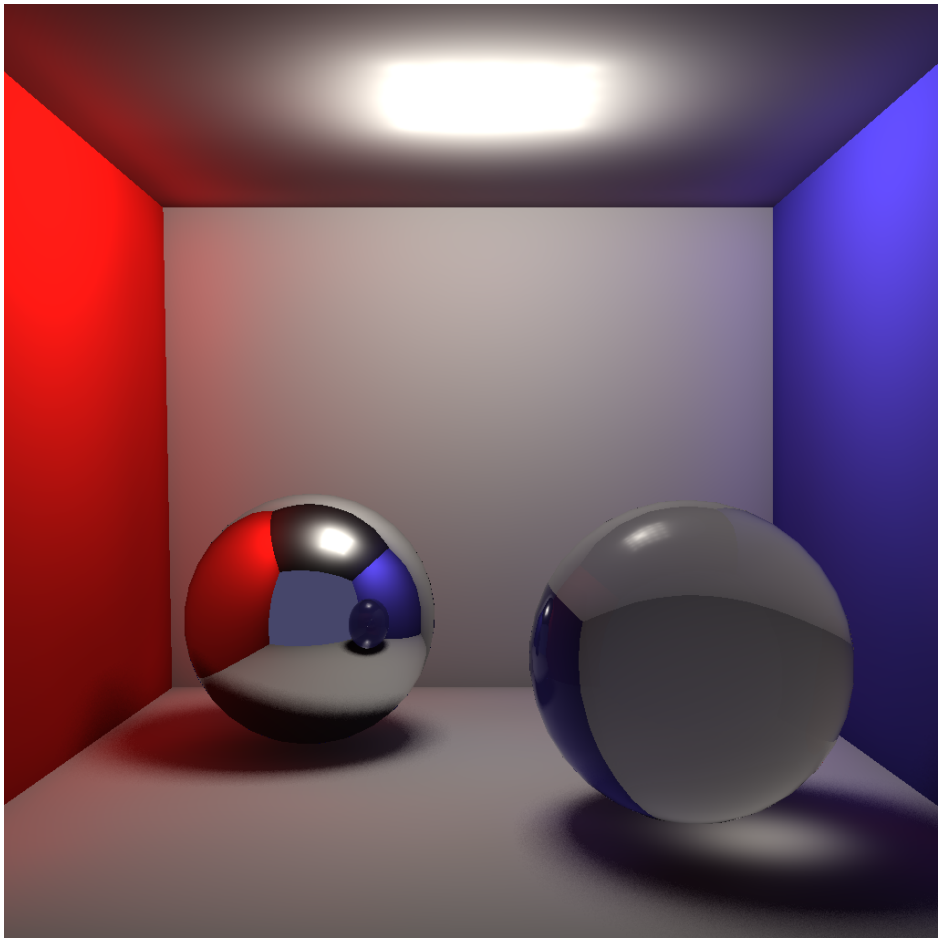
CHALMERS

ELUMI

A ray tracer for rendering photo-realistic images



DAVID ERIKSSON
PHILIP IRRI
JULIAN LINDBLAD
JIMMY MILLESON
NIKLAS ULVINGE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2012
Bachelor's Thesis DATX02-12-34

Abstract

This thesis describes the implementation of ELUMI, a cross-platform 3D renderer written in C++. It was conducted in the form of a case study, evaluating different techniques and their impact on the final product. This thesis discusses complications typically encountered when implementing a ray tracer and how these can be circumvented. Considerations were taken concerning quality, rendering speed, extensibility, configurability, and compatibility when features were chosen for implementation.

ELUMI is capable of simulating light interacting with surfaces, diffuse interreflections, physically accurate shadows, reflections and refractions, geometric properties using textures, and participating media. The rendering process is parallelized and accelerated with a K-D tree, as well as optimized with shadow caches and a cache aware traversal pattern. To further improve the rendering speed, certain parts can be accelerated using the GPU. ELUMI is also capable of further enhancing the image quality by using anti-aliasing techniques and a tone mapping operator.

To ameliorate the user experience, a visual interface was concurrently developed. The development process was carried out iteratively and was facilitated using a distributed version control system. The end results are presented in this thesis.

Sammanfattning

Denna avhandling beskriver implementationen för ELUMI, en plattformsoberoende 3D-renderare skriven i C++. Den var genomförd i form av en fallstudie, vars syfte var att utvärdera olika tekniker och dess påverkan på den slutgiltiga produkten. Avhandlingen diskuterar komplikationer som typiskt uppstår vid implementering av en ray tracer och hur dessa kan kringgås. Hänsyn i ELUMI togs kring kvalitet, renderingstid, utökningsbarhet, konfiguerbarhet och kompatibilitet när funktioner valdes för implementation.

Med ELUMI är det möjligt att simulera ljus som interagerar med ytor, diffusa reflektioner, fysiskt korrekta skuggor, reflektioner och refraktioner, geometriska egenskaper med hjälp av texturer och medverkande medium. Renderingsprocessen är parallelliserad och accelererad med med hjälp av ett K-D-träd, liksom optimerad med skugg-cachar, och ett cache-medvetet traverseringsmönster. För att ytterligare förbättra renderingshastigheten kan vissa delar accelereras med hjälp av GPU'n. ELUMI är även kapabel att förbättra bildkvaliteten med hjälp av kantutjämningsmetoder samt en tonmappningsoperator.

För att förbättra användarupplevelse, samt utvecklingsprocessen, utvecklades ett visuellt gränssnitt i samband med studien. Utvecklingsprocessen utfördes iterativt och underlättades med ett distribuerat versionshanteringssystem. Slutresultaten presenteras i denna avhandling.

Acknowledgements

First of all, we would like to thank Ulf Assarsson for providing guidance and feedback on the thesis. We would also like to thank Fritiof Hedman, who kindly allowed us to render our scenes on his server, even with unoptimized data structures. Furthermore, we would like to express our thanks to group number 37, who we shared rooms together with, for allowing us to play music during our coding sessions.

David Eriksson, Philip Irri, Julian Lindblad, Jimmy Milleson, Niklas Ulvinge

Gothenburg, 12 May 2012

Table of Contents

1

Introduction

This thesis discusses the implementation of ELUMI, a 3D rendering application. A renderer is an application that generates images from a scene created by one or more artists. A scene typically consists of data that defines the different objects, light sources, and their basic characteristics.

A renderer can be used to create animated movies. The animation studio Pixar, which are behind animated movies such as Cars, Finding Nemo, and the Toy story trilogy, is responsible for the rendering application RenderMan [Christensen et al. 2006]. RenderMan is used when rendering 3D animations and visual effects, which can be seen in the movies Terminator 2 and Jurassic Park. The RenderMan application can use ray tracing to improve image quality when rendering, with techniques like global illumination (see Section 5.2) and soft shadows (see Section 4.2). Other renderers that use ray tracing to increase realism are Blender's internal renderer, Yafaray, and Mental Ray. See Figure 1.1 for an example of a rendered image.

Figure 1.1: Image from the open source movie Big Buck Bunny, generated with Blender. *(c) copyright 2008, Blender Foundation / www.bigbuckbunny.org*

There are two main types of renderers: real-time and offline renderers. A real-time renderer is usually used in games, since a game needs to be immersive and provide real-time feedback. An offline renderer however, is not constrained to real-time speeds. They are instead used in the movie industry since realistic looking visuals are prioritized.

Rasterization is a technique most often utilized in real-time rendering. Today there is hardware that is highly optimized for real-time rendering using rasterizing [Lindholm et al. 2008]. Path tracing, in contrast, is an offline rendering technique which renders very realistic images at very slow speed [Kajiya 1986]. Ray tracing is another technique that can be used to render realistic images faster than a path tracer [Whitted 1980], however not as physically accurate.

1.1 Purpose

The main purpose of this thesis is to investigate how to create the best ray tracer possible given a limited amount of resources, while keeping an optimized balance between quality, rendering speed, extensibility, configurability, and compatibility:

Quality and rendering speed: A main focus is to produce photorealistic images by simulating real life phenomena. Different techniques to achieve this are therefore investigated. Where applicable, the ray tracer should render images with the quality comparable to a path tracer, but with considerably faster rendering time. Another focus of the project is functionality; therefore, features that are easy to implement are prioritized.

Extensibility: The ray tracer should be modular. This facilitates the development process, making it easier to implement and modify code, as well as allowing for future development.

Configurability: The ray tracer should be configurable for different purposes. It should provide artistic freedom for the user.

Compatibility: 3D-applications exist on the major to platforms (Windows, Mac OS X, and Linux) and on all popular hardware (NVIDIA, ATI, Intel, and AMD). Making the ray tracer cross-platform will enable it to reach a broader audience.

1.2 Method

The focus of this case study was to implement and evaluate as many features as possible. Therefore, a list of conceivable features was first created. It was then ordered such that basal features were given higher priority over more specific ones. Features were then progressively implemented and removed from the list in the order of their priority.

To speed up the development process and let us focus on ray tracing features, existing libraries were used for object importing, window management, threading, linear algebra, and image exporting.

Project management can be helpful to use to provide a solid development process. Therefore, an agile methodology was used for this project. The chosen methodology was Scrum. Following the Scrum methodology, the project was split into several iterations. These iterations lasted for one to three weeks. The first iteration focused on the core features needed to render a basic scene. This led to a basic application which was then further extended with more features in later iterations.

A distributed version control system, Git, was used to enable effective simultaneous development. In Git, every developer manages over a local copy of the entire development history [Chacon 2009]. This allowed features to be implemented in separate branches before being merged into the main branch.

C++ was chosen as the programming language because it is object-oriented, fast, compact, and portable [Prata 2004]. There exist several libraries for C++ which were used to speed up the development process. During the development process, the project was continuously compiled and run on Windows, Mac OS X, and Linux. This ensured that the ray tracer was always runnable on all three platforms.

1.3 Outline

This thesis is structured after the ray tracing algorithm described in Section 2. Section 2.1 describes what information is required in order to render an image. Then, a description of some general parts of a ray tracer follows in Section 2.2 and Section 2.3. Section 3 describes the different parts of the `trace` function. Section 4 describes the `shade` function, i.e. how the pixels are colored. Section 5 describes GPU accelerations and two extensions to the ray tracing algorithm that simulate global illumination and participating media. When the image is finally rendered, a method for presenting it on the screen is described in Section 6. Section 7 and Section 8 contain the results and discussion, which is followed by a final conclusion in Section 9.

Since every section more or less is a self-contained unit, this thesis is structured as follows: Every section has an introduction describing the purpose of the features, which is followed by a summary of previous work detailing the theory behind the methods. After our implementation of the feature is described, the results are given followed by a discussion.

2

Ray tracing introduction

The ray tracing algorithm, as first described by Whitted [1980], works by shooting rays, for each pixel, from the eye through a grid into the scene (see Figure 2.1). Our implementation (see Algorithm 1) begins by reading the scene from disk and then builds a data structure from it (see Section 3). Afterwards, the algorithm iterates over each pixel and generates a ray (see Section 2.3) which in turn is traced and colored thereafter.

The tracing is done by traversing the data structure for the closest surface (Section 3) and then using the information from this intersection to shade the pixel. The shading is dependent on the material of the surface that is hit. If the surface is reflective, a reflection ray is traced. Similarly, a refraction ray is traced for transparent surfaces. Furthermore, if a diffuse surface is hit, shadow rays are shot from each light source to the surface in order to determine if the object is in shadow. The final shading can also be a mix of all three effects. If there is no intersection for the pixel being traced, the background color is used to color the pixel.

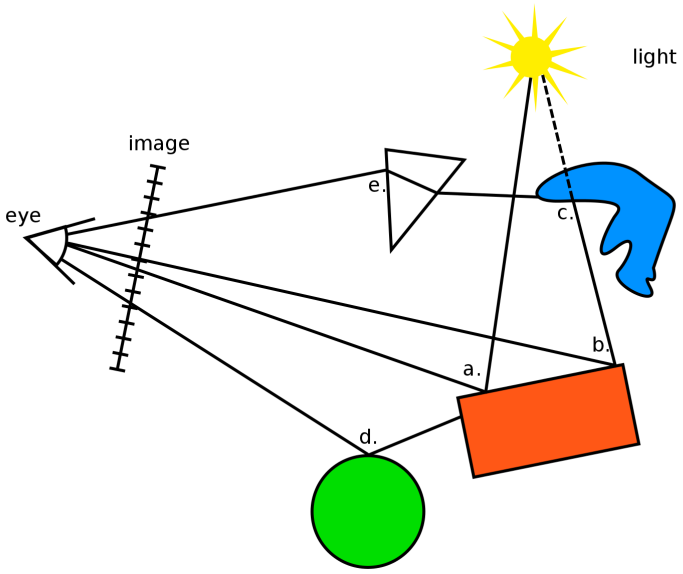


Figure 2.1: Illustrating the ray tracing algorithm. Showing rays shot from the eye through four pixels onto geometry in a scene. Different actions are taken depending on the material hit: **(a)** Point in light with a diffuse material. **(b)** Point in shadow with a diffuse material. **(c)** Blocker of a shadow ray. **(d)** Reflection ray for a shiny object. **(e)** Refracted ray for a transparent object.

2.1 Scene - defining the world

To render a synthesized image, a scene is required to define what will be rendered, and how it will be rendered. A scene includes geometry represented using 3D data, camera, and lights. The 3D data describes how the models in the scene should look. The camera is needed to describe what part of the scene should be rendered and is defined by a position and a direction. The lights describe how the scene is illuminated and, along with the 3D models material properties, determine how the geometry is rendered. This section will describe how a scene is created and defined in the ray tracer.

2.1.1 Method

Before anything can be rendered, a scene needs to be initialized. The simplest form of a scene is one that only holds a camera. A scene is made more interesting by adding geometry, which is done by importing 3D models. The geometry is made up of triangles, which in turn are represented by three points in space. Materials are assigned to every triangle, describing its properties. For example, triangles may have different colors, they may be reflective or refractive, or they could have different levels of transparency. Rendered images are made more realistic by adding lights to the scene. After the scene has been initialized, a synthesized image is ready to be rendered.

The different parts of a scene need to be specified. For the geometry, Wavefront's OBJ format was used as a representation of 3D models. However, lights as well as the camera need to be represented. Therefore, our ray tracer uses XML documents to describe scenes. The XML document has different elements that

Algorithm 1 The main parts of the ray tracing algorithm

```
function RAYTRACE(file) : color [] []
    scene ← read(file)
    datastructure ← build(scene)
    for all (x,y) do
        ray ← generate(x,y)
        image[x][y] ← TRACE(ray)
    end for
    postprocess(image)
    return image
end function

function TRACE(ray) : color
    intersection ← findClosest(ray)
    return SHADE(ray,intersection)
end function

function SHADE(ray, intersection) : color
    if miss(intersection) then
        return environment(ray)
    else
        color ← emissiveLight
        for all light do
            color ← color + calculateLighting(light)
        end for
        if refractive then
            color ← color + TRACE(refractionRay)
        end if
        if reflective then
            color ← color + TRACE(reflectionRay)
        end if
    end if
    return color
end function
```

describe different aspects of the scene. For example, a 3D model is referenced in the scene with an Object-element. Other elements include a camera, different lights, and a reference to a settings file. The settings file is also an XML document and holds parameters such as the resolution of the image, what effects should be used, and the maximum recursion depth. See Appendix A for examples of configuration files.

2.1.2 Result - discussion

The OBJ format was chosen as the format for 3D models. This lead to certain obstacles. For example, the camera and lights are not supported with this format. Another major drawback with the object format is that the tangent and bitangent vectors necessary for bump mapping are not exported with this format. To a certain extent, these drawbacks were circumvented by using XML. Support for more formats would therefore be an admirable feature.

2.2 Threading and traversal pattern

This section will introduce how the use of multiple threads drastically improves the rendering speed. It also discusses the advantages of different pixel traversal patterns.

2.2.1 Previous work

The ray tracing algorithm (described in Section 2), needs to traverse and shade each pixel. The tracing performance can be improved by altering the *traversal pattern*, i.e. the sequence that the pixels are being traced and rendered [Arvo 1994]. The idea of this is to better exploit cache coherence, that is, newly accessed elements are saved in a memory closer to the CPU (e.g. L1/L2/L3-caches) in the memory hierarchy. The next time the CPU will access that element, the access time is reduced if it still lies in a CPU-near memory. As a consequence, the overall rendering speed is improved. This is also exploited using shadow caches (see Section 4.2.2).

Cache coherence will be favored by geometric objects which screen projections span several pixels across and that share similar intersection and lighting conditions. The optimal way to exploit this would be to traverse and render geometries completely before moving on to another. A way to improve coherence, without knowing the details of the scene, is by only trying to traverse adjacent pixels. This can be done in several different ways. A trivial render sequence is to traverse the pixels in rows, from top to bottom, in a so called scanline pattern (see leftmost figure in Figure 2.2). However, the scanline traversal will then leave the object for a long time, not coming back until the next row. As a result, scanline traversal only takes one dimension of coherency into account.

In order to increase the coherence dimension to two, *space-filling curves* can be used as traversal sequences [Arvo 1994]. Space-filling curves were discovered by [Peano 1890] and have the domain of the unit interval $[0,1]$. They are single, continuous curves, which traverse every single position of a fixed N-dimensional hypercube once, in this case a square, without crossing previously visited positions Akenine-Möller et al. [2008]. Two different space-filling curves are the Peano [Peano 1890] and Hilbert [Hilbert 1891] curves. As can be observed in Figure 2.2, the curves traverse small local areas. Furthermore, the Hilbert curve traverses each area more tightly than the Peano curve, visiting all pixels in an area of 2x2, 4x4, 8x8, 16x16 etc., before moving on. Consequently, the Hilbert curve has an advantage in pixel coherency.

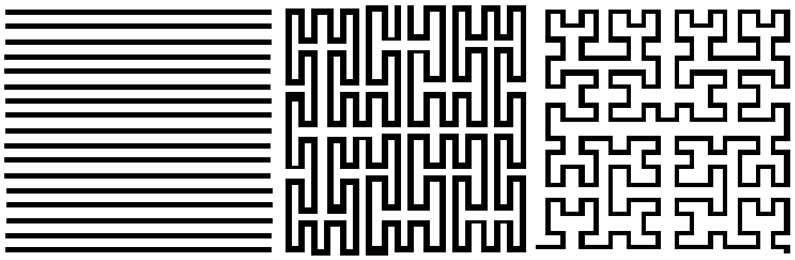


Figure 2.2: Different pixel traversal patterns. From left to right: scanline pattern, Peano curve, Hilbert curve.

2.2.2 Method

Tracing the rays is a highly parallel task, and since most computers today have multiple cores, the performance could be improved a lot by threading. A scheduler was written and the library Boost was used to spawn the threads. The scheduler fetches *batches of pixels* that are divided either by line or by an interval in the Hilbert curve as described below.

A discrete C-implementation of the mapping between the Hilbert index and the coordinates in an arbitrary dimension is given by Lawder [2000]. However, a much shorter implementation, limited to two dimensional coordinates, has been used in this renderer [Wikipedia 2012]. See Figure 2.3 for a series of the first building steps.

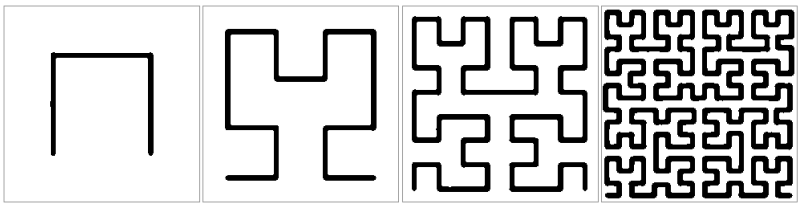


Figure 2.3: The first steps in building a Hilbert curve.

Since the range of the Hilbert coordinates always lies in the range $[0,2^n], n \in \mathbb{N}$, a problem occurs when the desired render image resolution does not match. This has been solved by choosing a n such that 2^n becomes the resolution's maximum component's next largest power of two (see Equation (2.1)). For example, if the image resolution to be rendered is 500x200 pixels, the maximum component is 500 pixels. The next (or equal) power of two for it is $2^9 = 512$. Therefore n is chosen to be 9.

$$n = \lceil \log_2 (\max (width, height)) \rceil \tag{2.1}$$

When implementing this there will presumably be a lot more pixels to be traced than there actually are in the desired resolution. This has been solved using a two-pass technique. First, the batches which will be fetched and traced later on need to be initialized. This is done by letting each batch contain an array of numbers which maps to a screen position in the scanline sequence (see Equation (2.2)). These numbers are generated by calling the Hilbert function with the current pixel count (0 to 2^n) to get the Hilbert coordinates. If the returned coordinates are located within the about to be rendered image, they are converted to the scanline ordering number mentioned above. Otherwise, if the coordinates are located outside, an error flag is returned. The batch's arrays now contain the scanline pixel numbers ordered by

the hilbert curve pattern, as well as a number of flags if the rendered resolution is not equal to a two to the power of n sized square (see Figure 2.4). Secondly, a new array is initialized with size $width \times height$ and all non-flagged values from the previously array are placed in it. This is the finalized array which contains the scanline pixel numbers mapped to a Hilbert coordinate.

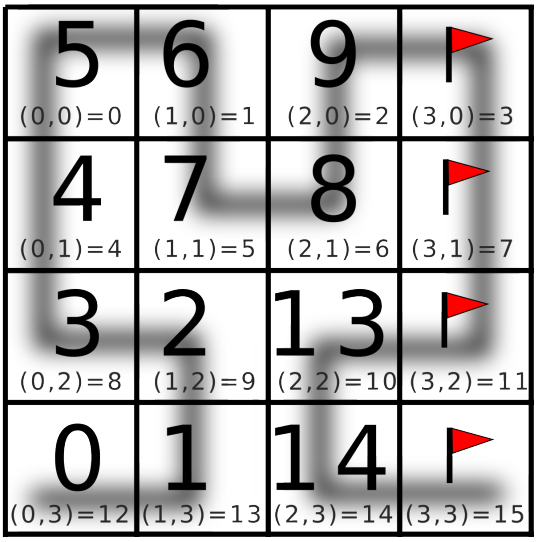


Figure 2.4: This particular image shows the Hilbert indices (large numbers), coordinates, and their respective scanline indices (small gray numbers). The image rendered is of size 3x4, but due to the fact that a Hilbert curve must be of size $2^n x 2^n$, the grid is expanded to 4x4. Note the error flags outside of the image size.

$$i = width \cdot y + x \tag{2.2}$$

2.2.3 Results

As can be seen in Table 2.1 the rendering performance roughly scales linearly with the number of cores used to render. Note that the performance is solely based on the `trace` function.

Table 2.1: Showing how render times are linearly scaling with increasing number of cores.

Nr of threads	**1**	**2**	**3**	**4**
Render time (s)	487	264	178	135
Relative speedup	1.00	1.85	2.73	3.61
Scale factor	100 %	93.5 %	91.0 %	90.3 %

For a scene which benefits from the use of shadow caches (see Section 4.2.2), the rendering time is slightly improved by approximately five percent when traversing with the Hilbert pattern rather than the scanline pattern.

Amdahl's law, which is a formula often used in parallel computing to predict speedups using multiple processors and/or cores, has been used to calculate the ratio of pure parallelizable code in the `trace` function [Amdahl 2007]. According to Equation (2.3), where `S` is the speedup (taken from Table 2.1) using `P` processors, ELUMI has approximately 95 % pure parallelizable code when tracing.

2.2.4 Discussion

$$Parallel_{estimated} = \frac{\frac{1}{S} - 1}{\frac{1}{P} - 1} \tag{2.3}$$

A problem that arises when dividing the pixels into batches is that threads may wait for the last uncompleted batch to finish, which may be slow to render due to complex material properties. This could have been solved by further dividing the last batch's pixels into new smaller batches recursively during the rendering, thus avoiding the threads from idling. Another solution could have been to assign the final batches with fewer pixels.

2.3 Ray generation and supersampling

Before the tracing of the rays can begin, it is necessary to create the primary rays, i.e. those that start from the camera. If more than one ray per pixel is generated and their color results are averaged, the pixel is said to be sampled multiple times, i.e. it is *supersampled*. Supersampling is used to reduce the noise in an image[1][2] and hide aliasing artefacts.

2.3.1 Previous work

The traditional way of generating the ray for a pixel, given by the coordinates (x_{pixel}, y_{pixel}), is by using the cameras directional vectors and the up-vector as in Equation (2.4).

$$\begin{aligned} position_{world}(x,y) =& u * camera_{up} + v * (camera_{up} \times camera_{direction}) + z_{near} * camera_{direction} \\ (u,v) =& \left(\frac{2x - width}{width} \tan(fov), \frac{2y - height}{height} \tan\left(\frac{height}{width} fov \right) \right) \end{aligned} \tag{2.4}$$

Rasterization on the other hand works by transforming vertices from model space into viewport space as described in Equation (2.5). The results of the two methods are, however, equivalent [Borman 2003].

$$position_{viewport}(p) = M_{viewport} M_{projection} M_{view} p \tag{2.5}$$

Supersampling is done by averaging multiple samples for every pixel and each sample has a coordinate within the pixel. These coordinates are represented by an offset (x_{offset}, y_{offset}), from the pixel coordinate, (x_{pixel}, y_{pixel}). The algorithms to choose these offsets are called *sampling patterns*. There are a multitude of sampling patterns. The most commonly used are listed below:

[1] This is even true in the real world and is done by using more light when filming, taking a picture or even seeing.

[2] Path tracing generates very noisy images and certain scenes require tens of thousands of traced rays per pixel to get a consistent result.

Grid sampling: The samples are arranged in a grid.

Random sampling: The samples are chosen randomly within the pixel.

Stratified sampling: The samples are arranged in a grid, and then chosen randomly within each grid cell.

2.3.2 Method

It is possible to use the same method as rasterization when generating the primary rays. This simplifies our implementation by only requiring a single description of the camera (the view matrix). This method also has the added benefit of enabling the algorithms described in Section 5.1 to work transparently. The implementation is done by transforming the pixel into viewport space: $(x_{pixel} + x_{offset}, y_{pixel} + y_{offset}, z_{near})$ where z_{near} is the distance to the near plane. This viewport coordinate is then transformed to worldspace by using the transformation matrices in Equation (2.6).

$$position_{world}(x,y) = (M_{viewport} \cdot M_{projection} \cdot M_{view})^{-1} \begin{bmatrix} x_{pixel} + x_{offset} \\ y_{pixel} + y_{offset} \\ z_{near} \end{bmatrix} \tag{2.6}$$

$$M_{viewport} = \begin{bmatrix} width/2 & 0 & 0 & width/2 \\ 0 & height/2 & 0 & height/2 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

Equation (2.6) shows the benefits of using the same matrices, (including the viewport matrix) as rasterization. OpenGL does not allow the viewport matrix to be set directly. Instead it can only be set with four parameters which are `x_offset`, `y_offset` (not to be confused with x_{offset} and y_{offset}), *width*, and *height* [Silicon Graphics 2006]. Our method is simplified by setting `x_offset` and `y_offset` to zero.

Supersampling has been implemented with a grid, a random, and a stratified pattern.

2.3.3 Results

A closeup of an edge with the different supersampling patterns can be seen in Figure 2.5.

Figure 2.5: Supersampling patterns. From left to right: **(a)** No supersampling. **(b)** Grid 2x2 supersampling. **(c)** 4 Random supersamples. **(d)** Stratified 2x2 supersampling. **(e)** Stratified 16x16 supersampling.

2.3.4 Discussion

Among all supersampling patterns, the random sampling was the slowest to converge. Jittered sampling was only slightly better. We found that the grid pattern converged fast and gave good results with few samples.

The method for generating rays described above is not the standard technique used in a ray tracer and may have been unnecessary complex. It was required, however, to make the algorithm described in Section 5.1 work.

2.4 Conclusions

This section has described the basics about the general ray tracing algorithm, followed by what a scene is and how it should be interpreted. Afterwards, different ways of handling how rays should be traversed, using threading and different patterns, was described and implemented. Lastly, how rays in the scene and supersampling should be generated, is described.

