Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2016

Global Illumination in Real-Time using Voxel Cone Tracing on Mobile Devices

Conrad Wahlén

LINKÖPING
UNIVERSITY

Master of Science Thesis in Electrical Engineering

Global Illumination in Real-Time using Voxel Cone Tracing on Mobile Devices

Conrad Wahlén

LiTH-ISY-EX–16/5011–SE

Supervisor: **Åsa Detterfelt**
CEO, Mindroad
Mikael Persson
ISY, Linköpings universitet

Examiner: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Division of Information Coding
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2016 Conrad Wahlén

To myself, nobody likes me as much as I do (except Mom).

Abstract

This thesis explores Voxel Cone Tracing as a possible Global Illumination solution on mobile devices.

The rapid increase of performance on low-power graphics processors has made a big impact. More advanced computer graphics algorithms are now possible on a new range of devices. One category of such algorithms is Global Illumination, which calculates realistic lighting in rendered scenes. The combination of advanced graphics and portability is of special interest to implement in new technologies like Virtual Reality.

The result of this thesis shows that while possible to implement a state of the art Global Illumination algorithm, the performance of mobile Graphics Processing Units is still not enough to make it usable in real-time.

Acknowledgments

The process of writing this thesis has been a long one. A bit longer than I (and others) thought at the beginning. But I am grateful to everyone involved for the support and for pushing me over the finish line.

Special thanks to Mindroad and Åsa Detterfelt, to Mikael Persson and to Ingemar Ragnemalm.

Linköping, November 2016
Conrad Wahlén

Contents

Acronyms **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 Purpose 2
1.3 Problem Statement 2
1.4 Limitations 2
1.5 Source Code 3
1.6 Additional Details 3

2 Theoretical Background **5**
2.1 Light Transport 5
2.1.1 Rendering equation 6
2.1.2 Bi-directional Reflectance Distribution Function 6
2.2 Evolution of GPUs 7
2.3 Programming Graphics 8
2.3.1 OpenGL (ES) 8

3 Global Illumination Algorithms **9**
3.1 Finite Elements 9
3.1.1 Radiosity 9
3.2 Virtual Lights 10
3.2.1 Virtual Point Lights 10
3.2.2 Light Propagation Volumes 11
3.3 Tracing Algorithms 12
3.3.1 Path Tracing 12
3.3.2 Photon Mapping 13
3.3.3 Voxel Cone Tracing 13
3.4 Conclusion 15

4 Implementation of Voxel Cone Tracing **17**
4.1 Overview 17
4.2 Shadow Mapping 18

4.3 Render Data 18
4.4 Voxelization 19
4.4.1 Direct Light 20
4.4.2 Data Storage 20
4.5 Mipmapping 21
4.6 Voxel Cone Tracing 21
4.7 Summary 24

5 Results **25**
5.1 Comparisons 25
5.1.1 Hardware 25
5.1.2 Timing method 25
5.1.3 Visual comparison 26
5.1.4 Average rendering time 30
5.1.5 Average time per step 30
5.1.6 Soft shadow angle 30
5.1.7 Voxel grid size 33
5.2 Analysis 36
5.2.1 Image comparison 36
5.2.2 Average rendering time 36
5.2.3 Average time per step 36
5.2.4 Soft shadows varying angle 36
5.2.5 Voxel grid size 37
5.3 Future Work 37

6 Conclusions **39**
6.1 Experiments 39
6.1.1 Method 40
6.1.2 Improvements 40
6.2 Problem Statement 40
6.2.1 Possibility 41
6.2.2 Scaling 41
6.2.3 Limits 42
6.3 Mobile and desktop development 43
6.3.1 OpenGL and OpenGL ES 43
6.3.2 Android 43
6.3.3 Hardware 44

Bibliography **45**

Acronyms

AEP Android Extension Pack.

AO Ambient Occlusion.

API Application Programming Interface.

AR Augmented Reality.

BRDF Bi-directional Reflectance Distribution Function.

BTDF Bi-directional Transmittance Distribution Function.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

GI Global Illumination.

GPU Graphics Processing Unit.

IR Instant Radiosity.

LPV Light Propagation Volume.

OpenCL Open Computing Language.

OpenGL Open Graphics Library.

OpenGL ES Open Graphics Library for Embedded Systems.

PM Photon Mapping.

PT Path Tracing.

RT Ray Tracing.

VCT Voxel Cone Tracing.

VPL Virtual Point Light.

VR Virtual Reality.

1 Introduction

As humans we are predominantly ocular creatures. Vision being our main sensory input to interpret and understand the world around us. It is not surprising then that recreating images of our world has been done since the dawn of humanity. Computer graphics has enabled an unprecedented opportunity to simulate and capture realistic images of our and other realities. Development of computational resources for these tasks has increased the quality and complexity of scenes dramatically. Still a lot of work remains to be able to interact with the rendered scenes.

1.1 Motivation

Ever since humans drew paintings on the walls of caves, we have been interested in making images and models of this world. From this innate passion both art and physics has some common ancestor.

The invention of computer graphics has resulted in a unique opportunity to merge art and physics; to create works of art that not only look real but stems from computational models of the real world, and to create unreal worlds that still behave as they would be real.

To achieve realism both the direct and indirect light must be simulated. Direct light meaning light that is directly shining on a surface and indirect light meaning that the light has interacted with the scene in some way first. Combining these two result in Global Illumination (GI).

Thanks to the work in [9], there is also an equation that can be used to calculate GI in a point, referred to as the rendering equation. While this equation is very difficult (which in science means practically impossible) to solve for most cases. By approximating it, it is possible to find solutions good enough for most purposes. As computational power is growing, fewer approximations need to be

made.

For most interactive and real-time applications, direct light and its effects are simple to compute. The problem with GI stems from the complexity with indirect light. Since environmental interactions could imply everything from simple bounces to effects such as caustics. These effects are usually approximated with techniques that use a minimal amount of resources. It can be precomputed textures where advanced lighting has been calculated before use. Or the screen information could be used to approximate indirect shadows, called screen space Ambient Occlusion (AO).

By using GI techniques, it is not only possible to remove many of the special solutions for lighting effects. But also to add effects that are otherwise difficult to simulate and add a lot of realism. For example caustics and soft shadows, both direct and indirect.

1.2 Purpose

Traditionally GI has been used for offline rendering [18]. Meaning it is not used in interactive or real-time applications. The increase in hardware performance and development of new algorithms has lead to implementations that are able to produce real-time frame-rates. There have also been demonstrations of simple variants on low-end hardware such as a mobile device.

While mobile hardware is still far from as capable as high-end desktop hardware, the chip architecture and the mobility it offers is unique. Considering the rise of Virtual Reality (VR) and Augmented Reality (AR), it offers a truly wireless experience. By making high-end graphics available on low-end hardware, it allows the experiences to be more immersive and easier to use.

An alternative to this is presented in [5], where graphics is calculated on a server and streamed to the device. The drawback of this approach is the need for a network connection which limits the mobility. A solution like this could also benefit by knowing the limits of the device.

1.3 Problem Statement

- *Global Illumination on mobile devices, is it possible using modern hardware?*
- *Is there a method for Global Illumination that scales well enough to be used on limited hardware such as a mobile device?*
- *What are the limiting factors of the mobile device? And are there any potential benefits of using mobile devices for GI?*

1.4 Limitations

The solution will only be available on devices with the following specifications.

- Android 5.0.1 (Lollipop), or later.
- Open Graphics Library for Embedded Systems (OpenGL ES) 3.1 + Android Extension Pack (AEP), or later.

The solution will be developed with the following priority.

- Frame rate
- Dynamic scenes
- Graphical glitches
- Visual Quality

The solution will be exclusively tested on a Samsung S7 Edge with the Mali T880 MP12 Graphics Processing Unit (GPU).

1.5 Source Code

The complete source code of the project is available as open source. It is licensed under the Beer-ware licence making it open to use for any purpose. The mobile implementation is available here [23], and the desktop implementation here [22].

1.6 Additional Details

This master's thesis has been conducted on the behalf of Mindroad.

2

Theoretical Background

In this chapter the theory behind GI as well as the practicalities with implementing it on a mobile device will be presented. Starting with light transport and the rendering equation in section 2.1. In section 2.2 follows a view on the hardware evolution of GPUs. Finally, section 2.3 talks briefly about graphics libraries and features a comparison of the graphics library Open Graphics Library (OpenGL) and OpenGL ES (section 2.3.1)).

2.1 Light Transport

Without light there is no visual information available and everything is dark. The physics of light is conceptually simple. Light leaves a light source, interact with the environment and end up in our eyes (even though many hold different views [25]). It is when light interacts with the environment that valuable information about the scene around us is created.

In the simplest case these interactions are limited to absorption and reflection. When light hits an object it is either absorbed by it or reflected. This is how certain objects appear in different colors. They absorb some of the light and the light that is not absorbed is reflected. It can then either interact with other parts of the environment or be observed. The light is absorbed and/or reflected according to the material properties of the object.

To add an additional layer of interaction, consider transparent objects, like glasses or windows. Here the light can either be absorbed (to give a tinted color to the window), reflected or transmitted through the transparent object. Based on the properties of this transparent object and its shape this can also cause refraction of the incoming light, meaning that the light will leave the object at a different angle than it arrived at. This will also cause other interesting ocular effects such as a pool appearing shallower than it actually is, and caustics on the

bottom of the pool.

Thicker transparent materials or semitransparent materials add the next layer of interaction. The aspect that needs to be considered is subsurface scattering. In this case the light enters the object but instead of exiting on the other side, it is reflected within the object and exits at some other location. A good example is a material like jade.

Another layer of interaction would be to consider the effect of the light on the media it is transmitted through, called participating media interaction. In air the effects are small and hardly noticeable for the most part. One noticeable effect is the color of the sky which is an effect of light interacting with the atmosphere. For other media, like water (when diving), the effects are more apparent.

There are other properties of light worth considering as well, like polarization, fluorescence and phosphorescence. These types of interactions between light and environment give noticeable effects but are not as commonly seen.

2.1.1 Rendering equation

To make a simulation of this physical description of light, the problem need to be rephrased in computable maths. A general computational model, called the rendering equations, was first presented in [9]. These equations are shown below.

Equation 2.1 states that outgoing radiance from a point ($\mathbf{x}$) in a direction (ω) in the environment will be the sum of emitted and reflected radiance.

$$L_o(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + L_r(\mathbf{x}, \omega) \tag{2.1}$$

The reflected radiance from a point in a direction is given by equation 2.2.

$$L_r(\mathbf{x}, \omega) = \int_{\Omega^+} L_i(\mathbf{x}, \omega_i) f_r(\mathbf{x}, \omega_i \to \omega) \max(\vec{n} \bullet \omega_i, 0) \mathrm{d}\omega_i \tag{2.2}$$

The integration is over the upper hemisphere (Ω^+ oriented around the normal of the point. The incoming radiance (L_i) is the outgoing radiance of a certain direction at another point in the scene. The second function (f_r) is the Bi-directional Reflectance Distribution Function (BRDF) and will be explained in the next section. The final term is a scaling factor based on the incident angle of the incoming light.

