Real-Time Simulation of Global Illumination
Using Direct Radiance Mapping

Thesis by

Jeppe Revall Frisvad

Rasmus Revall Frisvad

Supervisors:
Niels Jørgen Christensen
Peter Falster

Informatics and Mathematical Modeling
IMM, Technical University of Denmark,
Kgs. Lyngby, Denmark

2nd November 2004

Abstract

This report presents our master's thesis on global illumination and real-time computer graphics written at the Technical University of Denmark. The thesis has two objectives. First objective is to identify and study different traditional methods for the two computer graphics fields called photorealistic rendering and real-time graphics. Following this study in known methods we introduce some of our own ideas for improvements on, or construction of, global illumination effects in real-time. One idea was chosen for a more detailed investigation. We have named the method contained within this particular idea "Direct Radiance Mapping" (DRM) and it is mainly a method for real-time simulation of diffusely reflected indirect illumination. DRM is implemented and compared to other methods, which are able to simulate the same visual effects. Capabilities, drawbacks, advantages, and disadvantages of the method are discussed on this background.

The second objective is to investigate the work flow that is necessary if we want to create a scene using a modeling tool and move that scene to an application that can render global illumination effects in real-time. The idea is to put together the fundamental tools needed if the results of this report were to be useful for creation of a commercial dynamic real-time application such as an architectural previewer or a computer game. To meet this objective we have studied the modeling application Blender and provided the report with an introduction to its usage. Through export and import scripts we are able to create scenes in Blender and use them for demonstration purposes in a separate Windows application.

Keywords: Computer graphics, radiometry, light transport, local illumination, global illumination, ray tracing, photon mapping, real-time rendering, modeling, Blender, direct radiance mapping.

Resumé

Denne rapport præsenterer vores eksamensprokekt ved Danmarks Tekniske Universitet, som omhandler global illumination og realtids computer grafik. Projektet har to formål. Det første er at identificere og studere forskellige traditionelle metoder inden for fotorealistisk rendering og realtids grafik. Efter dette studie i kendte metoder, vil vi introducere nogle af vores egne idéer til konstruktion af globale illuminations effekter i realtid. Én idé er udvalgt til en mere detaljeret gennemgang. Vi har valgt at kalde denne metode "Direct Radiance Mapping" (DRM, kortlægning af direkte radians). Metoden bruges primært til simulation af diffust reflekteret indirekte illumination. DRM er implementeret og sammenlignet med andre metoder, der også kan simulere lignende visuelle effekter. Egenskaber, manglende egenskaber, fordele og ulemper vil blive diskuteret på baggrund af denne sammenligning.

Det andet formål er at undersøge det arbejdsforløb, der er nødvendigt for at skabe en scene vha. et modelleringsværktøj. Modellen skal kunne overføres til et program, som er i stand til at rendere globale illuminations effekter i realtid. Idéen er at sammenføre de værktøjer der skal bruges for at skabe en kommerciel dynamisk realtids applikation, som f.eks. et computer spil. For at imødegå dette formål har vi studeret modelleringsapplikationen Blender og indsat en tutorial i rapporten. Gennem export/import scripts er vi i stand til at modellere scener i Blender og bruge dem til demonstrationsformål i en seperat Windows applikation.

Preface

This thesis is written by Jeppe Revall Frisvad and Rasmus Revall Frisvad in partial fulfillment of the M.Sc. degree at the technical university of Denmark (DTU). The work was conducted from February to September 2004. Supervisors of the project were Niels Jørgen Christensen, Associate Professor at institute of Informatics and Mathematical Modeling, and Peter Falster, Associate Professor also at the institute of Informatics and Mathematical Modeling (IMM), DTU.

The subject of this thesis is simulation of photorealistic rendering in real-time. Some modeling of three dimensional scenes will be addressed as well. The reader should understand the most basic concepts of rendering in computer graphics.

An accompanying CD-ROM containing developed software, source code, models, and other material related to this thesis has been attached in appendix A. All programming and modeling tools used for this thesis are free of charge and most of them are also open source projects.

s991020 Jeppe Revall Frisvad _____

s973867 Rasmus Revall Frisvad _____

Acknowledgements

In alphabetical order, thanks to: Philippe Bekaert for answering questions of mathematical nature. Andreas Bærentzen for clarifying discussions. Niels Jørgen Christensen for being supervisor of this project, and for pointing out important issues in many different contexts. Peter Falster for supervision, encouragement, and good discussions on core issues as well as peripherical details. Mikkel Gjøl for hints, critique and conversations along the way. Bjarke Jacobsen for good hints. Henrik Wann Jensen for confirmation of our believes with respect to a few uncertainties. Christian Lange, in particular, for constructive reviews of our early drafts. Bent Dalgaard Larsen for good hints and constructive critique along the way. Kasper Høj Nielsen for encouraging comments now and then. Per Slotsbo for helpful remarks during the beginning of the project. Also thanks to the Computer Graphics and Image Analysis Lunch Club for good company at lunch time. Thanks to our parents for always being there and last but not least thanks to Monica and Iben for good support and for coping with our absence at late hours.

Contents

Chapter 1

Introduction

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke (1962): *Profiles of the Future: An Inquiry into the Limits of the Possible*
Clarke's Third Law

Through some decades now people have tried to generate computer graphics that could replicate sceneries from real life. Today they have almost reached their goal. We see movies with computer animated scenes that are truly difficult to distinguish from real life. Soon even experts will find it hard, if not impossible, to tell whether or not a scene is taken with a real camera or created by an animator in a studio. This thesis concerns the fundamental theory and methods behind the making of synthetic images.

The creation of realistic computer graphics images is a heavy computational process, and it can take a long time to create small movie scenes even using the most powerful computers. The art of creating synthetic images replicating the real world very much depends on the ability to simulate how light interacts with its surroundings. This interaction is referred to as *illumination*. Throughout this project we will discuss two kinds of illumination; local and global. Local illumination means that the shade of each point in a scene is based solely on the directions towards the light sources and the direction towards the eye point (or camera position). Global illumination also takes into account the geometry surrounding each point in which the shade is to be determined. This has the effect that global illumination includes such effects as shadows, reflections, refractions, caustics, color bleeding, and translucency. The price is, however, high. Global illumination is a computationally expensive model and therefore the methods proposed to simulate it are usually not considered in the context of real-time graphics. Contrariwise many different approaches have been employed in order to create visual effects similar to global illumination effects in an otherwise local illumination model.

To achieve perfect realism it is clear that global illumination must be introduced or at least a simulation of the effects resulting from global illumination. Sometimes we have plenty of time to generate our image but there are situations where we are very short on time. Situations like this are found in dynamic computer applications as for example computer games. In such applications it is impossible to foresee every user interaction, hence, images must be generated on the fly. The calculations must be carried out so fast that the user does not register each new picture. The process of generating images fast enough for the human eye not to register is referred to as real-time illumination. In real-time computer graphics we often use local illumination methods as these traditionally are the only ones that can be calculated at sufficient speed.

The making of synthetic (2-dimensional) images from a three-dimensional representation (or model) by combination of lighting, texturing, and geometry is in computer graphics referred to as *rendering*. The following is a good description of what it takes for a rendering method to be called 'real-time' [2, p. 1]:

> The rate at which images are displayed is measured in frames per

> second (fps) or Hertz (Hz). At one frame per second, there is little sense of interactivity; the user is painfully aware of the arrival of each new image. At around 6 fps, a sense of interactivity starts to grow. An application displaying 15 fps is certainly real-time; the user focuses on action and reaction. There is a useful limit, however. From about 72 fps and up, differences in the display rate are effectively indetectable.

It is our intension and the first objective of this thesis to explore how close we can bring realistic image synthesis to real-time execution rates.

As real-time computer graphics and realistic image synthesis are two different branches of computer graphics we have chosen to start at each end and move them slowly in the direction of each other in the firm belief that we can make them meet somewhere mid-way. We know, however, that compromises must be taken both with respect to real-time (72 fps) and realism, otherwise this objective will not be met.

Since we start at both ends of a long rope stretching the distance between the global illumination and real-time rendering we must start out having a basic implementation in both camps. First we must have a basic real-time renderer as seen in most video games today. This is achieved using the standard tools available in a 3D graphics library such as OpenGL. Second we must have a decent renderer simulating global illumination. We have chosen to let this global illumination method be based on the rendering concepts known as ray tracing and photon mapping.

After the emergence and rapid development of GPUs (Graphical Processing Units) the prevailing way to implement real-time graphics is through hardware. The foundation of 3D computer graphics is points and vectors and the rendering of 3D models consisting of thousands of triangles results in millions of calculations including such mathematical entities, it is, therefore, evident that an efficient implementation of vector math is crucial if image synthesis is to be brought closer to real-time graphics. Fortunately the GPUs are developed, in essence, to process such calculations concurrently and efficiently. The GPU processing is, however, not reached easily. GPUs are constructed to render triangles in particular and the rendering pipeline that a GPU implements is not necessarily well suited for the different approaches that has been found for global illumination.

We have studied different aspects of global illumination theory to create the foundation which we think is necessary in order to come up with new approaches to global illumination effects in real-time. Several ideas for methods will be discussed in the report and the most promising one, which we have chosen to call "Direct Radiance Mapping", will be thoroughly examined. Direct radiance mapping is a method mainly for calculation of *diffusely reflected indirect illumination*. Diffuse reflection occurs when light is equally likely to be scattered in any direction [19], this happens when the scattering

material looks rather dull. Oppositely specular surfaces (such as mirrors and glass) reflect (and refract) light closely around a particular direction. Surfaces that have material properties in-between diffuse and specular are called glossy. When light has scattered around in a scene multiple times on arbitrary surfaces (be they diffuse, glossy, or specular), it is called indirect illumination. If the indirect illumination has scattered on diffuse surfaces at least twice before reaching the eye, it is called diffusely reflected indirect illumination. As other approaches for creating diffusely reflected indirect illumination in real-time already exists, we will compare our method with these. Some methods only treat perfectly diffuse surfaces (a hypothetical material that scatters light uniformly in all directions), such surfaces are often referred to as Lambertian surfaces.

Coming up with new ideas for real-time global illumination is not easily accomplished and to come even close to something useful, most of the existing theory must be studied thoroughly. With this in mind we have implemented several existing global illumination methods both to get familiar with them and to have them later for comparison with real-time results.