3

Trace - finding the geometry

This section will introduce a method for checking ray/triangle intersection. Intersection tests dominate the runtime for a ray tracer and a data structure will be introduced to minimize the number of tests. A K-D tree is used as the data structure which organizes the 3D data by recursively splitting the scene in the axes. Lastly, two methods will be introduced that find close to optimal splitting planes, which makes the K-D tree even faster.

3.1 Triangle intersection - the cornerstone of ray tracing

One of the most fundamental operations in a ray tracer is the intersection test between a ray and geometry. This is a fundamental operation in the ray tracer, e.g. for finding the closest visible point for a sample ray, the reflection and refraction rays, shadow casting, photon mapping etc. Thus, this section will describe one of the most common and efficient methods for ray/triangle intersection. Furthermore, it is important that not only the intersection test itself is fast and efficient, but also that the tests are intelligently done. Therefore, acceleration data structures are used to lower the number of required intersection tests. The next section discusses how this is done in the final version of the ray tracer.

3.1.1 Previous work

There are many different ways of testing intersection between a ray and geometry. Essentially, any geometry such as spheres, cylinders, cuboids, AABBs[3], slabs[4] can be tested against a ray. However, the most important intersection test in the renderer is the test between a ray and a triangle, since triangles are the most common way of storing 3D data.

Möller and Trumbore [1997] describes an algorithm that is both computationally and memory efficient and also proved to be easy to implement. The algorithm is summarized in Akenine-Möller et al. [2008] and this thesis uses the same annotations.

[3] A axis aligned bounding box, AABB, is a bounding box which is aligned with the axes of the scene.

[4] A slab is the space between two parallel planes. Slabs can be used when testing for intersection in bounding volumes.

A point in a triangle can be described by

$$(1-u-v)\mathbf{p_0}+u\mathbf{p_1}+\mathbf{p_2}, \tag{3.1}$$

where $\mathbf{p_0}$,$\mathbf{p_1}$ and $\mathbf{p_2}$ are the vertices of the triangle and u and v are parameters specifying a point on the plane defined by the three vertices. A point on a ray can be described by

$$\mathbf{o}+t\mathbf{d}, \tag{3.2}$$

where $\mathbf{o}$ is the origin of the ray, $\mathbf{d}$ is the direction and t is the parameter that specifies where on the ray the point lies. The ray will intersect this plane where these two equations equal each other

$$\mathbf{o}+t\mathbf{d} = (1-u-v)\mathbf{p_0}+u\mathbf{p_1}+\mathbf{p_2}, \tag{3.3}$$

which can be expressed as the following equation

$$(-\mathbf{d}\ \ \mathbf{p_1}-\mathbf{p_0}\ \ \mathbf{p_2}-\mathbf{p_0})\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathbf{o}-\mathbf{p_0}. \tag{3.4}$$

By letting $\mathbf{e_1} = \mathbf{p_1}-\mathbf{p_0}, \mathbf{e_2} = \mathbf{p_2}-\mathbf{p_0}$, and $\mathbf{s} = \mathbf{o}-\mathbf{p_0}$, the solution is given by Cramer's rule:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{det(-\mathbf{d},\mathbf{e_1},\mathbf{e_2})}\begin{pmatrix} det(\mathbf{s},\mathbf{e_1},\mathbf{e_2}) \\ det(-\mathbf{d},\mathbf{s},\mathbf{e_2}) \\ det(-\mathbf{d},\mathbf{e_1},\mathbf{s}) \end{pmatrix}. \tag{3.5}$$

The final solution can be further simplified to the form presented in Equation (3.6).

$$\begin{aligned} \begin{pmatrix} t \\ u \\ v \end{pmatrix} &= \frac{1}{det(-\mathbf{d},\mathbf{e_1},\mathbf{e_2})}\begin{pmatrix} det(\mathbf{s},\mathbf{e_1},\mathbf{e_2}) \\ det(-\mathbf{d},\mathbf{s},\mathbf{e_2}) \\ det(-\mathbf{d},\mathbf{e_1},\mathbf{s}) \end{pmatrix} \\ &= \frac{1}{(\mathbf{d}\times\mathbf{e_2})\cdot\mathbf{e_1}}\begin{pmatrix} (\mathbf{s}\times\mathbf{e_1})\cdot\mathbf{e_2} \\ (\mathbf{d}\times\mathbf{e_2})\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{e_1})\cdot\mathbf{d} \end{pmatrix} \\ &= \frac{1}{-(\mathbf{e_1}\times\mathbf{e_2})\cdot\mathbf{d}}\begin{pmatrix} (\mathbf{e_1}\times\mathbf{e_2})\cdot\mathbf{s} \\ (\mathbf{s}\times\mathbf{d})\cdot\mathbf{e_2} \\ -(\mathbf{s}\times\mathbf{d})\cdot\mathbf{e_1} \end{pmatrix} \end{aligned} \tag{3.6}$$

An intersection *inside* the triangle occurs when the following relations uphold:

$$u \geq 0.0, v \geq 0.0, u+v \leq 1.0. \tag{3.7}$$

Furthermore, in order for a intersection to occur *in front of* the ray's origin, t must be positive.

3.1.2 Method

In order for us to quickly get started working on the ray tracer, this implementation was combined with an array data structure where the triangle data is stored. To find the closest intersection, the array was looped through and the intersection test was applied to every triangle to find the closest one (which had the smallest positive t).

3.1.3 Result

The algorithm described above provided us with a trivial, fast, and memory efficient way to do intersection tests between rays and triangles. However, this data structure is *very* inefficient, driving us to implement a more sophisticated algorithm, which is covered in Section 3.2.

3.1.4 Discussion

Optimizations other than minimizing the number of operations made are possible by introducing SSE optimizations, which is a method to execute SIMD[5] operations. Kensler and Shirley [2006] presents a method to introduce *ray packets* where SSE instructions can be used to perform the intersection tests on several rays in parallel.

3.2 Accelerating the trace function using a K-D tree

When checking for intersection all triangles have to be checked for intersection. This is because there is no easy way of telling which triangle lies closest to the ray. It is possible to sort the triangles before checking for intersection, but since the rays may come from any direction the triangles need to be resorted almost each time and it will in the end take longer time to perform. To minimize the intersection tests the triangles need to be stored and organized.

3.2.1 Previous Work

A K-D tree, an abbreviation of k dimensional, is a spatial data structure and, depending on the data used, is usually two or three dimensional [Bentley 1975]. A K-D tree is a special case of the binary space partitioning tree (BSP tree) [Havran 2000]. A BSP tree is built by recursively partitioning space into two half-spaces[6] using a hyperplane[7] [Clairbois 2006]. Then, the two half-spaces are recursively partitioned until there are only one or a few objects left in the half-space. This half-space in then called a leaf. The half-spaces create a tree where the two half-planes are the right and left children for the partitioned space (see Figure 3.1).

[5] *Single Instruction Multiple Data* is the term describing the procedure where a single *vector instruction* are applied on multiple data, executing them in parallel. The common form of executing instruction on a single data set is called *Single Instruction Single Data* (SISD). Other terms are *Multiple Instruction Single Data* (MISD) and *Multiple Instruction Multiple Data* (MIMD) [Siewert 2009].

[6] A half-space is created when splitting the k-dimensional space with a hyperplane. The split creates two half-spaces defined by the split.

[7] A hyperplane is a generalization of the plane that exists in k-dimensional space.

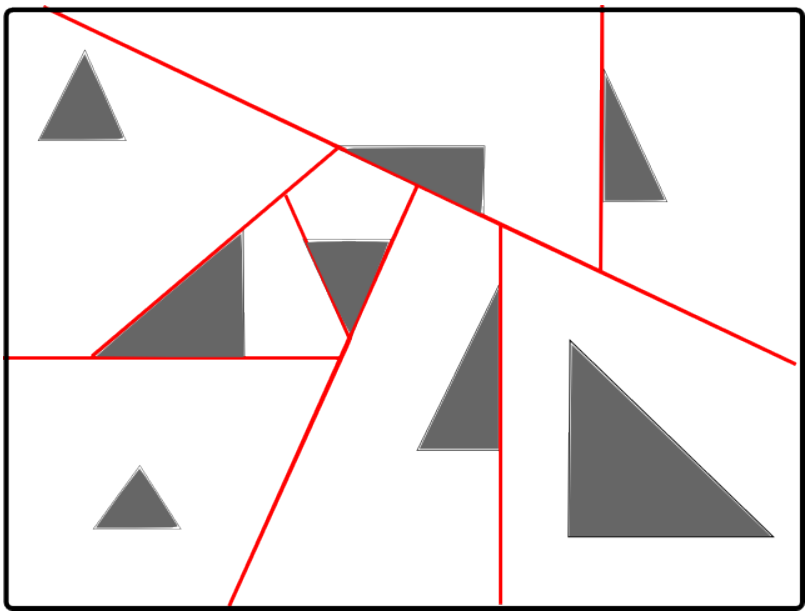


Figure 3.1: Showing splitting planes from a BSP Tree

Each node in the tree needs to contain: pointers to the left and right child, whether the node is a leaf, the split position, and which triangles are within the space. The max values on a certain axis of a triangle will be used to sort the triangles (see Figure 3.2).

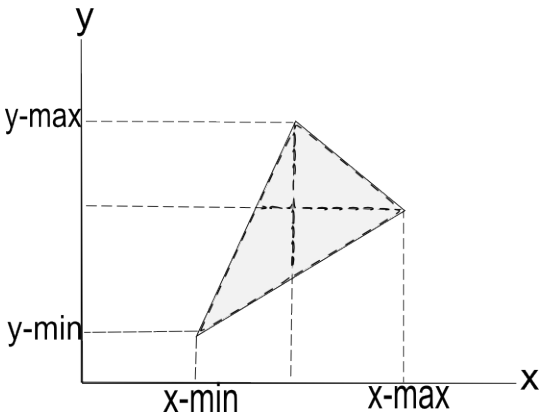


Figure 3.2: Showing which point on the triangle is used in the sorting. In a 2D space the min and max points on the y and x axis of the triangle are used.

A K-D tree is a BSP tree with the restriction that the splitting planes may only lie in the axes of the k-dimensional space [Bentley 1975]. By only choosing hyperplanes that are aligned with the axes, the building of the tree takes less time since there are fewer possible ways of choosing the hyperplanes. The disadvantage of only choosing hyperplanes aligned with the axes is that the hyperplanes will not always give optimal results. The worst case scenario is when both half-spaces contain all triangles of the parent space. This is not optimal since the split ends up not splitting nothing at all. This will result in the two

children producing identical subtrees, which will make the tree traversal worse.

A fast way to select which axis the hyperplane should use to split the space is to alternate between the axes: the first split is in the x-axis, the second in the y-axis, and the third in the z-axis. The result of this process is illustrated in Figure 3.3. Once the z-axis has been split and the node is not a leaf, the x-axis is repeatedly chosen again.

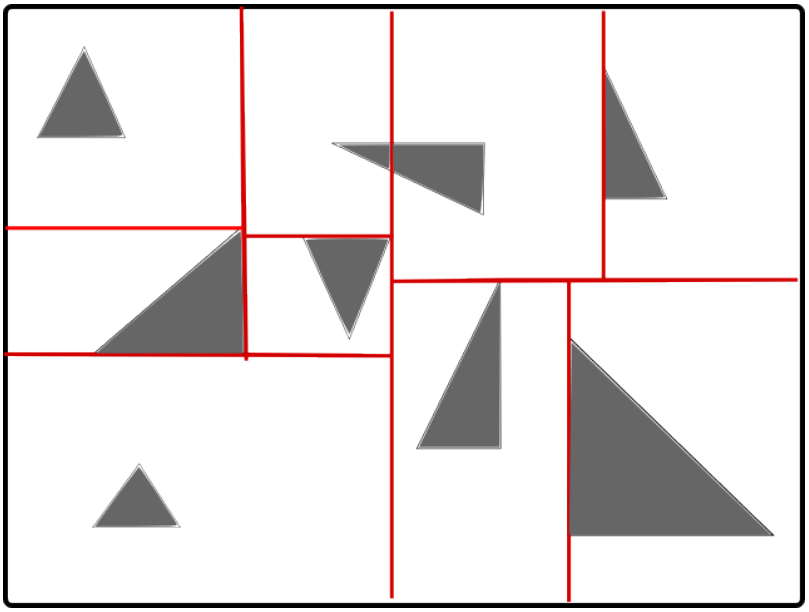


Figure 3.3: Image showing how triangles in 2D space can be split in a K-D tree.

There are several different methods for constructing a K-D tree. Wald and Havran [2006] present an algorithm for building a K-D tree. This algorithm recursively builds the tree by splitting the nodes and dividing the geometry in the left and right child nodes. Horn et al. [2007] introduced an algorithm that traverses a K-D tree using a stack implementation. The most common way to construct the tree is to use a recursive algorithm, as described by Pharr and Humphreys [2004]. Pharr and Humphreys [2004] also presents an algorithm for creating an AABB and checking ray/AABB intersection.

A crucial step to perform before the traversal is to create a bounding box around the scene. In the case of a K-D tree, the bounding box can be simplified to an axis-aligned bounding box, AABB.

The AABB for the scene is created when the scene is loaded. The minimum and maximum triangle points in the scene are used to setup the points of the AABB (See Figure 3.4).

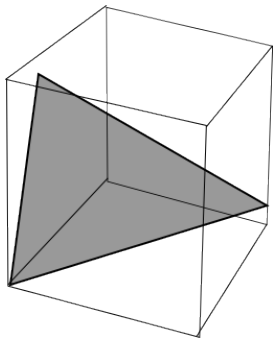


Figure 3.4: Scene containing one triangle in 3D space where a AABB is used to encapsulate the triangle.

For each ray in the scene the intersection against the tree has to be calculated. There is no need to traverse the tree if the ray misses the AABB. But if it hits, the tree is traversed by passing the intersection points and the entry and exit points, of the AABB to the traversal algorithm. Algorithm 2 shows the intersection test between a ray and an AABB.

Algorithm 2 Checking ray against AABB intersection

function INTERSECT(r,$AABB$,min_value,max_value)a
  $(inv_x,inv_y,inv_z) \leftarrow (1/r.dir_x, 1/r.dir_y, 1/r.dir_z)$
  $(min_x,min_y,min_y) \leftarrow ((AABB.pos_x - r.pos_x) * inv_x,$ ▷ calculate the distance to three slabs
  $(AABB.pos_y - r.pos_y) * inv_y,$
  $(AABB.pos_z - r.pos_z) * inv_z)$
  $(max_x,max_y,max_y) \leftarrow ((AABB.pos_x + AABB.size_x - r.pos_x) * inv_x,$ ▷ calculate the distance
  $(AABB.pos_y + AABB.size_y - r.pos_y) * inv_y,$ ▷ to the remaining three slabs
  $(AABB.pos_z + AABB.size_z - r.pos_z) * inv_z)$
  $min\_value \leftarrow max(min(min_x, max_x),$ ▷ calculate the max entry intersection
  $min(min_y, max_y),$
  $min(min_z, max_z))$
  $max\_value \leftarrow min(max(min_x, max_x),$
  $max(min_y, max_y),$
  $max(min_z, max_z))$
  if $max\_value < 0$ **then**
    return false
  else if $max\_value < min\_value$ **then**
    return $false$
  else
    return $true$
  end if
end function

The traversal starts by creating two variables, `best_intersection` and `best_t`, which are used to determine which intersection test result to return. Some optimizations are also done to reduce divisions in the tree traversal. This approach uses the same techniques used when creating the tree by creating stacks instead of using a recursive approach. The stacks start with the root node and the min and max values are passed through to the method.

The tree will be traversed until it has tested all possible intersections, which will be $\log(n)$ possibilities

on average. All possible intersections have been tested when the node stack is empty. When entering the loop, the node, max, and min values are extracted from the stacks. Every node the ray intersects will be traversed and all leaf nodes will have their triangles checked.

Another loop will be used to traverse the current node and it will run until it reaches a leaf. First the steps `t` from the origin of the ray to the splitting plane of the node is calculated. There are three possible ways of choosing a node once the steps are known:

a. The ray does not intersect with the splitting plane, or only intersects one side.

b. The ray intersects the splitting plane before the node.

c. The ray intersects both nodes.

In case **c**, the node that is furthest away is added to the node stack. The split value is then pushed to the min stack, and the max value is pushed to the max stack. The current max value is changed to the split value.

Once the traversal reaches a leaf, it exits the loop and checks all the triangles in the node for intersection (see Section 3.1). If the closest intersection is closer than the previous best intersection the return value is updated.

In Figure 3.5 it can be seen how the ray traverses the tree and selects the closest triangle.

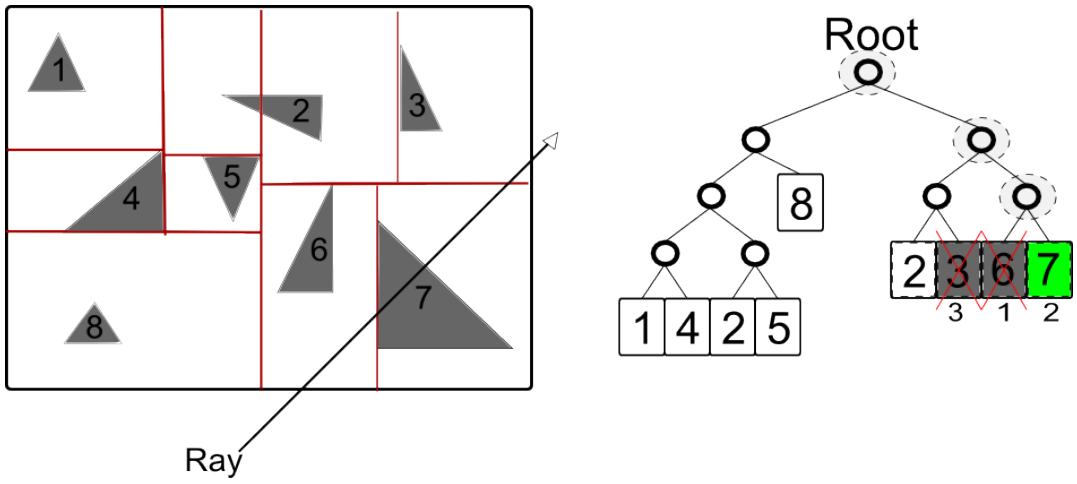


Figure 3.5: Left Shows a ray intersecting with the scene. The splits of the scene can also be seen. **Right** Shows how the ray traverse down the tree. The triangles with a cross are the triangles that have been checked for intersection; the triangle with green text is the triangle that was the closest intersecting triangle.

3.2.2 Method

The building phase uses both the algorithm for building presented by Wald and Havran [2006] and the stack concept introduced by Horn et al. [2007]. To make the later steps in the building process faster, optimizations need to be done before starting building. For each triangle an object, called `KDTriangle`, is created which contains the maximum and minimum values for each axis on the triangle.

Algorithm 3 Building a K-D tree with median split

Require: root node is created and root triangles are sorted on the x-axis
  function BUILD_MEDIAN_TREE($root, root\_triangles$)
    depthstack $D \leftarrow (\emptyset \cup 0)$
    nodestack $N \leftarrow (\emptyset \cup root)$
    trianglestack $T \leftarrow (\emptyset \cup root\_triangles)$
    while N is not $\emptyset$ **do**
      $depth \leftarrow D.top, D.pop$
      $axis \leftarrow depth \bmod 3$
      $tri \leftarrow T.top, T.pop$
      $node \leftarrow N.top, N.pop$
      if $|T| < MIN\_TRIANGLES \wedge depth < MAX\_DEPTH$ **then**
        $node \leftarrow (axis, is\_leaf, triangles)$
      else
        $split \leftarrow triangles_{size/2}.max_{axis}$
        list $triangles\_left = (T < split), triangles\_right = (T > split)$
        $sort(triangle\_left, axis + 1)$
        $sort(triangle\_right, axis + 1)$ ▷ sort the triangles with the next axis
        $N \leftarrow N \cup (node\_right \cup node\_left)$ ▷ push the new values to the top of the stacks
        $T \leftarrow T \cup (triangle\_rigmentedinht \cup triangle\_left)$
        $D \leftarrow D \cup (depth + 1 \cup depth + 1)$
        $node \leftarrow (axis, split, left\_node, right\_node)$
      end if
    end while
  end function

With the pre-building phase complete, the tree building can start. There are several ways to approach this. The algorithm shown in Algorithm 3 demonstrates in pseudocode how this is implemented.

Often these types of algorithms are implemented recursively. When a recursive method calls itself it stores all the values from the "calling method" to the call stack. Recursing numerous depths down can be problematic since there is no reasonable way of handling call stack overflow. By instead using internal stacks within the method, the amount of data stored on the call stack is minimized. Memory may still run out, but it will be easier to handle since debugging recursive methods are difficult.

Since the first hyperplane is chosen along the x-axis, the `KDTriangle` s need to be presorted along the x-axis. The value used for the sorting is the `triangle.max[x-axis]` , which is the point on the triangle with the highest value in the x-axis (see Figure 3.2). The triangles in the root node will therefore be sorted with using the x-coordinates of the triangles from the smallest to the largest.

The algorithm creates two types of nodes: a normal node and a leaf node. To determine if the node should be a leaf, either of two criterions need to be met:

- The current node contains fewer triangles than a specified minimum.
- The current tree depth is larger than a specified minimum.

There are no standards to what these constants should be. For this implementation, only the triangle count is used as termination criteria. The constant is set to $\log(n_{triangles}) * 8$. This number comes from experimenting with different values. Usually this constant is set manually depending on the scene. An

example could be a scene with a huge amount of triangles, if the constant is too low a large amount of nodes will be created and will take up very much memory.

If either one of these criterions are met, the node becomes a leaf. If the criterions are not met the algorithm need to split the current node. The first step is to sort the triangles using the next axis. For example, if the current axis is the x-axis the triangles will be sorted along the y-axis. Once the sorting is done, the splitting plane for the current node is determined by using the triangle in the middle of the sorted list. The triangles are then organized on the left and right side of the split into two lists, which are pushed to the stack. Two new nodes are also created and set to the left and right child of the current node, finally to be added to the stack (see Figure 3.6).

The algorithm will run until the node stack is empty and then the tree is completed.

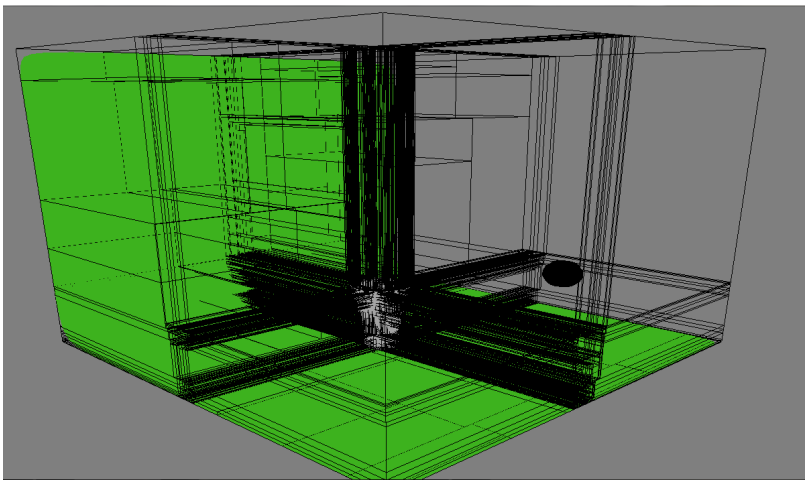


Figure 3.6: Showing a 3D scene where the splitting planes are visible. The scene is built using a K-D tree using the median split build technique.

Traversal

The traversal follows the implementation shown by Horn et al. [2007] which uses a stack approach to traverse the tree. The Algorithm 4 describes how it is implemented.

3.2.3 Results

When building using a K-D tree, depending on the amount of triangles in the scene the build time of the tree will vary. Triangle count, tree build time, max tree depth, node count, and leaf count of tree can be seen in Table 3.1.

Algorithm 4 Traversing a K-D tree looking for intersection

function FIND_CLOSEST_INTERSECTION(R,min,max,$root$) **return** I
  minstack $MIN \leftarrow (\emptyset \cup min)$
  minstack $MAX \leftarrow (\emptyset \cup max)$
  nodestack $N \leftarrow (\emptyset \cup root)$
  $dir\_inv \leftarrow (1/r.dir_x, 1/r.dir_y, 1/r.dir_z)$
  $I \leftarrow miss$
  $best_t \leftarrow \infty$
  while N is not $\emptyset$ **do**
    $node \leftarrow N.pop$
    $axis \leftarrow node_{axis}$
    $min \leftarrow MIN.pop$
    $max \leftarrow MAX.pop$
    while $node$ is not leaf **do**
      $split\_dist \leftarrow node_{split} - r.pos_{axis}$
      $split\_t \leftarrow split\_dist * dir\_inv_{axis}$
      if $split\_t < 0 \wedge split\_t > max$ **then**

$$node \leftarrow \begin{cases} node_{right} & split\_dist < 0 \\ node_{left} & \text{otherwise} \end{cases}$$

      else if $split\_t < min$ **then**

$$node \leftarrow \begin{cases} node_{right} & split\_dist > 0 \\ node_{left} & \text{otherwise} \end{cases}$$

      else

$$first \leftarrow \begin{cases} node_{right} & split\_dist > 0 \\ node_{left} & \text{otherwise} \end{cases}$$

$$second \leftarrow \begin{cases} node_{right} & split\_dist > 0 \\ node_{left} & \text{otherwise} \end{cases}$$

        $N \leftarrow (N \cup second)$
        $MIN \leftarrow (MIN \cup split\_t), MAX \leftarrow (MAX \cup max)$
        $node \leftarrow first, max \leftarrow split\_t$
      end if
    end while
    $I\_new \leftarrow ArrayTriangleIntersection(node_{triangles})$
    if $I\_new_t < best_t$ **then**
      $I \leftarrow I\_new$
    end if
  end while
  return I
end function

Table 3.1: Showing: triangle count of the scene, the tree build time in seconds, the maximum tree depth of the tree, the total node and leaf count (see Figure 3.7).

Scene	Triangles	Build time (s)	Max Tree Depth	Nodes	Leafs
3 balls	1452	0.01	10	558	280
Car	62724	5.49	24	14632	7279

When traversing the scene using a K-D tree gives a considerable speedup compared with a naive array data structure. It can be seen in Table 3.2 how using a K-D tree gives better performance, compared to the array data structure, with increasing number of triangles in a scene.

Table 3.2: Showing the render times in seconds for two scenes using Median split, and the array data structure (see Figure 3.7).

Scene	K-D Tree	Array data structure
3 balls	41.63	448.35
Car	64.40	5376.44

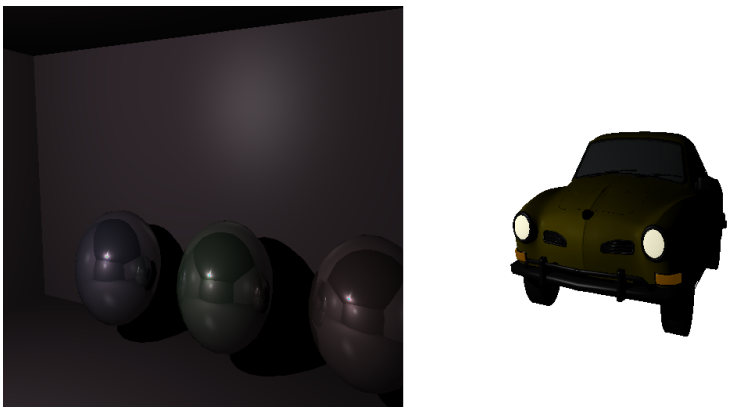


Figure 3.7: Left: Showing image rendered from the scene `3 balls`. **Right**: Showing image rendered from the scene `Car`.