The goal of a GI algorithm is to solve (or approximate an answer to) these equations.

2.1.2 Bi-directional Reflectance Distribution Function

In the previous section the general equations for how light can be modeled to interact with a scene were described. In this section the particular part of the BRDF will be explained. When light hits a surface it can be reflected or absorbed, as described in a previous section. However, the manner of how the light will be absorbed and reflected is determined by the surface BRDF, which is a mathematical representation of its material properties. As seen in equation 2.2, the

BRDF calculates the amount of outgoing light from the point **x** in the direction ω coming from direction ω_i.

There are two restrictions for the BRDF to be physically plausible. It has to conserve energy; it cannot send out more light that is coming in. And it has to be symmetric; the outgoing radiance has to be the same if the incoming and outgoing directions are swapped.

Different surfaces reflect incoming light in different ways. The two extremes are: light will be spread evenly on the hemisphere in the surface normal direction, or the light will be reflected. The first case is called a Lambertian surface and an example of one is matte paper. Regardless from where you look at surface, the brightness will be the same. The other extreme can also be called a mirror.

The first example is usually referred to as the diffuse part of the light calculation. A more general formulation of the second extreme where the light is not simply reflected but spread in the direction of reflection is usually called the specular part. Combining the diffuse and specular part creates a good approximation for most common BRDFs.

To describe more of the layers in the start of the chapter more details can be added to the rendering equation. For example transmitted radiance can be described with a Bi-directional Transmittance Distribution Function (BTDF). However, this thesis will not cover these effects.

2.2 Evolution of GPUs

Rendering scenes in 3D is different from the physical approach of how light interacts with the environment. Instead of light coming from light sources to interact with the scene and get to our eyes, in computer graphics the traditional approach is instead to shoot camera rays at the scene and collect the information from the objects that we hit. This saves a lot of work since only what is visible needs to be calculated.

Every object in the scene is represented by triangles. The triangle representation uses the least amount of points necessary to represent a plane. So each object is essentially approximated by a set of planes. To get a better approximation, more triangles can be used. This means that for complex scenes a lot of triangles need to be drawn. They also need to be modified; rotated, translated, projected to make them look like objects in a scene.

The rendering and modification of triangles is what the GPU in a computer is made for. GPUs are very good at doing many simple operations on points, lines, vectors and matrices at the same time. The modifications that need to be made to the input data are usually the same and only the data is different. This kind of architecture is referred to as a SIMD architecture, single input (code is same) multiple data (different points).

Over time GPUs have been getting faster (both in clock speed but also gotten more cores) and have started to perform more general calculations. As the amount of operations per second that a single core processor can achieve is reaching its limit, the need for parallelization in computation has become more impor-

tant. The evolution for the GPU (a massively parallell processor) from a graphics processor to a more general processing unit is therefore quite natural. This has also meant that the way that a GPU is used in graphics has changed from simply calculation transformations and simple light models, to more advanced light simulation approaches (trying to solve the rendering equation). Some of these approaches will be discussed in the next chapter.

2.3 Programming Graphics

To make it possible to use the processing power of the GPUs, several libraries for programming them exist. Some are fully focused on calculations; like Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). And some are more focused on graphics; like DirectX and OpenGL. The main difference, outside of some performance difference and difference in syntax, is the platform's availability. CUDA can only be used by Nvidia GPUs and DirectX can only be used on the Microsoft Windows operating system. OpenCL and OpenGL are more platform agnostic and try to be the solution for cross-platform implementations. Therefore programming graphics on a platform such as Android there is only one choice: OpenGL ES.

Recently, another option, Vulkan has entered the scene and tries to combine both calculation and graphics on a lower level with full cross-platform capability. Vulkan is developed by the Khronos group, the same group that develops OpenGL and OpenCL. It is thought of as the next generation of OpenGL and is aimed at high performance development, the downside is the considerable effort needed to get started.

2.3.1 OpenGL (ES)

The OpenGL and OpenGL ES were until recently, the only graphics library available for hardware accelerated 3D graphics on mobile platforms. Since the embedded systems traditionally has featured limited hardware, OpenGL ES has been stripped of much of its functionality. The recent improvement of the power consumption on processors has led to rapid incorporation of more advanced features into the ES version. There is no longer a big difference in the major features available if using the latest version of both. However, there are still some minor features missing.

3

Global Illumination Algorithms

As explained in the previous chapter, GI algorithms try to solve the rendering equations. There are many different ways to approach the solving of the equations, a comprehensive list can be found in [18]. Some of the more successful approaches will be explained in this chapter along with a couple of implementations.

3.1 Finite Elements

A classic approach to solve the rendering equation is the finite element approach. The general idea in this kind of approach is to calculate the light transport between discretized patches of the environment or scene. Finding the light transport solution is a matter of solving a set of linear equations, which can be solved directly or iteratively. This approach can deal with both diffuse and specular forms of light, but most implementations focus on the diffuse part. There are a couple of different implementations of this approach, one of which is radiosity, which is explained below in 3.1.1.

Using a finite element approach to solve the rendering equation and only focusing on diffuse light means that the result does not rely on the camera position. As long as the scene and lighting remains static the camera can move around the scene freely without recalculation of the light.

In short this approach is focused on getting a correct answer by calculation.

3.1.1 Radiosity

Radiosity is an iterative solution to the finite element method of solving the rendering equation. It typically only deals with diffuse forms of light. Although

radiosity has been used in other areas, the first computer graphics implementation can be found in [7]. The overview of the algorithm can be seen below in algorithm 1.

```
input  : 3D models constituting a scene
output : scene rendered to screen with GI
1 patches ←MakePatches(scene);
2 connections ←Connect(patches);
3 while iteration < MaxIteration or error > maxError do
4 |  patches ←IterateLight(patches, connections);
5 end
6 result ←Render(patches, scene);
```

Algorithm 1: Radiosity algorithm outline

The algorithm starts by discretizing the scene into patches. This is done to make each patch similar in size instead of the possibly large triangles a model is made of. This means an alternative representation of the scene is necessary.

Each patch is then matched with every other patch and a mutual light transport is calculated. If the patches are occluded from each other this transport is 0. The transport depends on the size of a patch projected on a hemisphere around the other patch. This means that patches with similar normals will not transmit much light to each other and similarly for patches that are far apart since their projection will be rather small.

When the transport of light has been calculated the final image is created by iterative application of the light between patches. In the first iteration only patches with direct light are lit up. Each following iteration will describe one bounce of the light. This iterative process can either continue until convergeance (`maxError`) or until a desired result has been achieved (`maxIteration`).

3.2 Virtual Lights

A different approach to solving the rendering equation is to insert virtual lights in the scene. Instead of finding the transport of light between all patches, virtual lights are inserted into the scene.

Parts of the implementations of virtual lights do depend on the position of the camera, therefore relying on some recalculation as the camera moves. However, the render time for each frame can be reduced if the scene is static.

In short this approach is focused on getting a fast approximate result.

3.2.1 Virtual Point Lights

The Virtual Point Light (VPL) algorithm is also known as Instant Radiosity and was first described in [11]. By placing virtual lights at locations in the scene and then rendering shadow maps for each virtual light, an approximation of diffuse GI can be achieved. In algorithm 2 below, the main outline is shown.

```
input  : 3D models constituting a scene
output : scene rendered to screen with GI
1 for i ← 0, i <= reflections do
2   pointLights ← Trace(scene, i);
3   foreach pointLight in pointLights do
4     shadowMaps ← RenderShadowedScene(scene, pointLight);
5   end
6 end
7 result ← Combine(scene, shadowMaps);
```

Algorithm 2: Virtual Point Lights algorithm outline

The VPL algorithm loops for a certain amount of bounces. In the first iteration only point lights from direct light sources are considered. And in the following iterations the light sources that are placed by tracing the scene will model indirect light bounces.

For each point light that is traced in the scene a shadow map is created. Different implementations of this algorithm use different resolution of the shadow map depending on the number of bounces of the point light. Since the diffuse light is very low frequency (no details) the more bounces the light has made the less detail is needed.

For the final rendering the shadow maps for each light are combined to make an approximation of indirect shadows and lights. If the amount of rendered lights are too few the result will be a banded image.

3.2.2 Light Propagation Volumes

Light Propagation Volumes (LPVs) is a newer algorithm made for real-time implementation and was first presented in [10]. The algorithm propagates the light in the scene using a grid representation of the light and scene. The main outline of the algorithm is shown below in algorithm 3.

```
input  : 3D models constituting a scene
output : scene rendered to screen with GI
1 LPV ← FindIndirectLights(scene);
2 blockerVolume ← UpdateBlockerVolume(scene);
3 LPV ← PropagateLight(LPV, blockerVolume);
4 result ← Render(scene, LPV, blockerVolume);
```

Algorithm 3: Light Propagation Volumes algorithm outline

In the algorithm above the indirect lights are found using an algorithm similar to the VPL algorithm in section 3.2.1, but tracing more points. This is possible since only the location and direction of the point lights are stored and no shadow maps are generated. The light information is then inserted into the Light Propagation Volume which is a grid representing the whole scene.

To represent the occluders of light, a second grid is created which is offset by half a grid from the light volume. This grid contains information about the scene geometry and which grid positions are occupied and not. To keep it dynamic and real time, the grid is automatically updated with information from the camera. The drawback is that geometry that is occluded from the camera (or has not been looked at) is not used in light calculations.

When blocking geometry and initial light has been calculated the LPV is updated by propagating the light in the volume. The light propagation is blocked by the geometry if necessary.

The rendering of the scene samples the resulting LPV to get an estimation of the indirect light in the scene. This volume can also be used for participating media effects by ray tracing.

3.3 Tracing Algorithms

The most popular approach to solving the rendering equation is by different tracing algorithms. These algorithms use paths, photons or cones to trace information about what light is likely to hit a certain surface or where light from a source should be applied to the scene. Tracing algorithms rely on performing a lot of traces to get an accurate result. They can also use approximations of either the distribution of light or the representation of the scene to get a better result with less traces.

This type of algorithm relies heavily on the location of the camera for most of the calculations. The traces are usually optimized to only consider paths that are seen by the camera to get results faster.

In short this approach is focused on getting an accurate answer by brute force (with some approximations).

