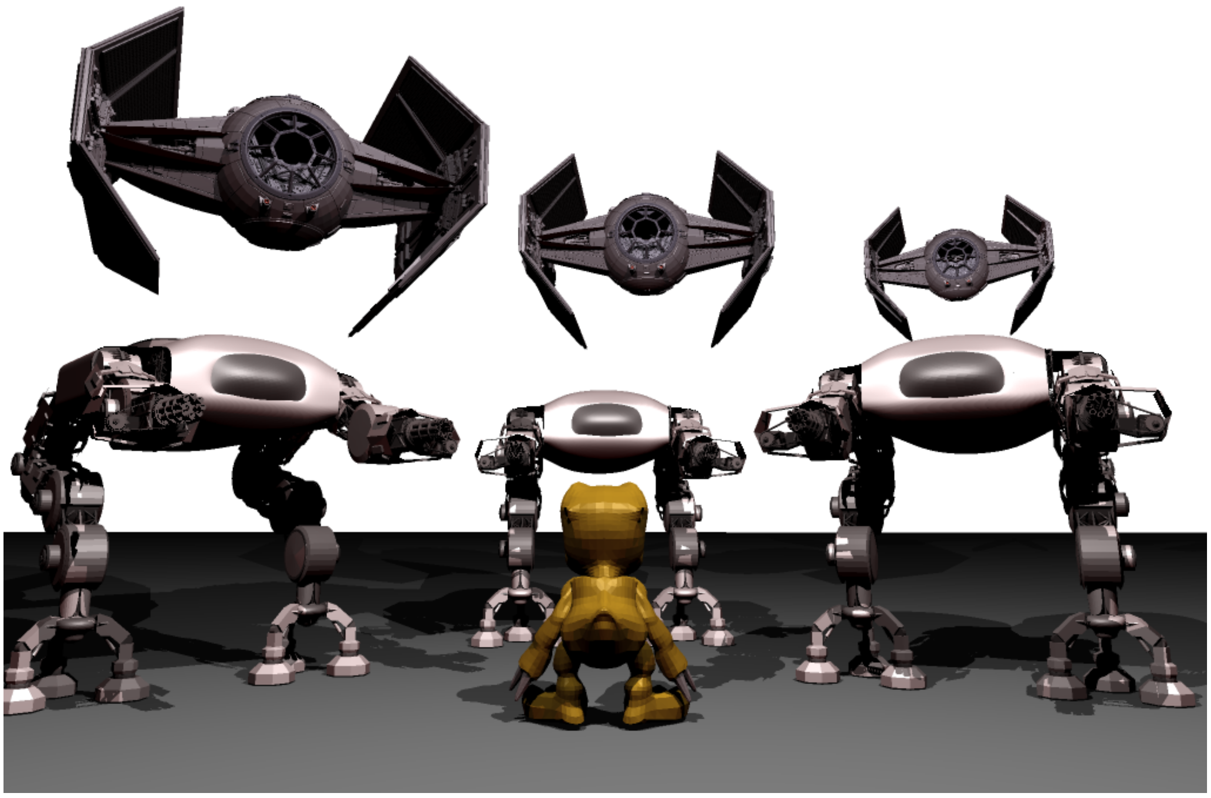
Real time ray tracing distributed

P2 project - Group B127
Computer science, Aalborg University Spring semester 2009

Authors:
Karsten Jakobsen
Anne K. Jensen
Jonas F. Jensen
Christina Lillelund
Sabrine Mouritsen
Thomas Nielsen

Supervisors:
Petur Olsen
Heather Baca-Greif

The faculty of Engineering, Science and Medicine at Aalborg University



Abstract

The purpose of this study is to investigate the relationship between the demand for real time ray tracing and the efforts needed to do this today. The main focus is the gaming industry, as it is expected to gain the most from more efficient rendering of photo realistic images. With the intention of learning whether there is a market for more photo realistic graphics, a questionnaire is conducted among presumed gamers and the market for graphics rendering is investigated. The efforts needed to achieve real time ray tracing today is investigated by making a ray tracer and examining the mathematical and programming related challenges. As the main obstacle today is hardware, the ray tracing software is distributed amongst several computers to increase the computing power available and thereby, to an extent, simulate the greater hardware of the future.

It is found that the questioned gamers do not see graphics as one of the most important things when buying new games. Many of the manufactures of computer hardware are working on faster ways to do ray tracing. Our own ray tracer was distributed among ten average computers - it was not possible to render a scene with an acceptable number of triangles in real time (approximately 1 million triangles). It is concluded that ray tracing will probably not be used in computer games within the next few years unless the rendering time is reduced.

Theme:
The reality of models

Project period:
P2, spring semester 2009

Project group:
B127

Group members:
Karsten Jakobsen
Anne K. Jensen
Jonas F. Jensen
Christina Lillelund
Sabrine C. H. Mouritsen
Thomas S. Nielsen

Supervisors:
Petur Olsen
Heather Baca-Greif

Prints: 10

Report page count: 94

Appendix page count: ???

Concluded the 25th of May 2009

The faculty of Engineering, Science and Medicine at Aalborg University
Computer science

Preface

This report documents the P2 project of group B127 at Aalborg University. This report describes the fundamentals of ray tracing, and briefly covers how some physical phenomenon can be simplified to a mathematical model that can be implemented in a ray tracer. This report is aimed at readers who are technically minded and not unfamiliar with programming, but who do not necessarily have any experience with ray tracing. The reader need not possess a knowledge of how physical light acts, however, familiarity with classical physics is a good idea, though not a necessity. This report assumes that the reader is familiar with the concepts of object orientated programming, as these concepts will not be explained in the report. The reader is also expected to be confident with vector operations, and a basic grasp of matrices would also be useful.

This report is released under the Creative Commons Attribution-Noncommercial-Share Alike 2.5 license and may be redistributed under these terms. The image on the frontpage is rendered in the TheMatrixDistributed, the ray tracer that accompanies this report. The scene contains around 10^6 triangles and was rendered with 4x anti-alias at 0.6 FPS using 6 laptops and one quadcore desktop computer.

Obtaining and running TheMatrixDistributed

The report is accompanied with the source for TheMatrixDistributed, which is a ray tracer written for this project. TheMatrixDistributed is release under GNU GPL and may be redistributed under these terms. The source, binaries and a pdf of this report should follow this report on a CD but can also be downloaded from
`http://jopsen.dk/blog/2009/05/thematrixdistributed/`.

TheMatrixDistributed runs easily under Linux, but requires libnetpbm 0.10 and SDL 1.2, which is available through most package managers. With these two dependencies TheMatrixDistributed should be easily executed or built from source. Guidelines for running and/or building TheMatrixDistributed are included on the CD and in the downloadable tarball as a README file.

TheMatrixDistributed can also with only minor issues be built for other unix-like systems, however, build system beyond what is offered by CodeBlocks does not accompany the source. Nevertheless, TheMatrixDistributed has been built on OS X and even Windows.

Contents

List of Figures

CHAPTER 1

Introduction

Today computer games can use two different rendering techniques to display graphics. The actual game is rendered with a technique called rasterisation, which is fast but does not create realistic light. In-game videos can be rendered with another technique called ray tracing, which makes it possible to create better graphics than rasterisation[Research@Intel, 2007]. Every scene consists of items like people, trees, houses and so on, which are built up by hundreds of triangles. To render a scene the computer must translate the 3D geometric representation of these triangles into a 2D screen image, and display it. The trick is to convert this in a realistic manner. To understand these techniques, an understanding of the underlying principles of three dimensional (3D) graphics is needed.

Rasterisation is the preferred technique today and has been used since the dawn of computer graphics. The rasterisation technique evaluates every triangle in the scene and projects them onto the screen. This method is fast because of the simplicity of the calculations. As it is necessary to evaluate every triangle, one at a time, the computing time increases with the number of triangles. However, some techniques can be applied to limit the number of triangles needing to be evaluated. The transformations used to project the triangles onto the screen are so characteristic that specialised hardware is used to speed up the process. This hardware is called a graphic processing unit (GPU). The downside of rasterisation is that realistic light and shadows are hard to achieve - even though techniques have been developed to approximate the real world - but creating them is time consuming [Research@Intel, 2007].

Ray tracing, on the other hand, is designed to render realistic light. Instead of projecting the triangles onto the screen, ray tracing emits rays from a camera and evaluates only the objects it hits. This approach - actually called backwards ray tracing - gives the opportunity to create shadows, by emitting a ray from the object towards the light sources to see if the point is visible from a light source. Reflection and transmission of light from one object to other objects can also be achieved easily by bouncing rays when they hit an object. The major shortcoming of ray tracing is this demand for many rays to be created, which increases the complexity. This is also why rasterisation is the preferred rendering technique today. Ray tracing needs more calculations, but it is possible to eliminate unneeded calculations. First of all, hierarchies can reduce the number of necessary rays and only shade the objects in front of the camera. Until today it has not been possible to use ray tracing instead of rasterisation because of the long computational time, but as CPUs get faster and bigger this obstacle might be overcome.

No hardware implementation is currently available for ray tracing. Therefore, ray tracing is usually implemented in software that runs on the Central Processing Unit (CPU). This is a ″number cruncher″ and is not specialised in performing the calculations required by ray tracing, which makes it much slower than rasterisation running on the GPU. Attempts have been made to apply ray tracing in real time applications. A recent example of this was a demonstration of QUAKE Wars: Ray Traced by Intel [Pohl, 2008]. It was intended to demonstrate the possibilities of ray tracing. The ray tracing engine was developed by the Intel Research department. They managed to get 15-20 frames per second (fps) in 720p resolution, using a 16 core setup clocked at 2.93GHz per core. This is approaching acceptable fps, as most casual gamers would be satisfied with +30 fps. Enthusiasts would however not be satisfied unless they were capable of getting at least 60 fps.

Researchers have been saying that real time ray tracing may become possible in the near future [Pohl, 2006, Suffern, 2007, Wald and Slusallek, 2001], however, real time ray tracing is not possible today on any readily available consumer PC. Nevertheless current hardware development indicates that it might be possible in the future.

Based on the current situation, the primary interest of this project will be:

- **Does the demand for real time ray tracing outweigh the efforts required to perform real time ray tracing today?**

CHAPTER 2

Project description

2.1 Problem analysis

The prevalent attitude toward ray tracing in real time today is that it is not yet possible [Bikker, 2008]. Some people who work with computer graphics see ray tracing as a part of a new rendering technique, and not as a substitute for rasterisation. Cevat Yerli from Crytek [Perspective, 2008a] and David Kirk, nVIDIA [Perspective, 2008b], are two experts both of whom see a need for a new rendering method where both ray tracing and rasterisation could be elements. Neither see the CPU or the GPU as able to do this rendering, but put their trust in programmable GPUs, which will not be limited to only one rendering technique, but programmable to do both rasterisation and ray tracing. David Kirk sees a hybrid of the two techniques as the real solution, because of rasterisation's speed and the realism of ray tracing.

Intel is of another opinion - they believe that a CPU with multiple cores will be able to do ray tracing in real time [Research@Intel, 2007]. Intel has chosen to take advantage of the advances made in CPU capabilities and are developing new hardware to follow this tendency.

Currently, CPUs are not the only hardware capable of ray tracing. It is possible to program some GPUs to do ray tracing, [Perspective, 2008b], but there are no standardised application programming interfaces (API) for accessing these features on the GPUs. This makes any larger scale development of GPU accelerated ray tracing impractical. Future hardware may be developed to do ray tracing in real time. Creating a real time ray tracing engine today usually means following Intels example by using many CPUs and limiting the graphical effects. This will of course reduce the realism and requiring more than one computer would not be practical. But to simulate the possibilities of the future of ray tracing, this is the only method available today. Cevat Yerli's and David Kirk's idea of combining rasterisation and ray tracing will perhaps be faster than pure ray tracing, although it is very likely that it would not be fast enough to be used in games. While rasterisation would speed up the process, the ray tracing element is still slow.

One thing is being able to make hardware capable of making ray traced computer games, another is selling it. Developing the needed hardware and the needed algorithms will demand much research, which in the end will demand a major market segment wanting better graphics. Without the demand for better graphics, the research will have a hard time becoming a reality.

2.2 Problem statement

Based on the above analysis the initial problem will be whether the demand for real time ray tracing outweighs the efforts required to do real time ray tracing today.

We will address this question by answering the following questions:

- Is there a demand for real time ray tracing?

The demand for real time ray tracing will be approached by sending out a questionnaire on gaming, which will then be analysed to conclude, amongst other things, if a graphical improvement of games is considered a requirement amongst gamers. The questionnaire will be distributed amongst the second semester students at the Faculty of Engineering, Science and Medicine at the Aalborg University. This group is appropriate, as the students generally belong in the age bracket, of which people can be expected to play computer games.

The history of graphics in computer games will be studied, to understand why the rendering of computer graphics is done the way it is today. How the hardware developers see the future of ray tracing is also studied, and the report will discuss possible upcoming solutions from hardware manufacturers. Intel's Larrabee and the new CausticOne from Caustic Graphics are some of the new initiatives to make real time ray tracing possible in the future.

- To what extent is real time ray tracing possible?

Finally, this area will be explored by investigating the mathematical foundation and fundamental algorithms for ray tracing. However, just investigating the fundamental algorithms will not be enough to fully understand the efforts required to do real time ray tracing. A ray tracing engine will therefore be made. It will be able to render different objects with shadows and image enhancing effects, and attempt to do this in real time. Additionally it will be possible to implement objects made in the graphics program Blender, to make it simple to render complex objects.

As described in the problem analysis, it will not be possible to make a real time ray tracing engine on a single CPU. To simulate the future, as technology develops further, our ray tracing engine will therefore be developed to be distributable amongst several computers. Additionally some optimisations will be implemented, such as utilising symmetric multiprocessing (SMP) instead of just running multiple processes. CPU instruction set optimisations will not be looked into beyond flags offered by state of the art compilers. Also there will not be any attempts to do real time ray tracing on a GPU or utilise General-Purpose computing on Graphics Processing Units (GPGPU) features of common GPUs. We will also analyse the complexity of our ray tracing engine, to establish if it is worth it from a computational standpoint. The complexity is one of the most important aspects, as the lower the complexity is, the more objects will be realistic, and more detailed images can be created.

By investigating these two main questions it will be possible to conclude whether the demand for real time ray tracing outweigh the efforts, and we will be able to discuss the future perspectives of real time ray tracing.

CHAPTER 3

Demand for realistic graphics

One of the aspects which is dealt with in this chapter, is whether there is a demand for real time ray tracing. The primary benefit of ray tracing is that the way light interacts with objects in the real world is achieved naturally as opposed to rasterisation. Ray tracing is therefore naturally more realistic than rasterisation - however, it is much slower. The following sections will try to describe the demand for this benefit of real time ray tracing by looking at the history of computer graphics and what hardware developers think of the future of real time ray tracing. Additionally a questionnaire will be conducted to understand the gamer′s demand. At last a summery will sum up on the findings and conclude whether there is a demand today.

3.1 History of 3D video games

A brief look at few remarkable 3D games throughout the history of computer games.

3.1.1 Maze War

In the first half of the 1970s, Maze War was created as a spare time project at NASA. Maze War is allegedly the first networked multiplayer 3D game. In the game, players moved in a 3D maze, where the other players were represented as eyes. When encountering another player, it was possible to shoot them. The angles in the maze were all fixed at 90 degrees, as was all movement, which enabled rendering the maze on the hardware available in 1974, where the network aspect was allegedly added [Museum, 2004].

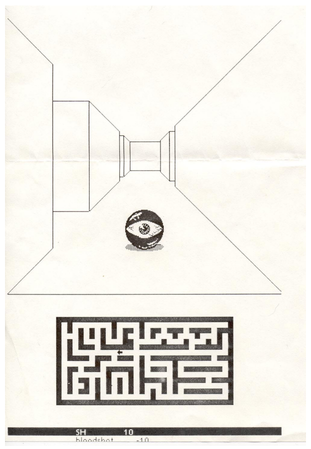


Figure 3.1: Maze War in action screen print from 1985 or 1986, by Dan Croghan [Museum, 2004].

Maze War is important because it serves as prior art for any multi user 3D environments today[1]. This may very well be the reason why nobody has claimed ownership of this ”invention”, which might have stifled the further innovation and development of 3D games and environments. From a technical perspective, Maze War was ahead of its time, though the rendering was very limited and technically not of any interest today. The 3D maze was represented using a 3D wireframe. Similar techniques were also used in other games later, such as the popular BattleZone of the 1980s.

[1]In patent law, prior art is a work which shows an invention and thus invalidates any later patent claims for said invention.

3.1.2 Alpha Waves

Alpha Waves, developed by Christophe de Dinechin and released in 1990, was one of the first games to feature animated flat shaded polygons. Previous 3D games only rendered wireframes, except the game Starglider 2, a flight simulator video game from 1988, which actually also served as inspiration for Christopher de Dinechin when developing Alpha Waves. Alpha Waves is noteworthy because it rendered fullscreen 3D, it did not limit the visible field depth, and drew the player in 3D as well.

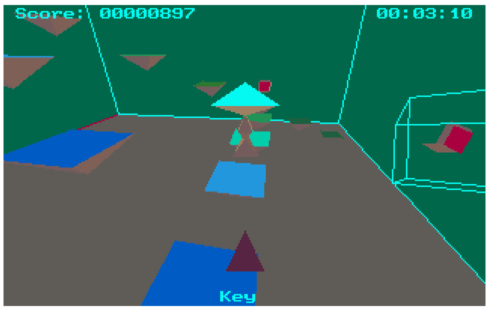


Figure 3.2: Alpha waves screenshot from de Dinechin [2007].

The gameplay in Alpha Waves was rather simple. The player bounces a simple 3D craft from platform to platform within a room/level. Each room contains a door that leads to the next room. The player must then reach this door, by bouncing from platform to platform, as each platform gets the player higher. If the player misses a platform the player must start from the floor again.

As Alpha Waves was developed on a 16-bit processor without floating point capabilities, and multiplication was considered expensive, the rendering engine used for Alpha Waves mainly used additions, and a table lookup was used for sine/cosine computations. Compared to other 3D games at the time, this made Alpha Waves able to show a great number of 3D objects on the screen [de Dinechin, 2007].

3.1.3 Doom

Doom, released as shareware in 1993, and with more than 10 million copies installed by 1995, it is without doubt one of the most popular games of its time. Along with Wolfenstein 3D, released a year earlier, Doom popularised the first person shooter genre on PCs.

Along with Wolfenstein 3D, Doom was also one of the first games to introduce textures, though characters in Doom and Wolfenstein 3D were still drawn using bitmaps and not 3D models. Doom also allowed walls to be non-perpendicular and rooms could have different heights, though no room could be on top of another room. From a graphical perspective, Doom was more realistic than any other game of its time, and depended greatly on these effects to generate a better overall gaming experience. For instance Doom supported different light levels, which was used to create the scary atmosphere in the game. This is just one example of how graphical effects came to be an important factor for games.

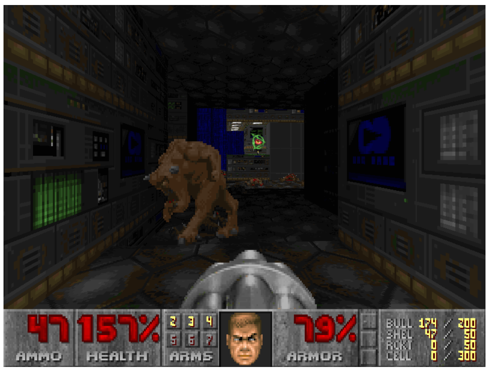


Figure 3.3: Light effects used to scare players [Wik, 2007].

3.1.4 Quake

Quake, released in 1996, was one of the first games to render the entire game in 3D. That means rooms within levels could be on top of each other and models were rendered as 3D instead of static 2D bitmaps. Quake was initially released only with software rendering. It later became one of the first games to support hardware acceleration for 3D rendering. First by VQuake which added support for graphic cards developed by Rendition Vérité using their proprietary API. Later, with the release of GLQuake, support for OpenGL was added, which was available on graphics card from 3dfx, who produced the first reasonably capable 3D graphic cards for consumers.

Throughout the history of computer games, graphics have developed from very primitive to near realistic 3D environments. What lays the ground for this development is not clear, but it must have some usage since the developers spent money on it. As seen with Doom, the graphics can have a very important impact on the gaming experience, as realistic games tend to make the gamer more engaged with the gameplay.

3.2 Questionnaire

This section describes the important aspects of formulating a questionnaire.

For game developers to use ray tracing for computer games, there must be a market. The market is twofold - the gamers must want to play ray traced games, and there must be hardware capable of doing ray tracing. The latter is, of course, also dependent on the former. To find out whether gamers are interested in buying new hardware to play new games with ray tracing, a questionnaire was sent out by e-mail to some of the students at Aalborg University.

For the results of a questionnaire to be usable, one must consider the focus group. Who would have an interest in this field? Having a variety of people who play computer games answer the questionnaire, would most likely give the most usable results.

The industry has an interest in getting new people to play computer games. It does not appear to be graphics that lure new gamers, as the Nintendo Wii has been praised for getting more people to play computer games [Casey, 2006]. Note that graphics are not one of the foci of the Wii - it sells itself mostly on the increased dynamic of the gameplay. It would still be important to see which aspects of computer games people are interested in when buying new games - it might be that the more experienced gamers value graphics more.

The demographic for this questionnaire was not so much based on age, gender or cultural background, as much as how often they play games.

Based on this, and the availability of respondents, it was decided to send the questionnaire to all second semester students at the Faculty of Engineering, Science and Medicine at Aalborg University. These students are between 18 and 40 years old, which is a group who, combined with the 10-18 year olds, can be expected to generally play more computer games than the rest of the population. An American study found that 35% of computer gamers are under 18 years old and 43% are 18-49 years old [Erickson, 2005]. As Denmark and USA are both western countries, it does not seem unreasonable to extrapolate that the picture is similar in Denmark.

3.2.1 Mechanics

It was decided to create a questionnaire, instead of interviewing people. Quantitative results would be more usable, as the interest was to see whether there generally was a market.

This questionnaire was distributed by e-mail. There were several reasons for selecting this approach. Amongst them was the fact that it was an easy way to reach all of the second semester students - hanging up a poster, or going from room to room would probably not have found so many respondents. Another reason was that it was easier for potential respondents to enter the questionnaire, when simply given a link. Had they received it on a piece of paper, they would have had to type in a long address - this is both tedious and error-prone.

When making a questionnaire, one must be careful not to pose coloured questions or give the respondents too much information about the intent of the questionnaire. By writing that the subject of the project was graphics, the respondents could gain an unconscious bias. To avoid this, the introduction merely mentioned computer games as the focus of the questionnaire.

Coloured questions often come in the form of ”How much do you agree with...” or having an answer scale of, for example, ”1 to 5 where 1 is very satisfied and 5 is not very satisfied”. These two extremes on the scale are not equal to each other. When one is ”very satisfied” the other should be ”very unsatisfied” [Hansen et al., 2008, p. 96].

To avoid this pitfall, the respondents were merely told that they should answer on a scale of 1 to 5, where 1 was the most important. This should ensure no uneven weight in the answers, as it was assumed the respondents would instinctively weigh 5 as equally not important.

The sequence of questions might also change the respondents' answers [Hansen et al., 2008, p. 65]. This is quite difficult to test for. It would be wise to consider if two questions near each other could influence the outcome.

Furthermore, in order for the questionnaire to be considered reliable, the response rate must be 70%, but it can be problematic to measure the response rate on a internet questionnaire [Hansen et al., 2008, p. 19]. This one was, as mentioned, e-mailed to the second semester students at the Faculty of Engineering, Science and Medicine at Aalborg University. The question is then, how to measure the response rate. Is it the number of people who answered the questionnaire, versus how many received the e-mail? Or versus how many read the e-mail. Or maybe versus how many opened the questionnaire. Even when knowing how many e-mail addresses the questionnaire was sent to, it is not entirely possible to know how many use that e-mail address, nor how many saw the e-mail before the questionnaire was closed for further answers. Neither was it possible to measure how many read the e-mail. The remaining possibility is to see how many opened the questionnaire, and how many answered it. This does not give an entirely accurate picture, but is the only real possibility in this situation. The questionnaire reached a response rate of 68.5%, and is thus valid in this regard.

The layout has a certain importance as well. It can be beneficial to have questions, which can be expected to be fun for the respondent, in the beginning of the questionnaire, to "lure them in", so to speak [Hansen et al., 2008, p. 64]. It is also reasonable to place similar questions together.

It is a risk that the respondents either misread or do not read the questions, but answer what they expect the question to be about. Limiting this can be difficult, but a few things can be done. Questions ought to be as short as possible, while still covering the necessities [Hansen et al., 2008, p.67]. The questions must also not be ambiguous, as this can render the answers invalid. Ambiguity can be hard to spot, as the writer often merely reads it the way it was intended. It can be limited by having outsiders, who have not been part of the writing, reading the questions and reporting their understanding. This was achieved by having our secondary supervisor and a gamer acquaintance look at it, and comment on unclear or superfluous questions.

3.3 Questionnaire analysis

Presentation of the questionnaire, its results and an analysis of these result.

3.3.1 The questionnaire

The purpose of the questionnaire was to see how important the "average gamer" rank graphics in games. To determine the demographics of the respondents, there was a few questions of how much and what they play. The intention was then to ask how important graphics were to the respondents. Both in general and in two specific situations: when buying and determining to keep playing a game. This could be used to see whether graphics were on average deemed important. Combined with the demographic information, it might be possible to find correlations. This might help determine the demographics of the ray tracing market.

Furthermore, it was asked how often the respondents bought new hardware and how expensive the games they bought were. Again to see if there was a correlation with how important graphics were rated.

To see the questionnaire see appendix A.1.

3.3.2 The questions

It was deemed necessary to ask which genre of games the respondents played, to ensure they did not only play games, where the graphic is of little importance (for example flash games on the Internet). The results were generally even between the genres - most genres had between 25% and 35% players. The big "winners" were action and strategy games, with 61% and 56% respectively. A reason for this might be that many games fall into several genres, where action and strategy often overlap with other genres. This can, to some extent, also affect arcade games, which had 45%. Action game developers often focus on graphics. Therefore it is an interesting group to ask, what importance they think graphics have.

The question of which platforms the respondents play computer games on, was mainly to document the demographics.

Income could be a factor for how many games are bought and how often hardware is upgraded, but this has not been factored. Most respondents can be expected to receive SU[2], but as it is possible to have a job on the side, it is not realistic to say anything on this subject.

3.3.3 Demographics

36% of respondents play computer games less than two hours per week. An almost equal number of respondents play 13-20 hours per week and more than 20 hours - 10.5% and 10.1% respectively (see figure 3.4).

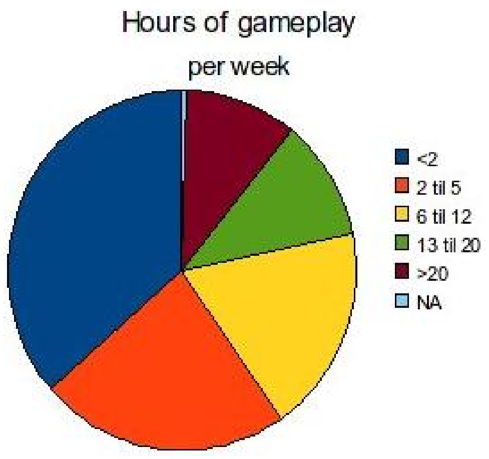


Figure 3.4: How many hours per week the respondents play.