In scientific articles, papers or reports we often see simple test scenes demonstrating the treated algorithm or method. This is of course suitable for proof of concept purposes, but for the method to be applicable in commercial applications it must demonstrate efficiency in more complicated setups. As the second objective of this thesis we want to make a platform for development of real-time applications. We examine not only the rendering method but also the entire process from scene creation to rendering and use of it. This implies that we must be able to model scenes and therefore elementary use of a modeling tool is described in this report. We must also describe integration between the modeling tool and our rendering method. In this thesis we establish a connection from scene creation to rendering of it by our own methods using free of charge tools only. Finally we have created a small demonstration application, serving the purpose of arranging and presenting all the implemented methods, but also to demonstrate that a scene can be build from scratch, exported, and then imported in another application.

We have chosen to restrict this project in a couple of ways. First we have chosen not to emphasize on the process of software development, rather on the application of mathematical tools to solve a complex problem. Implementation is regarded as a tool for experiments and verification of the ideas that we put forth. With respect to computer graphics, we have chosen not to discuss matters of anti-aliasing in detail at any point. We also consider real-time soft shadow methods to be outside the scope of this report. No particular emphasis will be placed on spatial data structures for optimization in this report. We also do not go into parametric curves and surfaces at any point. All these subjects are left out to save time for other parts of the project.

1.1 Background

Prior to this project our knowledge in the field of computer graphics was limited to introductory courses. This is reflected in the report by some issues being described more comprehensively than perhaps necessary. This should be regarded as a documentation of the learning process that we have gone through during this thesis.

Other courses that we have followed may have had an impact on the outcome of the project as well. We have a background in autonomous agents and computational intelligence, which has inspired several of the ideas presented in this report. A prior knowledge of array theory has also been a great source of inspiration.

1.2 Report structure

The report is divided into four parts. **Part I** concerns the theoretical subjects which are the foundation of the project. In this part we consider basic knowledge in the field of global illumination and construction of virtual scenes. Many subjects are introduced since they make us able to appreciate the ideas and conceptions of later parts better. **Part II** concentrates on scene creation, that is, creation of contents for the rendering methods. In here we introduce methods for building a dynamic computer scene, since this is what we seek to render in real-time. While part I is theoretically minded part II is more practically minded. **Part III** presents the ideas that we have come up with for bringing global illumination closer to real-time rendering, and it gives a thorough description of the method we saw as the most promising of our ideas; direct radiance mapping. Part III will also present the demonstration application as well as the test scenes that we have created using the tools described in part II. **Part IV** contains discussion and conclusion of the thesis.

Part I

Chapter 2 concerns some of the most basic subjects in computer graphics, subjects that are common to all applications and rendering methods in 3D graphics. We describe how the basic elements in a computer scene are set up and we introduce our math engine, which does all calculations that are not carried out on the GPU. The math engine builds on array theory, which is therefore also introduced here.

Chapter 3 is about the theory behind an illumination model. The chapter describes the physical model of light and how it is transformed into computer graphics. In this chapter we describe the mathematical model which all computer graphic methods and algorithms seek to simulate or solve.

In other words this is the theory that is the foundation of all illumination models that are used in computer graphics.

Chapter 4 describes traditional approaches to realistic image synthesis. Here we describe some of the most commonly used global illumination methods: Traditional radiosity, traditional ray tracing, Monte Carlo ray tracing, and photon mapping.

In **chapter 5** we look at rendering from a slightly different angle as traditional methods for real-time rendering are treated here. For that reason this chapter will mainly concern local illumination. We will discuss the rasterization pipeline which is most commonly used for real-time rendering and we will take a brief look at textures.

In **chapter 6** different methods for simulation of global illumination effects in real-time are presented. Some of the methods presented here will later be used in combination with direct radiance mapping while some of the methods will be used for comparison with direct radiance mapping.

Part II

Chapter 7 takes a practical angle on creation of a computer scene using the theory given in chapter 2. Different more or less acknowledged methods for creating 3D objects will be introduced. The description of the methods uses the free of charge modeling tool called Blender for examples. This is the modeling tool that we have chosen for this project as a part of the work flow from scene creation to rendering that we wish to describe.

Chapter 8 discusses the visual appearance of materials according to their color and material parameters. In this chapter we seek to take a more practical approach as compared to that of part I, since the main purpose is to describe how materials can be set in a modeling application. Nevertheless, we also need to relate the different conceptions to the theory presented in part I.

In **chapter 9** we introduce methods for creating dynamic objects in a scene. This is an interesting subject which arises when computer graphics are available in real-time. Two different ways of creating dynamic objects are presented. First, we describe how to make animation sequences for use in our real-time application. Secondly, we describe how user interaction can define dynamic movement of objects and camera in a real-time environment.

Chapter 10 specifies how scene creation, material settings, and animation of dynamic objects are carried out in Blender. In this way we provide the basics needed to build a scene from scratch, which is a part of the development platform that we strive at. The chapter also reflects the learning process that we have gone through, since none of us new Blender beforehand, some of the experiences that we have had may be useful to others.

Part III

Chapter 11 presents different ideas that we have come up with during the project. Those are the ideas that we did not have time to examine in detail during this thesis. The chapter serves the purpose of inspiration and reflects the process that we went through before reaching the idea of direct radiance mapping.

Direct radiance mapping is the subject of **chapter 12**. Here we describe the method in details. The chapter will discuss abilities, advantages, and drawbacks of the method. Direct radiance mapping will also be compared to the real-time methods providing the same effects which were described in chapter 6.

In **chapter 13** we address other illumination methods that have been implemented during the project. These are mostly global illumination methods or real-time methods supplementing direct radiance mapping.

Chapter 14 describes our demonstration application. We will give a description of the different options that the graphical user interface of the application offers. The options all correspond to methods or visual effects described elsewhere in the report. Contents and purposes of the different test scenes will also be described.

Chapter 15 presents a number of design diagrams to describe the implementation behind the demonstration application. We will not describe every function in every file of the application in detail, but concentrate on the overall data flow.

Part IV

Chapter 16 discusses the outcome of this thesis and the course of the project. Some additional experiments and future applicabilities will be discussed here as well. **Chapter 17** concludes the report.

Part I

Real-Time Rendering versus Realistic Image Synthesis

In the following chapters we will introduce the fundamental theory and terminology that is necessary in order to describe the relationship between photorealistic rendering and real-time computer graphics. The first chapters discuss some of the theoretical and mathematical subjects that are the building blocks of computer graphics and realistic image synthesis. The last chapters will discuss different traditional approaches to photorealistic as well as real-time rendering, and some of the latest combinations of these.

The purpose of part I is to provide the basic knowledge of computer graphics that is necessary in order to appreciate the methods and ideas (such as direct radiance mapping) presented in part III. Some chapters may be a bit more comprehensive than needed if they only served the purpose of making the contents of part III understandable. However, we feel that the information provided in the following chapters (especially chapter 3) gives valuable knowledge some of which is rarely included in modern computer graphics text books. Therefore we consider our analysis of the theoretical foundation to be an important part of the learning process that led us to most of the ideas described in part III. The comprehensiveness of the following chapters also reflects a desire to build or implement elements from the bottom up, and thereby to get a full understanding of concepts in as many areas as possible. An example is the array-based math engine. For several reasons we chose to rebuild a previous implementation of a math engine: First of all we felt that there was a good chance of some minor improvements concerning processing speed. Secondly, it gave us a complete overview of functions available and the capabilities of the engine. If new functionality was needed we could easily extend the math engine. Furthermore the array-based implementation of the math engine bases most operations on a few general geometrical operators such that improvements of those few operators will improve the performance of the entire math engine significantly.

The math engine along with other fundamental mathematical tools that are often used in computer graphics are presented in chapter 2. The intention of this chapter is to sum up some of the key tools used in the rest of the report. The chapter starts with a description of the math engine based on array theory; hence the concepts of array theory will also be addressed here. The rest of the chapter sums up basic geometrical and computational issues often used throughout the report. All in all it should provide a good background for the rest of the report.

Chapter 3 concerns the rendering equation, which in some version or another is the equation that global illumination methods seek to solve. The chapter treats optics, optical radiation, and radiometry which are the fundamentals of the rendering equation. Other areas of research that can be related to computer graphics are discussed in short (eg. photometry). For reasons mentioned above chapter 3 digs a bit deeper than perhaps necessary.

After a thorough examination of the basic theories and physics of light we move on to the actual algorithms used in computer graphics. Photorealistic

rendering seek to solve the global illumination model and since we would like to simulate photorealistic rendering in real-time we start out with chapter 4, where we describe traditional methods for global illumination such as ray tracing and radiosity. Hybrid methods such as photon mapping are described and some expansions are addressed shortly.

Not only do we want to create photorealistic effects, we also want them in a real-time scenario. Traditional approaches to real-time rendering are described in chapter 5. Most real-time graphics are based on a local illumination model rather than a global. The reason is that the calculations needed for local illumination are much simpler and, hence, so are the computation times. This is necessary if the illumination of a scene needs to run in real-time. We will try to describe briefly how real-time graphics are done traditionally using rasterization and a local illumination model. Even though we seek to generate global illumination effects we find that much of our final implementation has to be based on rasterization in order to run in real-time. Therefore it is necessary to describe the basics of rasterization as well as global illumination techniques.

The last chapter of part I, chapter 6, seeks to combine global illumination and real-time rendering by treatment of different visual effects that exist in global illumination, but not in local illumination, individually. Some of the visual effects that have been approached using real-time techniques are: Shadows, reflections, refractions, translucency, caustics, and colour bleeding. The techniques for real-time simulation of global illumination effects are plenty: Shadow volumes, environment mapping, light mapping, etc. Some of these techniques are described in chapter 6. Normally methods for real-time global illumination focus on one of the visual effects, therefore there will be examples of different approaches to address different visual effects. In short chapter 6 seeks to give a brief presentation of what others have done to approach global illumination in real-time. This is important to us, since our own methods do not present a solution for all global illumination effects and neither does it rule out a combination with methods presented by others.

Chapter 2

The Building Blocks of Computer Graphics

What are you able to build with your blocks?
Castles and palaces, temples and docks.
Rain may keep raining, and others go roam,
But I can be happy and building at home.

Robert Louis Stevenson (1850–1894): *Block City*
from "*A Child's Garden of Verses and Underwoods*"

In this chapter we will describe some of the basic concepts and mathematical tools that are (or could have been) used in this project. There are no particular graphical methods in this chapter rather the mathematical foundation to build these methods from. Most of the chapter concentrates on basic vector math in a three dimensional world.

In section 2.1 we will present an array-based math engine that implements the geometrical calculations and vector math that has been used for the implementation of algorithms that are described in chapters to come, with the exception of calculations that take place on the GPU (Graphical Processing Unit). We have chosen to rebuild a vector library called CGLA that has been implemented by Andreas Bærentzen, and was distributed during the DTU "Computer Graphics" course (02561). Though we could have used CGLA or other implementations of more or less the same functions, we chose to make our own implementation inspired by CGLA. As mentioned before the reason for starting over is that we get a much better understanding of what is needed in graphical calculations and if some things need to be changed or modified we are able to do so faster and easier. Moreover the idea of a math engine based on a few array theoretic operators is appealing.