3.2.4 Discussion

There are several different ways of constructing a data structure to accelerate intersection tests done in ray tracing. The most common data structures are the bounding volume hierarchies, BVH, and the spatial data structures. Recent papers have shown that using BVHs are better when ray tracing dynamic scenes (used commonly when animated movies should be rendered), since they are very fast when rebuilding the tree [Knoll et al. 2011].

The reasoning behind using a spatial data structure, instead of a BVH, was that our implementation's focus was not to render animated movies, but rather to render still images as realistic as possible.

When rendering scenes with few triangles, less than 100, the total speedup gained is fairly low. However, when increasing the amount of triangles it becomes apparent that using an accelerated data structure is very useful.

3.3 Optimizing the K-D tree using SAH

When using median split many triangles end up in several nodes since the split is not in most cases optimal; the need for finding better split positions is needed. *Surface Area Heuristics* (SAH) is another algorithm that can be used to find better splitting positions [Hunt et al. 2006].

3.3.1 Previous Work

The SAH chooses hyperplanes that has the lowest cost where the cost describes how expensive it would be to split at the chosen hyperplane. The expected cost for a given hyperplane is calculated by adding the cost of traversing the tree plus the expected cost of intersecting the two sub nodes.

The probability P of a ray hitting a sub node V_{sub} is

$$P_{[V_{sub}|V]} = \frac{SA(V_{sub})}{SA(V)}, \tag{3.8}$$

where $SA(V)$ is the surface area of the node. The SAH cost C_V of splitting a node is

$$C_V(p) = K_T + P_{[V_l|V]}C(V_l) + P_{[V_r|V]}C(V_r). \tag{3.9}$$

It is too expensive to calculate the expected cost. Instead, an estimation is done called the Local Greedy SAH.

$$C_V(p) \approx K_T + K_I\left(\frac{SA(V_L)}{SA(V)}|T_L| + \frac{SA(V_R)}{SA(V)}|T_R|\right) \tag{3.10}$$

This simplification tends to overestimate the cost which will in some cases give a lower cost than desired. But overall the approximation is good, and so far no better approximation has been found [Wald and Havran 2006].

To determine whether the node should become a leaf the SAH approach provides good criteria of when to terminate. If the cost of not splitting a node is lower than the cost of splitting a node, the node becomes a leaf. The cost of not splitting is trivial since it just the cost of traversing the tree times the amount of triangles.

There are an infinite number of potential hyperplanes, thus it is required to determine which hyperplane candidates to choose from. For any possible pair of planes (a,b) between which the number of triangles on each side remain the same, the cost will be the same for any possible plane between a and b. Therefore the possible split candidates will only be at the places where the number of triangles on the left side and right side change.

The intuitive approach would be to use the 6 planes, two in every axis, defined by the triangle AABB. But doing it that way is inaccurate and it may sort triangles into nodes that the triangle should not belong. A better way is to clip the triangles to the node AABB and then use the 6 planes defined by the clipped triangle. These split candidates are called perfect splits.

The last unknown parameters that need to be known in order to calculate the cost is N_l, N_r, V_l and V_r. V_l and V_r are trivial since they are the split of the parent node AABB using the splitting plane. To calculate N_l and N_r, an optimization can be made by placing triangles lying on the splitting plane on only one of the sides. This is done by introducing another parameter N_p which contains all triangles lying on the splitting plane. Instead of one, two SAH costs have to be calculated; one with $N_l + N_p$ and one with $N_r + N_p$.

The most trivial way of calculating the lowest SAH cost for a plane would cost $O(n^2)$ which is only feasible in scenes with few triangles. Therefore, clever algorithms need to be used to achieve a reasonable time.

Pharr and Humphreys [2004] presents an algorithm that finds the lowest SAH cost with the complexity $O(n \cdot \log^2 n)$. This algorithm finds the SAH for a bounding volume data structure but can be modified to work for a spatial data structure.The algorithm was initially introduced by Szécsi [2003]

SAH in $O(n \cdot \log^2 n)$

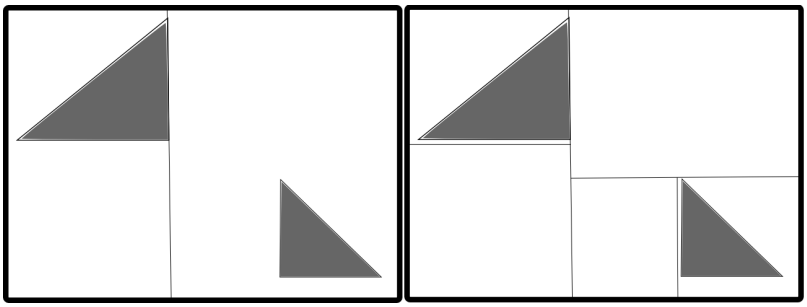


Figure 3.8: Compare BVH SAH split to K-D split **Left**: shows tree build using median split. **Right**: shows tree built using the SAH. Since the SAH considers empty space the triangles are better encapsulated than with the median split.

As shown in Figure 3.8 the hyperplane creates half-spaces that together describes the parent node. In the BVH implementation the hyperplane defines two bounding volumes that do not have to describe the parent node. This is the major difference that needs to be taken into consideration when implementing the algorithm from Pharr and Humphreys [2004].

There are three big steps in finding the lowest SAH for a specific node:

1. Create “events” for the splitting candidates
2. Sort the events
3. Calculate SAH for all events (with a few exceptions) and return lowest SAH cost

The first step is creating the events. An event is basically a split candidate for a certain axis containing one extra parameter to say whether it is a minima, maxima or planar split candidate of a triangle. The

reasoning behind this is that when stepping through the events looking for the best split candidate we do not want to check split candidates that split at the same location more than once.

When adding the events we use the perfect splits of each triangle in the current node and either

1. add a planar event if the triangle is planar,
2. or add two events, the triangle perfect split minima and maxima values for the current axis.

Once all events have been created they need to be sorted. They are sorted based on split candidate value, and if two values are the same the type is used where

$$(-,|,+) = (0,1,2). \tag{3.11}$$

The third step is to traverse the events. Since the algorithm is handling one axis at a time the events will be sorted from the smallest to the largest on a certain axis. Sorting this way eliminates having to check events that have the same split values. This means that the total amount of times the SAH needs to be calculated often is smaller than the total amount of events.

When all events with different split values have been been calculated the lowest SAH cost has been found.

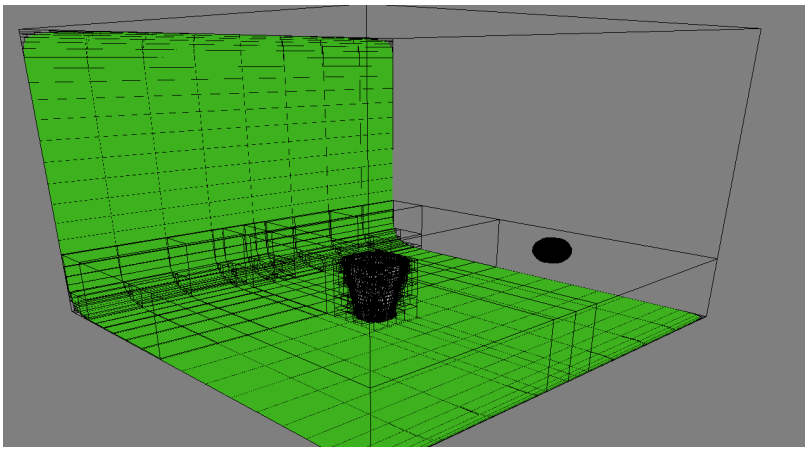


Figure 3.9: Showing a 3D scene where the splitting planes are visible. The scene is build using the SAH-based K-D tree.

SAH in $O(n \cdot \log n)$

The main problem with the $O(n \log^2 n)$ algorithm occurs when the events are sorted. In order to make it faster the sorting needs to be avoided each time the lowest SAH is calculated. Wald and Havran [2006] presents an algorithm that only sort the list once at the beginning. Before starting to build the tree a list of all the events is created that contains events for all the axes. The list is thereafter sorted using the same sorting done in the $O(n \log^2 n)$ implementation.

Finding the plane is essentially the same as the previous implementation, but some parts have been removed or changed. Since the events have already been created and sorted, there is no need to sort

them again. The only remaining loop when finding the plane is the event loop. Because the event list contains events from all axes it is necessary to differentiate between them when updating the number of triangles on each side of the splitting plane.

The other major part of the algorithm is finding out which events should belong to the two half-spaces that the split created. Finding out which triangles to belong to which side can be done in $O(n)$ time by classifying all the triangles in the node to be either:

- Both - The triangle should be in both half-spaces
- Only Left - Triangle should only be in the left half-space
- Only Right - Triangle should only be in the left half-space

Once they have been classified, four event lists are created

1. Events only in left half-space
2. New events clipped to the left half-space
3. Events only in right half-space
4. New events clipped to the right half-space

Since we are creating new events they need to be sorted. But since the old events are already sorted we only need to sort the new events. On average the amount of new events are not more than $\log(n)$, and sorting $\log(n)$ events can be done in $O(n)$. Once they are sorted the left and right lists are merged and are used when calculating the SAH for the two new half-spaces.

3.3.2 Method

Algorithm 5 shows how the SAH-based K-D tree build algorithm, which lies in $n\log^2(n)$ complexity, is implemented.

The algorithm for finding the plane using the algorithm that builds the tree in complexity $O(n \cdot \log(n))$. The splitting planes are the same as in the $O(n \cdot \log^2(n))$ algorithm since both algorithms build the same tree. (see Figure 3.9)

3.3.3 Results

When building a SAH-based K-D tree, depending on the amount of triangles in the scene the build time of the tree will vary. Build time and other information of tree can be seen in Table 3.3.

Table 3.3: Showing: triangle count of the scene, the SAH-based tree build time in seconds, the maximum tree depth of the tree, the total node and leaf count (See Figure 3.10).

Scenes	Triangles	Build time (s)	Max Tree Depth	Nodes	Leafs
3 balls	1452	0.5	25	16872	8437
Car	62724	24.50	39	577332	288667

Algorithm 5 Building K-D tree with SAH split in $O(n \cdot \log^2 n)$

function FIND_CLOSEST_INTERSECTION($T\{t0,t1,..\},V$) **returns** *best_split*
  $best\_split \leftarrow (INF,0,-1,0)$ ▷ contains (SAH cost, split, axis , side)
  N_l,$N_p \leftarrow 0$, $N_r \leftarrow |T|$
  eventlist $E \leftarrow \emptyset$
  for $i = 0 \rightarrow 3$ **do**
    for $t = 0 \rightarrow |T|$ **do**
      $triMin \leftarrow max(T_i.min_k, V.pos_k)$
      $triMax \leftarrow min(T_i.max_k, V.pos_k + V.size_k)$
      if $triMin = triMax$ **then** ▷ | = planar split candidate, - = minimum, + = maximum
        $E \leftarrow (E \cup Event(triMin, |))$
      else
        $E \leftarrow (E \cup Event(triMin, -) \cup Event(triMax, +))$
      end if
    end for
  end for
  $sort(E, < split)$
  for $i = 0 \rightarrow |E|$ **do**
    $p_+ \leftarrow p_- \leftarrow p_|$
    $split \leftarrow E_i.split$
    while $i < |E| \vee E_i.split = split \vee E_i.type = -$ **do**
      inc p_- $\vee$ inc i
    end while
    while $i < |E| \vee E_i.split = split \vee E_i.type = |$ **do**
      inc $p_|$ $\vee$ inc i
    end while
    while $i < |E| \vee E_i.split = split \vee E_i.type = +$ **do**
      inc p_+ $\vee$ inc i
    end while
  end for
  $N_p \leftarrow p_| \vee N_r \leftarrow N_r - (p_| + p_-)$ ▷ Update number of triangles on each side
  if $split < V.pos_k + EPS \cup split + EPS > V.pos_k + V.size_k$ **then**
    ▷ Handle splits that produce very small AABB that can give numerical instability
    $current\_split \leftarrow (INFINITY,0,LEFT)$ ▷ contains cost,split and side
  else
    $current\_split \leftarrow SAH(V,k,split,N_l,N_r,N_p)$ ▷ returns cost,split and side
  end if
  if $current\_split_{cost} < best\_split_{cost}$ **then**
    $best \leftarrow (current\_split_{cost}, split, k, current\_split_{side})$
  end if
  $N_l \leftarrow N_l + (p_+ + p_|)$
  $N_p \leftarrow 0$
end function

It can be seen in Table 3.4 how using a SAH-based K-D tree gives better performance, compared to a median-based K-D tree and not using any data structure, the more triangles the scene contains.

Table 3.4: Showing the render times in seconds for two scenes using SAH, Median split, and the array data structure (See Figure 3.10).

Scene	K-D Tree	Array Data Structure	K-D Tree using SAH
3 balls	41.63	448.35	5.12
Car	64.40	5376.44	2.01

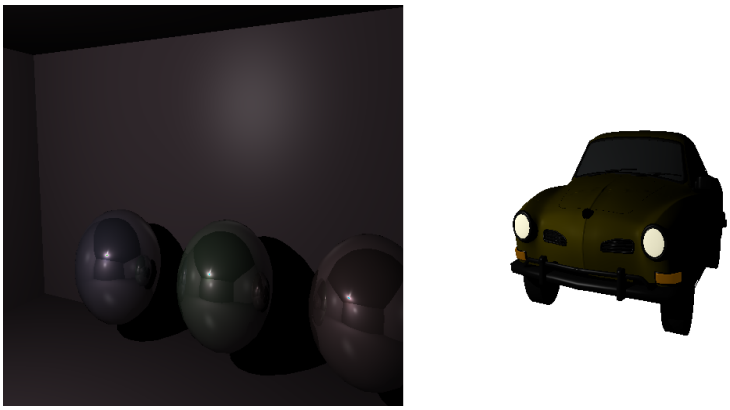


Figure 3.10: Left Showing image rendered from the scene `3 balls`. **Right** Showing image rendered from the scene `Car`.

3.3.4 Discussion

The time it takes to render an image can be greatly improved when using SAH compared to median split. Depending on the amount of triangles, the SAH split performs faster compared to the median split when using more triangles.

As seen in the results section, the improvement when using a SAH-based K-D tree over median split K-D tree gives better performance the more triangles are used in the scene. In Table 3.4 it can be seen that `'car` performs better than `'3 balls` even though car has more triangles. This is mainly because fewer rays hit the AABB of the car scene, and therefore less triangle intersection tests need to be performed.

3.4 Conclusions

In this section, methods for efficiently checking ray intersection against triangle and AABB has been shown and implemented. A data structure which stores and organizes the 3D data in an efficient and accessible way was also shown. Lastly two different ways of finding better split positions using the SAH which greatly reduces the rendering time.

4

Shade - painting the canvas

This section will first describe the fundamental process of colorizing the scene, the algorithm for our shading. Secondly, the section extends this with further features to enhance the image quality. Different shadowing techniques, reflections and refractions, the texture implementation, and how textures can be used to simulate tiny bumps on the surface are all discussed in their own subsection.

4.1 Shading - our lighting model

This section discusses one of the most important parts in a ray tracer: the shading process. This is a fundamental process that decides what color each pixel should have. First, the basic problem will be expressed in an equation, and an approximation to solve this is presented. This is followed up by our approach and results of the solution. Finally a discussion of the model, possible enhancements and extensions are given.

4.1.1 Previous work

The basic problem with rendering is to calculate the color of each pixel on the screen. This problem has been redefined as the goal to solve *the rendering equation*, first formulated by Kajiya [1986] and Immel et al. [1986], which can be written as

$$L_o(\mathbf{x}, \omega) = L_{ve}(\mathbf{x}, \omega) + \int_{S^2} f_r(\mathbf{x}, \omega, \omega') L_i(\mathbf{x}, \omega') cos\,\theta\, d\omega'. \tag{4.1}$$

The outgoing light L_o from position x towards the direction of the eye ω can be interpreted as the emissive light of the surface plus the *bidirectional reflectance distribution function* (BRDF) times the incident light attenuated by the angle between the incident and outgoing light for all directions in the hemisphere. The BRDF, a function introduced by Nicodemus [1977], describes how light is reflected for a given surface by using the ratio between the incident and outgoing light. In layman's terms, it describes the look of the surface.

However, the BRDF only takes reflections into account and does not consider how light interacts *inside* the material. Therefore, Nicodemus [1977] also suggest another function called *bidirectional surface scattering reflectance distribution function* (BSSRDF). This function is capable of describing the effect of subsurface scattering. Subsurface scattering is when light enters the surface at one point, scatters in the material and exits at another point on the surface. There exist other functions that describe how light interacts with a surface; such as the *bidirectional transmittance distribution function* (BTDF), which describes how light is transmitted through the material, and the *bidirectional scattering distribution function* (BSDF) [Veach 1997], which is a combination of the previous two.

The BRDFs equation is quite complicated to implement but there exists a number of shading models that simplifies this. One of the most common models is the Blinn-Phong shading model [Blinn 1977]. This algorithm uses a combination of three components to evaluate the perceived light

$$L = L_a + L_d + L_s, \tag{4.2}$$

where L is the perceived light, L_a is ambient light (usually a small constant), L_d is the diffuse light and L_s the specular. The diffuse component represents light that are scattered equally in all directions, see Figure 4.1, and is mathematically expressed as

$$L_d = max(0, \vec{n} \cdot \vec{l})\, m_d \otimes s, \tag{4.3}$$

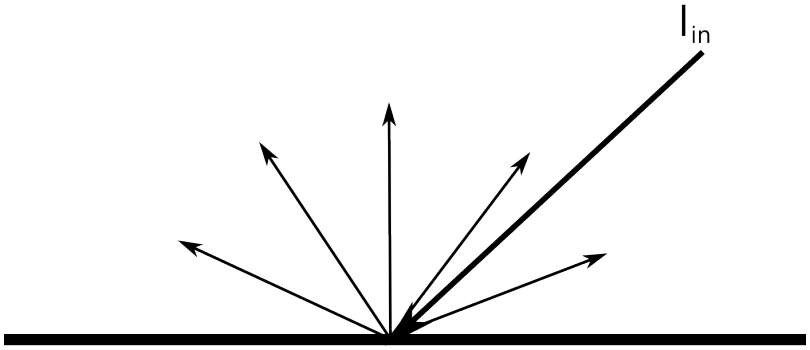


Figure 4.1: Lambertian reflection. The incident light $\vec{l}_{in}$ is equally scattered in all directions.

where $\vec{n}$ is the surface normal, $\vec{l}$ the direction to the lightsource, m_d the material color and s the light intensity (including color). The final component L_s represents specular highlight, typically seen on shiny objects such as bright spots which are a result of the reflectiveness in the material. Phong [1975] used the following equation to model specular highlights

$$L_s = max(0, \vec{r} \cdot \vec{v})^{m_{shi}} m_{spec} \otimes s, \tag{4.4}$$

where $\vec{r}$ is the light reflected on the surface, $\vec{v}$ is the view vector, m_{shi} controls the *shininess* of the surface and m_{spec} is the specular color of the material. This is only valid if $\vec{n} \cdot \vec{l} > 0$. Otherwise, $L_s = 0$. Blinn

[1977] modified this to use the scalar product of the surface normal with the *halfway vector* instead of the reflection and view vector. The halfway vector is defined as

$$\vec{h} = \frac{\vec{l} + \vec{v}}{\left|\vec{l} + \vec{v}\right|}. \tag{4.5}$$

4.1.2 Method

After the closest intersection is found as explained in Section 3, data such as position, color, normal, incident light, and colors from other effects (reflection, refraction, global illumination etc.) is given as input to the algorithm described by Blinn [1977].

4.1.3 Results

The results of this shading model are shown in Figure 4.2. It shows that the Blinn-Phong model can be used to simulate a wide range of materials. Although, it is quite limited since it does not simulate any kind of subsurface scattering.

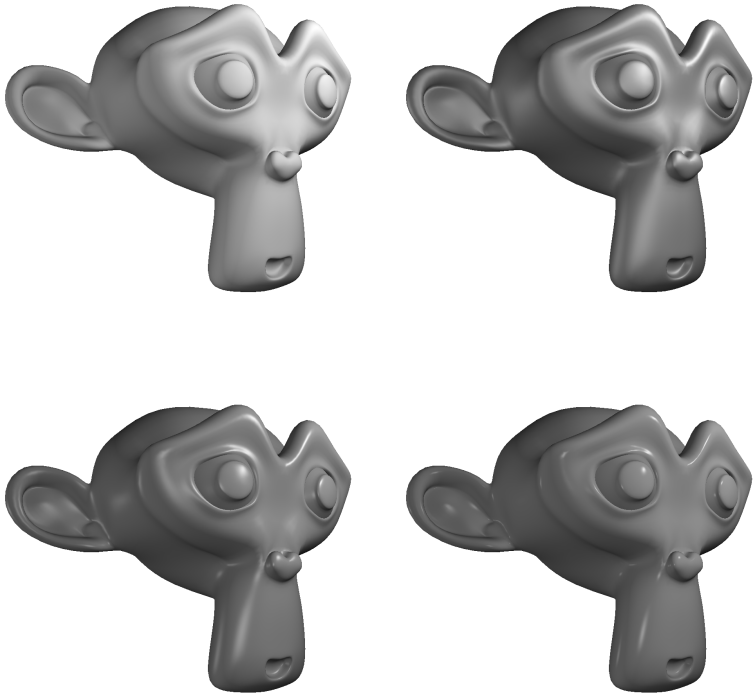


Figure 4.2: A monkey rendered using the Blinn-Phong shading model with a Three-point lighting setup. The top-left monkey is rendered with no specular and the monkey looks completely diffuse, almost like chalk. The top-right and bottom-left images have a shininess factor of 20 and 96 respectively. Note how the bottom-right image with an extreme shininess value of 500 almost looks lacquered.

4.1.4 Discussion

The implemented shading model is somewhat limited and only simulates a very special type of material. To render truly realistic images, a more complex shading model should be implemented, preferably one that accurately simulates more complex BRDFs or even BSSRDFs. Other plausible shading models that could be used in the renderer are the Cook-Torrence model [Cook and Torrance 1981], Oren-Nyar model [Oren and Nayar 1994], or the Ward anisotropic distribution [Ward 1992] etc.

However, only providing built in shading models may not be a suitable solution for a production ready rendering engine. When artistic freedom is a top priority, using a *shading network* could be a valid solution. This allows the user to implement their own shading models either using a node tree, such as the one implemented by Blender Foundation [2006], or using a shading programming language.

One possible solution could be to implement the open RenderMan API specification by Pixar Animation Studios [2005], which also includes the RenderMan Shading Language. Any RenderMan compliant shader would then be possible to use in the renderer. Another possibility would be to implement Sony Pictures Imageworks' Open Shading Language [Gritz 2010] which is another shader standardization. Using this project also has the advantage that Sony offers open source code of the implementation, which makes it much easier to implement.

4.2 Shadows

Shadows are one of the most important components when rendering realistic-looking images, since they give the viewer a visual aid for depth perception (see Figure 4.3). When shadows are absent, it is difficult to determine the exact position of an object in 3D space. This section discusses shadows and gives three examples of popular shadow rendering algorithms. A stochastic method for soft shadows is then explained, as well as a brief implementation of shadow caches.

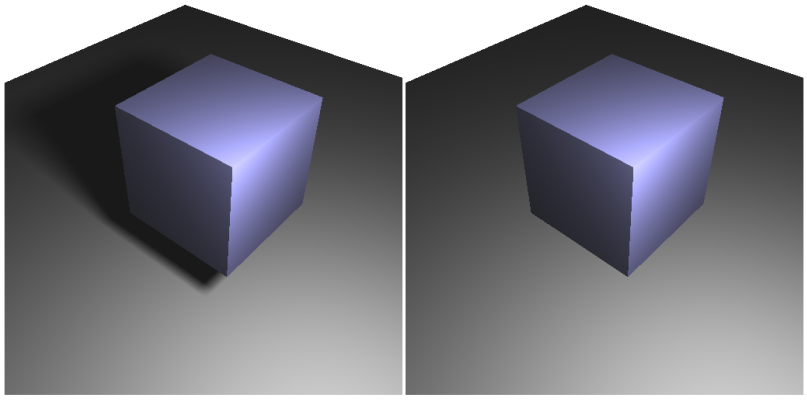


Figure 4.3: Left: With shadow. **Right**: Without shadow.

4.2.1 Previous work

The regions of a shadow can be divided in two separate parts (see Figure 4.4): the umbra and the penumbra. The umbra is the region which is completely blocked by direct light, while the penumbra is the transitional region towards light surrounded by the umbra. Shadows can be grouped into two main categories: hard shadows and soft shadows [Woo et al. 1990]. The difference is that hard shadows only have an umbra. Hence, hard shadows are binary, meaning that the surface is either lit or not.

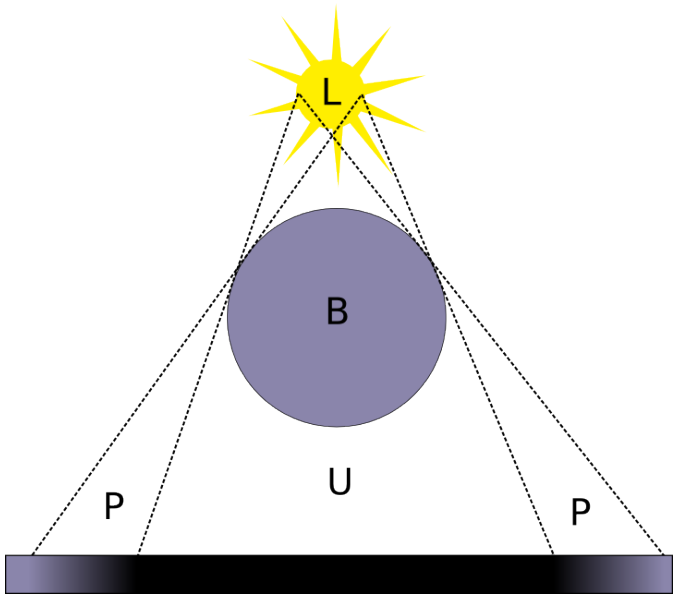


Figure 4.4: Showing the shadow regions: the umbra (U) which is completely occluded by the blocker (B) from the light (L), and the penumbra (P) which is partially blocked.

Shadow tests can be done in a couple of different ways:

Shadow mapping A fast method, commonly used in real-time rendering, introduced by Williams [1978], where shadows are created by comparing the depth value, rendered from the light's point of view (called the shadow map), with the tested point's distance to that light. If that distance is greater than the shadow map's depth value, the point is in shadow. The shadows can, however, be jagged if the resolution of the shadow map is too low. By increasing it the sharpness of the shadows will be further improved.

Shadow volumes This technique, also commonly used in real-time rendering, was introduced by Crow [1977]. In contrast to shadow mapping, it creates shadows which are sharp, but scales poorly with the scene's complexity. The shadow volumes are created by extending each triangle from its edges to infinity with three quads in the opposite direction of the light. These three quads linearly span a volume, hence the name shadow volume. When testing if a point is in shadow, the number of shadow volumes the ray (traced from the camera) is entering and leaving is counted until the point tested for shadow is reached. If the number of entered volumes is larger than the exited ones, the point is in shadow. Later, with the use of the stencil buffer, the shadow volume technique was further developed, by Heidmann [1991] to enable shadow volumes in real-time.

Shadow rays This method checks for intersections between the light and the tested point, which is the standard ray tracing technique [Whitted 1980]. When a ray intersects a triangle, tests are made to see if any of the light sources can be seen from that point. This is achieved by sending a *shadow ray* from each light source to the intersected point and test for any intersections between. If an intersection was found, the light is blocked, and thus the point is in shadow, otherwise the point is in light.

4.2.2 Method

Shadows in the ray tracer have been implemented using shadow rays and intersection tests as explained above. To avoid self-shadowing and surface acne, a value ε is used to assure that the distance from the light to the blocker is not too small. As can be seen in the first column of Figure 4.5, this will give perfectly sharp shadows. This is because just a single shadow ray is being shot; the point will either be blocked or not.

