.5 % of memory to have a speedup of thousands is very much worth it, even extra 117.5 % of memory for a speedup of around 2800 is worth if the device has enough memory. However, if an user wants to render a scene that barely fits in main memory of the device, then it would be impossible to build an acceleration structure for that scene.

On the smaller scenes, the extra memory used for the acceleration structures are not that high because these structures are smaller as the scene contains less primitives. For example, for the Cornell Box scene, the app demo used around 9 MB of memory in the Naive version while the Regular Grid used 13 MB of memory which is almost 1.45 times more memory. On the other side, the BVH used only 9 MB like the Naive version, which means that it just used a few hundreds of KB more than the Naive version. This, of course, makes the BVH the best default acceleration structure so far. It's also important to note that the Kd-tree used only 1 MB more of memory than the Naive version, but unfortunately, as it was seen before, the rendering time was far worse than the BVH and Regular Grid.

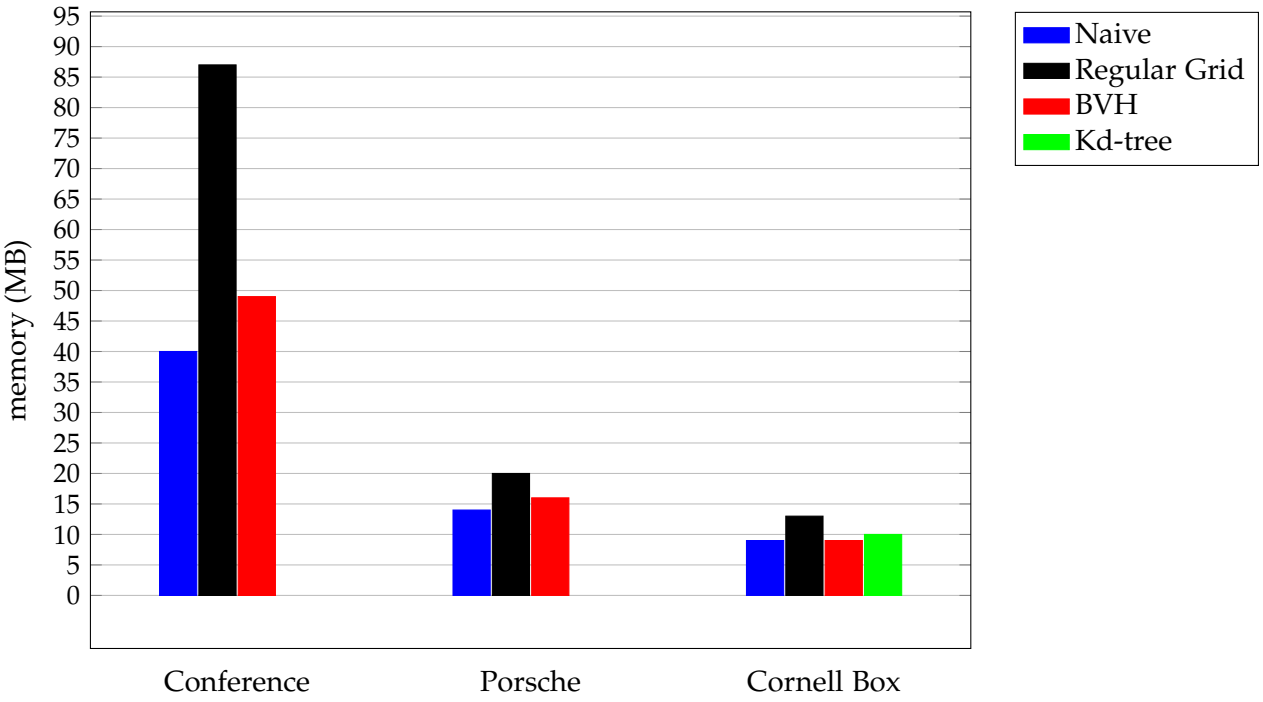


Figure 57.: Memory consumption of the demo app in Samsung smartphone.

Finally, we can't forget that all of these measurements were measured in a low budget smart phone with a low end CPU, and with just 512MB of main memory. So, in conclusion, if the device has enough memory to use an acceleration structure, it is well worth the extra time and memory used for building it. In this case, the BVH was the structure which had the most impact in the performance, in all the cases, and with the less extra memory necessary to build it.

Nokia 3.1

The tables 6, 7 and 8 show the measured rendering times of each scene for the Nokia 3.1 smart phone. In all 3 scenes, the acceleration structures reduced drastically the rendering time, just like with the previous device.

For example, in the Porsche scene, the Regular Grid and the BVH reduced from 3870.6 seconds of rendering to just 27.875 and 11.565 seconds respectively, without using more than 1 thread.

On the other side, if we just increase the number of threads in the Naive version, the speedup increases almost linearly from 1 up to 4 threads. Unfortunately, in all cases, the speedup with 8 threads was not even near the 8 times faster. This is due the developed library was not optimized to efficiently use the cache, and thus making many cache misses. It is also possible that, in the Renderer, the solution of dividing the image plane in 256 tiles and each thread gets to render an entire tile before asking for another is not very scalable. Because all the tiles don't render in similar time, as the type of materials of the primitives can be different. For example, in the Conference scene, the floor is specular while the rest of the primitives are diffuse. This can make each tile covering the floor need more time to render than the others.

Finally, by analyzing the results obtained in the Cornell Box scene, we see that, even with just a couple thousands of primitives, the acceleration structures are still worth it. In the case where it was used all 8 available threads, the 42.605 seconds spent rendering naively was reduced to just a mere 2.05 and 0.94 seconds with the Regular Grid and the BVH, respectively. This is a speedup of over 20 and 45 times compared with the Naive version with 8 threads. Once again, unfortunately, the Kd-tree was not on pair with the others structures.

Table 6.: Rendering times of Conference Room.

Accelerator	#Threads	Time (s)	Speedup
Naive	1	51 545.6	1.0
Reg Grid	1	17.6	2915.0
BVH	1	6.0	8505.0
Kd-tree	1	7771.9	6.6
Naive	2	19 649.2	2.6
Reg Grid	2	9.6	5324.0
BVH	2	3.0	16 790.0
Kd-tree	2	3847.2	13.3
Naive	4	14 078.0	3.6
Reg Grid	4	5.4	9475.0
BVH	4	1.5	32 418.0
Kd-tree	4	2670.3	19.3
Naive	8	9679.3	5.3
Reg Grid	8	3.6	13 950.0
BVH	8	1.1	46 437.0
Kd-tree	8	2159.2	23.8

Table 7.: Rendering times of Porsche.

Accelerator	#Threads	Time (s)	Speedup
Naive	1	3870.6	1.0
Reg Grid	1	27.8	147.0
BVH	1	11.5	355.0
Kd-tree	1	1553.3	2.4
Naive	2	2016.1	1.9
Reg Grid	2	14.6	281.0
BVH	2	5.7	711.0
Kd-tree	2	783.8	4.9
Naive	4	1032.3	3.7
Reg Grid	4	7.6	534.0
BVH	4	2.9	1393.0
Kd-tree	4	449.9	8.6
Naive	8	886.4	4.3
Reg Grid	8	4.8	847.0
BVH	8	1.9	2146.0
Kd-tree	8	389.0	9.9

Table 8.: Rendering times of Cornell Box.

Accelerator	#Threads	Time (s)	Speedup
Naive	1	263.4	1.0
Reg Grid	1	11.8	22.3
BVH	1	5.2	50.2
Kd-tree	1	70.7	3.7
Naive	2	130.2	2.0
Reg Grid	2	6.0	43.3
BVH	2	2.6	99.0
Kd-tree	2	35.6	7.4
Naive	4	65.8	4.0
Reg Grid	4	3.1	84.3
BVH	4	1.3	201.0
Kd-tree	4	18.3	14.3
Naive	8	42.6	6.1
Reg Grid	8	2.0	128.5
BVH	8	0.9	280.0
Kd-tree	8	12.3	21.3

Like it was said before, the speedups gained with the acceleration structures are not entirely free, as it require some building time and some extra memory.

Even with Nokia 3.1, which is a device somewhat better than Samsung Galaxy Fresh, it is required some time to build the acceleration structures. The figure 58 shows the building time of each structure. As expected, these structures took more time to be built in the Conference scene because it contains a greater number of primitives. The Regular Grid took 3.45 seconds while the BVH just took 1.705 seconds. Still, like before, those extra seconds are worth it because it speeds up the rendering time by thousands of time. The other 2 cases are similar as they also reduce rendering time drastically. It's important to note, however, the building time of Kd-tree is much higher than the other structures. It took around 1256.78 seconds to build in the Conference scene and 111.7 seconds to build in the Porsche scene. These measurements are omitted from the graph because they are too large compared to the others. In the Cornell Box scene, with 2186 primitives, it just took 0.2 seconds, but is still higher than 0.15 and 0.04 seconds from the Regular Grid and the BVH respectively.