The geometry displayed in 3D computer graphics, and especially real-time computer graphics, often build on polygons. Section 2.2 will discuss ways of representing objects in a virtual environment such as a computer scene. In this project we use polygons for object representation; hence, the section will introduce how a clever representation of polygons can be put together.

A 3D virtual scene is made visible to us on a 2D screen. In the virtual environment this is enclosed by a view plane. The position of the view plane is determined by the position of the viewer or eye point, normally represented by a camera. Section 2.3 briefly presents the different elements in a typical virtual scene.

Another important issue in computer graphics is visibility and culling, which we address in section 2.4. A lot of computations can be saved if the invisible parts of a scene are ruled out when rendering. A good idea is to look closer at how we leave out backsides of objects not visible to the eye in a scene. This is normally referred to as culling.

To do graphics fast it is necessary to handle the data representing the objects of a scene in a clever way. Section 2.5 addresses spatial data structures and scene graphs, which are used to handle the huge amounts of data representing a scene in a smart way that can speed up processing time. The subject of section 2.5 will only be treated briefly, since we did not have time to implement these improvements in our final application. However, the section has not been left out entirely since spatial data structures and scene graphs ought to be implemented in future versions of our application.

2.1 An Array-Based Math Engine

The amount of vector math needed for computer graphics is limited. First a data structure must be created to represent a vector $\mathbf{v}$ in the n dimensional Euclidian space denoted $\mathbb{R}^n$. An n-tuple is an ordered list of real numbers, which is used for this purpose [2]:

$$\mathbf{v} \in \mathbb{R}^n \iff \mathbf{v} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} \text{ with } \quad v_i \in \mathbb{R}, i = 0, \ldots, n-1 \tag{2.1}$$

Basic math operations on vectors must be implemented efficiently. Vector addition is done componentwise:

$$\boldsymbol{u} + \boldsymbol{v} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} + \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} = \begin{pmatrix} u_0 + v_0 \\ u_1 + v_1 \\ \vdots \\ u_{n-1} + v_{n-1} \end{pmatrix} \in \mathbb{R}^n \tag{2.2}$$

Likewise multiplication of a vector and a scalar is done componentwise:

$$a\boldsymbol{u} = \begin{pmatrix} au_0 \\ au_1 \\ \vdots \\ au_{n-1} \end{pmatrix} \in \mathbb{R}^n \tag{2.3}$$

Applying an operation componentwise, that is, to each element of an array (or tuple), is in fact a general geometric operation. To avoid indices, which are inherent in the standard matrix notation, we employ a more functional notation given in *array theory*, founded by Trenchard More in 1973 [84, 85, 86].

Array theory introduces the concept of higher order functions. What is generally known as a mathematical function is more likely a first order function. The values which the function takes as arguments are seen as a function of order zero or data so to speak. To get a grasp of second order functions we need only think of the integral operator, the differential operator or the composite operator, which are the most commonly known second order functions. The functions of second order take functions of first order as arguments. Clearly the integral operator is an unary second order function, while the composite operator is binary. Table 2.1 gives a schematic view of function orders.

A generalization of the concept reveals that there is nothing to prevent us from constructing third order functions or even n^{th} order functions. However,

Logic	Mathematics	Array Theory	Examples
0^{th} order function	Value, element	Data (box)	$1, \pi, \frac{2}{3}$
1^{st} order function	Operation	Function (gin)	$+, \cdot, \sin$
2^{nd} order function	Operator	Transformer (rig)	$\int, ', \circ$

Table 2.1: A schematic view of function orders including a few examples.

since even the functions of second order are quite abstract and at many occasions difficult to grasp, it is even more difficult to imagine the specific use of third order functions taking transformers (or second order functions) as arguments. On the other hand transformers have shown their worth and the few well known transformers given as examples in table 2.1 are only the top of the iceberg, so we may yet also find a third order function which is practically useful.

When applying a function f to a value x the usual notation is $f(x)$ returning a value y being a function of the same order as x. To treat functions of arbitrary order it is important that we can separate the function from its argument, so that

$$y = f(x) = f\ \ (x) = (f\ x) = f\ x$$

this has a meaning when a second order function T is introduced. Suppose we would like to transform f into a different function according to a general geometric operation T then $g = Tf$ defines a new function g, which is the transformation of f according to T, meaning that

$$g(x) = (T\ f)(x) = (T\ f)\ x = T\ f\ x$$

The equation above indicates that an array theoretic expression is left associative.

Definition 1 (Left Associativity) *Expressions having left associativity are analyzed from left to right. Let T be a transformer, f a function and x a value then the expression $T\ f\ x$ first evaluates $T\ f$ and then combines with x.*

Left associativity rises some questions of interpretation given an arbitrary expression. The meaning of xf is for example not immediately clear. Consider the expression $a + b$, here $+$ is a binary function taking two arguments: $+(a, b)$. This indicates that we can interpret $a+$ as a new unary function adding the value a to its argument, that is, xf binds x to the first argument of the function f:

	Arrays A and B	Functions f and g	Transformers T and U
A	Strand $A\ B$ Array	Currying $A\ f$ Function	Currying $A\ T$ Transformer
f	Application $f\ A$ Array	Composition $f\ g$ Function	Alloying $f\ T$ Transformer
T	Annexing $T\ A$ Transformer	Application $T\ f$ Function	Composition $T\ U$ Transformer

Table 2.2: Pairs of objects that may be encountered in an array theoretic expression. In each cell the terminology is stated first then the specific pair and finally the result of such a pair. The table closely resembles the one presented in [101].

$$x\ f\ y = ((x\ f)\ y) = f(x, y) = f\ (x\ y) = f\ x\ y$$

Binding arguments with a function is in array theory and functional programming called *currying*, which is a terminology named after of Haskell Brooks Curry, the founder of combinatory logic [23]. The latter equality $f\ (x\ y) = f\ x\ y$ of the equation above shows an exception to the rule of left associativity, the exception arises since x and y are both arrays of data[1]. In that case they are considered to be a *strand*, that is, a successive juxtaposition of two or more arrays, which might as well be interpreted as a single nested array A with its first element being x and its second element being y.

Table 2.2 describes the interpretation of different pairs that we may encounter in an array theoretic expression. The general notion of composite functions give rise to the following associative laws:

$$(f\ g)\ \chi = f\ (g\ \chi) \tag{2.4}$$
$$(T\ U)\ \chi = T\ (U\ \chi) \tag{2.5}$$

[1] A single value is merely an array consisting of a single item only.

where χ is a function of arbitrary order. Annexing, (2.6), and alloying, (2.7), are also bound by associative laws:

$$(T\ A)\ \psi = T\ (A\ \psi) \tag{2.6}$$
$$(f\ T)\ \chi = f\ (T\ \chi) \tag{2.7}$$

where ψ is an n^{th} order function and $n > 0$. Finally currying is also associative when the currying function is of an order greater than one:

$$(A\ \sigma)\ \chi = A\ (\sigma\ \chi) \tag{2.8}$$

where σ is an m^{th} order function and $m > 1$. We will not consider the impact on the associative syntax if a third order function was introduced.

Instead, now that the most basic syntax is in place, we can define some of the transformers that will come in handy. First a very basic array theoretic transformer called EACH[2] is introduced.

Definition 2 (EACH) *Let A be an array of data and let f be an unary first order function, then*

$$\text{EACH}\ f\ A \tag{2.9}$$

is defined as the function f applied to each element of the array A.

Since an n-tuple is a special case of an array we can redefine multiplication of a vector and a scalar, (2.3), as

$$a\boldsymbol{u} = \text{EACH}\ (a \star)\ \boldsymbol{u} \tag{2.10}$$

where $\star$ is used for multiplication to make sure that it is confused neither with the dot product nor the cross product.

To deal with componentwise addition we introduce another transformer from array theory called EACHBOTH.

[2] In array theory transformers are traditionally written in capital letters.

Definition 3 (EACHBOTH) *Let A and B be two equally shaped arrays of data and let f be a binary first order function, then*

$$A \text{ EACHBOTH } f\ B = \text{EACHBOTH } f\ A\ B \tag{2.11}$$

is defined as the function f applied to pairs of elements found at corresponding positions in the arrays A and B.

In light of the EACHBOTH transformer componentwise addition, (2.2), is simply given as

$$\boldsymbol{v} + \boldsymbol{u} = \boldsymbol{v}\,\text{EACHBOTH} + \boldsymbol{u} \tag{2.12}$$

and we can easily define componentwise multiplication[3]:

$$\boldsymbol{v} \star \boldsymbol{u} = \boldsymbol{v}\,\text{EACHBOTH} \star \boldsymbol{u} \tag{2.13}$$

Componentwise subtraction and division, as well as division by a scalar follows from (2.12), (2.13) and (2.10) respectively by use of negated or reciprocal values.

Similarly the comparison functions $<, >, \leq, \geq$ are defined to work in a componentwise manner returning an array of Boolean values holding the result of each comparison. An example is:

$$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} < \begin{pmatrix} 1 \\ 2 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} \text{true} \\ \text{true} \\ \text{false} \\ \text{false} \end{pmatrix}$$

Equality, however, is expected to work in a slightly different manner. When testing the equality of two arrays or vectors we expect a single Boolean value as the result, that is, we expect the function to compare for equality componentwise and afterwards a logical & operation is applied to the resulting array deciding whether the two arrays were equal in all cases. In other words equality is given by an inner product using the equality and logical & operations instead of the more common multiplication and addition. An inner product is, in fact, a geometrical operation, which can be defined as a binary transformer taking two functions as arguments.

[3]Some texts (eg. [2]) use $\otimes$ for componentwise multiplication.

Definition 4 (INNER) *Let A and B be two arrays of data and let f and g be binary first order function, then*

$$A \text{ INNER } [f, g] \; B = \text{INNER } [f, g] \; A \; B \tag{2.14}$$

is defined as the inner product of A and B with respect to f and g, where f is the "reductive" operation and g is the distributive operation.
(Meaning that INNER $[+, \star]$ *is matrix multiplication.)* [102]

Hence equality of two vectors is given as:

$$\boldsymbol{v} = \boldsymbol{u} \quad \Leftrightarrow \quad \boldsymbol{v} \text{ INNER } [\&, =] \; \boldsymbol{u} \tag{2.15}$$

The INNER transformer is obviously also convenient in defining the dot product[4]:

$$\boldsymbol{v} \cdot \boldsymbol{u} = \boldsymbol{v} \text{ INNER } [+, \star] \; \boldsymbol{u} \tag{2.16}$$

and as stated in Definition 4 matrix multiplication is given similarly:

$$\boldsymbol{AB} = \boldsymbol{A} \text{ INNER } [+, \star] \; \boldsymbol{B} \tag{2.17}$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times p}$.