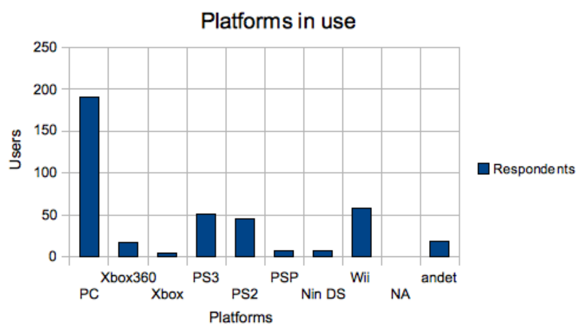
92% of the respondents play PC, 28% play on a Nintendo Wii, 25% on a PlayStation3 and 22% play the PlayStation2 (see figure 3.5a).

These three video game consoles are seventh generation consoles and can therefore be expected to compare to each other. The Nintendo Wii is the biggest seller[3] of the three, which is reflected on the results. Of all the consoles, it is the one being played by the most respondents. The Xbox 360 launched in 2005, making it the oldest of the three. This could partly explain why only 9% of respondents play it [Coola, 2005]. This does not, however, correspond with the large number of people playing the PlayStation2. It is a close second after the PlayStation3. The PlayStation2 is, in spite of its age, as popular as ever. Sony reports having sold more than 136 million of them since its launch in 2000, and having a library of nearly 1900 games in 2009 [Sony, 2009]. Because of its many sales, games are still made for PlayStation2. Games for the PlayStation3 often comes in a PlayStation2 version as well, which is often cheaper. With cheaper games, and new ones still coming out, people are still buying the PlayStation2. These two facts are assumably connected. As the Xbox, Nintendo DS and PSP only have a few percent of respondents each playing them, it seems the demographics of the respondents correspond to the sales and age of the consoles.

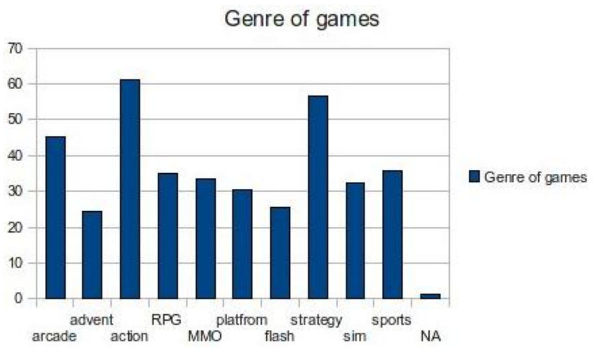
With regards to the genre of games played, 61% play action games, 57% play strategy games, and 45% play arcade games. All other game genres have between 25% and 35% of respondents playing them, which should give a broad range (see figure 3.5b).

[2]SU - Danish educational support.

[3]50.39 million Nintendo Wii consoles have, as of Mach 2009, been sold since the launch [Nintendo, 2009]versus 21.3 million PlayStation3 in December 2008 [Sony, 2008] and 28 million Xbox 360 in January 2009 [Microsoft, 2009].



(a) Platforms used by the respondents.



(b) Genres played by the respondents.

Figure 3.5: Platforms and genres played by the respondents.

92% of respondents play PC games. The old consoles are hardly in use, except for the PlayStation2, which 22% of the respondents still used. In relation to consoles, people often buy the newest to be able to play the newest games.

3.3.4 Importance of graphics

Graphics are important, in the sense that most respondents answered well in the middle of the spectrum, when asked how important graphics were, when buying a new game. This would indicate that they are not particularly important, but not unimportant either.

The importance of graphics, when buying a new game, is put on a scale of 1 to 5, where 1 is the most important. Both the average and the median were 3. It must be noted, though, that answering 3 may be an expression of graphics not being particularly important, or that the respondent has not made up his mind. It was not possible to add an "I do not know" option to the questionnaire, which would have been the best solution. It has been decided not to try and separate these two groups, partly as it is not possible, partly because the group who have not made up their mind assumably tend to give an average reply instead of not answering.

The games most often associated with breakthroughs in graphics are action games - they could possibly be the first ones to benefit from ray tracing. This begs the question: are action gamers more interested in buying new games because of new graphics? Figure 3.6 shows how the respondents, who answered they played action games, rate the importance of graphics when buying a new game, against how all of the respondents answered.

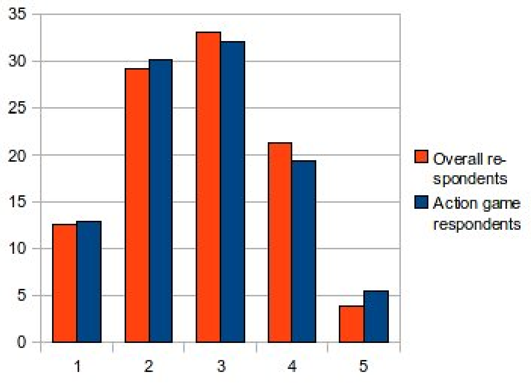


Figure 3.6: Comparison between how important action gamers think graphics are, when buying a game, versus what the whole group thinks.

There is no obvious difference in how the two groups answered. This might be explained partially by the action gamers making up 61% of all the repondents. It might also be that there is no particular difference - maybe people who play action games do not care more about graphics than people who do not.

When asked how satisfied they are with graphics today, 20% of the respondents replied that they are very satisfied. The average was 2.4 and the median 2. People may not be aware that graphics can be better or they may not care too much, but because they do not know it can be better, they feel it is good enough today.

There might be a difference in how satisfied the respondents are with the graphics of games today, depending on how many hours they play games. Figure 3.7 shows how people who play a number of hours have, on average, rated their satisfaction with graphics (where 1 is very satisfied and 5 is not at all).

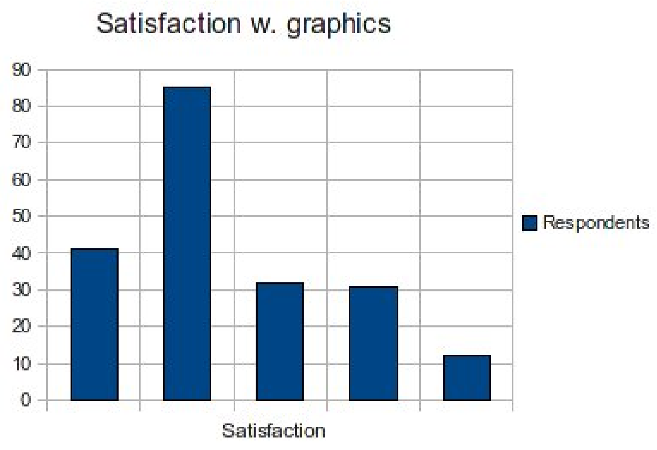


Figure 3.7: Comparison between how satisfied respondents on average are with graphics, depending on how many hours they play games.

The group who play 6 to 12 hours per week are on average the most satisfied with today's graphics, at 2.16, while the ones playing over 20 hours per week are the least satisfied, with 2.76 points on average. This is not a particularly big gap, but when breaking down the numbers, all groups have the most respondents answering 2. There were 21 people in the group that plays more than 20 hours per week. Of these, three people gave 1 as an answer, nine people gave 2 and four gave 5. That means about 20% think, today's graphics are not satisfying. There were 47 respondents, who played 2 to 5 hours per week - out of these, 22 gave a 2, and 12 gave 4. That is circa 25% who are mildly dissatisfied with recent graphics. If anything can be concluded from this, it must be that the respondents who game the most are the least satisfied. They may be more aware of the develpment of graphics, and thus be more knowledgable about the future possibilities. It could also be a coincidence - there is not enough respondents in each group to fully justify any correlations.

3.3.5 Hardware

When asked how many times they have upgraded their hardware in the last five years, 29% replied two times. The average is 2.9 times and the median is 3. Only 10% have not upgraded their hardware at all. This means that most people regularly upgrade their hardware. As ray tracing will likely need new hardware to run in real time, people will need to upgrade their hardware at some point. From these numbers we can conclude that most people would upgrade their hardware at some point, when the hardware developers release hardware capable of ray tracing. The price probably matters as well, and some people may wait until the new hardware is not so expensive anymore, but they will probably buy it when they need to upgrade their computer anyway.

25% of respondents play PlayStation3 games. The Playstation3 was released in March 2007 in Europe, as of the release of this report, it has only been two years [Europe, 2007]. It shows that some people are willing to buy new hardware to play new games. This could indicate that the same people would buy ray tracing hardware to play ray traced games.

There does not seem to be an obvious relation between how important the respondents think graphics are, when buying a game, and how expensive games they buy. This can be an indication of several things. One is that the respondents are not willing to pay more for good graphics. Another is that expensive games do not necessarily have better graphics than cheaper games, and it thus cannot be extracted whether these gamers are willing to pay more for good graphics.

3.3.6 Reliability

The webpage used for making and hosting the questionnaire measured how many times the questionnaire was opened. In the results of the questionnaire it is possible to see how many responses were gathered.

The questionnaire was opened 308 times, and answered 211 times. This gives a response rate of 68.5%.

As it is not quite 70%, it is strictly not entirely reliable. It is close, though, so the results can be used.

Who might have skipped answering? It could be that more people, who do not play many computer games, closed the questionnaire without answering, when learning it was about computer games. They might think they were not part of the focus group. Or it might just be the ones who opened to see what it was, and closed it because it was too long/they did not have time/it looked boring etc. It could also be that some respondents had opened the questionnaire and then closed it, and then answered it later. This cannot be read from the numbers though.

As the demographics were also deemed acceptable, it is concluded that the questionnaire is usable.

3.3.7 Conclusion

The questionnaire had an acceptable response rate and a demographic distributed as might be expected.

It was concluded that there was no particular wish for better graphics. They were neither important to the respondents when buying games, or when deciding whether or not to continue playing a game.

There was also no obvious correlation between the price range of games bought by people who claim to think graphics are important, nor between action gamers and what importance they put on graphics when buying a game. There was no clear-cut correlation between how many hours per week respondents played computer, and how satisfied they were with graphics.

In conclusion graphics are an integral part of a game - they need to be there. Respondents were generally satisfied with the graphics of today, but update their hardware often enough that ray tracing would be phased in, if graphic cards had in-built ray tracing abilities. When new, exciting games, working with ray tracing would come out, it would also be realistic that people would buy new hardware to play it.

3.4 Initiatives on ray tracing

This section describes which initiatives there are on the market of ray traced computer graphics.

The current interest in ray tracing is showing itself by certain initiatives by different companies and individuals. One of the big players on the ray tracing scene is Intel, which has created a QUAKE War ray tracing engine, and is currently developing a new highend GPU codenamed Larrabee. Their QUAKE Wars ray tracing experiment was primarily to show case current ray tracing capabilities on multi core systems, and spark an interest in the area. The engine was first presented on a 16 core system clocked at 2.93 GHz (4 Tigerton quadcore CPUs) at Research@Intel day in 2008. This setup was capable of generating 15 to 20 fps at a resolution of 720p.

The following is a list of features the engine was capable of, quoted from the project site [Pohl, 2008]:

- Water with reflection and refractions
- Realistic glass shaders
- Accurate shadows
- Camera portal effects
- Displaying the MegaTexture
- Collision detection using ray tracing

The setup was later upgraded at the Intel Developer forum in August 2008 to a 24 core system clocked at 2.66 GHz (4 Dunnington six-core CPUs). This setup provided 20 - 35 fps at the same resolution. The upgrade showed that the performance increased linearly by adding 8 cores [Pohl, 2008].

Intels Larrabee GPU is intended to compete against other highend GPU manufacturers, such as nVidia and AMD/ATI. It will therefore be focused on rasterisation, although it is also considered a general purpose GPU (GPGPU). Because of the

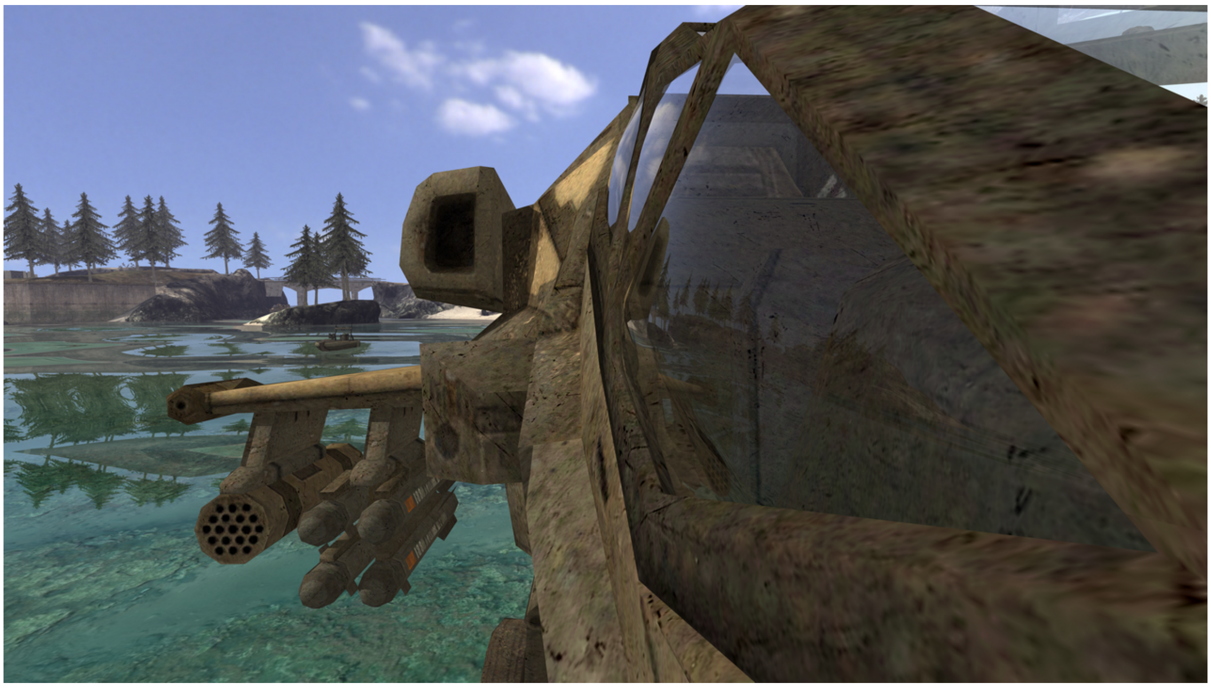


Figure 3.8: QUAKE WARS ray traced

Larrabee's design, which is a hybrid of a GPU and a CPU, it could theoretically run standard applications, just like a CPU. This means that the Larrabee could also be used for ray tracing, in other words, this allows Larrabee to act as specialised hardware for ray tracing [Intel, 2008].

Intel is not the only company interested in ray tracing. Hardware enthusiast sites, such as Tom's Hardware, claim that ATI already fully supports ray tracing capabilities, as far back as their Radeon 2900XT cards, and that their newer Radeon 4800 series is very capable when it comes to ray tracing. This is hard to confirm, since AMD/ATI never directly mention ray tracing. If the claim is true, then the ray tracing capabilities of today are already much greater than previously anticipated [Valich, 2008].

nVidia is also investigating ray tracing, but, like ATI, there is not much official word on the subject. There are several articles from various hardware sites about nVidia's ray tracing demo, where they were able to render an automobile at 1080p resolution at 30 fps. It was done by using a system with 4 Quadro GPUs, each with 1 GB memory. To do this they used the CUDA architecture, designed to allow access to GPGPU features of their GPU's [Hardwidge, 2008].

As of March 10, the company Caustic Graphics claims to have created a ray tracing accelerator card, the CausticOne, which allows ray tracing to be done 20 times faster than alternative solutions. It traces the scene and all the bounces first, then sorts all the pixel data into a form the GPU recognises and is efficient at calculating, and then passes on shader calculations to the GPU. Exactly how the software and hardware does this is still somewhat shrouded in mystery. Currently, the hardware capabilities of the CausticOnes are limited and are primarily intended to be a demonstration board for software developers. Their upcoming successor to the CausticOne, the aptly named CausticTwo, is expected to perform up to 14 times better than its predecessor, and will contain custom made processors. Caustic have provided a demonstration of what their card can do. The resolution was 640x480, there were approximately 5 million triangles and the antialiasing was set to four samples. The demonstration managed to get 3 to 5 FPS. They claim that this is with a much higher number of bounces than previous ray tracing attempts, effectively giving reflections within reflections within reflections. As the demonstration did not use a GPU, the CPU was utilised for calculating shading [Shrout, 2009].

If these different hardware manufacturers are successful in their ray tracing attempts, then ray tracing capable hardware is very close to becoming readily available. Unfortunately, this also requires software developers to use it. The benefits of our hardware capabilities will therefore not be readily used, because many game developers will likely prefer to stick with current methods of rendering, until most of their customer base have acquired ray tracing capable hardware.

3.5 Summary

This section sums up on whether there is a demand for real time ray tracing within the areas that have been assessed.

In the preceding sections the demand for real time ray tracing has been described. Looking at the history one can see that graphics have become more realistic over the years. From the seventies and throughout the nineties the games became more and more realistic, and the possibility of doing ray tracing in real time could continue this development.

The questionnaire showed that people did not rank graphics in computer games as that much more important than other aspects of the gameplay - but it did show that it was one of the things that just needed to be there. There was no correlation between how expensive games people bought amd how important they thought graphics were. There was also no obvious correlation between how satisfied respondents were with graphics and how many hours they played computer per week. However, the people who game the most, seem to be the least satisfied with today's graphics. They might be interested in both better graphics, and new games running ray tracing. As hardware is routinely updated by the average user it would be realistic to phase in ray tracing by introducing new hardware with ray tracing capabilities, along with new ray traced games.

Today there is no other readily available ray tracing capable hardware than the CPU, and this creates a limitation on how well a ray traced program will run. To make ray tracing achievable in a computer game, it has to be able to run on hardware available to the average gamer. Intel and Caustic Graphics are both working on new hardware that allegedly should be able to do real time ray tracing in the future. That Intel and Caustic Graphics are prepared to spend money on this area does show that the findings from the questionnaire might not be completely realistic: Intel and Caustic Graphics must think there is, or will be, a market for ray tracing.

CHAPTER 4

Ray tracing

The next three chapters will discuss the efforts needed to do real time ray tracing today. There are two different aspects to this question, and this chapter will describe some of the theories necessary.

The first section is a general review of the principles behind ray tracing and the following sections are elaborations of the theories described.

The first elaborating section is about the camera. It will describe how the camera works and how rotation and zoom is performed.

The following section will describe how intersection with different objects is calculated, as this is necessary for computing whether the object can be seen from the camera, and whether it is within the focus of a light source.

When an intersection is detected, the corresponding pixel must be shaded. There are different aspects to consider when shading in a realistic manner. Section 4.4 will describe these and present a mathematical simplification of how light acts in reality.

The chapter will end with a section describing the principles of light for different types of light sources, and how to represent these in a ray tracing engine.

4.1 Ray tracing in general

This section is a general introduction to the theories of ray tracing.

Ray tracing is, as its name suggests, about tracing rays in a 3D space as they hit objects and bounce off. The basics are much the same as in real life, where photons are emitted from a light source, bounces off an object and are then interpreted by the eye on impact. However, a light source emits millions of photons per second, all of which reflect and transmit when hitting objects, resulting in trillions of photons being in the air at the same time. Such a situation would require an enormous amount of computations if it was to be simulated on a computer. It would also be unnecessary, as most of the rays would never reach our eyes.

Instead, ray tracing works backwards. Instead of tracing the photons from their point of origin, they are traced from their final destination. Because of the nature of reflection and transmission, it is possible to begin at the end, and then predict whether a ray would hit it from another given start point. Figure 4.1 illustrates this principle in an environment. Notice that it is checked whether the eye would be able to see the light, not whether the light is able to reach the eye.

When creating an image using ray tracing, a ray is traced for each pixel on the screen. This will simulate the photons reaching every point on the eye. But instead of an eye, a camera consisting of a camera point and a spreading is used. (See figure 4.2)

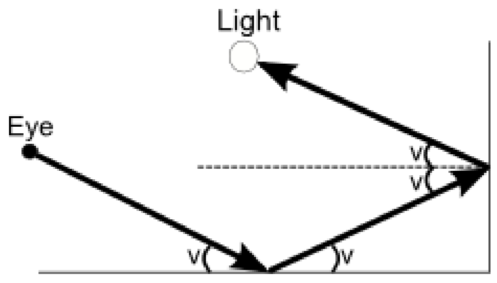


Figure 4.1: In and outcoming angles in an reflective environment. v is the incoming angle.

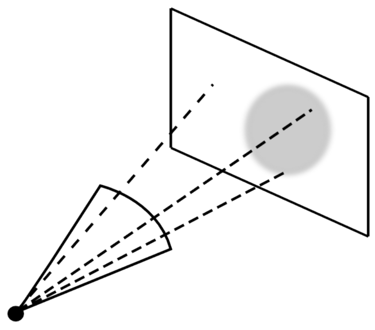


Figure 4.2: Ray spreading from camera point through a screen.

4.1.1 Viewing objects

An object is viewed if the emitted camera ray intersects with the object. The ray consists of a position vector and a direction vector. By using geometric calculations it is possible to calculate whether a line in the direction of the camera ray intersects with an object[1]. A scene might have several objects, and there might be more than one object in the line of intersection. However, only the first intersecting object is interesting, i.e. the object closest to the ray's origin.

When the first intersecting object has been found, this object must be shaded, i.e. the colour to be assigned to the pixel must be computed. To shade an object, the amount of light that hits the object in the intersection point (which is the point where the camera ray intersects the object) must be determined. To compute the amount of light that hits a point, the light sources that illuminate this point must be found. This is done by tracing a shadow ray from the point to each light source. If this shadow ray is intersected by another object before it reaches the light source, the light from the light source does not reach the point that is being shaded, thus the light source does not contribute light to the point.

Depending on how many light sources contributing light to the intersection point on the object being shaded, the point may appear to be in shadow. Figure 4.3 illustrates an environment where a shadow ray intersects another object so that some of the points on the sphere being shaded appears to be in shadow.

The shadow in figure 4.3 is a hard shadow, which is partially because the light is

[1]See chapter 4.3 for intersection calculations

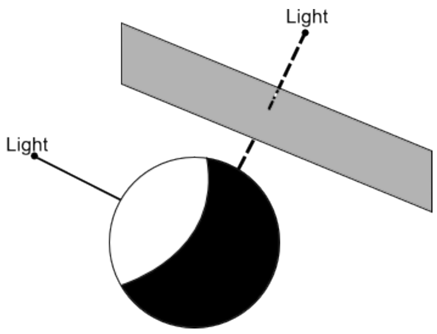


Figure 4.3: Environment with two lights and two objects.

represented by a dot. If the light was represented by a sphere, a so called *soft shadow* would appear. Representing a light source by a sphere would also be more realistic, as light in reality is not emitted from an infinitely small point. Representing a light source as a sphere would require more shadow rays to be traced for each light source, and enable a light source to partially illuminate a point, in case some of the shadow rays are blocked. This will create gradient shadows (see figure 4.4). Additionally soft shadows occur when light is reflected and transmitted within the environment and then eventually hitting an otherwise shadowed area, thus providing a little light.

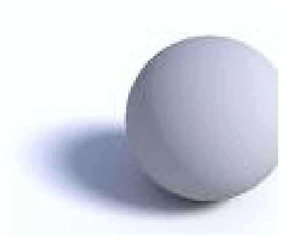


Figure 4.4: Example of a soft shadow. [wikipedia.org, 2009]

4.1.2 Reflection and transmission

As seen in figure 4.1 light can be reflected off an object. This implies that it is possible to see objects that are not directly in the line of the camera ray, just as in real life. Whenever a camera ray hits an object, and the object has a reflective or transmissive surface, it is reflected or transmitted, and some of the light that hits the next object is transferred to this object. This is what occurs in still water where, for example, surrounding trees are mirrored in the surface. A new ray is therefore created in the intersection point, with a direction based on the the incoming angle. Even though the intensity of the light fades off when it travels through a media (also air), the ray could bounce many times. It therefore becomes necessary to set a depth of field, which limits how many times a ray can bounce, or set a lower bound on the light value required to be passed on. For example, the

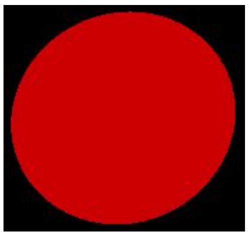


Figure 4.5: Principles of ambient shading.

reflection and transmission can stop after five intersections, or when the light is below 10% in intensity. To allow this approximation, an ambient shading is applied. This is a constant light factor, added at every point, so that every point receives some light, even though it is not lit by a light ray. If the depth of field was set to 10%, an ambient shading could add the remaining 10%. Figure 4.5 illustrates an object shaded with ambient shading.

4.2 Camera

This section briefly describes some of the different types of cameras and how the pin hole camera is implemented in a program. The technique anti-aliasing, which is used to make more a detailed picture, is also described.

The camera is a central part in ray tracing as it is the origin of the rays. A simple camera with no anti-aliasing capabilities sends one ray per pixel. This ray determines the colour the pixel will take. A ray consists of an origin and a direction, which are generated using the camera. Many different types of cameras are available, such as the orthographic camera, the fisheye camera and the pin hole camera [Suffern, 2007].

4.2.1 The pin hole camera

The pin hole camera is used for creating realistic perspective in an image. For each pixel on the image, the camera generates one ray from its position. To calculate the direction of a ray, the camera needs to know where it is pointing. In TheMatrixDistributed, three angles are used to identify the orientation of the camera. It is also possible to use a vector, but that would limit the camera's rotational abilities to only two axes. Next the camera creates a virtual plane in the 3D environment, so that the orientation of the camera is right through the center of the plane. This plane will represent the 2D screen on the computer and is also known as the view plane (see figure 4.6).

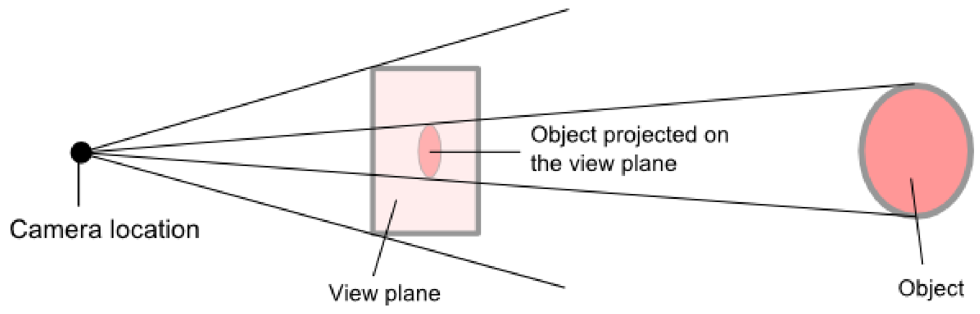


Figure 4.6: Elements of a pin hole camera.

The direction of the ray will be computed as a vector from the position of the camera to the 3D point on the view plane representing the pixel on the screen. As a result, what the camera sees is projected onto the view plane, which is what the computer displays. What the camera looks at depends on where the view plane is compared to the camera location. Thus the camera can rotate to look at something different by simply changing the orientation. The camera can also move around in the scene by changing the base point. Finally, everything hit by the rays sent from the camera will be displayed on the screen.

4.2.2 The orthographic camera

The orthographic camera is a very simple camera, as it has no perspective. Like the pin hole camera, it emits one ray for each pixel on the screen. But unlike the pin hole camera, all the rays of the orthographic camera are parallel to each other, pointing in the direction of the camera. Thus, the origin of the rays are not the same for each ray. One can perceive an orthographic camera as a pin hole camera without a position. Instead, the rays are emitted from the view plane itself in the position of the pixel they represent, as seen in figure 4.7.