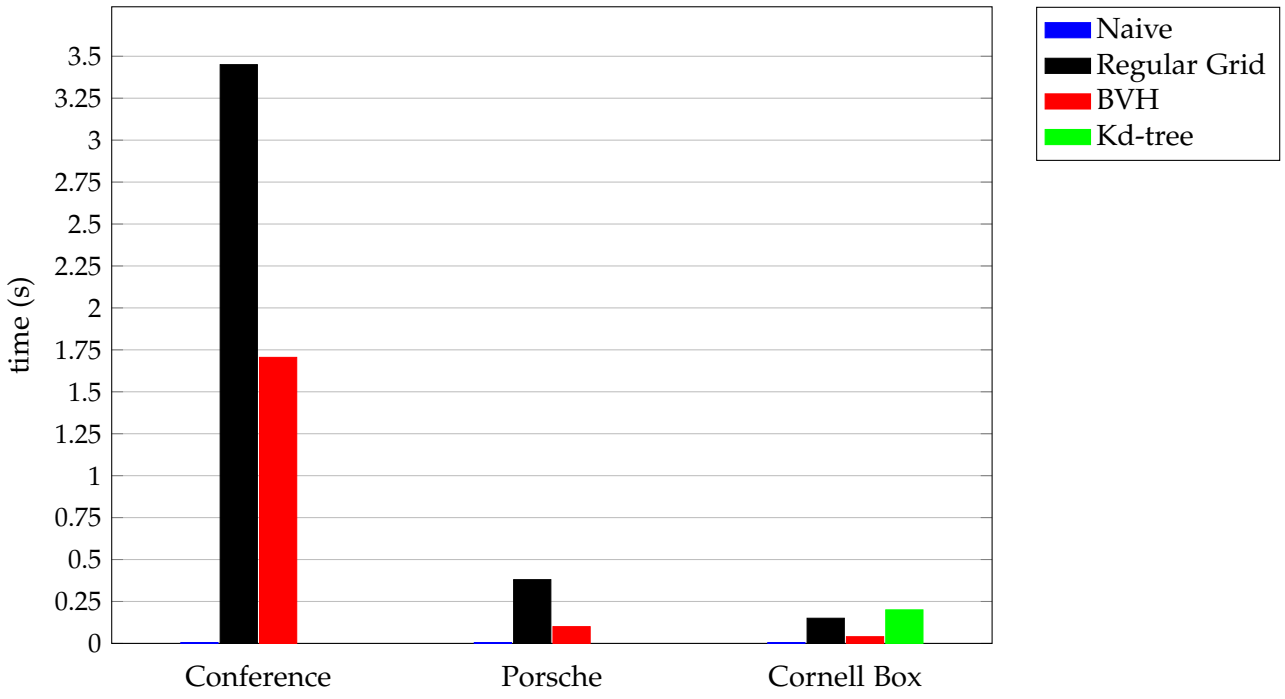


Figure 58.: Build times with Nokia 3.1 smartphone.

The figure 59 shows the memory consumption of the demo app while rendering with different acceleration structures in the Nokia 3.1. As it can be seen, everything was just like the previous case, where the Naive version is obviously the version with less memory used. And, the Regular Grid, is, once again, the one that consumes more memory. In conclusion, the BVH acceleration structure was the one which provided the most speedups and with lesser time to build and with lesser extra memory needed. And, once again, the implementation of the Kd-tree wasn't on pair with the other two.

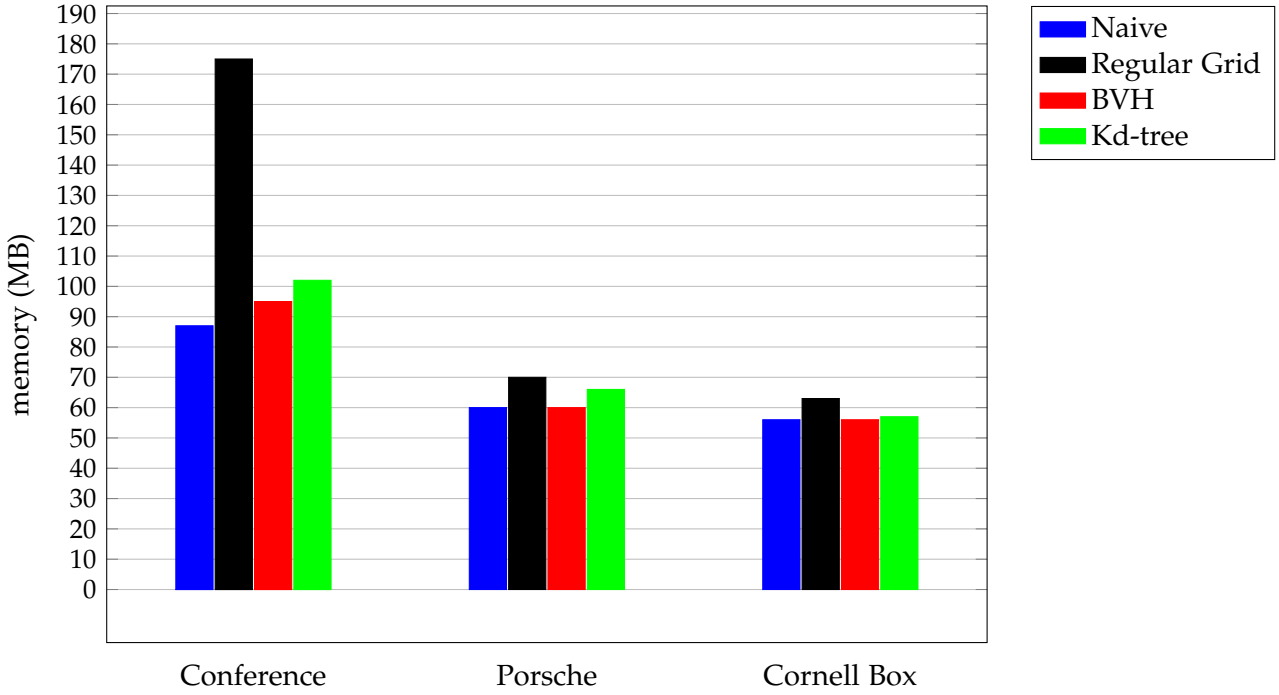


Figure 59.: Memory consumption of the demo app in Nokia 3.1 smart phone.

Nokia 7.1

The tables 9, 10 and 11 show the rendering times of the Whitted shader in the Nokia 7.1. The acceleration structures reduced drastically the rendering time in all the scenes. For example, in the scene Conference Room, the Naive version took 25978.88 seconds to render with 1 thread, while the Regular Grid and the BVH only took 21.79 and 8.48 seconds. And, once again, the Kd-tree had the worst reduction in the rendering time. Finally, in terms of scalability, this smartphone had the same behavior as the other devices, where the the rendering time reduced almost linearly with the number of threads until it reached 4 threads. Then, for 8 threads it didn't scale as expected because, as said previously, the developed library was not optimized to efficienly use the processor's cache. Also the current task scheduler implemented in the Renderer is not well balanced as one thread could take more execution time for a task than the others.

Table 9.: Rendering times of Conference Room.

Accelerator	#Threads	Time (s)	Speedup
Naive	1	25 978.8	1.0
Reg Grid	1	21.7	1192.0
BVH	1	8.4	3063.0
Kd-tree	1	10 972.3	2.3
Naive	2	17 420.8	1.4
Reg Grid	2	11.1	2335.0
BVH	2	4.2	6076.0
Kd-tree	2	7569.3	3.4
Naive	4	11 088.8	2.3
Reg Grid	4	6.1	4227.0
BVH	4	2.1	12 027.0
Kd-tree	4	3037.3	8.5
Naive	8	15 781.5	1.6
Reg Grid	8	4.2	6055.0
BVH	8	1.5	16 924.0
Kd-tree	8	3336.3	7.7

Table 10.: Rendering times of Porsche.

Accelerator	#Threads	Time (s)	Speedup
Naive	1	4230.8	1.0
Reg Grid	1	32.9	128.0
BVH	1	15.0	281.0
Kd-tree	1	1752.0	2.4
Naive	2	2053.4	2.0
Reg Grid	2	16.19	261.0
BVH	2	7.6	556.0
Kd-tree	2	889.8	4.7
Naive	4	1334.1	3.1
Reg Grid	4	8.3	503.0
BVH	4	3.8	1101.0
Kd-tree	4	498.0	8.4
Naive	8	820.7	5.1
Reg Grid	8	5.7	733.0
BVH	8	2.5	1633.0
Kd-tree	8	421.2	10.0

Table 11.: Rendering times of Cornell Box.

Accelerator	#Threads	Time (s)	Speedup
Naive	1	311.9	1.0
Reg Grid	1	12.5	24.8
BVH	1	6.3	49.1
Kd-tree	1	85.7	3.6
Naive	2	158.4	1.9
Reg Grid	2	6.2	49.8
BVH	2	3.2	96.5
Kd-tree	2	43.8	7.1
Naive	4	82.8	3.7
Reg Grid	4	3.3	91.9
BVH	4	1.7	182.4
Kd-tree	4	23.6	13.2
Naive	8	58.1	5.3
Reg Grid	8	2.2	136.8
BVH	8	1.2	251.6
Kd-tree	8	16.4	18.9

The building times of the acceleration structures are illustrated in the figure 60. The BVH was the fastest structure to build and the Kd-tree was the slowest. The building times of the Kd-tree in the Conference and Porsche scenes are not represented in the graph because

its too much slower than the others accelerators, as it took 364.53 seconds in the Conference and 43.685 in the Porsche scene.