Why go through all this theory in order to describe a relatively simple vector math library? Because it reduces the amount of work we have to do to implement it, and even more important; there is a good chance that we can do the implementation more efficiently, since the few transformers that have been used each can be implemented efficiently with the result that the entire library gets more efficient. In fact the transformers described in Definitions 2, 3, and 4 already have an efficient implementation in the C++ standard library.

The terminology in C++ is quite different, see [125]. Here transformers are referred to as *adapters* and since the functional syntax described above does not fit into the syntax of the procedural C++ programming language they use a special case of adapters, namely binders, to describe currying. Hence `bind1st` and `bind2nd` of the C++ standard library corresponds to $x\,f$ and y CONVERSE f respectively, where x and y are first and second argument to the binary function f and CONVERSE is a transformer that swaps the arguments of a binary function.

(2.9) and (2.11) are available in C++ as an overloaded function `transform` that treats either an unary function and input and output arrays or a binary

[4]Which may be short for the "inner plus dot product".

function and two input and one output array. Likewise (2.14) is implemented as the function `inner_product`. The exact description of these C++ functions are given in [125].

Moreover the theory that has been described in this section gives the fundamentals needed in order to describe functional algorithms mathematically, hence we will draw upon it in sections to come when we see fit.

In the following section we will describe the basics of polygons and how 3-dimensional virtual worlds are composed of polygonal geometry. The subject of section 2.2 may at first seem relatively distant from the math engine, but a mathematical representation and treatment of polygons goes hand in hand with vector math.

2.2 Polygonal Geometry

Definition 5 (Polygon) *A closed figure in the plane given by points* $p_0, p_1, \ldots, p_n$ *and bounded by line segments* $p_0p_1, p_1p_2, \ldots, p_{n-1}p_n, p_np_0$ *[98].*

As stated in definition 5, a polygon is defined as a closed planar figure bounded by line segments connecting vertices such that they enclose one and only one region. The following list describes the properties of a polygon [112, p. 245]:

1. The number of vertices in a polygon equals the number of its sides.
2. The number of vertex angles of a polygon equals the number of its sides.
3. Each side of a polygon is a side of two vertex angles.
4. A vertex angle is *not* a straight angle ($\neq 180°$).

All polygons can easily be divided into polygons of the lowest order; triangles. Triangles have certain appealing properties, an example is that triangulation of all polygons largely will eliminate view-dependent interpolation effects [19], which result from linear interpolation of shade or colors across a polygon. This is exploited by graphics hardware specializing in fast triangle processing. The result is that almost all real-time application base their graphics on triangle meshes, since this gives the highest possible resolution at a very low rendering time. The level of detail of an object depends on the number of triangles used in the mesh and so does the processing time for rendering the object.

Figure 2.1 shows an example of a cylinder represented by polygons. The figure also shows that polygons can be generated from the connection points or the edges between them. Polygonal objects are often represented by hierarchical data structures. Each object is defined by pointers into a list of surfaces and each surface by pointers into a list of vertices. Normally the data structures are optimized, so that each point or edge only needs to be stored once [135].

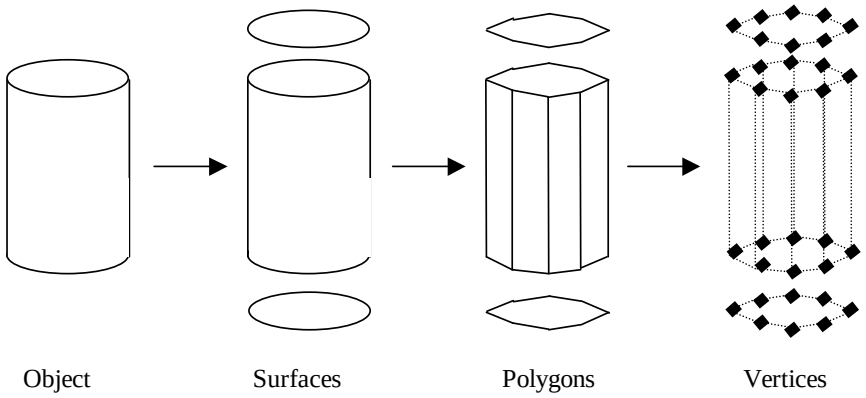


Figure 2.1: An object is represented by surfaces, which are represented by polygons, which are represented by vertices or edges. This figure is a combination of figure 1.1 in [135, p. 5] and figure 2.4 in [134, p. 38].

In computer graphics a vertex is a geometric entity consisting of a point in space, an associated normal, and possibly parametric (u, v)-coordinates specifying the position of the vertex on the surface of the object which it is a part of. The vertex position is represented by three coordinates. When manipulation of these positions is needed, it is evident that the math engine described in the previous section come in handy.

The vertex normal is used for shading. Shading is described in subsequent chapters of this part. Each polygon need to have a *face normal* defined as well (the face of a polygon is short for its surface). The face normal is the true geometric normal to the plane containing the polygon. Face normals are used for example in culling algorithms (section 2.4 relates to this subject). Sometimes it can be appropriate to store the edges between polygons as well as vertices, since they can be useful in shadow calculations where in principle only the silhouette of the object casting a shadow is interesting. A group of polygons forming an object is called a polygon mesh.

The biggest drawback of the polygonal representation is that the detail level of objects very much depends on the number of polygons used for its creation. Since all polygons are plane objects, curves in an object can only become more precise if more polygons are added. Moreover each manipulation of an object must be carried out on each polygon present. A high polygon count can therefore create a computational bottleneck on the CPU

(Central Processing Unit). An attempt to solve the problem is the rapidly developing GPU (Graphical Processing Unit) which is the core part of modern graphics cards. The GPU is concerned mainly with operations on polygons (in particular triangles).

The number of polygons is not the only issue. GPUs today are so fast that the problem is actually not rendering the required amount of triangles, the limitation lies in the amount of data that it is possible to transfer between the CPU and the GPU [134].

One way to speed up processing time is, therefore, to reduce, as much as possible, the data that need to be transferred. To do this we can either reduce the amount of data stored for each vertex, or seek to reduce the number of vertices. The latter option is typically done by exploiting that many polygons may share the same vertices or by simplifications of the object details according to the needs of a particular view (this concept is often referred to as level of detail, or LOD).

Having introduced the basic drawing unit that will be used with few exceptions throughout this project, namely polygons and especially triangles, and having presented a way to implement a basic math engine to process mathematical operations on points and vectors in three dimensions (as described in section 2.1), it is now time to describe the contents of a traditional three-dimensional virtual scene, which is the subject of the next section.

2.3 The Virtual Scene

When watching computer graphics, either on a TV or on the computer screen, we are presented with a window into a virtual world. In the photorealistic case this world often replicates our own world. This means that what we see is a three dimensional virtual environment.

To simulate this in a fairly realistic way we must represent all elements or objects in a scene in three dimensions. We must also define where we want to place the viewer in the scene and in which direction she should be looking. In movies we have a predetermined route for the viewer to follow, which means that we know exactly what will be visible to her at any given time throughout a sequence of pictures. This enables us to pre-calculate all the necessary pictures, meaning that we in principle are able to spend as much time as we like for each single image, or frame, in a movie sequence. In a dynamic application such as a computer game this is, however, not the case. There is no way to determine the exact movement of the viewer, that is, we need to create each image, or frame, on the fly according to the current location and direction of the viewer.

To produce a picture we need to keep track of all elements in the scene and most importantly the position of the viewer and the direction in which she is looking. Figure 2.2 represents a simple virtual scene in two dimensions.

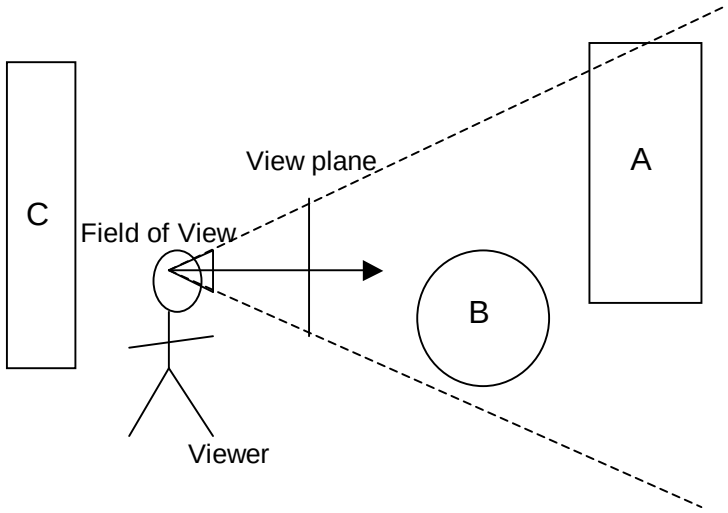


Figure 2.2: The visible part of the scene depends on the location of the viewer, the field of view (which is an angle specifying the size of the view plane), and the direction, which the viewer is facing. In this case objects A and B will be partly visible to the viewer, while object C is not visible at all. (Note that object B is also only partly visible, since the viewer can not see the backside of the sphere, or circle in the 2D case.)

The view plane is the virtual representation of the screen and the volume subtended by the field of view encloses the parts of the scene that becomes visible when projected onto the screen. This means that the field of view is an angle defining the visible area.

In figure 2.2 the front of object B is fully displayed while the front of object A is only partly visible. Some of the object lies behind object B and a part of the top corner will be missing since it is outside the visible area. To include the top corner of object A we can either move the eye point up or backwards or we could make the view plane larger by a broadening of the field of view. Object C is present in the scene but not inside the visible area, hence, we can usually leave out C in the calculations until it becomes visible. By knowing what is visible and what is not, we can save many computations. This is the subject of section 2.4.

The eye point and view plane are normally represented by a camera. The functionality of a camera is comparable to the functionality of eye and view plane; you frame out the part of the world that you want to preserve when you chose a motif for your picture. What happens outside the picture is cut off and forgotten. The simplest model of a camera is the pinhole camera, which is shown in figure 2.3.