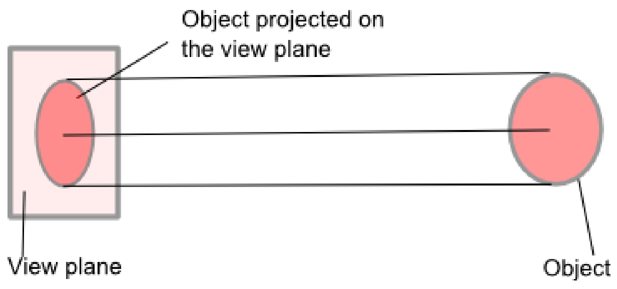


Figure 4.7: Elements of an orthographic camera.

Because the rays are parallel, the distance from the camera to the objects does not effect the result, and unlike the pin hole camera there is no horizon and no perspective in the image.

4.2.3 The fisheye camera

The fisheye camera is a nonlinear projection. Like in the pin hole camera, rays are emitted from the camera position. However, the view plane is no longer a plane, but a sphere surrounding the camera position. The view plane is then wrapped around that sphere, where the direction of the actual ray sent is pointing to the wrapped view plane position representing the pixel rendered (See figure 4.8).

The result of the fisheye camera can be an image of how an animal looks at the world with one eye on each side of its head. There are many ways of doing nonlinear projection and they can give a wide range of results with different perspectives. Nonlinear projection do not react to zoom and rotation in the same ways as the pinhole camera. This study will not go into details with nonlinear projection, as this is a special effect of ray tracing and beyond the scope of this report. For the same reason only the pin hole camera has been implemented in TheMatrixDistributed and the following sections will only talk about the pin hole camera.

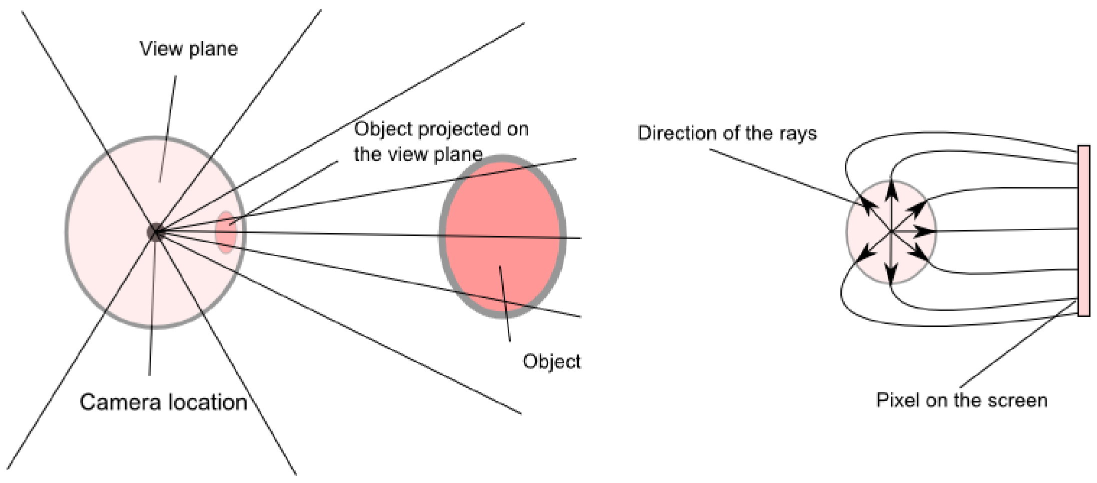


Figure 4.8: Elements of a fisheye camera.

4.2.4 3D rotation

To create the view plane in a 3D environment, 3D rotation is needed in order to allow the camera to point in all directions. Normally spherical coordinates are used to calculate a vector from two angles. In this case there are three angles, as the camera can also rotate around its own axes. This is done with Euler's matrix multiplications, which takes three angles to rotate a vector [Tomas Akenine-Möller, 2002].

$$\begin{aligned} V_x &= \begin{bmatrix} cos(\gamma)cos(\beta) - sin(\gamma)sin(\alpha)sin(\beta) \\ sin(\gamma)cos(\beta) + cos(\gamma)sin(\alpha)sin(\beta) \\ -cos(\alpha)sin(\beta) \end{bmatrix} \\ V_y &= \begin{bmatrix} -sin(\gamma)cos(\alpha) \\ cos(\gamma)cos(\alpha) \\ sin(\alpha) \end{bmatrix} \\ V_z &= \begin{bmatrix} cos(\gamma)sin(\gamma) + sin(\gamma)sin(\alpha)cos(\beta) \\ sin(\gamma)sin(\beta) - cos(\gamma)sin(\alpha)cos(\beta) \\ cos(\alpha)cos(\beta) \end{bmatrix} \end{aligned} \tag{4.1}$$

With this equation, the 3D point on the 3D view plane can be calculated for each pixel. In TheMatrixDistributed the `MatrixCamera` makes use of equation (4.1) to calculate the (x, y, z) direction of each ray.

However, this is very slow, as many calculations are needed. To reduce the number of computations, the `VectorCamera` was implemented. Instead of computing a direction for every pixel using the matrix above, this camera computes a coordinate system with the vectors **(u,v,w)** axis aligned in the direction of the camera. Now the only computations necessary for each pixel is to multiply the pixel position (x, y) on the screen with the axis aligned coordinate system as in (4.2) where c is the distance to the view plane.

$$\mathbf{V} = \mathbf{u}Pixel_x + \mathbf{v}Pixel_y + \mathbf{w}c \tag{4.2}$$

4.2.5 Zoom factor

If the camera is moved further away from an object, the object is going to appear smaller on the screen. This is because less rays from the camera will hit the object, as the rays are spread out over a greater range. However, this is not the same as zooming. Zoom determines how much the rays spread over distance, but unlike simply moving the camera, the zoom will affect how objects are viewed in perspective. Images taken with a telephoto lens at great distance will appear flatter and without perspective compared to images taken at close range. Zoom can be calculated by multiplying the zoom factor with the pixel and multiplying **w** with the constant c, which is the distance to the view plane (see equation 4.3) [Suffern, 2007].

$$\mathbf{V} = \mathbf{u}(Pixel_x Zoom) + \mathbf{v}(Pixel_y Zoom) + \mathbf{w}c \tag{4.3}$$

Equation 4.3 changes the size of the view plane in order to create zoom. However, to optimise the needed calculations the `VectorCamera` uses another method for calculating zoom. As all the rays have to pass through the virtual plane, it is the distance from the camera position to the view plane that determines the spread of the rays. The camera makes use of this to create zoom by moving the view plane further back to increase zoom, and thus equation 4.3 can be rewritten to equation 4.4.

$$\mathbf{V} = \mathbf{u}Pixel_x + \mathbf{v}Pixel_y + \mathbf{w}Zoom \tag{4.4}$$

As **w** and $Zoom$ are constant for each rendered frame, equation (4.4) can be further reduced to equation 4.5 where k is computed along with **u**, **v** and **w** when the camera direction or zoom changes. Equation 4.5 therefore needs less calculations for each pixel.

$$\mathbf{V} = \mathbf{u}Pixel_x + \mathbf{v}Pixel_y + k \tag{4.5}$$

4.2.6 Anti-aliasing

When multiple objects overlap each other or when objects are textured, the pixels can appear jagged. This effect is called aliasing and is a result of only one ray being sent from the camera and returning a single colour equal to what the colour should be in the center of that pixel.

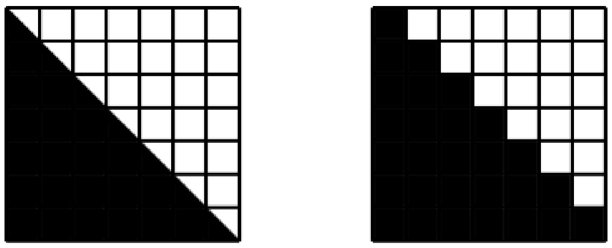


Figure 4.9: The left picture shows what the image should look like. The right picture shows how the pixel will be coloured.

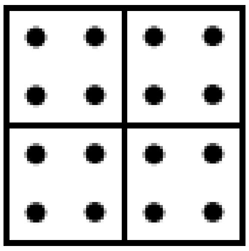
This occurs because the ray is considered a point. while it actually represents the colour of a square. Therefore it does matter where in the pixel the ray is sent, as it do not necessary return the same colour.

To avoid this jagged effect, anti-aliasing is used. Instead of just sending one ray, multiple rays are sent toward different positions in the pixel and then return the average colour of all the rays. To do this (4.5) can be expanded to (4.6) which sends rays from different positions in the pixel.

$$\mathbf{V} = (\mathbf{u}(Pixel_x + AA_x)) + (\mathbf{v}(Pixel_y + AA_y)) + k \tag{4.6}$$

Where AA_x and AA_y is the position in the pixel the ray is sent to. Depending on how many rays are needed to be sent from every pixel, AA_x and AA_y should be set to spread the rays to give the best average. Below figure 4.10a shows how the rays are spread with four rays per pixel.

The result of anti-aliasing is more smooth pixels and a more realistic image (see figure 4.10b).


(a) Rays in the pixel

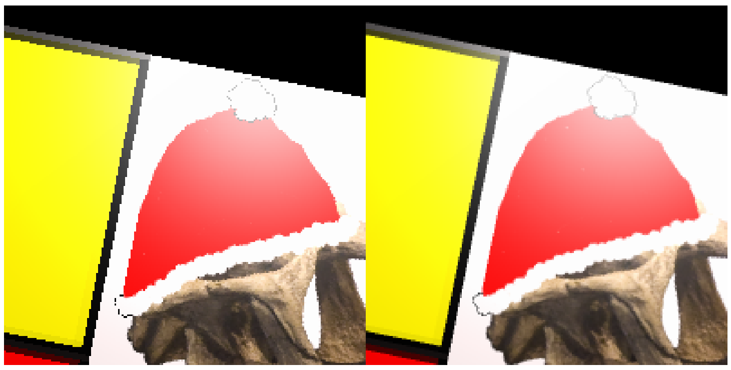

(b) Image without and with antialiasing in TheMatrixDistributed

Figure 4.10: (a) Rays in the pixels when using antialiasing. (b)The left image shows an image rendered with no anti-aliasing in TheMatrixDistributed. The right image is anti-aliased with four rays per pixel.

There is no limit to how many rays that can be sent - more rays give a better average and hence a more accurate result. The downside of this method is that it is very slow, though. That is because extra rays have to be sent from every pixel - thus more calculations will have to be made. It has been suggested to approximate anti-aliasing by sampling each pixel with its neighbour pixel using a filtering technique [Suffern, 2007]. Filtering works by weighing the result of the rays, based on the range from the ray to the center of the pixel. This technique should give a better result than rendering without anti-aliasing, but could be optimised to weighing the result of the the rays already computed in the neighbour pixel. Thus no extra rays will have to be sent, resulting in much faster calculations compared to anti-aliasing. However, this technique removes the advantage of being able to render each pixel individually, as all the neighbour pixels will have to be rendered as well, before it is possible to anti-alias the pixel.

Another way of optimising anti-aliasing is to detect if a pixel's neighbour pixels intersect with the same object. As aliasing mostly occur on the edge of objects, anti-aliasing could be limited to pixels whose neighbour pixels do not intersect with the same object. This would limit the number of pixels needing to be anti-aliased. However, like the filtering method above, this will also prevent individual rendering of pixels, as the object of intersection will have to be compared with the neighbour pixels to see if it is the same. This method will not work on textured surfaces as the objects of intersection is the same but the texture itself can be aliased. This problem could be solved by using bilinear filtering on the texture instead, as this is much faster than anti-aliasing all the textured surfaces.

4.3 Primitives

This section describes how intersection with different primitives are calculated.

A ray traced scene consists of a list of simple mathematically described objects like the plane, sphere and triangle. The ray tracer requires a formula for calculating the intersection point, normal and distance in order to draw it. The reason for using simple figures is that it is very difficult to create complex mathematical objects, and complex objects can be created from multiple simple objects. When there are multiple objects in a scene, they can exist next to, in front of or behind each other, depending on the camera angle and position. To figure out which objects to draw, and which to ignore, a distance needs to be calculated. This is the distance from the intersection point to the camera position. The object closest to the camera can then be displayed, as this must naturally be the object in front.

4.3.1 Sphere

A sphere is one of the primitives in TheMatrixDistributed. It consists of a position vector to the center, **C**, and a radius r. From these two parameters a discriminant will be calculated in order to check if the camera ray hits the sphere. If the ray hits the sphere, a normal vector at the intersection point and a distance from the intersection point to the camera will be calculated. The following equations have been deduced from the spherical equation 4.7.

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2 \tag{4.7}$$

Instead of calculating if the ray intersects with the sphere in its current location, the sphere is moved to the center of the coordinate system $(0,0,0)$ and the ray is moved equally. This will reduce the calculations needed.

$$NewRayPosition = \mathbf{C} - \mathbf{O} \tag{4.8}$$

O and **D** are the origin and direction of the original ray respectively. Next the dot product of the direction of the ray and the NewRayPosition can be calculated.

$$RayDotProduct = \mathbf{D} \cdot NewRayPosition \tag{4.9}$$

Using equation 4.8 and 4.9 the discriminant can be calculated.

$$d = RayDotProduct^2 - |NewRayPosition|^2 + r^2 \tag{4.10}$$

If the discriminant d is less than 0, the ray does not intersect with the sphere and no further calculations are needed. If d is 0 there is one intersection, and if d is greater than 0 there are two intersections. Given that there is an intersection, a distance from the origin of the ray to the intersection point can be determined, using (4.11). Note that if there are two intersections the second one could be found using $+$ instead of $-$ in (4.11) - however, we are only interested in the closest intersection.

$$distance = RayDotProduct - \sqrt{d} \tag{4.11}$$

When the distance is known, the intersection point can be calculated with equation 4.12. Note that it is now the original position of the ray which is used.

$$intersectionPoint = \mathbf{O} + (\mathbf{D} \cdot distance) \tag{4.12}$$

The normal vector, **N**, at a sphere has the same direction as a vector from the center of the sphere to the intersection point on the spheres surface. Therefore **N** can be calculated as in (4.13)

$$\mathbf{N} = intersectionPoint - \mathbf{C} \tag{4.13}$$

4.3.2 Plane

Like the sphere, the plane is one of the simple objects in TheMatrixDistributed. To draw a plane, a normal vector, **N**, and a point, **P**, is required. A ray will always hit the plane unless the ray is parallel to the plane - the dot product of the planes normal vector and the ray direction will then be 0. The intersection point on a plane can be found by first calculating the distance from the ray to where it intersects with the plane [Pedersen, 2006].

$$Distance = \frac{(\mathbf{O} - \mathbf{P}) \cdot \mathbf{N}}{\mathbf{D} \cdot \mathbf{N}} \tag{4.14}$$

Equation 4.14 calculates the distance from the rays origin **O** to where it intersect with the plane. **D** is the direction of the ray. If the distance is less than 0 the intersection happened behind the camera and the intersection is discarded. The intersection point can now be calculated from the distance and the ray as in equation 4.15.

$$IntersectionPoint = \mathbf{O} + (\mathbf{D} \cdot Distance) \tag{4.15}$$

The normal vector of a plane will always be the same regardless of the intersection point and no further calculations are needed.

4.3.3 Triangles

Triangles are one of the primary primitives in the computer graphic industry. 3D models, such as characters, created for games or movies are often the product of several triangle "meshes". The idea is to use many small triangles, to the point where you cannot tell that the object is created by triangles. This requires very efficient triangle intersection tests as such 3D models can contain many thousands of polygons[2].

A fast and efficient technique for calculating intersection involves calculating the triangle's barycentric coordinates. First, however, it is prudent to perform a distance test, in order to ensure that checking for an intersection is necessary.

$$DistanceA = -\frac{(\mathbf{O} - \mathbf{A}) \cdot \mathbf{N}}{\mathbf{D} \cdot \mathbf{N}} \tag{4.16}$$

Equation 4.16 is the formula for calculating the distance along the ray to the plane embedding the triangle. **N** is the normal to the plane which can be created with the triangle as a base. In the case of the TheMatrixDistributed, **N** is precalculated when the triangle is created. **A** is a vertex in the triangle **ABC**, and **O** and **D** are the origin and direction respectively of the ray that is being tested.

The dot product of **D** and **N** is computed separately from the rest of the equation, and checked to ensure that it is not zero. If this is the case then the ray will be parallel with the plane embedding the triangle, and no intersection would occur.

The distance can then be checked against other objects that have been tested in the scene - that is, computation can be aborted, if the previous distance is less than the computed distance. Once the distance test has been passed, the algorithm must check to see if the ray intersects within the boundaries of the triangle. First the intersection point must be calculated.

$$intersection_{point} = \mathbf{O} + Distance \cdot \mathbf{D} \tag{4.17}$$

Once the intersection point has been established, the barycentric coordinates can then be calculated by solving the series of equations in equation 4.18. The barycentric coordinates of the triangle are the values α, β and γ. If $intersection_{point}$ is inside of the triangle, then they will always be positive numbers, and their sum will be precisely 1.

[2]Normally an object with many edges, but in this connection polygons will refer to objects with only three edges.

It is sufficient to calculate β and γ and ensure that they are within the intervals (4.20), (4.21) and (4.22) [Wald, 2004].

$$\begin{aligned} intersection_{point} &= \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C} && (4.18)\\ 1 &= \alpha + \beta + \gamma && (4.19)\\ \beta &\geq 0 && (4.20)\\ \gamma &\geq 0 && (4.21)\\ \beta + \gamma &\leq 1 && (4.22) \end{aligned}$$

To further optimise the algorithm, TheMatrixDistributed takes advantage of the fact that the barycentric coordinates do not change if they and the intersection point are projected into another plane, as long as it is not orthogonal with the original plane. It therefore project the triangle into one of the 2D coordinate planes (XY, XZ or YZ). To maintain numerical stability, the triangle is projected into the plane in which it has the maximum projected area. This corresponds to the dimension in which the normal has its largest absolute value. Computations can thereafter be more efficiently performed in 2D rather than 3D. Equation 4.23 displays equation 4.18 projected into a 2D plane. Substituting $\alpha = 1 - \beta - \gamma$ into (4.19) and rearranging it gives (4.24). This can then be solved using the Horner Scheme and yields equations 4.25 and 4.26, where $b = \mathbf{C}' - \mathbf{A}'$, $c = \mathbf{B}' - \mathbf{A}'$ and $h = intersection'_{point} - \mathbf{A}'$. The x and y denotes the two coordinate axis after projection [Wald, 2004].

$$\begin{aligned} intersection'_{point} &= \alpha\mathbf{A}' + \beta\mathbf{B}' + \gamma\mathbf{C}' && (4.23)\\ \beta(\mathbf{B}' - \mathbf{A}') + \gamma(\mathbf{C}' - \mathbf{A}') &= intersection'_{point} - \mathbf{A}' && (4.24)\\ \beta &= \frac{b_x h_y - b_y h_x}{b_x c_y - b_y c_x} && (4.25)\\ \gamma &= \frac{h_x c_y - h_y c_x}{b_x c_y - b_y c_x} && (4.26) \end{aligned}$$

Texture mapping

As an added feature TheMatrixDistributed can create texture mapping for triangles. The triangle will be the only primitive which will be texture mapped, due to its dominant use in the computer graphics industry.

It is necessary first to map the triangle with a so-called uv-map. This is done to properly associate texture pixels with the triangle coordinates. The triangle itself must be uv-mapped before this is possible, which means that each vertex has a uv-coordinate associated to it. The uv-coordinate for the intersection point of the ray can then easily be calculated by reusing the barycentric coordinates that were calculated when computing the intersection point.

The uv-coordinates are calculated from the parametric equation describing the 2D plane the triangle has been projected onto. This is displayed in equation 4.27.

$$\begin{aligned}
\begin{pmatrix} u \\ v \end{pmatrix} &= \begin{pmatrix} A_x \\ A_y \end{pmatrix} + \beta \begin{pmatrix} B_x - A_x \\ B_y - A_y \end{pmatrix} + \gamma \begin{pmatrix} C_x - A_x \\ C_y - A_y \end{pmatrix} && (4.27) \\
Texel_x &= Image_{width} \cdot u && (4.28) \\
Texel_y &= Image_{height} \cdot v && (4.29)
\end{aligned}$$

With the uv-coordinates generated for the intersection point we can calculate the texel, which is the term used for a pixel on a texture. Equations 4.28 and 4.29 show the simple multiplications performed to calculate the texel.

Bilinear filtering

When calculating the colour on a textured surface, uv-coordinates represent the point on a texture image from which the colour is from. However, as the uv-coordinate is a point and a texture image consist of squared pixels, the uv-point often ends up being right between two pixels, resulting in the texture being aliased. A cheap way to fix this problem is to use bilinear interpolation. Bilinear interpolation works by first calculating the linear interpolation on one axis and then on the other. Linear interpolation approximates the value of a function in a given point between two known values. Bilinear interpolation is similar, but uses four points to approximate two values on the first axis (equation 4.30 and 4.31) which are then used to approximate the final value on the second axis, equation 4.32. This can be used to approximate the colour of a point in between four pixels, by using the known colour of the four closest pixels from the texture image. Bilinear interpolation is used on the three colours red, green and blue individually to approximate the colour in the point given by the uv-coordinate. Bilinear starts with two approximations on the first axis:

$$c_a = c_0 + \frac{c_1 - y_0}{u_1 - u_0} \cdot (u - u_0) \tag{4.30}$$

$$c_b = c_2 + \frac{c_3 - y_2}{u_3 - u_2} \cdot (u - u_2) \tag{4.31}$$

$$c = c_a + \frac{c_b - y_a}{v_2 - v_0} \cdot (v - v_0) \tag{4.32}$$

The c values are the colour values at the associated uv-coordinates, subscript denoting which values belong to which. The order and location of the data points can be seen in figure 4.11a

If the texture is square, several reductions can be introduced based on the following relations:

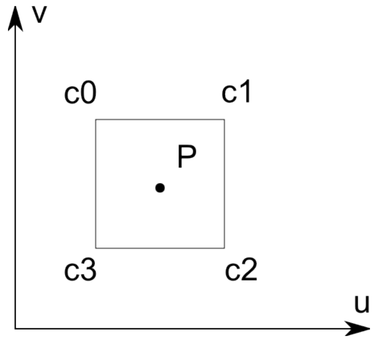
$$
\begin{aligned}
u_0 &= u_2 \\
u_1 &= u_3 \\
v_0 &= v_1 \\
v_2 &= v_3 \\
v_3 - v_0 = u_3 - u_0 &= w \\
AxisU &= \frac{u - u_0}{w} \\
AxisV &= \frac{v - v_0}{w}
\end{aligned}
$$

With this in mind, the previous interpolation equations can be reduced to the following:

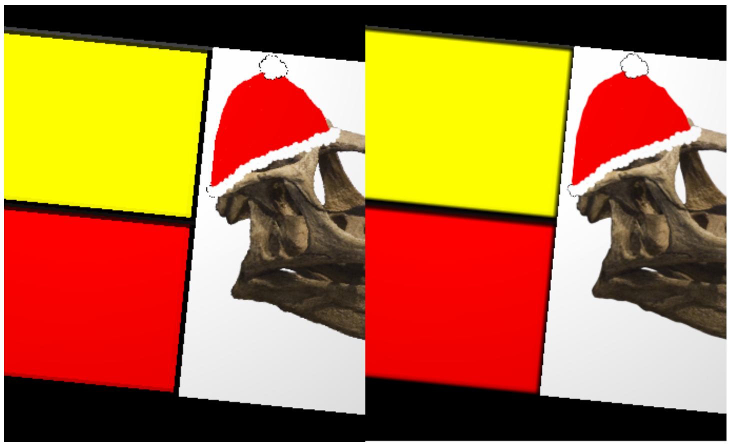
$$c_a = c_0 + (c_1 - c_0)AxisU \tag{4.33}$$
$$c_b = c_2 + (c_3 - c_2)AxisU \tag{4.34}$$
$$c = c_a + (c_b - c_a)AxisV \tag{4.35}$$

The end result is then combined in equation 4.36 [Shirley, 2005].

$$c = (c_0 + (c_1 - c_0)AxisU) + (c_2 - c_0)AxisV + (c_3 - c_2 - c_1 + c_0)AxisU \cdot AxisV \tag{4.36}$$



(a) The four data points around the intersection point P



(b) Image with and without texture filtering in TheMatrixDistributed

Figure 4.11: Bilinear filtering. (a) The data points around the intersection point p. (b)The left image shows an image rendered without texture filtering in TheMatrixDistributed. The right image is with bilinear texture filtering.

As can be seen from image 4.11b, texture filtering can make a large difference in image quality. Bilinear filtering is one of the simple methods that does not take much performance, but on the other hand, does not provide as elegant results as you get closer to an image. There are other methods, such as trilinear and anisotropic, that provide better quality renderings at the cost of performance. A small benchmark test in TheMatrixDistributed on 4 textured triangles at 1024x764 resolution with no sample gave 3.79 FPS without filtering and 3.00 FPS with filtering. Considering the improvement in the image, 0.79 FPS is a small price to pay.

4.4 Shading

The following sections will describe how objects in a ray tracer are coloured or shaded. In the end the methods used in TheMatrixDistributed are described. The section is inspired by the book Glassner [1989], except when cited otherwise.

4.4.1 Light in computer graphics

Light can be described as millions of photons, emitted every second from a light source. This is quite simplified though, as photons are also emitted at different frequencies. This is often converted to wavelength through the formula $\lambda = \frac{c}{f}$, where c is the speed of light in a vacuum and f is the frequency. The human eye perceives different ranges of wavelength as different colours - 475 nm gives blue light and 570 nm gives yellow light [Center, 2007]. White light is created when several photons with different wavelengths arrive at the eye at the same time - the eye is unable to distinguish between the different colours, and thus a white colour occurs.

A ray, consisting of several photons, with different wavelengths, forms the spectrum of a light source.

One might assume that the photons in a ray, travel in exactly the same direction with exactly the same speed. This is not the case. When light is reflected or transmitted on a surface, it comes out in an uneven manner. This is because no surface is truly smooth, though it may look it. A hole or bump will make the reflection uneven, even if it is only a nanometer wide. For a constant wavelength every surface has a reflection and transmission curve that describes the returned colour at a given wavelength and angle, $F(\lambda, \theta)$. This curve can also be thought of as an absorption curve, which is a function that describes how the surface reacts to a photon. The details of this curve, which returns the Fresnel term, will not be described, as it will be substituted with a constant in the final colour function.

Because the emitted light consists of many photons at different wavelengths, the light can have different colours. This is simplified to three variables that define a colour as a mix of red, green and blue light. Any reference to light or colour in the following sections refer to these three variables in vector form.

$$\mathbf{c}_l = \begin{pmatrix} r \\ g \\ b \end{pmatrix} \tag{4.37}$$

If the combination is $(1, 0, 0)$ the colour is red, and $(1, 1, 1)$ results in a white light.

In the following sections the term **L** is used to denote the direction of the light vector in unit length and **V** is a unit vector in the direction of the viewer from the intersection point. These two vectors both point toward the intersection point, due to backwards ray tracing.

Objects can have different properties when reacting to light. Light can be reflected off the surface or transmitted through the object. In addition the light can interact with the object. Table 4.1 lists the four different ways light can react to a surface, where diffuse reflection interacts with the surface, and specular reflection does not interact.