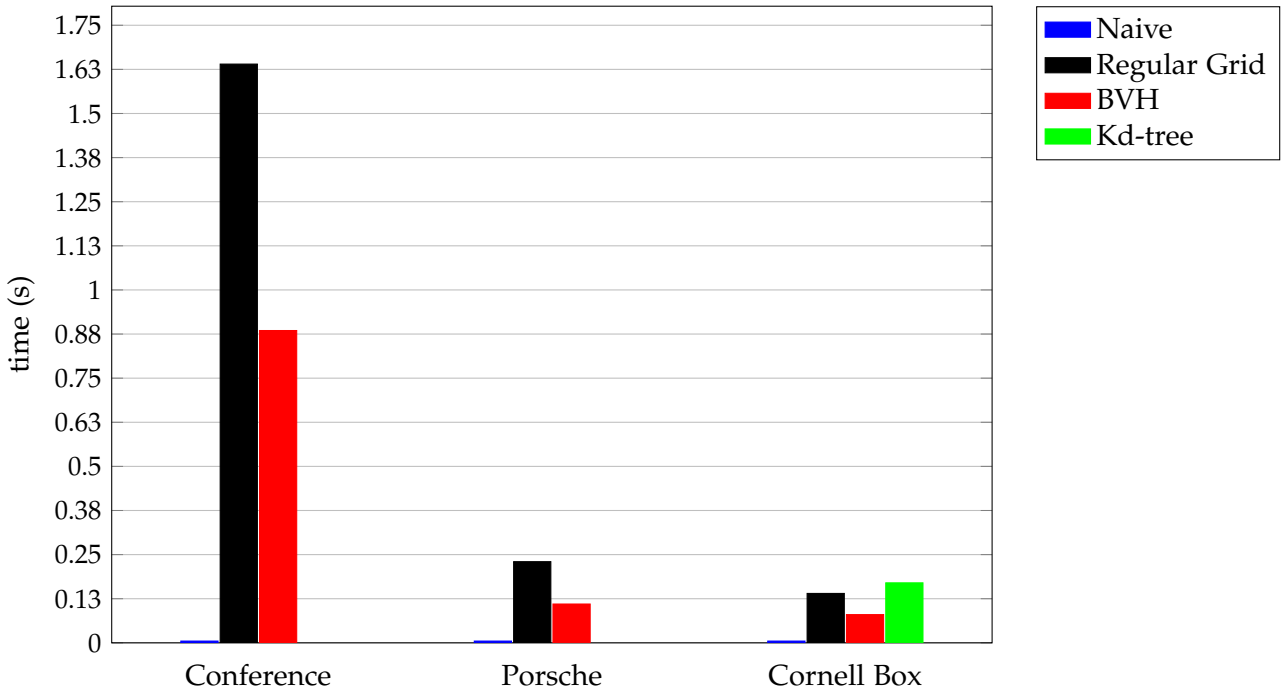


Figure 60.: Build times with Nokia 7.1 smart phone.

Finally, the memory consumption is illustrated in the figure 61. And, as it can be seen, the BVH was the structure which used less extra memory and the Regular Grid was the one which used the most.

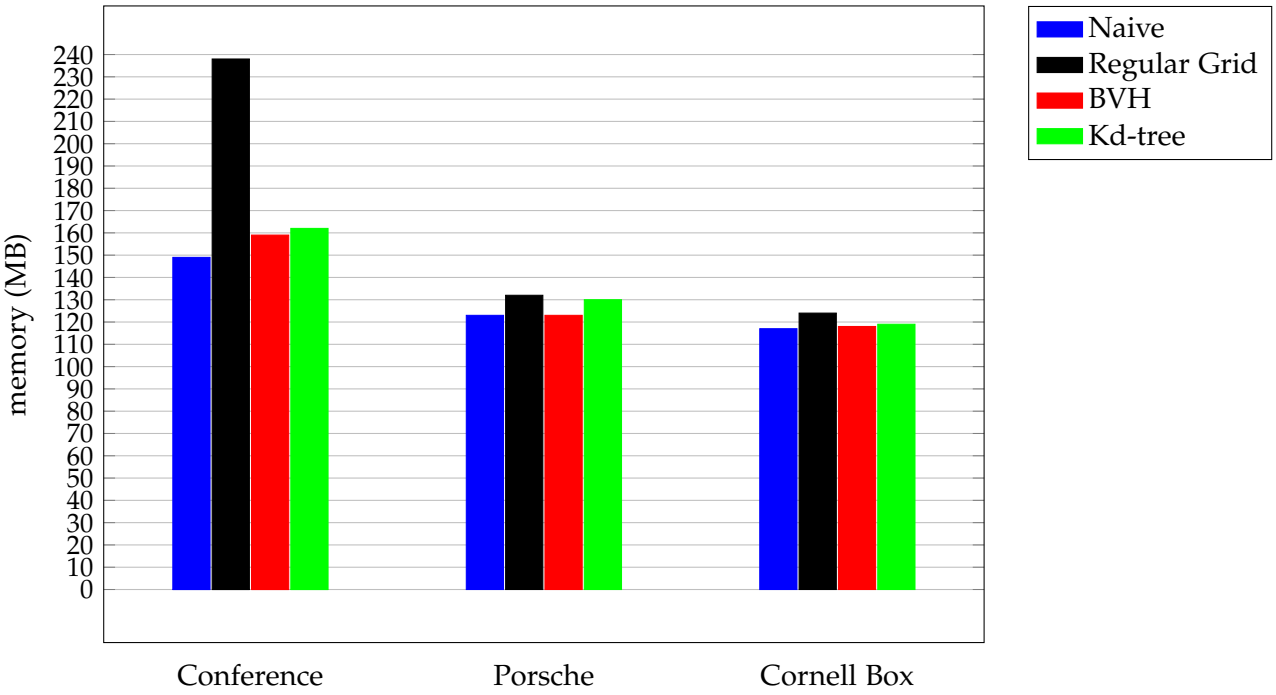


Figure 61.: Memory consumption of the demo app in Nokia 7.1 smart phone.

6.1.2 *Path Tracing Shader*

As it was seen in the measurements of the Whitted algorithm, the BVH was the acceleration structure which made the biggest impact in the performance of the developed library. The Path Tracing algorithm is very CPU intensive, because, unlike Whitted, it attempts to simulate Global Illumination. So, this time, in order to measure the rendering times of the developed Path Tracer, it was decided to only use the BVH as this structure outperformed the other two. It was also decided to compare its performance with PBRT by rendering the same scene used in the section 3.2.

Figure 62.: Scene rendered with Path Tracer algorithm at 896x896 with 8 threads, 64 spp and 1 spl.

The model rendered is a Dragon with 871306 triangles as diffuse/Lambert material inside a Cornell Box with 12 triangles as also diffuse/Lambert material, as shown in the figure 62. So, this subsection shows the median of the measured times to render a scene with the Path Tracer shader in each device. In all measurements, the scene was rendered with 64 spp and 1 spl, at the resolution of 896x896 and with 8 threads in all devices.

As it can be seen in the graph 63, the W230SS (from table 3) outperformed the Android devices as expected. What is interesting, is the fact that the developed library rendered the scene in 90.87915 seconds, while using the PBRT in the same hardware took 187.02 seconds. The PBRT required more than twice the rendering time of the developed MobileRT library. This means that the developed acceleration structures as C++ templates and the primitives without the use of inheritance did in fact impact on the overall performance of the application. As the PBRT acceleration structures and the primitives use instead inheritance for a more flexible approach.

The Android devices required more rendering time as expected, because their processors are overall simpler than the Intel® Core™ i7-4710MQ presented in the W230SS laptop. For example, the Nokia 3.1 required 327.19 seconds, which is 3.6 times slower than W230SS with the developed library, and 1.75 times slower than with PBRT. The Nokia 7.1 has a better CPU than Nokia 3.1 and was only 1.011 times slower than PBRT. This means that today's high end smartphones are already capable of rendering a scene with ray tracing at reasonable rendering times compared with PBRT while using a mid range laptop CPU from 2014. Of course that to achieve this, it is necessary to lose some flexibility in order to gain some performance as mobile devices have simpler CPUs than desktops.

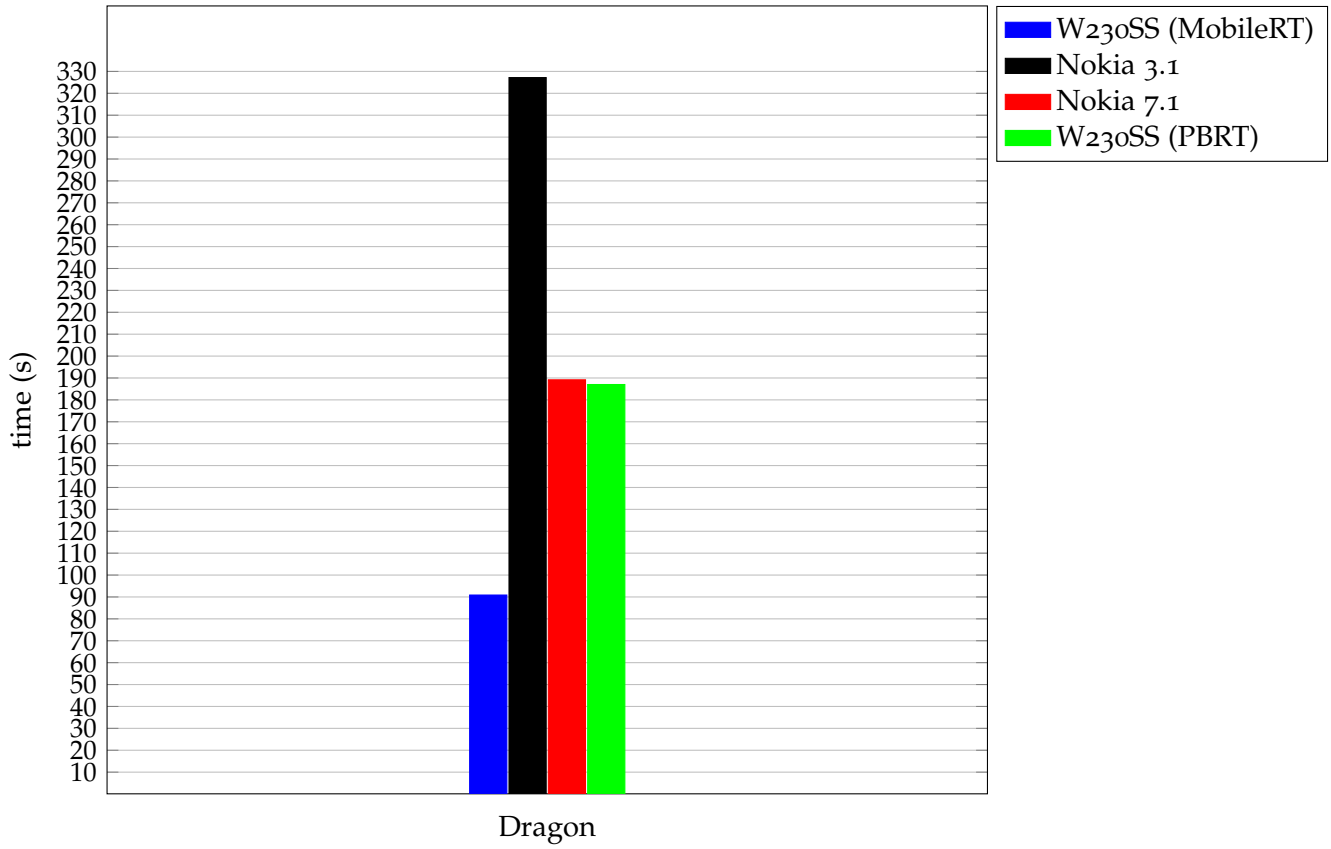


Figure 63.: Rendering times in all devices with Path Tracer shader.