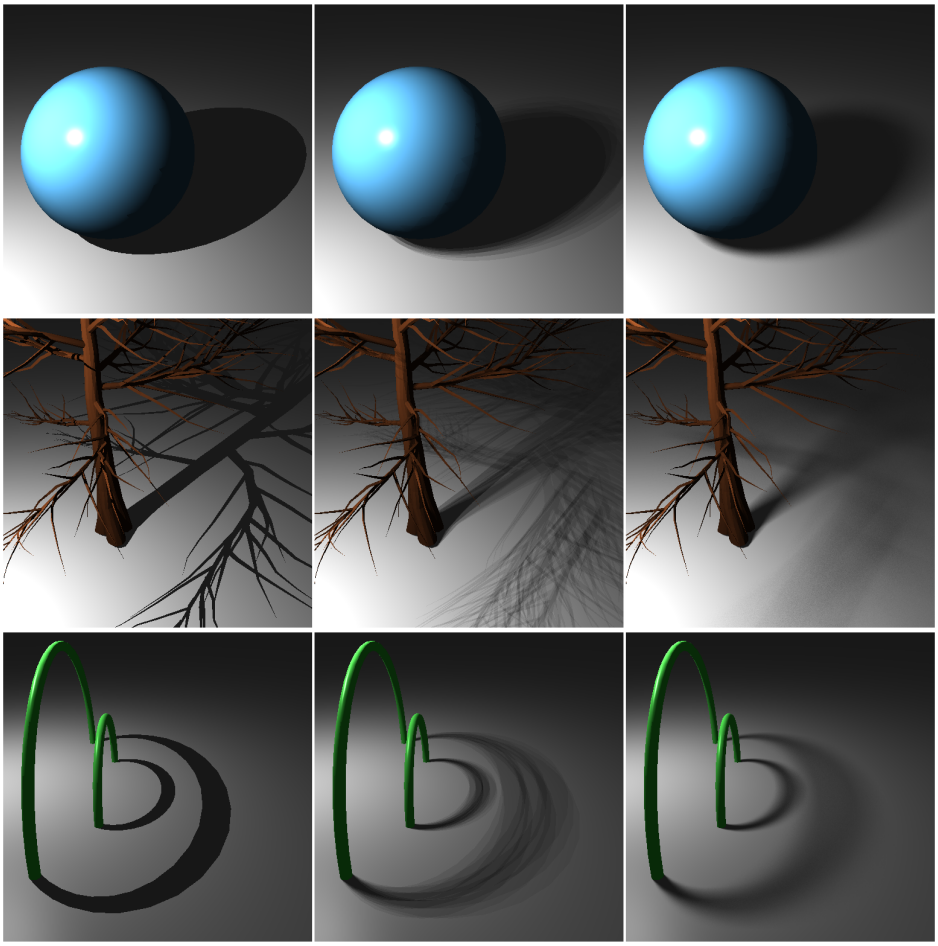


Figure 4.5: The first column shows perfectly sharp shadows since only one shadow ray is shot. The middle column shows banded shadows since several light sources have been used. The right column also contains several light sources, but their positions have stochastically been altered, thus making the transition between the shadow bands smooth.

Soft shadows have been implemented in the most naive way by increasing the number of lights sources in a light. The new light model, called *area light*, is composed internally by several point light sources aligned in a N $\times$ M grid. The axes of the grid are defined by two vectors, making it possible to scale, rotate and place the area light freely in 3D space (see Figure 4.6).

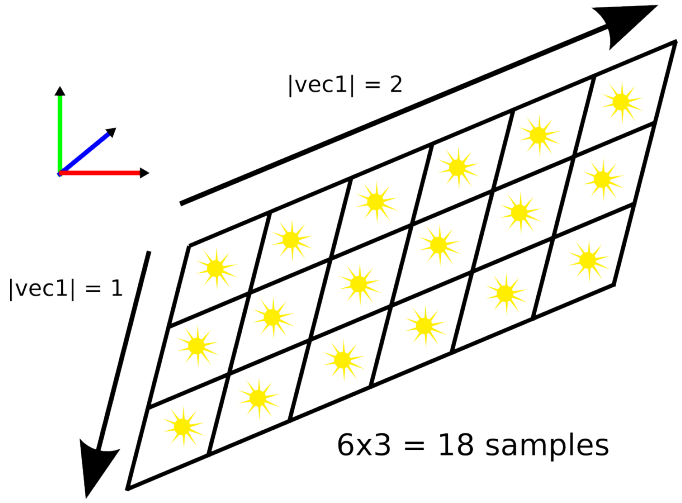


Figure 4.6: Area light implementation using an arbitrary sized grid of of point light sources. The grid can be freely located, scaled and rotated in 3D space.

With the area light implementation, the shadows from a light source are no longer limited to a binary value. However, the shadows are not yet soft, but rather, banded i.e. you can clearly see the internal light sources' own shadows (see Figure 4.5). In order to create the soft transition between the bands, thus making the shadow "soft", a stochastic method has been used for varying the internal light source positions' inside their cells for each shadow ray (see Figure 4.7). This will, with an increasing number of light samples, give the penumbra a smoother, more realistic look as seen in Figure 4.8.

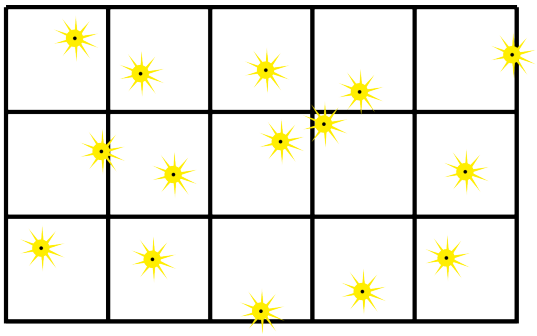


Figure 4.7: Randomly selects the internal lights positions inside their cells for each shadow ray.

Shadow cache

Rendering realistically looking soft shadows require a lot of light samples, therefore a lot of shadow rays are needed. A method to exploit the cache coherency has been implemented to minimize the number of intersection tests for the shadow rays. The main idea for each light in the scene is to store a reference to the shadow ray's latest intersected triangle. When rendering adjacent pixels it is likely that the next rendered pixel will be blocked by the same triangle. Therefore, instead of executing the intersection test with the whole triangle data tree as usual, a single intersection test is performed for the cached triangle.

If there is a hit, the shadow test is done, otherwise it will continue with the regular intersection test. If the pixel is blocked by a new triangle, it will be set as the current cached one.

Since our renderer is multi-threaded, care must be taken when dealing with shared memory. The triangle cache in each light could potentially be overwritten by two separate threads simultaneously. This has been solved by letting each light have a separate cache for each thread.

4.2.3 Results

In order to avoid grainy penumbras, the number of samples in the area light needs to be relatively large which in turn increases the amount of shadow rays shot. This will linearly increase the rendering time. See Figure 4.8 for a comparison between the penumbra quality and the number of light samples. Table 4.1 shows the rendering time for respective number of shadow ray samples. As can be seen in Figure 4.9, scenes where occluders are relatively few, the use of shadow caches does not improve the rendering time significantly. However, in scenes where occluders are frequent, rendering time can be greatly reduced.

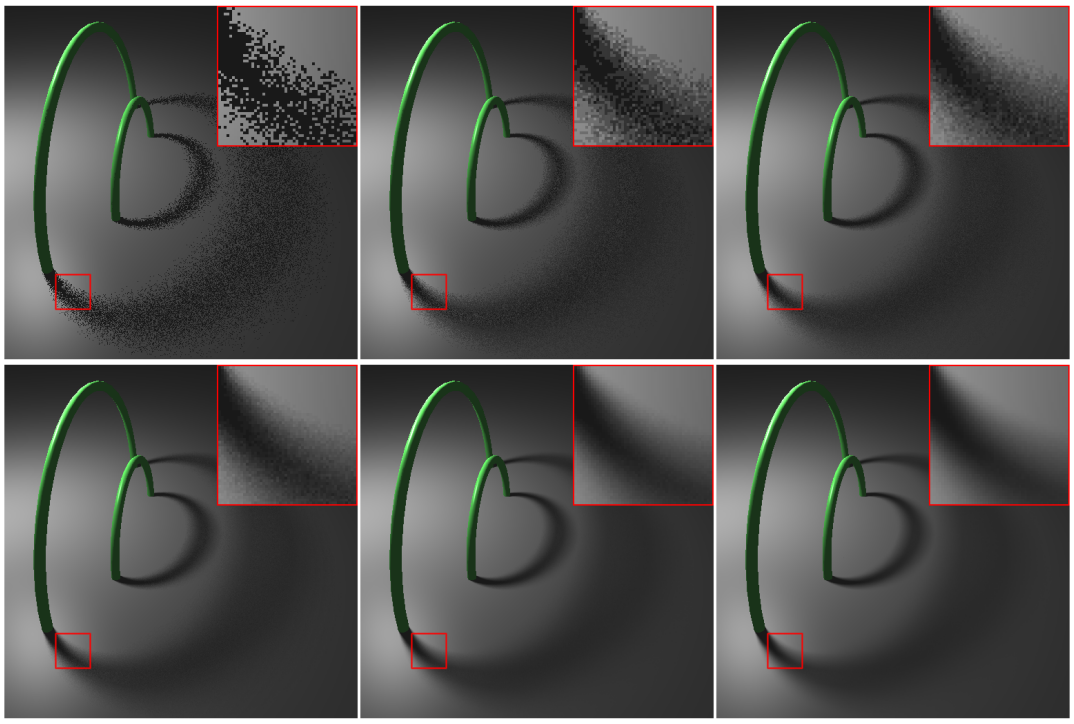


Figure 4.8: Rendered with an area light source with increasing number of samples. From top left to bottom right: 1, 4, 9, 25, 100, and 225 samples.

Table 4.1: Showing the rendering times with increasing shadow ray samples. The first five results correspond to the images in Figure 4.8.

Shadow ray samples	**1**	**4**	**9**	**25**	**100**	**255**	**900**
Render time (s)	0.24	0.59	1.16	2.99	11.0	24.1	94.7
Relative render time	1.00	2.46	4.83	12.5	45.8	100	395

The use of shadow caches improved the rendering performance for most of the scenes as can be observed in Figure 4.9.

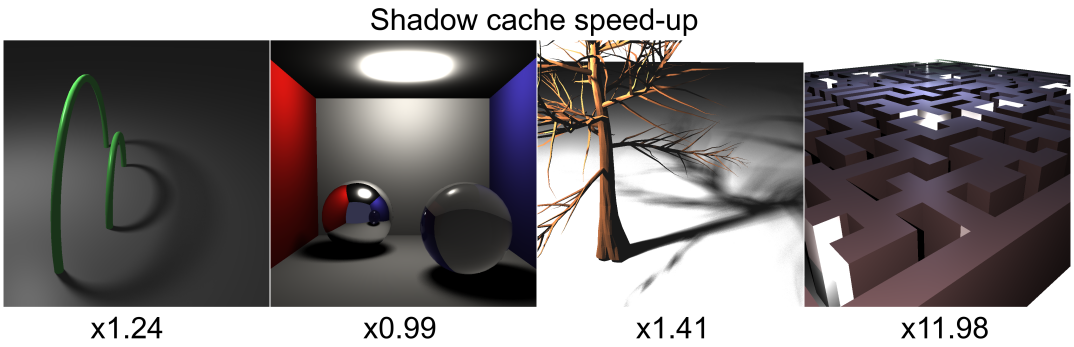


Figure 4.9: Speedup gained from the use of shadow caches. The rightmost image is a scene which typically favors the most from it, since adjacent points in the scene often share the same occluder.

4.2.4 Discussion

Since our soft shadows are based on a stochastic method, the shadows rendered in the same scene twice will slightly differ from each other. This is not a problem for individual rendered images, but when outputting several images in a sequence (e.g. for an animated movie) this will result in noticeable noise between the frames.

Our implementation of shadows does not take into account the use of transparent geometry. For instance, the shadows from a drinking glass are as dark as if the glass was completely opaque. This could have been solved by tracing the shadow rays from the surface to the light instead. However, this method would have to take transparent objects' refraction carefully into account, since the direction from the surface to the light source might not be straight. When using a path tracing algorithm however, this effect is simulated accurately.

Shadow caches were only implemented for the first recursion depth level (see Section 4.3). This has the consequence that speedups are not gained from shadow rays from reflected and transmitted rays. However, it has been discussed that shadow caches for these kind of rays increases the overhead greatly, thus negating the effect of shadow caches [Smits 2005]. As can be observed in the second image in Figure 4.9, this is what has happened. The scene did not favor enough from the shadow caches, making the use of shadow caches excessive.

There are several other speedup techniques that could have been implemented in order to reduce the rendering time but still maintain the smooth penumbra transition. One is the *Single Sample Soft Shadow*, a approximative stochastic method, introduced by Parker et al. [1998], which as the name implies only uses one sample to acquire soft shadows. Another one that also uses only one shadow ray is the *Soft Shadow Volumes for Ray Tracing*-implementation by Laine et al. [2005], which accurately creates physically-based soft shadows with the help of shadow volumes.

4.3 Reflection and refraction

A major functionality that separates a ray tracer from a regular ray caster is that in the latter, the rays stop at the first intersection with the mesh. In that way, the final rendered scene only depends on the actual intersected mesh, and does not take other surroundings into account. In this section, techniques to simulate accurate reflections and refractions are discussed. That is, the processes where light either bounces of the surfaces, such as a mirror, or travels through surfaces, such as glass. If the material is reflective or refractive, new rays are spawned from the intersection point and traced recursively.

A basic implementation of reflections and refractions only simulates surfaces that are perfectly smooth which results in perfect reflections and refractions without any deviations. This is not always the case; in reality surfaces often have microscopic bumps. This results in imperfect reflections and refractions of the rays that scatter the rays depending on the roughness of the surface. An implementation that simulates this is also presented below. Real world examples of this effect can be observed in frosted glass which glossily refracts the incident rays. Another example of glossy reflections is brushed steel.

In addition to this, the Fresnel effect, which is the observation that the amount of reflected light on a surface is angle-of-view dependant, is a key feature that further improves the image realism. The Fresnel effect can be observed for instance when looking at a shopping window: when looked at from narrow angles the window reflects light at a much higher rate than when looked straight at in front of it.

4.3.1 Previous work

The fundamental equations for reflective and refractive light directions are *The law of Reflection* and *Snell's law*. These are shown in Equation (4.6) and Equation (4.7) where $\vec{i}$ is the incoming direction, $\vec{r}$ is the outgoing direction, θ_1 and θ_2 are the angles measured from the normal, and η_1 and η_2 are the refractive indices of respective medium. See Figure 4.10.

$$\theta_i = \theta_r \tag{4.6}$$

$$\frac{sin(\theta_1)}{sin(\theta_2)} = \frac{\eta_2}{\eta_1} \tag{4.7}$$

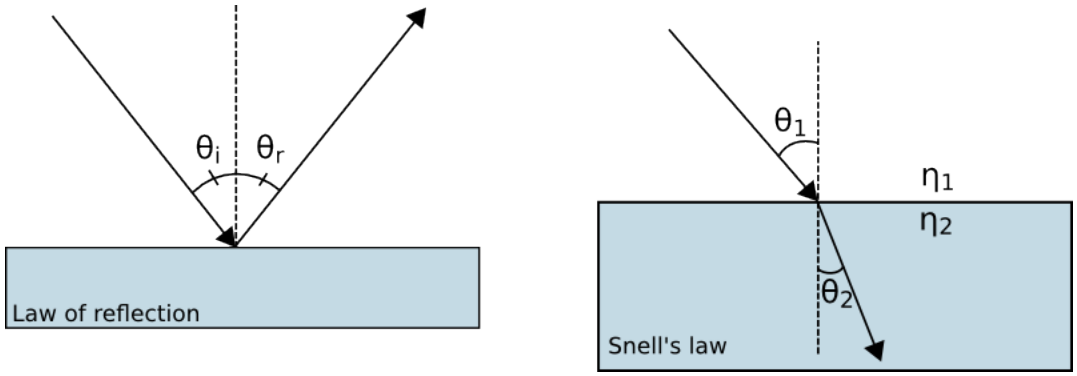


Figure 4.10: Left: Law of reflection, showing how an incident light ray is reflected at a surface. **Right:** Snell's law, showing how an incident light ray is refracted into another medium.

From these optic laws, two formulas optimized for vectorized computations are derived [Heckbert 1989]:

$$\begin{aligned} c_1 &= \vec{n} \cdot -\vec{i}, \\ \vec{refl} &= \vec{i} + \vec{n} \cdot c_1 \cdot 2, \end{aligned} \tag{4.8}$$

$$\begin{aligned} c_2 &= \sqrt{1 - \eta^2 \cdot (1 - c_1^2)}, \\ \vec{refr} &= \eta \cdot \vec{i} + (\eta \cdot c_1 + c_2) \cdot \vec{n}, \end{aligned} \tag{4.9}$$

where η is the relative index of refraction (η_1/η_2).

The following equation, stated by Blinn [1977], shows the Fresnel reflection function which also takes specular lighting into account:

$$F = \frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right), \tag{4.10}$$

where $c = (\vec{v} \cdot \vec{h})$ (where h is the halfway vector defined in Equation (4.5)) and $g = \sqrt{\eta^2 + c^2 - 1}$.

4.3.2 Method

Depending on whether the material properties' *reflectance* and *transmittance* values (both with a range of $[0,1]$) are larger than zero, new directions of the rays will be calculated using Equation (4.8) and (4.9). Furthermore, care must be taken if the expression beneath the square root operand in Equation (4.9) is negative; it will cause the physical phenomena known as *total internal reflection* [Heckbert 1989]. Therefore, the new ray's direction will be calculated using Equation (4.8) instead.

It is vital to distinguish between rays that are entering or leaving an object when dealing with refractions. The sign of the dot product of the intersection point normal and the incoming ray direction, as seen

in Equation (4.11), will give that information. If the sign is positive it means that the ray is entering. Otherwise, it means it is exiting the intersected object and the relative index of refraction has to be inverted.

$$sgn(n \cdot i) \tag{4.11}$$

When new reflective/refractive rays are spawned, precautions must be made to avoid self-intersections due to floating point rounding errors. A small value ε is introduced to translate the starting point of the ray away from the mesh in the normal direction. See Figure 4.11. Once again, Equation (4.11), will be utilized for determining if the ray enters or leaves. Equation (4.12) shows the new starting point for the ray, where addition is used for the reflection vectors and subtraction is used for the refraction vectors.

$$startpoint = startpoint \pm n \cdot sgn(n \cdot i) \cdot \varepsilon \tag{4.12}$$

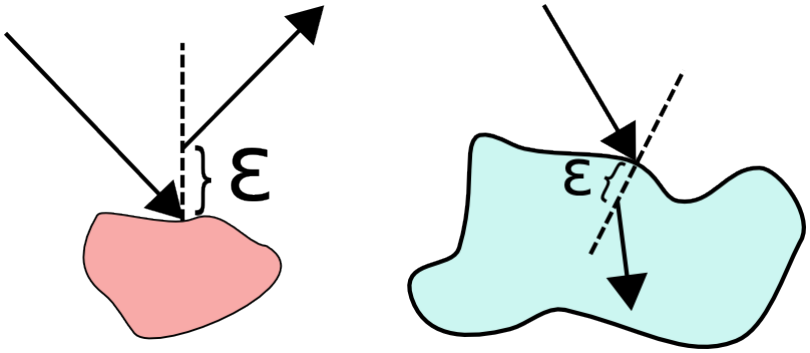


Figure 4.11: Showing the offset value ε for reflective and refractive surfaces.

The new rays are then traced recursively; the trace returns the reflected or refracted color, which in turn are mixed with the input color according to Equation (4.13).

$$\begin{aligned} color &\leftarrow color \cdot (1 - transmittance) + transmittance \cdot refrColor \\ color &\leftarrow color \cdot (1 - reflectance) + reflectance \cdot reflColor \end{aligned} \tag{4.13}$$

One method to achieve glossy reflections and refractions is to take several samples and average these. This is implemented by shooting a number of extra reflection and refraction rays for each ray that hits a reflective/refractive surface.

For a given outgoing ray, a new random ray is generated, which angle deviates no more than α degrees. This is done using the following algorithm:

The above algorithm assumes that the outgoing ray lies in the (0, 0, 1) direction. The generated ray is therefore then rotated back according to its outgoing ray's direction. The final color is then averaged for all of the taken samples.

The Fresnel effect has been implemented using a modified version of Equation (4.10). Instead of using $c = (\vec{v} \cdot \vec{h})$ which depends on the halfway vector (which in turn depends on the light source direction), it

Algorithm 6 Algorithm for calculating random vector in cone

$z \leftarrow rand(cos(\alpha), 1)$
$t \leftarrow rand(0, 2\pi)$
$w \leftarrow \sqrt{1 - z^2}$
$x \leftarrow w \cdot cos(t)$
$y \leftarrow w \cdot sin(t)$
$randVec \leftarrow vec3(x,y,z)$

has been defined as $c = (\vec{v} \cdot \vec{n})$ and thus depends on the normal. This has the consequence that Equation (4.10) now only gives the albedo value for the surface.

To stop the recursion, a combination of an adaptive depth and a max recursion depth has been used. Adaptive recursion depth has been implemented using a weighted value starting at 1.0. This value is attenuated for each recursion level with the reflectance/transmittance value. The recursion stops either when the weighted value becomes lower than the fixed attenuation threshold[8], or when the recursion level have reached its maximum depth.

4.3.3 Results

As can be seen in Figure 4.12, ELUMI is capable of rendering physically correct reflections and refractions. Figure 4.13 shows the Fresnel effect and Figures 4.14 and 4.15 shows how different recursion depths and attenuation values affect rendered images.

Figure 4.12: Showing two spheres on a checkerboard surface. The left sphere is fully reflective while the right one is completely transparent with the index of refraction set to 1.5 (glass). Notice how the Fresnel effect can be observed towards the contours of the refractive sphere in form of increased reflectiveness.

[8] typically values are the reciprocal of a power of 2, e.g. $1/2^8$, which in this case means that the threshold is equal to an 8-bit color's minimum difference.

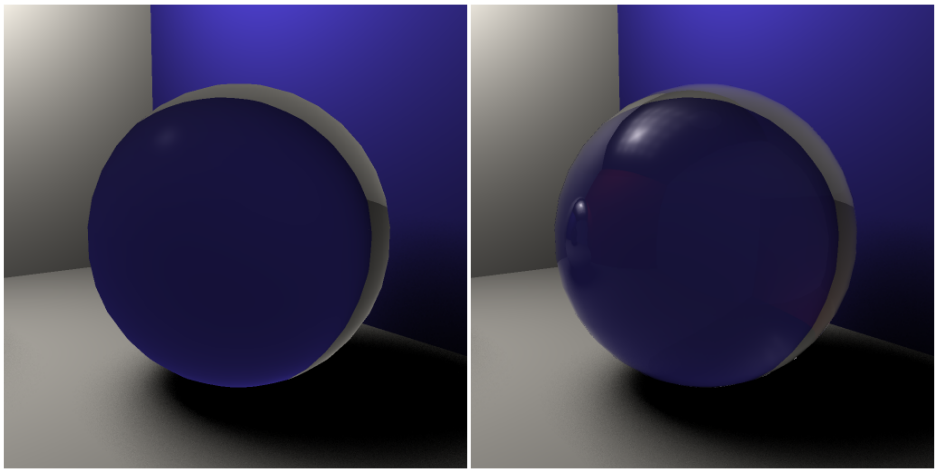


Figure 4.13: Displaying the difference when applying the fresnel equation on a glass sphere. Notice the sphere's reflections.

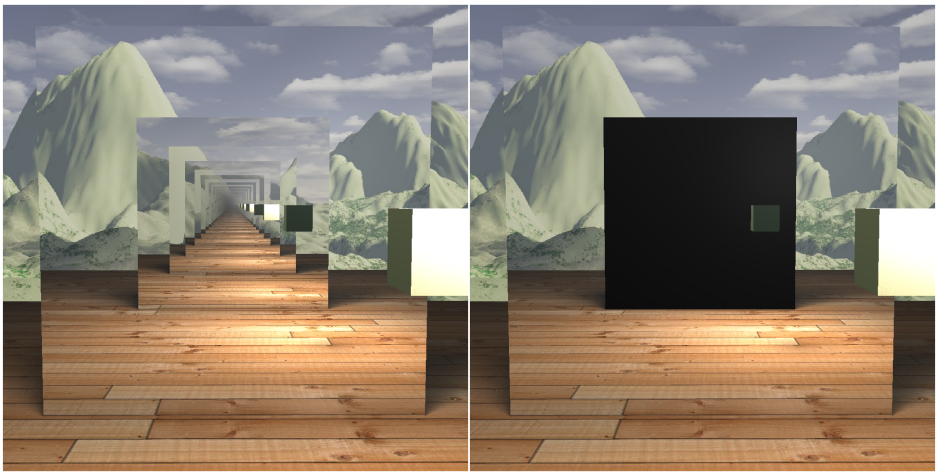


Figure 4.14: Showing a scene with two mirrors placed in front of each other. The left image has no recursion limit, whilst the right one stops at the first recursion level.

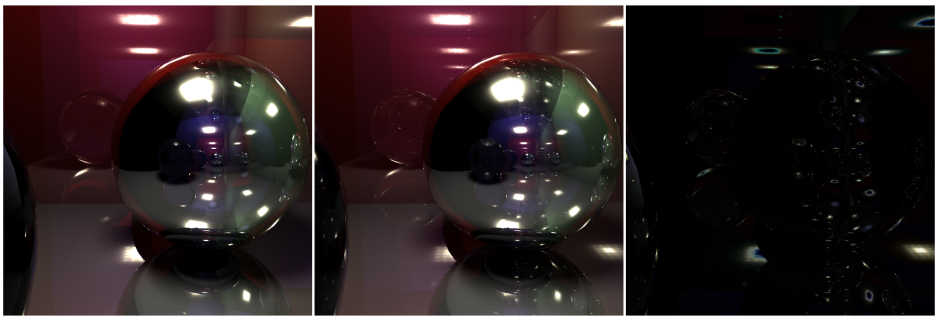


Figure 4.15: Showing a scene with only reflective and refractive surfaces. The left and middle images are rendered with a recursion attenuation threshold of 0.16 and 10^{-6} respectively. The left image was rendered 5.2 times faster than the middle. The right image shows the difference.

The final results with glossy reflections and refractions with increased spread to simulate the roughness of the surface can be shown in Figure 4.16. It can be noticed that this technique gives a realistic result which simulates more blur as the distance from the reflected/refracted surface increases.

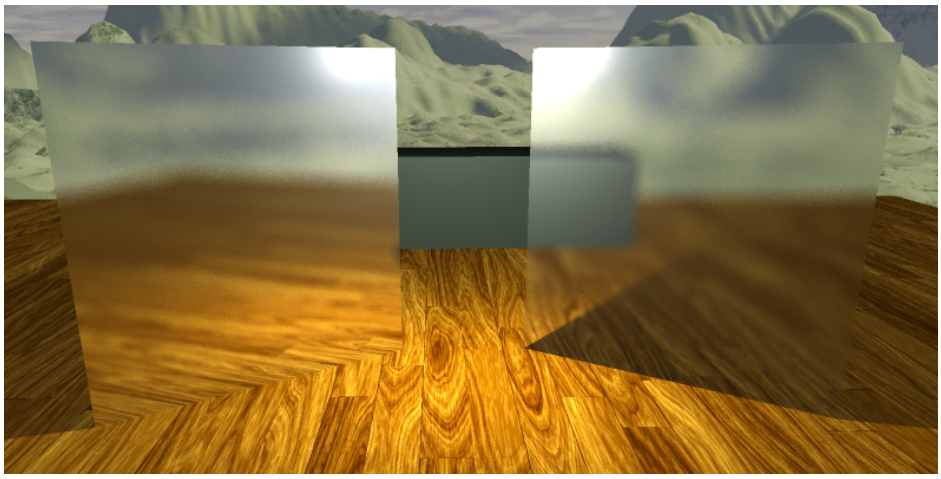


Figure 4.16: Two planes with glossy reflections (left) and glossy refractions (right). Notice how as the distance to the reflected and refracted surface increases, it gets more blurred.

