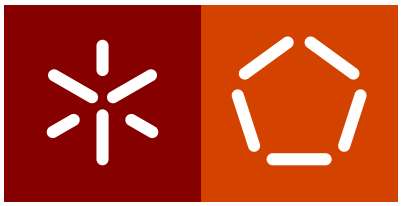
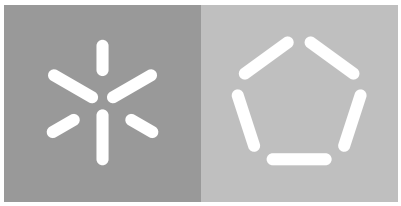


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Manuel da Silva Santos

Mobile Ray Tracing

July 2018



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

July 2018

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 and replace and / or complement 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

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 libraries that provide functionalities to render images based in different techniques. Ray tracing is a technique for rendering an image by tracing the path of light through pixels in an image plane and simulating the effects of its interactions with virtual objects. 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, the need for more libraries to help the programmers develop applications for these systems is increasing too. 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 assess different integrators (numerical solutions to the rendering equation) and / or assess different approaches in ray tracing like Packet Traversal. It is also not focused on assess different quasi random numbers generators neither assess the performance of different computer systems.

These rendering algorithms based in ray tracing are very useful because they allow rendering photo-realistic images with a degree of realism much better than the graphics cards allow by default through rasterization. As previously stated, the computing power in mobile devices processors has been increasing and this may allow a mobile device with a mid-range processor to 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 have access to. But, there are almost no rendering libraries based in ray tracing available today for mobile systems like Android, iOS or Windows 10 Mobile. And 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.

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, will be developed a library that supports 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 DOCUMENT STRUCTURE

This dissertation is organized in five chapters: Introduction, State of the Art, Software Architecture, Demonstration: Global Illumination and Conclusion & Future work.

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 have. 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 and in the rendering components. This chapter ends with an explanation of some Android specifics, such as the user interface by characterizing its work flow and mentioning some of the challenges overcame during the development of the application.

The fourth chapter summarizes the key results obtained by executing different algorithms with different number of threads and different acceleration structures. And it will also contain a small comparison between the developed application and the Android CPU Raytracer (Dahlquist).

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

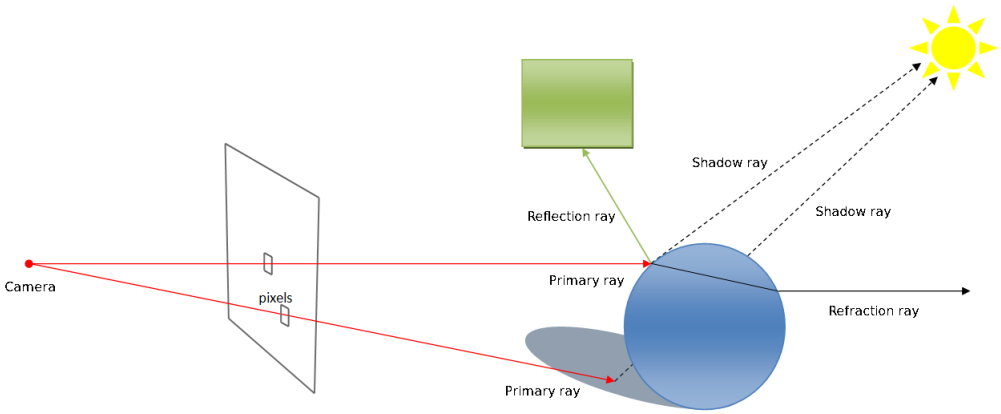
2

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 luminance 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 with 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 2.: Illustration of a typical ray tracer algorithm.



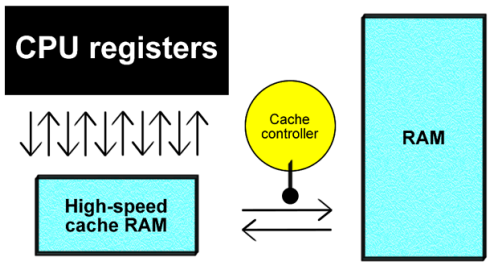
A typical image of 1024 x 1024 pixels tends to cast at least a million primary rays and a multiple of that as shadow, reflection, refraction and secondary rays. This is why ray tracing can't provide interactive frame rates with ease.

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 features developed inside the processor like the cache, the multilevel pipeline, the hardware prefetching and 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. The downside is that it is very small because its cost are much higher than a typical RAM memory. Nowadays, 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 memory RAM.

Figure 3.: Illustration of a typical cache memory inside a microprocessor (Karbo and Aps.).



The multilevel pipeline is another feature inside a processor which allows to reduce the processor's own latency by allowing a form of parallelism called instruction-level parallelism within a single processor. Basically, one instruction is divided in some stages and it allows the possibility to execute different stages of different instructions simultaneously. In the early days, the Classic RISC pipeline were typically divided in 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 value in the register

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.

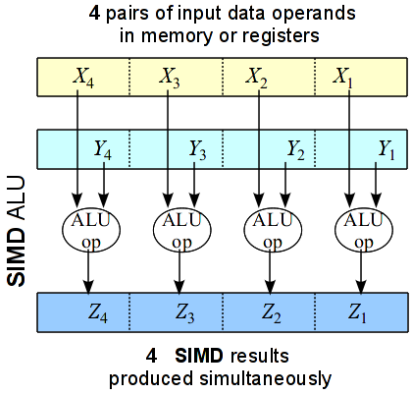
Figure 4.: Illustration of the Classic RISC multilevel CPU pipeline (Dictionaries and Encyclopedias).

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

The hardware prefetching is, as the name implies, a feature in the processor that makes the processor fetch the data and the instructions before it really needs to execute. This allows to reduce the time that the processor waits for the data in the main memory.

Finally, SIMD instruction set is a set of special instructions that allows the processor to read or write bigger sizes of data. 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 to read or write up to 512 bits by using special registers and additional ALUs provided in the processor.

Figure 5.: 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.

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 where can be executed, interactivity, progressive and the type of rendering components.

2.3.1 *Type of software license*

A software application can have different type of software license, but in order to simplify this dissertation, the software license were grouped into 3 types: Free, Commercial and Open Source. The license Free means that the user has the right to execute freely the application but cannot copy, modify or distribute the implemented code. The license Commercial means that the user cannot even execute the application without buying it first and cannot copy, modify or distribute the implemented code. The license Open Source means that the user has the right to execute freely the application and can even copy, modify and distribute the implemented code. The software license is very important in this work because it lets the user know if he can develop something over the provided 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 also or may not be executed on another platform, like a mobile device. This information is very important in this work because it inform us whether a ray tracer can be executed in a mobile device with the typical Operating System like Android, iOS or Windows 10 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 Accelerator 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 in 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 1 shows some applications or frameworks that use ray tracing available today and compares them according to their type of license, platform compatibility, interactivity, 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 1.: Comparison of different applications/frameworks that use ray tracing

Product	License	Platform	Mobile	Interactivity	Progressive	Programmable Components
Optix (Nvidia (a))	F	Nvidia GPU	N	Y	Y	Y
Optix Prime (Nvidia (b))	F	CPU & Nvidia GPU	N	Y	Y	Y
RenderMan RIS (Pixar)	C	CPU	N	Y	Y	Y
OctaneRender (Inc)	C	GPU	N	Y	Y	N
Embree (Intel	O	Intel CPU	N	Y	Y	Y
Radeon Rays (AMD)	O	CPU & GPU	N	Y	Y	Y
PBRT (Matt Pharr)	O	CPU	N	N	N	N
Visionaray (Zellmann)	O	CPU & Nvidia GPU	N	Y	Y	N
YafaRay (Gustavo Pichorim Boiko)	O	CPU	N	N	N	N
tray_rust (Usher (c))	O	CPU	N	N	N	N
micro-packet (Usher (a))	O	Intel CPU	N	N	N	N
tray (Usher (b))	O	CPU	N	N	N	N
The G3D Innovation Engine (Morgan)	O	CPU	N	N	N	N
HRay (Kenneth)	O	CPU	N	N	N	N
Mitsuba (Jakob (2010))	O	CPU	N	N	N	N
Indigo RT (Limited)	O	CPU & GPU	N	Y	N	N
jsRayTracer (Chedeau)	O	CPU	N	Y	Y	N
Android CPU Raytracer (Dahlquist)	O	CPU	Y (Android)	Y	N	N

2.4.1 *Conclusions*

As the table 1 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 even allows interactions with the objects 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 their own rendering components like the sampler, integrator, camera and shapes of virtual objects. It also studies the drawbacks that these mobile devices may have comparing with the average multi-core personal computer hardware. And 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

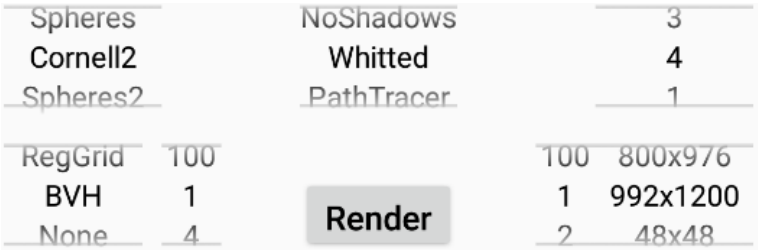
SOFTWARE ARCHITECTURE

3.1 APPROACH

The development of this demonstration application involves the distinction of three layers of abstraction: user interface, rendering components and the library itself.

The top layer is the User Interface which is obviously application, and eventually device, dependent. The user interface of this dissertation's demo application is very simple and just allows the user to see the rendered image and choose some rendering components to use, like the integrator, the sampler, the number of threads and samples and choose the scene to render. Although this layer is useful, this dissertation focuses only on the middle and bottom layers.

Figure 6.: Illustration of the developed User Interface.

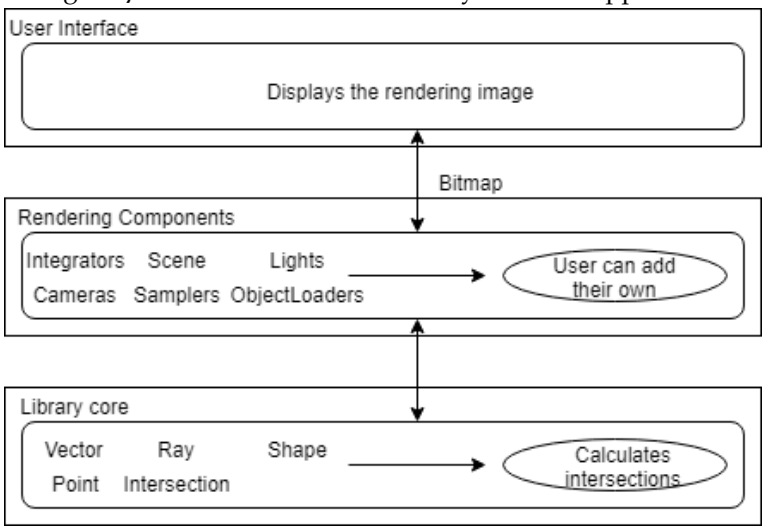


The middle layer provides the rendering components which are abstracted concepts about rendering that use functionalities that the library offers to the programmer. Some of these rendering components are the camera, the light, the sampler and the integrator. These rendering components are useful for 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.