3.3.1 Path Tracing

Path Tracing (PT) is similar to the concept of ray tracing. In a ray tracer, camera rays are traced into the scene. When a surface is hit, the contribution of all light sources to that point is calculated. A ray tracer does not typically deal with light bounces (unless in the case of a perfect mirror). The path tracer on the other hand lets the light bounce on surfaces to get the indirect light contributions as well. There are a couple of alternatives to PT; one alternative is to trace paths from light to the camera; another is to trace from the camera to the light sources. It is also possible to combine these two into a general concept of tracing paths from both camera and light and then combine the paths. This is called bi-directional path tracing, from [12]. An overview of this is seen in algorithm 4.

The algorithm traces paths from the camera into the scene, saving each bounce as a possible camera vertex. This is then repeated by tracing light paths from each light source and saving each such vertex as well.

The vertices are then connected together if there is a clear path between them. An improvement on this idea is called Metropolis Light Transport, from [21],

input : 3D models constituting a scene
output: scene rendered to screen with GI

```
1 cameraVertices ←TraceCamera(scene);
2 lightVertices ←TraceLights(scene);
3 paths ←Connect(lightVertices, cameraVertices);
4 result ←CalculateContributions(paths);
```

Algorithm 4: Bi-directional path tracing algorithm outline

which mutates the paths and saves the ones that contribute the most to the result. This alteration creates a better result faster, especially in scenes with narrow passages for the light.

When the paths have been established the result is rendered taking the contribution to each pixel from the paths it is connected to. To get a good result, a lot of paths are needed which takes a lot of time to calculate.

3.3.2 Photon Mapping

Photon Mapping (PM) traces photons from each light source and places them in the scene. The photons density for each pixel is then sampled to render the scene. The algorithm was first introduced in [8] and an outline is seen below in algorithm 5.

input : 3D models constituting a scene
output: scene rendered to screen with GI

```
1 photonMap ←TracePhotons(scene);
2 result ←Sample(photonMap);
```

Algorithm 5: Photon mapping algorithm outline

The algorithm starts by tracing photons from each light source into the scene. The photons can either be absorbed, reflected or transmitted when hitting a surface. When a photon is absorbed it is saved to the photon map of the scene. This acts as a density map of photons in the scene, storing positions, directions and colors of the photons that have been traced.

To render the resulting scene the photon map is sampled for the closest photons for each pixel. Then an estimation of the incoming light in that point is created, which is used to color the final pixel value.

3.3.3 Voxel Cone Tracing

Voxel Cone Tracing (VCT) was first described by [3] and [20]. It calculates an approximation of the indirect light in a rendered scene using a voxel representation of the scene. The voxel representation makes it possible to sample the scene in real-time. The outline is shown below in algorithm 6.

The algorithm starts by creating the voxel representation. There are several approaches for voxelizing a scene. In [3] the rendering pipeline is used to create

```
input  : 3D models constituting a scene
output : scene rendered to screen with GI
1 voxels ← Voxelize(scene);
2 while rendering do
3 |   voxels ← UpdateVoxels(voxels);
4 |   voxels ← Mipmap(voxels);
5 |   Trace(scene, voxels);
6 end
```

Algorithm 6: Main algorithm outline

voxels from fragments. [17] adds a triangle based algorithm to create a hybrid solution. A solid voxelization single pass algorithm is shown in [6]. During the voxelization information about the scene is saved in each voxel. What information to save is an implementation detail and several variations have been made, in [17] three different representations of voxel data are compared.

After the scene has been voxelized the direct light information should be added to the voxels. There are several variations to this problem as well. In [4] the light information is injected to the voxels by rasterizing the scene from each light source and adding the light information from each fragment. In [16] and [17] the reflected light is calculated at the moment of voxelization using the material data and a simple shadowmap.

There are some alternatives for storing the voxel data. In [3] a sparse octree data model is used for the voxel representation. The sparse octree approach removes the memory needed to store empty voxels and only stores actual voxels as leaves in the tree. The main issue with this approach is that the data structure is difficult to implement and update efficiently on the GPU.

A full octree is shown in [19], this approach is simple and can be implemented as a 3D texture, which also allows for simple mipmapping. The drawback of this approach is that it consumes a lot of memory even for scenes that are mostly empty. The structure does not offer a simple way of finding non-empty voxels. However, by constructing an active-voxel list in the voxelization step only active voxels need to be updated when lights or objects change, as shown in [17].

A more recent approach is to use clipmaps as shown in [16]. This is a modification of the full octree, instead of storing all data in the detailed levels they are clipped by distance. The mentioned drawback of this approach is flickering effects on smaller objects that are far from the camera.

In the final step of the algorithm each fragment is cone traced. Larger cones are used for diffuse light and smaller ones for shadows and specular light. Cone tracing steps through the voxel representation of the scene, each step sampling the scene using quadrilinear interpolation. The cone angle determines how quickly the step and mip level will increase.

3.4 Conclusion

An overview of the different algorithms presented in this chapter is shown in table 3.1.

Algorithm	Scalable	Alt. Representation	Real-time	Effects
Radiosity	No	Yes	Yes	D
VPL	Yes	Yes	Yes	D
LPV	Yes	Yes	Yes	D & S & E
PT	No	No	No	D, S & E
PM	Yes	Yes	Yes	D, S & E
VCT	Yes	Yes	Yes	D & S

Table 3.1: *Algorithm overview*

An algorithm is considered scalable if there are several implementation options that can be tuned for performance. An alternative representation is some kind of static representation that does not depend on the camera. Real-time means that there are current implementations of the algorithm that runs in real-time on any hardware. The effects are; D for diffuse indirect light, S for specular light and E for Extra, meaning extra effects such as caustics or participating media.

Of the presented algorithms, many would be good choices for performance GI. Indeed, VPL has already been implemented on mobile in [1]. However, the purpose of this thesis was to explore the limits of mobile GPUs so a more advanced algorithm was chosen. The issue with radiosity is that it mainly considers diffuse indirect lights, and for this implementation a full model of GI should be possible. Among the tracing algorithms the path tracers are the go to methods for exact results, unfortunately the rendering time of such algorithms are measured in minutes. Making them unsuitable for real time rendering. LPVs would be a good alternative, but lack of papers and adopted implementations disqualified it. The two remaining methods were therefore PM and VCT. And the choice fell on VCT since has soared in popularity and is the method of choice for Nvidias VXGI engine [15]. PM would have been an interesting choice but the implementation results seem to be closer to the interactive range than real-time.

4

Implementation of Voxel Cone Tracing

The algorithm implemented and investigated in this work is VCT.

4.1 Overview

The main outline of the implementation is presented below in figure 4.1 below. Each step of the algorithm will be presented in the following sections.

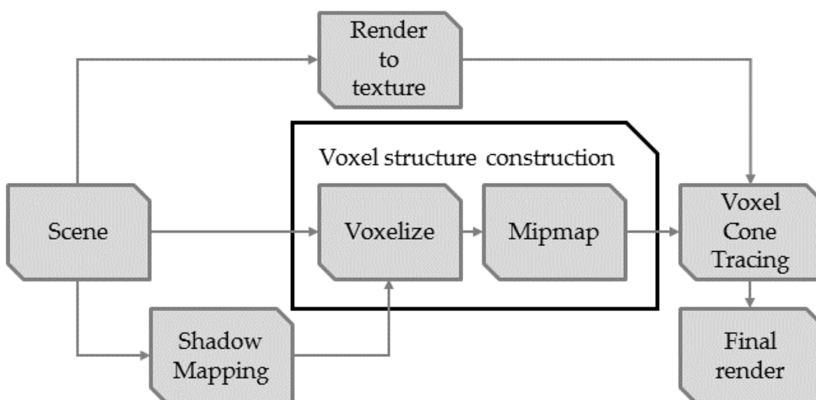

Figure 4.1: *Algorithm overview*

4.2 Shadow Mapping

The first step in the rendering pipeline is to calculate the shadow map as it is used by the later steps. Since only one directional light was used the shadow map is calculated by an orthographic rendering of the scene from the lights viewpoint, saving the depth buffer to a texture. When the shadow map is sampled the sample is considered to be in light if the depth of the fragment or voxel is less than or equal to that of the shadow map.

To reduce visual artifacts a bias term is used during the comparison. The term is calculated using equation 4.1, 4.2 and 4.3 below. Where $\vec{n}$ is the normal of the surface and $\vec{l}$ is the direction towards the light.

$$\theta = \max(\vec{n} \bullet \vec{l}, 0) \tag{4.1}$$

$$bias = 0.005 \cdot \tan(\arccos(\theta)) \tag{4.2}$$

$$bias = \text{clamp}(bias, 0, 0.01) \tag{4.3}$$

To make the shadows less harsh, percentage closer filtering is used. This means that each point on the surface will sample the shadowmap four times. The average of these comparisons is the final result, and can be seen below in figure 4.2. White in the image are parts of the scene which are hit with direct light. The implementation is based in large parts on [2].

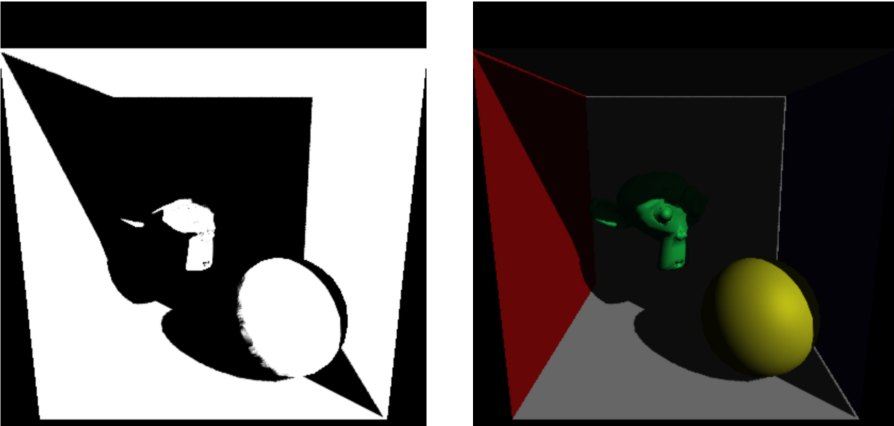

Figure 4.2: *Shadow map result (left) and scene shaded with shadow map (right)*

4.3 Render Data

The next step of the algorithm is to render the scene data to textures, in other words deferred shading. This is necessary since the tracing is expensive and

should only be performed for pixels that are going to be displayed. The data that is stored for each fragment is; albedo (diffuse color from texture or set color), position (in world coordinates), normals, tangents, bitangents (only on desktop), and depth. On mobile only four textures could be written in addition to the depth buffer in one pass. This resulted in the bitangent being calculated in the shaders using the normal and tangent, instead of rendering the scene a second time.