	Interaction	**Non interaction**
Reflection	Diffuse reflection	Specular reflection
Transmission	Diffuse transmission	Specular transmission

Table 4.1: Table of the four different types of shading.

These four different shading methods are described in detail in the following subsections.

4.4.2 Diffuse shading

When photons interact with the atoms of a surface diffuse shading is applied to it. The emitted photons can be thought of as vibrating, creating a small amount of energy, and when it hits an object, it interacts with that object's atoms. If the photon has just the right amount of energy the atom absorbs the energy for a brief moment. But it cannot stay in this condition for long, and emits the energy in the form of a new photon. This new photon seems to be reflected off the surface and the wavelength has changed with respect to the object.

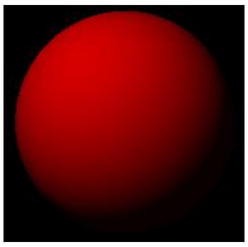


Figure 4.12: Principles of diffuse shading.

Ideal diffuse reflection

When a photon is reflected off a diffuse surface, the new photon could be emitted in any arbitrary direction, but with several photons hitting the same spot we can assume that light is reflected in all directions. Therefore the only concern for making a formula for the reflected colour is whether that point of the surface is visible from a light source. The Lambertian reflection model for ideal diffuse reflection says that the amount of light leaving the surface is proportional to the cosine of the angle between the surface normal, **N**, and the light vector, **L**. Thereby the smaller the angle, the bigger the amplitude, or energy if, as in this case, the frequency is constant. Figure 4.12 gives an example of diffuse reflection.

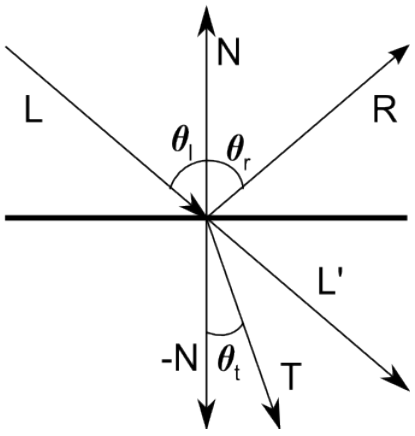


Figure 4.13: Reflection and transmission of a light ray.

Diffuse light reflection

The amount of light leaving an object after reflection with a diffuse surface can be described through the information above. The Lambertian reflection model is the only geometric consideration and can be calculated by the dot product between the inverse of the light vector[3], -**L**, and the normal vector, **N**. This accounts for the dependency on the angle, because the smaller the angle, the more reflected light. Because light reflected by a diffuse surface is not dependent on the viewer's angle, but only the spectrum of the incoming light and the physics of the surface, the dot product can be multiplied with the spectrum of the light source, $I_{lj}(\lambda)$, and the diffuse reflection curve of the surface, $F_{dr}(\lambda)$. Because there might be more than one light source, this part of the equation must be looped over for all light sources. At last a term for the ability for diffuse surfaces to reflect light, k_{dr}, is added, and the final equation, (4.38), for the light reflected from a diffuse surface from all light sources is done.

Because light can also reflect off other bodies, and then hit other objects, an equation for this must also be applied. However, there is no function which describes this for a diffuse shading, as there are no functions that can predict the reflected light's direction. Nevertheless, the aforementioned ambient colouring[4] can be applied here. Therefore equation 4.39, for how lights from other bodies affects the object, only depends on the spectrum of the "ambient light", the reflection curve of the surface, and the constant that represents the ability for the diffuse surface to reflect light.

$$\begin{aligned} I_{dr} &= k_{dr}\sum_j I_{lj}(\lambda)F_{dr}(\lambda)(\mathbf{N}\cdot -\mathbf{L}_j) && (4.38)\\ BI_{dr} &= k_{dr}I_a(\lambda)F_{dr}(\lambda) && (4.39)\end{aligned}$$

[3]See figure 4.13

[4]See section 4.1.2 about ambient shading

Ideal diffuse transmission

Diffuse transmission occurs when light hits an objects that allows light to pass through, but because it interacts with the atoms, the shapes behind are not clear. A perfectly diffused transmission scatters light in all directions, and just as perfect diffuse reflection the intensity is the same in all directions. Again there are no other geometrical considerations needed that the Lambertian reflection model.

Diffuse light transmission

The practical use of diffuse transmission is much the same as that of diffuse reflection, only the coefficients are renamed. the normal vector, **N**, is also inversed, because the transmitted ray is on the other side of the object. Therefore **L** is also moved so that it begins in the intersection point, but has the same direction. This is denoted **L′**

$$I_{Idt} = k_{dt} \sum_j I_{lj}(\lambda) F_{dt}(\lambda)(-\mathbf{N} \cdot -\mathbf{L'}_j) \tag{4.40}$$

$$BI_{dt} = k_{dt} I_a(\lambda) F_{dt}(\lambda) \tag{4.41}$$

Diffuse transmission versus diffuse reflection

Because of the physics in diffuse transmission and reflection, both of the shading methods cannot be applied at the same time. If the light source is on the same side of the object as the camera, the light will be subjected to diffuse reflection, but if they are on opposite sides of the the object, diffuse transmission will be the case. This fact applies both to the light emitted from a light source and the light reflected/transmitted off other bodies.

4.4.3 Specular shading

Specular shading is when the photon does not interact with the surface, and therefore almost does not obtain any colour from that object. As with diffuse shading we can only talk about *ideal* situations, even though they do not exist in reality. Almost ideal specular reflection occurs with, for example, a mirror where the colour of the light almost does not change when it is reflected off the mirror.

Ideal specular reflection

Contrary to diffuse reflection, specular reflection only reflects in one direction, as it does not interact with the object itself. The reflection vector is defined on the basis that the reflection ray **R** and the light ray **L** are in the same plane, and that the angle between **L** and **N**, θ_l, and the angle between **R** and **N**, θ_r, is the same[5]. Therefore it can be deduced that **R** is a linear combination of **L** and **N**, where both **L** and **N** are in unit

[5]See figure 4.13

length.

$$\begin{aligned}\theta_l &= \theta_r && (4.42)\\ \mathbf{R} &= \alpha\mathbf{L} + \beta\mathbf{N} && (4.43)\end{aligned}$$

Equation 4.42 indicates that the cosine of the two angles are also identical, (4.44). The cosine can be calculated by the dot producted of two unit vectors, which implies equation 4.45. Here **R** can be substituted with the definition of **R** from equation 4.43, and α and β can be removed from the dot product. Equation 4.46 and 4.47 show these calculations.

$$\begin{aligned}cos(\theta_{\mathbf{L}}) &= cos(\theta_{\mathbf{R}}) && (4.44)\\ -\mathbf{L}\cdot\mathbf{N} &= \mathbf{R}\cdot\mathbf{N} && (4.45)\\ &= (\alpha\mathbf{L} + \beta\mathbf{N})\cdot\mathbf{N} && (4.46)\\ &= \alpha(\mathbf{L}\cdot\mathbf{N}) + \beta(\mathbf{N}\cdot\mathbf{N}) && (4.47)\\ -\mathbf{L}\cdot\mathbf{N} &= \alpha(\mathbf{L}\cdot\mathbf{N}) + \beta && (4.48)\end{aligned}$$

Since $\mathbf{N}\cdot\mathbf{N} = 1$ this term can be ignored in (4.47). Because α and β only is constants used to describe the relation between **L** and **R**, one of them can be set to 1 and the other can describe this relation alone. If α is set to 1, only β is unknown, and can therefore be deduced.

$$\begin{aligned}\alpha &= 1 && (4.49)\\ \beta &= -\mathbf{L}\cdot\mathbf{N} - (\mathbf{L}\cdot\mathbf{N}) && (4.50)\\ &= -2(\mathbf{L}\cdot\mathbf{N}) && (4.51)\end{aligned}$$

Now the unknowns in equation 4.43 are defined, and equation 4.52 can be used to calculate a vector in the direction of the reflected ray. Remember that **L** and **N** must be in unit length.

$$\mathbf{R} = \mathbf{L} - 2(\mathbf{L}\cdot\mathbf{N})\mathbf{N} \qquad (4.52)$$

Most objects will not be completely shaded with perfect specular reflection. However, most objects do have some areas where specular reflection occurs to some extent. Looking at a piece of plastic, some areas will look brighter than others, and the colour of the object does not seem to influence that area. Such phenomenons are called highlights, and occur when the angle between **L** and **N** is small. Figure 4.14 shows the same object as in figure 4.12 shaded with specular reflection.

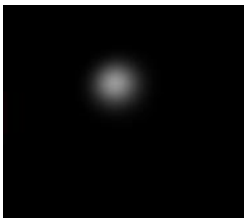


Figure 4.14: Principles of specular shading.

Specular light reflection

As with diffuse reflection, specular reflection is not ideal. Some objects have rough areas, but do still generate specular reflection. A

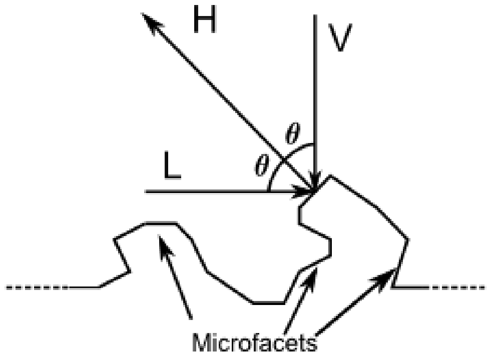


Figure 4.15: A surface with microfacets showing the placement of **H**.

new vector **H** is added to evaluate when specular reflection occurs,
and is located right between **L** and **V**, see figure 4.15.

$$H_j = \frac{-\mathbf{V} - \mathbf{L}_j}{||-\mathbf{V} - \mathbf{L}_j||} \tag{4.53}$$

$$I_{sr} = k_{sr} \sum_j I_{lj}(\lambda) F_{sr}(\lambda, \theta_{r,j}) (cos(\theta_{r,j}))^{p_r} \tag{4.54}$$

If **H** is calculated as in equation 4.53 (remember, V is the ray from the viewer to the intersection point), specular reflection will occur when the angle between **H** and **N** is small. Therefore the first component of the equation for specular light reflection is the inverse cosine of the dot product between **H** and **N**. This angle will for future reference be denoted as $\theta_{r,j} = cos^{-1}(\mathbf{N} \cdot \mathbf{H}_j)$. However, this is not realistic enough, because the function will fade off gradually, creating a bigger highlight than in real life. Therefore a Phong exponent, p_r is added. Like the equation for diffuse reflection, specular reflection depends on the surface's specular reflection curve, but because specular reflection depends on the angle of the light, this is also a part of the function. Additionally the spectrum of the light source is added, and the equation is looped over for all light sources. At last the equation for specular light reflection is multiplied with the constant k_{sr} to account for the surface's ability to specularly reflect light. This completes equation 4.54.

Light can also be specularly reflected off objects and thereby shade other objects. This can for example be seen when a bowl is reflected in the shiny surface of the table. When calculating the reflection from other bodies the reflected ray R from equation 4.52 is used. If the camera ray, r_a, hits object A, a ray r_b is thrown in the direction described in equation 4.52 and if it hits another object, e.g. B, it is checked if this point is in shadow or not. If not, the point on object B is used to shade object A. However B could also be shaded by another object, which makes this a recursive function that continues until a certain point: the depth of field. The depth of field can either be set to a certain colour or a certain number of reflections/transmissions.

Equation 4.55 accounts for this shading from other bodies. It is quite like I_{sr}, (4.54), except for the dot product. Because the specular reflection from other bodies only apply to rays that are already perfectly specularly reflected, the light spectrum will not have to be bounded by the angle. However, another fact must be applied to the equation: the fall in light intensity over distance. The term T denotes this fall in light intensity per distance unit in a medium, and Δsr denotes the distance between the two intersection points. This results in a lower light intensity the longer the distance.

$$BI_{sr} = k_{sr}I_{sr}(\lambda)F_{sr}(\lambda,\theta_r)T_r^{\Delta sr} \tag{4.55}$$

Ideal specular transmission

Light is perfectly specular transmitted when it passes through an object without the colour changing from the interaction with the object itself. The light does reflect off every atom of the object, in the same way as a specular reflection, but because of the size of θ_l and the density of the object, the light is reflected through the object instead of off the object. The reflections create the deviation seen, for example, through water, where the object seems out of place. Therefore we need an expression for the outgoing angle depending on the incoming angle. For a ray passing through two mediums Snell's law describes this connection.

$$\frac{sin(\theta_1)}{sin(\theta_2)} = \frac{\eta_2}{\eta_1} = \eta_{21} \tag{4.56}$$

Here η_1 is the index of refraction of the first medium with respect to vacuum, η_2 is the index of refraction of the second medium with respect to vacuum, and η_{21} the index of refraction of the first medium with respect to the second medium.

To find an expression for the transmitted ray **T**, the same approach as for **R** will be used. Figure 4.13 shows the vectors involved, where **L** is the light ray, **L′** is a continuation of **L**, **N** is the normal vector and **T** is the transmitted ray. Snell's law and the fact that **N**, **L** and **T** are in the same plane, from the the two initial equations - equation 4.57 and 4.58.

$$\frac{sin(\theta_t)}{sin(\theta_l)} = \eta_{lt} \tag{4.57}$$

$$\mathbf{T} = \alpha\mathbf{L} + \beta\mathbf{N} \tag{4.58}$$

When solving **T**, α and β are the only unknowns, and hence must be determined. This is done by finding two equations for α and β and solving for them. By isolating $sin(\theta_t)$ from equation 4.57 and raising the equation to the second power, we get equation 4.59. Because of the sine/cosine relations in $sin(\theta)^2 + cos(\theta)^2 = 1$, it is possible to remove the sine calculation from equation 4.59. This creates equation 4.60

$$sin(\theta_l)^2\eta_{lt}^2 = sin(\theta_t)^2 \tag{4.59}$$

$$(1 - cos(\theta_l)^2)\eta_{lt}^2 = 1 - cos(\theta_t)^2 \tag{4.60}$$

Now only θ_t is unknown, but because the cosine calculation can be substituted with the dot product between **-N** and **T** we derive equation 4.62. Here **T** is again unknown, but from equation 4.58 it can be substituted with α and β, as in equation 4.63. Because the dot product between $-L$ and N is $cos(\theta_l)$, the dot product between L and $-N$ can be substituted with this. Together with $-N \cdot -N = 1$ this gives the final equation 4.65.

$$\begin{aligned}
(1 - cos(\theta_l)^2)\eta_{lt}^2 - 1 &= cos(\theta_t)^2 && (4.61)\\
&= (-\mathbf{N} \cdot \mathbf{T})^2 && (4.62)\\
&= (-\mathbf{N} \cdot (\alpha\mathbf{L} + \beta\mathbf{N}))^2 && (4.63)\\
&= (\alpha(-\mathbf{N} \cdot \mathbf{L}) + \beta(-\mathbf{N} \cdot \mathbf{N}))^2 && (4.64)\\
(1 - cos(\theta_l)^2)\eta_{lt}^2 - 1 &= (\alpha cos(\theta_l) - \beta)^2 && (4.65)
\end{aligned}$$

The second equation needed to determine α and β, can be calculated if we set **T** to be of unit length, leading to (4.66). By substituting the definition of **T** from equation 4.58 in this equation, and collecting the dot products in equations 4.67, equation 4.68 is formed. Again $\mathbf{L} \cdot \mathbf{N}$ can be substituted with $-cos(\theta_l)$, $\mathbf{N} \cdot \mathbf{N} = 1$ and $\mathbf{L} \cdot \mathbf{L} = 1$. Thereby the last needed equations are stated in equation 4.69.

$$\begin{aligned}
1 &= \mathbf{T} \cdot \mathbf{T} && (4.66)\\
&= (\alpha\mathbf{L} + \beta\mathbf{N}) \cdot (\alpha\mathbf{L} + \beta\mathbf{N}) && (4.67)\\
&= \alpha^2(\mathbf{L} \cdot \mathbf{L}) + 2\alpha\beta(\mathbf{L} \cdot \mathbf{N}) + \beta^2(\mathbf{N} \cdot \mathbf{N}) && (4.68)\\
1 &= \alpha^2 - 2\alpha\beta cos(\theta_l) + \beta^2 && (4.69)
\end{aligned}$$

Equations 4.65 and 4.69 only contain the two unknowns: α and β. These can be solved by substitution. Solving for α and β gives four different values where only the first are relevant, because the others are projections in the other quadrants. These are stated in equations 4.70 and 4.71.

$$\begin{aligned}
\alpha_1 &= \eta_{lt} && (4.70)\\
\beta_1 &= \eta_{lt} cos(\theta_l) - \sqrt{1 + \eta_{lt}^2(cos(\theta_l)^2 - 1)} && (4.71)
\end{aligned}$$

With every unknown determined, the expression for **T** is determined.

$$\mathbf{T} = \eta_{lt}\mathbf{L} + \left(\eta_{lt}\sqrt{1 + \eta_{lt}^2(cos(\theta_l)^2 - 1}\right)\mathbf{N} \qquad (4.72)$$

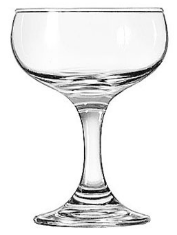


Figure 4.16: Total internal reflection in a champagne glass. [Association]

This expression for **T** contains a square root, which can only be calculated for positive values. However the physics does account for the situation when the expression is negative. The phenomenon is called *total internal reflection* and occurs when a ray tries to go from a highly dense medium to a low density medium with just the right angle, and is then reflected back in to the highly dense medium. If this ray is unable to leave the medium it will create black areas of total internal reflection like those seen in for example, the foot of a glass (See figure 4.16). In real life total internal reflection is utilised in fiber optics, where the tubes have a higher density than the surroundings, and signals can thereby be transmitted over long distances.

Specular light transmission

Again the equations for specular light transmission are much the same as for specular light reflection, only the variables are different.

$$\begin{aligned} I_{Ist} &= k_{st}\sum_j I_{lj}(\lambda)F_{st}(\lambda,\theta_{t,j})(cos(\theta_{t,j}))^{p_t} && (4.73)\\ BI_{st} &= k_{st}I_{st}(\lambda)F_{st}(\lambda,\theta_t)T_t^{\Delta st} && (4.74)\end{aligned}$$

4.4.4 Hall's model

Hall's model collects all the above mentioned equations for light transportation in one equation.

$$\begin{aligned} I(\lambda) &= k_{dr}\sum_j I_{lj}(\lambda)F_{dr}(\lambda)(\mathbf{N}\cdot-\mathbf{L}_j) + k_{dr}I_a(\lambda)F_{dr}(\lambda)\\ &+ k_{dt}\sum_j I_{lj}(\lambda)F_{dt}(\lambda)(-\mathbf{N}\cdot-\mathbf{L}_j) + k_{dt}I_a(\lambda)F_{dt}(\lambda)\\ &+ k_{sr}\sum_j I_{lj}(\lambda)F_{sr}(\lambda,\theta_{r,j})(cos(\theta_{r,j}))^{p_r} + k_{sr}I_{sr}(\lambda)F_{sr}(\lambda,\theta_r)T_r^{\Delta sr}\\ &+ k_{st}\sum_j I_{lj}(\lambda)F_{st}(\lambda,\theta_{t,j})(cos(\theta_{t,j}))^{p_t} + k_{st}I_{st}(\lambda)F_{st}(\lambda,\theta_t)T_t^{\Delta st} && (4.75)\end{aligned}$$

where

k_{dr} is the term for diffuse reflectance
k_{dt} is the term for diffuse transmission
k_{sr} is the term for specular reflectance
k_{st} is the term for specular transmission
$I_{lj}(\lambda)$ is the spectrum of the light source j
$I_a(\lambda)$ is the spectrum of the ambient light
λ is the wavelength
F_{dr} is the diffuse reflection curve
F_{dt} is the diffuse transmission curve
F_{sr} is the specular reflection curve
F_{st} is the specular transmission curve
$\mathbf{N}$ is the normal vector in the intersection point
$\mathbf{L}_j$ is the incoming light vector
$\theta_{r,j} = cos^{-1}(\mathbf{N}\cdot\mathbf{H}_j)$
$cos(\theta_{r,j}) = \mathbf{N}\cdot\mathbf{H}_j$
$H_j = \frac{-\mathbf{V}-\mathbf{L}_j}{||-\mathbf{V}-\mathbf{L}_j||}$
p_r is the Phong exponent for reflection
p_t is the Phong exponent for transmission
θ_r is the angle between the reflected ray **R** and the normal vector **N**
θ_t is the angle between the transmitted ray **T** and the normal vector **N**
T_r is a term for the fall in light intensity per unit in a medium

T_t is a term for the fall in light intensity per unit in a medium
Δsr is the distance traveled by a reflected ray between since the last intersection
Δst is the distance traveled by a transmitted ray between since the last intersection

Throughout this section, pictures of the different aspect of the shading models have been shown. With the final equation defined a complete picture is possible. Figure 4.17 shows a reflective surface, shaded with ambient, diffuse and specular shading.

4.4.5 Shading in TheMatrixDistributed

The main purpose of TheMatrixDistributed is to be able to perform these calculations in real time, and therefore it has been necessary to limit the effects. Because reflection and transmission are very expensive, and are possible to leave out and still create a somewhat realistic image, it has been chosen to leave them out. Therefore TheMatrixDistributed is not able to reflect colour from other bodies nor able to transmit light, and the Hall shading model from section 4.4.4 has to be reduced. Equation 4.76 contains what is needed from equation 4.75, to shade objects with diffuse and specular reflection as well as ambient shading.

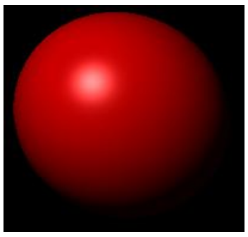


Figure 4.17: Principles of Phong shading.

$$\begin{aligned} I(\lambda) &= k_{dr} \sum_j I_{lj}(\lambda) F_{dr}(\lambda)(\mathbf{N} \cdot -\mathbf{L}_j) + k_{dr} I_a(\lambda) F_{dr}(\lambda) \\ &+ k_{sr} \sum_j I_{lj}(\lambda) F_{sr}(\lambda, \theta_{r,j})(\mathbf{N} \cdot \mathbf{H})^{p_r} \end{aligned} \tag{4.76}$$

In our implementation the functions $I_{lj}(\lambda)$, $F_{dr}(\lambda)$, $I_a(\lambda)$ and $F_{sr}(\lambda, \theta_{r,j})$ will not depend on wavelengths, but only be constants chosen between 0 and 1[6]. In equation 4.77 the functions have been changed to constants.

Because both k_{dr} and F_{dr} describes the surfaces ability to diffusely reflect light, they can be united in one constant, c_{dr}. The same accounts for k_{sr} and F_{sr} which can be collected in one constant c_{sr}. In equation 4.78 this change has been made, and c_{dr} has been isolated.

When this function is implemented in the program it is easier to loop over just one function, and therefore the last editing will be to loop over the entire function. Thereby the final equation is (4.79).

$$I = k_{dr} \sum_j I_{lj} F_{dr}(\mathbf{N} \cdot -\mathbf{L}_j) + k_{dr} I_a F_{dr} + k_{sr} \sum_j I_{lj} F_{sr}(\mathbf{N} \cdot \mathbf{H})^{p_r} \tag{4.77}$$

$$I = c_{dr} \sum_j (I_{lj}(\mathbf{N} \cdot -\mathbf{L}_j) + I_a) + c_{sr} \sum_j (I_{lj}(\mathbf{N} \cdot \mathbf{H})^{p_r}) \tag{4.78}$$

$$I = \sum_j (c_{dr}(I_{lj}(\mathbf{N} \cdot -\mathbf{L}_j) + I_a) + c_{sr} I_{lj}(\mathbf{N} \cdot \mathbf{H})^{p_r}) \tag{4.79}$$

[6]Se section 4.4.5

Choosing values

Equation 4.79 contains several constants which need to be addressed. The constant c_{dr} describes the surface's ability to diffusely reflect light. However, because diffuse reflection occurs when the light ray interacts with the object's atoms, a good approximation of c_{dr} would be the colour of the object itself. c_{sr}, on the contrary, does not have anything to do with colour, but only describes how well the surface specularly reflects. Therefore it could be set at a value between 0 and 1, where 1 is a very sharp highlight, and 0 will give no highlight. I_l is the intensity of the emitted light, and is therefore set for the individual light source. I_a is the intensity of the ambient light source, which is not really a light source but an overall light intensity. Both can be set within the interval $[0, 1]$, where TheMatrixDistributed has $I_a = 0.05$. p_r is the phong exponent for reflective surfaces. It can be set to any value - however, the smaller the value the bigger the highlight. In TheMatrixDistributed values between 1 and 150 are usually used. Lastly, **N** is the normal vector in the intersection point, **L** is a vector the direction of a light source, pointing towards the object, and **H** is a vector halfway between a vector in the direction of the camera and **L**. All three are unit length.

TheMatrixDistributed also enables the possibility to only use diffuse and ambient shading. This is achieved by setting $c_{sr} = 0$ and thereby eliminating the last segment of (4.79) and making (4.80).

$$I_d = \sum_j (c_{dr}(I_{lj}(\mathbf{N} \cdot -\mathbf{L}_j) + I_a)) \tag{4.80}$$

This additional shading method is also implemented in TheMatrixDistributed because it is simpler and faster due to the removal of a dot product and an exponent. The only difference to Phong shading is the lack of highlights. When choosing shading method in TheMatrixDistributed one can therefore chose between Phong shading and diffuse shading.

4.5 Light sources

In this section the difference between directional light, point light *and* spot light *will be described. This section is based on the books by Suffern [2007] and Glassner [1989] except when cited otherwise. However, first some parameters used in the following subsections will be explained.*

Lighting up a scene may be done by *direct illumination* or *indirect illumination*. Direct illumination is when the light rays emitted from a given light source hits an object directly. Indirect illumination is when the light rays emitted from a given light source are reflected from at least one surface before hitting the object. This section will describe three different types of light sources which may be used to add direct illumination to a scene.

All light sources have a vector **L** describing in which direction the light rays are emitted. When viewing the light from a point on a surface, the direction of the incoming light rays may therefore be defined as the inverse of the direction of the light rays, thus **-L**. The colour of a light ray is given by the colour vector $\mathbf{c}_l$ based on the RGB colour model (explained in section 4.4.1). Light rays have an angle in which they hit a surface, which also is called the angle of incidence and is denoted θ_l. The angle of incidence is calculated as the dot product of the surface normal **N** and the inverse of the direction of the hitting light rays **-L**.

4.5.1 Directional light

Directional light is a mathematical abstraction not existing in real life. The light rays of a directional light are emitted from a light source with a distance from a given object so massive that the light rays appears to be parallel with each other when they reach the object. The sun is an example of a directional light source. Thus, a directional light in a scene will apply a constant and uniform light to the scene. The direction from where the light originates, will not be specified by a point or distance, but only by a direction **L** and a colour $\mathbf{c}_l$. Describing the incoming light I_{in} from a directional light source at a point **p** on a surface is done as in equation 4.81 and illustrated by figure 4.18

$$I_{in} = \mathbf{c}_l(\mathbf{N} \cdot \textbf{-}\boldsymbol{L}) \tag{4.81}$$

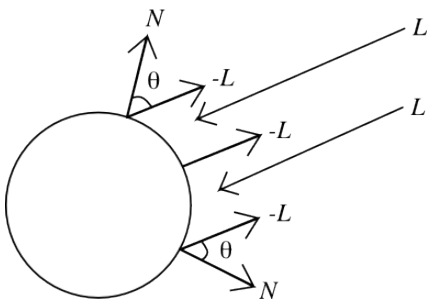


Figure 4.18: Illustration of the light direction **-L** from a directional light source. **L** is the same for all hit points.