As in figure 2.2 the front of objects C, D and partly B will be visible on the screen. Objects A and E are both invisible due to the near and far clipping planes. The clipping planes are in principle not a part of the pinhole camera model, but are practical especially in large scenes. The clipping planes simply

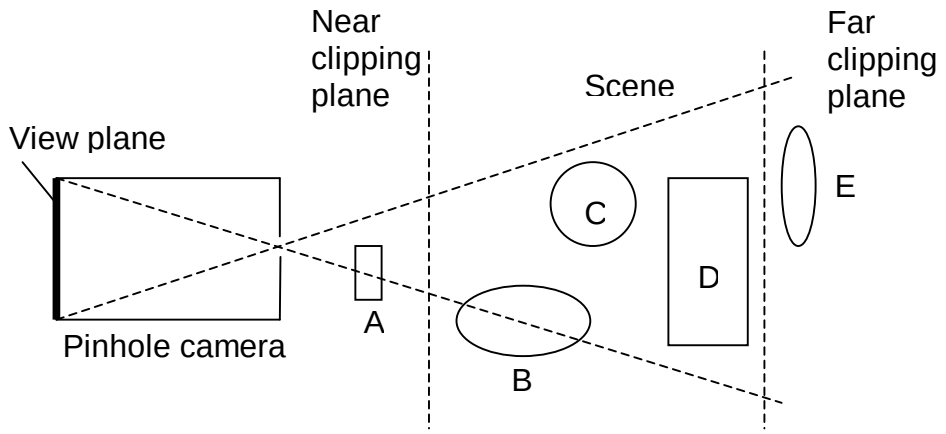


Figure 2.3: In computer graphics the eye point and view plane are represented by a camera. This figure shows the simplest camera model; the pinhole camera and how it captures a scene.

rule out objects that are too close or too far away. When using near and far clipping planes, the volume subtended by the field of view is cut by two parallel planes and is, hence, called a *view frustum*. Rasterization, which is the rendering method used for real-time rendering most often, can effectively cut away all objects that are not partly or fully contained within the view frustum. This is not always possible in photorealistic rendering, since light may be reflected off objects residing outside the view frustum.

Normal cameras adjust the size of the view plane by use of different lenses placed in the hole. Lenses are also used for adjustment of the visible area. The simple pinhole camera has no lens, therefore if the visible area, or the field of view, is to be changed, we need to change the size of the camera by changing hight, breadth, or length and thereby increase the size of the view plane at the bottom of the camera. This is similar to moving the eye point or opening the eye more in figure 2.2. In this case a lens is much more convenient. Moreover a lens also lets in more light, which makes it preferable in most cases.

There is another difference between a pinhole camera and a lens camera. In the ideal pinhole camera everything is in focus. This has the effect that computer graphics often produce synthetic images that tend to have an "unnatural" sharpness about them. Lens cameras (and the eye as well) has a certain adjustable distance at which the pictured objects will be sharp, meaning that we can only keep focus on objects in a given depth interval, while the pinhole camera has an infinite depth of field [3]. In real life the lens camera is preferred, since it is practical to keep the camera in the same size and we would like a lot of light to pass through the camera. In computer graphics our camera is arbitrary and we can change the size and light without any problems. Therefore we prefer to use the much simpler pinhole

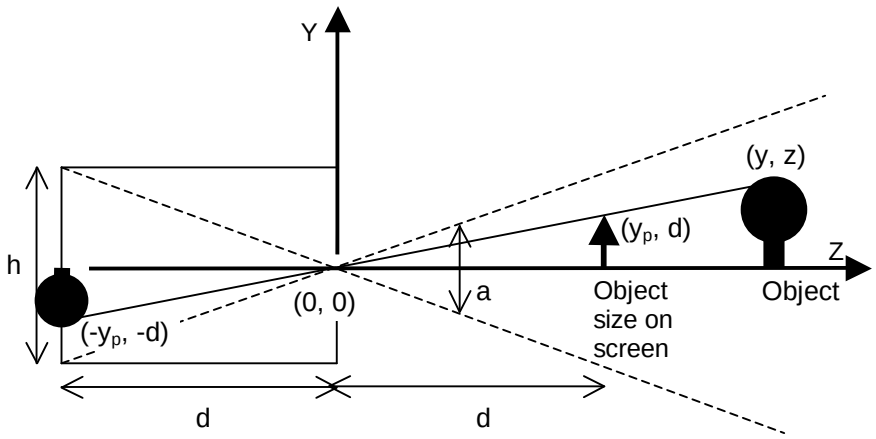


Figure 2.4: The pinhole camera is used as a model for the virtual camera in our scene. A more detailed description can help us define some parameters for determining our view. The figure is a combination of figures 1.15, 1.16 and 1.18 in [3].

model.

Looking closer at the pinhole camera model a few parameters can define our view. Figure 2.4 shows the pinhole camera again, this time including parameters. The angle a is the field of view. Considering figure 2.4 it is seen that the lines of length d and $h/2$ defines a triangle of which:

$$\tan\frac{a}{2} = \frac{h}{2d}$$

giving us the following way to calculate the field of view:

$$a = 2\tan^{-1}\frac{h}{2d} \tag{2.18}$$

where h is the height of the view plane and d is the distance to it from the pinhole (In three dimensions h is sometimes specified as the diagonal of the view plane, see eg. [14]). In practice we see that the closer our eye point gets to the view plane the bigger angle we get and the larger an area of the scene becomes visible. Normally d is the parameter that is changed, since the size of the screen then remains the same. Setting d is equivalent to choosing the right lens for a camera.

The object in front of the camera in figure 2.4 is projected to the view plane through the hole. Since there is no lens in the camera it is possible to draw a straight line between a point in the scene and the same point in the view plane. The line passes through the origin (the hole) and the angle between the z-axis and the line will be the same on each side of the y-axis. The size of the object depends on d, this follows the simple rules of projection where $(0, 0)$ is the center of projection. The point on the view plane will be:

$$(y_p, z_p) = (y_p, -d)$$

where

$$y_p = -\frac{y}{z/d} \tag{2.19}$$

and a similar calculation can be made for x_p:

$$x_p = -\frac{x}{z/d} \tag{2.20}$$

The resulting image on the view plane, which is incident with the back-side of the pinhole camera in figure 2.4, will as indicated be turned upside down, this is called *back projection*. In order to achieve *front projection* by projection of the image back through the origin to a view plane in front of the camera (at position $z = d$), we need merely change the sign in equations (2.19) and (2.20) [14].

This section has focused on a virtual scene with a coordinate system with origin in the eye point and z-axis along the line of sight. A virtual scene has, however, several coordinate systems to keep track of. The space with a coordinate system as the one used in this section is called eye (or view) space. Besides eye space we have a world space, which has a predetermined coordinate system that globally stays the same throughout a rendering session. Objects, lights, and the viewer are placed in world space. Sometimes each object has its own local coordinate system around which it was modeled, this is called object (or model) space. When rasterization is used for rendering, the view frustum (including its contents) is transformed into the unit cube, this space is called clip space. Last we have the two-dimensional window coordinate system which is the coordinate system of the screen or view plane. These spaces each have their purpose, and they are described in more detail in chapter 5. In the following we shall shortly describe how transformations are represented mathematically in computer graphics.

Homogenous Coordinates

Consider a point in space $\boldsymbol{p} = (p_x, p_y, p_z)$ and a vector in space $\boldsymbol{v} = (v_x, v_y, v_z)$. The point describes a location, while the vector describes a direction and has no location. Both are represented by the same three-tuple, which makes it difficult to distinguish between them with respect to transformations.

We can perform linear transformations, such as rotations, scalings, and shears, on a three-tuple using 3×3 matrices (this will be described in chapter 9). This suffices for transformation of vectors, since they do not have a location. If we, however, want to translate a point it is not possible using a 3×3 matrix. Because of this obvious limitation to the three-tuple representation of points and vectors computer graphics employs a mathematical tool called homogenous coordinates.

Suppose we represent vectors and points using a four-tuple (x, y, z, w). Then, when $w \neq 0$, homogenous coordinates are given as:

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right)$$

In other words we let points and vectors be defined in three-dimensional projective space (or projective three space). Now, transformations can be represented by 4×4 matrices:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & t_x \\ m_{10} & m_{11} & m_{12} & t_y \\ m_{20} & m_{21} & m_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

is the same transformation matrix as the 3×3 matrix that, as mentioned, can be used for rotation, scaling, and shearing. (t_x, t_y, t_z) is the translation of a point to which this transformation is applied.

It now becomes clear that points are given in homogenous coordinates as $\boldsymbol{p} = (p_x, p_y, p_z, 1)$ and vectors are given as $\boldsymbol{v} = (v_x, v_y, v_z, 0)$. In this way vectors will be unaffected by the translation, while points indeed will be translated.

Even though we have changed to projective three space, matrix-matrix multiplications and matrix-vector multiplications are still the same. Therefore the homogenous coordinates are very useful.

Previously equations were given for projection of a three-dimensional virtual scene into the view plane representing the screen output. In fact we could say that the scene (in eye space) is represented in two-dimensional projective space with respect to the view plane.

Having the above description of homogenous coordinates in mind, we can define a (4×4) projection matrix $\boldsymbol{P}_p$ finding the perspective projection of a point $\boldsymbol{p}$ according to (2.19) and (2.20):

$$\boldsymbol{q} = \boldsymbol{P}_p \boldsymbol{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \tag{2.21}$$

where $\boldsymbol{q}$ is the resulting point on the view plane given in projective three space.

With a basic knowledge of the virtual scene, the representation of objects, and a simple camera model, we can move on to a few efficiency schemes that

are often useful in computer graphics. Section 2.4 describes how visibility can be exploited and section 2.5 shows how spatial data structures often can be an advantage.

2.4 Hidden Surface Removal

When a scene is rendered there is usually (at least in the case of local illumination) no need to spend unnecessary time calculating lighting and shading conditions for object parts that are partly occluded or not visible at all. The previous section showed how only a part of the scene is visible to the virtual camera. This section will discuss how to rule out invisible objects or parts of objects before doing expensive lighting calculations.

There are three steps to go through when removing hidden surfaces. First of all we must remove all objects outside the visible area, the visible area corresponds to the view frustum, see section 2.3. In figure 2.5 the visible part of the scene is bound by six planes and in a moment we will show how to find them. Furthermore we can remove all back facing surfaces, that is, surfaces with normals pointing away from the viewer and last we can remove all parts of objects that lie behind other objects in the scene.