4.4 Voxelization

As the voxelization of the scene is crucial for the tracing part it is important that a good result is reached in this step. The implemented voxelization algorithm is from [3], which is a simple method to implement with good results for scenes with a mix of large and small triangles, seen in [17]. In this approach the rendering pipeline is utilized to create the voxels. The steps are described in algorithm 7. There are some issues with this approach which are discussed in [17].

The reason for choosing 3D textures in this implementation is mainly for simplicity. Clipmaps would have been the preferred alternative but little documentation for 3D clipmaps were found during the information gathering process.

```
input  : scene to be voxelized
output : Voxel representation of scene
1  foreach object in scene do
2      inner loop is standard rendering pipeline;
3      foreach triangle in object do
4          axis ←DominantAxis(triangle);
5          triangle ←Project(axis);
6          foreach fragment from Rasterize(triangle) do
7              data ←SampleFragmentData(fragment);
8              data ←CheckDirectLight(fragment);
9              texture ←Write(data);
10             if activeVoxelList [fragment ] is empty then
11                 activeVoxelList ←AddVoxel(fragment);
12             end
13         end
14     end
15 end
```

Algorithm 7: Voxelization process

This approach to voxelization utilizes the standard GPU pipeline to rasterize triangles into 3D textures. The conservative rasterization was skipped since the focus in the thesis was computation time and not accuracy, and the result was good enough without it. Basically this approach loops over all fragments which correspond to voxels in this case.

The scene is input to a vertex shader which simply passes along all parameters to a geometry shader. The geometry shader will calculate the dominant axis of the

normal and will then project the input triangle along that axis for rasterization. In the fragment shader the fragment data, color and shadow map result, is used to create a voxel (shown in 4.4.2. The count part of the voxel is set to eight, and the shadow map result is multiplied by eight. This is done so that the first iteration of the mipmapping is not a special case. The voxel is then inserted into a 3D texture using the fragment coordinates (x,y and z) as texture coordinates. The order they are used depends on the dominant axis.

An active voxel list is also created which contains the position of all voxels which are not empty along with the count of active voxels. The positions are stored in a 32 bit integer as a RB11G10. This allows for at least 1024 integer positions in each dimension. This list is used both for mipmapping the 3D texture and also for rendering the voxels. It could also be used to update relevant parts of the texture in dynamic scenes.

4.4.1 Direct Light

In this implementation the direct light contribution is calculated during the voxelization. The color of each voxel is calculated using equation 4.4.

$$color_{voxel} = \max(\vec{n} \bullet \vec{l}, 0) \cdot color_{diffuse} \tag{4.4}$$

The shadow map is then sampled to check if the fragment is hit with direct light.

This approach for direct light was chosen because it seemed to minimize the visual artifacts in a simple and efficient way. It fit nicely in to the overall pipeline and the overhead of calculating the shadow map was low.

4.4.2 Data Storage

The final step in algorithm 7 is $Write$(`data`). In this implementation isotropic voxels are used and the data saved is showed below in figure 4.3. The data is stored in a 3D texture. The reason for choosing isotropic voxels is that they are simple to implement, require least amount of memory and has the best performance, as seen in [17].

Figure 4.3: *Voxel data representation*

To be able to use the necessary atomic operations all information has to be stored in a 32 bit integer texture. To make the most use of the data the bits are used as showed in figure4.3. The information that is saved for each voxel is the result of the shadow map comparison and the color of the voxel. The order of the bits is important because voxelization uses the `atomicMax` operation to decide

if a value should be overwritten or not. This ordering makes lit fragments more important that unlit ones.

The reasoning behind the other bits is explained in the next section.

4.5 Mipmapping

The mipmap pipeline starts during the voxelization process by creating the first level of the active voxel list. The active voxel list consists in part of an indirect draw command buffer for each mipmap level, which is used for drawing only the non-empty voxels. The second part is an indirect compute command buffer for each mipmap level, which can be used for compute shader operations on the voxels, for example for creating all the mipmap levels in the 3D texture, as seen in algorithm 8.

```
input  : VoxelList of base level
output : VoxelList for each mip level
1 foreach MipLevel do
2     Each MipLevel has a corresponding VoxelList;
3     foreach Voxel in VoxelList do
4         VoxelNext ←Combine(Voxel);
5     end
6 end
```

Algorithm 8: Mipmapping process

The compute shader goes through the active voxel list and uses it to calculate the values for each voxel on the next level in the list. Since the data is saved in a 3D texture 8 voxels are used to create the data in the next level. The voxel data from the current level is combined in the following way. Each voxel will atomically add data to the next voxel to maximize parallelization. For each voxel the next level counter will increase by 1, the light counter will increase following equation 4.5. The color of the next level voxel will increase by $\frac{1}{8}$ of the color value of the current voxel. This allows the sampling of the voxels to calculate the average, though some precision is lost.

$$light_{next} = \begin{cases} 1 & \text{if } light_{current} > counter_{current}/2 \\ 0 & otherwise \end{cases} \tag{4.5}$$

4.6 Voxel Cone Tracing

The final step of the rendering is the voxel cone tracing. The cone trace samples points from the voxel representation in increasingly higher mip levels. In figure 4.4, a cone trace outline is shown. The trace starts at point x, in the direction ω and begins by sampling at point p_0, the distance d_0 to the first sample point is chosen carefully to not intersect with the starting objects voxels. The radius that

the sample should occupy r is given by equation 4.6. From that the mip level is given by equation 4.7. To find the next sample radius equation 4.8 is used, where the last term is the cone ratio and is constant given the angle θ of the cone to trace.

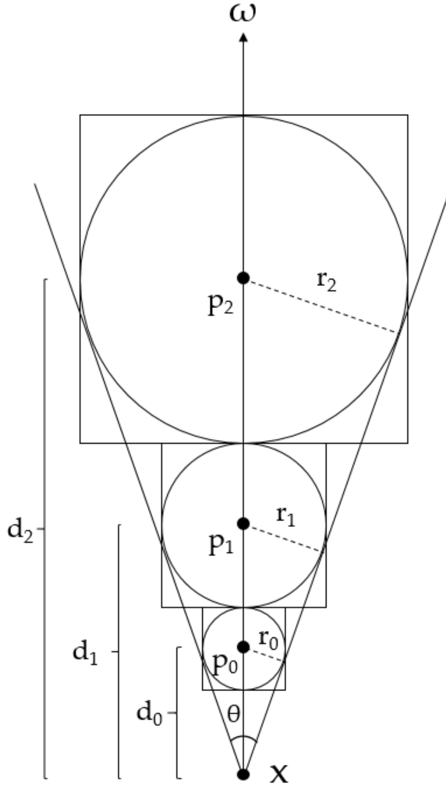

Figure 4.4: *Overview of the sample points when cone tracing*

$$r = d \cdot \sin\frac{\theta}{2} \tag{4.6}$$

$$mip = \max(\log_2(2 \cdot r), 0) \tag{4.7}$$

$$r_{i+1} = (r_i + d_i) \cdot \frac{\sin\frac{\theta}{2}}{1 - \sin\frac{\theta}{2}} \tag{4.8}$$

Two special cases can be noted in these equations, $\theta = 0$, which leads to $mip = \log_2(0)$ and $\theta = \pi$, which leads to $r_{i+1} = (r_i + d_i) \cdot \frac{1}{0}$. However, these cases are solved using two different approaches. A $\theta \approx 0$ would mean that the cone is close to a ray and should therefore use other methods for tracing. The case of $\theta = \pi$

means that the cone covers the whole hemisphere, which would not give useful results. Instead to sample a larger angle, multiple cones with different directions are used.

In this implementation two different cone traces are used. The one described above is the general cone trace and is used for the soft shadows and could be used for specular reflections. To get the shadows for each pixel a cone is traced towards the light. If a sample point includes filled voxels shadow value will decrease (zero is full shadow and one full light) depending on how filled the sampling area is. If the trace reaches the boundary of the scene or the shadow value has been decreased to zero the trace is stopped. The angle of the cone will determine how soft the edges of the shadow will be. The total shadow value for each pixel is calculated using equation 4.9.

$$S_{tot} = i \cdot S_{trace} \cdot S_{AO} + (1 - i) \cdot S_{AO} \tag{4.9}$$

One problem with the regular tracing is that the sample radii do not overlap, which might cause light leaking through thin geometry. In the diffuse case a modified version of the cone trace is used to minimize this risk. The specialized cone trace is seen in figure 4.5. By using 6 cones, each with an angle of 60°, the cone ratio is equal to 1. This means that if the first sample radius is half the size of the smallest voxel and the starting distance is the exact distance of the smallest voxel, each new sample radius will be twice the previous radius. The result of this is that only even mip levels will be sampled, which means there is no need for quadrilinear sampling and only 8 voxels need to be sampled instead of 16 each step of the trace.

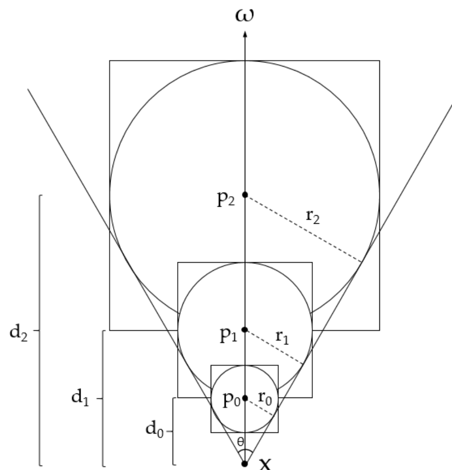

Figure 4.5: *Overview of the sample points when cone tracing*

The resulting color for each pixel is then calculated using equation 4.10, where c is the resulting color, either from a fragment or trace. As seen in the

equation, the result is only considering diffuse light, both when it comes to direct and indirect light. Multiplying the resulting diffuse trace with the color of the fragment is important to get realistic color interactions. A red indirect light should not color a green surface for example, this is an approximation of different colors being absorbed by different materials.

$$r = \vec{n} \bullet \vec{l} \cdot S_{tot} \cdot c_{light} \cdot c_{fragment} + c_{diffuse} \cdot c_{fragment} \tag{4.10}$$

The color of the trace is gathered in a similar way as the shadow value. Only during this trace two separate values are filled; the accumulation of the result and the color of the result. The color is determined by the average of the lit sampled voxels. The accumulation increases regardless if the sampled voxel is lit or not. When the accumulation reaches one or if the boundary of the scene is reached the trace of the pixel is stopped.

4.7 Summary

In table 4.1 below a short summary of the alternatives for the implementation is presented. The bold alternatives are the ones implemented in this thesis. The other alternatives presented were considered but disregarded due to the reasons presented earlier in this chapter. These are some of the available choices, but the ones seen below represent the most important ones.