4.5.2 Point light

A point light source differs from a directional light as it has a position $\mathbf{p}_l$, and emits light rays in all directions, thus the direction of the light, **-L**, is different for each given point **p** on the surface. This is illustrated in figure 4.19. A point light is also an abstraction because a point light in a program only will be represented by a coordinate. Therefore the point light will not have any surface area, which it would in the real world. The effect is that a point light does not create soft shadows (See figure 4.4). In reality the intensity of a light, would decrease, the greater the distance is between the hit point on the surface and the light source. This is, in physics called *the inverse square law*, and in computer graphics *the distance attenuation*. The intensity of the light

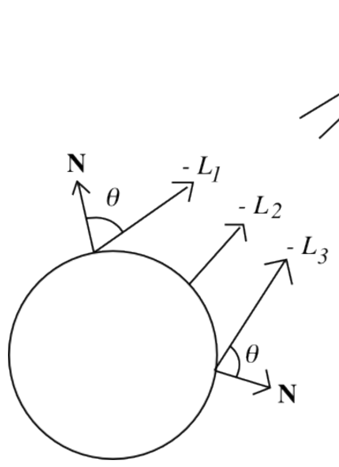


Figure 4.19: Illustration of a point light. The direction ***l*** for a point light depends on the origin p_l. of the light rays

at a point on a surface **p** therefore depends on the distance **r** to the light source p_l. The distance attenuation may be described by applying $(\mathbf{r}^2)^{-1}$ to equation 4.81, where $r = |||\mathbf{p}_l - \mathbf{p}|||$ is the absolute distance between the intersection point and the point light. The intensity of the light from a given point light source is then given by equation 4.82:

$$I_{in} = \mathbf{c}_l(\mathbf{N} \cdot \mathbf{-L})(\mathbf{r}^2)^{-1} \tag{4.82}$$

A point light may be created without applying a distance attenuation, but then the light intensity will not depend on the distance.

4.5.3 Spot light

A spot light is created in the same way as a point light, but instead of emitting photons in all directions, the spread of photons is limited by a beam angle ω. The direction of the light is the center line l_c of the light cone, as illustrated in figure 4.20. From trigonometry it is given that the dot product between two unit vectors is the cosine of the angle between the two vectors. If the dot product is 0 the vectors are perpendicular and the spread is at its maximum. If it is 1 the vectors are parallel and no light is emitted.

When determining if a point **p** on a surface is within the light cone of a spot light, a ray is shot from the surface point **p** to the light source $\mathbf{p}_l$ and a normalised vector $\mathbf{p}_1$ is created. Then the dot product between $\mathbf{p}_1$ and $\mathbf{l}_c$ denoted $cos(\alpha)$ is compared to $cos(\frac{\omega}{2})$. If $cos(\alpha) > cos(\frac{\omega}{2})$ the point is illuminated by the spotlight. If $cos(\alpha) \leq cos(\frac{\omega}{2})$ the point is not illuminated by the spot light.

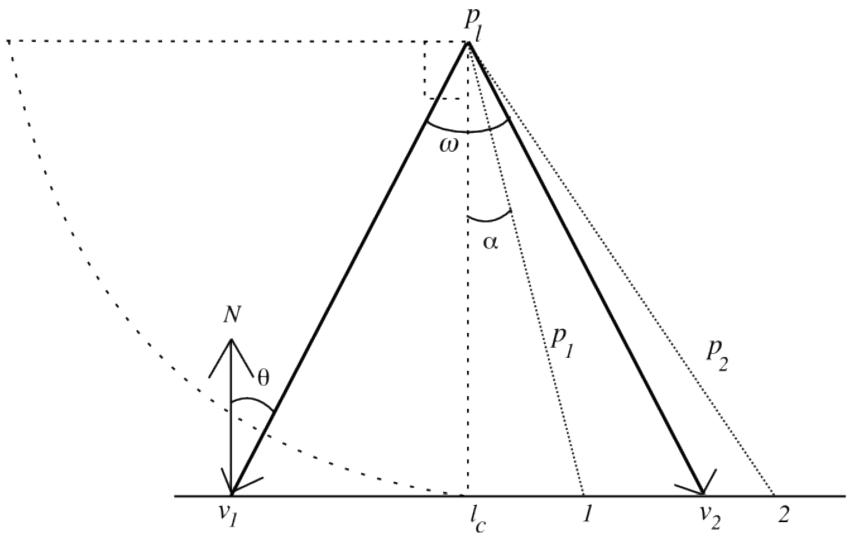


Figure 4.20: Illustration of a spot light and the beam angle ω restricted by the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$

CHAPTER 5

TheMatrixDistributed

This chapter will describe the program TheMatrixDistributed, which has been written to investigate some of the problematics of doing ray tracing in real time. The mathematical theories described in chapter 4 are used in the program, and TheMatrixDistributed will utilise basic ray tracing techniques such as diffuse shading, phong shading and shadows. TheMatrixDistributed supports three primitives: sphere, plane and triangle. Triangle is the only one that offers textures with bilinear filtering. A pin hole camera has also been implemented, to allow the user to move around in the scene. Two different light sources, point light and spot light, have been added and it is possible to have multiple light sources in the scene at once. Furthermore TheMatrixDistributed supports 4x anti-aliasing, to increase image quality.

To allow the creation of large scenes with hundred and thousands of objects, a parser has been implemented to import 3D models from obj files to TheMatrixDistributed as models. To speed up the rendering TheMatrixDistributed uses bounding volume hierarchies, and to further speed up the ray tracer, multithreading has been implemented. Finally, TheMatrixDistributed allows the rendering process to be distributed among multiple computers using TCP/IP, to increase the computational power and allow faster rendering.

5.1 Program overview

TheMatrixDistributed begins by creating a `Scene` where all information about objects, camera and lights are stored. `Scene` has a method called `renderPixel` which renders a specific pixel. TheMatrixDistributed can be created as both a server-client application or a standalone application. In the standalone version the main function is responsible for updating the screen, handling user input and calling `renderPixel` for all the pixels. In the server-client version the server serializes the `Scene` and sends it to all the clients in the beginning of each frame. The server then sends requests on segments for the client to render. The clients call the `renderPixel` on the pixels in the segment and return the colour of pixels to the server. When the server has received the colours for all pixels on the screen it updates the screen, handles events and transmits a new `Scene` to all the clients which then starts rendering requested segments.

The `renderPixel` method works by first calling the camera which returns the ray used to trace objects. That ray is then traced for intersection with objects on the scene using the bounding volume hierarchies. When the closest object that intersects has been found, the method `applyShading` is called on the closest object. This method is responsible for computing light, shadow and colour of the surface, and returns the result to `renderPixel`.

5.2 Development method

This section briefly describes development methods and discusses ours. Information about the different development methods has been acquired from Larman [2003]

When developing various programs or solutions, there are a lot of different development methods available to help guide and focus the development team. Some methods are better then others at certain goals, so choosing which development method to follow can be confusing. Choosing a development method is also about looking at how you and your development team work together individually, so the chosen development method will often reflect how the group works together.

Iterative development methods splits the development into segments. Each segments goal is to fulfill a requirement, and at the end of each segment a partially tested, but working, system is developed. This system is then developed further in the next segment. The partial tests provide feedback for further development, and the final segment is the final product that is to be released. Furthermore, each segment acts as a full miniproject, with analysis, design, programming and tests, and the general recommended iteration length is between one to six weeks.

Evolutionary methods are in some ways extensions of iterative methods. Plans, solutions and requirements can evolve during the development process. When using evolutionary methods, the outcome is not necessarily written in stone, but evolves as the project advances. This allows for flexibility when unexpected discoveries emerge.

Agile methods are difficult to define. At the core of agile methods are the evolving methods of the evolution/iterative methods. The keyword in agile methods is change, however, the different agile methods can vary greatly. Some popular agile methods are Scum and Extreme Programming (XP).

Scum is directed towards working in a common project room with self-directed and self-organising teams. There are daily stand-up meetings to pose questions and express comments and each iteration is adaptively planned.

XP focuses on quick and skillful programming, and iterations are often one to three weeks. Like Scum, XP is directed towards working in a common project room for communication amongst the team. The method also has a great deal of focus on programming, as the name suggests. This is expressed through methods such as pair programming, in which a pair of programmers code together, in order to create higher quality code at an efficient speed. The development is also very test driven. Unit tests are often created for each segment of code before the code is written, and the code is then written to pass these unit tests. As a trade off, XP methods have less documentation as the focus is on the programming.

We have not used any one method during the development of TheMatrixDistributed. Instead we have taken a few aspects from different methods, and suited them to our purposes. For example, we have created TheMatrixDistributed, by literally adding features and abilities as we discovered how to add them. This occasionally meant that the entire framework had to be redone, however we always had a working program to fall back on. We have also used unit testing on some of our classes where unit tests were relevant, e.g. our `Vector3D` class. These tests have allowed us to find and correct bugs, which would otherwise have been very difficult to spot. The coding structure of TheMatrixDistributed was not something that was initially planned and thought out. It was something that evolved during the development process, which meant that occasionally some features and classes had to be revised to allow other features to properly use them.

In conclusion there has not been used one single development method; only the aspects that we felt suited our group the best has been selected.

5.3 Tools and libraries

This section describes which tools and libraries that have been used for TheMatrixDistributed and why.

5.3.1 C++

An object oriented programming language was desired in this project because of the structure of ray tracing. The primary reason for using an object orientated language is that it is possible to make abstracts classes to e.g. represent the primitives. This makes it easy to add new primitives without the need to edit the ray tracer in many other places.

Since the programming language should have minimal overhead, because a ray tracer is a CPU bound program, C++ was considered a good solution. It is relatively close to C, which we have previously used, and the syntax is also similar to that of Java, which we currently have a course in. Languages such as Java, C# and Python were not considered good solutions because of the overhead. It is harder to do dynamic allocation in C++, but dynamic allocation is also expensive and something we would like to avoid.

Initially a basic ray tracer with a sphere and a very simple camera, was written in Python, mainly to become familiar with the concept of ray tracing. The performance of this ray tracer was not good as it took about 30 seconds to render a frame[1]. Rewriting the exact same code[2] in C++ yielded 30 frames per second. After this simple test it was easy to see that high level languages were not suitable for CPU bound applications.

While C++ may be harder to use and even though none of us have any experience with C++ other than very basic and years old knowledge, C++ is the language that offers the best performance and features for the purpose of real time ray tracing. There are also other unmanaged object orientated programming languages, however, it is easy to interface C libraries from C++, which is very useful because low level graphics and network libraries are often written in C. A last reason for choosing C++ is that we have access to Intel's C++ compiler through their non-commercial license, and this compiler is probably the best compiler for optimising today.

[1]Specializing the ray tracer with psyco reduced rendering time to 18 seconds.

[2]The Python implementation used pygame as SDL bindings, and the C++ implementation interfaced SDL through the C API.

Compilers

TheMatrixDistributed can be compiled with GCC (GNU Compiler Collection), ICC (Intel C++ Compiler) and MSVC[3] (Microsoft Virtual C++). We have chosen GCC because its debugger is integrated with CodeBlocks[4], which is a very simple IDE (Integrated Development Environment). However, ICC has proved to offer better optimisations than GCC, so for this reason we have chosen to make TheMatrixDistributed compatible with ICC, a process that only requires that we do not use any GCC specific extensions.

5.3.2 SDL

TheMatrixDistributed uses SDL, (Simple Directmedia Layer) a free cross platform multimedia library written in C. SDL makes it possible to create a simple window where graphics can be displayed, and also offers a low level direct pixel access and double buffers the frame buffer, thus has a low overhead. SDL is also free software and available for a wide range of platforms, and since TheMatrixDistributed is written in C++ it is also easy to interface SDL. The library itself is also easy to use and rather well documented.

5.3.3 SimpleSockets

TheMatrixDistributed uses a lightweight cross-platform socket library called SimpleSockets. SimpleSockets is a free software library that encapsulates BSD sockets, which ought to be portable, however, Microsoft Windows does not implement them correctly. Also when using this library we do not need to worry about BSD sockets, as this library makes it rather easy to do network communication, though it does not handle any threading at all. Note, this library is very small and only has two classes, which are both compiled directly with the client and server.

5.3.4 libnetpbm

To read images for textures TheMatrixDistributed uses PPM (Portable PixMap), which is a uncompressed format and easy to read. For reading PPM files TheMatrixDisitrbuted uses libnetpbm. This library is simple to use, unfortunately it has also turned out not to be very memory efficient. According to its documentation [PPM, 2003] it uses 16 bytes to represent a pixel that only takes 3 bytes in raw PPM format. So avoiding libnetpbm and just reading the PPM files directly could be faster, however, we have chosen not to focus on this aspect. Replacing libnetpbm by simply reading binary PPM files directly would be a desirable improvement for future work. The primary reason why we decided to use a library for reading the files, was that we initially did not want

[3]Compilation with MSVC may require a few changes, as compatibility with this compiler is a low priority.

[4]Better IDEs are probably readily available, however, we do not need a wide range of features, just a simple and easy to use IDE.

to waste development time learning how to read images. However, considering that PPM is a rather simple format, it should be possible to change this.

5.3.5 UnitTest++

For unit testing we have used UnitTest++, which is a free lightweight cross platform unit testing framework for C++. This library is also very simple to use and fairly stable. We have mainly used this library for testing if vectors works.

5.4 Complexity theory

This section describes aspects of complexity analysis

5.4.1 Worst-case complexity

One of the problems with ray tracing is that it uses many calculations, and therefore needs optimisations to run in real time. This creates problems when evaluating which optimisation is the fastest. At first glance some might think that measuring the time required to perform a calculation is a good measurement, however, because of the rapid development in hardware, such a measurement would be outdated the moment it is published. This is why another measurement is needed. Big-O notation denotes the asymptotic increase in computation time in relation to the number of objects. This does not mean that big-O denotes the actual time it takes to compute a scene, but the worst case increase in time with a variable number of objects. An example of a worst-case complexity could be $O(n)$, which means that there will be a linear increase in computational time with an increase in the number of objects. This measurement is independent on the computers hardware, and can therefore be used to compare e.g. optimisations across ages.

Definition 1 Let f and g be functions $f, g : N \rightarrow R+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$

$$f(n) \leq cg(n) \tag{5.1}$$

If $f(n) = O(g(n))$ $g(n)$ is said to be an *asymptotic upper bound* for $f(n)$. [Sipser, 2006]

By this definition $g(n)$ is the worst-case complexity of $f(n)$ if a constant makes it bigger or equal to $f(n)$ for a number $n > n_0$.

Example 1 If $f(x) = 6x^2 + 3x + 1$ then for $x > 1$, $1 \leq x^2$ and $x \leq x^2$. This implies the following

$$0 \leq 6x^2 + 3x + 5 \leq 6x^2 + 3x^2 + x^2 = 10x^2 \tag{5.2}$$

when $x > 1$. From equation (5.2) $c = 10$ and $n_0 = 1$, which by definition 1 implies that $f(x)$ is $O(x^2)$.

[Sipser, 2006]

Complexity analysis will be used later in section 5.5 to analyse the different methods of generating trees.

5.5 Bounding volume hierarchy

This section describes the bounding volume hierarchy used to accelerate TheMatrixDistributed.

The main obstacle when doing real time ray tracing is the many intersection calculations. These intersections are done with different types of objects, where some are easier to calculate than others. A way of optimising the ray tracer could therefore be to eliminate the difficult intersections, and substitute them with easier ones. A method is to enclose the difficult objects in other objects that are cheaper to check intersection with, e.g. spheres or cubes, and intersect with these first. For instance it is only necessary to test for intersections on a triangle if the rays hits a cube surrounding the triangle.

5.5.1 Creating bounding volume hierarchy

In TheMatrixDistributed axis aligned bounding boxes (AABB) are used to eliminate the difficult intersection calculations by checking for intersections with a cube before the object itself. Cubes are chosen because sphere intersections contain an expensive square root and because cubes fits more thighly around a triangle. Bounding boxes can also be used in a way to optimise the engine even further. A bounding volume hierarchy (BVH) places bounding boxes in a hierarchy, where the leafs are objects and the nodes are bounding boxes. A bounding box encloses its children, whether they are leafs or nodes. Thus if a node is excluded, i.e. a ray does not intersect the bounding box, then there is no need to perform an intersection test with the children of the node.

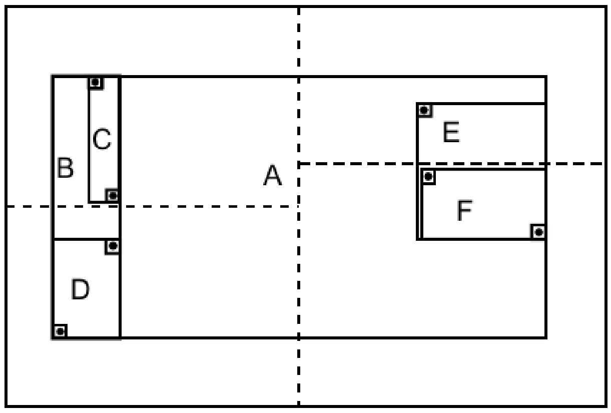


Figure 5.1: Result from creating a bounding volume hierarchy with the SpaceBalancedTree algorithm.

Figure 5.1 illustrates a BVH for six objects. There are several methods for creating BVHs, two of these methods will be discussed here. The BVH tree in figure 5.1 is called a space balanced tree (SBT). It is created by taking the list of objects and dividing them in two lists so that there is an even amount of space represented by the two sublists. This is then repeated recursively, until there are only two objects in every list. These two object are then placed in a bounding box, and that bounding box is put into a bounding box with the other half of the list it came from. Once this has been repeated recursively a SBT BVH has been created. On figure 5.1 the subdivisions are illustrated as dotted lines.

Another tree called minimal space tree (MST) is generated by selecting the two objects, bounding boxes or surfaces, that create the smallest area and putting them into a bounding box. This is repeated until there is only one bounding box left, at which point you have a MST BVH. The scene from figure 5.1 in a MST BVH can be seen in figure 5.2.

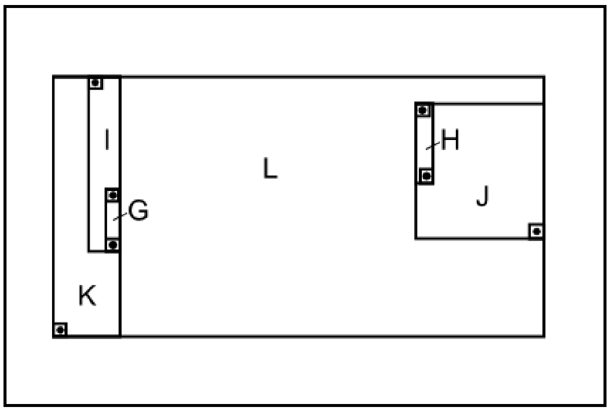


Figure 5.2: Result from creating a bounding volume hierarchy with the MinimalSpaceTree algorithm.

TheMatrixDistributed uses a class called `BoundingNode` which has a right and a left node used to create the binary tree. The right and left fields holds a pointer to a `Surface` or a `BoundingNode`, depending on the values of the fields `leftIsLeaf` and `rightIsLeaf`.

TheMatrixDistributed can generate both MST and SBT hierarchies. The MST is generated by going through a list of surfaces and finding the two nodes or surfaces which stretches the smallest volume and then create a bounding box around them. The new bounding box will then be a new `BoundingNode` and will be added to a list of nodes. The two surfaces or BoundingNodes will be removed from the list of surfaces. This continues until the list of surfaces is empty and there only is one `BoundingNode` in the list of bounding nodes. This node will then be the root of the tree.

TheMatrixDistributed generates SBT's using the method
`SplitSurfacesInBoundingBox` to split the scene into two.
`SplitSurfacesInBoundingBox` takes a list of objects and an axis of operation as a parameter, and then finds the maximum and minimum values, of the objects, on the axis of operation. Once the maximum and minimum are found, the difference between these is divided by two and added to the minimum value, which gives the split value. Then the list of objects is split into two depending on whether or not an object is less than or greater than the split value on the axis of operation.

Algorithm 1 MinimalSpaceTree

```
Pick an arbitrary Surface
while number of Surfaces $> 0$ and number of BoundingNodes $> 1$ do
  Find the two Surfaces or BoundingNodes that stretches the smallest volume
  Create a new BoundingNode and add it to the list
  Delete the two old Surfaces or BoundingNodes from the list
end while
```

Then `SplitSurfacesInBoundingBox` is called once for both lists with a new axis of operation, and the two resulting `BoundingNode`s from this call is joined in a `BoundingNode` which is returned. Through this recursion an SBT tree is quickly built.

Algorithm 2 Space balanced tree generation

```
if list of surfaces only contains one then
  Return the surface
end if
$max = -\infty$ and $min = \infty$
for all $S$ in list of surfaces do
  if $S.max[axis] > max$ then
    $max = S.max[axis]$
  end if
  if $S.min[axis] < min$ then
    $min = S.min[axis]$
  end if
end for
Calculate the split value $split = \frac{max-min}{2} + min$
for all $S$ in list of surfaces do
  $mid = \frac{S.max[axis]-S.min[axis]}{2} + S.min[axis]$
  if $mid > split$ then
    $right \cup \{S\}$
  else
    $left \cup \{S\}$
  end if
end for
Create new BoundingNode $N$
$N.left =$ SpaceBalancedTree(left, next axis)
$N.left =$ SpaceBalancedTree(right, next axis)
Return $N$
```

The two different methods do not create the same hierarchy as seen in figure 5.3, where SBT is deeper than the MST. But this is not the only difference as the complexity of their generation algorithms are also different. SBT runs through all the objects first, and subsequently runs through the half of the previous objects, which leads to a complexity of $O(nlog(n))$, where n is the number of objects. MST first encloses all objects in a bounding box, and then checks all bounding boxes to find the two that will create the smallest volume and encloses these two in a new bounding box. Next it checks $n - 1$ bounding boxes to find the two that gives the smallest volume and encloses these in a new one. After every iteration there is one bounding box less to check which leads to a complexity of $O(n!)$.

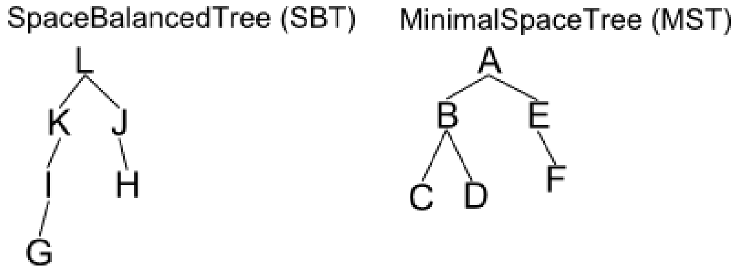


Figure 5.3: A BVH of figure 5.1 and 5.2.

These complexities can be used to evaluate which method to use. It is clear that $n!$ will grow much faster than $nlog(n)$ with an increase of n. As an example $10! = 362800$ where $10 \cdot log(10) = 23.03$. Based on this growth, SBT will in most cases be the best method, as the increase in computational time is small for an increase in n. However, not all hierarchies in TheMatrixDistributed must be generated at runtime, thus MST's may still be useful. An evaluation of the performance of the two trees will be conducted in chapter 6.

5.5.2 Searching trees

A BVH will lower the required number of intersections because some objects can be excluded from the check. If the MST hierarchy, from figure 5.2, is used the ray, r, is first checked for intersection with A. If it does intersect A then r is checked for intersection with B and E. If E does not intersect then object F can be discarded. If B intersects, then C and D is checked for intersection, and if only C intersects, bounding box D can also be discarded, and only the objects in C must be checked for intersection. This will lower the number of intersection tests needed drastically if the scene contains many objects.

The worst-case scenario of searching through the hierarchy is that all the bounding nodes will have to be checked for intersection. Additionally all objects have to be checked for shadows, which implies a dependency on the number of objects and the number of light sources. All other functions use constant time. Therefore, the worst-case scenario when rendering a scene using a binary hierarchy is $O(n) * O(n * m) = O(n^2 m)$, where n is the number of objects and m is the number of light sources. By only evaluating the dependency of the number of objects m can be assumed to be constant. This implies a worst-case scenario of $O(n^2)$. This complexity is why deep hierarchies should be avoided, as this will require more bounding box intersections.

Big-O notation denotes the worst-case complexity, but sometimes this is not relevant because the function on average will do much better. Searching through a BVH like the above will on average do better than $O(n^2)$, however, calculating the average-case complexity is beyond the scope of this report.

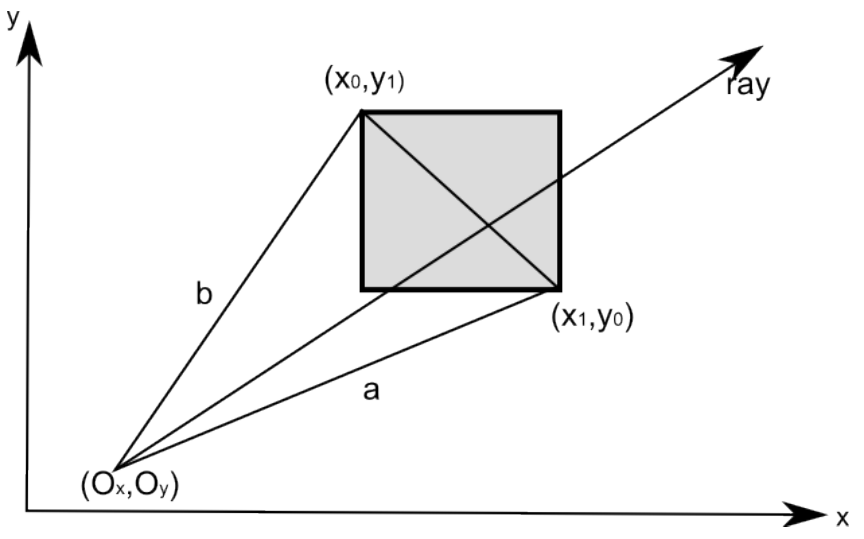


Figure 5.4: Ray intersection in the xy-plane

5.5.3 Bounding box intersection

In TheMatrixDistributed two different methods of calculating bounding box intersections have been implemented.

Eisemann slope intersection method

The first method is proposed in the paper Eisemann et al. [2008] and calculates bounding box intersections using ray slopes. First the rays are classified based on the sign of the components of their direction; if they are positive, negative or zero, i.e. a ray with the direction $(1, -1, 0)$ will be classified PMO (plus, minus, zero). Each possible classification will need its own function. Because a ray can at most hit an AABB on three sides, the problem of intersection can be reduced to test if the 2D projected ray, hits the three faces in 2D. If it hits all three the ray intersects with the AABB. Now the problem has been reduced to 2D. See figure 5.4 where a PPP ray is tested for intersection in the xy-plane.

For the ray to intersect with the box it has to cross the line going from (x_1, y_0) to (x_0, y_1). To determine if the ray intersects with this line the slope of the ray is calculated and compared with the slope of a and b. If the slope of the ray is less than b and greater than a the ray will intersect with the box. The slope of a and b can be computed by using vector calculations.

$$a_{slope} = \frac{y_0 - O_y}{x_1 - O_x} \tag{5.3}$$

$$b_{slope} = \frac{y_1 - O_y}{x_0 - O_x} \tag{5.4}$$

The slope of the ray in the xy-plane can be calculated from the rays direction.

$$Ray_{XYslope} = \frac{\mathbf{D}_y}{\mathbf{D}_x} \tag{5.5}$$

In order to reduce the total number of computations needed, it is more practical to determine if the rays miss the AABB rather than calculating if they hit. This is because if just one of these equations fails, the ray will miss and no further calculations are needed.