6.2 COMPARISON WITH ANDROID CPU RAYTRACER (DAHLQUIST)

In the previous chapters, all the features implemented in the developed library and also in the Rendering Components made for the demo were described.

Also, in the previous sections, it was shown some rendering times obtained with the developed library. All tested scenes consist of just triangles arranged in different manners. It was also shown that the BVH acceleration structure made a big improvement in the rendering time.

The Android CPU Raytracer (Dahlquist) is the only free ray tracer engine for Android with some documentation and consists of only an application which uses ray tracing with far less features, and for a completely different objective. That ray tracer only supports one prebuilt scene with a few spheres which the user can only interact with. As, it is just a demo app, it can not be used to build some rendering applications based on ray tracing and it can't even render triangles which is the most used primitive for building complex scenes.

Table 12.: Comparison of the developed library with Android CPU Raytracer (Dahlquist).

Product	License[a]	Platform	Mobile	Interactivity	Progressive	Programmable Components
Android CPU Raytracer (Dahlquist)	O	CPU	✓ (Android)	✓	✗	✗
MobileRT	O	CPU	✓ (Android)	✗	✓	✓

[a] F - Free, C - Commercial, O - Open source

So, in conclusion, for mobile devices, the developed library is currently the only known ray tracing library available for free. And its performance was twice times faster than the PBRT in the scene rendered in the section 6.1.2.

7

CONCLUSION & FUTURE WORK

7.1 CONCLUSIONS

The constant evolution of technology leverage and massify mobile devices. With, more and increasingly powerful, mobile devices, it is possible to perform more and more complex and useful tasks in them. This is a huge market where the programmers can develop their applications too, and where the development time can be a huge factor in their career success.

But as noted in section 2.4, there is clearly a lack of well documented rendering libraries for mobile systems. There is then the need to change this reality, since more and more the Internet of Things is more present each passing year, because there are increasingly powerful mobile systems. Rendering libraries like this can be used to simulate the propagation of light, as demonstrated in the demo application. It can also be used to simulate the propagation of sound or even wifi networks.

In this dissertation, it was developed a ray tracing library which allows the execution of the basic features of a ray tracer, such as the intersection of rays with the different shapes of primitives. Besides that, it allows the programmer the possibility to develop several types of rendering components, like cameras, lights, object loaders, samplers and shaders. With this, the programmer can create rendering applications that just use the rendering components offered together with the library, or even create very complex applications that use rendering components developed by the programmer himself.

During the development of this ray tracer library, it was identified some challenges caused by the fact that it was developed for an Android mobile device, like, the smaller amount of RAM available for the software applications. And, also, the simpler CPUs microarchitectures and smaller computational power available from these devices can make computational demanding applications, like ray tracers, difficult to perform the required calculations in a useful time for the user. This fact is corroborated by the obtained results as demonstrated in section 6.1. The mobile devices do, in fact, require more time to render a scene with ray tracing than laptop or desktop computers, as demonstrated in the section 6.1.2. So, realtime ray tracing is obviously out of question for mobile devices, but offline

rendering is still feasible with some optimized acceleration structure like the BVH. In the developed library, generic programming techniques were used in the implementation of these structures, and that proved to make a big impact in the overall performance of the application, but at the expense of some flexibility for the programmer. The developed library proved to be capable of using all the available cores in a mobile device and, as demonstrated, it is portable for other systems besides Android.

7.2 FUTURE WORK

The developed library allows the user to implement rendering applications in an easy, safe and fast way, as it was intended in this dissertation. Unfortunately, this does not allow rapid development of all kinds of rendering applications. One of the things that can be improved in this library is allowing to put more than one camera in the scene, so that it can cast rays to the scene from more than one position.

In section 6, it was analyzed the performance of the 3 developed acceleration structures. And it was observed that the KD-Tree structure had rendering and building times much greater than the other two structures. This means that it is still possible to optimize it even further, and on top of that, taking advantage of SIMD instructions available in the devices by using ray packets instead of single rays. And thus allowing the possibility to intersect a single primitive with multiple rays, making the code even more efficient. Also, there is always the possibility to improve the ray tracer performance by redeveloping the library using a Data-oriented design (Nikolov) instead of the traditional Object-oriented design used. This approach was motivated by taking advantage of cache coherency, used in video game development, usually in the programming languages C or C++.

Last but not least important, it could be interesting to develop a rendering library capable of sharing the scene geometry across multiple mobile devices and let all contribute to the rendering of it simultaneously, as it was proven that one device alone may not render a complex scene in a useful time.

8

BIBLIOGRAPHY

AMD. Radeon rays technology for developers. URL http://developer.amd.com/tools-and-sdks/graphics-development/radeonpro/radeonrays-technology-developers/. Accessed: January 2017.

Tavian Barnes. Fast, branchless ray/bounding box intersections, part 2: Nans. URL https://tavianator.com/fast-branchless-raybounding-box-intersections-part-2-nans/. Accessed: January 2019.

Armen Barsegyan. Reality check. what is nvidia rtx technology? what is directx dxr? here's what they can and cannot do. URL http://cgicoffee.com/blog/2018/03/what-is-nvidia-rtx-directx-dxr. Accessed: January 2019.

belthaczar. Ray/plane intersection point. URL http://www.idevgames.com/forums/thread-5744.html. Accessed: January 2019.

Russ Bishop. Swift 2: Simd. URL http://www.russbishop.net/?page=5&x=272. Accessed: January 2019.

Christopher Chedeau. jsraytracer. URL http://blog.vjeux.com/2012/javascript/javascript-ray-tracer.html. Accessed: January 2017.

Chirag. Ray sphere intersection. URL http://ray-tracing-conept.blogspot.pt/2015/01/ray-sphere-intersection.html. Accessed: January 2019.

COOLFINESSE. 5 reasons we love android devices, 2017. URL https://iheanyiigboko.wordpress.com/2017/09/18/5-reasons-we-love-android-devices/. Accessed: January 2019.

G-Truc Creation. Opengl mathematics. URL https://github.com/g-truc/glm. Accessed: January 2019.

Nic Dahlquist. Android cpu raytracer. URL https://github.com/ndahlquist/raytracer. Accessed: January 2017.

Joey de Vries. Learnopengl. URL https://learnopengl.com. Accessed: January 2019.

Valgrind™ Developers. Valgrind. URL http://valgrind.org/. Accessed: January 2019.

Academic Dictionaries and Encyclopedias. Instruction pipeline. URL http://enacademic.com/dic.nsf/enwiki/141209. Accessed: January 2019.

Luís Paulo Peixoto dos Santos. Ray tracing clássico. URL gec.di.uminho.pt/psantos/VI2/Acetatos/02-RayTracing.ppsx. Accessed: January 2019.

Le Journal du Net. Gérer et sécuriser ses flottes de smartphones et tablettes. URL https://www.journaldunet.com/solutions/mobilite/mobile-device-management-comparatif-des-offres. Accessed: January 2019.

John Feminella. How to set up quadratic equation for a ray/sphere intersection? URL https://stackoverflow.com/questions/1986378/how-to-set-up-quadratic-equation-for-a-ray-sphere-intersection. Accessed: January 2019.

António José Borba Ramires Fernandes. Ray-triangle intersection. URL http://www.lighthouse3d.com/tutorials/maths/ray-triangle-intersection/. Accessed: January 2019.

The Foundry. Casting shadows. URL https://learn.foundry.com/nuke/8.0/content/user_guide/3d_compositing/casting_shadows.html. Accessed: January 2019.

Free3D. Free 3d models. URL https://free3d.com/3d-models/. Accessed: January 2019.

© 2019 Galaxie. Le raytracing peut-il supplanter la rastérisation ? URL https://www.tomshardware.fr/le-raytracing-peut-il-supplanter-la-rasterisation/2/. Accessed: January 2019.

Google. Android developers, a. URL https://developer.android.com/index.html. Accessed: May 2017.

Google. Google test, b. URL https://github.com/google/googletest. Accessed: January 2019.

Brendan Gregg. perf examples. URL http://www.brendangregg.com/overview.html. Accessed: January 2019.

group of C++ enthusiasts. C++ standard library header files. URL https://en.cppreference.com/w/cpp/header. Accessed: January 2019.

Rodrigo Placencia & David Bluecame & Olaf Arnold & Michele Castigliego Gustavo Pichorim Boiko. Yafaray. URL http://www.yafaray.org/. Accessed: January 2017.

© OTOY Inc. Real-time 3d rendering. URL https://home.otoy.com/render/octane-render/. Accessed: January 2017.

GitHub Inc. The world's leading software development platform, a. URL https://github.com/. Accessed: January 2019.

The Khronos™ Group Inc. Opengl es overview, b. URL https://www.khronos.org/opengles/. Accessed: January 2019.

Intel. High performance ray tracing kernels. URL https://embree.github.io/index.html. Accessed: January 2017.

Alec Jacobson. Barycentric coordinates and point-triangle queries. URL http://www.alecjacobson.com/weblog/?p=1596. Accessed: January 2019.