Part	Alt. 1	Alt. 2	Alt. 3
Direct Light	Light injection	**Shadow map sampl.**	Voxel sampl.
Voxelization	**Per Fragment**	Per Triangle	Solid
Data storage	**3D Texture**	Clipmap	Sparse Octree
Voxel data	**Isotropic**	Anisotropic	Sp. Harmonics

Table 4.1: *Overview of the different choices for this implementation*

5

Results

The implementation of the selected algorithm presented in the previous chapter resulted in a mobile implementation of VCT. It was also implemented on desktop to compare the performance of the algorithm on different platforms and to compare how much code could be re-used on the different platforms.

5.1 Comparisons

Five different scenes were rendered to compare the performance of the algorithm and show the increasing detail and realism added by the implemented algorithm.

5.1.1 Hardware

The algorithm was tested on the following hardware to compare the performance of the implemented algorithm on a mobile, laptop and desktop GPU.

- Samsung Galaxy S7 Edge with Mali T880 MP12 GPU.
- Lenovo Y50 laptop with Nvidia GeForce GTX 860M GPU.
- Desktop with a Nvidia GeForce GTX 970 GPU.

Further specifications of the GPUs are shown below in table 5.1. An asterisk in the VRAM column signifies shared memory with the Central Processing Unit (CPU).

5.1.2 Timing method

To get the average times taken to render a frame, CPU timers together with the `glFinish` command was used. The `glFinish` commands made it possible to

GPU	Clock rate	VRAM	Cores	L2 Cache	GFLOPS
Mali T880	850 MHz	4 GB*	12	768 - 1536 kB	346.8
Nvidia 860M	1020 MHz	2 GB, 6 GB*	640	2048 kB	1317.1
Nvidia 970	1178 MHz	4 GB	1664	1792 kB	3494

Table 5.1: *Hardware specifications*

time individual steps for each frame by waiting for the command queue to execute. CPU timers were used because there are no GPU timers available for OpenGL ES 3.1, and using them on the other platforms could create skewed results. Each scene was rendered for a number of frames before a frame average over 5 frames was recorded. This was then repeated by restarting the program to get a fair average between multiple runs.

The Cornell box scene was run in multiple lighting conditions as follows.

- Scene 1: No GI
- Scene 2: No GI and shadowmapping
- Scene 3: AO and shadowmapping
- Scene 4: Diffuse indirect light, AO and shadowmapping
- Scene 5: Diffuse indirect light, AO and cone traced shadows

Each item in the list increases the performance and visual quality of the scene.

- In the first example the scene is shaded with a simple phong shader with both diffuse and specular reflections of the global light.
- In the second example a shadowmap is added to increase the realism.
- In the third scene cone traced ambient occlusion is added.
- In the fourth scene a full diffuse bounce is added.
- In the fifth scene the shadowmap is replaced by cone traced shadows.

Each scene, unless otherwise stated, was tested with a resolution of 400x400 pixels and used a voxel grid with 256x256x256 voxels. The traced shadow used a 5 degree angle and the shadow map resolution was 512x512 pixels.

5.1.3 Visual comparison

The resulting images from mobile and desktop are shown below in figures 5.1, 5.2, 5.3, 5.4 and 5.5.

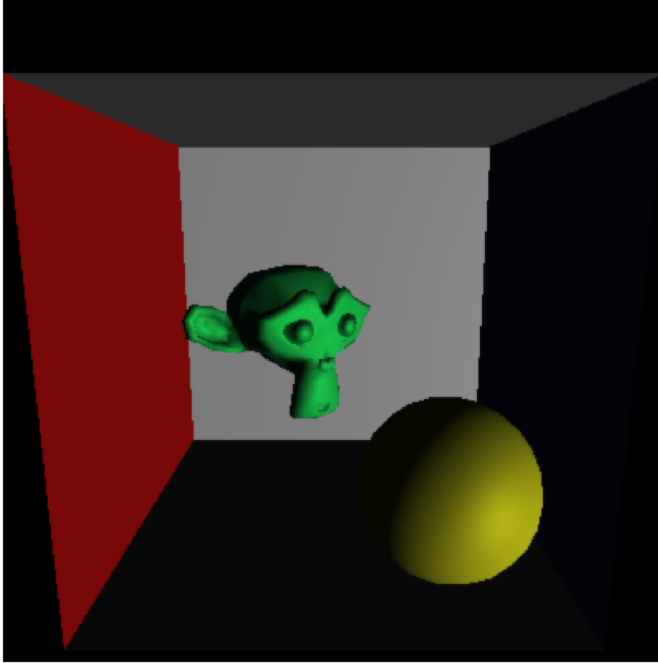

Figure 5.1: *Scene with only basic lighting*

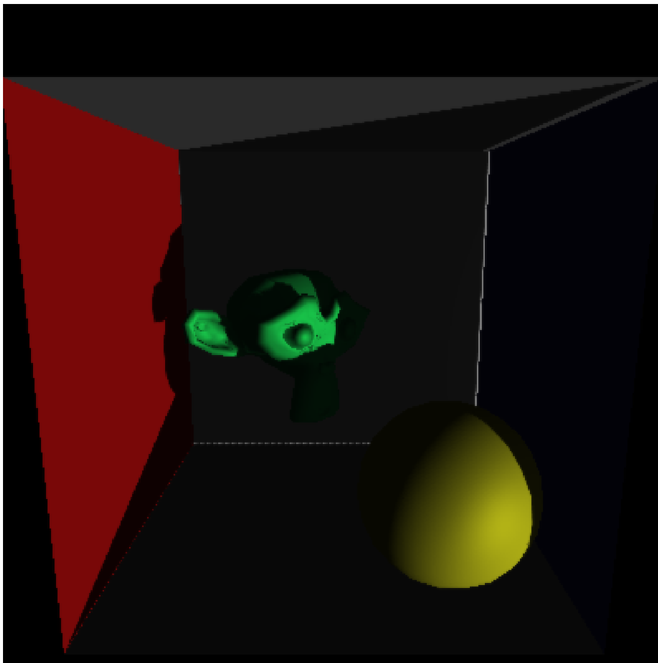

Figure 5.2: *Shadowmapped scene*

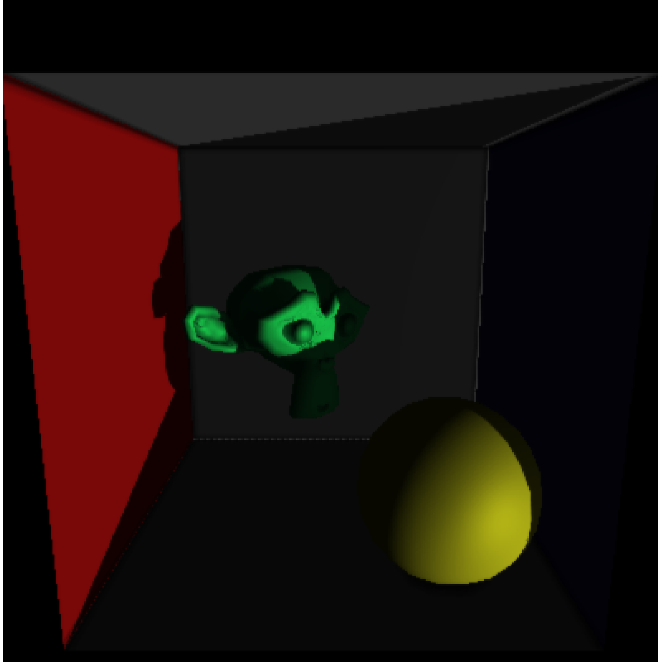

Figure 5.3: *Shadowmapped scene with AO*

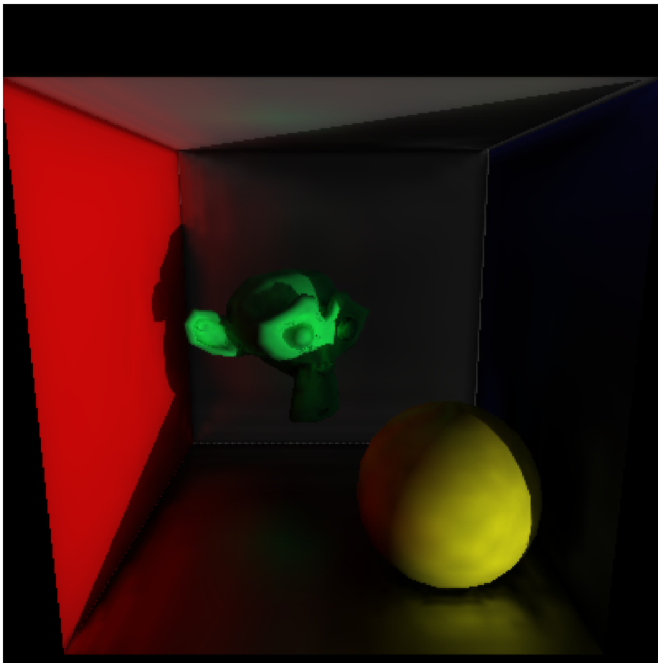

Figure 5.4: *Shadowmapped scene with diffuse GI and AO*

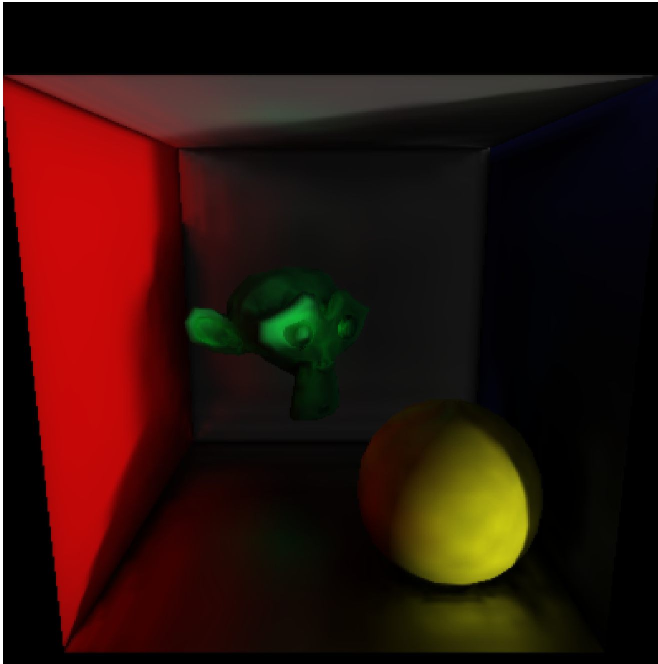

Figure 5.5: *Scene with diffuse GI and traced shadows*

5.1.4 Average rendering time

The average time taken to render a frame from each scene is shown in table 5.2. The rows marked with an asterisk are the averages when not voxelizing each frame.