To check that the ray is not within the slopes of a and b a simple comparison can be done by comparing (5.3) and (5.4) with (5.5).

$$Ray_{XY\,slope} < \frac{y_0 - O_y}{x_1 - O_x} \tag{5.6}$$

$$Ray_{XY\,slope} > \frac{y_1 - O_y}{x_0 - O_x} \tag{5.7}$$

To minimise the calculations needed equation 5.6 can be reduced to (5.8)

$$\begin{aligned} Ray_{XY\,slope} \cdot (x_1 - O_x) - y_0 + O_y &< 0 \\ Ray_{XY\,slope} \cdot x_1 - y_0 + (O_y - Ray_{XY\,slope} \cdot O_x) &< 0 \end{aligned} \tag{5.8}$$

Equation 5.8 determines if the ray misses on one side of the AABB as it will only be true if the slope of the ray is less than the slope of a. Another equation similar to (5.8) will have to be made in order to determine if the slope of the ray is greater than the slope of b. The easiest way of doing this is by changing the order of x and y, and the slope is therefore calculated as in equation 5.9.

$$Ray_{YX\,slope} = \frac{\mathbf{D}_x}{\mathbf{D}_y} \tag{5.9}$$

Then a comparison can be deduced in the same way as (5.8).

$$\begin{aligned} Ray_{YX\,slope} \cdot (y_1 - O_y) - x_0 + O_x &< 0 \\ Ray_{YX\,slope} \cdot y_1 - x_0 + (O_x - Ray_{YX\,slope} \cdot O_y) &< 0 \end{aligned} \tag{5.10}$$

If both equations 5.8 and 5.10 fails, the ray must be within the box in the xy-plane, however, as there are three sides in an AABB, four other equations similar to (5.8) and (5.10) need to be checked to fully determine if the rays hits. However if just one of the equations fail the ray will miss and the function will break. Note it must also be checked that the origin of the ray is not behind the bounding box, before it can be concluded that the ray intersects the bounding box.

Fast intersection

The second method to calculate AABB intersection has been proposed in the article Williams et al. [2005]. It works by narrowing down the minimum and maximum distance from the ray origin and to where the ray enters and exits the box. First the distance to where the ray intersects with the lowest and highest values on the x-axis of the AABB is calculated as shown in (5.11).

$$\begin{aligned} tmin &= \frac{boxmin_x - O_x}{D - x} \\ tmax &= \frac{boxmax_x - O_x}{D - x} \end{aligned} \tag{5.11}$$

The values on the y-axis is found in the same way, as in equation 5.12

$$\begin{aligned} tymin &= \frac{boxmin_y - O_y}{D - y} \\ tymax &= \frac{boxmax_y - O_y)}{D - y} \end{aligned} \tag{5.12}$$

If $tmin > tymax$ or $tymin > tmax$ the ray will not intersect with the AABB as the minimum distance of t on one axis can not be greater than the maximum distance on the other axis when the ray intersects. If so the function will break. If the function does not break it will find the largest minimum value, $tmin$, and the smallest maximum value, $tmax$, of the two axes. Then the distance to the z-axis is computed in the same way as equations 5.11 and 5.12. Again it is checked if the minimum distance is greater that the maximum - so if $tmin > tzmax$ or $tzmin > tmax$ the function breaks. If this is not true the ray must intersect with the AABB. The function can continue and again find the largest minimum value and the smallest maximum value. The minimum and maximum values will be the distance to the entrance and exit points of the AABB respectively.

5.5.4 Bounding box intersection benchmarks

Fast intersection and Eisemann slope intersection uses different methods to intersect with an AABB and therefore the number of calculations and comparisons the CPU has to make in order to calculate the intersection also differs. As both methods can break several places it is not possible to calculate the exact number of calculations and comparisons used, as these can change.

In the best case scenario Fast intersection requires four additions, four multiplications and one comparison to break and return no intersection. The Eisemann method on the other hand will in the best case scenario break after just one comparison. However the worst case scenario for the Eisemann method has twelve additions, six multiplications and nine comparisons. The worst case scenario for Fast intersection on the other hand only has six additions, six multiplications and ten comparisons.

Both methods have been tested with different objects in TheMatrixDistributed, and based on tests it was found that the Fast intersection method was a little faster than the Eisemann method. In a scene with 12000 triangles Eisemann slope intersection provided an average of 0.302 FPS, and in the same scene Fast intersection provide an average of 0.382 FPS. This might have something to do with it being unlikely for the Eisemann method to break early in the algorithm and Fast intersection having less calculations in its worst case. However to choose one method over the other deepens on many factors such as the compiler, the CPU and the scene rendered. One extra advantage of Fast intersection is that it also returns the distance to the intersection which can be further used to optimise the code. Therefore TheMatrixDistributed uses Fast intersection as the default AABB intersection method.

5.6 Parallelisation

The following section describes aspects of parallelisation.

Parallelisation is the process of performing a computation by executing commands in parallel, i.e. at the same time. For instance computing the sum of 10 numbers can be computed by executing nine additions, however, if it is possible to execute two commands at once, as it is on an SMP (symmetric multiprocessing) machine, one could execute two additions at once four times and then do a final addition to get the sum of the two subsums. Most modern computers offers SMP, which means that they have multiple cores and can execute multiple commands, or instructions as CPU commands are called, at once.

This means that in order to utilise all the cores of the CPU, a computation must be split into multiple sequences of operations, which must be independent of each other so that they can be executed at the same time. To split a ray tracer into multiple sequences of operations is rather trivial, for instance each frame could be split in two and the parts could be computed by two different sequence of operations at the same time.

On modern operating systems there are two different approaches to executing two sequences of operations simultaneously. The first approach is to use multiple processes - the second approach is to use multiple threads within a single process. The difference is that a process has its own isolated memory. So when parallelising using multiple processes, communication between processes can be complicated. IPC (inter-process communication) often happens through files, pipes or sockets, whereas communication between threads is simple because all threads have access to the same memory within a process. However, if a thread crashes, the entire process crashes, which is not the case for processes, i.e. one process does not necessarily crash because its subprocess crashes, however, it can still deadlock. It should also be noted that processes are more expensive to create than threads, because they have their own address space, etc. Threads is used in TheMatrixDistributed for faster direct memory access, and because TheMatrixDistributed should not be allowed to continue if a thread crashes.

5.6.1 Thread synchronisation

When accessing the same memory from multiple threads it is important to ensure that two threads never access the same memory simultaneously for other purposes than reading, e.g. one thread may not write to the same memory which another thread is accessing. If this happens the read value cannot be predicted. For this reason it is necessary to restrict memory access, for instance using locks or semaphors as specified in POSIX.

Locks

A lock is a simple variable, `mutex_t`[5], that can be locked and unlocked. When it has been locked it cannot be locked again before it has been unlocked. So a variable can be protected from simultaneous access by putting a lock statement before it is accessed, and an unlock statement after the access, as sketched in example 2.

Example 2 Using a lock to synchronize access to a variable.

```
pthread_mutex_lock(p_data_lock);
data = something_I_computed;
pthread_mutex_unlock(p_data_lock);
```

A common bug that occurs when using locks is that threads can deadlock. If for some reason the thread tries to acquire a lock that is already locked, and something prevents the program from releasing the lock, the thread will simply lock, and the program may stop responding. This could for instance happen if a goto statement causes an unlock statement to be neglected in certain circumstances. It could also happen if two threads acquire the same two locks at once but in a different order, as illustrated in example 3, which might only deadlock in very rare occasions, thus making it hard to find the bug.

Example 3 Two threads that may deadlock if they are locking their first lock simutanously.

```
void* thread_one(void* args){
pthread_mutex_lock(p_data1_lock);
pthread_mutex_lock(p_data2_lock);
//Do something with data1 and data2
pthread_mutex_unlock(p_data1_lock);
pthread_mutex_unlock(p_data2_lock);
}
void* thread_two(void* args){
pthread_mutex_lock(p_data2_lock);
pthread_mutex_lock(p_data1_lock);
//Do something with data1 and data2
pthread_mutex_unlock(p_data2_lock);
pthread_mutex_unlock(p_data1_lock);
}
```

Semaphors

A semaphore is a lock with an internal counter. The function `sem_post` increments this internal counter by one, and `sem_wait` decrements the internal counter by one, if the internal counter is larger than zero. If the internal counter is zero `sem_wait` waits for the internal counter to be incremented, in the same way that a lock statement waits for the lock to be unlocked [IEEE and Group, 2004]. Semaphores are sometimes easier to use than locks, and they are for instance used to synchronize an unknown number of threads in TheMatrixDistributed.

[5]A mutex as specified by POSIX has other features too but these are beyond the scope of this project.

5.6.2 Ray tracing in parallel

As previously mentioned it is easy to parallelise ray tracing; all that is necessary is to split the frame in two and compute the parts on two different threads. However, some parts of the frame is faster to render than others, so simply splitting the frame in two parts, launch a thread for each part and wait for the threads to finish is not optimal. A notable performance enhancement was gained by letting the worker threads compute every n^{th} line, starting from different positions, n being the number of threads that TheMatrixDistributed is compiled with. This approach was better, nevertheless, locking access to a counter, counting which line to compute next, turned out to be slightly faster[6] on a dualcore machine.

5.7 Serialisation

The following section describes the serialisation used for networking.

Serialisation is the process of converting an object to a data stream, from which the object can be restored. This can be used to save objects between sessions, parse objects between processes or transmit objects over a network. Methods to serialise objects are present in most modern frameworks. Libraries that offer serialisation are also available for C++, however, to control and minimise overhead TheMatrixDistributed has its own serialisation system implemented.

The goal of the serialisation system is to be able to serialise and transmit a complete scene over the network. Also, serialisation and deserialisation should have a minimal overhead, both in terms of memory and CPU usage. The serialisation is therefore done in binary. Alternatively serialisation could have been done in a human-readable format, XML for instance. This would be a lot easier to debug, as a serialised object would be readable in a text editor. However, human-readable formats, such as XML, integers, floats, enums etc. are often converted to a string format, and this is obviously not space efficient. Human-readable formats are also more complicated, and thus more CPU intensive to read and write, than simply copying the byte values. For these reasons we have chosen to do binary serialisation, because we consider performance more important than easy debugging and portability. The serialisation was therefore done in binary, because performance was rated higher then easy debugging and portability.

We are well aware that our current serialisation system does not encourage portability beyond x86. For instance we copy integers and floats as they are in memory. We do not convert integers to a network byte order, instead we serialise them to the host byte order. As our computers are running x86 this means that we are actually storing integers in little-endian [Int, 1999, section 1.4.1: Bit and byte order]. The difference between little-endian and big-endian[7] is the byte order, see figure 5.1 for an example of a 32-bit unsigned integer. Not converting to a network byte order does seriously hinder portability beyond x86. However, TheMatrixDistributed was not intend to be distribute to any other architectures, so this will not become a problem. If

[6]Improving FPS from 3.23 to 3.31 measured over the course of 90 frames.

[7]Big Endian is often used as the network byte order

it was to become a problem, macros for conversation between host and network order are available as a part of the socket API.

Byte number	0	1	2	3
Little endian	0x04	0x03	0x02	0x01
Big endian	0x01	0x02	0x03	0x04

Table 5.1: The integer 16909060 in little- and big-endian representation.

As mentioned earlier serialisation is done by copying memory from objects to a byte stream. However, since it is not necessary to serialise all the fields of an object in order to restore it, e.g. buffered values need not be serialised, and some fields cannot be serialised, for example pointers, TheMatrixDistributed cannot simply read an object as a byte array and copy it to byte stream. Instead it must decide which fields of an object need to be saved and which do not, and when deserialising it must recalculate buffered values and initialise pointers. For this purpose, serialisable classes have a method called ″serialise″ and a constructor that takes a byte array as its argument, along with any pointers that needs to be initialised with external values.

In TheMatrixDistributed all serialisable classes inherit from the abstract class `Serializeable` (See figure 5.5). The solid arrow from e.g. `Scene` to `Serializable` on figure 5.5 indicates that `Scene` derives from `Serializeable`.

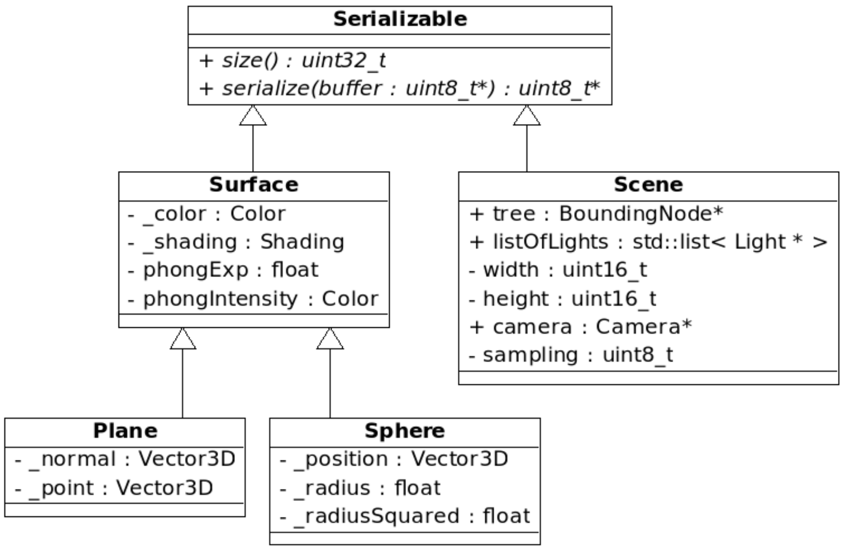


Figure 5.5: Serializable and a few classes implementing it.

As can be seen on figure 5.5 `Serializeable` defines the methods `uint8_t* serialize(uint8_t* buffer)` and `uint32_t size()`. The `size` method returns the buffer size necessary to serialise the object. The `uint8_t* serialize(uint8_t* buffer)` method serialises the object to a byte array and returns the byte array pointer incremented to point of the next unused byte. This makes it easy to allocate a large buffer for multiple objects and serialise them to the buffer one by one. However, in order to deserialise a large buffer with multiple objects, it is necessary to know the size of each serialised object and its type. Thus every serialised object starts with a `SerializedData` structure as can be seen on figure 5.6 below. The solid diamond denotes composition, i.e. that the field `type` of `SerializedData` is the enum `ObjectTypeIdentifier`.

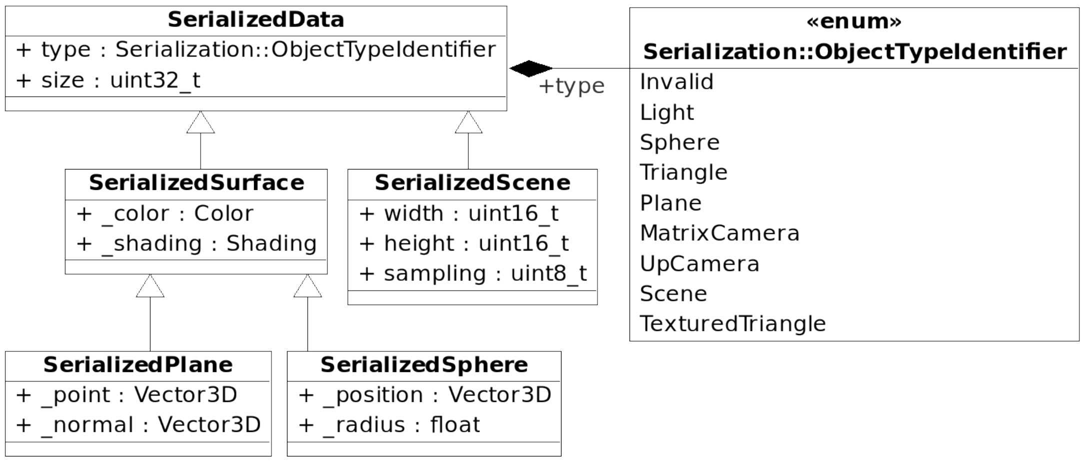


Figure 5.6: Serialisation structures.

When doing the actual serialisation of an object inside `serialize`, the buffer pointer is cast to a pointer to a serialisation structure that derives from `SerializedData`. Then the fields on the structure is set. Some of the serialisation structures can be seen on figure 5.6. Using `sizeof()` to find the size of the particular serialisation structure in `size()` ensures that the requested buffer size matches the size of the serialisation structure which the buffer is cast to in `serialize`. For some classes, such as `Scene` and `BoundingNode`, the size is variable, and in these cases serialisation can be more complicated. However, the format is the same - first the size needed for serialisation is estimated, then the object is serialised.

5.8 Distributed rendering

The following section describes how TheMatrixDistributed distributes the rendering process.

In order to gain more computation power than what is available on a single computer, the rendering can be distributed to multiple computers. TheMatrixDistributed uses TCP/IP[8] for communication between server and clients. In order to distribute the rendering, the scene to be rendered must be available to all clients. TheMatrixDistributed serialises the scene once for each frame and transmits it to all the clients, along with two segments of the frame that the clients should render. A diagram of the communications between server and client can be seen on figure 5.7.

When a client has rendered a segment of the frame it transmits this segment to the server and starts computing the next segment it has waiting. If no segments are

[8]TCP/IP: Transmission Control Protocol on top of the Internet Protocol.

waiting, the client waits for the server to send one. When the server receives a rendered segment from a client, it stores the data in the framebuffer and sends a new uncomputed segment to the client. Two segments are initially given to the client to ensure that the client can continue rendering while the server is responding with a new segment.

This minimises the impact of the network latency, because the client does need to wait for the server to response to start computing, as it already has a segment waiting for computation. Also dividing the frame into many segments and assigning these to the clients as they are computed is a good idea, because different parts of a frame may not take the same time to compute nor can we expect the clients to operate at the same speed, thus dividing the entire frame among all client initially is not a good solution.

The server always keeps a list of requested segments, this way, when there are no more new segments it can request a segment that another client is already working on. This makes the server robust and ensures that it will not deadlock if a client suddenly disconnects. It also ensures that if one client is a lot slower than all the others, then all the other clients will not go idle waiting for the slowest client to compute the last segment. But this robustness comes at a price, as all clients will most likely be computing unnecessary segments at the end of each frame. However, it is possible to minimise this cost by checking whether a new scene has been received during computation of a segment in the client, then aborting the computation and discarding the computed pixels instead of finishing the segment it started.

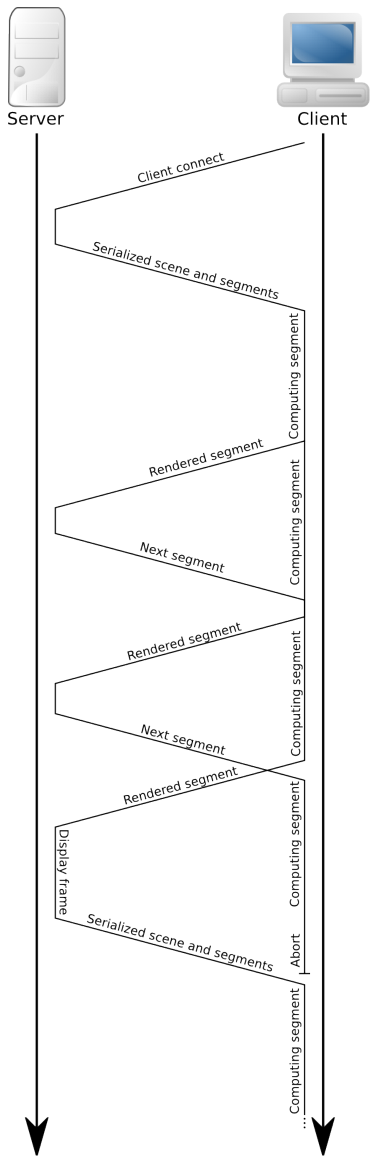


Figure 5.7: Overview of server and client communication.

5.8.1 Server/client protocol

TheMatrixDistributed's server and client communicate using a very simple binary protocol with three types of packages: update, compute or data. Each package consists of a header and a payload. The header is 8 bytes: 4 bytes to determine the type of the package and 4 bytes to indicate the total size of the package. A list of the different packages, their size (or minimum size), payload and direction, i.e. from server to client or from client to server, can be seen in table 5.2.

Type	**Size**	**Payload**	**Direction**
update	> 24	Serialized scene $+$ 2 segments	Server $\rightarrow$ Client
data	> 16	Segment $+$ computed pixels	Server $\leftarrow$ Client
compute	16	Segment	Server $\rightarrow$ Client

Table 5.2: Package types, sizes and payloads.

5.8.2 Frame segmentation

A frame is split into multiple segments by letting the first n lines be the first segment and the next n lines be the next segment and so on. This way the frame is split in $\frac{h}{n}$ segments where h is the height of the frame. The best segment size n depends on the height of the frame and the number of clients connected to the server. Because too few segments causes some clients to compute the same segment and helps account for the fact that segments are not computed at same pace, because clients may have different hardware and some segments are easier to compute than others, e.g. contains more empty space. On the other hand too many segments causes more network traffic and thus drives more network overhead. This along with a simple technique for minimizing the impact of a bad segment size choice is discussed in the section 6.2.

5.9 Class hierarchy

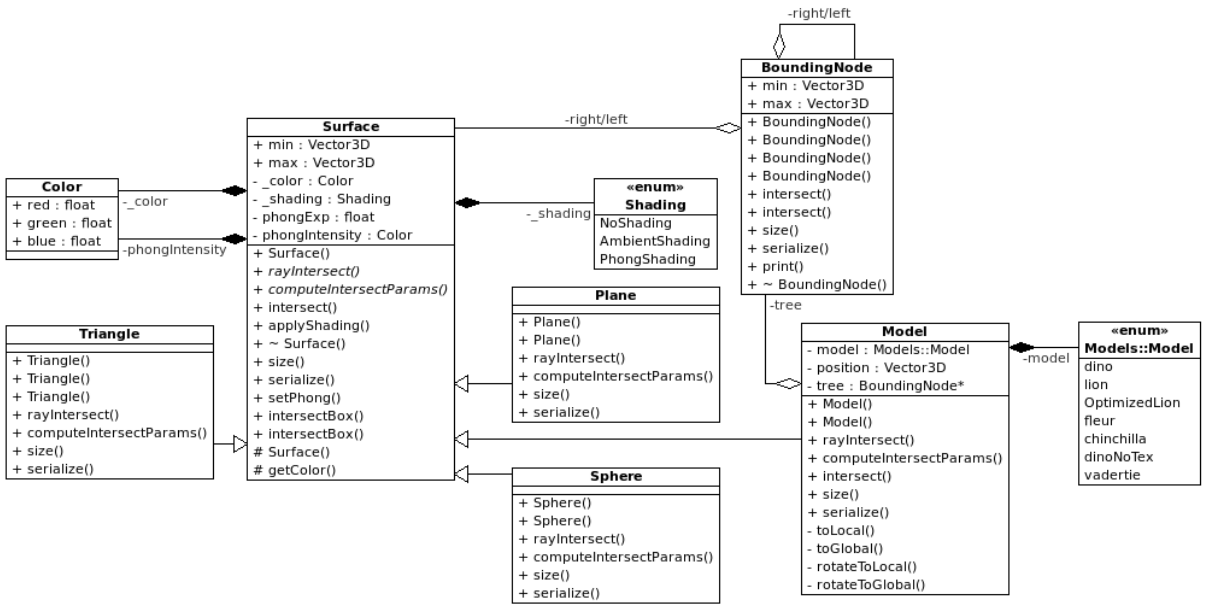
A description of the important and interesting aspects of TheMatrixDistributed structure.

An object orientated programming language was desired for TheMatrixDistributed because primitives in a ray tracer are easily derived from a common base class. This makes it easy to implement new primitives and extend the ray tracer relying on polymorphism to avoid changing code other places in the ray tracer. TheMatrixDistributed is written in C++, and this section outlines the class hierarchy and discusses the cost/benefit aspects of extensibility in terms of performance. Also note that even though classes are often used to hide members, e.g. fields and methods, from public access in order to improve usability of the code, this has not been a concern in TheMatrixDistributed. Fields are often private, as this is good practice, however, this has not been done consistently and whenever performance could suffer good practice has been disregarded. Additionally the usability of the classes has not been as big a concern as making the code fast.

5.9.1 Surfaces

All objects within a scene in TheMatrixDistributed that can be hit by a ray must be represented as an implementation of `Surface`. This makes `Surface` the base class for all primitives, e.g. the class `Triangle` derives from `Surface`. As previously mentioned polymorphism enables the ray tracer to treat any primitive as a `Surface`, which makes it easy to add a new primitive without changing all the other parts of the ray tracer.

Figure 5.8: Class diagram of surface and classes that derives from it.



The classes that derive from `Surface` can be seen on figure 5.8, where the solid lines with arrows denotes derivation, the lines with a solid diamond denotes composition and the lines with a hollow diamond denotes aggregation. Composition and aggregation are both 'has a' associations, i.e. one type has another type, but aggregation is not as strong as composition which implies that the type contained is destroyed with the destruction of the container. E.g. `Surface` has a `Color` field, which is destroyed with `Surface`, thus a composition, and `BoundingNode` may have a `Surface` which is not necessarily destroyed when the `BoundingNode` is destroyed, thus aggregation. In C++ an aggregation could be a pointer, and a composition would be a field.

5.9.2 Intersecting and shading a surface

As can be seen in figure 5.8 `Surface` has four methods to compute intersections and a method to apply shading. The method `rayIntersect(const Ray &ray, IntersectionComputation &computation) const` determines if there is an intersection between the ray and an instance of a derived `Surface`. The method may not alter any fields on the `IntersectionComputation` that is passed as argument, unless the distance to an intersection with the ray is shorter than the `distance` field on the `IntersectionComputation`. If the distance is the shortest, the `distance` field and the `surface` field on the `IntersectionComputation` must be set. The `distance` field must be set to the distance to the intersection and the `surface` field must be a pointer to the `Surface` that is intersected, i.e. in `rayIntersect` the C++ keyword `this` would be appropriate. `rayIntersect` may modify all the other fields of `IntersectionComputation`, except the field `sub_surface` which may only be used by `Model`.

When the `rayIntersect` method works as described above the closest `Surface` can be found by calling `rayIntersect` on all `Surface` objects. Using bounding volume hierarchies, as discussed in section 5.5, this can be reduced to a set of candidate objects. Notice that it is possible to do these calls with the same `Ray` and `IntersectionComputation`, since the `Ray` parameter is marked `const`, thus when this has been done the `surface` field on the `IntersectionComputation` is a pointer to the closest object, given that such an object is found. By setting `distance` to infinity, as defined for IEEE floating points, any distance an intersecting object finds will be smaller than the initial value of the `distance` field [P754, 2008].

When the closest intersected object has been found, the intersection point and normal vector of the object can be found using the method `computeIntersectParams`, this method sets the `position` and `normal` fields on `IntersectionComputation` to the intersection point and the normal vector of the object in the intersection point respectively.

Both the `rayIntersect` and the `computeIntersectParams` method are not implemented in `Surface` and an implementation must be provided for each derivative. However, the shading method `applyShading` is not virtual and thus implemented on `Surface`. This method is used to apply a shading to the color of the object. Derivatives of `Surface` may provide the desired shading as a parameter to the constructor of `Surface` along with the desired color. If a solid color is not desired, derivatives may overwrite the protected method
`Color getColor(IntersectionComputation, Scene) const`. For instance the implementation of `getColor` in `Surface` is overwritten in `Triangle` in order to support textures.