One obvious downside of this algorithm is the noise and graininess on the reflected or refracted surface that is an effect of undersampling. By using a higher number of samples this can be overcome, with the obvious downside of increased render time (see Figure 4.17).

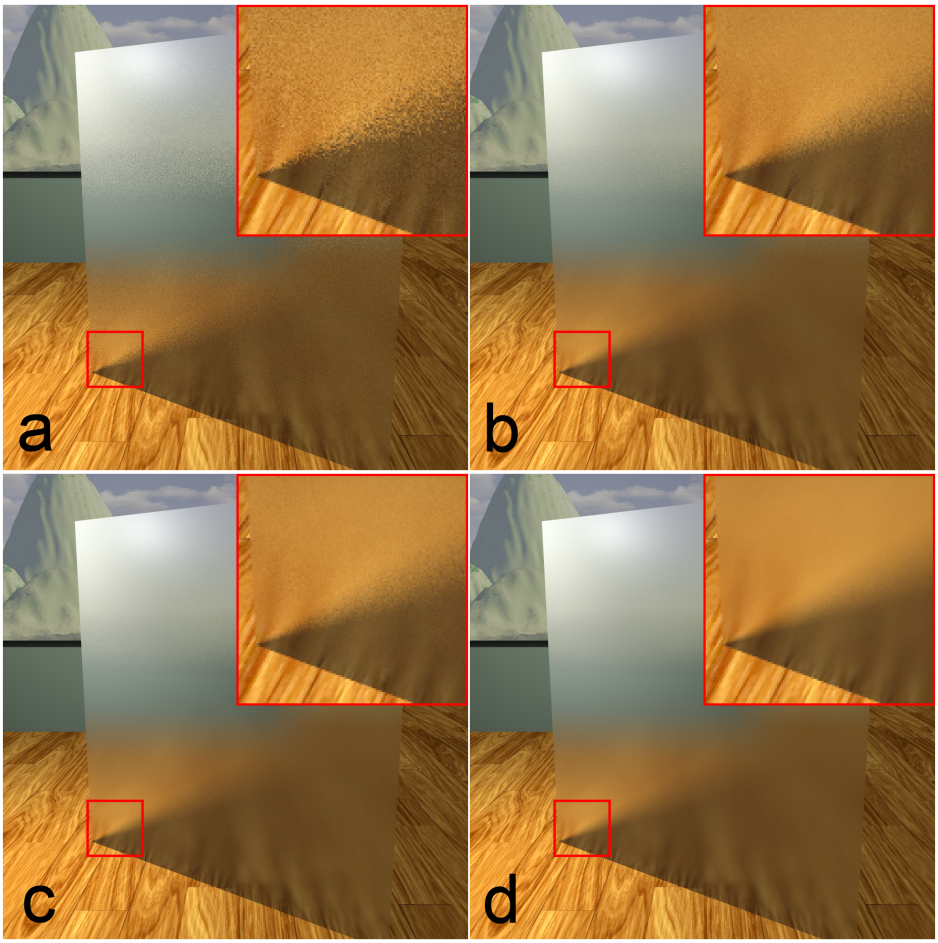


Figure 4.17: A glossy plane rendered with refraction spread equals to 0.1 and 4 super sampling. **(a)** refraction samples: 1, render time: 4.51s. **(b)** refraction samples: 5, render time: 9.48s. **(c)** refraction samples: 10, render time 15.75s. **(d)** refraction samples: 100, render time: 125.79s. Notice how the planes get gradually smoother with increasing number of samples.

4.3.4 Discussion

One way of improving the glossy reflections and refractions would be to introduce another way of doing super sampling. For example, stratified sampling (discussed in Section 2.3) is one strategy of improving rendering times. Quasi-Monte Carlo sampling is another possible solution [Ohbuchi and Aono 1996].

4.4 Environment mapping

Adding an environment which encloses the scene can drastically improve the realism in the rendered image; virtual objects can appear to be located inside a real environment. This section gives a basic introduction of what environment maps are. Then a specular reflection implementation with cube and spherical mapping are shown.

The basic idea is that a ray which misses all objects in the scene is given a color from the surrounding environment, instead of the default background color. A range of different approaches are available: everything from physically based dynamic atmospheric lighting models [Bruneton and Neyret 2008] to static textures in form of spheres or cubes [Blinn and Newell 1976].

4.4.1 Previous work

The main idea of cube mapping is to use six square texture images for each of the positive and negative x, y, and z-axes respectively (see Figure 4.18). The images are mapped at infinitely long distances in their respective directions.

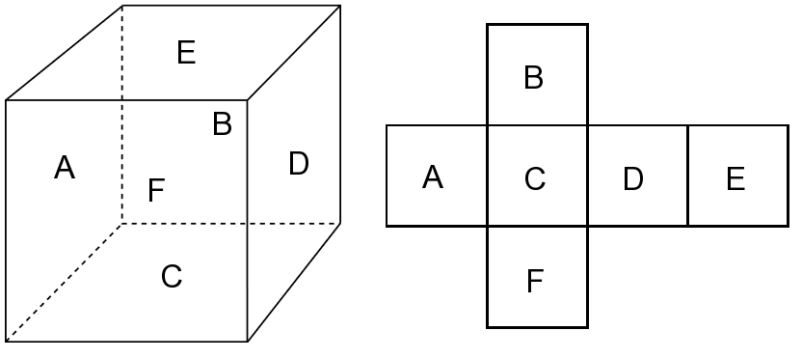


Figure 4.18: Showing a cube and its six faces unfolded. Each face of the cube is mapped to a respective square image texture.

To find the corresponding color of the environment, the following procedure is used [Akenine-Möller et al. 2008]. First off, the largest magnitude of a component in the ray direction vector will decide which cube face to be used. For example, the face in the negative y direction will be used in the vector $(4.0, -6.0, 0.5)$. Next, the coordinates of the face will be given by the other two components divided by the absolute value of the largest magnitude component. This will give coordinates values in the range $[-1,1]$. Finally they will need to be remapped to $[0,1]$ to fit the textures; this is easily achieved by adding one and dividing by two. Referring back to the example above, this becomes $((4.0/6.0+1)/2, (0.5/6.0+1)/2) \approx (0.83, 0.54)$. With these coordinates a regular texture lookup is performed and the environment map's texture's color is returned.

Another type of environment mapping is the spherical mapping where the surrounding environment is treated as if it were a sphere. Blinn and Newell [1976] proposed a method for mapping a texture to a spherical object by converting a reflected vector into polar spherical coordinates (ρ, θ) [Akenine-Möller et al. 2008]:

$$\rho = \arccos(-r_z), \tag{4.14}$$

$$\theta = \arctan(r_y, r_x). \tag{4.15}$$

A variation of this equation is proposed by Dunlop [2005]. It uses the arcsine function to compute the texture coordinates (u,v) from the normal:

$$u = \arcsin(N_x)/\pi + 0.5, \tag{4.16}$$
$$v = \arcsin(N_y)/\pi + 0.5. \tag{4.17}$$

One problem with this method is that more texels are mapped to the poles than the equator. Thus, singularities will occur at the poles [Akenine-Möller et al. 2008].

4.4.2 Method

Environment mapping has been implemented with cube mapping (according to the algorithm described above) since it is an efficient and easy method to implement [Greene 1986]. In order to align each face quad correctly with its neighbours, it is necessary to rotate and flip the image textures accordingly when being read from disk.

4.4.3 Results

As can be seen in Figure 4.19, environment mapping adds more realism to the rendered image, since it appears to be located in an actual photo realistic image.

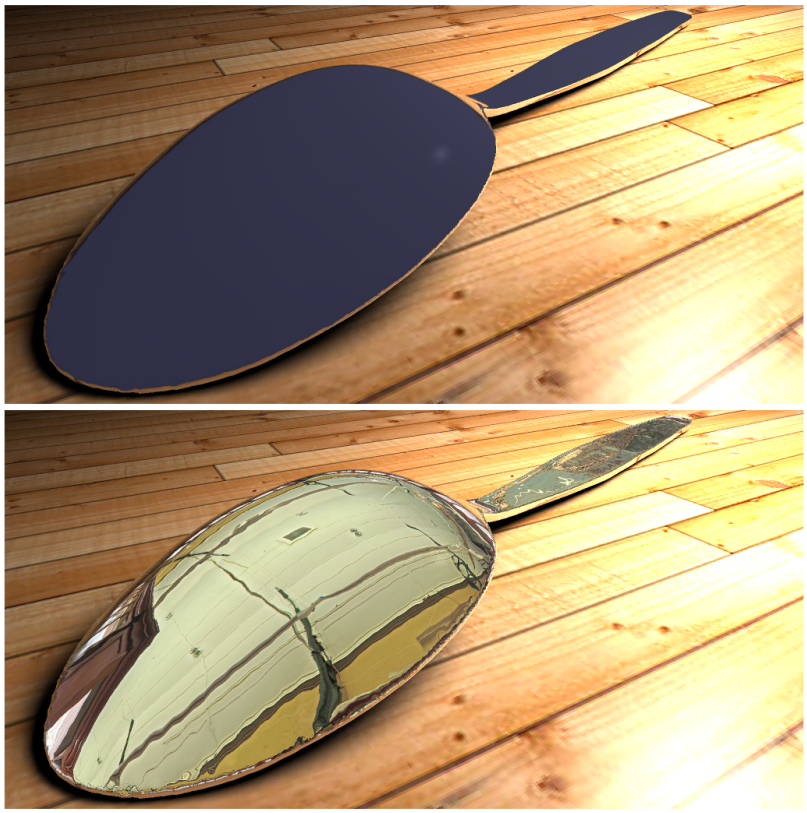


Figure 4.19: A comparison of a scene rendered with and without the aid of an environment map.

4.4.4 Discussion

Whilst this implementation of environment maps simulates specular reflections, there are more techniques which could have been implemented. For example, Ramamoorthi and Hanrahan [2001] presented an efficient representation for irradiance environment maps. Irradiance environment maps are typically high dynamic range textures that are used to enlighten diffuse objects in the scene. The environment is thus used as a continuous light source which varies with the HDR color.

4.5 Textures - increasing geometric detail

Creating a 3D scene with only triangles of different colors would not be feasible for most real-life replications. For example, a brick wall with its many different colors would be very difficult to render solely with triangles. For a larger scene with hundreds of different objects the complexity increases drastically. One way to decrease scene complexity is by using textures. They add significant detail to a model with little effort. Texturing can be thought of as applying wallpaper to a wall. For example, if an image of a brick wall is applied to a plane, then the texture would give the illusion of complex, geometric detail while the geometry remains planar. This illusion holds as long as the surface is viewed at a distance. Heckbert [1986] presented a number of other uses:

- Specular reflection
- Bump mapping
- Transparency
- Diffuse reflection
- Shadow maps

This section will discuss the different techniques of applying a two-dimensional image on a three-dimensional object. Section 4.6 will introduce even more uses of textures.

4.5.1 Previous work

The mapping of a two-dimensional image onto a three-dimensional object can be described using a generalized texturing pipeline [Akenine-Möller et al. 2008]. See Figure 4.20. The texture pipeline takes an object space location and converts it into a value that can be used to modify the object's surface. However, for this to work the object needs to be parameterized. This is done using a *projector function*. The projector function typically takes the three-dimensional location, *(x,y,z)*, of a surface and converts it into a set of two-dimensional coordinates, *(u,v)*, that can be used to access values in a texture. However, the output of the projector function is not only restricted to the two-dimensional space. Instead, it may be a new set of three-dimensional coordinates that can be used to lookup values in a three-dimensional texture. This can give the effect of an object being carved out of a material [Heckbert 1986]. Likewise, the input of the projector is not limited to positions, rather, almost any parameter may be used to generate the texture coordinates [Akenine-Möller et al. 2008]. This includes parameters such as: surface normals, the viewing direction, or even the temperature of the surface.

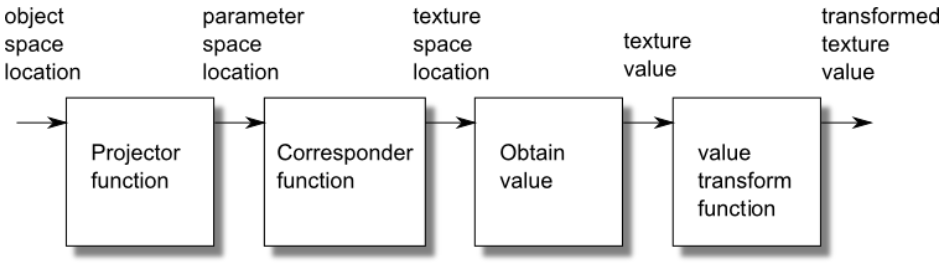


Figure 4.20: Showing the generalized texture pipeline which describes the steps of applying an image to an object. First, an object space location is parameterized using a projector function. Thereafter, the obtained values are converted into texture space coordinates using a corresponder function. These coordinates are used to retrieve values from textures. These values may then be transformed before they are used.

There exist a number of ways to map coordinates from three to two dimensions. One way is by using planar mapping, which projects the object space location onto a plane. This method may seem natural considering that the texture is already on a plane. However, since the mapping often is from three to two dimensions, one coordinate is lost and problems arise as to what plane should be used for the projection, where on the plane the texture should be placed, and how it should be scaled [Bier and Sloan 1986].

Another type of mapping is the spherical mapping. Spherical mapping can be implemented with either of the spherical mapping methods presented in Section 4.4. The normal is then converted to spherical coordinates. However, for certain objects, the normals are not good candidates for conversion into texture coordinates. For example, for a cube, where the normal is the same for all pixels on one side, all pixels on a side will receive the same color. A solution is then to use the intersection point instead of the normal [Dunlop 2005].

Cube mapping is another projector function which could be implemented with the cube mapping algorithm described in Section 4.4.

Another projector function is the cylindrical mapping where the u coordinate is calculated with spherical mapping, and the v coordinate is the distance along the cylinder's axis [Akenine-Möller et al. 2008].

The projector function returns coordinates in the interval $(-\infty,\infty)$. To lookup values in an image, these coordinates should be converted to the range $[0,1)$. This is done using a *corresponder function*. The corresponder function controls what happens to the image when the uv-coordinates fall outside the $[0,1)$ range. The corresponder function can be implemented in several different ways. For example, an image may be repeated along the surface or mirrored, i.e. the image is flipped after every repetition. The uv-coordinates could also be clamped to the range $[0,1)$. This results in the image edges being repeated. Also, a separate color could be used for values outside the $[0,1)$ range.

The coordinates received from the corresponder function can be used to lookup texture values. These texture values may then be transformed before they are used. One example of a transformation is the conversion from an unsigned range, $[0,1)$, to a signed range, $(-1,1)$ [Akenine-Möller et al. 2008]. Another transformation is to use the texture values to perform comparisons and transform the values into flags. Shadow maps are examples where the texture values are used for comparisons and then transformed into flag values, indicating if an object is in shadow or not (See Section 4.2).

Textures can be used in other ways besides coloring the objects. For example, they can be used as reflection maps, i.e. to tell what parts of an object should be reflective and by how much. The textures can be used in similar ways to set the object's specularity, opacity, and other material properties.

4.5.2 Method

The texturing process follows the texture pipeline described by Akenine-Möller et al. [2008]. Firstly, if a ray intersects with a triangle and that triangle should be textured, then the uv-coordinates are retrieved. If there are already texture coordinates provided with the triangle, then these coordinates will be used directly. Otherwise, the triangle needs to be parameterized using a projector function. The next step is to convert the parameterized value into texture coordinates using a corresponder function. The coordinates produced by the corresponder function are then used to lookup a value in a texture. A retrieved value is then used to change a triangle's material properties.

Parameterizing the triangles with projector functions

ELUMI provides four different projector functions. The first projector function is a planar mapping. Input to this function is a plane on which the location should be projected on. The second projector function treats the object as if it were lying inside a cube. It is based on the method for environment mapping, mentioned in Section 4.4. The third projector function is the spherical mapping. It is based on the method proposed by Dunlop [2005], also mentioned in Section 4.4. The last projector function is the cylindrical mapping. For this projector function, the u and v coordinates are calculated differently. Calculating the u coordinate is done by projecting the normal onto a plane. As with the spherical mapping, the chosen plane determines where singularities will occur. The coordinates of this two-dimensional projection are then treated as the coordinates (x,y) for a circle pivoting the plane normal. These coordinates are then converted to polar coordinates (r,θ). However, the radius is not needed, and as such, only θ is calculated:

$$\theta = \tan^{-1}\left(\frac{y}{x}\right). \tag{4.18}$$

The variable θ, is then used as the u coordinate. The v coordinate on the other hand is the coordinate that was left out after projecting the normal onto the plane.

Corresponder functions

Four different corresponder functions are provided: repeat, mirror, clamp, and border. Repeating an image is done by removing the integer parts from the u and v coordinates using the following equations:

$$u = u - \lfloor u \rfloor, \tag{4.19}$$
$$v = v - \lfloor v \rfloor. \tag{4.20}$$

The repeat function is used as a basis for the mirror function. The mirror function works by first calculating the signs for u and v respectively. The signs are calculated with the following equations:

$$sign_u = (-1)^{\lfloor u \rfloor}, \tag{4.21}$$
$$sign_v = (-1)^{\lfloor v \rfloor}. \tag{4.22}$$

The u and v coordinates are then multiplied with their respective signs before being passed on to the repeat function. The clamp function is implemented by clamping u and v to the range $[0,1]$. The border function returns a specified color when either of the u and v coordinates fall outside this range.

Retrieving the texture value

Retrieving a texture value is done by taking the texture coordinates and multiplying u and v with the width and height respectively. Depending on the texture type, the use of the retrieved value may differ. For example, if the texture is a diffuse map then the value is multiplied with the diffuse parameter used for the shading. Any value found in a texture map overrides the corresponding value in the material. For example, if a transparency value of one, meaning fully transparent, is found in the texture while the transparency value in the material is zero, then the value of texture value will be used and the pixel will be rendered transparent. In this implementation, only RGB images are used, with color values ranging from 0 to 255. These values are then remapped to the range $[0,1)$.

4.5.3 Results

The effect of using the different projector and corresponder functions in the ray tracer can be seen in Figures 4.21 and 4.22.

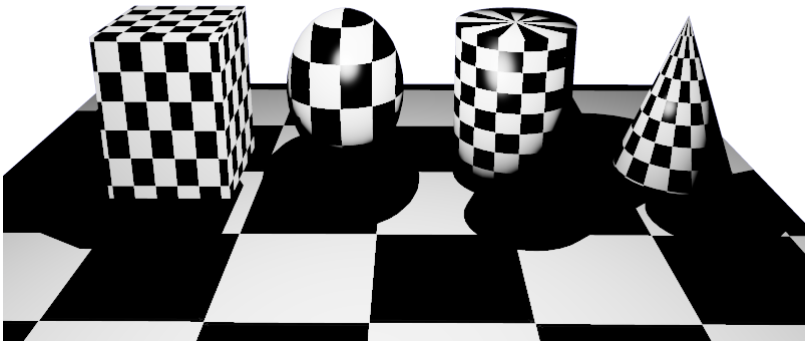


Figure 4.21: Application of different projector functions. Five different shapes with different projector functions applied to them. From left to right: planar mapping on a plane, cubical mapping on a cube, spherical on a sphere, and cylindrical mapping on a cylinder and a cone.

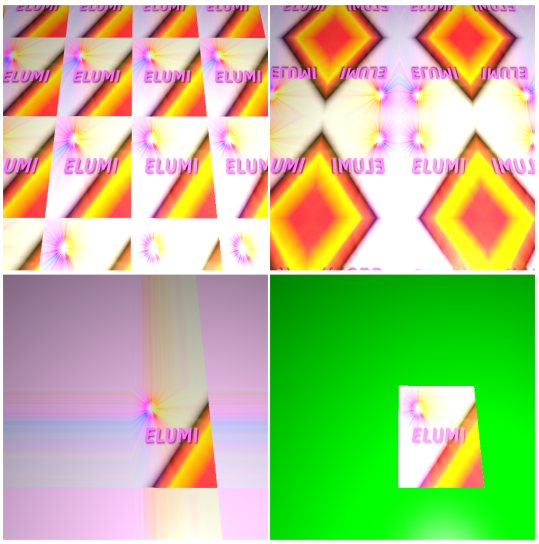


Figure 4.22: Application of different corresponder functions. Top row: repeat, mirror. Bottom row: clamp, and border.

4.5.4 Discussion

ELUMI provides the basic features of texture mapping. However, it does not consider aliasing. It is somewhat compensated for with supersampling (Section 2.3). Nevertheless, supersampling is rather slow, compared to using specific texture anti-aliasing techniques. A future improvement could be to implement such techniques, for example mip-maps.

Textures can be used for more than what has been implemented. For example, textures need not necessarily be static. Instead, a texture may be animated such that it changes for every rendered image. In this way, a waterfall could be modelled with an animated texture to make it look like falling water. A video could thus be generated by rendering several images of the waterfall and changing the texture for every frame.

4.6 Bump mapping - simulating displacements

The geometry of an object can be divided into three scales of detail; macro-, micro-, and meso-geometry [Akenine-Möller et al. 2008]. See Figure 4.23. Macro-geometry covers several pixels and are represented by geometric primitives. Micro-geometry covers features that are smaller than pixels, and are rendered using textures. In between is meso-geometry which covers features that are too small to render with geometric primitives, but are too large to render only with texture mapping. Such features include folds and wrinkles on a sheet and smaller stones on the ground. A collection of methods that simulate meso-geometry are commonly called *bump mapping* [Akenine-Möller et al. 2008].

Figure 4.23: Image depicting the differences between macro-, micro, and meso-geometry. Looking straight at the stone wall from a distance, it looks rather flat. Therefore, modelling the geometry with triangles, as seen from this view, seems natural. A texture is better suited for the color of the stone wall. When looking at a shallow angle close to the wall, more bumps and features are noticeable. This is the meso-geometry of the wall.

This section will discuss different implementations of bump mapping. These methods extend the use of textures by increasing the perception of geometric detail.

4.6.1 Previous work

Bump mapping can be implemented in different ways. One way is to perturb the surface normal for each pixel. By doing this, the actual geometry remains planar while still giving the illusion of a bumpy surface. This works because the perception of bumps on a surface is mainly due to the fact that they distort the surface normal [Blinn 1978].

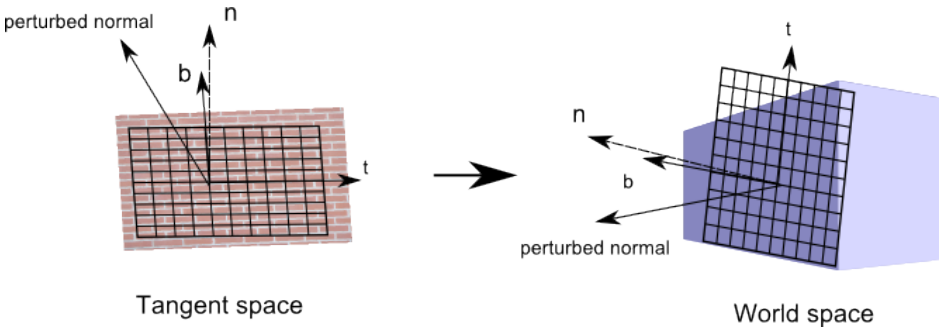


Figure 4.24: Tangent to world space transformation. The vectors b and t define how a bump map is applied to an object. These vectors are then used to transform a vector from tangent to world space, or vice versa.

Perturbing the surface normal can be done by using a normal map. A normal map stores a normal vector $\vec{n}$ at each texel. A normal retrieved from the normal map is then used as the perturbed normal. It is important that the the calculations are made with the same frame of reference [Akenine-Möller et al. 2008]. Therefore, a *tangent space basis* is stored for each vertex, describing how the normal map is mapped to the object. The tangent space is defined by a normal, a tangent, and a bitangent vector. Together these form the axes of the normal map. So, in order to use the perturbed normal it must first be

converted into world space[9]. This can be done with the following matrix, where $\vec{t}$, $\vec{b}$, and $\vec{n}$ stand for the tangent, bitangent, and the normal vector:

$$\begin{pmatrix} t_x & t_y & t_z & 0 \\ b_x & b_y & b_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.23}$$

Earlier bump mapping techniques used textures with only one or two components per texel; this was to reduce memory use. The drawback is that more computations are needed per pixel to derive the perturbed normal. For a texture with two components per texel, the normal is derived by calculating the cross product of these components multiplied with the tangent and bitangent vectors. On the other hand, for a texture with only one component per texel, each texel value is treated as a height and the normal is derived by calculating the differences between neighboring texels.

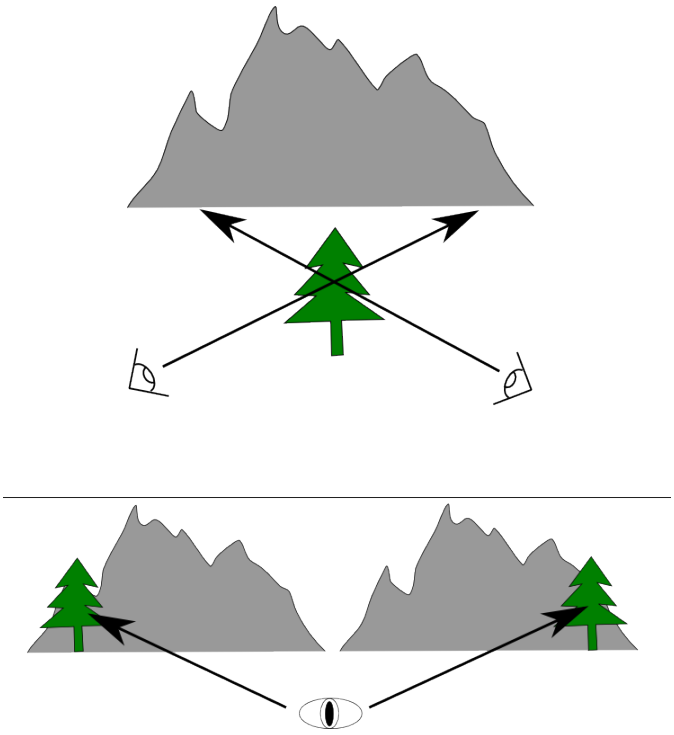


Figure 4.25: The parallax effect. As the viewer moves, the tree and the mountain appear to move relative to each other.

One weakness with normal maps is that while the perception of depth increases, areas on the surface lack the *parallax* effect demonstrated in Figure 4.25. Parallax is an effect that occurs when two objects at different positions seem to move relative to each other as the view position changes. *Parallax mapping*, first introduced by Kaneko et al. [2001] and later generalized by Welsh [2004], is a technique that

[9]Bump mapping calculations can also be done by converting the lights into tangent space and then perform the calculations.

addresses this issue. Parallax mapping uses height maps to store the bumps. Height values are retrieved from the height map and used to offset the texture coordinates in the viewing direction. The height value is scaled and biased before use, such that it better represents the physical properties of the texture

$$h_{sb} = s \cdot h + b. \tag{4.24}$$

The scale determines how far the height map stretches above the surface, and the bias gives the height where no shifting will occur. With this scaled and biased height value and a view vector, v, the new texture coordinates are calculated as:

$$t_n = t_0 + (h_{sb} \cdot v_{x,y}/v_z). \tag{4.25}$$

At shallow angles, this method will lead to very large texture coordinate shifts. The correlation between the height and the original texture coordinates will then be very small [Akenine-Möller et al. 2008]. Welsh [2004] proposes a solution to this by never shifting the texture coordinates farther than the retrieved height:

$$t_n = t_0 + h_{sb} \cdot v_{xy}. \tag{4.26}$$

The height value thus defines a radius, around the texture location, that determines the maximum texture coordinate shift.