Wenzel Jakob. Mitsuba renderer. URL http://www.mitsuba-renderer.org. Accessed: January 2017.

Bla... joojaa. How do i use barycentric coordinates to interpolate vertex normal? URL https://computergraphics.stackexchange.com/questions/5006/how-do-i-use-barycentric-coordinates-to-interpolate-vertex-normal. Accessed: January 2019.

Michael Karbo and ELI Aps. Chapter 28. the cache. URL http://www.karbosguide.com/books/pcarchitecture/chapter28.htm. Accessed: January 2019.

Kenneth. Hray - a haskell ray tracer. URL http://kejo.be/ELIS/Haskell/HRay/. Accessed: January 2017.

Mark Kilgard. Cs 354 acceleration structures. URL https://www.slideshare.net/Mark_Kilgard/26accelstruct. Accessed: January 2019.

kitware. Cmake. URL https://cmake.org/. Accessed: January 2019.

Iggy Krajci and Darren Cummings. Android on x86: Java native interface and the android native development kit. URL http://www.drdobbs.com/architecture-and-design/android-on-x86-java-native-interface-and/240166271. Accessed: January 2019.

Yurii Lahodiuk. Kd tree for triangle meshes is too slow. URL https://stackoverflow.com/questions/20019110/kd-tree-for-triangle-meshes-is-too-slow. Accessed: January 2019.

Glare Technologies Limited. Indigo rt. URL http://www.indigorenderer.com/indigo_rt. Accessed: January 2017.

Greg Humphreys Matt Pharr, Wenzel Jakob. Physically based rendering. URL http://pbrt.org/. Accessed: January 2017.

Morgan McGuire. Computer graphics archive, July 2017. URL https://casual-effects.com/data. Accessed: January 2019.

Vlastimil Havran Michal Hapala. Review: Kd-tree traversal algorithms for ray tracing. URL http://dcgi.felk.cvut.cz/publications/2011/hapala-cgf-kdtree. Accessed: January 2019.

Microsoft. Guidelines support library. URL https://github.com/Microsoft/GSL. Accessed: January 2019.

Tomas Moller and Ben Trumbore. Fast minimum storage ray/triangle intersection, 1997. URL https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf. Accessed: January 2019.

Michael Mara Morgan. The G3D innovation engine. URL http://g3d.cs.williams.edu/. Accessed: January 2017.

Stoyan Nikolov. Oop is dead, long live data-oriented design. URL https://www.youtube.com/watch?v=yy8jQgmhbAU. Accessed: January 2019.

Nvidia. Nvidia® optix™ ray tracing engine, a. URL https://developer.nvidia.com/optix. Accessed: January 2017.

Nvidia. Baking with optix, b. URL https://developer.nvidia.com/optix-prime-baking-sample. Accessed: January 2017.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation (3rd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, October 2016. ISBN 9780128006450.

Mini Physics. Refraction of light. URL https://www.miniphysics.com/refraction-of-light.html. Accessed: January 2019.

Pixar. Pixar ris. URL https://renderman.pixar.com/resources/current/RenderMan/risOverview.html. Accessed: January 2017.

Raph Schim. From perspective picture to orthographic picture. URL https://stackoverflow.com/questions/36573283/from-perspective-picture-to-orthographic-picture. Accessed: January 2019.

Computer Science and Engineering. Anti aliasing computer graphics. URL https://www.slideshare.net/DelwarHossain8/anti-aliasing-computer-graphics. Accessed: January 2019.

Scratchapixel. Ray tracing: Rendering a triangle. URL https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle. Accessed: January 2019.

© StatCounter. Mobile operating system market share worldwide. URL http://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: January 2019.

syoyo. tinyobjloader. URL https://github.com/syoyo/tinyobjloader. Accessed: January 2019.

GCC team. Options that control optimization, a. URL https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: January 2019.

GCC team. Gcc, the gnu compiler collection, b. URL https://gcc.gnu.org/. Accessed: January 2019.

The Clang Team. Clang: a c language family frontend for llvm, a. URL https://clang.llvm.org/. Accessed: January 2019.

The Clang Team. Clang-tidy, b. URL https://clang.llvm.org/extra/clang-tidy/. Accessed: January 2019.

Daniel A. Thompson. Ray tracing glossy reflection: sampling ray direction. URL https://stackoverflow.com/questions/32077952/ray-tracing-glossy-reflection-sampling-ray-direction. Accessed: January 2019.

Raghu Machiraju Torsten Möller. Ray intersection acceleration. URL http://slideplayer.com/slide/7981872/. Accessed: January 2019.

Will Usher. upacket - a micro packet ray tracer, a. URL https://github.com/Twinklebear/micro-packet. Accessed: January 2017.

Will Usher. tray - a toy ray tracer, b. URL https://github.com/Twinklebear/tray. Accessed: January 2017.

Will Usher. tray_rust - a toy ray tracer in rust, c. URL https://github.com/Twinklebear/tray_rust. Accessed: January 2017.

Menno Vink. Ray to plane intersection. URL http://www.echo-gaming.eu/ray-to-plane-intersection/. Accessed: January 2019.

Marek Vinkler, Vlastimil Havran, and Jiri Bittner Bittner. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. URL http://doi.acm.org/10.1145/2643188.2643196. Accessed: January 2019.

Marsette Vona and Tong Pan. Computer graphics (cs 4300) 2010s: Lecture 21. URL https://course.ccs.neu.edu/cs4300old/s10/L21/L21.html. Accessed: January 2019.

Zack Waters. Realistic raytracing. URL https://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/realistic_raytracing.html. Accessed: January 2019.

Wikipedia. Superscalar processor, a. URL https://en.wikipedia.org/wiki/Superscalar_processor. Accessed: January 2019.

Wikipedia. Constant (computer programming), b. URL https://en.wikipedia.org/wiki/Constant_(computer_programming). Accessed: January 2019.

Wikipedia. Depth map, c. URL https://en.wikipedia.org/wiki/Depth_map. Accessed: January 2019.

Wikipedia. Halton sequence, d. URL https://en.wikipedia.org/wiki/Halton_sequence. Accessed: January 2019.

Wikipedia. Metropolis light transport, e. URL https://en.wikipedia.org/wiki/Metropolis_light_transport. Accessed: January 2019.

Wikipedia. Path tracing, f. URL https://en.wikipedia.org/wiki/Path_tracing. Accessed: January 2019.

Wikipedia. Rendering equation, g. URL https://en.wikipedia.org/wiki/Rendering_equation. Accessed: January 2019.

Wikipedia. Spherical coordinate system, h. URL https://en.wikipedia.org/wiki/Spherical_coordinate_system. Accessed: January 2019.

Alan Wolfe. Randomcode. URL https://github.com/Atrix256/RandomCode. Accessed: January 2019.

Stefan Zellmann. Visionaray. URL https://github.com/szellmann/visionaray. Accessed: January 2017.

A

API

```
public:
explicit Renderer () noexcept = delete;

explicit Renderer(::std::unique_ptr<Shader> shader,
  ::std::unique_ptr<Camera> camera,
  ::std::unique_ptr<Sampler> samplerPixel,
  uint32_t width, uint32_t height,
  uint32_t samplesPixel,
  ::std::function<::glm::vec3(const ::glm::vec3 &,
  uint32_t)> toneMapper
) noexcept;

Renderer(const Renderer &renderer) noexcept = delete;

Renderer(Renderer &&renderer) noexcept = delete;

~Renderer() noexcept = default;

Renderer &operator=(const Renderer &renderer) noexcept = delete;

Renderer &operator=(Renderer &&renderer) noexcept = delete;


void renderFrame(uint32_t *bitmap, int32_t numThreads, uint32_t stride) noexcept;

void stopRender() noexcept;

uint32_t getSample() const noexcept;

const ::std::vector<::glm::vec3> &getVecColors() const noexcept;
```

Listing A.1: Renderer API

```
public:
explicit Scene() = default;

Scene(const Scene &scene) noexcept = delete;

Scene(Scene &&scene) noexcept = default;

~Scene() noexcept;

Scene &operator=(const Scene &scene) noexcept = delete;

Scene &operator=(Scene &&scene) noexcept = default;


bool traceLights(Intersection *intersection, const Ray &ray) const noexcept;

void resetSampling() noexcept;

void clearPrimitives() noexcept;
```

Listing A.2: Scene API

```
public:
explicit Ray () noexcept = delete;

explicit Ray(const ::glm::vec3 &dir, const ::glm::vec3 &origin,
int32_t depth, const void *primitive = nullptr) noexcept;

Ray(const Ray &ray) noexcept = default;

Ray(Ray &&ray) noexcept = default;

~Ray() noexcept = default;

Ray &operator=(const Ray &ray) noexcept = delete;

Ray &operator=(Ray &&ray) noexcept = delete;
```

Listing A.3: Ray API

```
public:
explicit Intersection () noexcept = delete;

```

```
4  explicit Intersection(float dist) noexcept;
5
6  explicit Intersection(
7  const ::glm::vec3 &intPoint,
8  float dist,
9  const ::glm::vec3 &normal,
10 const void *primitive, int32_t materialId,
11 ::glm::vec2 texCoords = ::glm::vec2 {-1}) noexcept;
12
13 Intersection(const Intersection &intersection) = default;
14
15 Intersection(Intersection &&intersection) = default;
16
17 ~Intersection() noexcept = default;
18
19 Intersection &operator=(const Intersection &intersection) noexcept = delete;
20
21 Intersection &operator=(Intersection &&intersection) noexcept = default;
```