Lastly, 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 basic functionalities of a ray tracer engine like: create vectors and points, create different primitives with different materials, cast rays and intersect rays with the primitives.

It is also important to mention that the demo application was developed in order to show the developed features, the performance achieved in the mobile devices and also to help promote the library. This demo application is a good way to test it in several Android devices like a smart phone or a tablet.

Figure 7.: 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.

The first decision was: the primary rays always have origin in the camera. This decision was made in order to not mix the code of the integrator with the ray tracer renderer engine.

Other decision made was to make the rendering process progressive, which means that the rendered image is incrementally refined with more and more traced rays. As the integrator will be converging to better values. This is important in order to give the user a fast rendered image, with some noise or aliasing, and converge it progressively to a better solution with higher details and practically without any visual noise or aliasing.

Another thought aspect that was studied is the permission of dynamic scenes and / or dynamic cameras, which means to let the programmer modify the camera or the scene while the ray tracer is rendering it. This makes possible to build challenging applications and also provide more interesting scenes and more eye candy applications for the final user.

Last, but not least, is that the code was developed in a modular way, in this case was programmed in an object oriented manner. This allows the programmers to code their own rendering components, like the integrator, camera, sampler and light, without having to develop the basic features in the renderer engine.

3.2 METHODOLOGY

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. The User Interface 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 user interface. Despite that, its performance is not very important because it doesn't interfere significantly with the others layers of the application.

3.3 LIBRARY

As stated above, this library was implemented in an object-oriented fashion. The most important classes that provide functionalities already developed for the user 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 information in vectors and provides methods to trace rays in the scene without any acceleration structures.
- 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
- Primitive: class that stores the shape and material of each primitive in the scene.
- Ray: class that represents a ray casted into the scene.

3.3.1 *Third parties dependencies*

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

- OpenGL Mathematics (Creation): used to create 3d points and vectors, perform geometry calculations and store pixels and primitives colors.
- tinyobjloader (syoyo): used to load scenes from wavefront obj files into the main memory.
- Google Test (Google (b)): used to create some unit tests and to mock some classes.

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

Besides being dependent on these 3 libraries, this library also depends on OpenGL ES 2.0 (Inc.) from Android SDK and depends on CMake (kitware) in order to compile the application.

3.3.2 *Renderer*

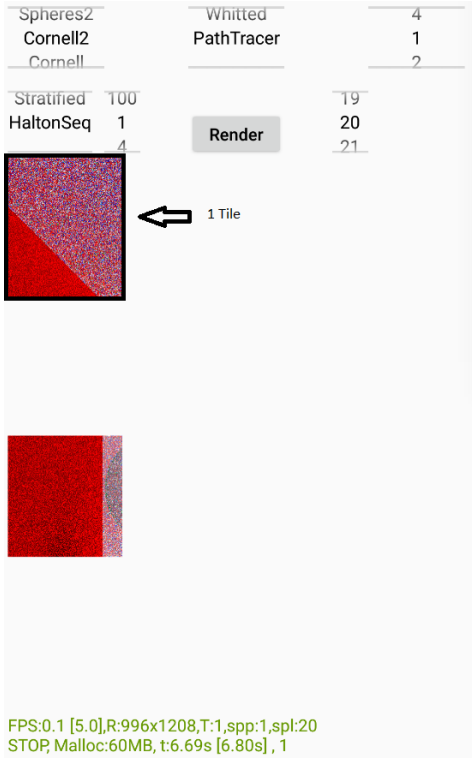
The Renderer is the closest class to the application that starts the rendering process. This class provides two main methods:

```
1 void renderFrame(::std::uint32_t *bitmap, ::std::int32_t numThreads, ::std::uint32_t stride) noexcept;
2 void stopRender() 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 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 array. The image plane is divided into 16 tiles of pixels and it is traced one primary ray per pixel in each tile.

Figure 8.: Illustration of tiling the image plane.



The *stopRender* method only serves to stop the rendering process without cleaning the pixels' colors already calculated.

3.3.3 *Scene*

The Scene is the class that handles the process of intersecting a ray with the primitives and source lights in the scene. Besides providing a vector to add the light sources and a vector to add primitives to the scene, it also provides two main methods:

```
1 Intersection trace(Intersection intersection, const Ray &ray) noexcept;
2 Intersection shadowTrace(Intersection intersection, const Ray &ray) noexcept;
```

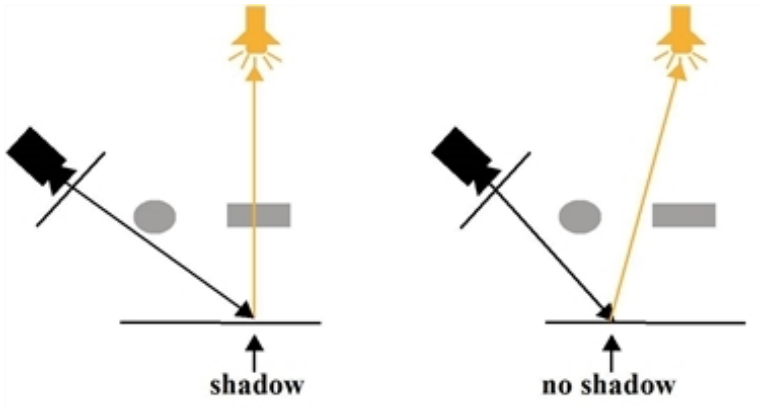
Listing 3.2: Main methods in Scene

The *trace*, as the name implies, is a method that tries to intersect a ray with all the primitives and light sources in the scene. And It returns the closest intersection to the origin of that ray. This method is used to determine the intersection of the casted ray with the primitives in the scene.

The *shadowTrace*, is similar to the *trace* method but with the difference that returns the first intersection found. The purpose of this method is to simulate the shadows in the scene,

like it is illustrated in figure 9. The shadow ray is represented by the yellow line pointed to the light source.

Figure 9.: Illustration of shadow rays casting (Foundry).



3.3.4 *Shapes*

In order to make possible to generate scenes with all kind of objects, it was developed three types of shapes: plane, sphere and triangle. All shapes provide one important method:

```
1 Intersection intersect(const Intersection &intersection, const Ray &ray) const noexcept;
```

Listing 3.3: Main methods in Shape

This method determines if a ray intersects the shape and returns the intersection. It is important to mention that, obviously, each shape was developed with different algorithm.

Plane

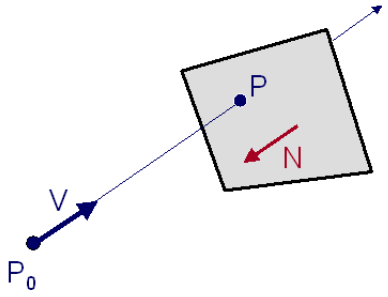
The plane is an essential primitive shape because it allows the user to build indoor scenes.

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

```
1 projection = planeNormal . rayDirection
2 if (|projection| <= 0) return false
3 distance = planeNormal . (planePoint − rayOrigin) / projection
4 if (distance <= 0 || distance > rayMaxDistance) return false
5 intersectionPoint = rayOrigin + rayDirection ∗ distance
6 return Intersection(intersectionPoint, planeNormal, material)
```

Listing 3.4: Algorithm of Ray plane intersection

Figure 10.: Illustration of a ray intersecting a plane (Vink).



Sphere

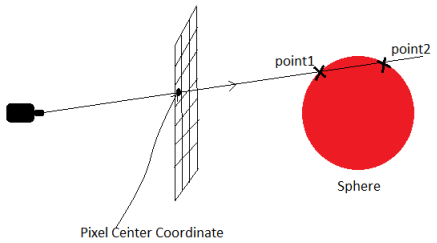
The sphere is also an important primitive shape because it allows the user to build some common objects with a shape of a ball.

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. So, the intersect method implemented has the following algorithm:

```
centerToOrigin = rayOrigin − sphereCenter
B = 2 ∗ centerToOrigin . rayDirection
C = centerToOrigin.magnitude − radius
disciminant = B^2 − 4*C
if (discriminant <= 0) return false
distance1 = (−B + sqrt{discriminant} ∗ 0.5)
distance2 = (−B − sqrt{discriminant} ∗ 0.5)
distance = min(distance1, distance2)
if (distance <= 0 || distance > rayMaxDistance) return false
intersectionPoint = rayOrigin + rayDirection ∗ distance
sphereNormal = intersectionPoint − sphereCenter
return Intersection(intersectionPoint, sphereNormal, material)
```

Listing 3.5: Algorithm of Ray sphere intersection

Figure 11.: Illustration of a ray intersecting a sphere in two points (Chirag).



Triangle

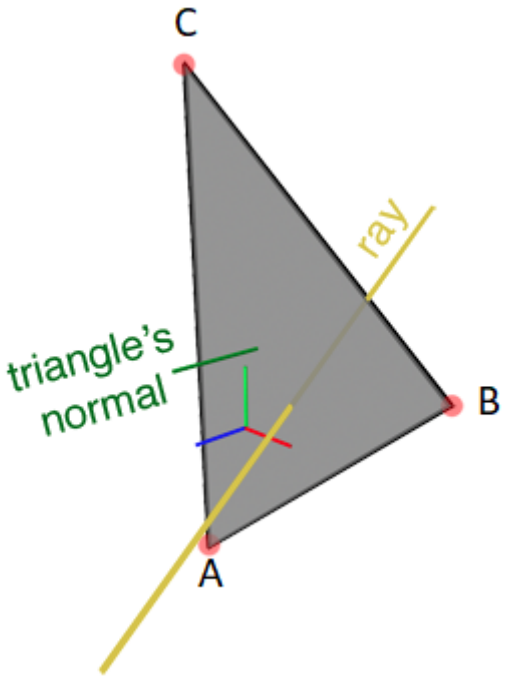
And finally, obviously the triangle has also been implemented because, as it is the simplest primitive with an area, it allows to build many different object shapes.

The construction of a triangle requires three points $[A, B, C]$, two vectors $[AB, AC]$ and the normal of the triangle. So, the intersect method implemented has the following algorithm:

```
1  perpendicularVector = rayDirection x AC
2  projection = AB . perpendicularVector
3  if(|projection| <= 0) return false
4  vectorToRay = rayOrigin − A
5  u = (vectorToRay . perpendicularVector) / projection
6  if (u < 0 || u > 1) return false
7  perpendicularVector2 = vectorToRay x AB
8  v = (rayDirection . perpendicularVector2) / projection
9  if(v < 0 || (u + v) > 1) return false
10 distance = (AC . perpendicularVector2) / projection
11 intersectionPoint = rayOrigin + rayDirection * distance
12 return Intersection(intersectionPoint, triangleNormal, material)
```

Listing 3.6: Algorithm of Ray triangle intersection

Figure 12.: Illustration of a ray intersecting a triangle (Scratchapixel).



3.3.5 *Acceleration Structures*

As previously stated, the rendering algorithms based on ray tracing can be very computationally demanding, like Path Tracing and Bidirectional Path Tracing. These algorithms are very computationally demanding because if the scene is very complex, made with millions of primitives, it is necessary to try to intersect every ray casted into the scene with those millions of primitives. This task is, obviously, very time consuming and can waste a lot of battery power on the mobile device. Luckly, there are already known techniques, called acceleration structures, that helps to accelerate this process by reducing the number of intersections.

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

- Subdivision of Space: the 3D space of the scene is divided in smaller portions of space which the volume elements can be uniform or irregular:
  - Regular Grids
  - Octrees
  - Kd-trees
- Subdivision of Objects: the 3D space of the scene is divided by aggregating the 3D primitives in groups which are next to each others:

– 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 these are not in the same direction of the ray. These structures also allow the rapid and simultaneous rejection of groups of primitives.

In contrast, structures based on Subdivision of objects only allow the rapid and simultaneous rejection of groups of primitives.

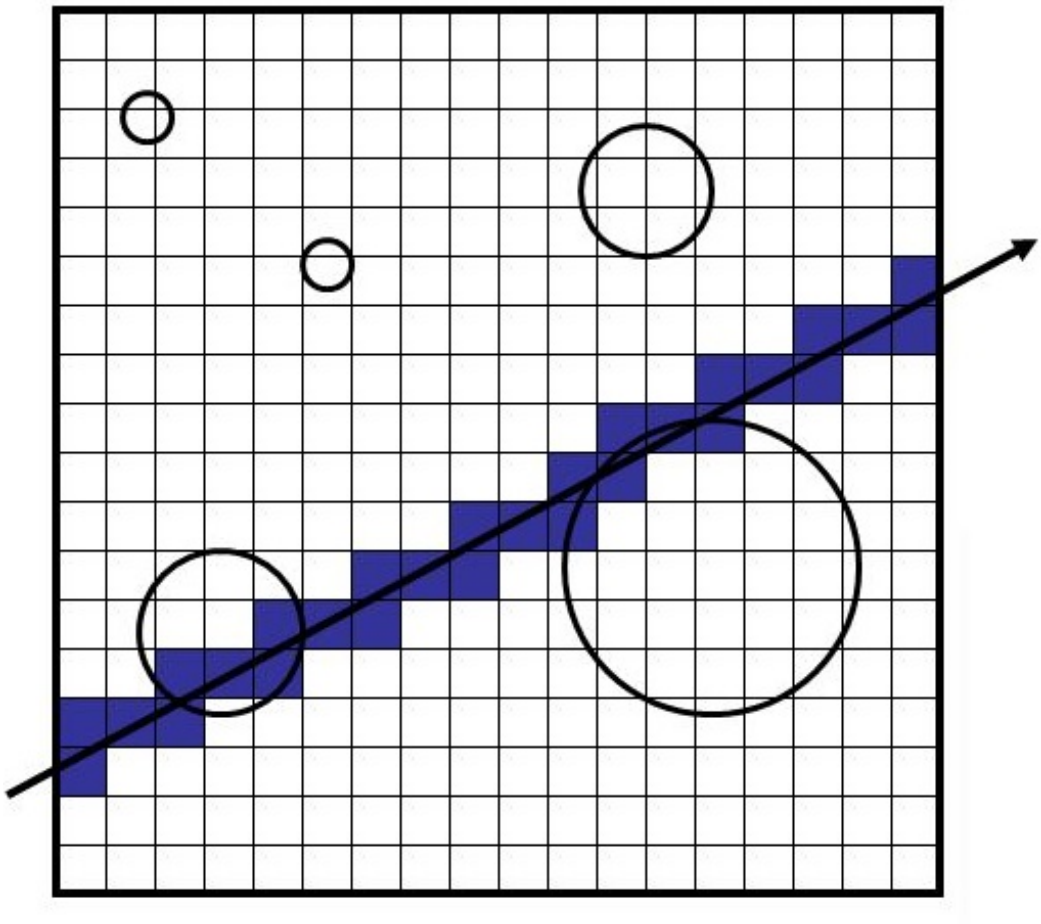
The developed library provides only two types of structures, one of each approach: the regular grid and the bounding volume hierarchy. Both structures are built with Axis Aligned Bounding Volumes (AABB), which are volume elements (voxels) that can cover, one or more primitives from the 3D scene. An AABB 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 that voxel and it only needs to store, in memory, two points of that box. Besides AABB, it is possible to build the structures with other types of voxels, like:

- Sphere: intersection with a sphere is simpler than AABB but covers more 3D space than the scene primitive needs.
- Oriented Bounding Volume: its like an AABB but rotated according to the orientation of the primitive.
- k-Discrete Oriented Prototype (k-DOP): a generalization of AABBs defined by k hyperplanes with normals in discrete directions.
- Convex Hull: is the smallest convex volume containing the object.

Regular Grid

A regular grid is an acceleration structure where the scene space is subdivided into equal voxels. Each voxel is a volume element where it could have inside one or more primitives of the scene, if a ray intersects that voxel, then it tries to intersect the primitives inside it. 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. The figure 13 shows the process of intersecting a ray with the primitives in a scene.

Figure 13.: Illustration of traversing a regular grid acceleration structure (Torsten Möller).



The developed ray tracer library provides a regular grid as an acceleration structure and its operation is as shown in the figure.

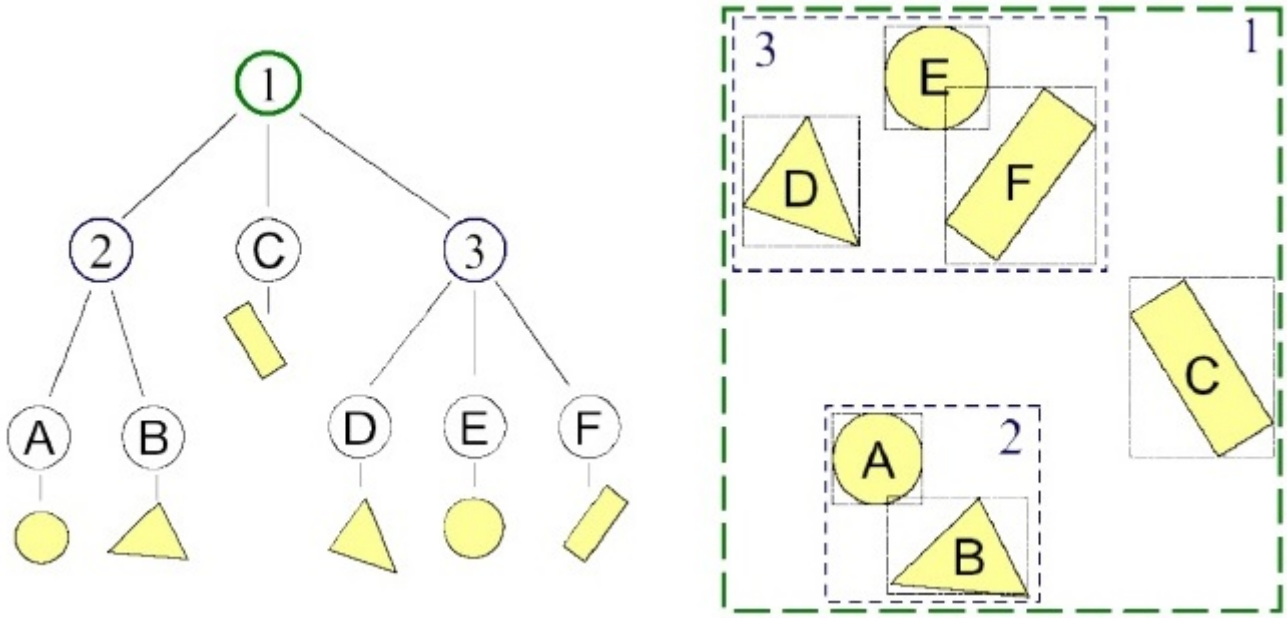
Bounding Volume Hierarchy

Bounding Volume Hierarchy is another acceleration structure provided in the developed library. This structure is different from the regular grid because the space is not divided into voxels of the same size. The space is divided in a way where the primitives of the scene are grouped according to the nearest primitives of each. This allows to build a structure of voxels where a voxel has, for sure, one or more primitives inside.

The figure 14 shows an example of a BVH structure of a 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 the scene is divided by a fake plane that divides the scene into two AABBs. The calculation of those planes

is performed with the Surface Area Heuristic (SAH) algorithm which is a very popular heuristic commonly used in ray tracers.

Figure 14.: Illustration of traversing a Bounding Volume Hierarchy acceleration structure (Kilgard).



Surface Area Heuristic

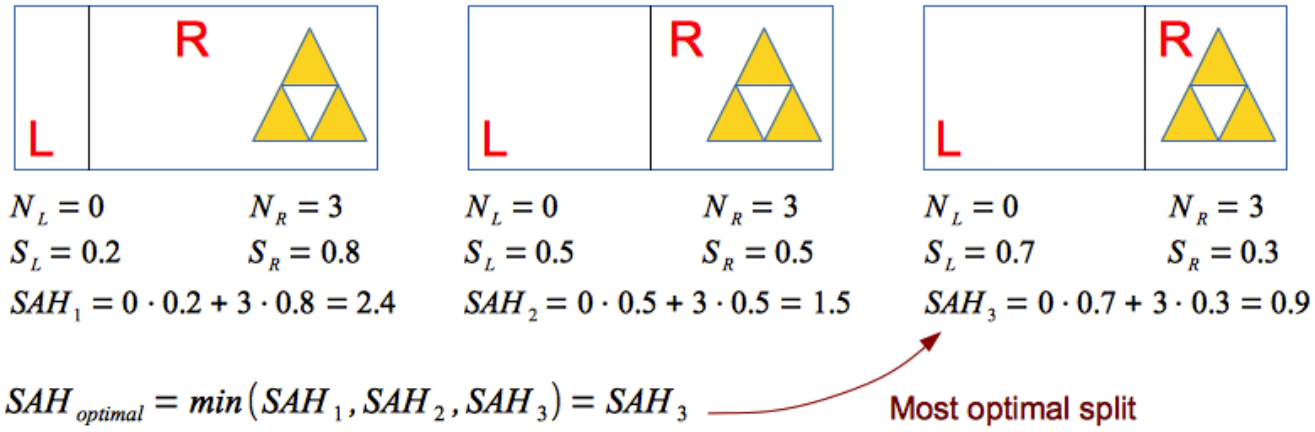
As it was said above, the library uses the Surface Area Heuristic in order to split the scene. The algorithm used in the library is very simple, it just needs to calculate the minimum area surface of the sum of left and right subtrees, as following:

$$SAH_{optimal} = min(S_L * N_L + S_R * N_R) \tag{1}$$

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

The figure 15 shows a very simple example of how the SAH algorithm works.

Figure 15.: Illustration of how Surface Area Heuristic works in Bounding Volume Hierarchy acceleration structure (Lahodiuk).



3.4 RENDERING COMPONENTS

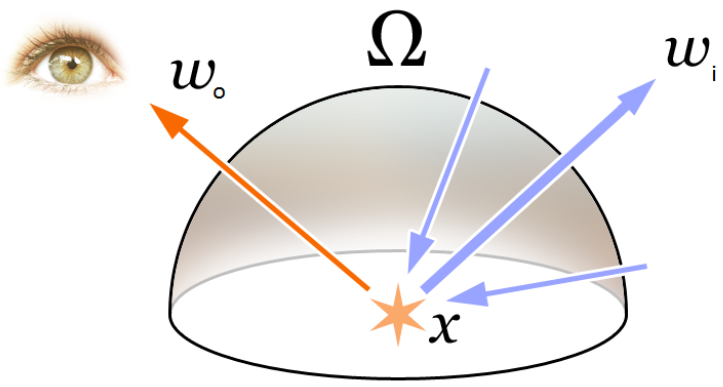
In order to show the functionalities provided by the library, it was developed a few Rendering Components. It is provided the perspective and orthographic cameras, the area and point lights, six Samplers, five Shaders and one Object loader.

3.4.1 *Shaders*

The implemented ray tracer was programmed in an objected oriented fashion, so 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.

A shader is the Rendering Component that describes how the Rendering Equation is approximated. The Rendering Equation describes how the total radiance reflected by any point p of a surface in a direction ωr is calculated. The Bidirectional Reflectance Distribution Function (BRDF) is a function that tries to approximate the Rendering Equation. In summary, a shader, in this context, is the algorithm of a BRDF.

Figure 16.: 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.



There were developed five shaders: NoShadows, Whitted, PathTracer, DepthMap and DiffuseMaterial. All shaders provide only one main method that allows the user to calculate the RGB color of a pixel with the information of a ray casted into the scene and its nearest calculated intersection.

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

Listing 3.7: Main methods of Shader

The method *shade* calculates the color of the pixel that will be stored in the parameter *rgb* and the *intersection* contains the calculated information about the intersection of a ray with the nearest primitive of the scene like the intersection point, its normal and the material of the intersected material. Finally the parameter *ray* is where the information about the ray like the origin of the ray and its direction is accessed.

The method *getCosineSampleHemisphere* calculates a random direction in World coordinates around an hemisphere with a given *normal*.

This method is used to cast new secondary rays in the scene in order to simulate the indirect light. This is typically used in Path Tracers.

DepthMap

DepthMap is a very simple shader that calculates the distance in percentage of each primary ray casted to the scene.

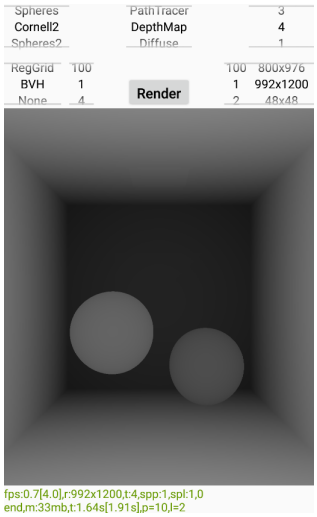
The algorithm is as much as simple as:

```
float maxDist = glm::length(maxPoint − rayOrigin) * 1.1
float depth = std::max((maxDist − intersectionLength) / maxDist, 0.0)
rgb = {depth, depth, depth}
```

Listing 3.8: Algorithm of DepthMap Shader

Where the *maxPoint* is the furthest 3D point that the user send as parameter in the DepthMap constructor.

Figure 17.: DepthMap shader.



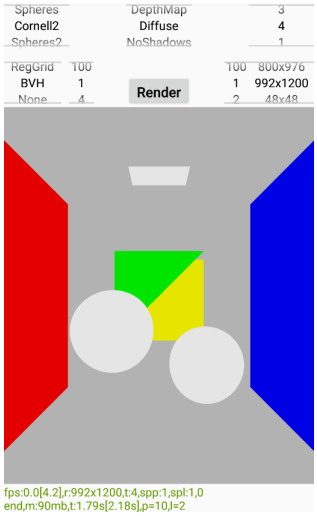
DiffuseMaterial

The DiffuseMaterial is another very simple shader that only computes directly the diffuse material color of the intersected primitive. But, in case the primitive material does not have diffuse color, the specular color is computed or the transmission color or even the light emission. As illustrated below:

```
if (kD > 0) {
  rgb = kD;
} else if (kS > 0) {
  rgb = kS;
} else if (kT > 0) {
  rgb = kT;
} else if (Le > 0) {
  rgb = Le;
}
```

Listing 3.9: Algorithm of DiffuseMaterial Shader

Figure 18.: DiffuseMaterial shader.



NoShadows

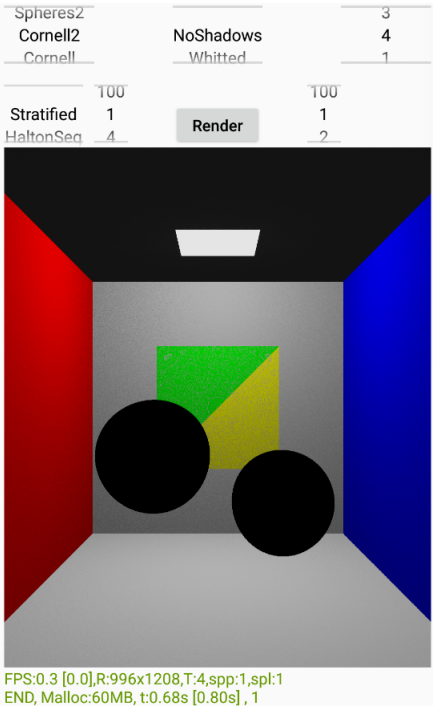
NoShadows is a simple shader because, as the name implies, it does not synthesize the shadows and it only simulates the direct lighting. This BRDF only simulates direct lighting in primitives with diffuse surfaces and it does not take into account the light coming from other points. As already said, the indirect lighting is not fully simulated but rather simplified in a way of fixed ambient light of about 10% of the color of the objects.

The algorithm developed is as following:

```
1  if (intersected material is diffuse) {
2   for each light source
3    for each sample
4     vectorToLight = lightPosition − intersectionPoint
5     cos_N_L = vectorToLight . intersectionNormal
6     if (cos_N_L > 0) rgb += kD ∗ radLight ∗ cos_N_L
7   rgb /= #samples
8   rgb /= #lights
9   rgb += kD ∗ 0.1 //Ambient light
10 }
```

Listing 3.10: Algorithm of NoShadows Shader

Figure 19.: NoShadows shader.



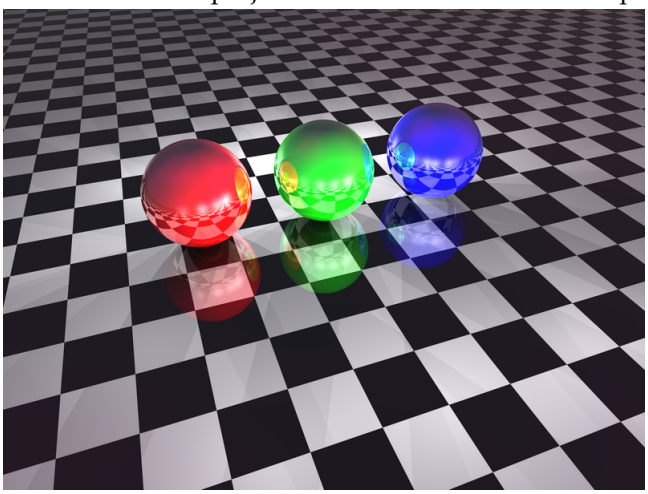
Whitted

As the name of this shader implies, Whitted is the algorithm demonstrated by Whitted in 1980s. Like the previous shader, it doesn't simulate indirect lighting. This BRDF calculates the reflected light on diffuse and specular surfaces, as well as the light refracted on surfaces with a degree of transmission, such as water. In order to simulate a reflective and refractive surfaces, the algorithm was divided into each case. This makes it possible to simulate refractive surfaces, reflective surfaces and even refractive and reflective surfaces.

The reflected light in a diffuse surface is calculated by adding the casting rays in direction to the lights. These rays are called shadow rays and their radiance are multiplied by the dot product between vector to the light and the shading normal. At the end, the summation is multiplied by the diffuse color of the intersected primitive and divided by the number of samples taken.

The reflected light in a specular surface is calculated in a different way. The specular ray is casted and ray traced, then the obtained radiance is multiplied by the specular color of the intersected primitive. The reflection direction is the subtraction between the double of dot product between the inverse of the ray direction and the shading normal multiplying by the shading normal, and the inverse of the ray direction.

Figure 20.: Reflections projected on the floor and on the spheres.



Finally, the refracted light in a transmission surface is calculated by using the refractive index of the intersected primitive and the ray direction and shading normal.

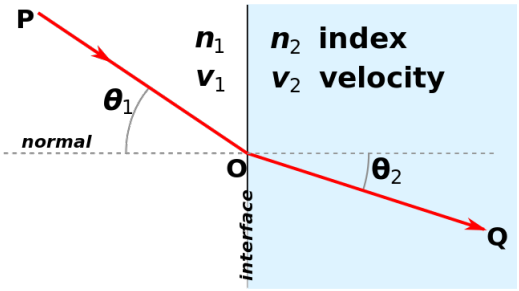
The specular refraction is calculated as follows:

```
1 float refractiveIndiceInv = 1.0 / refractiveIndice
2 glm::vec3 refractDir = glm::refract(rayDirection, shadingNormal, refractiveIndiceInv)
3 Ray transmissionRay (refractDir, intersectionPoint, rayDepth + 1, intersectionPrimitive)
4 glm::vec3 LiT_RGB
5 rayTrace(&LiT_RGB, transmissionRay)
6 glm::vec3 specularTransmissionColor = kT * LiT_RGB
```

Listing 3.11: Algorithm of Specular Transmission

And, as usual, the transmission ray is traced and its radiance is multiplied by the transmission component of the intersected primitive.

Figure 21.: Refraction of light at the interface of two media of different refractive indices.



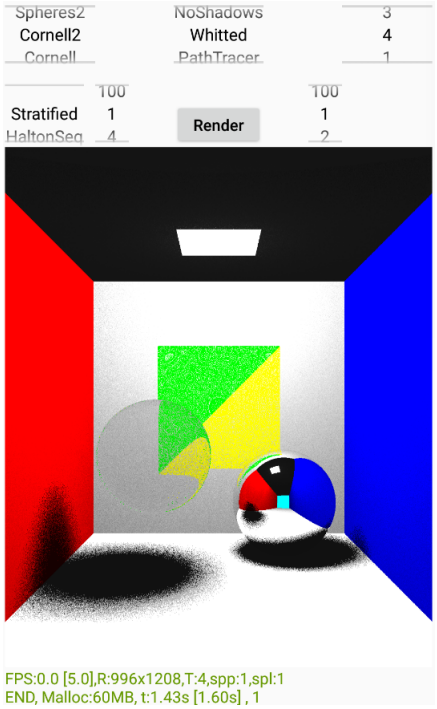
The algorithm developed is as following:

```
1 if (intersected material is diffuse) {
2  for each light source
```

```
3     for each sample
4       vectorToLight = lightPosition − intersectionPoint
5       cos_N_L = vectorToLight . shadingNormal
6       if (cos_N_L > 0) {
7         Ray shadowRay(intersectionPoint, vectorToLight, distanceToLight, rayDepth + 1)
8         if (!shadowTrace(shadowRay)) rgb += radLight * cos_N_L
9       }
10    rgb *= kD
11    rgb /= #samples
12  }
13
14  if (intersected material is specular reflective) {
15    glm::vec3 reflectionDir = glm::reflect(rayDirection, shadingNormal)
16    Ray specularRay (reflectionDir, intersectionPoint, rayDepth + 1, intersectionPrimitive)
17    glm::vec3 LiS_RGB
18    rayTrace(&LiS_RGB, specularRay)
19    rgb += kS * LiS_RGB
20  }
21
22  if (intersected material is specular refractive) {
23    float refractiveIndiceInv = 1.0 / refractiveIndice
24    glm::vec3 refractDir = glm::refract(rayDirection, shadingNormal, refractiveIndiceInv)
25    Ray transmissionRay (refractDir, intersectionPoint, rayDepth + 1, intersectionPrimitive)
26    glm::vec3 LiT_RGB
27    rayTrace(&LiT_RGB, transmissionRay)
28    rgb += kT * LiT_RGB
29  }
```

Listing 3.12: Algorithm of Whitted Shader

Figure 22.: Whitted shader.



PathTracer

Finally, the last shader developed is the canon Path Tracer. Unlike the previous shaders, this one fully simulates both direct and indirect lighting. But, like the previous shader, this algorithm simulates the light reflected on diffuse and specular surfaces, as well as the light refracted on transparent primitives.

The light reflected on diffuse surfaces is divided in two parts: direct lighting and indirect lighting. The direct lighting is calculated in a similar way to the Whitted shader but with the difference that samples are not taken from all sources of light but rather only one. The light source in each sample is chosen randomly. Then the obtained light radiance is multiplied by the Probability Density Function (PDF) in order to approximate the expected value. In this case the PDF is as simple as $1/\#lights$.

Finally, the indirect lighting reflected in diffuse surfaces is calculated in a much more complex way. First it is generated, from the intersected point, a random direction on an unit hemisphere with a PDF proportional to cosine-weighted solid angle $(PDF : p(\Theta) = cos\theta/\pi,\ x = cos(2\pi r1)\sqrt{1-r2},\ y = sin(2\pi r1)\sqrt{1-r2},\ z = \sqrt{r2})$. This kind of PDF is one of the best ways to render images with path tracing that the rendering equation converges to the expected value as fast as possible.

Then to generate a new ray direction in a world coordinates from a hemisphere it is needed to distinguish the six different possible materials:

- Diffuse reflection
- Glossy reflection
- Specular reflection
- Diffuse transmission
- Glossy transmission
- Specular transmission

The figure 24 illustrates those six possible scenarios.

To calculate the secondary ray direction in a diffuse reflective material, it was implemented a generator of random directions on unit hemisphere proportional to cosine-weighted solid angle around the normal of that hemisphere. Sampling in a cosine weighted hemisphere is better than uniform sampling over the hemisphere about the normal because the rays casted at the bottom of the hemisphere will not contribute as much as the other directions because of the multiplication by the cosine of theta.

To do this, it was implemented a method called *getCosineSampleHemisphere*, which generates a random direction around an hemisphere oriented by an arbitrary normal.

```
glm::vec3 getCosineSampleHemisphere(glm::vec3 normal);
```

Listing 3.13: Method in Shader which generates a random direction in an hemisphere.

The algorithm to generate a random direction in an hemisphere was taken from Wolfe, which is as follows:

```
float phi = glm::two_pi<float>() * uniformRandom1;// azimuthal angle
float r2 = uniformRandom2;// random distance from center
float cosTheta = std::sqrt(r2);// cos(theta) = cos(elevation angle)

glm::vec3 u = std::abs(normal.x) > 0.1 ? glm::vec3 {0.0, 1.0, 0.0} : glm::vec3 {1.0, 0.0, 0.0};
u = glm::normalize(glm::cross(u, normal));// second axis
glm::vec3 v = glm::cross(normal, u);// final axis

glm::vec3 newDirection = glm::normalize(u * (std::cos(phi) * cosTheta) + v * (std::sin(phi) * cosTheta)
+ normal * std::sqrt(1.0 - r2));
```

Listing 3.14: Method in Shader which generates a random direction in an hemisphere.

The figure 23 shows the spherical coordinate system with the respective polar angle θ (theta) and the azimuthal angle ϕ (phi).

And the indirect lighting on diffuse reflective surfaces is then the multiplication of the ray traced light radiance with the material diffuse color of the primitive and π , and divided by the probability of continuing the Russian roulette. The equations 2 shows the simplification of the reflected diffuse color calculation equation.

The resolution of diffuse reflected color of an object.

$$cos(\theta) = cos(\overrightarrow{NewDirection}, \overrightarrow{N})$$

$$\begin{aligned} PDF &= cos(\theta)/\pi \\ PDF &= cos(\overrightarrow{NewDirection}, \overrightarrow{N})/\pi \end{aligned} \tag{2}$$

$$\begin{aligned} LiD+ &= kD * RGB_{SecondaryRay} * cos(\overrightarrow{NewDirection}, \overrightarrow{N})/(PDF * ContinueProbability) \\ LiD+ &= kD * RGB_{SecondaryRay} * \pi/ContinueProbability \end{aligned}$$

Figure 23.: The spherical coordinates.

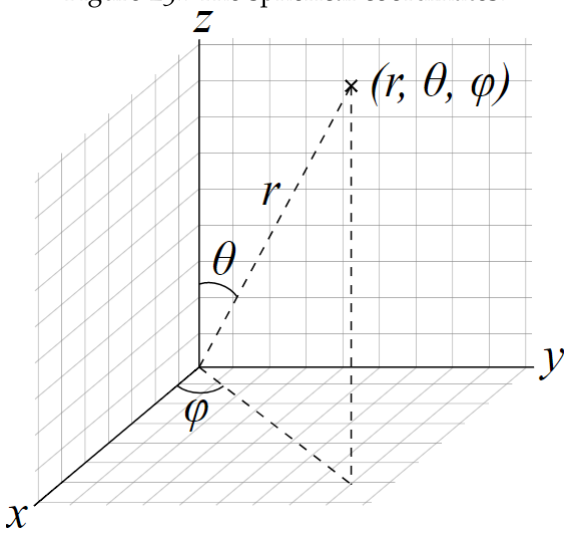
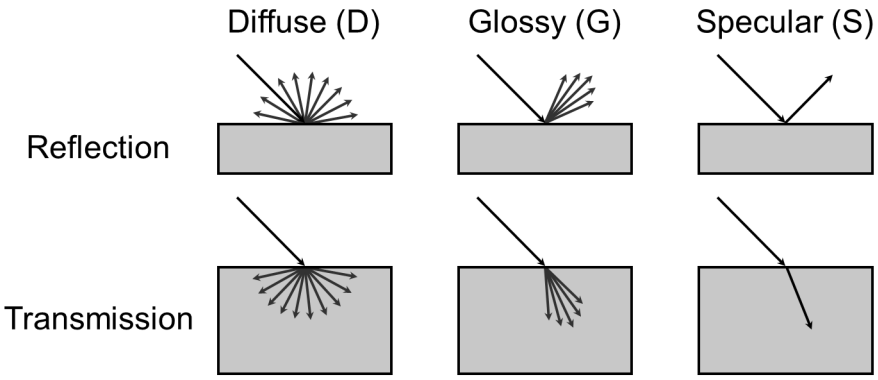


Figure 24.: The possible directions in secondary rays.



The reflected light in a specular surface and the refracted light are simulated in a similar way to the Whitted algorithm presented previously.

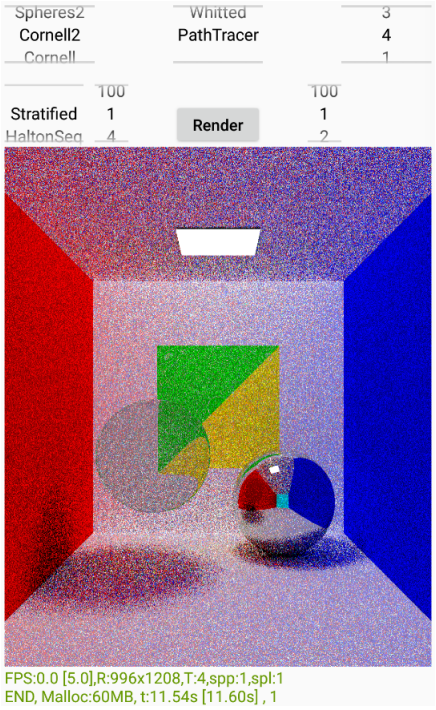
The algorithm developed in Path Tracer Shader is as following:

```
if (intersected material is diffuse) {
 for each light
  for each sample
   light = samplerLight.getSample()
   vectorToLight = lightPosition − intersectionPoint
   cos_N_L = vectorToLight . shadingNormal
   if (cos_N_L > 0) {
    Ray shadowRay(intersectionPoint, vectorToLight, distanceToLight, rayDepth + 1)
    if (!shadowTrace(shadowRay)) rgb += radLight * cos_N_L
   }
 rgb *= kD
 rgb *= #lights
 rgb /= #samples

 if (rayDepth <= RAY_DEPTH_MIN || uniform_dist(gen) > finish_probability) {
  glm::vec3 newDirection = getCosineSampleHemisphere(shadingNormal)
  Ray normalizedSecundaryRay (newDirection, intersectionPoint, rayDepth + 1,
  intersectionPrimitive)

  glm::vec3 LiD_RGB
  intersectedLight = rayTrace(&LiD_RGB, normalizedSecundaryRay)

  if (rayDepth > RAY_DEPTH_MIN) LiD /= continue_probability
  if (Ld > 0 && intersectedLight) LiD = 0

  rgb += LiD
```

```
27   }
28
29 }
30
31 if (intersected material is specular reflective) {
32   glm::vec3 reflectionDir = glm::reflect(rayDirection, shadingNormal)
33   Ray specularRay (reflectionDir, intersectionPoint, rayDepth + 1, intersectionPrimitive)
34   glm::vec3 LiS_RGB
35   rayTrace(&LiS_RGB, specularRay)
36   LiS += kS * LiS_RGB
37 }
38
39 if (intersected material is specular refractive) {
40   float refractiveIndiceInv = 1.0 / refractiveIndice
41   glm::vec3 refractDir = glm::refract(rayDirection, shadingNormal, refractiveIndiceInv)
42   Ray transmissionRay (refractDir, intersectionPoint, rayDepth + 1, intersectionPrimitive)
43   glm::vec3 LiT_RGB
44   rayTrace(&LiT_RGB, transmissionRay)
45   LiT += kT * LiT_RGB
46 }
```

Listing 3.15: Algorithm of Path Tracer Shader

Figure 25.: PathTracer shader.



3.4.2 *Samplers*

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

All samplers just provide one method for the user to use:

```
1 float getSample(unsigned sampleNumber);
```

Listing 3.16: Main methods of Sampler

The *getSample* method receives as parameter the number of the current sample and returns the the actual sample. An atomic variable *sample* is used to count the current index of the samples.

Constant

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

The algorithm of the *getSample* is just:

```
1 return value
```

Listing 3.17: Algorithm of Constant Sampler

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.

The algorithm of the *getSample* is then:

```
1 //atomic operation
2 current = sample++
3 if (current >= (domainSize * (sampleNumber + 1))) {
4  sample--
5  return 1
6 }
7 unsigned task = current - (sampleNumber * domainSize)
8 resSample = task / domainSize
9 return resSample
```

Listing 3.18: Algorithm of Stratified Sampler

This algorithm ensures that if the current sample is greater than the domain size, then it will always return 1 which means the end of the sampler.

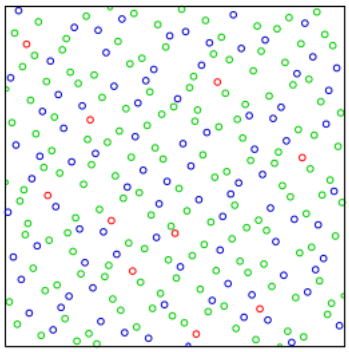
HaltonSequence

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 (with fewer samples).

Figure 26.: Halton sequence in a 2D image plane.



The algorithm of the *getSample* is as following:

```
current = sample++ //atomic operation
if (current >= (domainSize * (sampleNumber + 1))) {
  sample--
  return 1
}
unsigned task = current - (sampleNumber * domainSize)
float resSample = MobileRT::haltonSequence(task, 2)
return resSample
```

Listing 3.19: Algorithm of HaltonSequence Sampler

As it can be seen in listing 3.19, most of the Halton Sequence Sampler is similar to the Stratified Sampler. The only difference is the call to the *haltonSequence* method where it computes the Halton sequence for an arbitrary index and base number.

The Halton sequence algorithm is very simple as shown in listing 3.20 and it was taken from Wikipedia.

```
float haltonSequence(unsigned index, unsigned base) {
```

```
  float f = 1.0
  float result = 0.0
  while (index > 0) {
    f /= base
    result += f * (index % base)
    index = std::floor(index / base)
  }
  return result;
}
```

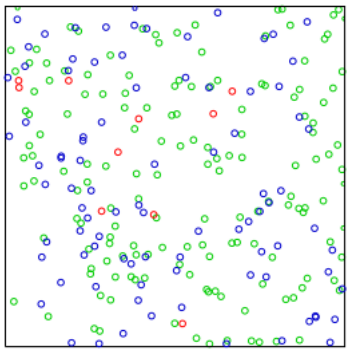
Listing 3.20: Developed method of HaltonSequence

MersenneTwister

This sampler is just a wrapper to call the constructor of std::uniform_real_distribution() and the constructor of std::random_device() of the standard C++ library.

This allow to construct a random number generator which calculates a random number between 0 and 1 with an uniform distribution. 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 commonly used a std::random_device for the generator seed because it produces non-deterministic random numbers, when supported. These sequences are usually used in simulations that use Monte Carlo methods like the Monte Carlo Ray Tracing.

Figure 27.: Pseudorandom sequence in a 2D image plane.



So the algorithm of the *getSample* is just:

```
static std::uniform_real_distribution<float> uniform_dist {0.0, 1.0}
static std::mt19937 gen(std::random_device {}())
```

```
3 float res = uniform_dist(gen)
4 return res
```

Listing 3.21: Algorithm of MersenneTwister Sampler

As can be seen in listing 3.21, the algorithm is very simple because it just constructs two objects in the first time the method is called:

1. std::uniform_real_distribution - which produces random floating-point values, uniformly distributed on the interval [a, b), that the user specifies.
2. std::mt19937 - pseudo-random generator of 32-bit numbers with a state size of 19937 bits.

Then every time the user calls *getSample*, the MersenneTwister Sampler just calls the operator() of std::uniform_real_distribution generator, which generates the next random number in the distribution.

3.4.3 *Lights*

There were implemented two types of light sources: PointLight and AreaLight. These light sources provide the user two main methods:

```
1 glm::vec3 getPosition()
2 Intersection intersect(Intersection intersection, Ray ray)
```

Listing 3.22: 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.

PointLight

The PointLight is the simplest form of a light because, as the name implies, is just a point of light.

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

```
1 return lightPosition
```

Listing 3.23: Algorithm of getPosition in PointLight

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.

```
return intersection
```

Listing 3.24: Algorithm of intersect in PointLight

AreaLight

The AreaLight implemented has a shape of a triangle. This shape is intended as it allows to generate a random point in a triangle with barycentric coordinates.

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 3.25: 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
}
```

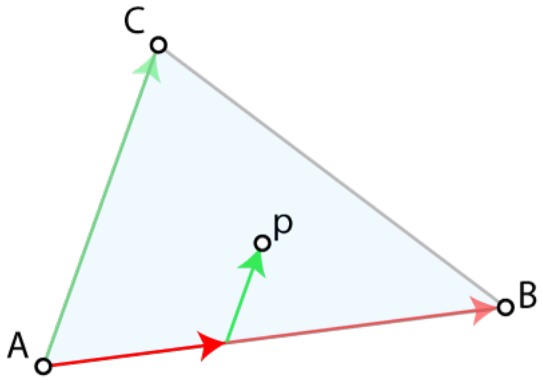
Listing 3.26: Algorithm of AreaLight

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:

```
RandomPointPosition = A + R∗AB + S∗AC
```

Listing 3.27: Algorithm of AreaLight

Figure 28.: Calculation of point P by using barycentric coordinates starting at point A and adding a vector AB and a vector AC.



So, the *getPosition* algorithm is as following:

```
1 float R = samplerPointLight->getSample()
2 float S = samplerPointLight->getSample()
3 if (R + S >= 1.0) {
4   R = 1.0 - R;
5   S = 1.0 - S;
6 }
7 glm::vec3 position = triangle.pointA + R * triangle.AB + S * triangle.AC
8 return position
```

Listing 3.28: Algorithm of getPosition in AreaLight

And the *intersect* method just calls the intersection of a ray with a triangle method presented earlier, and if it intersects updates the material.

```
1 float lastDist = intersection.length
2 intersection = triangle.intersect(intersection, ray)
3 intersection.material_= intersection.length < lastDist? &radiance : intersection.material
4 return intersection
```

Listing 3.29: Algorithm of intersect in AreaLight

3.4.4 *Cameras*

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

The camera only provides one method for the user to cast a ray from the camera position in direction to the image plane:

```
1 Ray generateRay(float u, float v, float deviationU, float deviationV);
```

Listing 3.30: Main methods in Camera

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 figure 29, 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 figure 30.

Figure 29.: Illustration of how sampling the plane image with jittering works.

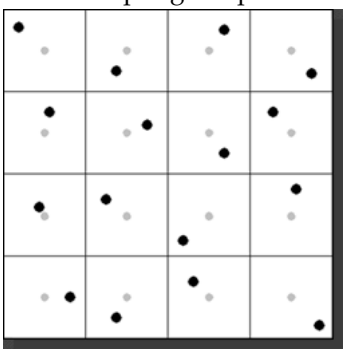
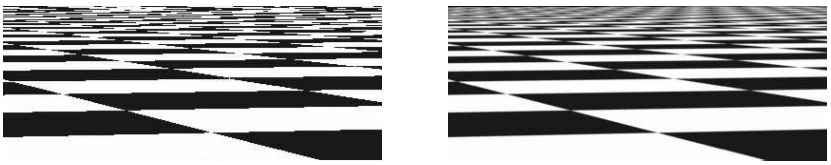


Figure 30.: Aliasing effect.

Sampling with jittering is, as previously stated, a good technique to avoid aliasing in the image. The figure 31 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.

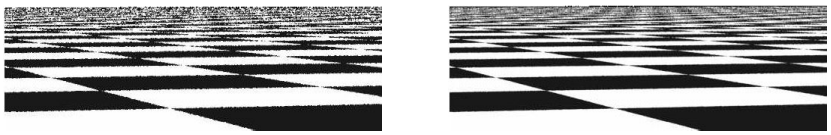


(a) Stratified sampling with 1 sample per pixel.

(b) Stratified sampling with 256 samples per pixel.

Figure 31.: Stratified sampling.

Sampling with jittering is, obviously not perfect, as shown in the figure 32. 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.



(a) Jittered sampling with 1 sample per pixel.

(b) Jittered sampling with 4 samples per pixel.

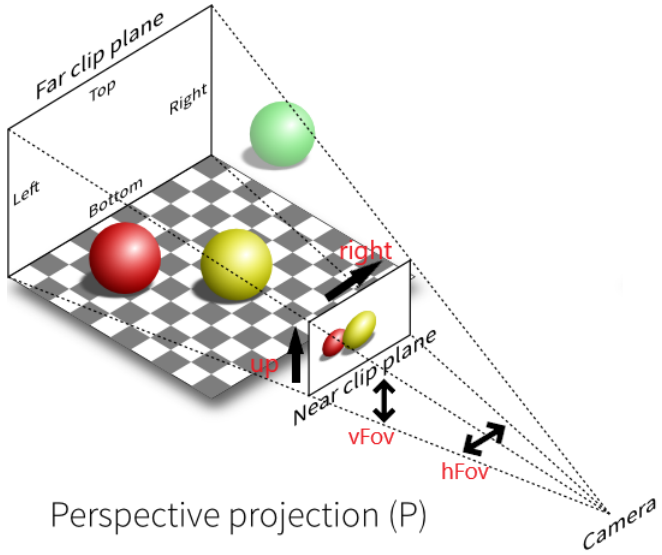
Figure 32.: Jittered sampling.

Perspective Camera

With this type of camera we 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 33.: 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.

```
1 float rightFactor = arctan(hFov * (u − 0.5)) + deviationU
2 glm::vec3 rightVector = right * rightFactor
3 float upFactor = arctan(vFov * (0.5 − v)) + deviationV
4 glm::vec3 upVector = up * upFactor
5 glm::vec3 destinationPoint = cameraPosition + cameraDirection + rightVector + upVector
6 glm::vec3 rayDirection = destinationPoint − cameraPosition
7 Ray ray = Ray(glm::normalize(rayDirection), cameraPosition)
8 return ray
```

Listing 3.31: Algorithm of generateRay in Perspective Camera

Orthographic Camera

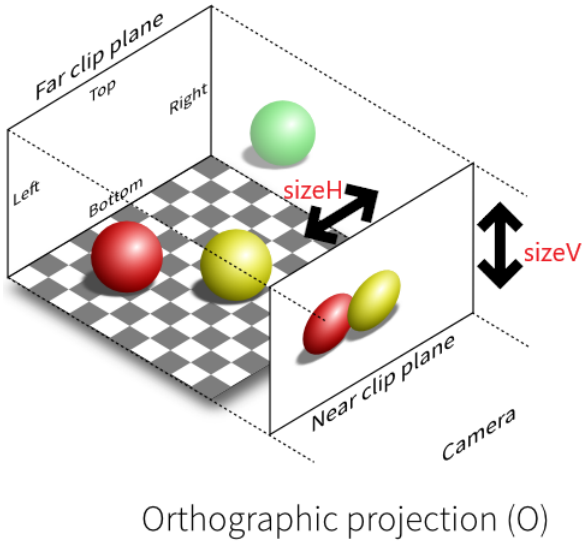
The orthographic camera removes the sense of perspective by drawing the image plane without simulating the depth of the objects, making all the objects looking flat. This is achieved by inverting the logic of the perspective camera. Instead of calculating the direction and always having the same origin, 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: $rayOrigin = cameraPosition + rightVector + upVector$.

```
1 float rightFactor = (u − 0.5) ∗ sizeH
2 glm::vec3 rightVector = right ∗ rightFactor + right ∗ deviationU
3 float upFactor = (0.5 − v) ∗ sizeV
4 glm::vec3 upVector = up ∗ upFactor + up ∗ deviationV
5 glm::vec3 rayOrigin = cameraPosition + rightVector + upVector
6 Ray ray = Ray (cameraDirection, rayOrigin)
7 return ray
```

Listing 3.32: Algorithm of generateRay in Orthographic Camera

Figure 34.: Orthographic camera with sizeH and sizeV (Schim).