Platform	Scene 1	Scene 2	Scene 3	Scene 4	Scene 5
Mobile	4.19	6.38	348.50	393.19	856.10
Mobile(*)	4.19	6.38	30.03	74.72	537.64
Laptop	1.11	1.30	6.06	10.52	27.16
Laptop(*)	1.11	1.30	2.24	6.70	23.34
Desktop	0.46	0.64	2.85	4.81	10.27
Desktop(*)	0.46	0.64	1.07	3.03	8.49

Table 5.2: *Average time (ms) per frame for each scene*

5.1.5 Average time per step

Table 5.3 below shows the time taken for each step in the algorithm rendering the scene with diffuse indirect light, ambient occlusion and traced shadows. The first column (CS) is the computation of the shadow map. Second (RT) is the time taken to render the scene data to textures. Then follows the voxelization (V) and mipmapping (M). The second last column is the trace or actual rendering of the scene (Tr). The final column shows the row sum.

Platform	CS	RT	V	M	Tr	Tot
Mobile	1.93	2.53	300.98	17.48	533.18	856.10
Laptop	0.22	0.89	2.87	0.96	22.23	27.16
Desktop	0.17	0.35	1.32	0.47	7.97	10.27

Table 5.3: *Average time (ms) per step for scene 5*

5.1.6 Soft shadow angle

In table 5.4 render times per frame using different cone angles for tracing shadows are shown. In figures 5.6, 5.7, 5.8 and 5.9, the scenes rendered with the different angles are displayed. The cone angles tested in this case are: 10 degrees, 7.5 degrees, 5 degrees and 2.5 degrees. A smaller angle gives a sharper shadow.

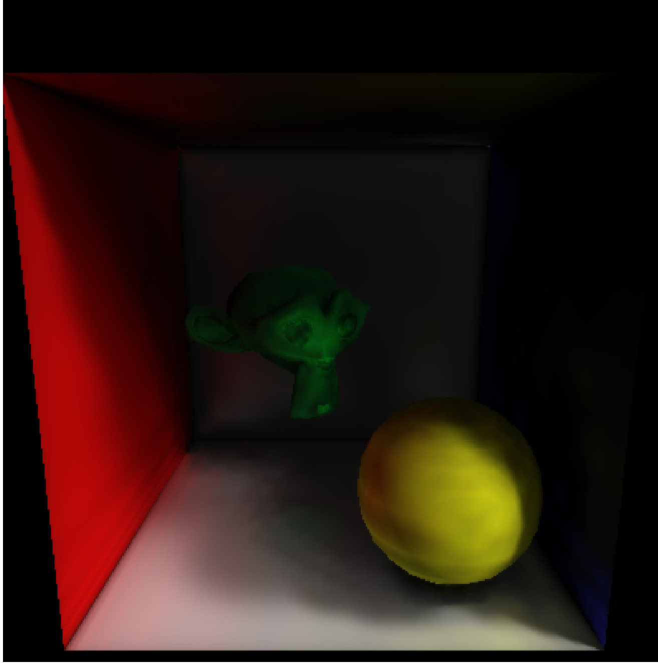

Figure 5.6: *Shadow angle of 10 degrees*

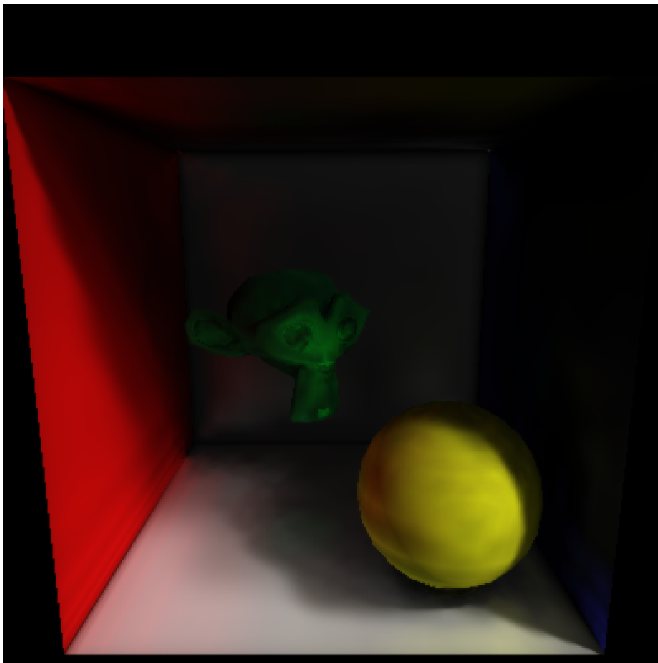

Figure 5.7: *Shadow angle of 7.5 degrees*

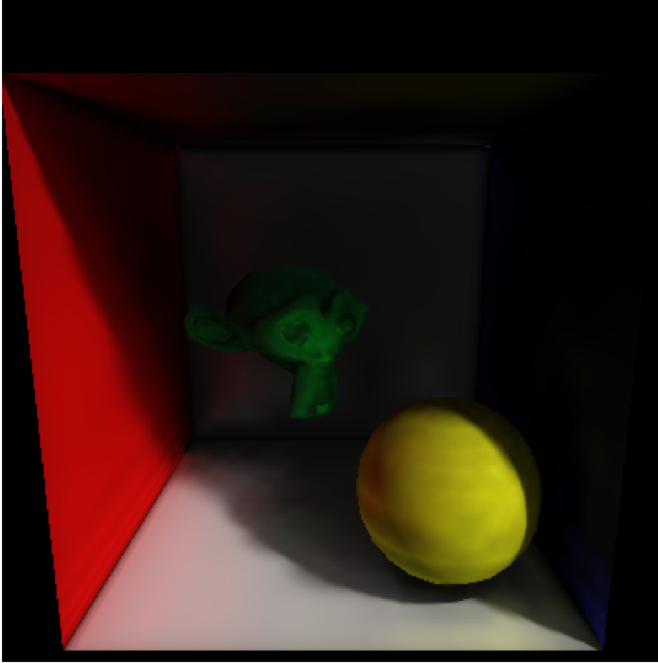

Figure 5.8: *Shadow angle of 5 degrees*

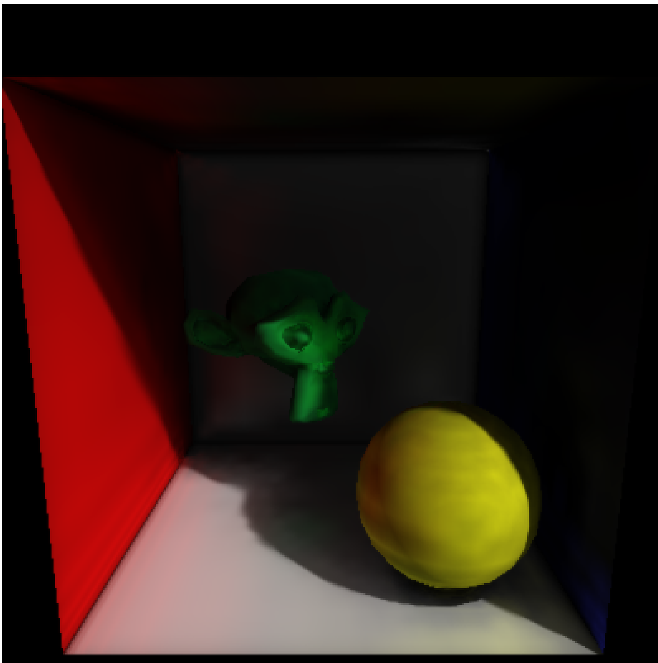

Figure 5.9: *Shadow angle of 2.5 degrees*

Platform	10 degrees	7.5 degrees	5 degrees	2.5 degrees
Mobile	291.57	378.04	536.82	1076.49
Laptop	14.37	17.11	22.68	38.31
Desktop	4.95	5.88	7.78	12.50

Table 5.4: *Average time (ms) per frame for different angles during shadow tracing*

5.1.7 Voxel grid size

In table 5.5 below the render time for different voxelgrid resolutions are shown. The voxelization (V) and mipmapping (M) are averaged over the scenes since it is the same calculations. The tracing (Tr) is shown separately for each scene. In figures 5.10, 5.11, 5.12 and 5.13, the scene rendered with different voxelgrid resolutions is shown.

	Voxel grid	128			256		
Platform	Scene	V	M	Tr	V	M	Tr
	3			14.59			25.68
Mobile	4	247.64	9.69	60.60	301.35	17.93	70.32
	5			484.38			530.43
	3			0.64			1.21
Laptop	4	0.75	0.37	4.78	2.84	0.99	5.54
	5			19.08			22.33
	3			0.31			0.53
Desktop	4	0.51	0.24	1.97	1.34	0.48	2.26
	5			7.00			7.56