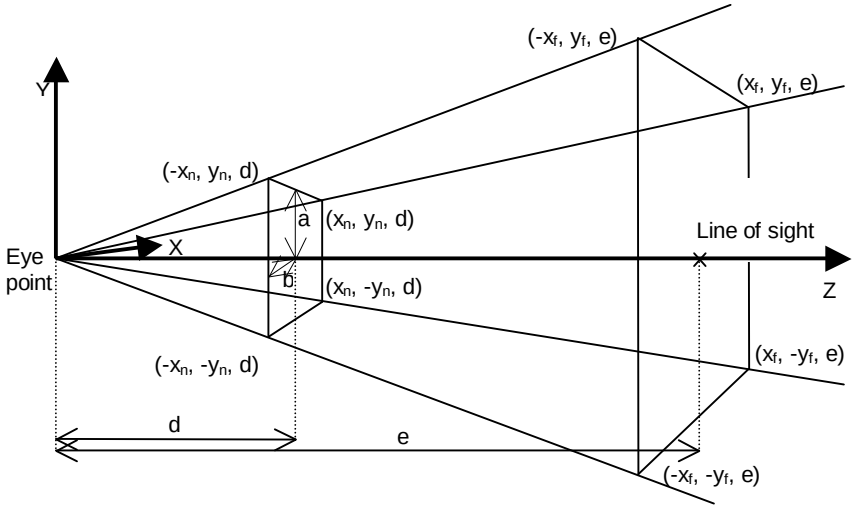


Figure 2.5: Illustration of the view frustum constrained by six planes. The view plane is incident with the near clipping plane. $2a$ is the height and $2b$ is the width of the view plane. d is the distance from the eye point (residing in the origin since we are working in eye space) to the near clipping plane, while e is the distance to the far clipping plane. The n subscript denotes $n$ear, while the f subscript denotes $f$ar. The figure resembles figure 1.9 of [135].

The six planes bounding the view frustum are referred to as the top, bottom, left, right, near, and far planes. In practice it is normal to place the near clipping plane just in front of the camera so that you can not see

through objects. Another practical issue is what to do when the visible area exceeds the far clipping plane, which often is the case when simulating outdoor environments. Simply cutting when the far clipping plane is reached can give unwanted effects, for example objects suddenly disappearing or emerging in the horizon (this is called popping). Fog is a simple atmospheric effect that can gradually hide distant objects giving a smoother transition [2].

To define the view frustum we identify each plane individually. A plane can be described by a normal $\boldsymbol{n} = (\alpha, \beta, \gamma)$ and a point in space $\boldsymbol{x} = (x_0, y_0, z_0)$:

$$\alpha x + \beta y + \gamma z + \delta = 0 \tag{2.22}$$

where $\delta = -(\boldsymbol{n} \cdot \boldsymbol{x})$. The normal can be determined as the cross product of two linearly independent vectors, which are referred to as the basis of the plane.

The left, right, top and bottom planes of the view frustum all have the eye point in common. From each corner of the view plane a common basis vector for two of those four planes can be found by subtraction of the eye point.

The view plane is perpendicular to the line of sight and follows the z-axis in the eye space coordinate system. Therefore the line of sight is a normal for both the near and the far clipping planes, and a point on the plane is given by the distances d and e as shown in figure 2.5. Using the line of sight and the basis vectors defined by the corners of the view plane we can define an equation for each of the six planes bounding our view frustum.

Consider a point in eye space (x_e, y_e, z_e), for the four points in the near clipping plane $z_e = d$ and for the far clipping plane $z_e = e$. Since the eye point resides in the origin the four basis vectors given by the corners of the view plane (which in this case is incident with the near clipping plane) are given as:

$$\begin{aligned}
\boldsymbol{b_0} &= \begin{pmatrix} -x_n \\ -y_n \\ d \end{pmatrix} = \begin{pmatrix} -b \\ -a \\ d \end{pmatrix} \\
\boldsymbol{b_1} &= \begin{pmatrix} -x_n \\ y_n \\ d \end{pmatrix} = \begin{pmatrix} -b \\ a \\ d \end{pmatrix} \\
\boldsymbol{b_2} &= \begin{pmatrix} x_n \\ y_n \\ d \end{pmatrix} = \begin{pmatrix} b \\ a \\ d \end{pmatrix}
\end{aligned}$$

$$\boldsymbol{b_3} = \begin{pmatrix} x_n \\ -y_n \\ d \end{pmatrix} = \begin{pmatrix} b \\ -a \\ d \end{pmatrix}$$

The cross product of each pair of adjacent basis vectors defines the normal of a side plane, and since all the side planes have the eye point in common $\delta = 0$ in eqn. 2.22. This results in the equation for each of the six planes shown in table 2.3.

Plane	**Equation**
Near clipping plane	$z_e = d$
Far clipping plane	$z_e = e$
Right plane	$(\boldsymbol{b_0} \times \boldsymbol{b_1}) \cdot (x_e, y_e, z_e) = 0 \quad \Leftrightarrow \quad dx_e + bz_e = 0$
Top plane	$(\boldsymbol{b_1} \times \boldsymbol{b_2}) \cdot (x_e, y_e, z_e) = 0 \quad \Leftrightarrow \quad dy_e - az_e = 0$
Left plane	$(\boldsymbol{b_2} \times \boldsymbol{b_3}) \cdot (x_e, y_e, z_e) = 0 \quad \Leftrightarrow \quad dx_e - bz_e = 0$
Bottom plane	$(\boldsymbol{b_3} \times \boldsymbol{b_0}) \cdot (x_e, y_e, z_e) = 0 \quad \Leftrightarrow \quad dy_e + az_e = 0$

Table 2.3: Equations defining the bounding planes of a view frustum in eye space. Note that left and right are left and right as seen from the eye point in figure 2.5, and that the coordinate system of figure 2.5 is right-handed

Having defined the view frustum which contains the visible part of a scene, the task is now to limit, according to the frustum, the number of objects that we must process. Suppose we enclose each object in the scene by a *bounding volume* of much simpler geometrical shape than the object itself. Then the point is that it should be much easier to test the bounding volume for containment within the view frustum than to test the object itself. Examples of bounding volumes are spheres and axis aligned bounding boxes (AABBs). The bounding sphere is indifferent to rotation, which is often an advantage, but unfortunately they grow unnecessarily large and do not always fit well the object that it bounds, for example if the object is long and thin. The axis aligned bounding box (AABB) is easy to handle because its faces are axis aligned, and it resizes in three directions not uniformly in all directions as the sphere. Therefore AABBs often fit the bounded object better. A disadvantage is that its size must be adjusted after rotation. We propose an idea in section 11.3 which use bounding spheres and we use AABBs for ray tracing, see section 4.2.

Using a bounding volume for each object in the scene we can test whether an object is (a) completely inside, (b) partly inside, or (c) completely outside the view frustum. In the second case (b) the object should be clipped against the view frustum. There is a standard algorithm for clipping which is described in many computer graphics textbooks (eg. [134, 38, 135, 3]). clipping is a part of the rendering pipeline that will be described in chapter 5, and is often implemented in hardware. While a good understanding of the view frustum is useful when operations are carried out in different coordinate

systems (world space, eye space, clip space, etc.), we feel that clipping is a procedure so standardized that there is no reason to replicate it here.

Usually the polygons left by the clipping algorithm for further processing are still plenty. To further bring down the number of triangles we can remove all back facing surfaces. This process is referred to as *back face culling*.

Back face culling consists of a simple geometrical test. If we describe the line of sight by the directional vector $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$ and the face normal of a polygon as $\boldsymbol{n} = (n_x, n_y, n_z)$, then the dot product between the two determines whether the polygon is facing towards the eye point or away from it. Line of sight is sometimes described as the direction from the point on the surface towards the eye point, in that case visibility is determined as $\boldsymbol{\omega} \cdot \boldsymbol{n} > 0$. In eye space the test simplifies to $n_z > 0$.

When drawing triangles the face normal is often determined as the cross product of the directional vectors given by the first two edges drawn. The result is that, by convention, polygons drawn in a counter clockwise manner (from the point of view of the eye point) will have a normal pointing towards the eye point and will, hence, be front facing.

The last and most tricky part is to remove all objects or parts of objects covered by other objects. This is referred to as *occlusion culling*. Efficient algorithms for occlusion culling are complex. The problem is that it is difficult to determine which objects that are likely to be occluders [134]. The subject of occlusion culling will not be addressed in detail in this project, a few references on the subject are [2, 134, 50].

Another way to speed up the process of choosing visible objects for rendering is scene graphs. Scene graphs are often useful in computer graphics and they are usually constructed using some kind of spatial data structure to split up the scene in a sensible way. Scene graphs and spatial data structures will be described briefly in the following section.

2.5 Scene Graphs and Spatial Data Structures

Both realistic image synthesis and real-time techniques run into the problem of "unrealistic computation times" ([135]) if they choose a naïve brute force rendering technique. The problem is usually the huge amount of triangles that must be processed in order to display images in the desired quality. One way to rule out large parts of a scene quickly is by spatial subdivision of the scene.

Spatial data structures encompass data structures such as octrees (and quadtrees), kd-trees, BSP (Binary Space Partitioning) trees, bounding volume hierarchies, Voronoi diagrams, etc. which are useful for spatial subdivision of a two- or three-dimensional scene.

The kd-tree is an important part of the rendering technique called photon mapping, see [60, Chap. 6]. Recent articles have, however, tested other

spatial data structures with photon mapping. Günther et al. have had success with a uniform grid of voxels [44].

A kd-tree is, in fact, short for a k-dimensional tree. The special case of 3d-trees, which are most commonly used in computer graphics, split (or subdivide) the scene using planes perpendicular to the axes of the world coordinate system. Each split results in two subsets of photons. The concept is illustrated in figure 2.6.

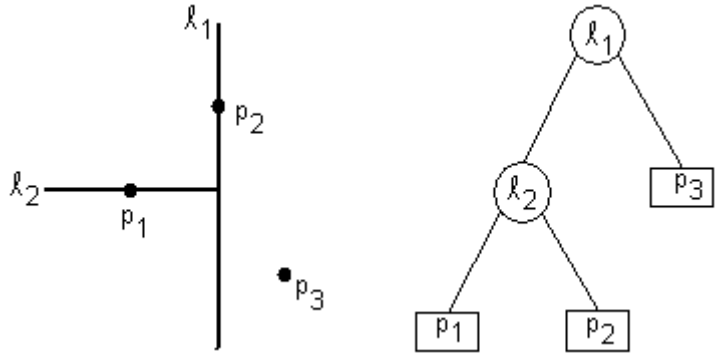


Figure 2.6: A simple example of a kd-tree. p_1, p_2, and p_3 represent photons stored in a photon map. ℓ_1 and ℓ_2 represent splitting planes.

An algorithm for the construction of a 3d-tree could be as follows, where d denotes the recursion depth and the $d = 0$ at the root node:

1. If $d \bmod 3 = 0$ split with a plane perpendicular to the x-axis, if $d \bmod 3 = 1$ split with a plane perpendicular to the y-axis, or if $d \bmod 3 = 2$ split with a plane perpendicular to the z-plane.