3.4.5 *Object Loaders*

Last, but not least, the library allows to develop different 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

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.

This library only provides one Object Loader that allows loading the scene geometry from Wavefront obj files. 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.

OBJ Loader

The OBJ file format is a simple data format that allows to store only the position of each vertex, the UV position of each texture coordinate vertex, vertex normals, and the faces that make each polygon defined as a list of vertices, and texture vertices. Vertices are stored in a counter-clockwise order by default, making explicit declaration of face normals unnecessary.

```
1  # This is a comment
2
3  # List of geometric vertices
4  # x   y   z   w
5  v 1.0 2.0 3.0 1.0
6  v 5.0 6.0 7.0 1.0
7  ...
8
9  # List of texture coordinates
10 # x   y   w
```

```
vt 0.5 1.0 0.0
vt 0.1 0.3 0.0
...

# List of vertex normals
#  x   y   z
vn 0.3 0.0 0.7
vn 0.2 1.0 0.3
...

# List of polygonal face elements
# v1/vt1/vn1  v2/vt2/vn2  v3/vt3/vn3
f 03/01/01     04/02/03    05/03/04
f 06/06/03     07/01/02    08/02/06
...
```

Listing 3.33: .OBJ file format

The Wavefront OBJ file format is described as shown in 3.33, and is accompanied by another file, Material Template Library (MTL), that describes the visual aspects of the polygons. It is this file that describes surface shading (material) properties of objects within one or more .OBJ files. It allows to define multiple materials, with, ambient color, diffuse color, specular reflective color and specular refractive color. And even allows to load texture maps stored in TGA files. The listing 3.34 shows an example of a description of a material.

```
# This is a comment

# Define a material named 'Colored'
newmtl Colored

# Ambient color
Ka 1.0 0.0 0.0  # red

# Diffuse color
Kd 0.0 1.0 0.0  # green

# Specular color
Ks 0.0 0.0 1.0  # blue

# Dissolved (transparency)
d 0.4
```

Listing 3.34: .MTL file format

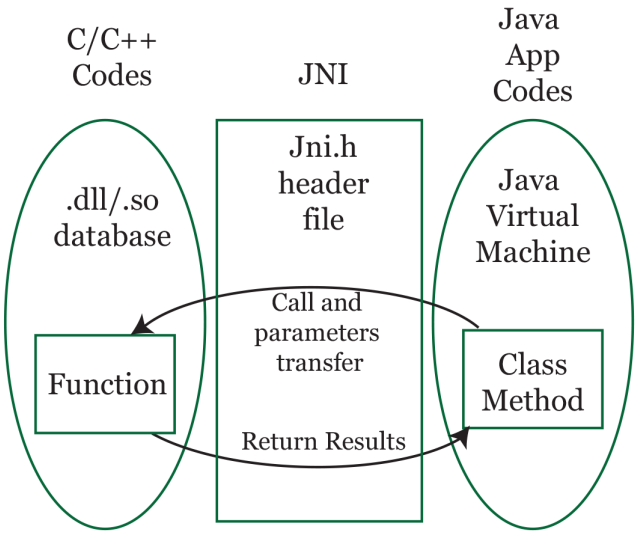
3.5 ANDROID SPECIFICS AND CHALLENGES

3.5.1 *Specifics*

Before developing the ray tracer to Android, it is necessary to understand how an Android application works.

A typical Android application is programmed in Java programming language. And it usually needs to communicate with an User Interface which also is programmed in Java. The code is compiled with Android Software Development Kit (SDK) along with any data and resource files into an Android package (APK). But, in order to avoid using the Java Virtual Machine (JVM) in our code and program it in native code, it is necessary to use Android Native Development Kit (NDK). Unfortunately, the NDK tool-chain doesn't provide any User Interface libraries like Qt that facilitates the build of a User Interface in native code. So, in order to obtain the best performance possible in a mobile device, the ray tracer library and components were programmed in C++ and the User Interface was programmed in Java. The User Interface calls the ray tracer available methods using the Java Native Interface (JNI) which is a programming standard that allows Java code running on a JVM to call native applications (programs specific to a hardware platform and operating system) and libraries written in other languages such as C, C ++ and assembly.

Figure 35.: Diagram explaining how the JVM can call native code.(Krajci and Cummings).



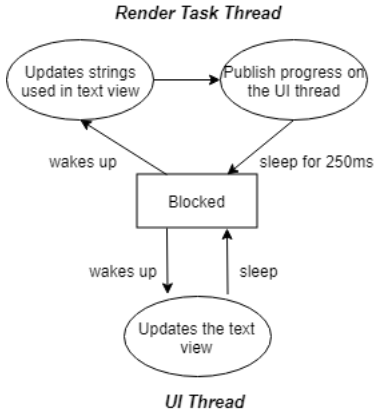
3.5.2 *User Interface*

The Android User Interface is programmed in Java and only one thread can refresh the User Interface (UI thread).

There are only two goals in the User Interface: 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 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 text view. Before it finishes, it publishes the progress on the UI thread. Then the UI thread wakes up and updates the text view.

Figure 36.: Execution flow of UI thread and Render Task thread.



3.5.3 *Challenges*

Developing applications for mobile devices have different challenges compared with the traditional personal computer hardware.

The Android User Interface has some particularities like only one thread can modify the UI, and is usually called the UI thread.

The size of RAM available for executing programs is typically smaller and the CPU microarchitecture is generally simpler and with smaller computational power. Also the Operating System (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 Java Native

Interface (JNI), so that the native code can send and receive information from Java Virtual Machine (JVM).

Because ray tracing may involve rendering complex scenes, placing all this information in the main memory may not be possible, and so this memory management can be a worthy challenge.

4

DEMONSTRATION: GLOBAL ILLUMINATION

4.1 RESULTS OBTAINED

The developed application was tested in five different devices:

- Samsung Galaxy Fresh Duos GT-S7392
- Raspberry Pi 2 Model B
- MINIX NEO X8-H PLUS (k200)
- Android Virtual Device (AVD) in an HP Z440 desktop
- Android Virtual Device (AVD) in a Clevo W230SS laptop

Table 2.: Devices specifications

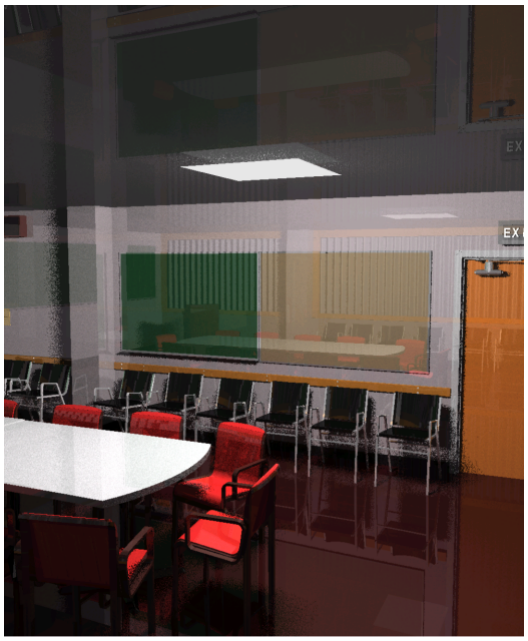
Device	CPU	Cache(L1/L2/L3)	GPU	RAM
Samsung Galaxy	1xARM Cortex A9 @1GHz	64KB/Unknown	1xBroadcom VideoCore IV	512MB
Raspberry Pi 2 Model B	4xARM Cortex A7 @900MHz	64KB/1MB	1xBroadcom VideoCore IV	1GB
MINIX NEO X8 PLUS	4xARM Cortex A9 @2.0GHz	64KB/Unknown	4xMali-450 MP	2GB
HP Z440 desktop	4xIntel Xeon E5-1620 v4 @3.5GHz	64KB/256KB/10MB	1xNvidia Quadro P2000	16GB
Clevo W230SS laptop	4xIntel® Core™ i7-4710MQ @2.5GHz	64KB/256KB/6MB	1xNvidia 860M	16GB

As you can see in the table 2, the application has been tested on a variety of devices. Unfortunately, it was only possible to test it on one mobile device, the Samsung Galaxy Fresh Duos GT-S7392. This device is a low-end smartphone with a low-end single core CPU.

It was also possible to test it on two different computers that are portable and even smaller than a common laptop. The Raspberry Pi 2 Model B and the MINIX NEO X8-H PLUS which are devices with the Android operating system installed.

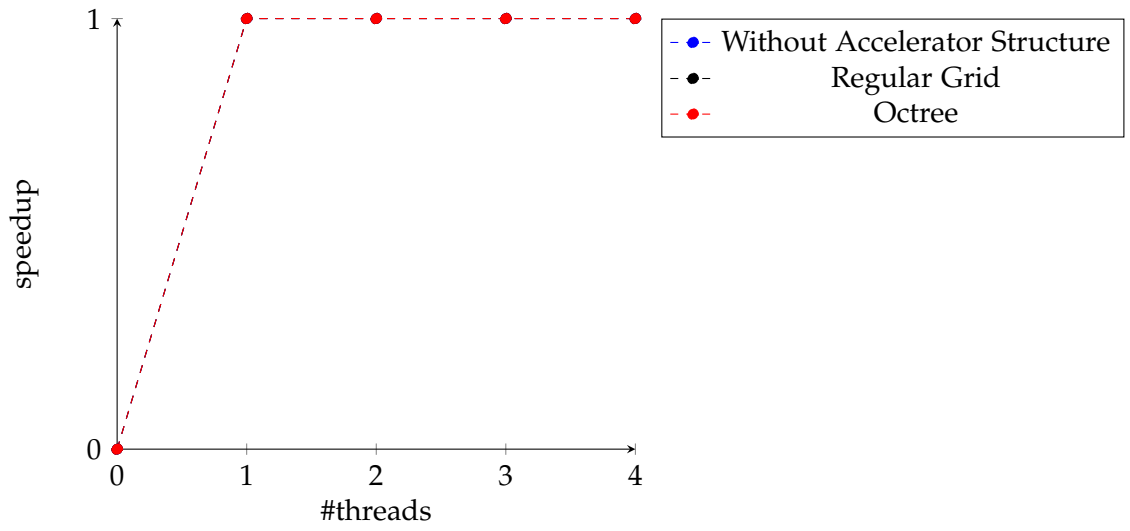
Last, but not least, it was also tested in two Android Virtual Devices, one desktop and one laptop. The desktop is a HP Z440 computer, which is a workstation, and the laptop is a Clevo W230SS, which is a middle end laptop.