Table 5.5: *Average time (ms) per step for different voxel grid resolutions*

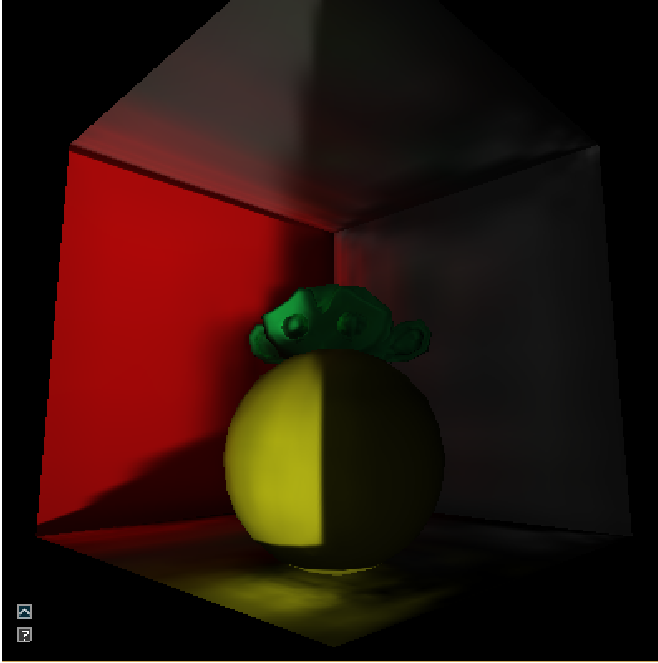

Figure 5.10: *Voxel resolution of* 64^3

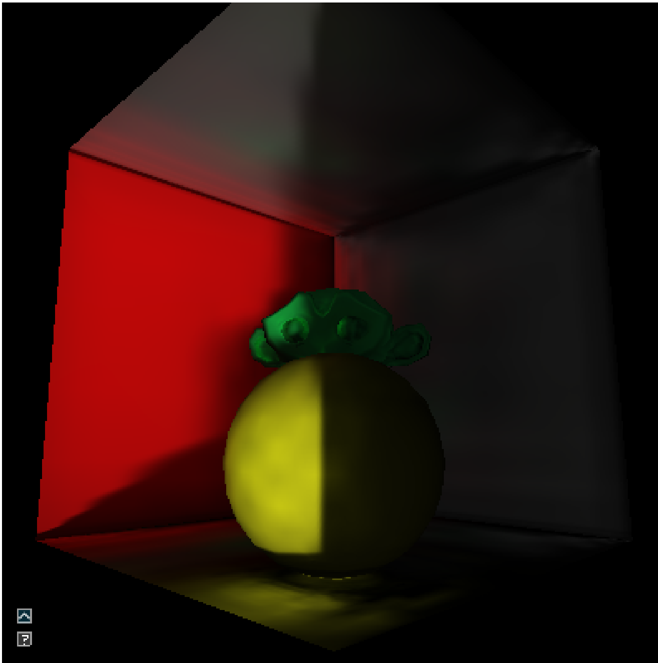

Figure 5.11: *Voxel resolution of* 128^3

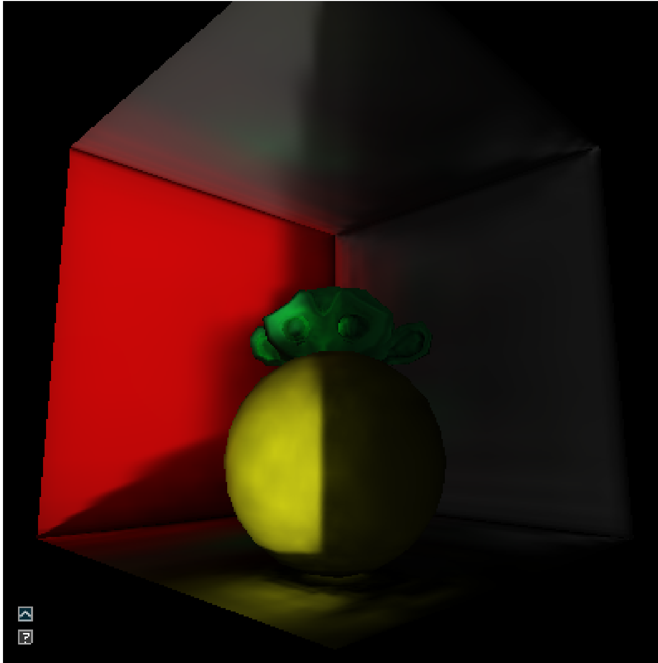

Figure 5.12: *Voxel resolution of* 256^3

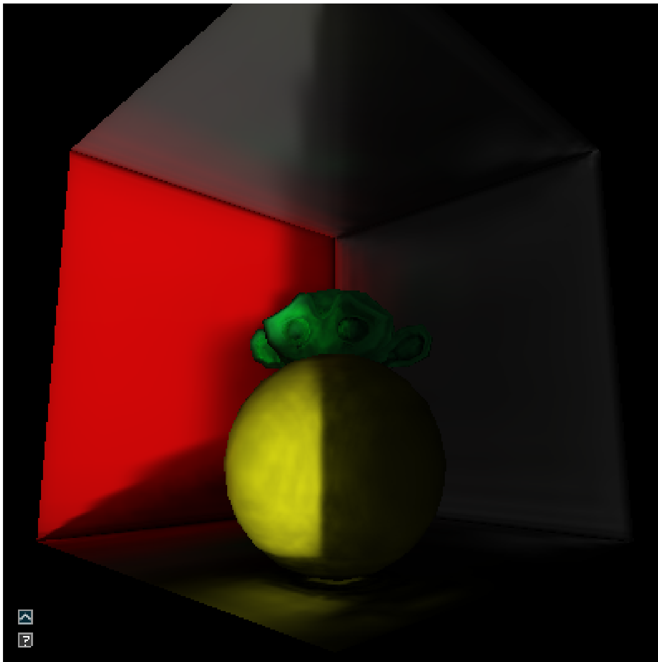

Figure 5.13: *Voxel resolution of* 512^3

5.2 Analysis

The results from the mobile and desktop show the same final rendering of the scene. The performance difference is clear between the different platforms, and though most of the results do not show a real time solution on mobile, the AO does run on real time frame rates.

5.2.1 Image comparison

As seen in the images in figures 5.1 to 5.5 the realism of the rendering increases for each image. The first image contains no spatial information, only the direction of the light can be gathered from looking at the image. The next scene adds direct shadows which gives some information about where the objects are located. In scene 3 the AO adds locality by shading areas that are close, like the corners and under the ball. In the fourth scene the indirect light is added which adds light to places that were previously unlit. In the final scene the soft shadows are added.

5.2.2 Average rendering time

Looking at table 5.2, it is clear that the overhead from voxelizing the scene each frame is significant, especially for the mobile platform. All scenes, both with and without voxelization, reaches real time frame-rates on desktop and laptop. For mobile it is only achieved when calculating ambient occlusion without voxelization each frame.

From the table it is also clear that traced shadows adds a lot to the rendering time of each frame, while there is less of a computational load to go from AO to diffuse indirect light.

5.2.3 Average time per step

As shown in table 5.3 the heaviest part of the algorithm is the cone tracing within the voxelized structure. The voxelization process, while taking a significant portion, still needs less time than the tracing. This is especially true for the desktop and laptop platforms where the ratio of time spent is dominated by the tracing. The mobile platform is more balanced between the voxelization and the tracing.

5.2.4 Soft shadows varying angle

The results of varying the shadow cone is shown to be significant in table 5.4, and as can be seen in figures 5.6 to 5.9 it has a clear effect on the result of the rendering. Looking at table 5.4 and 5.3, it can be seen that the larger angles decrease the time for cone tracing by enough to make it equal to the voxelization process on the mobile platform. The other platforms perform better on the voxelization even with a angle of 10 degrees.

5.2.5 Voxel grid size

The voxel grid resolution impacts the result of the voxelization, mipmapping and tracing as shown in figure 5.5. The voxelization and mipmapping increase noticably while the tracing times are increased slightly. The resulting renderings are shown in figures 5.10 to 5.13. There are some visual errors seen in the lower resolution renders, namely the light leaking under the yellow ball. There is also a noticable difference in the quality of the shadow close to the red/white corner. The diffuse indirect light is more focused in the higher resolutions, which is seen under the ball, but also on the red diffuse light.

5.3 Future Work

This work has a lot of areas to improve upon when it comes to performance. The most interesting ideas that were not implemented are the following.

- Use 3D clipmaps instead of mipmaps.
- Use filled voxel representation.
- Use low resolution light rendering and extrapolate to high resolution model rendering.

6 Conclusions

The results from the previous chapter show that the implemented algorithm does not reach real time frame rates on mobile. However, the scalability of the algorithm result in real time AO. Even though most of the code is usable on both mobile and desktop, there are some differences worth noting.

6.1 Experiments

The results of the experiments, as shown in the previous chapter, indicate that there is still more development required before real time GI is realistic on mobile hardware. The only applicable real-time use of the implemented algorithm was the AO. It might be possible with extensive optimization to realize diffuse indirect light as well. The tracing of shadows, and therefore also specular indirect light, is a long way off. Using VCT on a mobile device for something like AO might increase the visual quality compared to screen space AO, but it has a high cost in memory and does not work for dynamic objects.

The voxelization times on the mobile platform are high when compared to the other platforms. One reason for this might be that the only measurement on the mobile platform was during the initial construction since the 3D texture could not be cleared. It is therefore difficult to say how the voxelization process would perform when done continuously. Unfortunately there is no `clearTexture` in the next version of OpenGL ES (3.2) either. So a continuous voxelization has to be implemented with Vulkan if at all possible. However, it is possible to voxelize the scene each frame using a laptop or desktop. This also means that the voxel structure is completely static on the mobile platform.

In [24] it is suggested that mipmapping is a bottleneck, but in this thesis it is demonstrated that this is not the case. The active voxel list reduces the mipmapping to only those sections of the 3D texture which have voxels, which causes the

mipmapping to require less time than the actual voxelization. Another possibility which was not explored in this thesis is the possibilities of using the active voxel list to update light changes.

The choice of isotropic voxels were part because of the performance and memory aspects but also to utilize the `glGenerateMipmap` command, which would have been used instead of developing a mipmap solution. Unfortunately that command did not work while working on this implementation and mipmapping was done using the active voxel list instead.

6.1.1 Method

The different scenes used in the previous chapter were selected to highlight some important differences in visual quality and performance. The reason no specular bounce was demonstrated is because the same code was used to trace shadows, which gave a more distinct visual difference. Since it performs a similar function the performance difference would be insignificant.

Unfortunately there was no simple way of clearing a 3D texture in OpenGL ES 3.1, which meant that the voxelization could not be dynamic without reallocating a new texture each frame. This resulted in the timings for creating the voxel representation being based on fewer measurements and more importantly the measurements were taken on the first run of the function. This might skew the result towards a slower time than expected because of extra initialization costs.

6.1.2 Improvements

A problem with the voxel representation is that objects are empty inside. This causes several minor problems. For example when tracing two nearby objects, the initial sampling offset might cause the first sample to be taken from inside the hollow object. This causes the ambient occlusion tracing to fail, and the error is easily seen. For example in figure 6.1, the shadow looks nice a bit away from where the object touches the ground but closer to the contact point the shadow disappears.

The empty voxel objects also cause problems with the mipmapping. Deciding if a certain voxel in the next resolution should be empty or filled must depend on the existence of a single voxel, rather than deciding depending on the number of available voxels. This might cause objects to grow too much in higher mip-levels. With filled objects, small objects would instead be removed in higher mip-levels.

6.2 Problem Statement

With the support of the experiments in the previous chapter the three questions in the beginning of the thesis can be analyzed in more detail.

6.2.1 Possibility

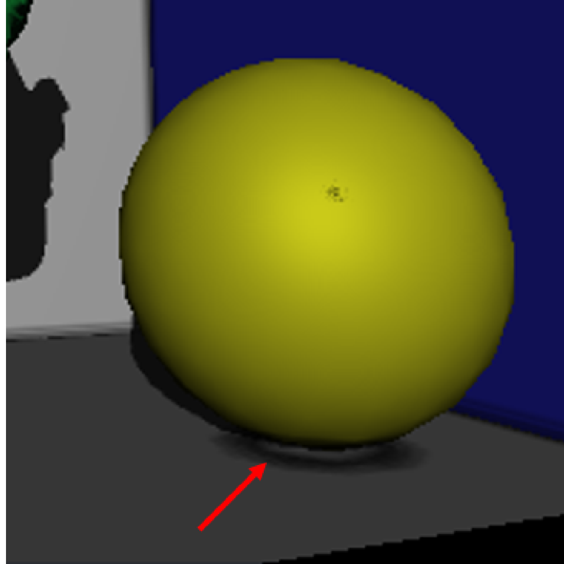

Figure 6.1: *Ambient occlusion error visible below the yellow ball*

Global Illumination on mobile devices, is it possible using modern hardware?