2. Store the splitting plane in the node.

3. Let the left child receive the subset of points behind or on the splitting plane.

4. Let the right child receive the subset of points in front of the splitting plane.

5. For each child; return to step 1 unless the maximum recursion depth has been reached.

Proper placement of the splitting planes is important. In the original algorithm the splitting planes should be placed at the median of each point set. The kd-tree for photon mapping (as described in [60]) also find medians.

If the kd-tree is properly balanced, the resulting data structure locates a leaf node at complexity $O(\log N)$, where N is the number of leaf nodes in the tree. For specific details on the use of kd-trees for photon mapping we

refer to [60], and for further details on the geometrical aspects of kd-trees and many other useful spatial data structures [25] is a good reference.

In addition to the use of kd-trees for photon mapping, spatial data structures have countless applicabilities in computer graphics. A few examples are: A quad-tree storing objects computed on the fly and used as a fast scene graph for real-time environments [35], octrees, BSP trees, and bounding volume hierarchies for reduction of ray-triangle intersections in ray tracing [135] (ray-triangle intersections are described in section 4.2), and a quad-tree for shaft occlusion culling and shadow ray acceleration [117]. In general spatial data structures are used to achieve speed-ups whenever the objects in a scene have to be searched in one way or another.

A scene graph, as presented in [35], is a higher level tree structure storing more than just geometry. Light sources can be stored in a scene graph, textures, and transformation matrices as well. When rendering, the tree is traversed in a depth-first order and textures, tranformations, and light sources can be associated with an internal node so that it is only applied to the subtree of that particular node [2]. When a dynamic environment grows in size scene graphs are indispensable, both to keep track of objects in the scene and to speed up rendering. Scene graphs and spatial data structures are some of the subjects that we have chosen not to treat thoroughly in this project. Nevertheless they are quite useful, and they should be studied in the future if our test scenes grow larger.

The drawback of spatial data structures for real-time rendering is that they are often quite costly to construct and re-construct when things change dynamically. This has the result that they must be either very simple having a reasonable size or they must be pre-computed.

For a truly dynamic scene it is difficult to pre-compute all possible versions of spatial data structure, not to say impossible if we want, for example, a BSP tree with a single triangle in each leaf node. Tiles are therefore often used in scenes for real-time rendering and a graph is established and pre-computed between those, or simple data structures (such as the quad-tree mentioned before) are computed on the fly.

Unfortunately we have not been able to spare time during this project for a thorough investigation of spatial data structures. Most of the scenes that we have tested have not been sufficiently complex to draw advantage of BSP trees or the like. However, we have developed a simple spatial data structure based on solid angles, and used a few other optimization schemes for ray tracing, those will be described in part III.

Chapter 3

The Mathematical Model of Illumination

Socrates: *Though vision may be in the eyes and its possessor may try to use it, and though color be present, yet without the presence of a third thing specifically adapted to this purpose, you are aware that vision will see nothing and the colors will remain invisible.*

Glaucon: *What is this [third] thing of which you speak? he said.*

Socrates: *The thing, I said, that you call* light.

Glaucon: *You say truly, he replied.*

Plato (427-347 BC.): *The Republic* 507e

One purpose of this project is to visualize a three-dimensional digital model as photorealistic images on a computer screen, and to make those images appear on the screen at a frequency that allows interactive animation of the pictured elements.

These apparently harmless objectives are, in fact, quite contradictive. At least they are contradictive on current computer hardware using the current visualization techniques. Why so? Since a perfect synthetic representation of the real world would consist of infinite complexities. Luckily we need not compute an exact copy of the real world, we can merely focus on the human visual interpretation of the world.

What we need is a model that can simulate vision (or photography) as it takes place in real life. What we see is light, hence, what the mathematical model must capture is the interaction of light with matter, or the *illumination* of a scene. Such a mathematical model will be referred to as an illumination model.

Simple as it may seem, it is not at all simple to model or even give an exact explanation of light. Through times many attempts have been made on the nature of light (see appendix B). Today *quantum mechanics* explain light on the particle level while light in general is considered to be *electromagnetic radiation* in the infrared, visible, and ultraviolet spectrum. The spectrum of electromagnetic waves is illustrated in figure 3.1.

In order to construct an illumination model we must give a mathematical description of light. First the propagation of light through different media must be modeled. The scientific field of *optics* includes a mathematical model for the description of light propagation. Section 3.1 presents some fundamental postulates concerning light propagation on which we can build our illumination model.

Next we must introduce a terminology in our model by which we can specify how light is measured by the eyes and other optical detectors. *Radiometry* offers mathematical definitions of the necessary physical terms and is described in section 3.2.

The strict physical measures of light do not necessarily fit the visual response of the human eye exposed to light. *Photometry* is closely related to radiometry except for the fact that it includes the visual response of a standard observer in the quantification of light measurements. Photometry is shortly introduced in section 3.3.

Since illumination includes the interaction of light with matter we must include in our model how light is reflected off, refracted through, and absorbed by different materials. These processes, and the relation between them, are referred to as *light scattering* and are modeled in the field of *heat transfer*. Light scattering will be the topic of section 3.4.

Consecutively the mathematical description of light and how it scatters will be gathered in an integral equation that we can use for rendering. The rendering equation is described in section 3.5.

To explore the model a little deeper section 3.6 will go into the recursive nature of the rendering equation. It is shown how the illumination of a scene can be described as recursive steps of light propagation and light scattering.

Section 3.7 describes some of the different illumination models that can be derived from the rendering equation, and, finally, section 3.8 will present some of the tools that can be used when we need to solve recursive integral equations such as the rendering equation.

All in all the purpose of this chapter is to put forth a mathematical model describing how light illuminates its surroundings.

3.1 Optical Radiation

In order to describe how light propagates, we must introduce a number of physical terms from the broad literature of optics, radiometry, and thermal radiation.

Radiation is energy propagated in the form of electromagnetic waves or particles (photons). The range of radiation which can be reflected, imaged, or dispersed by optical components, such as mirrors, lenses, or prisms, is referred to as *optical radiation*[1]. Hence, light is composed of optical radiation.

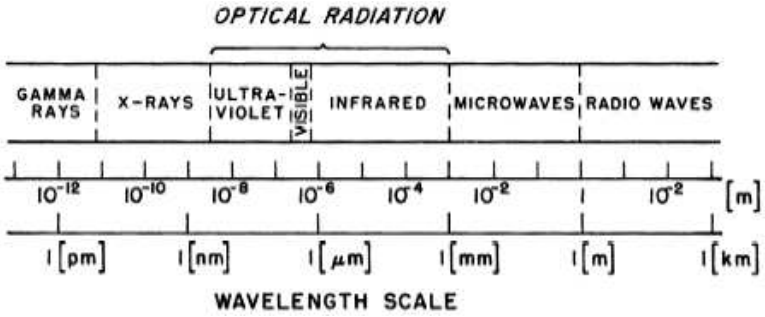


Figure 3.1: Spectrum of electromagnetic waves. Identical to fig. 1.1 of [89]. (Courtesy of F. E. Nicodemus et al.)

Optics is a theory or model for the description of physical phenomena involving the generation, propagation, and detection of light. Like most other physical models optics is an abstraction or idealization that approximates 'real life'. The historical development (see app. B) of optical models has resulted in different levels of sophistication, "each with its own region of useful validity" [89].

As seen in figure 3.1 the visible light region of the electromagnetic spectrum extends from wavelengths of approximately 380 to 780 nanometers.[2]

[1] The definition of optical radiation is adopted from [89].

[2] Note that only approximate boundaries exist between the wavelength regions of the electromagnetic spectrum.

When light waves propagate through and around objects whose dimensions are much greater than the wavelength (this is, for example, a reasonable assumption in virtual realities and 3D environments for games and animation), the wave nature of light is not readily discerned and for that reason light can adequately be described by rays obeying a set of geometrical rules [116]. This model of light is referred to as *ray optics* or *geometrical optics*. Approximating light waves with rays is the simplest approach to optics. However, it accounts very well for the way in which optical radiation is propagated from the most common light sources [89].

The shortcomings of ray optics are revealed by optical phenomena such as diffraction and interference found in focal regions where rays sharply converge. To treat such phenomena light must be described in terms of *wave optics* or *electromagnetic optics*. Moreover if we want to treat the interaction of light with matter in microscopic detail it is necessary to recognize that optical radiation is being propagated in discrete "packets" or photons, whose large numbers produce average energy distributions in time and space corresponding to electromagnetic waves [89]. Such a model relies on quantum theory and is, hence, referred to as *quantum optics*. Figure 3.2 describes the increasing levels of sophistication in optical models. In computer graphics the simplifying assumption of ray optics is reasonable and therefore this thesis will exclusively make use of ray optics in order to solve problems of light wave propagation in virtual worlds.

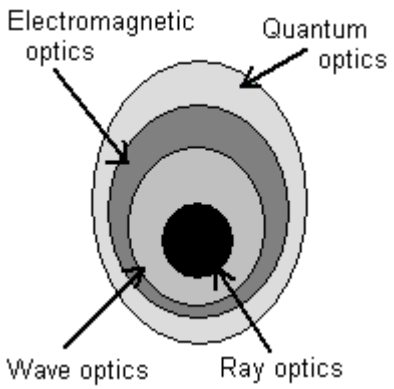


Figure 3.2: The theory of quantum optics provides an explanation of virtually all known optical phenomena. Electromagnetic optics provides the most complete treatment of light within the confines of rigorous electromagnetic theory. Wave optics is a scalar approximation of electromagnetic optics. Ray optics is the limit of wave optics when the wavelength is infinitesimally small. The figure is adopted from [116].

Ray optics are based on four postulates that are provided here without proof (though they follow naturally from the more sophisticated models). From those we can "determine the rules governing the propagation of light rays, their reflection and refraction at the boundary of different media, and their transmission through various optical components" [116].

The four postulates are given in [116] and are replicated below in short:

1. Light travels in the form of rays. The rays are emitted from light sources and can be observed when they reach an optical detector.

2. An optical medium is characterized by a *refractive index* $\eta = \frac{c_0}{c} \geq 1$, where c is the speed of light in a medium and c_0 is the speed of light in vacuum.

3. In an inhomogeneous medium the refractive index $\eta(\boldsymbol{r})$ is a function of the position $\boldsymbol{r} = (x, y, z)$.

4. **Fermat's Principle**. Optical rays traveling between two points follow a path such that the time of travel is an extremum relative to the neighboring paths. The extremum is most often a minimum in which case the ray follows the path of "least time" as Fermat originally expressed it.