Derivatives of `Surface` can also overwrite the
`bool intersect(Ray, float length, Surface* exluded) const` method that is used to determine if an object is intersected by a shadow ray. A shadow ray is a line between an intersection point and a light source, used to determine if an intersection point is in shadow from a given light source. The `exclude` parameter on the `intersect` method is a pointer to the object that is being shaded, and ensures that no primitive can shadow itself due to rounding issues. Alternatively a small value can be added to the length and spawn a ray from the light to the intersect point. However, since there are no complex primitives in TheMatrixDistributed that need to shadow themselves, this is not an issue, and this approach saves an intersection with the object being shaded. By overwriting the `intersect` method it is possible for an object to provide a faster method for line intersection, compared to using `rayIntersect` as the implementation of `intersect` in `Surface` does.

5.9.3 Models

TheMatrixDistributed has a special derivative of `Surface` called `Model` which can hold other derivatives of `Surface` in a local coordinate system that can be moved and rotated relative to the global coordinate system. This is achieved by overwriting the intersection methods described in section 5.9.2 to intersect with a new ray in an internal bounding volume hierarchy of the `Model`. The idea is to make it cheap to load complex models by precomputing the bounding volume hierarchy. A precomputed model is stored as a .tmd file and is just a serialised bounding volume hierarchy, i.e. a single instance of `BoundingNode` that has recursively serialised its children. These model files, or .tmd files, can be generated using the command line program called obj2tmd that partially supports .obj files.

When TheMatrixDistributed is started, all the models that can be used are loaded from .tmd files, i.e. the bounding volume hierarchy of the models are deserialised. These deserialised hierarchies are stored in an array and passed to `Model` when it is constructed. Depending on which model the instance of `Model` is asked to create it sets its private pointer `tree` to point to the deserialised hierarchy. This means that it is possible to create two instances of `Model` using the same model file without loading the actual model twice - actually the memory is not even copied once for each instance.

Model rotation and displacement

When a ray is passed to the `rayIntersect` method of the `Model`, a new ray is computed based on the rotation and displacement of the model. The displacement is computed by subtracting the displacement vector from the position vector of the ray. The rotation is computed using the simple rotation matrix that can be seen in equation 5.13, for rotation around one axis. Rotation around 3 axes could easily be supported, however, it would be more expensive and in most cases not necessary. Nevertheless, the matrix rotation described in section 4.2.4 on the camera could also be used here, if rotation around 3 axes was desired.

$$\begin{bmatrix} x_{rotated} \\ y_{rotated} \\ z_{rotated} \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{5.13}$$

Once the position of the ray has been displaced and rotated, using equation 5.13, the direction vector of the ray is rotated as well. If an intersection is found in `rayIntersect` on a `Model`, a pointer to the intersecting object is stored as the `sub_surface` pointer on the `IntersectionComputation` and the `surface` pointer is set to the model that holds the intersected object. Then when `computeIntersectParams` is called on `Model` it calls the same method on the `sub_surface` and rotates and displaces the results before returning. Note that it also sets the `surface` pointer on the `IntersectionComputation` to point to the intersecting object, so that shading and color is applied correctly.

Obj2tmd

Obj2tmd can currently only load .obj files. It does not however fully support the .obj extension, and is only capable of loading polygonal objects. Furthermore obj2tmd will not read any normal vectors that are included in the object file. Obj2tmd does not fully support the material integration of the .obj standard. This section will therefore only describe the elements that are supported by Obj2tmd.

For polygonal objects, the obj format contains vertex data, vertex texture data, face normals and finally faces. The obj format contains the filename for the material file as one of the first lines of the file with the prefix `mtllib`. All lines with the `#` prefix are comments. Furthermore the obj format splits the different objects in the file with the `o` prefix.

Example 4 The following is an example of a .obj polygonal object.

```
mtllib filename.mtl
o Object_001
#The vertices:
v 1.000342 2.000543 1.234083
v ...
...
#The texture coordinates
vt 0.332789 0.1
vt ...
...
#Face normals
vn 1.234876 5.993321 6.423498
vn ...
...
#Faces
f 1/3/7 8/5/3 9/1/5
f ...
...
usemtl filename.ext
f 43/32/56 12/56/33 34/654/3
f ...
...
```

The format is fairly simple. The vertices comes first, one per line and they are prefixed with a `v`. The vertices are represented by 3 float values: x, y and z which are separated by a single space. Next is the vertex texture coordinates, also known as the uv-coordinates. The uv-coordinates begin with the prefix `vt` and may only be followed by 1 to 3 floats separated by a single space. The first two floats represent u and v respectively for two dimensional(2D) mapping. The final float is w, which represents depth in a 3D texture mapping. Not all files contain uv-coordinates as some models may not have textures.

Next in the file is the vertex normals which are prefixed `vn`. The vertex normals are 3 floats representing a vector, with the components i, j and k again separated by a single space. obj2tmd does not support normals from obj files, and will ignore them.

Finally the faces are listed in the file, and are prefixed with `f`. A face can have up to 12 integer values, 3 values per vertex and 4 vertices. The values for each vertex are separated by a slash. Normally programs that export to .obj possess the ability to triangulise all polygons. However, Blender occasionally has trouble doing this, especially when converting curves and curvey surfaces. It will then occasionally export a face with 4 vertices, which obj2tmd is also capable of interpreting. The integers represents the vertex number in a list, starting from 1 at the first vertex. So in the case of the example 4, the first face is a reference to the 43rd vertex, the 32nd uv-coordinate and the 56th vertex normal as its first vertex. Each vertex is to be counted separately. If there are no uv-coordinates or vertex normals then the faces will contain no slashes, however, if there are vertex normals, but no uv-coordinates the face will contain two slashes as if there were uv-coordinates. Finally the face will contain one slash if there are uv-coordinates, but no vertex normal coordinates.

In files with material associations the prefix `usemtl` may occur between faces. This denotes when a new material should be used. The parser will load the material file, and store relevant Phong and colour data for each material. However, the parser will ignore images and other information in the material file, as TheMatrixDistributed does not support effects such as optical density. The values are then given to the triangles, when the prefix `usemtl` is encountered. This allows for coloured models to gain their intended colour, which is often useful for simple models where a separate texture would be unnecessary.

At the end of the faces a new object may begin, in which the format starts over with vertices. The prefix for an object is as mentioned `o`, however, obj2tmd does not distinguish between several objects in one file. While we are not familiar with 3D modeling or animation, it is speculated that this is to allow individual members or part movement in complex objects during animations [Bourke].

5.10 Optimisation

This sections discusses techniques for optimising C++ code.

5.10.1 Inlining

When a function[9] is called in binary code it has an overhead. This is because the processor must jump within the program. However, this overhead can be completely removed by placing a copy of the function where it is called, instead of actually calling the function. If the function is called only once this should be faster than calling the function, as this removes the overhead of a function call. However, if a function is called more than once, this might not be faster because it would increase the size of the binary. So this is only faster for small functions or functions that are only called a few times.

This optimisation technique is called inlining or inline expansion. Inlining can be done at the discretion of the compiler or linker, or a function or method can be marked `inline` in C++, which can force the compiler to inline a method or function. In TheMatrixDistributed all the methods for the vector class have been inlined, which resulted in a huge performance improvement. Though these methods are called often, they are very small, e.g. a method like addition is only three operations, but final binary code does become much larger when these methods are inlined. Note that due to the nature of this optimisation, i.e. the function call is replaced with the function body, it is not possible to inline recursive methods.

[9]This applies to methods as well.

5.10.2 Virtual methods

A virtual method is a method that can be overwritten in a subclass. For this to be possible an object that has a virtual method must have some way of determine if its virtual function have been overwritten and if so where the new function can be found. In C++ this is solved by using a vtable (virtual table) that contains pointers to the implementation of the virtual methods. Then when a method is overwritten its entry in the vtable is overwritten to point to the new implementation of the method. This means that it is more expensive to call a virtual method compared to calling a static method. For this reason it makes sense to minimize the number of virtual methods. This is why `BoundingNode` does not inherit from `Surface` in TheMatrixDistributed, instead `BoundingNode` has a boolean for each children to determine whether or not one of its children is a `BoundingNode` or `Surface`. Inheriting from `Surface` would also require `BoundingNode` to implement some other functions that should never be used anyway, e.g. `getColor` would not make sense for a `BoundingNode`.

5.10.3 Profiling

When optimising an application it can be important to know what parts of the program that consumes most CPU time. Such information can be found by profiling an application. Profiling is done by giving the compiler an argument that tells it to generate a profile when the application is executed, and once the application is executed a profile will be generated. Using GCC and gprof will easy generate a list of methods and functions of how many times these were called and the amount of runtime spend in these functions. This can be useful for deciding which parts of an application that optimisation should be focused on. It can also be used to determine whether or not to inline a method or just in general to see if one approach to solving a problem is faster than another approach.

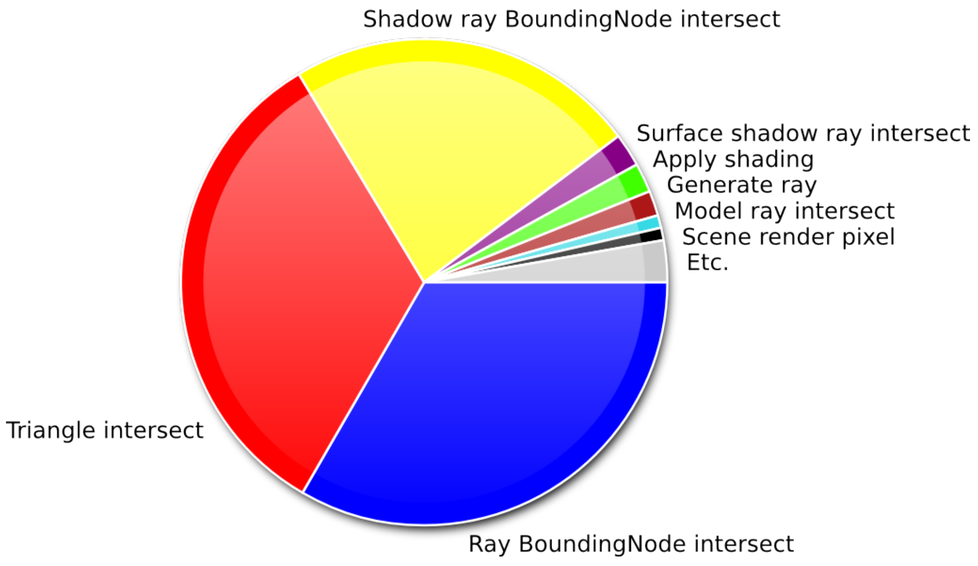


Figure 5.9: Profiling results of TheMatrixDistributed.

Figure 5.9 shows a profile of TheMatrixDistributed rendering a scene with a single light source and looking closeup on a complex model with about 95,000 Phong shaded triangles. The results of this profile does indeed depend on what is being rendered, but this setting was considered a realistic task for a real time ray tracer in the gaming industry - actually more triangles would probably be necessary, but a million triangles would most probably take a long time. This setting is obviously not one that TheMatrixDistributed can handle in real time, at least not closeup, however, the purpose of this profile is to discover what limits the performance of TheMatrixDistributed and thus determine which parts of the application that should be the primary focus for optimisation. Rendering less complex scenes the `generateRay` method takes about 10% of the execution time, however, as this profile shows when rendering complex scenes `generateRay` should not be the primary focus for optimisation as it only consumes 1.6% of the execution time in this context. This profile was run over the course of 3 minutes and 40 seconds and over a billion intersections with triangles was performed during this profile.

As can be seen on figure 5.9 the intersection between bounding box and primary ray, using 33.4% of the execution time, should be the primary focus for optimisation. Notice that this profile was done with one light source, and if there had been more lights, shadow rays would probably be taking the first place instead of the third place using 23.5% of the execution time. Also note that "Triangle intersect" covers both primary ray and shadow ray intersection with triangles, which is because triangles does not provide a specialized method for shadow ray intersection. Shadow ray intersection with triangles is done using a virtual method on `Surface` that uses the ray intersection method on triangle. This causes some overhead because one virtual method is calling another virtual method, and this overhead does also account for 2.2% of the execution time. Based on this profile a small but obvious optimisation would be to implement a method specifically for shadow ray intersection with triangles. But all in all this profile shows that bounding box intersection should be the primary point of focus for optimisations as these, counting both primary and secondary rays, accounts for 56.9% of the execution time. Another interesting thing that can be seen from figure 5.9 is that the rotation of rays in `Model` is not very expensive.

Profile Guided Optimizations

Most modern compilers can use the profile of an application to optimise the code at compile time, and this is called PGO (Profile-Guided Optimisation). A scene with one light source, one plane, three spheres, four textured triangles and a complex model of about 95,000 Phong shaded triangles - the same scene as used for the profile on figure 5.9 just not closeup on the model - compiled with the Intel C++ Compiler with all optimisation activated except PGO, rendered with about 1.72 FPS. Activating PGO improved this to 1.83 FPS.

According to Int [2008] PGO can, among many other things, assist the compiler to only inline functions where the application will benefit from it. It should also improve the effect of IPO (Interprocedural Optimisation) and help the compiler arrange memory so that caching is improved. PGO should also help the compiler improve branch prediction, where this cannot be done reliably at compile time. Branch prediction is the process of guessing whether or not the body of an if-statment[10] will be executed, and fetch the next instruction so the codeflow is already in the processor pipeline. Following the conventions of Int [2009, section 3.4.1] the instructions can be ordered so that the processors branch predictor picks the branch that was considered most likely at compile time. For all purposes in TheMatrixDistributed this is only done using optimising compilers as any other approaches is beyond the scope of this project.

5.10.4 SSE Vectorisation

Many processors support some form of SIMD (Single Instruction, Multiple Data), on modern x86 processors, from both Intel and AMD, and these instructions are known as SSE (Streaming SIMD Extensions) instructions. SSE instructions allows operations to be performed on multiple data elements in one instruction. For instance two SSE vectors of four single precision floating points can be added or subtracted in one instruction. The typical SSE instruction operates on two vectors of data elements in parallel, as sketched on figure 5.10. Here $X0$ and $Y0$ are data elements of two different SSE vectors, and if the operation was addition of the first data element of the result register would be $X0 + Y0$. Most SSE instructions operates on the data elements in parallel as illustrated on figure 5.10, however, instructions that operate in parallel does also exist.

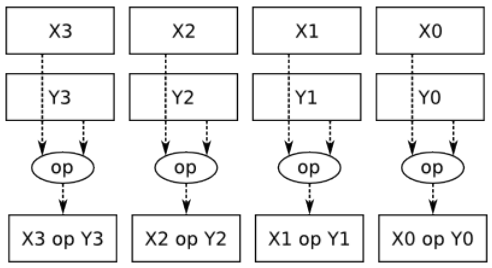


Figure 5.10: A SSE instruction operating on data elements in parallel.

It is not hard to imagine that it is possible to optimise TheMatrixDistributed by using SSE instructions for vector operations such as addition, subtraction, scalar multiplication, normalisation and computing the dot product. It is indeed possible to do these things with SSE instructions, Int [2009] has code samples to show how this could be done, and in Toth [2005] it is shown that normalisation and multiplication using SSE instructions is faster than using normal instructions. However, this does not necessarily imply that using SSE vectors to represent vectors in a 3D space is good, since an SSE vector have a size that is a power of 2 and a 3D vector in TheMatrixDistributed has a size of 3 bytes. So SSE optimising the `Vector3D` structure would change its size from 3 bytes to 4, and these would need to be aligned correctly [Int, 2009]. Considering how serialisation is done in TheMatrixDistributed this would require some large changes to TheMatrixDistributed. For this reason, and because it is hard to do, TheMatrixDistributed have not been optimised using SSE instructions.

[10]Branches occurs at gotos, loops and function calls as well.

CHAPTER 6

Evaluation of TheMatrixDistributed

This chapter presents and discusses different benchmark tests in TheMatrixDistributed. This chapter will also present the best obtained result, and discuss the resources and efforts required to do real time ray tracing.

6.1 Network throughput

A brief description of the initial networking issue.

Network throughput could be a potential bottleneck for TheMatrixDistributed. When the rendering is distributed, all segments are transmitted uncompressed, disregarding the fact that some of the segments may be computed by clients on the server and only transmitted over the loopback device. The frame size of a 1024×768 frame can be computed as in equation 6.1. Initially our network switch and some of the cables we used could only handle $100\frac{Mbit}{s}$. Assuming this as the theoretical network throughput limits the maximum framerate to $5,298$ fps, which can be computed as in equation 6.2. As this result is not acceptable, i.e. the framerate in (6.2) is too low, we have decided to use a $1000\frac{Mbit}{s}$ switch and replaced some of the slow cables. With this change the theoretical limit is as in equation 6.3.

$$frame_{size} = (1024 \cdot 768 \cdot 3)\frac{Byte}{frame} \tag{6.1}$$

$$\frac{100\frac{Mbit}{s}}{frame_{size} \cdot 8\frac{bit}{byte}} = 5,298\frac{frame}{s} \tag{6.2}$$

$$\frac{1000\frac{Mbit}{s}}{frame_{size} \cdot 8\frac{bit}{byte}} = 52,981\frac{frame}{s} \tag{6.3}$$

6.2 Frame segmentation for distributed rendering

Presentation of benchmarks and discussion of the best approach for segmentation.

6.2.1 Smart segment size

As briefly discussed in section 5.8.2 the best segment size, which determines the number of segments, depends on the number of clients connected to the server. As previously mentioned the performance suffers if there are too few segments because, amongst other things, some of the segments might be computed twice by two clients at the end of each frame. To minimise the impact of this, the last segments of a frame could be split in two, producing smaller segments at the end of each frame. This shall be referred to as smart segmentation.

In TheMatrixDistributed, compiled with smart segmentation, this is done by cutting the last $\frac{n_{client}}{2}$ segments in half, effectively creating n_{client} number of segments at the end of each frame with half the size of the other segments, $n_{clients}$ denoting the number of connected clients. This approach has been tested with four clients, a 1024×768 size frame and 13 different segment sizes: 1, 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 128, 256. The result of this test can be seen on figure 6.1, with the segment sizes on the x-axis and FPS on the y-axis.

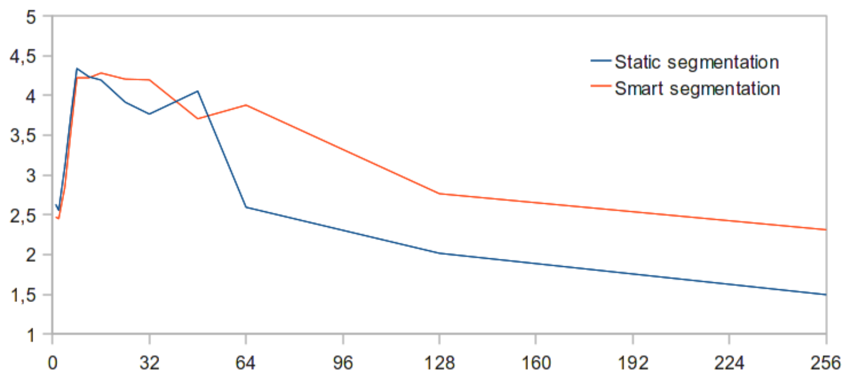


Figure 6.1: Smart segmentation vs. static segmentation.

Zooming in on figure 6.1, as in figure 6.2, smart segmentation is slightly slower initially, however, in the long run it is significantly faster, which makes it possible to reduce the number of segments. Static segmentation may give a slightly higher peak, however, it requires having to test for the best segment size every time a new setup is tried. Smart segmentation reduces the impact of a bad choice without any serious costs. Ideally, the best segment size should be determined dynamically depending on the number of connected clients, however, this could easily be complicated, so it has been decided to leave it at this.

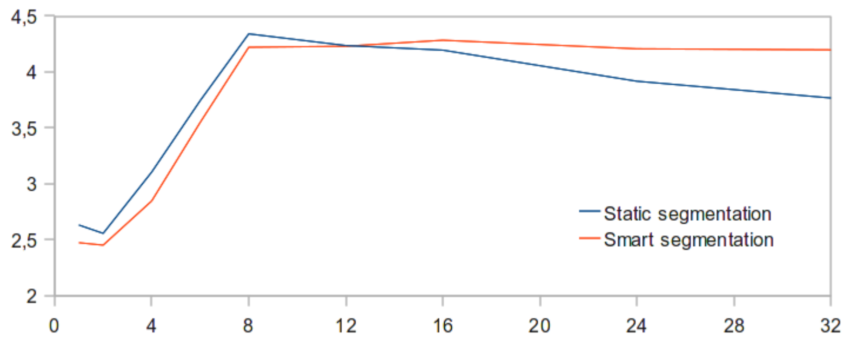


Figure 6.2: Smart segmentation vs. static segmentation (Zoomed).

6.2.2 Segment size

Though smart segmentation reduces the importance of choosing a good segment size, it is still important to choose a good segment size, depending on how many clients that are connected. Figure 6.3 shows the results of a test with different segment sizes and different numbers of clients. These tests were done with smart segmentation activated. The frame in this test was 1024×768 and was tested with 13 different segment sizes: 1, 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 128, 256.

From figure 6.3 it can be concluded that a very small segment size is not good, and as we suspected a very large segment size also works bad. In fact segment sizes between 8 and 32 seem to be best, unless there is only one client connected. This is probably because one client would never compute the same segment twice, before the frame is rendered. These results only reflect a maximum of four clients.

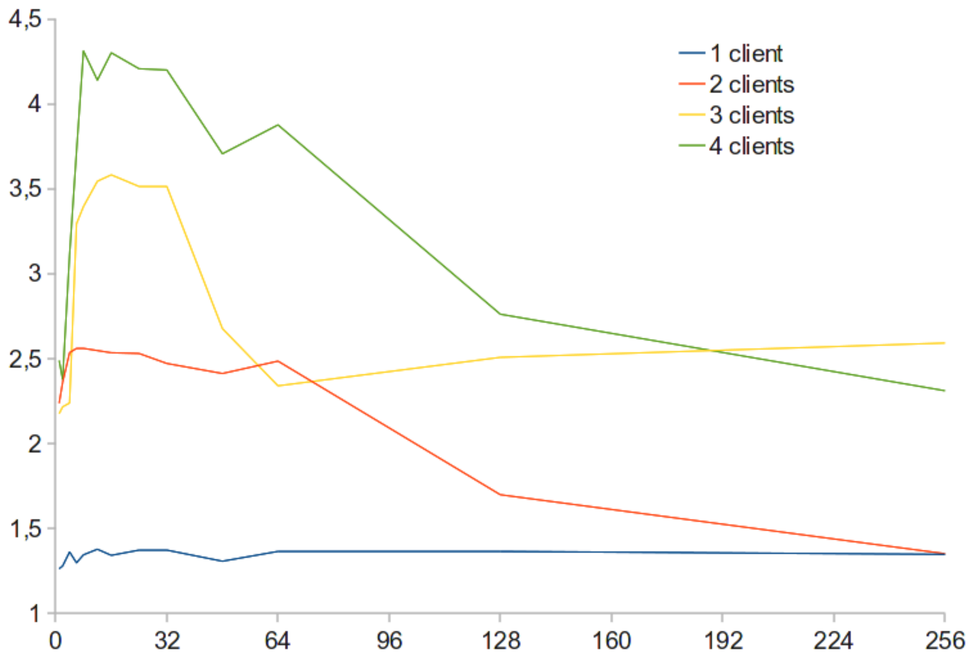


Figure 6.3: FPS for different numbers clients with different segment sizes.

6.3 Cores vs. Computers

This section will compare results between distribution and parallelisation, and discuss the reasons behind the results.

One of the great things about ray tracing is that the fps should be able to scale linearly with the amount of processing power. The computations in ray tracing also easily allow the use of multiple processors. There is however a challenge in getting the processors to communicate to each other efficiently, which can cause overhead and cause the fps to scale less linearly with the amount of processors.

When distributing over a network to several computers additional overhead is added from the network protocol. This overhead can be reduced by ensuring a proper segment size, small enough to allow complex parts of the scene to be split to several computers, but also large enough to reduce the amount of synchronisation required by the server.

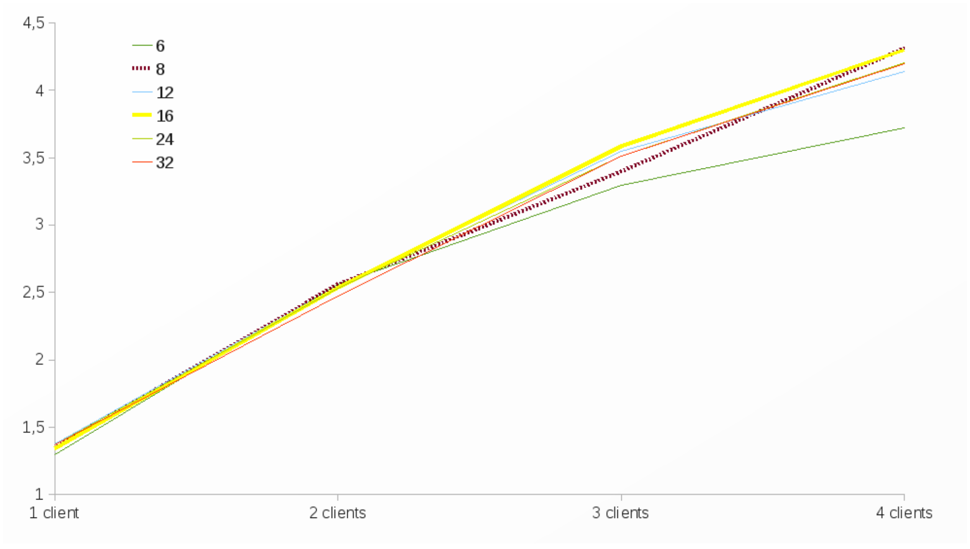


Figure 6.4: Scaling over several computers, at different segment sizes.

Figure 6.4 shows how TheMatrixDistributed scales as more clients are added. The figure also shows several different segment sizes, as these are relevant when distributing to several computers. It is obvious that the impact an extra computer has, declines the more computers there are. This can be due to the expense of synchronising the frame over the network, the inherit albeit small delay in trafficking over the network and because several computers may end up calculating the same segment at the same time, especially towards the end of the frame. Figure 6.4 also confirms the observation in 6.2.2, that the best segment size for four computers is between 8 and 32.

The standalone aspect of TheMatrixDistributed however, scales linearly with the amount of threads / cores activated as can be seen in figure 6.5. This was tested on an Intel Core i7; a Quad Core CPU with Hyperthreading (HT), which gives the CPU a total of eight logical cores. The first four points on figure 6.5 use the four physical cores, showing that once the threads increase beyond the amount of physical cores the CPU has the performance gain is reduced. The performance gained per logical core is however still linear. These results bode well for todays trends of ever increasing amount of cores per CPU, and multithreaded applications. If this trend continues, it can be concluded that it would be technically possible to scale ray tracing to the increasing amount of CPU cores. Unfortunately, todays quad core CPUs are not up for the task of ray tracing independantly in real time, and require either more physical CPUs or help from specialised hardware.