There have been previous results, like in [1], that show that simple GI algorithms are possible. But the results in this thesis indicate that the performance of modern mobile hardware is still not sufficient. A state of the art algorithm is still too much to handle without extensive optimizations. And with the current result in mind, the resulting visual quality might no longer be of interest.

In short advanced GI using VCT is still far from mobile graphics.

6.2.2 Scaling

Is there a method for GI that scales well enough to be used on limited hardware such as a mobile device?

When talking about scaling there are two major aspects to consider; memory scaling and performance scaling.

VCT requires a lot of memory for the voxel representation of the scene. For example the final implementation used in this thesis used a 3D-texture with a resolution of 256^3. Each voxel is represented by 4 bytes. This means $256^3 \times 4 \approx 67$ MB is needed. Increasing the resolution to 512 in each dimension yields, $512^3 \times 4 \approx 536$ MB. For good quality in larger scenes a high resolution is needed to sufficiently represent small objects. In this thesis each voxel described a part of the scene using only 4 bytes, describing color and light. Using more data would make it possible to also include addition information, such as a direction or even directed colors which should increase the visual quality and resolve some visual errors. An example of storing spherical harmonics for each voxel can be seen

in [17]. This means that the baseline for visual quality and memory is quite high, especially for larger scenes. Improvements such as clipmaps instead of mipmaps help to reduce the memory footprint, but it consumes a lot of memory compared to methods without alternative representations of the scene.

VCT is a tracing algorithm with little dependency between traces. But in the case of cone tracing, each trace consist of many cones, looping over steps. This can result in imbalanced workloads between threads. If the threads are grouped, like in the case of most GPUs, this leads to idle threads. Therefore, just scaling the performance is not as effective.

To scale the performance of VCT there are other options. Resolution is an important parameter that influences performance, since each pixel will result in a new trace, likely with a similar result. Two options to reduce this dependency are:

- Grouping traces after a certain distance, like in [14].
- choosing pixels to trace and filter the result.

Resolution is of particular interest for mobile devices. High-end mobile devices usually come with high resolution screens, often higher than a laptop screen. Meaning that mobile devices come with a double disadvantage. First they have an equal or higher resolution than laptops and the GPU performance is less than that of a laptop.

Looking at the rendering time in table 5.2 and comparing that to the GFLOPS performance of each GPU in table 5.1, the scaling between desktop and laptop corresponds equally. The GFLOPS scaling, $\frac{3494}{1317.1} \approx 2.65$, and the performance scaling (using scene 5, with voxelization), $\frac{27.16}{10.27} \approx 2.64$. However comparing the difference between desktop and mobile in GFLOPS, $\frac{3494}{346.8} \approx 10.07$ with the performance in scene 4 (without voxelization), $\frac{74.72}{3.03} \approx 24.7$, shows that the mobile performs worse that what might be expected. Especially considering that the cores on the mobile phone are independent. One explanation for this might be that the memory performance on mobile is slow, which would also explain why the voxelization performs so poorly.

6.2.3 Limits

What are the limiting factors of the mobile device? And are there any potential benefits on using mobile devices for Global Illumination?

The major limiting factor of the mobile device is the lack of performance of the GPU, which in turn is limited by power and size. A comparison of desktop and mobile GPUs in [13], also shows that there are different considerations that need to be made when implementing and optimizing algorithms. When it comes to the particular algorithm implemented in this thesis, there are some potential benefits using mobile hardware for advanced computer graphics (that could possibly be shown with more comprehensive experiments). Since the algorithm is completely implemented on the GPU, moving data between CPU and GPU is not a

factor when it comes to performance, which in other cases can make a big difference. As explained in the previous section the workload for threads in the same workgroup might be imbalanced, which can cause a problem on a GPU architecture that clusters multiple threads together (like most desktop GPUs). However, the mobile GPU used in this thesis has 12 independent cores meaning that this should not be a problem.

6.3 Mobile and desktop development

Developing for a mobile environment has both benefits and drawbacks compared to desktop development. This section will discuss differences encountered during the work on this thesis.

6.3.1 OpenGL and OpenGL ES

While the overall thinking in OpenGL 3.0+ is nearly identical to OpenGL ES 2.0+ there are differences to be aware of. Most differences deal with the often limited hardware that OpenGL ES is targeted at. Developing a high-end algorithm in this environment therefore had more limitations than just hardware.

Since OpenGL ES is aimed at low-performance hardware, many performance settings are explicit. Selecting precision in shaders is an example of this, which is something that is easy to forget coming from desktop OpenGL. This can cause errors that are hard to find.

Since many of the advanced features from OpenGL 4.3+ are available in OpenGL ES 3.1+ it is surprising to find some of the simpler ones are not available. For example the function `clearTexture`. The debug print tools does not appear until version 3.2 and neither does `CLAMP_TO_BORDER` for textures, which would have been useful to handle the case of a trace reaching the edge of the scene. Another feature missing is the ability to use subroutines in shaders instead of relying on switch case statements.

6.3.2 Android

Developing for a mobile platform, the availability of third-party tools and code is less frequent compared to desktop development. Especially when it comes to native development of GPU programming on mobile. Development for mobile devices is very specific for model and brand of the device which determines OpenGL implementations and available features. Compared to bigger programming languages, Android NDK is not very commonly used which affects the possibility to find answers to problems on Google and Stack Overflow. Just comparing the `android` tag to `android-ndk` tag on Stack Overflow speaks of the difference, over 900000 for `android` and just over 9700 for the `android-ndk` (October 2016). OpenGL is a bit closer with 27000 for `opengl` and 12000 for `opengl-es`. The NDK does allow more low level control which is necessary for certain optimizations, but it is also more complicated and lacks many of the included help functions available when using higher level Java code.

Utilizing the very latest features may also cause some problems. The mobile device used was able to run OpenGL ES 3.2, which has some useful features. However, there were no libraries available which made it possible to compile code for the newer API, since it was not part of the available version of Android. Also, no third-party solutions could be found.

6.3.3 Hardware

When developing for mobile it is not straight-forward to receive output and performance data. Especially not directly on the device. Even though current high-end mobile devices are powerful they are still not able to stand completely on their own. A laptop or desktop is still needed for debugging, tracing and output. Even though the screen has a high resolution there is no multi-tasking and because of the small physical size, text and input to be displayed is limited.

Another major difference is the lack of input methods for mobile devices. Everything has to made for touch instead of mouse and keyboard. This makes it much more difficult to modify variables live and more difficult to navigate a 3D environment.

Development on specific hardware can be both helpful and problematic. Performance evaluation tools are usually available for most platforms, and Android is no exception. However, the Android Studio performance tool required root access which was not available at the time of writing, and seemed limited when it came to OpenGL. Since the application used native C++ code and OpenGL, the number of tools available decreased. Many of the mobile GPU manufacturers have their own tools, which in the Mali case were very useful. However, the profiling tool to measure performance of the application was not free and not directly available in a free version.

Bibliography

[1] Minsu Ahn, Inwoo Ha, Hyong-Euk Lee, and James D. K. Kim. Real-time global illumination on mobile device. In *Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications*, volume 9030, pages 903005–903005–5, 2014. Cited on pages 15 and 41.

[2] contact@opengl-tutorial.org. Shadow mapping tutorial. https://web.archive.org/web/20160818072411/http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping//, 2016. Accessed: 2016-10-27. Cited on page 18.

[3] Cyril Crassin and Simon Green. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, pages 303–318, 2012. Cited on pages 13, 14, and 19.

[4] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011. Cited on page 14.

[5] Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirley, and Peter-Pike Sloan1 Chris Wyman. Cloudlight: A system for amortizing indirect lighting in real-time rendering. *Journal of Computer Graphics Techniques Vol*, 4(4), 2015. Cited on page 2.

[6] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0. URL http://dl.acm.org/citation.cfm?id=1375714.1375728. Cited on page 14.

[7] Cindy M Goral, Kenneth E Torrance, Donald P Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 213–222. ACM, 1984. Cited on page 10.

[8] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*, volume 364. Ak Peters Natick, 2001. Cited on page 13.

[9] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986. Cited on pages 1 and 6.

[10] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 99–107. ACM, 2010. Cited on page 11.

[11] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56. ACM Press/Addison-Wesley Publishing Co., 1997. Cited on page 10.

[12] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, 1993. Cited on page 12.

[13] Arian Maghazeh, Unmesh D Bordoloi, Petru Eles, and Zebo Peng. General purpose computing on low-power embedded gpus: Has it come of age? In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 1–10. IEEE, 2013. Cited on page 42.

[14] James McLaren. Cascaded voxel cone tracing, 2014. CEDEC Presentation. Cited on page 42.

[15] Nvidia. Nvidia vxgi engine. `https://web.archive.org/web/20160408115041/https://developer.nvidia.com/vxgi`, 2016. Accessed: 2016-11-07. Cited on page 15.

[16] Alexey Panteleev. Practical real-time voxel-based global illumination for current GPUs, 2014. GTC Presentation. Cited on page 14.

[17] Randall Rauwendaal. *Voxel based indirect illumination using spherical harmonics*. PhD thesis, Oregon State University, 2013. Cited on pages 14, 19, 20, and 42.

[18] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. *Comput. Graph. Forum*, 31(1):160–188, February 2012. ISSN 0167-7055. Cited on pages 2 and 9.

[19] Andrei Simion, Victor Asavei, Sorin Andrei Pistirica, and Ovidiu Poncea. Practical GPU and voxel-based indirect illumination for real time computer games. In *20th International Conference on Control Systems and Computer Science (CSCS)*, pages 379–384. IEEE, 2015. Cited on page 14.

[20] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games*, pages 103–110. ACM, 2011. Cited on page 13.

[21] Eric Veach and Leonidas J Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997. Cited on page 12.

[22] Conrad Wahlén. Desktop implentation source code. `https://github.com/AnotherHermit/VoxelConeTracing`, 2016. Accessed: 2016-11-12. Cited on page 3.

[23] Conrad Wahlén. Mobile implentation source code. `https://github.com/AnotherHermit/VoxelConeTracingMobile`, 2016. Accessed: 2016-11-12. Cited on page 3.

[24] Oscar Westberg and Mikael Zackrisson. Voxel cone tracing. Technical report, Department of Electrical Engineering, Linköping University, 2016. Cited on page 39.

[25] Gerald A Winer, Jane E Cottrell, Virginia Gregg, Jody S Fournier, and Lori A Bica. Fundamentally misunderstanding visual perception: Adults' belief in visual emissions. *American Psychologist*, 57:417, 2002. Cited on page 5.