Listing A.4: Intersection API

```
1  public:
2  explicit Material () noexcept = default;
3
4  explicit Material(
5   const ::glm::vec3 &Kd,
6   const ::glm::vec3 &Ks = ::glm::vec3 {},
7   const ::glm::vec3 &Kt = ::glm::vec3 {},
8   float refractiveIndice = 1.0f, const ::glm::vec3 &Le = ::glm::vec3 {},
9   ::std::vector<Texture> &&textures = {}) noexcept;
10
11 Material(const Material &material) = default;
12
13 Material(Material &&material) = default;
14
15 ~Material() noexcept = default;
16
17 Material &operator=(const Material &material) noexcept;
18
19 Material &operator=(Material &&material) = default;
20
21
22 bool operator==(const Material &material) const noexcept;
```

Listing A.5: Material API

```
public:
explicit Triangle () noexcept = delete;

explicit Triangle(
  const ::glm::vec3 &pointA, const ::glm::vec3 &pointB,
  const ::glm::vec3 &pointC, int32_t materialId) noexcept;

explicit Triangle(
  const ::glm::vec3 &pointA, const ::glm::vec3 &pointB, const ::glm::vec3 &pointC,
  int32_t materialId,
  const ::glm::vec3 &normalA, const ::glm::vec3 &normalB, const ::glm::vec3 &normalC) noexcept;

explicit Triangle(
  const ::glm::vec3 &pointA, const ::glm::vec3 &pointB, const ::glm::vec3 &pointC,
  int32_t materialId,
  const ::glm::vec3 &normalA, const ::glm::vec3 &normalB, const ::glm::vec3 &normalC,
  const ::glm::vec2 &textureCoordenatesA,
  const ::glm::vec2 &textureCoordenatesB,
  const ::glm::vec2 &textureCoordenatesC) noexcept;

explicit Triangle(
  const ::glm::vec3 &pointA, const ::glm::vec3 &pointB, const ::glm::vec3 &pointC,
  int32_t materialId,
  const ::glm::vec2 &textureCoordenatesA,
  const ::glm::vec2 &textureCoordenatesB,
  const ::glm::vec2 &textureCoordenatesC) noexcept;

Triangle(const Triangle &triangle) = default;

Triangle(Triangle &&triangle) noexcept = default;

~Triangle() noexcept = default;

Triangle &operator=(const Triangle &triangle) noexcept = delete;

Triangle &operator=(Triangle &&triangle) noexcept = delete;


bool intersect(Intersection *intersection, const Ray &ray) const noexcept;

```

```
41 bool intersect(const Ray &ray, float dist) const noexcept;
42
43 bool intersectBox(const AABB &box) const noexcept;
44
45 AABB getBoundingBox() const noexcept;
```

Listing A.6: Triangle API

```
1  public:
2  explicit Plane () noexcept = delete;
3
4  explicit Plane(const ::glm::vec3 &point, const ::glm::vec3 &normal,
5    int32_t materialId) noexcept;
6
7  Plane(const Plane &plane) noexcept = default;
8
9  Plane(Plane &&plane) noexcept = default;
10
11 ~Plane() noexcept = default;
12
13 Plane &operator=(const Plane &plane) noexcept = delete;
14
15 Plane &operator=(Plane &&plane) noexcept = delete;
16
17
18 bool intersect(Intersection *intersection, const Ray &ray) const noexcept;
19
20 bool intersect(const Ray &ray, float dist) const noexcept;
21
22 bool intersectBox(const AABB &box) const noexcept;
23
24 AABB getBoundingBox() const noexcept;
```

Listing A.7: Plane API

```
1  public:
2  explicit Sphere () noexcept = delete;
3
4  explicit Sphere(const ::glm::vec3 &center, float radius, int32_t materialId) noexcept;
5
6  Sphere(const Sphere &sphere) noexcept = default;
7
8  Sphere(Sphere &&sphere) noexcept = default;
```

```
9
10 ~Sphere() noexcept = default;
11
12 Sphere &operator=(const Sphere &sphere) noexcept = delete;
13
14 Sphere &operator=(Sphere &&sphere) noexcept = delete;
15
16
17 bool intersect(Intersection *intersection, const Ray &ray) const noexcept;
18
19 bool intersect(const Ray &ray, float dist) const noexcept;
20
21 bool intersectBox(const AABB &box) const noexcept;
22
23 AABB getBoundingBox() const noexcept;
```

Listing A.8: Sphere API

```
1 #define LOG(...) {\
2   ::MobileRT::log(::MobileRT::getFileName(__FILE__), ":", __LINE__, ": ", __VA_ARGS__);}
3
4 template<typename T>
5 ::std::vector<const T *> createPointerVector(const ::std::vector<T> &source) noexcept;
6
7 bool equals(float a, float b) noexcept;
8 bool equals(::glm::vec3 a, ::glm::vec3 b) noexcept;
9
10 int32_t roundDownToMultipleOf(int32_t value, int32_t multiple) noexcept;
11
12 uint32_t roundUpToPowerOf2(uint32_t value) noexcept;
13
14 uint32_t usedBitsCounter(uint32_t n) noexcept;
15
16 float haltonSequence(uint64_t index, uint64_t base) noexcept;
17
18 ::glm::vec3
19 toneMap(const ::glm::vec3 &colorAccumulate, uint32_t numSample, float gamma) noexcept;
20
21 uint32_t convertVec3ToIntColor(const ::glm::vec3 &color) noexcept;
22
23 float balanceHeuristic(float pdf0, float pdf1) noexcept;
24
25 float powerHeuristic(uint32_t num0, float pdf0, uint32_t num1, float pdf1) noexcept;
```

```
26
27 ::glm::vec3 getCosineWeightedHemisphereSample(const ::glm::vec3 &normal) noexcept;
28
29 float fresnelEquation(const ::glm::vec3 &I, const ::glm::vec3 &N, const float &iorIn,
30  const float &iorGoing) noexcept;
31
32 Texture createTextureFromFile (const ::std::string &filename, Texture::TextureType textureType)
33  noexcept;
34
35 float degToRad(float deg) noexcept;
36
37 float radToDeg(float rad) noexcept;
38
39 void sumBox(const ::MobileRT::AABB &primitiveBox, ::MobileRT::AABB *box) noexcept;
40
41 template<typename T>
42 void getBounds(const ::std::vector<T*> &primitives,
43 ::MobileRT::AABB *box) noexcept;
44
45 float getArcTan(float radians) noexcept;
```

Listing A.9: Utils API

```
1 public:
2 explicit AABB() noexcept = default;
3
4 explicit AABB(const ::glm::vec3 &pointMin, const ::glm::vec3 &pointMax) noexcept;
5
6 AABB(const AABB &aabb) noexcept = default;
7
8 AABB(AABB &&aabb) noexcept = default;
9
10 ~AABB() noexcept = default;
11
12 AABB &operator=(const AABB &aabb) noexcept = default;
13
14 AABB &operator=(AABB &&aabb) noexcept = default;
15
16
17 bool intersect(const Ray &ray, const ::glm::vec3 &rayInvDir) const noexcept;
18
19 bool intersect(const Ray &ray, const ::glm::vec3 &rayInvDir,
20 float *tmin, float *tmax) const noexcept;
```

```
21
22 float getSurfaceArea() const noexcept;
23
24 ::glm::vec3 getCentroid() const noexcept;
25
26
27 AABB getSurroundingBox(const AABB &box1, const AABB &box2) noexcept;
28
29 void getSurroundingBox(float box1[6], const float box2[6]) noexcept;
30
31 float getSurfaceArea(const float box[6]) noexcept;
```

Listing A.10: AABB API

```
1 public:
2 explicit BVH() noexcept = default;
3
4 explicit BVH<T> (::std::vector<T> &&primitives) noexcept;
5
6 BVH(const BVH &bVH) noexcept = delete;
7
8 BVH(BVH &&bVH) noexcept = default;
9
10 ~BVH() noexcept;
11
12 BVH &operator=(const BVH &bVH) noexcept = delete;
13
14 BVH &operator=(BVH &&bVH) noexcept = default;
15
16
17 bool findNearestIntersection(Intersection *intersection, const Ray &ray) const noexcept;
18
19 bool findIntersection(const Ray &ray, float dist) const noexcept;
20
21 const ::std::vector<T> &getPrimitives () const noexcept;
```

Listing A.11: BVH API

```
1 public:
2 explicit RegularGrid() noexcept = default;
3
4 explicit RegularGrid<T> (::std::vector<T> &&primitives, AABB sceneBounds, int32_t gridSize)
5   noexcept;
```

```
6
7  RegularGrid(const RegularGrid &regularGrid) noexcept = delete;
8
9  RegularGrid(RegularGrid &&regularGrid) noexcept = default;
10
11 ~RegularGrid() noexcept;
12
13 RegularGrid &operator=(const RegularGrid &regularGrid) noexcept = delete;
14
15 RegularGrid &operator=(RegularGrid &&regularGrid) noexcept = default;
16
17
18 bool findNearestIntersection(Intersection *intersection, const Ray &ray) const noexcept;
19
20 bool findIntersection(const Ray &ray, float dist) const noexcept;
21
22 const ::std::vector<T> &getPrimitives () const noexcept;
```

Listing A.12: RegularGrid API

```
1  public:
2  explicit Naive() noexcept = default;
3
4  explicit Naive(::std::vector<T> &&primitives) noexcept;
5
6  Naive(const Naive &naive) noexcept = delete;
7
8  Naive(Naive &&naive) noexcept = default;
9
10 ~Naive() noexcept;
11
12 Naive &operator=(const Naive &naive) noexcept = delete;
13
14 Naive &operator=(Naive &&naive) noexcept = default;
15
16
17 bool findNearestIntersection(Intersection *intersection, const Ray &ray) const noexcept;
18
19 bool findIntersection(const Ray &ray, float dist) const noexcept;
20
21 const ::std::vector<T> &getPrimitives () const noexcept;
```