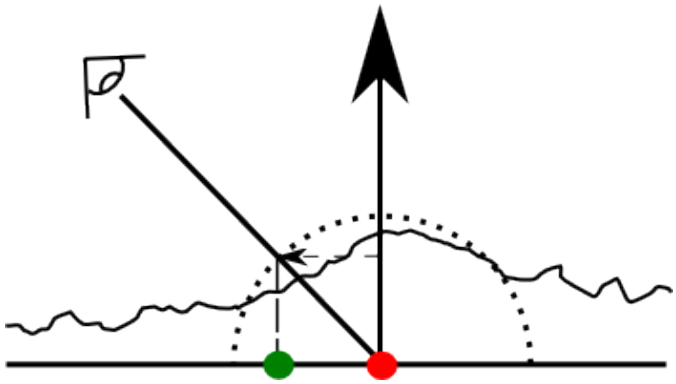


Figure 4.26: A texture coordinate is offset by projecting a view vector onto the texture plane. The height value retrieved from the height map along with the view vector determines the texture offset.

While parallax mapping simulates the parallax effect, it does not simulate areas of geometry occluding other areas. *Relief mapping*, or parallax occlusion mapping, are methods that tackle the occlusion problem. Akenine-Möller et al. [2008] present the generalized concept of relief mapping. The basic idea is to reversely ray trace the height map to find where the ray intersection occurs [Brawley and Tatarchuk 2004]. One problem with this algorithm is that if too few samples are used, then the ray might miss an intersection with the height map (See Figure 4.27). Tatarchuk [2006] summarize the algorithm as follows:

- Compute the tangent space viewing direction v and light direction l.

- Compute the parallax vector, p.
- Ray cast the view ray v along p. Sample the height map along p and compare with v to compute the texture offset.
- Ray cast the light ray and sample the height map for occlusions.

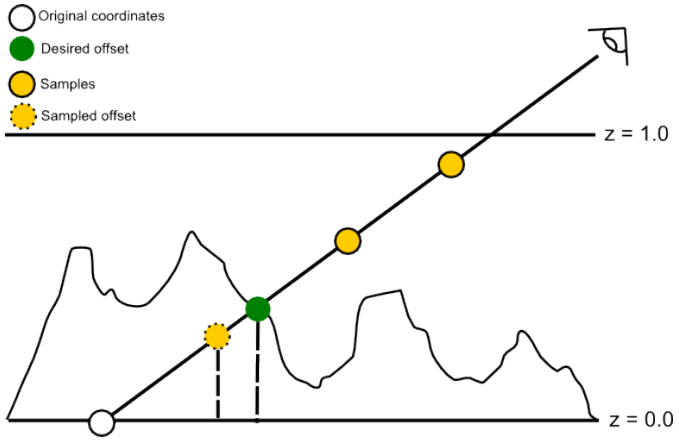


Figure 4.27: A view ray is sampled several times along the height map. The goal is to find the "real" ray/height map intersection (green). However, with too few samples the intersection may be missed.

This algorithm allows the height map to cast shadows on itself. This is done by ray tracing the height map with a shadow ray . If the light vector intersects with the height map, then the pixel is occluded. This can then be used to generate hard shadows by setting the light coefficient to zero whenever the pixel is occluded. However, soft shadows can also be generated by estimating how much the pixel is occluded.

4.6.2 Method

Three different forms of bump mapping have been implemented; normal, parallax mapping, and relief mapping. For each implementation, a texture is assumed to be lying on the $z = 0$ plane with its normal going in the positive y direction.

Parallax mapping was implemented using the equation proposed by Welsh [2004] described in Equation (4.26). First, the texture coordinates are used to find a texture value in the heightmap. Then, a vector going from the intersection point to the camera position is created. This vector is normalized and multiplied with the value found in the heightmap. The intersection point is then offset with this vector and used to lookup the new texture coordinates. These new texture coordinates are then used to find the diffuse color, bump normal, etc.

The relief mapping implementation is based on the algorithm presented by Tatarchuk [2006]. Pseudocode for the implemented algorithm is described in Algorithm 7. Similar to parallax mapping, a parallax vector is first created. An offset vector is also created by extracting the x and z coordinates from the parallax vector. It is normalized and multiplied with a depth constant which determines how "deep" the perceived occlusion effect will be. The y coordinate is similarly extracted from the parallax vector and used as the depth increment, i.e. how much the z value should be incremented for each sample. Then, for every

sample, a depth value is retrieved from the height map and compared with the current z value. If the z value is less than the value found in the height map then the highest intersecting sample is set to the current sample. After the loop has ended, the original texture coordinates is offset with the offset vector multiplied with the highest intersecting sample found.

Algorithm 7 Algorithm for relief mapping

$v \leftarrow faceForward(v,vec(0,1,0),normal)$
$offset \leftarrow depth \cdot normalize(view.x,view.z)/n_{samples}$
$dz \leftarrow v_y \cdot sqrt(3)$
$z_0 \leftarrow getDepth(tex\_coords)$
$z\_prev \leftarrow z_0$
for $i = 1 \rightarrow n_{samples}$ **do**
  $z \leftarrow getDepth(tex\_coords)$
  $steep\_bias \leftarrow 0$ ▷ Add bias if depth varies greatly
  if $abs(z - z\_prev) > bias$ **then**
    $steep\_bias \leftarrow z_0$
  end if
  $z\_prev \leftarrow z$
  if $dz \cdot i + steep\_bias < z$ **then**
    $lowest \leftarrow i$
  end if
end for
$tex\_coords \leftarrow tex\_coords + offset \cdot lowest$

Self-shadowing with relief mapping has also been implemented in the ray tracer. A slightly modified version of Algorithm 7 was used. First off, the parallax vector is created by taking a light's position and subtracting from it the intersection point. Then, instead of trying to find where this vector intersects with the height map, a light occlusion parameter, s, is instead estimated. The occlusion parameter is attenuated whenever the z value lies beneath the corresponding height map value. This is done using the following equation:

$$s = s - 1/n_{samples}. \tag{4.27}$$

Other equations could be used to estimate the light occlusion. For example, the different samples could be weighted such that samples closer to the viewer occlude the pixel less and vice versa.

4.6.3 Results

The effect of using relief mapping in the ray tracer can be seen in Figure 4.28. Figure 4.29 shows a comparison between the three different bump mapping methods.

Figure 4.28: Relief mapping applied on a texture with different depths. The different depths from left to right, top to bottom: 0.0, 0.3, 0.6, 1.2. Notice that as the depth increases, more and more artifacts are introduced into the image.

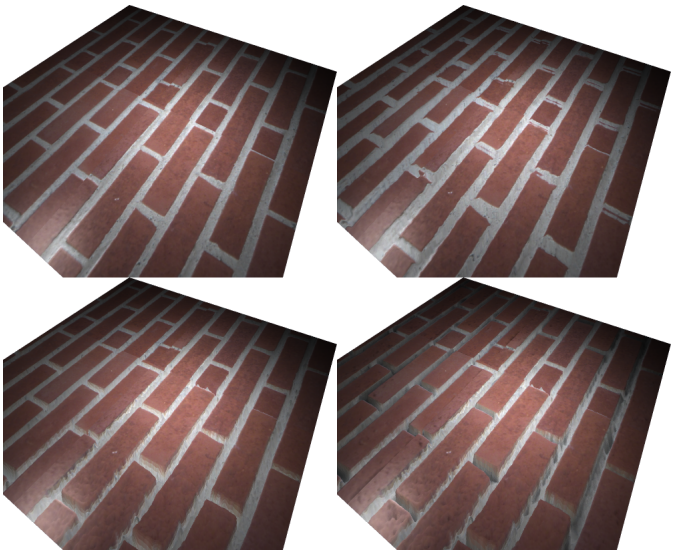


Figure 4.29: From left to right, top to bottom: Normal mapping, parallax mapping, relief mapping without shadows, relief mapping with shadows.

4.6.4 Discussion

A limitation with the bump mapping algorithms in this implementation is that they only work for planes with normals going in the positive y direction. The reason for this is because the OBJ format does not

support the tangent and bitangent vectors which are necessary to do the tangent space transform. A future improvement could thus be to change to another format, or calculate the tangent and bitangent with the help of the projector functions.

The relief mapping algorithm underperforms for height maps where the average difference between two texels are very large. A solution is to blur the height map before it is used, thus minimizing the differences.

There are even more ways of simulating bumps and wrinkles on a surface. Cook [1984] introduced displacement mapping, a technique where the actual locations are perturbed. Musgrave [1988] presents a displacement technique by ray tracing and triangulating a grid which can be used to simulate fractal mountains.

4.7 Conclusions

This section describes different ways of simulating most of the basic phenomena found in real-life. Methods for simulating diffuse shading as well as shadows, reflections, and refractions are described with details on the implementations. These are all different techniques of approximating the rendering equation.

Furthermore, this section also describes different techniques of adding details to geometry by using images. These techniques require little extra effort but add a significant amount of detail.

5

GPU acceleration and multi scattered light

The lighting model described up until now only considers direct illumination. There are more aspects to light than direct illumination, such as the set of effects commonly called global illumination. Another set of effects occurs when light interacts with the media it traverses. This section will present two techniques that simulate these effects: photon mapping and participating media, which are not part of the ray tracing algorithm.

This section also describes two techniques on how to accelerate certain parts of the ray tracing algorithm with a GPU: deferred rendering of the primary rays and photon gathering implemented as image space scattering.

5.1 Accelerating the primary rays

Most of the ray tracing algorithm is inherently parallel, and graphics processors are very fast when executing parallel programs. This section will describe a technique that exploits these facts to make ELUMI up to more than twice as fast.

5.1.1 Previous work

One observation about rasterization is that it is equivalent to shooting a grid of rays from a pinhole camera [Borman 2003]. This is the exact same problem as tracing the primary rays for every pixel (see Section 2.3).

Deferred rendering is a technique that uses rasterization to save the geometry data for the closest point for each pixel into buffers. Then, the lighting is calculated in a separate pass. Using deferred rendering, the lighting will only be calculated once for every pixel. This gives a considerable speedup if a complex lighting model is used [Saito and Takahashi 1990].

5.1.2 Method

Since the rays are generated exactly the same as for rasterization, accelerating the primary rays can be used almost interchangeably, while gaining some performance. This is done by using the first part of the deferred rendering technique. Four buffers are saved: depth, normal, texture coordinate, and material buffer. To get the intersection points of the ray for each pixel Equation (5.1) can be used.

$$position_{world}(x,y) = (M_{viewport} * M_{projection} * M_{view})^{-1} \begin{bmatrix} x+0.5 \\ y+0.5 \\ depthmap[x,y] \end{bmatrix} \tag{5.1}$$

5.1.3 Results

As can be seen in Figure 5.1 the difference between the images is only noticeable at a couple of pixels (the difference is at most $2^{-}8$ at the bottom half), and the rasterized primary rays are considerably faster. To render the same scene using ray casting, two rays need to be shot: one to determine geometry and one shadow ray to determine if the object is in shadow. Using rasterization, the former ray can be avoided. Thus, half as many rays are shot in this simple scene. In a more complex scene the speedup will be less, but rasterization is typically always faster.

Figure 5.1: Ray casted scene with one shadow ray (one point light). From left to right: Ray casting 87 s, rasterization of primary rays 41 s, and the absolute difference scaled by a factor of 100.

5.1.4 Discussion

This technique does make the ray tracer faster. However, it becomes slightly more complex and code duplications are inevitable. Furthermore, it can only optimize the rays in the first pass, which may not be a large amount compared to the total number of rays. In a scene with hundreds of lights and several reflective, refractive, and possible glossy surfaces, there may be more than a thousand rays traced per pixel and the speedup is then only one part in a thousand. The time may have been better spent optimizing other parts of the ray tracer instead. On the other hand, in a very simple scene (like the one used in Section 7), the speedup is considerable.

Another problem that can be solved in a similar way is the tracing of the shadow rays. If rasterizing from the light, instead of from the camera, *shadow maps* are acquired (for a brief description of shadow maps, see Section 4.2). Implementing this feature may have generated a greater speed up, but may also have

generated more inaccuracies since shadow maps are inherently less accurate than shadow rays [Williams 1978].

5.2 Photon mapping - a global illumination solution

Global illumination is a set of effects of light not covered by the direct illumination described in previous sections. The two most important effects of global illumination are diffuse interreflections and caustics. Caustics occur when light is focused by a lens causing a bright spot to appear. Diffuse interreflection is an effect that occurs when light bounces of a diffuse surface and then illuminates another surface. An example is described in Figure 5.2.

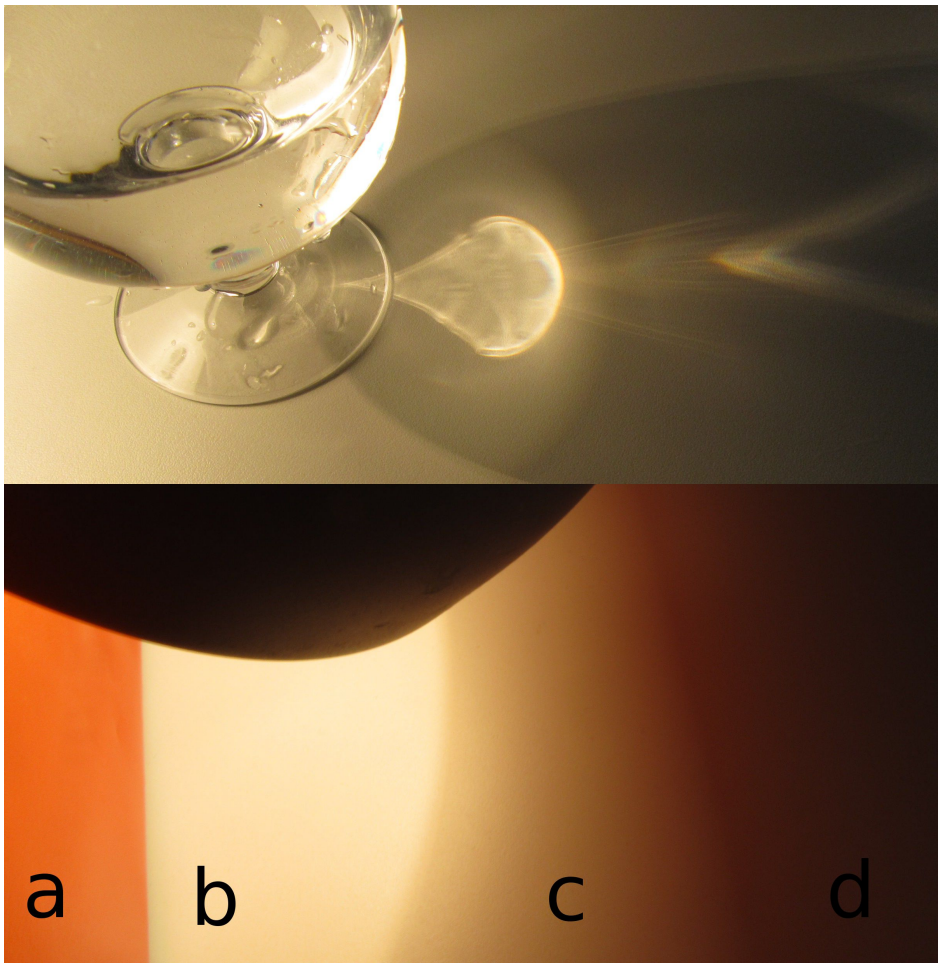


Figure 5.2: Images showing different aspects of global illumination. **Top**: Showing caustics, i.e. focused light, in the shadow of the glass. **Bottom**: A light with a light shade shining on a red screen and a table. **(a)**: A red screen illuminated by the light. **(b)**: Direct illumination of the table. **(c)**: Indirect illumination, the light bounces off the light shade. **(d)**: The shadow is red because it is illuminated by light bouncing off the red screen to the left.

Path tracing and Photon mapping are two algorithms that implement these effects. Path tracing has the benefit of being trivial to implement[10]. Photon mapping, on the other hand, has the benefit of generally requiring fewer rays traced than path tracing [Jensen 1996]. Radiosity is another algorithm in this category, but it does not simulate caustics [Goral et al. 1984].

[10] [Beason 2008] Did an implementation in 99 lines of C code.

5.2.1 Previous work

The photon mapping algorithm, first described by Jensen [1996], modifies the ray tracing algorithm at two places:

Photon tracing: Before tracing, photons are shot randomly from light sources and bounced around the scene according to each material's BRDF, similar to path tracing. At each bounce, the following data is stored in a photon map:

- $\vec{x}$: Position of bounce
- Φ_i: Incident luminance
- $\vec{\omega}_i$: Incident angle
- $\vec{n}_p$: Normal at bounce point

If another lighting model is used for the direct lighting (for example the model described in Section 4) then the first bounce of the photons should not be stored [McGuire and Luebke 2009]. This is because they represent the same data and should not be duplicated for reasons of precision and efficiency.

Photon gathering: In the shade function at point $\vec{s}$, photons are gathered to get the outgoing luminance L_p in direction $\vec{\omega}_o$. Either the contribution from the g closest photons or all photons within a fixed radius r are summed. The contribution of each photon is given according to Equation (5.2) (where f is the BRDF function) [McGuire and Luebke 2009]. This sum is used as the ambient light factor in shade.

$$L_p = f(\vec{s}, \vec{\omega}_i, \vec{\omega}_o) * \Phi_i * \max(0, \vec{\omega}_i \cdot \vec{n}) * \kappa(\vec{x} - \vec{s}) \tag{5.2}$$

The filter kernel κ affects the smoothness of the final image. Jensen [1996] used $\kappa(\vec{d}) = 1/(\pi r^2)$ as a kernel. McGuire and Luebke [2009] used a kernel of greater complexity that scaled by the normal of the vector as in Equation (5.3) (where r_z is the scaled radius) and used a 3D gaussian function as in Equation (5.4).

$$t = \frac{\left|\vec{d}\right|}{r}\left(1 - \left|\frac{\vec{d}}{\left|\vec{d}\right|} \cdot \vec{n}_p\right|\right)\frac{r - r_z}{r_z} \tag{5.3}$$

$$\kappa\left(\vec{d}, \vec{n}_p\right) = \frac{1}{\pi r^2} e^{\left(\frac{-t^2}{2\sigma^2}\right)} \tag{5.4}$$

5.2.2 Method

The tracing was done by giving each light n_l photons out of n photons total according to Equation (5.6) and giving each of those Φ_l power according to Equation (5.7). Jensen [1996] used a method equivalent to Equation (5.8). By refactoring $\frac{1}{n}$ it is not necessary to know the total number of photons when building the photon map. This factor is instead carried out when gathering. This enables shooting photons progressively. The power of a photon is the sum given by Equation (5.5).

$$P_i = L_{red} + L_{green} + L_{blue} \tag{5.5}$$

$$n_l = n\frac{P_l}{\sum_i P_i} \tag{5.6}$$

$$\Phi_{l\lambda} = \frac{\sum_i P_i}{P_l} L_\lambda \tag{5.7}$$

$$\Phi_{l\lambda} = \frac{1}{n}\frac{\sum_i P_i}{P_l} L_\lambda \tag{5.8}$$

The direction of the bounce is chosen by russian roulette depending on the properties of the material at the intersection. This works the same as the russian roulette in path tracing. A 10% chance of absorption was also factored into the roulette. Algorithm 8. gives a clear view of how the roulette was implemented.

Algorithm 8 The roulette used in photon mapping tracing

function BOUNCE(p)
  $P(transmittance) \leftarrow transmittance_m$
  $P(reflectance) \leftarrow (1 - P(transmittance)) * reflectance_m$
  $P(absorbtion) \leftarrow (1 - P(transmittance) - P(reflectance)) * 0.1$
  $P(diffuse) \leftarrow 1 - P(transmittance) - P(reflectance) - P(absorbtion)$
  $r \leftarrow random(0,1)$
  if $r < P(transmittance) \vee \vec{\omega}_i \cdot \vec{n} > 0$ **then**
    $\vec{\omega}_o \leftarrow refract(\vec{\omega}_i, \vec{n}, \eta)$
  else if $r < P(transmittance) + P(reflectance)$ **then**
    $\vec{\omega}_o \leftarrow reflect(\vec{\omega}_i, \vec{n})$
  else if $r < P(transmittance) + P(reflectance) + P(diffuse)$ **then**
    $\vec{\omega}_o \leftarrow$ *random direction in hemisphere of* $\vec{n}$
  else
    stop *tracing ray*
  end if
end function

To store the photons, a hashed grid was used, instead of using a K-D-tree as suggested by Jensen [1996]. The hash is implemented as an array of lists, and these lists are called buckets. Equation (5.9) was used to get the bucket index where the photon should be placed or retrieved. s was chosen to be equal to $2r$ making `gather` only requiring to look at 8 buckets. p_1, p_2, p_3 are arbitrarily big primes and $n_{buckets}$ is the number of available buckets.

$$\texttt{hash}(x,y,z) = \left(p_1 \left\lfloor \frac{x}{s} \right\rfloor + p_2 \left\lfloor \frac{y}{s} \right\rfloor + p_3 \left\lfloor \frac{z}{s} \right\rfloor\right) \mod n_{buckets} \tag{5.9}$$

The procedure of storing and gathering the photons can be seen in Algorithm 9.

The method described above with the scaled gaussian kernel was used to get the luminance contribution of each photon. This was then scaled by $\frac{1}{n}$ as described above to enable progressive photon tracing.

Algorithm 9 The main parts of the photon mapping algorithm

function STORE(x,y,z,p)
  $buckets\,[hash\,(x,y,z)] \leftarrow buckets\,[hash\,(x,y,z)] \cup p$
end function

function GATHER(p,r) :: $photon\,[]$
  $X \leftarrow [x_p - r, x_p - r + s, x_p - r + 2s, \cdots, x_p + r + s]$
  $Y \leftarrow [y_p - r, y_p - r + s, y_p - r + 2s, \cdots, y_p + r + s]$
  $Z \leftarrow [z_p - r, z_p - r + s, z_p - r + 2s, \cdots, z_p + r + s]$
  $H \leftarrow \{hash\,(x,y,z) : x \in X, y \in Y, z \in Z\}$
  return $\bigcup_{h \in H} \{f : f \in buckets\,[h]\,, \left|\vec{p} - \vec{p}_f\right| < r\}$
end function

5.2.3 Results

Figure 5.3 shows caustics and diffuse interreflection, examples of GI. Figure 5.4 shows a comparison of the kernel suggested by Jensen [1996] and the gaussian filter kernel. As can be seen, the gaussian kernel gives a smoother output, at the cost of rendering speed. Using the hash map described above instead of a trivial array data structure, a speedup of almost 4 times was achieved (1.12 s vs 4.44 s) in the same scene as above.

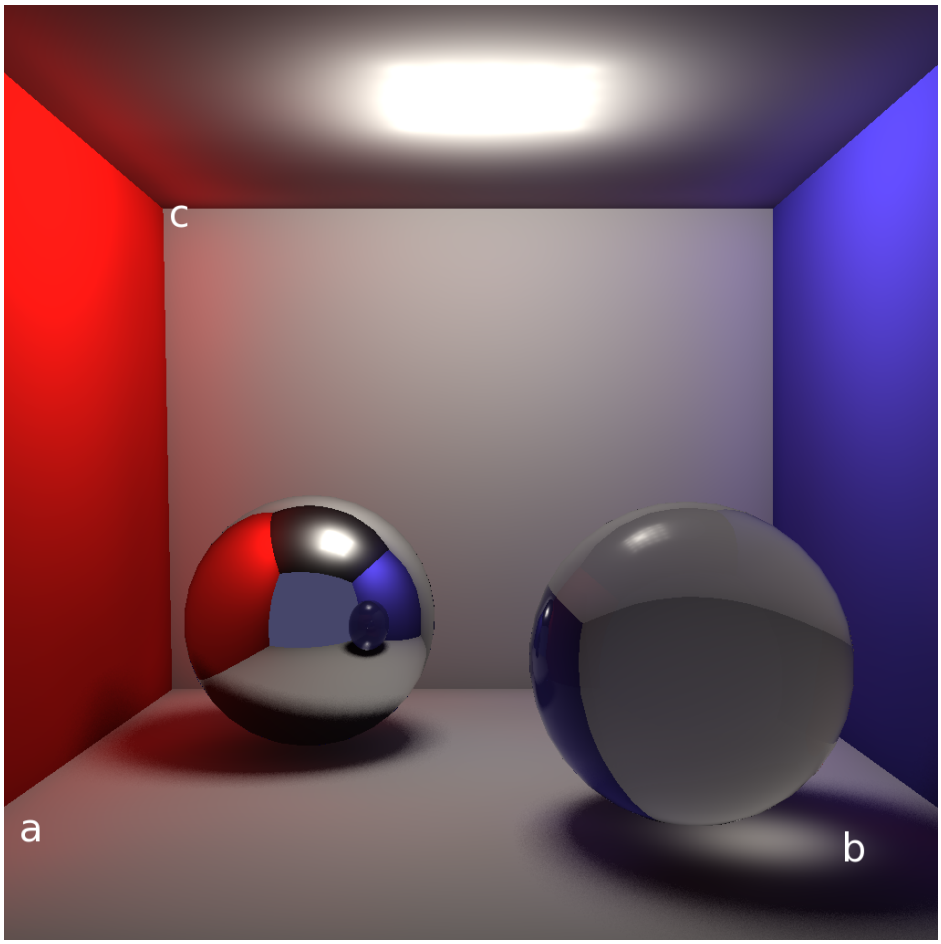


Figure 5.3: Different renderings of a 256x256 scene with 64k photons. **(a)** Diffuse interreflection (color bleeding). **(b)** Refractive caustics. **(c)** Contact shadows (ambient occlusion).

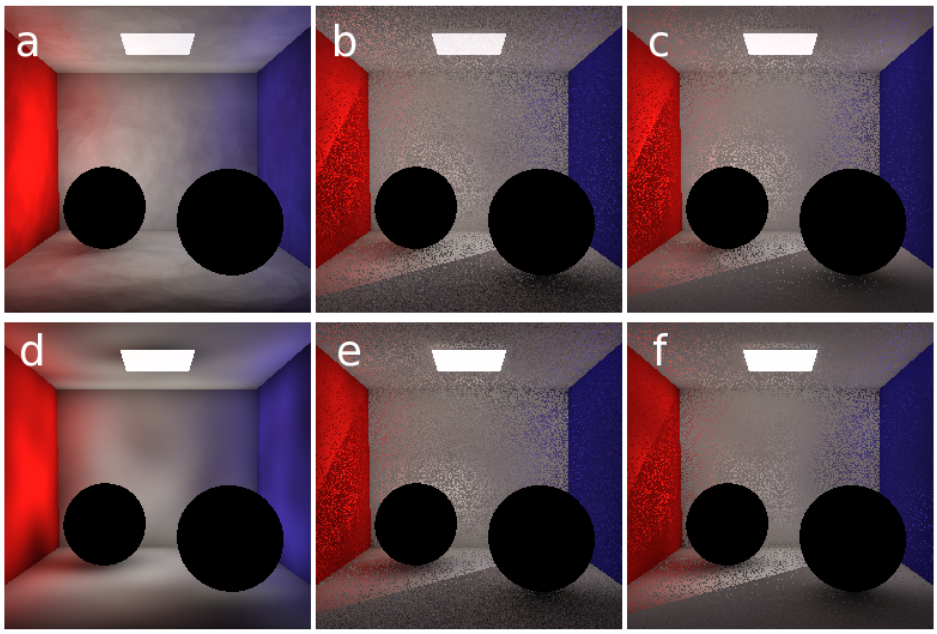


Figure 5.4: Different renderings of a 256x256 scene with 64k photons. **(a)** Jensens kernel, direct. **(b)** Jensens kernel, 32 final gather samples. **(c)** Jensens kernel, 1024 final gather samples. **(d)** Gauss kernel, direct. **(e)** Gauss kernel, 32 final gather samples. **(f)** Gauss kernel, 1024 final gather samples.