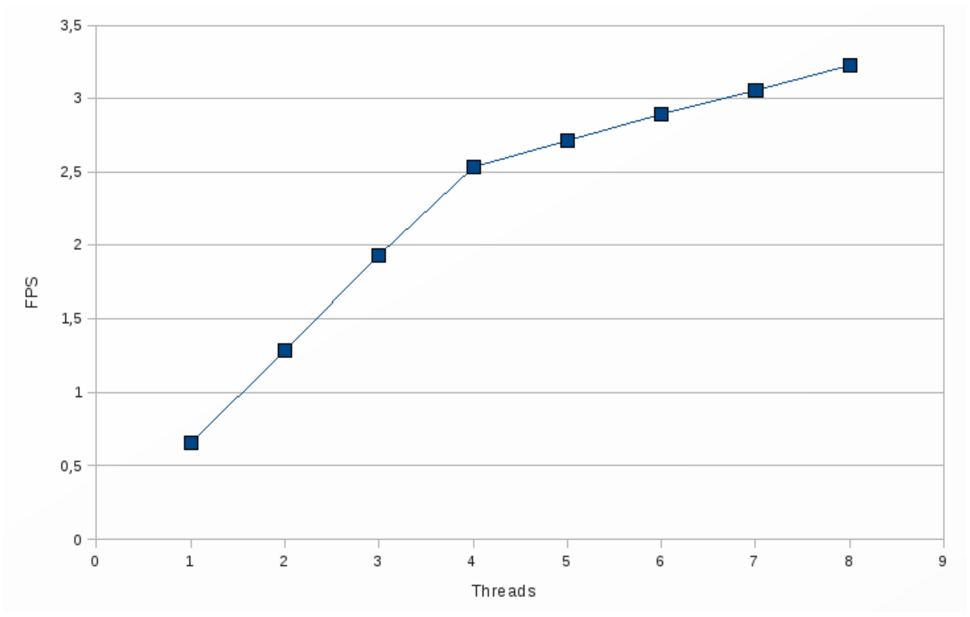


Figure 6.5: Scaling over several cores inside one physical CPU.

6.4 Benchmarking bounding volume hierarchies

As presented in section 5.5.1 TheMatrixDistributed has two methods for generating bounding volume hierarchies. MST (Minimal Space Tree), which is generated in $O(n!)$ time, and SBT (Space Balanced Tree), which is generated in $O(n \cdot log(n))$ time. This section tests which of these trees provides the best performance, and discusses if it is worth the possibly added complexity.

Model name	**Triangles**	**MST built time**[1]	**SBT built time**[1]
"Dino"	$12,516$	$70,873ms$	$101ms$
"Woman"	$2,648$	$4,100ms$	$58ms$

Table 6.1: Properties of the models used for benchmarking MST and SBT.

To test which tree offers the best performance two 3D models, here called "Dino" and "Woman", have been generated with both methods. Properties of the models can be seen in table 6.1, including the time used to build the two different trees of the models. In order to benchmark the trees of these models fairly against each other, they must be rotated during the test. For this test the models are rotated $1/39.789$ radians for each frame, this was done for 500 frames while the average FPS of the last ten frames was printed out for every 10 frames. This causes the model to be rotated $500/39.789 \approx 4\pi$ radians, during the test.

The results of this test can be seen on figure 6.6, with FPS on the y-axis. In table 6.2 the worst and best FPS can be seen, note that these are for ten frames. The average FPS can also be seen in table 6.2. Looking at figure 6.6 it is obvious that the MST trees are more stable, i.e. their performance is less dependant on rotation. Looking at table 6.2 shows that the MST trees offers the best average performance, though their best

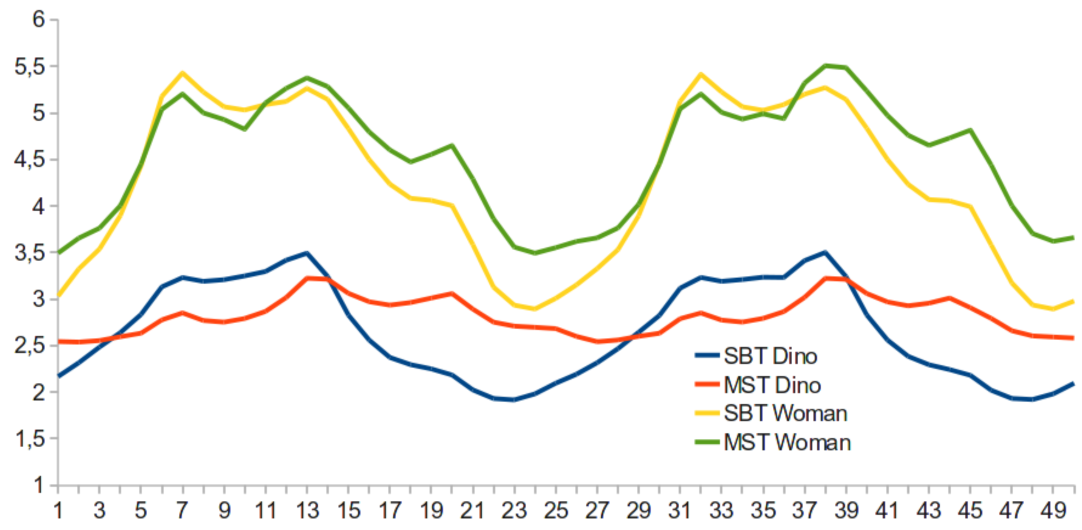


Figure 6.6: FPS of models during rotation.

results may be slightly less than that of the SBT trees.

Model (tree)	"Dino" (MST)	"Dino" (SBT)	"Woman" (MST)	"Woman" (SBT)
Worst result	$2.54FPS$	$1.92FPS$	$3.49FPS$	$2.89FPS$
Best result	$3.22FPS$	$3.5FPS$	$5.51FPS$	$5.43FPS$
Average FPS	$2.83FPS$	$2.66FPS$	$4.56FPS$	$4.27FPS$

Table 6.2: Worst, best and average results of the models during rotation.

Considering these results it can be concluded that the MST trees are slightly faster than the SBT trees. However, considering the time it takes to generate the trees in table 6.1 this minor speed improvement cannot be justified for any real time purposes. Nevertheless, MST trees can still be used to generate models before runtime. This is possible in TheMatrixDistributed, because the `obj2tmd` utility converts .obj files to serialised bounding volume hierarchies. Thus time used to generate the tree does not affect runtime at all. However, considering that an MST tree of 50,000 triangles takes approximately 30 minutes to generate, MST trees are not practical for any large model.

6.5 TheMatrixDistributed Benchmark

This section will describe the best result acquired from TheMatrixDistributed.

One of the last benchmarks performed on TheMatrixDistributed involved connecting as many computers as could be found to a server, to see how high the fps could go. Figure 6.7 shows the scene used for this benchmark. Six laptops with Dual Core Processors and a desktop computer with an Quad Core processor were used in the benchmark. The scene is comprised of approximately 100,000 triangles and one light source at 1024×768. Anti-aliasing was not activated for this benchmark.

The laptops were running two single-threaded clients, while the desktop machine ran four two-threaded clients and the server. The two single-threaded clients gave better performance then one two-threaded client, however, because segmentation became a problem (performance was best at a segment size of 1 line) the number of clients had to be reduced because, TheMatrixDistributed cannot in its current state send half a line as a segment.

TheMatrixDistributed yeilded 8 fps with this setup.

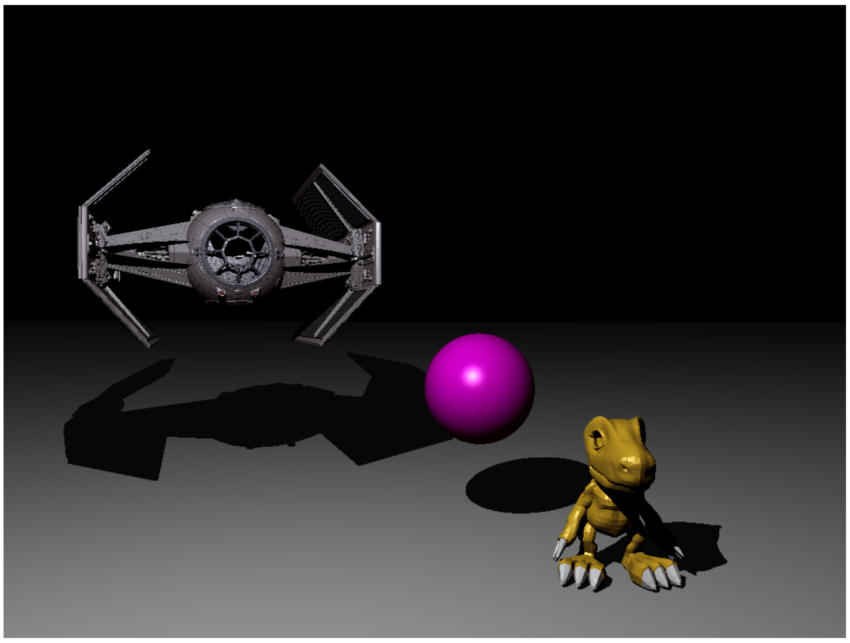


Figure 6.7: Scene used to benchmark TheMatrixDistributed

6.6 Development evaluation

An evaluation of some of the choices regarding the development of TheMatrixDistributed

Looking at the tools and choices made during the development of TheMatrixDistributed, it can be said that most of these choices were not bad. SDL has been fairly portable, the same goes for most of the other libraries that have been used for TheMatrixDistributed. Though some users, mainly Windows users, have experienced minor problems with libnetpbm, and as previously discussed this library is not memory efficient. Thus another or no image library would probably have been better.

C++ has also turned out to be fast, however, during the development we have realised, that we might not need the abstractions that an object orientated programming language like C++ offers. Nevertheless, OOP (Object Orientated Programming) offers us the ability to encapsulate code in an object, thus making the codebase more readable. For instance have a vector class that overloads operators such as $+$, $-$, $\cdot$ etc. is very nice, and if these methods were to be accessed as functions on a struct it would easily make the code far less readable. For these reasons C++ with no or only a very few amount of classes where all methods are not inline would probably be the best, and offer roughly the same performance as C.

For the development of TheMatrixDistributed the concepts that we have borrowed from different development methods have also served us well. Unit testing the vector class has helped find some fundamental bugs and made it a lot easier to maintain this class. However, unit testing the sphere implementation did not pay of, mainly because it was easier to test it by running the ray tracer. Working a little iterative, i.e. not having an idea of what the finished result would look like when we started, has also proved to be a good idea. Mainly because we can easily see that we were not even remotely familiar with the concepts used in TheMatrixDistributed when we started the project.

6.7 Summary

A summary of the effort required to ray trace.

Summing up this chapter, there is a theoretical limit of roughly 50 fps using a 1000Mbs switch and network technology. For the purposes of benchmarking TheMatrixDistributed this was adequate, however, any real world application would require some kind of compression to reduce the amount of bandwidth used.

Futhermore, tests show that TheMatrixDistributed can scale linearly when using internal CPU cores via multithreading, and close-too linear when distributing calculations to several computers over the network. This means that TheMatrixDistributed would be able to render scenes faster and faster as CPUs gain more and more cores. Regarding distribution however, TheMatrixDistributed would reach a point where adding more computers for additional computational power would not be efficient. Further development in the distribution aspects of TheMatrixDistributed could very likely reduce the decline of performance gained when adding additional computers. Distributing ray tracing would still require a substantial amount of computers to be able to ray trace complex scenes in real time. This limits its usability to areas where this sort of computational power is available. It may therefore be prudent to instead focus development on some of the hardware initiatives mentioned in section 3.4, in order to be able to more efficiently ray trace on a single computer.

Finally we were able to squeeze 8 fps out of TheMatrixDistributed when attempting to ray trace a scene with some fairly complex models using multiple computers.

CHAPTER 7

Real time ray tracing in perspective

Throughout this report real time ray tracing has been evaluated from a gaming point of view. The analysis of whether there is a demand for real time ray tracing has only investigated the demand amongst gamers and studied the development and initiatives within gaming graphics. To understand the needed efforts to do real time ray tracing, the main focus point has also been to make it real time rather than utilising many effects, because this most likely will be the focus in games.

But real time ray tracing could have applications in other industries as well. Movie producers already, to some extent, use ray tracing today when making animated videos [Suffern, 2007, p. xiv]. This is possible because they are not dependent on real time rendering, and can therefore afford adding more effects. Many other industries, like architects, industrial designers or scenographers, also need to easily present ideas visually. Using rasterisation does provide a general idea of the scene, but to attain realism is difficult. With a real time ray tracing engine made with all of the features that makes the scene look realistic, it would be possible to make changes on a house or car, and see the effect immediately. Imagine, for example, a salesman trying to sell a penthouse apartment in the city. The sales point of the apartment is the architectural details. But as it has not been built yet, the buyer can have problems imagining the full prospect of the apartment. Therefore a 3D model of the apartment has been made using ray tracing. Because it works in real time, the salesman can move around the apartment, showing it from different angles. It will also be possible to get a realistic idea of how the light falls, to see where it would be best to place the dinner table or the TV. It could even be possible to insert some furniture, move it around, change their colour or the colour of the the wall and see the the changes it creates. A program able to do this would presumably be a benefit for many industries, as it is easy and fast to see the changes and the effects they would have on other aspects of the scene.

The only problem with this is that it will demand so much computer power that it will only be portable using a truck. However, as opposed to real time ray tracing applications in the gaming industry, companies will most likely be able to have several computers in another room performing the computation. All there is needed is a server in the showroom that distributes the computations to other computers and displays the scene.

Future computer games may utilise a mixture of rasterisation and ray tracing to create complex environments with realistic light. This may create a need for ray tracing hardware. As mentioned in 3.4 both Caustic Graphics and Intel are attempting this. If they succeed and there is a market, nVidia and other graphics card developers may face a falling demand for GPUs, which could necessitate a restructuring of the companies. As ray tracing is not a patented invention, hardware solutions is a free-for-all who wish to enter the market. This could lead to increased competetion on price and quality, to the consumer's advantage, as well as different ways of utilising ray tracing. Some companies might go for only ray tracing, while others might create a new form of graphics, using aspects of both rasterisation and ray tracing.

Ray tracing may benefit companies wishing to present ideas to costumers, before it reaches gamers, as companies often have bigger funds for buying new hardware, and often have more space to keep client computers in than the average computer user.

CHAPTER 8

Conclusion

Throughout history, the graphics of computer games have developed from very simple to more realistic graphics. The improved graphics have had an impact on the atmosphere of games. For example, the computer game Doom from 1993 was the first to use the light settings to create a scary atmosphere in the game. This game and others since have used special effects such as lights and shadows to create a realistic environment that involves the player in ways very different what the first computer games did. Ray tracing would make the creation of such atmospheres easier than rasterisation does today. But as it is not possible to do ray tracing in real time today, improvements on the algorithms or hardware must be made. Research on which initiatives exist on this subject showed that almost every big company working with computer graphics are trying to implement ray tracing in their graphics engine. Intel expect their new Larrabee to be able to do real time ray tracing, while Caustic have made a completely new hardware device aimed only at doing real time ray tracing. Generally it is found that all the major hardware manufacturers are, to some extent, working to make real time ray tracing possible.

A questionnaire done amongst expected gamers at Aalborg University showed that the graphics in computer games are of average importance. This means that they do not buy a game for its graphics, but expects it to be a part of the gameplay.

The investigation of the needed efforts to do real time ray tracing was researched by implementation of our own ray tracer. With the effects chosen for this project it was possible to render simple scenes in real time on one computer. When increasing the number of objects to 10^5 using 6 laptops and a quad-core computer it was possible to render the scene with about 8 FPS. This is not real time, however, using around 20 PCs to render 10^5 objects could perhaps lead to a real time result. This is because TheMatrixDistribued is able to distribute the calculations and thereby increase the computational power. Tests have showed that there is a nearly linear increase in FPS with an increase in number of PCs, which can indicate that 20 PCs would be able to run TheMatrixDistributed in real time. However, there are still possible optimations that could be implemented in TheMatrixDistribued, which would make it render faster. Nevertheless, it must be noticed that TheMatrixDistributed does not enable reflection and transmission or soft shadows. If these effects were implemented in TheMatrixDistributed it would be much slower.

The final question is if the demand can make up for the extensive need for computer power. Real time ray tracing in computer games is not a possibility today, as several computers are needed to render a scene with minimal demand for realism. The improvement in graphics will not outweigh this fact and since the demand for better graphics have proven to be of lesser importance, it can be concluded that real time ray tracing not will be utilised in computer games in the near future.

On the other hand ray tracing is used today to view and present new products such as cars and air planes. This does not necessarily require real time rendering as presentations can be pre-rendered, as when making movies, and the big companies have the ability and space to use several computers in another room to render the scene.

To present 3D models in real time using ray tracing is not possible today for the average company because of the required efforts. In the forthcoming years it could be possible to do real time ray tracing, but it would demand better hardware and/or more optimisations of the software.

CHAPTER 9

Future work

Reflection, transmission and soft shadows have yet to be implemented in TheMatrixDistributed. These features were given a lower priority, as this focus of the project was real time ray tracing, rather than attaining better graphics.

Additional optimisations could be made in how the rendering is distributed. This includes dynamically determining the segment size on the server at runtime and buffer optimisation.

It would also be possible to utilise SSE instructions on some of the methods, for example the box intersection, triangle intersection and camera. Rewriting the algorithms to utilise SSE instructions could be a challenge.

TheMatrixDistributed makes a lot of virtual calls which are very expensive. Some methods have been made inline but work can still be done to optimise the code further. The bounding volume hierarchy used in TheMatrixDistributed can also be improved. For instance a method of building balanced trees would be interesting.

It would be interesting to implement kd-trees, which is another ray tracing acceleration structure, e.g. an alternative to BVH. This would allow some distance checks and a search of the closest axis-aligned bounding box for intersection to be performed first, which might improve performance.

This study has used Blender, a 3D program, to edit and export the different models used to OBJ-format. However, TheMatrixDistributed was unable to load some of the models, as they contained special effects like animation and skeletons. Improvements could be made to allow TheMatrixDistributed to load these objects and ignore the special effects.

Currently TheMatrixDistributed can only load models and textures when the program starts. Dynamically loading models and textures during runtime could increase the interactive nature of the program. As it is, the only interactive part of TheMatrixDistributed is the camera, which can be moved inside the program. However, models can moved and rotated, but this is not used interactively, though it is possible.

Reading textures without netlibpbm might lead to better performance, as netlibpbm uses lot of memory.

There is also still work to be done on the controls of the camera, as the rotations and movements are not intuitive.

APPENDIX A

Appendix

A.1 Questionnaire

Below are the questions from the questionnaire sent to the second semester students a the Faculty of Engineering, Science and Medicine at Aalborg University. The on-line version can be found through the following link:
http://www.askpeople.dk/ask/real/scheme.php?frm=1&pubFormId=47395449213c858.
First question: Which platforms do you use?

- PC
- XBox 360
- mXBox
- Playstation 3
- Playstation 2
- PSP
- Nintendo DS
- Wii
- Other
- Do not wish to answer/Do not know

Second question: What game types do you play?

- Arcade (ex. Guitar Hero, Sing Star, Ping Pong)
- Adventure (ex. The Longest Journey, Perry Rhodan, Escape from Monkey Island)
- Action (ex. Crysis, Halo, Half-Life)
- Role Playing Game (ex. Mass Effect, Neverwinter Nights, Fallout)
- Mass Multiplayer Online (ex. World of Warcraft, Eve Online, Warhammer Online)
- Platform (ex. Mario, Pacman, Rayman)
- Flash (ex. Fly the Copter, Bubbleshooter, Motherload)
- Strategy (ex. Command and Conquer, Age of Empires, Homeworld)
- Simulation (ex. Sims, Flight Simulator, Tycoon games)
- Sport (ex. Fifa, NHL, Wii Sports)
- Do not wish to answer/Do not know

Third question: What is important for you when, you buy a game? Answer on a scale from 1 to 5.

- Graphics
- Game mechanics

- Multiplayer
- Story
- Completion time
- Replay value
- Price
- Platform
- Do not wish to answer/Do not know

Fourth question: Which of the following is important for you to keep playing a game? Answer on a scale from 1 to 5.

- Graphics
- Gameplay
- Multiplayer
- Story
- Completion time
- Platform
- Do not wish to answer/Do not know

Fifth question: How satisfied are you with the graphics in games today? Answer on a scale from 1 to 5.

Sixth question: How expensive are the games you buy?

- Less than kr 100
- Kr 101 - kr 250
- Kr 251 - kr 400
- Kr 401 - kr 600
- More than kr 600

Seventh question: How many times have you upgraded your hardware in the last five years?
For example bought new RAM, video card, a new computer, or a new console.

- 1
- 2
- 3
- 4
- 5
- 6
- More than six times
- Do not wish to answer/do not know

Eighth question: How many hours a week do you play computer/console games?

- Less than 2 hours
- 2 - 5 hours
- 6 - 12 hours
- 13 - 20 hours
- More than 20 hours
- Do not wish to answer/do not know

Bibliography

libnetpbm documentation, December 2003. URL http://netpbm.sourceforge.net/doc/libnetpbm.html.

Doom darkness (image). Retrieved 17. May 2009, from http://en.wikipedia.org/wiki/File:Doom_darkness.png, October 2007.

American Rental Association. Glassware. Retrieved 19. May 2009 from http://party.rainbow-rental.com/dinnerware/.

Jacco Bikker. Ray tracing in games: A story from the other side. Retrieved 22. May 2009, from http://www.pcper.com/article.php?aid=582, June 2008.

Paul Bourke. .obj documentation. Retrieved 18. May 2009 from http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/.

Kevin P. Casey. Nintendo hopes wii spells wiinner, August 2006. URL Retrieved18.May2009from\url{http://www.usatoday.com/tech/gaming/2006-08-14-nintendo-qa_x.htm}.

NASA Atmospheric Science Data Center. What wavelength goes with a color? Retrieved 19. May 2009 from http://eosweb.larc.nasa.gov/EDDOCS/Wavelengths_for_Colors.html, September 2007.

Coola. Xbox 360 launch takes europe by storm. Retrieved 18. May 2009 from http://www.xbox365.com/news.cgi?id=GGPPLLiddi12060836, December 2005.

Christophe de Dinechin. The dawn of 3d games... Retrieved 17. May 2009, from http://grenouille-bouillie.blogspot.com/2007/10/dawn-of-3d-games.html, November 2007.

Martin Eisemann, Thorsten Grosch, Stefan Mueller, and Marcus Magnor. Fast ray/axis-aligned bounding box overlap tests using ray slopes. 2008. Available at http://graphics.tu-bs.de/publications/Eisemann07RS.pdf.

David Erickson. Demographics of video gamers. Retrieved 19. May 2009 from http://e-strategyblog.com/2005/10/demographics-of-video-gamers, October 2005.

Sony Computer Entertainments Europe. Ps3 to launch in europe and australasia on 23rd march 2007. Retrieved 18. May 2009 from http://www.scee.presscentre.com/Content/Detail.asp?ReleaseID=4317&NewsAreaID=2, January 2007.

Andrew S. Glassner. *An introduction to ray tracing*. Academic press, 2 edition, 1989. ISBN 0-12-286160-4.

Niels-Henrik M. Hansen, Bella Marckmann, and Ester Naerregaard-Nielsen. *Spoergeskemaer i virkeligheden*. Samfundslitteratur, 1 edition, 2008. ISBN 978-87-593-1374-9.

Ben Hardwidge. Nvidia demos real-time gpu ray tracing at 1920 x 1080. Retrieved 19. May 2009 from `http://www.custompc.co.uk/news/604656/nvidia-demos-real-time-gpu-ray-tracing-at-1920-x-1080.html`, August 2008.

IEEE and The Open Group. The open group base specifications issue 6 — ieee std 1003.1. Retrieved 18. May 2009 from `http://www.opengroup.org/onlinepubs/009695399/`, 2004.

Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel, March 2009.

Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. Intel, 1999.

Intel C++ Compiler User and Reference Guides. Intel, 2008.

Intel. Larrabee: Next-generation visual computing microarchitecture. Retrieved 19. May 2009 from `http://www.intel.com/technology/visual/microarch.htm?iid=SEARCH`, August 2008.

Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, August 2003. ISBN 978-0-13-111155-4.

Microsoft. Xbox 360 records its biggest year ever. Retrieved 18. May 2009 `http://www.microsoft.com/presspass/press/2009/jan09/01-05XBoxBigYearPR.mspx`, January 2009.

Digibarn Computer Museum. The maze war 30 year retrospective at the digibarn. Retrieved 17. May 2009, from `http://www.digibarn.com/collections/games/xerox-maze-war/index.html` and subpages., November 2004.

Nintendo. Consolidated financial statements. Retrieved 18. May 2009 from `http://www.nintendo.co.jp/ir/pdf/2009/090507e.pdf#page=23`, May 2009.

IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. June 2008. ISBN 978-0-7381-5752-8.

Lars Pedersen. *Matematik 112*. Nyt Teknisk Forlag, 1 edition, 2006. ISBN 978-87-571-2581-8.

PC Perspective. Crytek's cevat yerli speaks on rasterization and ray tracing. Retrieved 18. May 2009 from `http://www.pcper.com/article.php?aid=546`, April 2008a.

PC Perspective. Nvidia comments on ray tracing and rasterization debate. Retrieved 18. May 2009 from `http://www.pcper.com/article.php?aid=530`, March 2008b.

Daniel Pohl. Ray tracing and gaming - quake 4: Ray traced project. Retrieved 19. May 2009 from `http://www.pcper.com/article.php?aid=334&type=expert&pid=4`, December 2006.

Daniel Pohl. Quake wars ray traced. Retrieved 19. May 2009 from `http://www.qwrt.de/`, August 2008.

Research@Intel. Real time ray-tracing: The end of rasterization? Retrieved 18. May 2009 from `http://blogs.intel.com/research/2007/10/real_time_raytracing_the_end_o.php`, October 2007.

Peter Shirley. *Fundamentals of Computer Graphics*. Wellesley, Mass., AK Peters, 2 edition, 2005. ISBN 978-1568812694.

Ryan Shrout. Caustic graphics ray tracing acceleration technology review, April 2009.

Micheal Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, 2006. ISBN 0-534-95097-3.

Sony. Playstation2, the world's most popular computer entertainment system, now more affordable than ever at at $99.99. Retrieved 22. May 2009, from `http://www.scei.co.jp/corporate/release/pdf/090331ae.pdf`, March 2009.

Sony. Unit sales of hardware (since april 2006). Retrieved 18. May 2009 from `http://www.scei.co.jp/corporate/data/bizdataps3_sale_e.html`, December 2008.

Kevin Suffern. *Ray Tracing from the Ground Up*. A K Peters, Ltd., 1 edition, 2007. ISBN 978-1-56881-272-4.

Eric Haines Tomas Akenine-Möller. *Real Time Rendering*. A K Peters, 1 edition, 2002. ISBN 1-56881-182-9.

Balazs Toth. Speed optimized recursive ray-tracer with kd-tree and sse vector mathematics. 2005.

Theo Valich. Watch out, larrabee: Radeonn 4800 supports a 100using directx 9. Retrieved 19. May 2009 from `http://www.tomshardware.com/news/Larrabee-Ray-Tracing,5769.html`, June 2008.

Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

Ingo Wald and Philipp Slusallek. State of the art in ray interactive tracing. Retrieved 19. May 2009 on `http://graphics.cs.uni-sb.de/Publications/2001/star.pdf`, 2001.

wikipedia.org. Area light source soft shadow. Retrieved 19. May 2009 from `http://commons.wikimedia.org/wiki/File:Area_light_source_soft_shadow.png`, May 2009.

Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray–box intersection algorithm. 2005. Available at http://www.cs.utah.edu/ awilliam/box/box.pdf.