Figure 37.: Illustration of conference scene, rendered with shader Whitted in MobileRT.

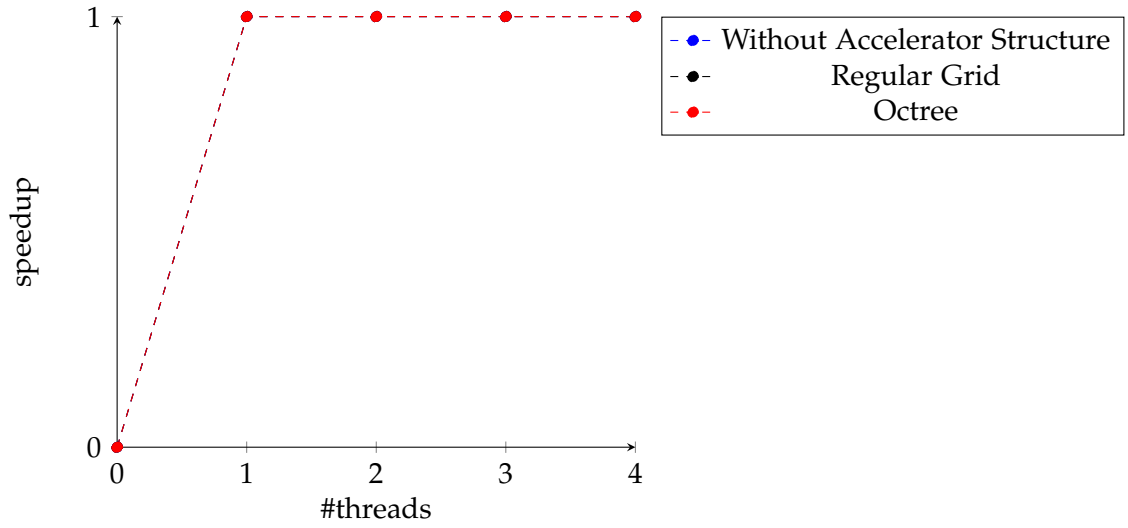


The scene used for testing was the conference scene, as illustrated in figure 37. This scene consists of 331179 triangles and has two area lights in form of triangles.

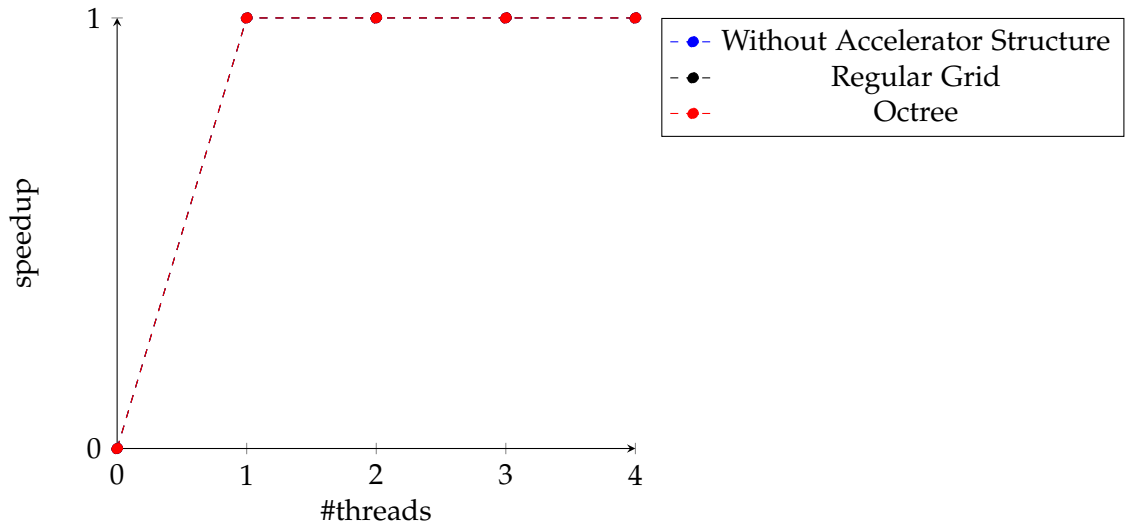
4.1.1 *Raspberry Pi 2 Model B*



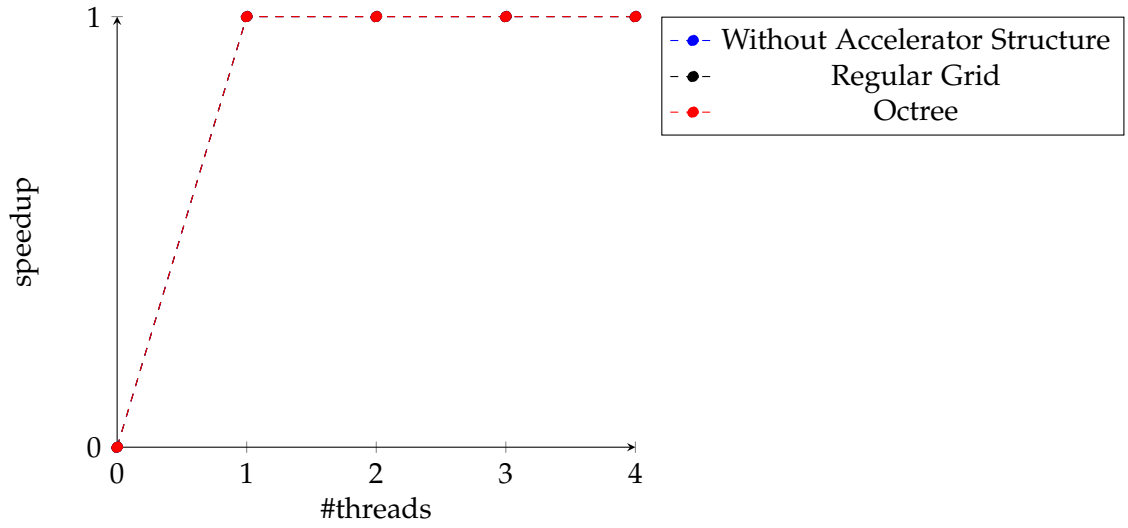
4.1.2 *MINIX NEO X8-H PLUS*



4.1.3 *Android Virtual Device in an HP Z440 desktop*



4.1.4 *Android Virtual Device in a Clevo W230SS laptop*



4.2 comparison with android cpu raytracer (Dahlquist)

Comparison ...

5

CONCLUSION & FUTURE WORK

5.1 CONCLUSIONS

The constant evolution of technology allowed to leverage and massify the 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 to, and where the development time can be a huge factor in their career success.

But as noted in the 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.

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 4.1.

5.2 FUTURE WORK

With respect to future work, it should be the development of the library for Android and the analysis of possibilities and challenges that may arise. Upon completion of the development process, besides writing the dissertation and the article, it is expected to provide a demo application in order to illustrate the functionalities and performance that the library will offer.

The library was implemented as planned, having been developed:

- two acceleration structures

- three primitive shapes
- intersection
- material
- primitive
- ray
- renderer
- scene
- possibility to develop different types of cameras
- possibility to develop different types of lights
- possibility to develop different types of object loaders
- possibility to develop different types of samplers
- possibility to develop different types of shaders

All these developed components allow 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 user the possibility to develop several types of rendering components, like cameras, lights, object loaders, samplers and shaders. With this, the user can create simple rendering applications that just use the rendering components offered together with the library, or even very complex applications, that use rendering components developed by the user.

Besides the library, it was also developed several rendering components:

- two cameras
- two lights
- one object loader
- six samplers
- five shaders

This allows the user to develop rendering applications in an easy, safe and fast way, as 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 you can cast rays to the scene from more than one different point. And thus open the possibility of allowing the user

to develop more different shaders, such as Bidirectional Path Tracing, or even allowing to render the 3D scene into a 3D image.

Other thing that could be implemented is a texture loader for the ray tracer. This allows the possibility to render more beautiful scenes.

And last but not least, there is always the possibility to improve the ray tracer performance by redeveloping the application using a Data-oriented design instead of the traditional Object-oriented design used. This approach approach was motivated by cache coherency, used in video game development, usually in the programming languages C or C++.

6

BIBLIOGRAPHY

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

Russ Bishop. Swift 2: Simd. URL http://www.russbishop.net/?page=5&x=272. Accessed: May 2018.

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: May 2018.

G-Truc Creation. Opengl mathematics. URL https://github.com/g-truc/glm. Accessed: May 2018.

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

Academic Dictionaries and Encyclopedias. Instruction pipeline. URL http://enacademic.com/dic.nsf/enwiki/141209. Accessed: May 2018.

Luís Paulo Peixoto dos Santos. Ray tracing clássico. URL gec.di.uminho.pt/psantos/VI2/Acetatos/02-RayTracing.ppsx. Accessed: May 2018.

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: May 2018.

The Foundry. Casting shadows. URL https://learn.foundry.com/nuke/8.0/content/user_guide/3d_compositing/casting_shadows.html. Accessed: May 2018.

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: May 2018.

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.

The Khronos™ Group Inc. Opengl es overview. URL https://www.khronos.org/opengles/. Accessed: May 2018.

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

Wenzel Jakob. Mitsuba renderer, 2010. URL http://www.mitsuba-renderer.org. Accessed: January 2017.

Michael Karbo and ELI Aps. Chapter 28. the cache. URL http://www.karbosguide.com/books/pcarchitecture/chapter28.htm. Accessed: May 2018.

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: May 2018.

kitware. Cmake. URL https://cmake.org/. Accessed: May 2018.

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: July 2018.

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: May 2018.

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.

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

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.

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: May 2018.

Computer Science and Engineering. Anti aliasing computer graphics. URL https://www.slideshare.net/DelwarHossain8/anti-aliasing-computer-graphics. Accessed: May 2018.

Scratchapixel. Ray tracing: Rendering a triangle. URL https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle. Accessed: May 2018.

syoyo. tinyobjloader. URL https://github.com/syoyo/tinyobjloader. Accessed: May 2018.

Raghu Machiraju Torsten Möller. Ray intersection acceleration. URL http://slideplayer.com/slide/7981872/. Accessed: May 2018.

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: May 2018.

Wikipedia. Halton sequence. URL https://en.wikipedia.org/wiki/Halton_sequence. Accessed: July 2018.

Alan Wolfe. Randomcode. URL https://github.com/Atrix256/RandomCode. Accessed: July 2018.

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

A

USER DOCUMENTATION