Table 5.1: Timings of Figure 5.4 in seconds.

Kernel	Direct	32 Final gather samples	512 Final gather samples
Jensen	6	165	2631
Gauss	10	294	4852

5.2.4 Discussion

The hash map described above was very simple to implement. If a K-D tree was used instead as recommended by Jensen [1996] the gathering may have been faster.

The slowdown experienced with the gaussian kernel, as seen in Table 5.1, may decrease if the kernel is optimized.

The caustics generated are not of as high frequency compared to what Jensen [1996] acheived. In Figure 5.2 spectral caustics are visible. This is an effect not normally done with photon mapping, and our renderer does not generate this effect.

5.3 Photon gathering on the GPU

A part that takes a lot of time in a Photon mapper is the gathering, since it has to be done for every bounce. Accelerating this part on the GPU can thus yield a great speedup.

5.3.1 Previous work

McGuire and Luebke [2009] used the same principle described in Section 5.1 to accelerate the photon shooting algorithm. Shooting rays from a spotlight is the same as shooting rays from a pinhole camera, which in turn is the same as rasterizing the scene from the lights view (similar to a shadow map) [Borman 2003].

McGuire and Luebke [2009] also changed the Photon gathering algorithm to a scattering algorithm in image space. McGuire and Luebke [2009] did this by rasterizing the spherical (or elliptical if scaling by the normal) volumes that each photon can contribute to in image space, calculating the contribution at each rasterized point and then additively blending the results together (see Figure 5.5). This can be done in a single pass in OpenGL using instance drawing to achieve real-time rendering [McGuire and Luebke 2009].

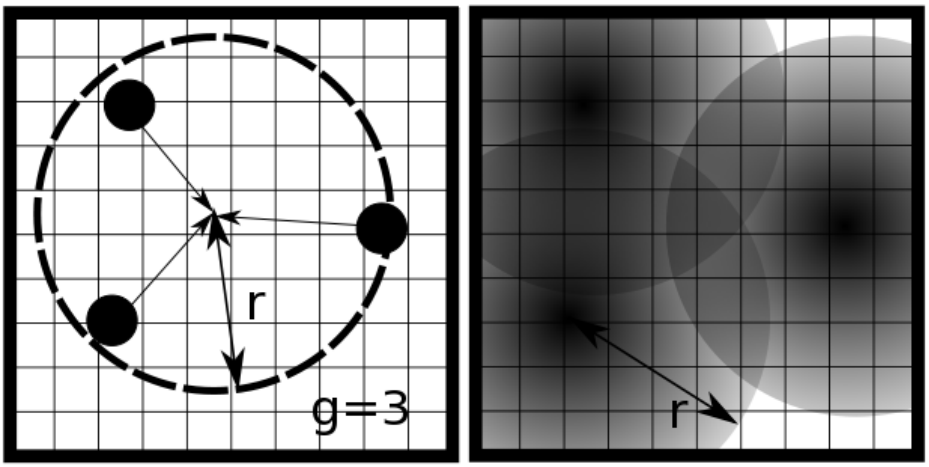


Figure 5.5: Showing different methods to do the photon gathering. **Left**: Gather all photons within radius `r` or find `g` closest photons. **Right**: Scattering of photon volumes that are additively blended.

There are other methods to render spheres except for instancing. Sigg et al. [2006] used point sprites to render spheres, using a single vertex to render each sphere, which in theory should be faster. However, this method has the drawback that on some hardware, the whole volume is culled if the vertex is outside of the viewport. Another drawback is that more fragments are wasted because the covered area is a square. Since the scattering can be fill-limited[11] [McGuire and Luebke 2009], this may have a large impact on performance. Instancing on the other hand has the drawback that the volumes are culled when the bounding volume is behind the camera or the near plane.

[11] Fill-limited means that the fill-rate of the GPU-card is the bottleneck in the algorithm. The fill-rate is the rate of fragments a card can process.

5.3.2 Method

Scattering of the photons as described by McGuire and Luebke [2009] was used instead in the normal photon gathering. Both instancing and point sprites were implemented to render the photon volumes.

The kernel was enhanced by multiplying it with a normal scaling factor ($\vec{n}_{photon} \cdot \vec{n}_{position}$).

5.3.3 Results

In Figure 5.6 it is shown that this method is considerably faster, with almost perfect output. Using instancing instead of point sprites, not much performance was lost (0.22 s with point sprites vs 0.30 s with instancing) and accuracy was gained.

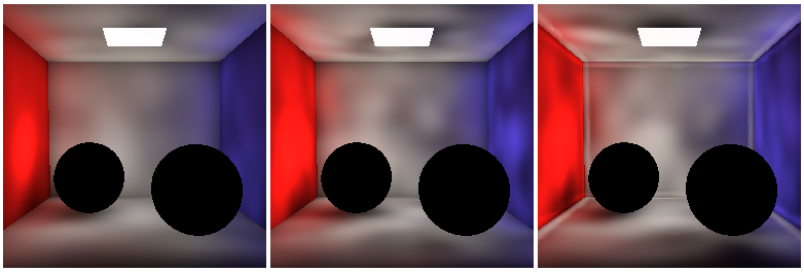


Figure 5.6: Left: Hash map (10 s). **Middle**: GPU gathering with normal scaling (0.59 s). **Right**: GPU gathering without normal scaling (0.55 s).

5.3.4 Discussion

McGuire and Luebke [2009] implemented a photon gatherer that achieved faster rendering speeds (0.10 s vs 0.59 s of a comparable rendering) than what our method accomplished. Their method had more optimizations than ours and we did not implement all parts of their algorithm.

5.4 Participating media

The default mode of rendering is to assume that the light can travel freely between the geometry in the 3D scene. However, how light interact with different particles distributed in the medium needs to be considered when rendering photorealistic images. This section will present the basal equations used to describe such phenomena, and a relatively trivial implementation to simulate this effect.

In reality, as light travels through a medium, it will collide with the particles inside of the medium. This can cause the light to change direction and decrease in strength as it is absorbed by the particles. The particles can also be emitting, which further increase the illumination. This interaction is called *participating media*. Simulation of participating media allows phenomena such as smoke, clouds, fog, water, and oceans to be rendered. Likewise, the color of the sky is also a result of light traveling through a participating medium (the atmosphere) which affects the light on different wavelengths causing it to look blue during the day and red during the twilight [Bruneton and Neyret 2008]. Another effect achieved by rendering participating media is *light shafts*, which are typically observed when light travels through a window in a dark dusty room allowing the viewer to “see” the light.

5.4.1 Previous work

This section will use the word *volume* defined as a region in space with a medium that alters the light that travels through it. A volume can in reality have a number of spatially varying parameters, but can be simplified for easier computations. Another term used for rendering participating media is *volume rendering*.

As light travels through a medium, there are three phenomena that need to be simulated in order to render realistic volumes. These phenomena are light *emission* , *absorption*, and *scattering* which are all described by Pharr [2010].

Emission is the phenomenon that occurs when particles in the medium emit energy in the form of light. In reality, light is emitted in all directions in a sphere around the particle. However, because of complexity, emissivity is only accounted for in the sample rays that travel in the medium. A volume that is strongly emissive and enlighten the surroundings, e.g. an explosion, can be approximated using regular light sources.

The second phenomenon, absorption, occurs when light energy decreases as it is absorbed by particles in the medium. This is modelled using an *absorption coefficient* σ_a, representing the probability that the light is absorbed by the medium.

Lastly, scattering both increases and decreases the light in the medium. *Out-scattering* further contributes to the extinction as light is redirected out of the sample direction. This is modelled using a *scattering coefficient* σ_s which gives the probability that light is scattered out of the light direction. The sum of the scattering and absorption coefficient is called *extinction coefficient* σ_t and describes the total loss in light:

$$\sigma_t = \sigma_a + \sigma_s. \tag{5.10}$$

The total ratio of extinction of the light from one point p to another p' is given by the *beam transmittance* which is given by the following formula where d is the distance between the two points:

$$T_r(p \to p') = e^{-\int_0^d \sigma_t dt}. \tag{5.11}$$

This integral can be further simplified when light travels through a homogeneous medium where it is expressed as *Beer's law*:

$$T_r(p \to p') = e^{\sigma_t d}. \tag{5.12}$$

As out-scattering scatter[12] light out of the light ray, *in-scattering* does the opposite and scatters incident light which contributes to additional energy, see Figure 5.7. This process is very complicated to simulate as the light can bounce an indefinite number of times before it reaches a certain point in the volume. When rendering participating medium, *single- and multi-scattering* is often considered differently.

[12]Scattering is the process where light is spread when it hits a surface or particles in the medium.

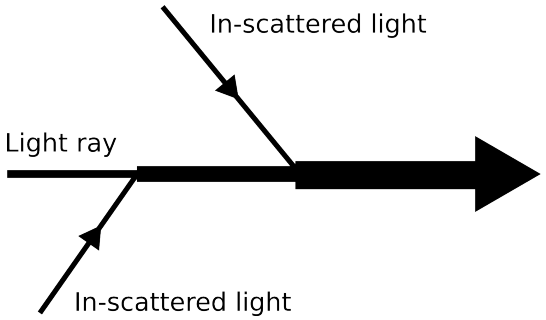


Figure 5.7: Light from the surroundings are in-scattered and contributes to the energy of the light ray.

Single-scattering is the less complex process of the two. It is the process when light travels from the light source, scatters one time, diverts towards the camera, and contributes to the final color. This is the major cause to the light shaft phenomena as light is blocked in the volume. Multi-scattering describes light as it bounces several times before it reaches the camera. This drastically increases the complexity of the problem, especially when you have to consider arbitrary geometry that occludes the light.

The final equation for the final luminance in a certain direction is following:

$$L_i(p,\omega) = Tr(p_0 \rightarrow p)L(p_0,-\omega) + \int_0^t Tr(p' \rightarrow p)S(p',-\omega)dt'. \tag{5.13}$$

where p is the position of the observer with a view vector ω. p_0 is the position on the surface where the light reflected and the the distance from p to p_0. S is the in-scattering function and gives the in-scattered light to the view direction at position p'. The in-scattering function is given by Equation (5.14) [Siegel 2002]:

$$S(\mathrm{p},\omega) = L_{ve}(\mathrm{p},\omega) + \sigma_s \int_{S^2} pf(\mathrm{p},-\omega \rightarrow \omega')L_i(\mathrm{p},\omega')\mathrm{d}\omega' \tag{5.14}$$

This function sums up the emissive light and all the light coming from all directions and multiplies it with the scattering coefficient and the *phase function* pf. This is a function that gives the amount of light scattered in the outgoing direction from the incident direction. Equation (5.13) gives the final luminance along a sample ray. The first term in the equation describes the extinction of light as it travels from the reflection point to the camera by multiplying the reflected light L_o. The integration sign sums up the in-scattering along the direction, and attenuates it using the beam transmittance.

5.4.2 Method

The rendering of volumes is done using *ray marching*. Ray marching is the procedure where, for each sample ray shot from the camera, the ray is stepped along the interval of the ray that is within a volume. This procedure is capable of rendering participating media using the physical Equations (5.10) to (5.13) described in previous sections. Even though this method is rather trivial it still renders accurate results, at the cost of being computationally heavy.

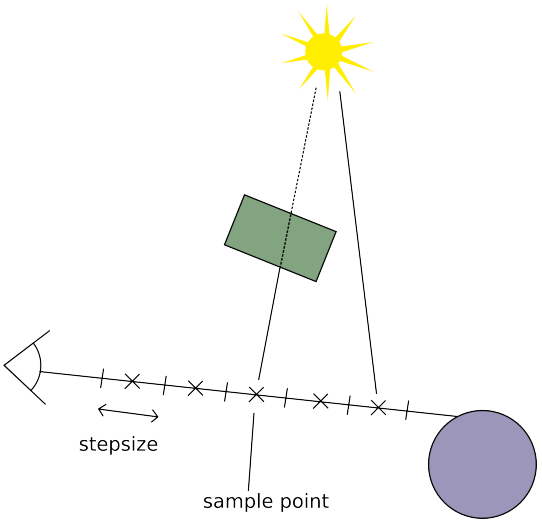


Figure 5.8: The ray is divided into segments with length *stepsize*. The incident light is evaluated at *samplepoint* for each segment and the resulting light for the ray is the sum of all segments.

The implementation has a number of simplifications and limitations. All parameters such as density are considered to be spatially uniform. Uniform density, in particular, allows the renderer to use Equation (5.12), Beer's Law [Beer 1852]. This removes one integral and simplifies the calculations greatly. Furthermore, the implementation does not allow volumes to be arbitrarily shaped, but only considers *oriented bounding boxes* (OBBs). The implementation also uses a simplified model for the scattering and phase functions. Scattering and absorption is only modelled as a single constant. Three different phase functions are provided; one simple isotropic function that scatters the light uniformly in a sphere, one more generic phase function (Henyey-Greenstein's phase function [Henyey and Greenstein 1941]) and its close approximation, the Schlick's phase function [Blasi et al. 1993]. These functions are modelled by the following equations:

$$pf_{isotropic}(cos\,\theta) = \frac{1}{4\pi}, \tag{5.15}$$

$$pf_{HG}(cos\,\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g(cos\,\theta))^{3/2}}, \tag{5.16}$$

$$pf_{schlick}(cos\,\theta) = \frac{1}{4\pi}\frac{1-k^2}{(1-k\,cos\,\theta)^2}. \tag{5.17}$$

The parameter g decides the shape of the phase function where $g = 0$ scatters uniformly, positive g scatters mostly forwards and negative backwards. The parameter k in Schlick's approximation behaves similarly. To get a value of k that behaves similarly as g, the following equation can be used:

$$k = 1.55g - 0.55g^3. \tag{5.18}$$

The implementation uses a fixed step size for the estimation of the integrals, in contrast to using a fixed number of steps along the ray. The advantage is consistent render quality throughout the volume independent of the distance traveled. The downside is the varying rendering time.

For extra artistic freedom, the scattering coefficient has been separated from the extinction coefficient. This allows the user to have media that spread the light extra without having a very dense looking volume.

The basic algorithm used is expressed as

$$L_i(p,\omega) = Tr(p_0 \to p)L(p_0,\omega) + \sum_{s \in steps} Tr(p_s \to p)\left[L_{ve}(p_s,\omega) + \sigma_s \sum_{l \in Lights} pf(p,\omega \to \omega_l)L_i(p,\omega_l)\right]. \quad (5.19)$$

5.4.3 Results

The results of a scene rendered with participating media at different step sizes are shown in Figure 5.9. Notice how bandings are very obvious when the step size is not small enough. Also notice how the render time varies almost linearly depending on the step size.

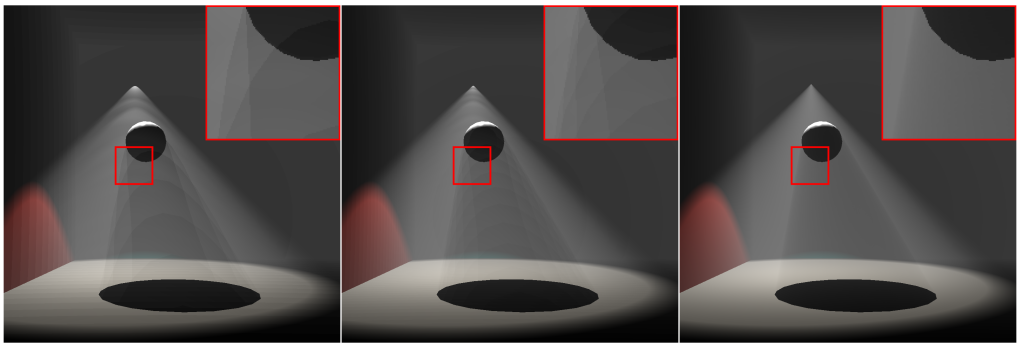


Figure 5.9: Each image is rendered with gradually decreasing step size. **Left**: step size = 0.1, render time = 14.70s. **Middle**: step size = 0.05, render time = 27.41s. **Right**: step size = 0.005, render time = 211.93s. Other settings were set as: extinction = 0.0, scattering = 1.0, and emission = 0.1.

Figure 5.10 shows how different settings can heavily affect the final results. Figure 5.10 have much increased extinction and emission settings in comparison to Figure 5.9 and renders the fog much more dense and milky looking.

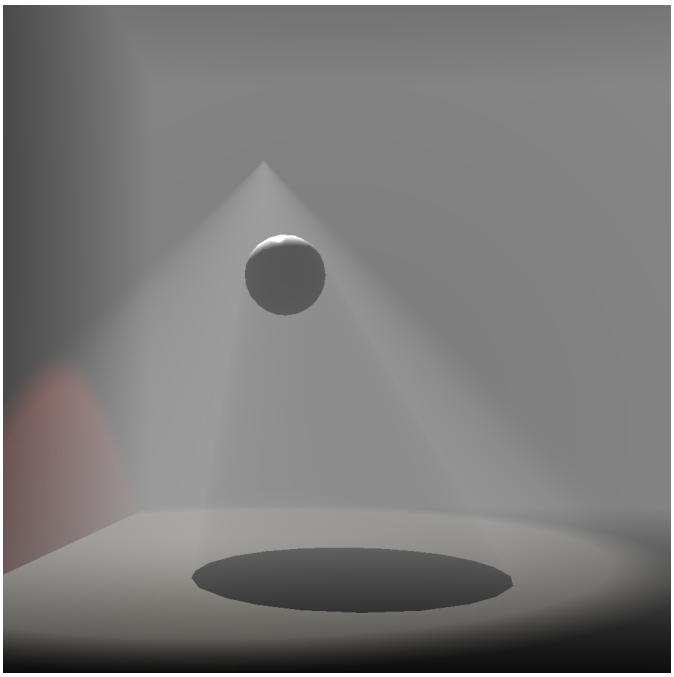


Figure 5.10: This image was rendered using the same scene as in Figure 5.9 with different settings: stepsize = 0.01, extinction = 0.8, scattering = 1.0, and emission = 0.5.

5.4.4 Discussion

The trivial integration estimation implemented causes the *bandings* observable in Figure 5.9. The integration is done by just accumulation of the sampled value for the given step multiplied by the step size. By storing more values from the function, this could be changed for a more advanced integration technique such as the trapezoidal and Simpson's rule.

Another possible extension is to allow different kinds of shapes of the volumes other than just OBBs in order to more closely fit the wanted shape. Globes, cylinders and slabs are other possible shapes. Furthermore, adding support for varying density of the volume is an obvious extension. One valid implementation is an exponentially varying volume which could be used to simulate the atmosphere and fog. For more precision, 3D grids could be used to hold simulation data for smoke and fire. Although, since Beer's law is only valid for homogeneous volumes, yet another integration needs to be sampled and solved, expressed in Equation (5.11).

For further increased realism, more advanced and specialised phase function could be implemented. For example, special scattering functions and phase functions exist for atmospheric scattering that needs to be taken into account if realistic sky rendering is wanted. However, it is also important to take the wavelength into account when rendering atmospheric scattering, since different wavelength are scattered differently. This is the reason of the different colors of the sky (blue color during day time and red and orange color during twilight).

The implementation described in this section only simulates single scattering. In order to achieve a more realistic effect, multi scattering has to be simulated as well. A few different methods to render this effect exist. However, since ELUMI implements photon maps to simulate global illumination, an

extension called volumetric photon mapping would be a valid choice for future development [Jensen and Christensen 1998].

5.5 Conclusions

A technique to accelerate ray tracing was described that achieved a speedup. Photon mapping was implemented, and the photon gathering was accelerated with a GPU giving a considerable speedup. Then participating media interaction within a ray tracer was shown to yield realistic results.

6

Presentation

Using the techniques described above a color value is obtained. Next we have to display this color value. This section will answer how we map a images of varying intensity to values that can be displayed on screen. A description then follows of how other data can be visualized to help develop a ray tracing engine.

6.1 Tone mapping

Computer displays and traditional image formats typically only use eight bits per color to represent the image data. However, when rendering realistic images, such limitations are undesired. For example, a typical surface lit by starlight only emits a luminance level of about 10^{-3} cd/m^2 while a surface lit by the sun typically has a luminance level of 10^5 cd/m^2. [Ledda et al. 2005], which illustrates the importance of this. Therefore, the light and color in the scene have to be able to represent this, especially with scenes with high dynamic contrast ratios[13]. Using eight bits leads to a precision of only 256 steps to represent luminance which, if not handled correctly, will neither yield correct nor realistic results.

Modern renderers are therefore always using either single-precision or, if necessary, double-precision to represent the color and luminance in the scene. However, rendering the scene to a buffer is only the first step in producing an image of a 3D scene. In order to give pleasing results, the rendered buffer also has to be correctly displayed. To tackle this problem, a separate pass is used on the rendered buffer called *tone mapping*. This section discusses one such operator and its implementation. Further extensions to handle more scenes with higher contrasts are discussed at the end of the section, as well as alternatives.

In general, there exist two categories of *tone mapping operators* (TMO): global and local. The difference is that global TMOs only work globally on the image. These operators typically calculate a value globally valid for the image ,e.g. the average luminance, and alter the pixels thereafter. In contrast, local operators analyse the local surroundings (and maybe also the global surroundings) and map the pixels accordingly. This means that local operators typically render more realistic images as they can take locally bright or dark spots in the image in account and make details within these regions visible. Although the local TMOs may render more pleasing results than the global TMOs, they typically take more time to run.

[13] High dynamic contrast scenes are here defined as scenes with very high differences of luminance values, e.g. scenes with both very dark and very bright regions (values greater than 1).

6.1.1 Previous work

There exist a range of different tone mapping operators, some more complicated than others. Two elementary operators are the *clamp* and *maximum to white* operators. The clamp operator is the most basic which cuts values larger than one to make the image renderable on the screen. This will however produce very bright images (if they have not been scaled previously). The other operator finds the brightest pixel in the image and linearly scales the whole image such that the brightest pixel represents one. One major disadvantage with this TMO is that if the image contains one very bright pixel, all other pixels will be rendered very dark.

Reinhard et al. [2002] introduced a tone mapping operator which probably is the most used operator. Although this statement is difficult to prove, one reason is that it generally produces pleasing results, shown in [Ledda et al. 2005]. Another reason is that it is relatively easy to implement, especially the global operator. Yet another reason is because of tradition. There exists a lot of example implementations and is the example TMO in DirectX SDK [Microsoft 2004].

The first step in the algorithm is to calculate the average luminance in the scene and is given in Equation (6.1). Reinhard proposes to use a *logarithmic average* when calculating this. The reason is not given in his paper, but Akenine-Möller et al. [2008] argued that this prevents very bright pixels from dominating the scene.

$$\bar{L}_w = exp\left(\frac{1}{N}\sum_{x,y} log(\delta + L_w(x,y))\right), \tag{6.1}$$

where $\bar{L}_w$ denotes the logarithmic average luminance of the image and L_w denotes the luminance of pixel (x,y). The result is then used to linearly scale each individual pixel using Equation (6.2):

$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y). \tag{6.2}$$

Equation (6.2) corresponds to mapping the average luminance (which can be translated as grey) to the scene's *key* value, which is denoted as a. Changing the key either gives darker (lower a) or brighter results (higher a). Reinhard suggests setting a to 0.18 for typical scenes. The effect of different values of a is shown in the left column in Figure 6.1.

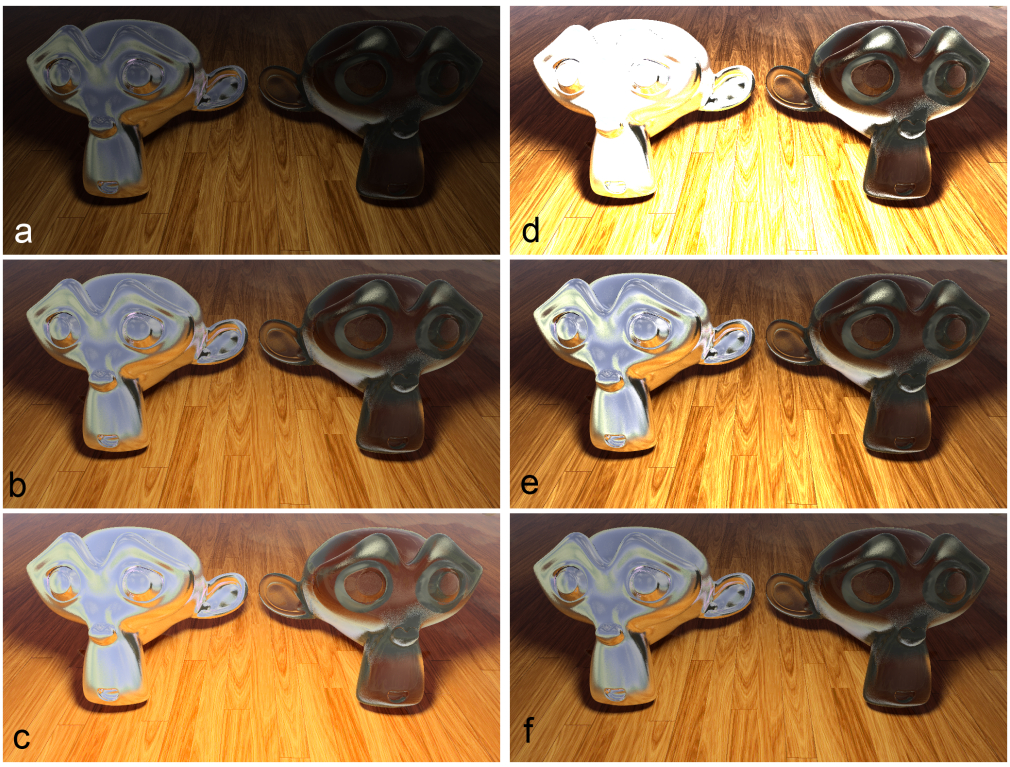


Figure 6.1: The left image shows how the *key* parameter affects the brightness of the tone mapped. **(a)** key = 0.18. **(b)** key = 0.54. **(c)** key = 1.44. The right column shows how the *white* parameter introduced in Equation (6.4) affects burn-outs. All images in the right column uses same key = 0.54 but with varying white value. **(d)** white = 0.68. **(e)** white = 1.86. **(f)** white = ∞.

The following equation maps the scaled luminance L to the range $[0,1]$. This is done using Equation (6.3) which gives the final luminance L_d:

$$L_d(x,y) = \frac{L(x,y)}{1+L(x,y)}. \tag{6.3}$$

This equation maps the range $[0,\infty)$ to $[0,1]$, which is the property sought after. Reinhard also suggests that this equation can be extended to Equation (6.4). This modifies Equation (6.3) to simulate *over-burn*, the effect often observed when looking into bright light, and only renders the nearby region white. The effect is shown in the right column in Figure 6.1.

$$L_d(x,y) = \frac{L(x,y)\left(1+\frac{L(x,y)}{L_{white}^2}\right)}{1+L(x,y)}. \tag{6.4}$$

This equation exposes yet another parameter L_{white} which enables the user to control the limit for over-burn. L_{white} corresponds to the level of luminance that is mapped to one (white). Observe how Equation (6.4) equals Equation (6.3) as the parameter L_{white} goes towards infinity

6.1.2 Method

The global Reinhard operator was implemented for ELUMI to provide with tone mapping algorithm in order to preview the rendered results. In order to implement the operator, the luminance levels of the colors in the image buffer are needed. Since tone mapping does not not yield realistic results when applied individually to each channel, a color transformation is needed. Reinhard et al. [2002] proposes to use the Yxy color space by transforming from RGB to XYZ and then to Yxy[14]. The luminance can then be read from and written to the Y-component of the vector.