Listing A.13: Naive API

```
public:
explicit KdTree() noexcept = default;

explicit KdTree<T> (::std::vector<T> &&primitives) noexcept;

KdTree(const KdTree &kdTree) noexcept = delete;

KdTree(KdTree &&kdTree) noexcept = default;

~KdTree() noexcept;

KdTree &operator=(const KdTree &kdTree) noexcept = delete;

KdTree &operator=(KdTree &&kdTree) noexcept = default;


bool findNearestIntersection(Intersection *intersection, const Ray &ray) const noexcept;

bool findIntersection(const Ray &ray, float dist) const noexcept;

const ::std::vector<T> &getPrimitives () const noexcept;
```

Listing A.14: KD-Tree API

```
protected:
virtual bool shade(
::glm::vec3 *rgb, const Intersection &intersection, const Ray &ray,
bool lightEmission = true) const noexcept = 0;

::MobileRT::Light &getRandomLight () const noexcept;

public:
explicit Shader () noexcept = delete;

explicit Shader(
  Scene &&scene,
  uint32_t lightsSamples,
  Accelerator accelerator) noexcept;

Shader(const Shader &shader) noexcept = delete;

Shader(Shader &&shader) noexcept = default;

```

```
20 virtual ~Shader() noexcept;
21
22 Shader &operator=(const Shader &shader) noexcept = delete;
23
24 Shader &operator=(Shader &&shader) noexcept = delete;
25
26
27 bool rayTrace(::glm::vec3 *rgb, Intersection *intersection, const Ray &ray, bool lightEmission = true)
28   const noexcept;
29
30 bool shadowTrace(const Ray &ray, float dist) const noexcept;
31
32 virtual void resetSampling() noexcept;
33
34 void initializeAccelerators() noexcept;
35
36 const ::std::vector<Triangle>& getTriangles() const noexcept;
37
38 const ::std::vector<Material>& getMaterials() const noexcept;
39
40
41 ::glm::vec3 sampleLights(const ::glm::vec3 &origin, const ::glm::vec3 &normal) const noexcept;
42
43 Intersection trace(const Ray &ray) const noexcept;
```

Listing A.15: Shader API

```
1 public:
2 explicit Sampler() noexcept = default;
3
4 explicit Sampler(uint64_t domainSize) noexcept;
5
6 explicit Sampler(uint64_t width, uint64_t height) noexcept;
7
8 Sampler(const Sampler &sampler) noexcept = delete;
9
10 Sampler(Sampler &&sampler) noexcept = delete;
11
12 virtual ~Sampler() noexcept;
13
14 Sampler &operator=(const Sampler &sampler) noexcept = delete;
15
16 Sampler &operator=(Sampler &&sampler) noexcept = delete;
```

```
17
18
19 void resetSampling() noexcept;
20
21 virtual float getSample() noexcept = 0;
```

Listing A.16: Sampler API

```
1  public:
2  explicit Camera () noexcept = delete;
3
4  explicit Camera(const ::glm::vec3 &position,
5   const ::glm::vec3 &lookAt, const ::glm::vec3 &up) noexcept;
6
7  Camera(const Camera &camera) noexcept = delete;
8
9  Camera(Camera &&camera) noexcept = delete;
10
11 virtual ~Camera() noexcept;
12
13 Camera &operator=(const Camera &camera) noexcept = delete;
14
15 Camera &operator=(Camera &&camera) noexcept = delete;
16
17
18 virtual Ray generateRay(float u, float v,
19  float deviationU,
20  float deviationV) const noexcept = 0;
21
22 virtual AABB getBoundingBox() const noexcept;
23
24 float getBlock(uint32_t sample) noexcept;
25
26 void resetSampling() noexcept;
27
28 void resetSampling() noexcept;
```

Listing A.17: Camera API

```
1  public:
2  explicit Light () noexcept = delete;
3
4  explicit Light(Material radiance) noexcept;
```

```
5
6  Light(const Light &light) noexcept = delete;
7
8  Light(Light &&light) noexcept = delete;
9
10 virtual ~Light() noexcept;
11
12 Light &operator=(const Light &light) noexcept = delete;
13
14 Light &operator=(Light &&light) noexcept = delete;
15
16
17 virtual ::glm::vec3 getPosition() noexcept = 0;
18
19 float getArea() const noexcept;
20
21 virtual ::glm::vec3 getNormal (const ::glm::vec3 &point) const noexcept = 0;
22
23 virtual void resetSampling() noexcept = 0;
24
25 virtual bool intersect(Intersection *intersection, const Ray &ray) const noexcept = 0;
```

Listing A.18: Light API

```
1  public:
2  explicit ObjectLoader() noexcept = default;
3
4  ObjectLoader(const ObjectLoader &objectLoader) noexcept = delete;
5
6  ObjectLoader(ObjectLoader &&objectLoader) noexcept = delete;
7
8  virtual ~ObjectLoader() noexcept;
9
10 ObjectLoader &operator=(const ObjectLoader &objectLoader) noexcept = delete;
11
12 ObjectLoader &operator=(ObjectLoader &&objectLoader) noexcept = delete;
13
14
15 virtual int32_t process() noexcept = 0;
16
17 bool isProcessed() const noexcept;
18
19 virtual bool fillScene(Scene *scene,
```

```
20 ::std::function<::std::unique_ptr<Sampler> ()> samplerLambda) noexcept = 0;
```

Listing A.19: ObjectLoader API

```
1  public:
2  explicit Texture () noexcept = default;
3
4  explicit Texture(::std::vector<char> &&data,
5  uint32_t width, uint32_t height, uint32_t bytesPerPixel,
6  TextureType textureType) noexcept;
7
8  Texture(const Texture &texture) = default;
9
10 Texture(Texture &&texture) noexcept = default;
11
12 ~Texture() noexcept = default;
13
14 Texture &operator=(const Texture &texture) noexcept = delete;
15
16 Texture &operator=(Texture &&texture) noexcept = default;
17
18
19 bool operator==(const Texture &texture) const noexcept;
20
21 bool isValid () const noexcept;
22
23 ::glm::vec3 getColor(const ::glm::vec2 &texCoords) const noexcept;
24
25 TextureType getTextureType() const noexcept;
```

Listing A.20: Texture API

B

LOADING A SCENE

```
// Setup light sampler
auto samplerLight = []() { return std::make_unique<Components::MersenneTwister>(); };

// Setup scene
char *pathOBJ = "path to file .obj";
char *pathMTL = "path to file .mat";

Components::OBJLoader objLoader = Components::OBJLoader (pathOBJ, pathMTL);
  // Read OBJ & MTL files into objLoader
objLoader.process();
  // Check if no error
if (!objLoader.isProcessed()) {
 return -1;
}

MobileRT::Scene scene;
  // Add triangles from objLoader to the scene
bool sceneBuilt = objLoader.fillScene(&scene, samplerLight);
if (!sceneBuilt) {
 return -1;
}

  // Add a sphere to the scene
glm::vec3 spherePos {0.45, -0.65, 0.4};
float sphereRadius = 0.35;
int sphereIndexMaterial = scene.materials_.size();
scene.spheres_.emplace_back(MobileRT::Sphere {spherePos, sphereRadius,sphereIndexMaterial});
glm::vec3 matDiffuseColor {0.1, 0.1, 0.1};
glm::vec3 matSpecularColor {0.9, 0.9, 0.9};
MobileRT::Material mirrorMat {matDiffuseColor, matSpecularColor};
scene.materials_.emplace_back(mirrorMat);


```

```
// Setup camera
int width = 1024;
int height = 1024;
  // Fix aspect ratio
float ratio = std::max(width / height, height / width);
float hfovFactor = width > height ? ratio : 1;
float vfovFactor = width < height ? ratio : 1;

std::unique_ptr<MobileRT::Camera> camera = std::make_unique<Components::Perspective>(
glm::vec3{0.0f, 0.0f, -3.4f},// position
glm::vec3{0.0f, 0.0f, 1.0f},// look at
glm::vec3 {0.0f, 1.0f, 0.0f},// up
45.0f * hfovFactor, 45.0f * vfovFactor);// fov

// Setup pixel sampler
  // Mersenne Twister
std::unique_ptr<MobileRT::Sampler> samplerPixel =
  std::make_unique<Components::MersenneTwister>();

// Setup shader
int spl = 8;
  // Whitted
std::unique_ptr<MobileRT::Shader> shader = std::make_unique<Components::Whitted>(
  std::move(scene), spl, MobileRT::Shader::BVH);
  // Path Tracing
std::unique_ptr<MobileRT::Sampler> samplerRussianRoulette;
samplerRussianRoulette = std::make_unique<Components::MersenneTwister>();
std::unique_ptr<MobileRT::Shader> shader = std::make_unique<Components::PathTracer>(
  std::move(scene), std::move(samplerRussianRoulette), spl, MobileRT::Shader::REGULAR_GRID);

// Setup tone mapper
std::function<const glm::vec3(const glm::vec3 &,
uint32_t)> lambdaToneMapper = [](const glm::vec3 &color, uint32_t numSamples) {
 float gamma = 2.2f;
 return MobileRT::toneMap(color, numSamples, gamma); };

// Setup renderer
int spp = 1;
MobileRT::Renderer renderer = MobileRT::Renderer (
  std::move(shader), std::move(camera), std::move(samplerPixel),
  width, height, spp, lambdaToneMapper);


```

```
77 // Start rendering
78 int nThreads = 4;
79 int strideInBytes = width * sizeof(int);
80 int *bitmap = new int[width * height];
81 renderer.renderFrame(bitmap, nThreads, strideInBytes);
```

Listing B.1: How to load a 3D scene.

C

EXECUTION TIMES

C.1 WHITTED

C.1.1 *Samsung*

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	52974.08	18.85	6.96	X
2	X	18.82	7.41	X
3	X	20.23	6.89	X
4	X	17.77	6.97	X
5	X	21.45	6.98	X
6	X	18.82	6.95	X
Median (s)	52974.08	18.835	6.965	X

Table 13.: Execution times of scene Conference with 1 thread.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	8628.48	26.74	14.88	X
2	X	27.75	23.48	X
3	X	27.68	15.45	X
4	X	26.82	15.78	X
5	X	26.94	15.46	X
6	X	26.7	15.77	X
Median (s)	8628.48	26.88	15.615	X

Table 14.: Execution times of scene Porsche with 1 thread.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	358.58	12.88	7.29	104.42
2	360.5	12.65	7.22	95.49
3	354.71	12.68	7.34	97.27
4	333.04	12.47	7.29	101.01
5	386.83	12.48	8.68	96.84
6	428.77	13.34	7.8	105.28
Median (s)	359.54	12.665	7.315	99.14

Table 15.: Execution times of scene Cornell Box with 1 thread.

C.1.2 *Nokia 3.1*

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	51545.6	17.93	6.03	7900.99
2	X	17.68	6.11	7643
3	X	17.67	6.06	X
4	X	17.69	6.08	X
5	X	17.68	6.02	X
6	X	17.68	6.06	X
Median (s)	51545.6	17.68	6.06	7771.995

Table 16.: Execution times of scene Conference with 1 thread.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	19649.28	9.68	3.05	3847.26
2	X	9.68	3.07	X
3	X	9.68	3.07	X
4	X	9.67	3.07	X
5	X	9.68	3.08	X
6	X	9.68	3.09	X
Median (s)	19649.28	9.68	3.07	3847.26

Table 17.: Execution times of scene Conference with 2 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	14078.08	5.43	1.68	2670.32
2	X	5.45	1.58	X
3	X	5.44	1.62	X
4	X	5.44	1.58	X
5	X	5.44	1.59	X
6	X	5.42	1.59	X
Median (s)	14078.08	5.44	1.59	2670.32

Table 18.: Execution times of scene Conference with 4 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	9679.36	3.71	1.15	2159.24
2	X	3.71	1.08	X
3	X	3.43	1.11	X
4	X	3.68	1.13	X
5	X	3.71	1.11	X
6	X	3.44	1.07	X
Median (s)	9679.36	3.695	1.11	2159.24

Table 19.: Execution times of scene Conference with 8 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	3870.6	27.89	11.54	1553.39
2	X	28.24	10.86	X
3	X	27.60	11.86	X
4	X	27.86	11.85	X
5	X	28.36	11.59	X
6	X	27.62	11.12	X
Median (s)	3870.6	27.875	11.565	1553.39

Table 20.: Execution times of scene Porsche with 1 thread.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	2016.11	14.57	5.8	783.89
2	X	14.87	5.76	X
3	X	14.44	5.76	X
4	X	14.61	5.76	X
5	X	14.60	5.85	X
6	X	14.60	5.81	X
Median (s)	2016.11	14.6	5.78	783.89

Table 21.: Execution times of scene Porsche with 2 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	1032.35	7.68	2.96	449.92
2	X	7.66	2.97	X
3	X	7.71	3.05	X
4	X	7.67	2.91	X
5	X	7.76	2.93	X
6	X	7.83	2.94	X
Median (s)	1032.35	7.695	2.95	449.92

Table 22.: Execution times of scene Porsche with 4 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	886.4	4.93	1.93	389
2	X	4.88	1.91	X
3	X	4.83	1.89	X
4	X	4.8	1.87	X
5	X	4.61	1.92	X
6	X	4.87	1.94	X
Median (s)	886.4	4.85	1.92	389

Table 23.: Execution times of scene Porsche with 8 threads.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	263.59	11.73	5.26	70.9
2	264.10	11.85	5.22	70.63
3	263.10	11.84	5.19	71.38
4	264.84	11.66	5.3	70.63
5	263.10	11.84	5.26	70.83
6	263.34	11.77	5.19	70.39
Median (s)	263.465	11.805	5.24	70.73

Table 24.: Execution times of scene Cornell Box with 1 thread.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	130.10	6.03	2.65	35.31
2	130.11	6.07	2.66	35.89
3	130.11	6.09	2.61	35.13
4	130.34	6.09	2.65	35.13
5	130.34	6.09	2.66	35.89
6	130.35	6.01	2.65	35.88
Median (s)	130.23	6.08	2.65	35.60

Table 25.: Execution times of scene Cornell Box with 2 threads.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	65.84	3.15	1.31	18.5
2	65.59	3.13	1.30	18.38
3	65.84	3.11	1.30	18.39
4	66.10	3.12	1.33	18.14
5	65.84	3.1	1.31	18.39
6	66.09	3.21	1.30	18.40
Median (s)	65.84	3.125	1.305	18.39

Table 26.: Execution times of scene Cornell Box with 4 threads.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	43.09	1.97	0.96	12.28
2	43.60	2.13	0.92	13.31
3	42.35	1.98	0.95	12.14
4	42.35	2.09	0.92	12.39
5	42.35	2.01	0.93	12.11
6	42.86	2.09	0.96	13.14
Median (s)	42.605	2.05	0.94	12.335

Table 27.: Execution times of scene Cornell Box with 8 threads.

C.1.3 *Nokia 7.1*

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	25978.88	21.77	8.45	10972.31
2	X	21.77	8.44	X
3	X	21.78	8.49	X
4	X	21.85	8.52	X
5	X	21.88	8.48	X
6	X	21.8	8.48	X
Median (s)	25978.88	21.79	8.48	10972.31

Table 28.: Execution times of scene Conference with 1 thread.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	17420.8	11.1	4.28	7569.35
2	X	11.1	4.28	X
3	X	11.15	4.29	X
4	X	11.14	4.26	X
5	X	11.11	4.27	X
6	X	11.2	4.24	X
Median (s)	17420.8	11.125	4.275	7569.35

Table 29.: Execution times of scene Conference with 2 threads.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	11088.896	6.19	2.18	3037.36
2	X	6.14	2.17	X
3	X	6.15	2.16	X
4	X	6.19	2.16	X
5	X	6.14	2.16	X
6	X	6.12	2.15	X
Median (s)	11088.896	6.145	2.16	3037.36

Table 30.: Execution times of scene Conference with 4 threads.

#Measurement \ Accelerator	Naive	Regular Grid	BVH	KD-Tree
1	15781.54	4.27	1.51	3336.34
2	X	4.36	1.53	X
3	X	4.31	1.56	X
4	X	4.26	1.54	X
5	X	4.3	1.53	X
6	X	4.28	1.55	X
Median (s)	15781.54	4.29	1.535	3336.34

Table 31.: Execution times of scene Conference with 8 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	4230.82	32.86	15.05	1752.09
2	X	33.05	15	X
3	X	32.86	15.09	X
4	X	33.01	15.09	X
5	X	33.07	15.05	X
6	X	32.73	15.04	X
Median (s)	4230.82	32.935	15.05	1752.09

Table 32.: Execution times of scene Porsche with 1 thread.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	2053.43	16.19	7.58	889.89
2	X	16.15	7.59	X
3	X	16.19	7.73	X
4	X	16.24	7.54	X
5	X	16.48	7.61	X
6	X	16.19	7.76	X
Median (s)	2053.43	16.19	7.6	889.89

Table 33.: Execution times of scene Porsche with 2 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	1334.12	8.37	3.84	498.03
2	X	8.44	3.84	X
3	X	8.43	3.82	X
4	X	8.42	3.84	X
5	X	8.37	4.01	X
6	X	8.3	3.91	X
Median (s)	1334.12	8.395	3.84	498.03

Table 34.: Execution times of scene Porsche with 4 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	820.73	5.77	2.59	411.79
2	X	5.67	2.78	420.12
3	X	5.67	2.59	422.28
4	X	6.01	2.55	427.2
5	X	5.78	2.71	X
6	X	5.76	2.58	X
Median (s)	820.73	5.765	2.59	421.2

Table 35.: Execution times of scene Porsche with 8 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	312.25	12.62	6.32	85.62
2	312.08	12.56	6.35	85.72
3	311.98	12.63	6.34	85.72
4	311.79	12.53	6.36	85.75
5	311.86	12.56	6.37	85.92
6	311.99	12.51	6.35	85.56
Median (s)	311.985	12.56	6.35	85.72

Table 36.: Execution times of scene Cornell Box with 1 thread.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	158.4	6.26	3.23	43.86
2	158.4	6.21	3.28	44.66
3	159.29	6.26	3.23	44.18
4	158.54	6.4	3.23	43.39
5	158.39	6.26	3.19	43.9
6	159.31	6.29	3.22	43.79
Median (s)	158.47	6.26	3.23	43.88

Table 37.: Execution times of scene Cornell Box with 2 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	85.15	3.41	1.7	24.13
2	80.96	3.38	1.66	23.19
3	83.95	3.26	1.73	22.85
4	84.98	3.21	1.66	24.02
5	81.74	3.44	1.77	24.14
6	81.78	3.41	1.72	22.22
Median (s)	82.865	3.395	1.71	23.605

Table 38.: Execution times of scene Cornell Box with 4 threads.

Accelerator / #Measurement	Naive	Regular Grid	BVH	KD-Tree
1	57.64	2.22	1.26	15.53
2	58.9	2.32	1.17	16.92
3	58.57	2.46	1.23	16.29
4	55.35	2.41	1.18	16.69
5	55.35	2.21	1.25	15.33
6	59.82	2.24	1.31	17.06
Median (s)	58.105	2.28	1.24	16.49

Table 39.: Execution times of scene Cornell Box with 8 threads.

C.2 PATH TRACING

Device / #Measurement	W230SS (MobileRT)	Nokia 3.1	Nokia 7.1	W230SS (PBRT)
1	91.0445	327.61	189.04	186.04
2	91.735	326.77	189.18	187.01
3	90.8876	326.48	189.48	191.01
4	90.8707	329.12	189.15	187.02
5	90.7004	325.95	188.16	187.03
6	90.5611	332.18	190.15	187.02
Median (s)	90.87915	327.19	189.165	187.02

Table 40.: Execution times of scene Dragon in all devices, with 8 threads at 896x896.