6.1.3 Results

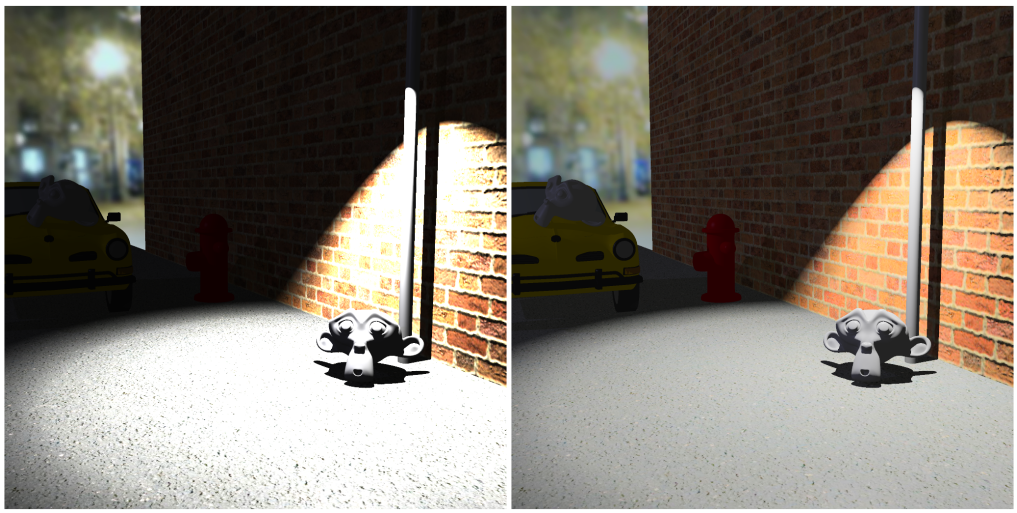


Figure 6.2: A scene rendered without (left) and with (right) the tone mapper. Notice how details are lost in the image without tone mapping, e.g. the fire post and the car is hardly visible in the dark and the texture of the wall and ground disappears in the bright light.

A scene which contains high contrasts is shown with and without the global reinhard operator in Figure 6.2. It also shows how the tone mapper can effectively process images that are heavily under- or overexposed and render them highly visible.

However, the correctness of a tone mapper is highly biased. Since tone mapping is designed to mimic how humans perceive luminance, it is not possible to logically deduce whether the tone mapping operator is "correct" or not, in difference to other procedures in computer graphics that is founded on physical phenomena and mathematical formulas such as shadows and reflections. One could argue over the correctness of the mathematics of the operator and how realistic the results look, but in the end, the result is still subjectively perceived.

There have been research and tests done to show how well different operators perform. Ledda et al. [2005] set up an experiment where a number of test subjects where to look at the results of different tone

[14] XYZ is another color space and an alternative to the more common RGB. The details of this standard is quite complex, however, it is useful to use its alternative form, Yxy, where the Y-component contains the luminance. For further details, see the publication by Smith and Guild [1931].

mapping operators and compare it with the "real" image displayed on a HDR-monitor. In this research, the Reinhard operator was only outperformed by one, much more complicated, operator (iCam).

6.1.4 Discussion

Only the global tone mapper was implemented in the renderer. Even though the local version of Reinhard's operator is relatively straightforward, certain steps in the operator require more complex computations. The pros were not evaluated important or valuable enough to be considered worthwhile the effort. Also, since the global operator turned out to be very efficient, this was deemed sufficient for our renderer.

One problem with Reinhard's initial proposal is that the key value of the scene has to be decided beforehand. Krawczyk et al. [2005] suggest a method for automatically decide the key value given in Equation (6.5).

$$a(\bar{L}_w) = 1.03 - \frac{2}{2 + log_{10}(\bar{L}_w + 1)} \tag{6.5}$$

The Reinhard operator, both the global and local, has been shown to be implementable in GPU hardware. Goodnight et al. [2005] and the co-researchers have published a paper where they discuss one way to implement the tone mapper on the GPU. Microsoft has also included an implementation in their DirectX SDK for use in GPU applications [Microsoft 2004]. Future work could therefore include an implementation of this operator in hardware.

There are quite a number of other operators, however more or less complicated. The iCam operator by Fairchild and Johnson [2002] was the operator that performed best in the tests done in Ledda et al. [2005]. However, the gains were deemed too expensive and too complicated for this project's scope. Another quite common tone mapping operator is the *Filmic Tone mapping Operator* used in the Uncharted 2 game [Hable 2010]. As the name implies, the goal of this operator is to achieve a more *filmic* experience with crisper blacks, often used in films. This is in contrast to the Reinhard operator which purpose is to reproduce the procedures of photography. The downside is that there are no papers with the details of the operator. Hable [2010] has published the formula used for the operator, however with a lot of constants that are not explained.

6.2 Visualizations

Graphics code was experienced inherently hard to debug. The easiest way to tell what is wrong is to *see* what is wrong, and to see what is wrong the wrong data has to be visualized. This section will answer which visualization methods we found useful when debugging our program.

6.2.1 Method

An application was written that used our renderer that had a number of different visualization capabilities. For displaying the geometry OpenGL was used because it is the biggest cross-platform graphics library. GLFW was used as a window library because of the freedom of threading it makes available. The user interface is run in a separate thread, preventing the `tracer` function from blocking the user input.

Since preserving the quality of the renderer's output is an important factor, a lossless compressed bitmap image format is desirable. Libpng, the official library for the png format is used to export the rendered scene to an image. It is cross-platform, open source, have a permissive licence, and has been used and tested extensively over the past 16 years [Roelofs 2012].

6.2.2 Results

The features we found useful when debugging our graphics code where:

General ray tracing

- View the pixels rendered as they render and aborting a rendering. This helps when looking at traversal patterns and also allows aborting a rendering early if something wrong is noticed.
- Move the camera and re-rendering the image from different locations enables visualization of bugs from different angles, which is crucial when pinpointing a bug.
- View intersection points, either without without geometry. This helps when debugging intersection tests and data structure traversal. It is also helpful when trying make sure a ray is going in the direction it is supposed to, for example at reflections.
- Visualization of normals, and materials was useful when making sure interpolation of vertices was done correctly.
- Turn GPU optimizations on and off, and view differences between the two modes was useful when debugging deferred renderered primary rays code.

Post-processing

- Since tone mapping is a destructive process, the final result is not always desirable. Therefore, being able to switch the tone mapper on and off was a valuable feature.
- Since the results of the tone mapper depends greatly on the parameters of the operator, adjusting parameters on the fly, such as outburn and key, turned out to be a valuable feature.

Photon mapping

- View the photons, either with or without geometry with photon intensity and color to debug the photon tracing.
- View only a subset of photons (for example only refracted photons) to debug the tracing of that subset.
- Viewing the paths photons have taken is useful when debugging reflection and refraction code.
- Adjust gather radius determine a good looking radius.
- Read and write the photon-map to disk making the photon tracing not necessary, which speeds up rendering and helps debug the photon gathering.

K-D-tree

- By viewing the split-planes of the tree it is possible to debug the ray/K-D tree intersection. For example, it is possible by viewing the split planes if triangles are missing in the tree leaf nodes. It can be seen

6.2.3 Discussion

Although it is very handy to have built-in tone mapping in the renderer to preview the rendered result, a more sophisticated tone mapping operator may be required. A valid extension of the exporter would be to allow HDR image formats as the target format. Since tone mapping is a destructive process, an uncompressed raw image format can sometimes be crucial for final image processings such as color and luminance correction.

6.3 Conclusions

The implementation of a tone mapper was described in this section. Then followed a description of the visualization techniques that were helpful to create this application.

7

Results

The ray tracer is able to replicate several of the phenomena observed in real life:

- Replicating light phenomena such as diffuse reflections, mirror reflections, refractions, and their glossy counterparts (Section 4.3)
- Simulating geometry with texture mapping (Section 4.5 and 4.6)
- Simulating light interfering with particles in the air (Section 5.4)
- Global illumination with diffuse interreflection. (Section 5.2)
- Producing images with natural color representations (Section 6.1)

The rendering speed is accelerated with a number of techniques and optimizations:

- The ray tracer supports multithreading in a scanline or Hilbert pattern. (Section 2.2)
- Shadow calculations are optimized with shadow caches. (Section 4.2)
- The intersection tests are sped up with an acceleration data structure. (Section 3.2)
- The first intersection test for each ray is optionally done on the GPU. (Section 5.1)
- The photon gatherer was implemented on the GPU. (Section 5.3)

Furthermore, image quality is augmented with supersampling in grid, jittered, and random patterns. (Section 2.3)

Our implementation gave as good results as a path tracer and was considerably faster, see Figure 7.1.

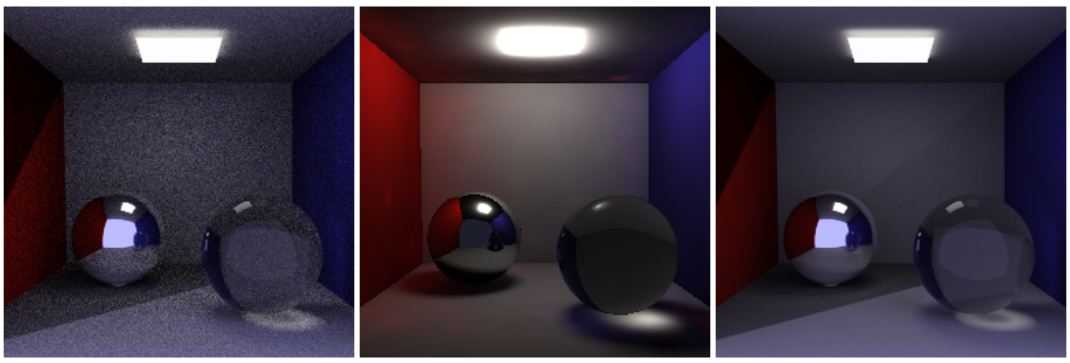


Figure 7.1: Different renders of a 256x256 image on an Intel Core 2 Quad Q8200 with an AMD HD4830 graphics card. **Left**: Path tracer of comparable rendering time to the middle image. 192 samples per pixel, 163 seconds. **Middle**: ELUMI ray tracer. 2x2 supersampling, 65536 photons, 185 seconds. **Right**: Path tracer of comparable image quality. 8192 samples per pixel, 7113 seconds.

8

Discussion

Since there are always more features to implement, a ray tracer can never be complete. We could for example accelerate the ray intersection on a GPU which could enable rendering in real-time. There are also several features that could be implemented that would further improve the image quality, like depth of field[15] and subsurface scattering. Another feature would be to support another model and material importer.

The distributed revision control tool Git has been a valuable tool for us. It has helped us to effectively collaborate and keep a structured workflow. Initially we had problems to get to grips with how we should divide the work between us. Since the code base was constantly changing we had problems with multiple people editing the same code. It was not until we had a stable base that we truly could start exploiting the functionality that Git offers.

Something that could have facilitated the whole process from modeling, texturing and illuminating to the actual rendering of the scene, could be to integrate our rendering engine into already existing modeling application. In our case, the 3D modeling application Blender would be a suitable choice since it is open-source, hence possible to integrate our own renderer into it.

Since rendering images with high resolution, quality, and complexity could take several hours, in some cases even days, to be completed, a technique could be implemented to speed up the rendering by distributing the `trace` function over several workstations. This would theoretically scale the processing power linearly with the number of total cores and their respective clock frequency.

[15] Depth of field is the effect of objects being blurred, caused by being out of focus.

9

Conclusions

Overall, we have fulfilled the set of conditions we set out to do. We have kept a balance between quality, speed, extensibility, configurability and compatibility. By using libraries for functionality that does not lie within the ray tracing algorithm, we have managed to implement numerous amounts of features. The OpenGL visualization has helped us maintain a consistent workflow; this is mainly because the visualization provides an approximation of the final render.

Before starting the development, a lot of effort went into making a modular architecture. This facilitated the development process since adding new features required few changes to the program structure. Each developer was able to create a new module without interfering greatly with the rest of the team. This resulted in a code base that was easily extendable for future work.

Almost every feature is configurable in text files loaded by the renderer. These are exposed to the user as global, scene, and material settings. For example, the user is able to choose the image size, supersampling procedure, traversal pattern, and recursion depth for reflections and refractions. This enables the user to balance quality against rendering speed. A user may for example choose to use additional samples, at the cost of longer rendering time.

The ray tracer was thoroughly tested and run on Windows, Mac OS X, and Linux during the whole development process. All features work on all platforms, with the exception of OpenGL 3 features on Mac OS X. However, whenever OpenGL 3 is not available, OpenGL 2 is used as a fallback solution.

In conclusion, this thesis presents an implementation of a ray tracer renderer that is capable of producing satisfactory images. Using an iterative development model and modular software architecture, it was possible to implement several features that both improved upon rendering quality and speed.

10

References

[Akenine-Möller et al. 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-time rendering*. AK Peters Ltd.

[Amdahl 2007] Amdahl, G. M. (2007). Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18), afips press, reston, va., 1967, pp. 483-485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *Solid-State Circuits Newsletter, IEEE*, 12(3):19 -20.

[Appel 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30-May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37-45, New York, NY, USA. ACM.

[Arvo 1994] Arvo, J. (1994). *Graphics Gems II*. The Graphics Gems Series. Academic Press.

[Beason 2008] Beason, K. (2008). smallpt: Global illumination in 99 lines of c++. `http://www.kevinbeason.com/smallpt/` (12 May 2012).

[Beer 1852] Beer, A. (1852). Bestimmung der absorption des rothen lichts in farbigen flüssigkeiten. *Ann. Phys. Chem*, 86(2):78-90.

[Bentley 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509-517.

[Bier and Sloan 1986] Bier, E. and Sloan, K. (1986). Two-part texture mappings. *Computer Graphics and Applications, IEEE*, 6(9):40 -53.

[Blasi et al. 1993] Blasi, P., Le Saec, B., and Schlick, C. (1993). A rendering algorithm for discrete volume density objects. *Computer Graphics Forum*, 12(3):201-210.

[Blender Foundation 2006] Blender Foundation (2006). Blender material nodes. `http://www.blender.org/development/release-logs/blender-242/blender-material-nodes/` (12 May. 2012).

[Blinn 1978] Blinn, J. (1978). Simulation of wrinkled surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 12, pages 286-292. ACM.

[Blinn 1977] Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192-198.

[Blinn and Newell 1976] Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542-547.

[Borman 2003] Borman, S. (2003). Raytracing and the camera matrix - a connection.

[Brawley and Tatarchuk 2004] Brawley, Z. and Tatarchuk, N. (2004). Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. *ShaderX3: advanced rendering with DirectX and OpenGL*, pages 135-154.

[Bruneton and Neyret 2008] Bruneton, E. and Neyret, F. (2008). Precomputed atmospheric scattering. *Computer Graphics Forum*, 27(4):1079-1086.

[Chacon 2009] Chacon, S. (2009). *Pro Git*. Springer.

[Christensen et al. 2006] Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. (2006). Ray Tracing for the Movie 'Cars'. *Symposium on Interactive Ray Tracing*, 0:1-6.

[Clairbois 2006] Clairbois, X. J. L. (2006). Optimal binary space partitions by.

[Cook 1984] Cook, R. (1984). Shade trees. In *ACM Siggraph Computer Graphics*, volume 18, pages 223-231. ACM.

[Cook and Torrance 1981] Cook, R. L. and Torrance, K. E. (1981). A reflectance model for computer graphics. *SIGGRAPH Comput. Graph.*, 15(3):307-316.

[Crow 1977] Crow, F. C. (1977). Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11(2):242-248.

[Dunlop 2005] Dunlop, R. (2005). Spherical texture mapping. `http://http://www.mvps.org/directx/articles/spheremap.htm` (12 May 2012).

[Engel and Engel 2005] Engel, W. and Engel, W. (2005). *ShaderX3: Advanced Rendering with DirectX and OpenGL*. ShaderX Series. Charles River Media.

[Fairchild and Johnson 2002] Fairchild, M. and Johnson, G. (2002). Meet icam: A next-generation color appearance model.

[Glassner 1989] Glassner, A. S. (1989). *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK.

[Goodnight et al. 2005] Goodnight, N., Wang, R., Woolley, C., and Humphreys, G. (2005). Interactive time-dependent tone mapping using programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA. ACM.

[Goral et al. 1984] Goral, C., Torrance, K., Greenberg, D., and Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 213-222. ACM.

[Greene 1986] Greene, N. (1986). Environment mapping and other applications of world projections. *Computer Graphics and Applications, IEEE*, 6(11):21 -29.

[Gritz 2010] Gritz, L. (2010). Osl introduction. `http://code.google.com/p/openshadinglanguage/wiki/OSL_Introduction` (24 Apr. 2012).

[Hable 2010] Hable, J. (2010). Filmic tonemapping operators. `http://filmicgames.com/archives/75).aspx` (25 Apr. 2012).

[Havran 2000] Havran, V. (2000). *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[Heckbert 1986] Heckbert, P. (1986). Survey of texture mapping. *Computer Graphics and Applications, IEEE*, 6(11):56-67.

[Heckbert 1989] Heckbert, P. (1989). Derivation of refraction formulas. *Introduction to Ray Tracing, (Andrew Glassner, ed.)*, pages 288-293.

[Heidmann 1991] Heidmann, T. (1991). Heidmann - 1991 - real shadows real time.pdf. *IRIS Universe Number 18*, 18:28-31.

[Henyey and Greenstein 1941] Henyey, L. and Greenstein, J. (1941). Diffuse radiation in the galaxy. *The Astrophysical Journal*, 93:70-83.

[Hilbert 1891] Hilbert, D. (1891). Über die stetige abbildung einer linie auf ein flächenstück. *Math*, 38:459-460.

[Horn et al. 2007] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. (2007). Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167-174, New York, NY, USA. ACM.

[Hunt et al. 2006] Hunt, W., Mark, W., and Stoll, G. (2006). Fast kd-tree construction with an adaptive error-bounded heuristic. *2006 IEEE Symposium on Interactive Ray Tracing*, 21(v):81-88.

[Immel et al. 1986] Immel, D. S., Cohen, M. F., and Greenberg, D. P. (1986). A radiosity method for non-diffuse environments. *SIGGRAPH Comput. Graph.*, 20(4):133-142.

[Jensen 1996] Jensen, H. W. (1996). Global illumination using photon maps. In *Rendering Techniques '96*, Eurographics, pages 21-30. Springer-Verlag Wien New York.

[Jensen and Christensen 1998] Jensen, H. W. and Christensen, P. H. (1998). Efficient simulation of light transport in scences with participating media using photon maps. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 311-320, New York, NY, USA. ACM.

[Kajiya 1986] Kajiya, J. T. (1986). The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143-150.

[Kaneko et al. 2001] Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., and Tachi, S. (2001). Detailed shape representation with parallax mapping. In *Proceedings of ICAT*, volume 2001, pages 205-208.

[Kensler and Shirley 2006] Kensler, A. and Shirley, P. (2006). Optimizing ray-triangle intersection via automated search. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 33 -38.

[Knoll et al. 2011] Knoll, A., Thelen, S., Wald, I., Hansen, C. D., Hagen, H., and Papka, M. E. (2011). Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *Proceedings of IEEE Pacific Visualization 2011*. (accepted for publication).

[Krawczyk et al. 2005] Krawczyk, G., Myszkowski, K., and Seidel, H.-P. (2005). Perceptual effects in real-time tone mapping. In *Proceedings of the 21st spring conference on Computer graphics*, SCCG '05, pages 195-202, New York, NY, USA. ACM.

[Laine et al. 2005] Laine, S., Aila, T., Assarsson, U., Lehtinen, J., and Akenine-Möller, T. (2005). Soft shadow volumes for ray tracing. *ACM Trans. Graph.*, 24(3):1156-1165.

[Lawder 2000] Lawder, J. (2000). Calculation of mappings between one and n-dimensional values using the hilbert space-filling curve. *School of Computer Science and Information Systems, Birkbeck College, University of London, London Research Report BBKCS-00-01 August*.

[Ledda et al. 2005] Ledda, P., Chalmers, A., Troscianko, T., and Seetzen, H. (2005). Evaluation of tone mapping operators using a high dynamic range display. *ACM Trans. Graph.*, 24(3):640-648.

[Lindholm et al. 2008] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39 -55.

[McGuire and Luebke 2009] McGuire, M. and Luebke, D. P. (2009). Hardware-accelerated global illumination by image space photon mapping. In *High Performance Graphics*, pages 77-89. ACM.

[Microsoft 2004] Microsoft (2004). Hdr lighting (direct3d 9). `http://msdn.microsoft.com/en-us/library/windows/desktop/bb173486(v=vs.85).aspx` (15 Apr. 2012).

[Möller and Trumbore 1997] Möller, T. and Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21-28.

[Musgrave 1988] Musgrave, F. (1988). Grid tracing: Fast ray tracing for height fields. *Research Report No. RR-639, Dept. of Computer Science, Yale Univ*.

[Nicodemus 1977] Nicodemus, F. (1977). *Geometrical considerations and nomenclature for reflectance*. US Department of Commerce, National Bureau of Standards.

[Ohbuchi and Aono 1996] Ohbuchi, R. and Aono, M. (1996). Quasi-monte carlo rendering with adaptive sampling. In *Computer Graphics Forum*, volume 15, pages 225-234.

[Oren and Nayar 1994] Oren, M. and Nayar, S. K. (1994). Generalization of lambert's reflectance model. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 239-246, New York, NY, USA. ACM.

[Parker et al. 1998] Parker, S., Shirley, P., and Smits, B. (1998). Single sample soft shadows. Technical report, University of Utah.

[Peano 1890] Peano, G. (1890). Sur une courbe qui remplit toute une aire plane. *Math*, 36:157-160.

[Pharr 2010] Pharr, M. (2010). *Physically based rendering from theory to implementation*. Morgan Kaufmann Elsevier Science distributor, San Francisco, Calif. Oxford.

[Pharr and Humphreys 2004] Pharr, M. and Humphreys, G. (2004). *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Phong 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18(6):311-317.

[Pixar Animation Studios 2005] Pixar Animation Studios (2005). The rispec. `http://renderman.pixar.com/products/rispec/index.htm` (12 May 2012).

[Prata 2004] Prata, S. (2004). *C++ Primer Plus, Fith Edition*. Sams Publishing.

[Ramamoorthi and Hanrahan 2001] Ramamoorthi, R. and Hanrahan, P. (2001). An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 497-500, New York, NY, USA. ACM.

[Reinhard et al. 2002] Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. (2002). Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21(3):267-276.

[Roelofs 2012] Roelofs, G. (2012). Libpng home page. `http://libpng.org/pub/png/libpng.html` (15 Apr. 2012).

[Saito and Takahashi 1990] Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197-206.

[Segal et al. 2008] Segal, M., Akeley, K., Frazier, C., Leech, J., and Brown, P. (2008). The opengl graphics system: A specification. `http://www.opengl.org/registry/doc/glspec33.core.20100311.pdf` (22 Apr. 2012).

[Siegel 2002] Siegel, R. (2002). *Thermal radiation heat transfer*. Taylor & Francis, New York.

[Siewert 2009] Siewert, S. (2009). Using intel® streaming simd extensions and intel® integrated performance primitives to accelerate algorithms. `http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms/` (10 May 2012).

[Sigg et al. 2006] Sigg, C., Weyrich, T., Botsch, M., and Gross, M. (2006). GPU-based ray-casting of quadratic surfaces. In *Symposium on Point-Based Graphics*, pages 59-65, Boston, Massachusetts, USA. Eurographics Association.

[Silicon Graphics 2006] Silicon Graphics, I. (2006). Opengl sdk reference for glviewport. `http://www.opengl.org/sdk/docs/man/xhtml/glViewport.xml` (20 Apr. 2012).

[Smith and Guild 1931] Smith, T. and Guild, J. (1931). The cie colorimetric standards and their use. *Transactions of the Optical Society*, 33:73.

[Smits 2005] Smits, B. (2005). Efficiency issues for ray tracing. In *ACM SIGGRAPH 2005 Courses*, page 6. ACM.

[Szécsi 2003] Szécsi, L. (2003). An effective implementation of the k-d tree. In *Graphics programming methods*, pages 315-326. Charles River Media, Inc., Rockland, MA, USA.

[Tatarchuk 2006] Tatarchuk, N. (2006). Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 81-112, New York, NY, USA. ACM.

[Veach 1997] Veach, E. (1997). *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University.

[Wald and Havran 2006] Wald, I. and Havran, V. (2006). On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61-69.

[Ward 1992] Ward, G. J. (1992). Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26(2):265-272.

[Welsh 2004] Welsh, T. (2004). Parallax mapping. *ShaderX3: Advanced Rendering With DirectX And OpenGL*, pages 89-96.

[Whitted 1980] Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343-349.

[Wikipedia 2012] Wikipedia (2012). Wikipedia: Hilbert curve. `http://en.wikipedia.org/w/index.php?title=Hilbert_curve&oldid=433722487#Applications_and_mapping_algorithms` (9 Apr. 2012).

[Williams 1978] Williams, L. (1978). Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270-274.

[Woo et al. 1990] Woo, A., Poulin, P., and Fournier, A. (1990). A survey of shadow algorithms. *Computer Graphics and Applications, IEEE*, 10(6):13 -32.

A

Example XML files

A.1 Scene XML file

```
<Object fileName="cornell.obj"/>
<Settings fileName="settings.xml"/>


<Camera>
  <Position x="0" y="0.717527" z="-2.69957"/>
  <Direction x="0" y="0" z="1"/>
  <Normal x="0" y="1" z="0"/>
</Camera>


<Light type="Point" intensity="1.5" falloff="Linear" visible="1">
  <Position x="5" y="1.2" z="2.9"/>
  <Color r="0.5" g="1" b="0.8"/>
</Light>

<Light type="Area" intensity="1.5" falloff="Quadratic" visible="0">
  <Position x="0.1" y="1.2" z="0.2"/>
  <Color r="1" g="1" b="1"/>
  <Axis1 x="0.5" y="0" z="0" samples="3"/>
  <Axis2 x="0" y="0" z="0.5" samples="3"/>
</Light>

<Light type="Spot" intensity="2" falloff="Linear" outer="45" inner="40">
  <Position x="-1.5" y="1.0" z="2.1"/>
  <Direction x="0" y="-1" z="0"/>
  <Color r="1" g="1" b="1"/>
</Light>
```

```
<Environment type="Cube">
  <Top src="land_top.png" />
  <Bottom src="land_bottom.png" />
  <Front src="land_front.png" />
  <Right src="land_right.png" />
  <Back src="land_back.png" />
  <Left src="land_left.png" />
</Environment>

<Volume type="uniform" absorption="0.0" scattering="1.0" emission="0.1">
  <PhaseFunction type="isotropic" />
  <Position x="-2" y="0.75" z="2"/>
  <U x="1.00725" y="0" z="0"/>
  <V x="0" y="1.00725" z="0"/>
  <W x="0" y="0" z="1.00725"/>
</Volume>
```

A.2 Settings XML file

```
<Screen width="1024" height="1024"/>
<Tracer version="4" pattern="1" batches="auto" first_bounce="0"/>
<Recursion maxDepth="10" attenuationThreshold="0.015625"/>
<Threading threads="auto"/>
<Tonemapping on="1" key="0.18" white="10"/>
<Tree version="3"/>
<Wireframe enable="0"/>
<Supersampling samples="1" pattern="0" />
<Photonmapper photonmap="1" photonmap_size="1024" final_gather_samples="0" photon
<Volume step_size="0.1" />
```

A.3 Material file

```
Ns 200              // Specularity constant
Ka 0.01 0.01 0.01   // ambient rgb
Kd 0.64 0.64 0.80   // diffuse rgb
Ks 0.30 0.30 0.50   // specular rgb
Ni 1.33             // Index of refraction
f 1.33              // Fresnel index
d 0.50000           // Transparency
r_spread 0.05       // Glossy reflection
r_samples 0.05
Ni_spread 0.05      // Glossy refraction
Ni_samples 10
map_Kd path         // Diffuse map
map_Ks path         // Specular map
map_Bump path       // Bump map
map_Norm path       // Normal map
map_R path          // Reflection map
map_D path          // Transparency map
axis 0              // Axis x,y, or z (0,1,2)
scale 0.1           // Scale texture
projector 0         // Projector function 0,1,2,3
corresponder 0      // Corresponder function 0,1,2,3
use_position        // Use position instead of normal
relief              // Use relief mapping
```