To illustrate shortly the consequences of the postulates we may consider the propagation of light in a homogenous medium. The refractive index is constant throughout a homogenous medium, and it follows then from the second postulate that so is the speed of light. The path of least time, which is required by Fermat's principle, is then also the path of minimum distance. This property of light was first discovered by Hero of Alexandria (see appendix B) and is referred to as *Hero's principle*. A consequence of Hero's principle, which follows from the postulates, is therefore that *light rays travel in straight lines in a homogenous medium*. Throughout this report, unless stated otherwise, it will be assumed that all media are homogenous.

Furthermore we can derive from the postulates how light reflects off and refracts through a perfectly specular surface marking the boundary between two homogenous media.

Reflection

Consider two locations, A and C, in a homogenous medium, see figure 3.3. If a ray of light is to travel from A to C by reflection off a perfectly specular mirror surface $\mathcal{S}$, it will according to Hero's principle follow the path of minimum distance. That is, we must choose a point B on the mirror surface such that the distance $\overline{AB} + \overline{BC}$ is minimized. First to follow the shortest path the light must travel in the plane of incidence, which is spanned by the two vectors $C - A$ and $B - A$. Figure 3.3 pictures the plane of incidence.

Let C' be a mirror image of C in the tangent plane T_p to the specular mirror surface S, then $\overline{BC} = \overline{BC'}$. Now $\overline{AB} + \overline{BC'}$ must be minimum, which is indeed the case when a connection of the points A, B, and C' is a straight line. If ABC' is a straight line, we have that $\angle(\overline{AB}, T_p) = \angle(T_p, \overline{BC'})$, and because C' is a mirror image of C it is evident that $\angle(T_p, \overline{BC'}) = \angle(\overline{BC}, T_p)$:

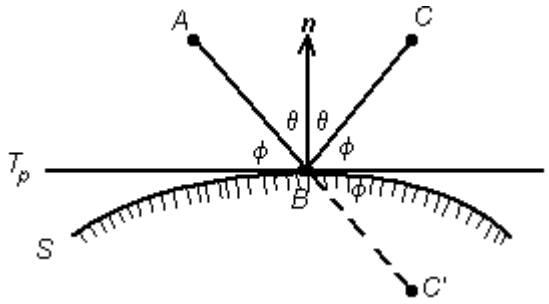


Figure 3.3: Light traveling in a homogenous medium from a point A reflecting off a perfectly specular mirror surface S at the point B ending in the point C. T_p is the tangent plane to S, and C' is the mirror image of C in T_p.

$$\angle(\overline{AB}, T_p) = \angle(T_p, \overline{BC'}) = \angle(\overline{BC}, T_p) = \phi$$

Since $\angle(\overline{AB}, T_p) = \angle(\overline{BC}, T_p)$, and since the normal $\boldsymbol{n}$ at the surface point B by definition is perpendicular to T_p, it follows that:

$$\angle(\overline{AB}, \boldsymbol{n}) = \angle(\boldsymbol{n}, \overline{BC}) = \theta \tag{3.1}$$

which states that the angle of reflection equals the angle of incidence (cf. fig. 3.3).

Let $\boldsymbol{\omega}' = \frac{A-B}{||A-B||}$ denote the direction from which the incident radiance is arriving at the mirror surface S, then, according to (3.1), the direction of the reflected ray $\boldsymbol{\omega}_s$ can be found as:

$$\boldsymbol{\omega}_s = 2\cos\theta\,\boldsymbol{n} - \boldsymbol{\omega}' = 2(\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}' \tag{3.2}$$

where θ is the angle of incidence and $\boldsymbol{n}$ is the *unit* normal at the point of incidence. (3.2) is one of many ways to express the *law of reflection*. Another enunciation is as given in [116, p. 5]:

> The reflected ray lies in the plane of incidence;
> the angle of reflection equals the angle of incidence.

□

Refraction

Again the fourth postulate (Fermat's principle) gives means by which we can establish how light refracts when it passes from one homogenous medium to another. The difference between the two media is stated as two different indices of refraction, η_1 and η_2 (one for each media). In order to find the path that light takes from a point A in the medium of refraction index η_1

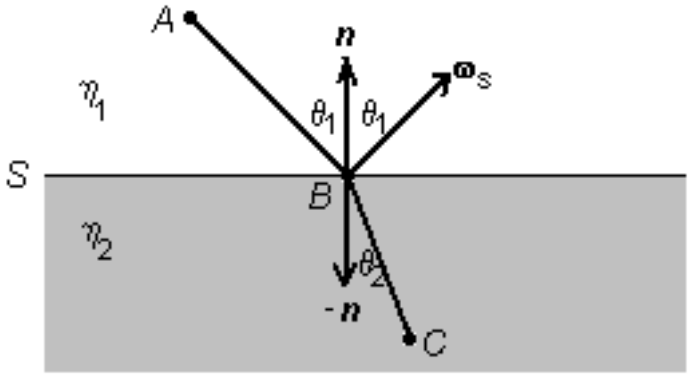


Figure 3.4: The path of light when it travels from one homogenous medium to another.

to a point B in the medium of refraction index η_2, we must find the path of least time. Again we need only consider the plane of incidence as pictured in figure 3.4.

Let B be the point of incidence at the surface S, which marks the boundary between the two different media, and let θ_1 be the angle of incidence such that:

$$\cos\theta_1 = \boldsymbol{\omega}' \cdot \boldsymbol{n}$$

where $\boldsymbol{\omega}' = \frac{A-B}{||A-B||}$ is the direction of incident light, and $\boldsymbol{n}$ is the unit normal at the point of incidence. Now, we denote the angle of the refracted light θ_2 such that:

$$\cos\theta_2 = \boldsymbol{\omega}_r \cdot \boldsymbol{n}$$

where $\boldsymbol{\omega}_r = \frac{C-B}{||C-B||}$ is the direction of the refracted light.

The time it takes for light to travel a distance d equals $d/c = \eta d/c_0$. The travel time is thus proportional to the *optical path length*, which is defined as ηd. To find the path of least time we can, therefore, minimize the optical path length instead of the travel time. Meaning that what we seek to minimize is $\eta_1\overline{AB} + \eta_2\overline{BC}$.

Suppose we let the normal $\boldsymbol{n}$ define a v-axis in the plane of incidence, then $B = (u, 0)$ in (u, v)-coordinates. Furthermore let $A = (u_1, v_1)$ and $C = (u_2, v_2)$, then $\overline{AB} = \sqrt{(u-u_1)^2 + v_1^2}$ and $\overline{BC} = \sqrt{(u_2-u)^2 + v_2^2}$. The derivative of the optical path length is then given as:

$$\frac{d(\eta d)}{dx} = \eta_1 \frac{u - u_1}{\sqrt{(u-u_1)^2 + v_1^2}} + \eta_2 \frac{-u_2 + u}{\sqrt{(u_2-u)^2 + v_2^2}}$$

Setting the derivative equal to zero we will discover an extremum (which will most probably be a minimum) as is required by Fermat's principle. This yields the following:

$$\eta_1 \frac{u - u_1}{\sqrt{(u - u_1)^2 + v_1^2}} = \eta_2 \frac{u_2 - u}{\sqrt{(u_2 - u)^2 + v_2^2}}$$

or put differently:

$$\eta_1 \frac{d_1}{\overline{AB}} = \eta_2 \frac{d_2}{\overline{BC}}$$

where d_1 and d_2 are the projections on the tangent plane of $\overline{AB}$ and $\overline{BC}$ respectively. Now, *Snell's law* follows by the trigonometry of a right triangle:

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2 \tag{3.3}$$

Snell's law gives means by which we can find a formula for the calculation of the direction of refraction $\boldsymbol{\omega}_r$. Suppose $\boldsymbol{t}$ is a unit tangent vector to the perfectly specular surface S at the point B in the plane of incidence, and that $\boldsymbol{n}$ is the unit normal pointing into the medium of refraction index η_1, then the direction of the refracted light is:

$$\boldsymbol{\omega}_r = -\boldsymbol{n}\cos\theta_2 + \boldsymbol{t}\sin\theta_2 \tag{3.4}$$

Suppose the normal $\boldsymbol{n}$ and the direction towards the incident light $\boldsymbol{\omega}'$ are given. Then the component of $\boldsymbol{\omega}'$ that is perpendicular to the normal, is given as:

$$\boldsymbol{\omega}'_\perp = (\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}'$$

while the length of $\boldsymbol{\omega}'_\perp$ simply is $\sin\theta_1$. An expression for the unit tangent vector $\boldsymbol{t}$ then follows:

$$\boldsymbol{t} = \frac{\boldsymbol{\omega}'_\perp}{||\boldsymbol{\omega}'_\perp||} = \frac{(\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}'}{\sin\theta_1}$$

To find $\cos\theta_2$ of (3.4) we use the property that the angle of incidence always lies in the interval $[0, \frac{\pi}{2}]$, and get that:

$$\cos\theta_2 = \sqrt{1 - \sin^2\theta_2}$$

(3.4) then has the following form:

$$\boldsymbol{\omega}_r = -\boldsymbol{n}\sqrt{1 - \sin^2\theta_2} + \frac{\sin\theta_2}{\sin\theta_1}((\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}')$$

and by Snell's law and a few rearrangements, we retrieve the *law of refraction* as it is expressed in many text books (eg. [38, 60, 30]):

$$\begin{aligned}
\boldsymbol{\omega}_r &= \frac{\sin\theta_2}{\sin\theta_1}((\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}') - \boldsymbol{n}\sqrt{1 - \sin^2\theta_2} \\
&= \frac{\eta_1}{\eta_2}((\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}') - \boldsymbol{n}\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\theta_1} \\
&= \frac{\eta_1}{\eta_2}((\boldsymbol{\omega}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{\omega}') - \boldsymbol{n}\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (\boldsymbol{\omega}' \cdot \boldsymbol{n})^2)} \qquad (3.5)
\end{aligned}$$

The more textual enunciation of the law of refraction is given in [116, p. 6] as:

> The refracted ray lies in the plane of incidence; the angle of refraction θ_2 is related to the angle of incidence θ_1 by Snell's law,
>
> $$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

□

Based on the four postulates of ray optics, it has now been established how light propagates in homogenous media. The three simple rules are:

- Propagation in straight lines
- The law of reflection
- The law of refraction

The laws of reflection and refraction concerns only perfectly specular material, nevertheless, they are important since the reflection and refraction of glossy materials (which are materials with properties that lie in-between perfectly specular and perfectly diffuse) in general are centered around the perfect directions.

Having established the basic properties of light propagation on which to build our illumination model, we can proceed to the radiative properties of light, which will allow us a description of the light that can be measured at optical detectors such as the eye.

3.2 Radiometry

While optics concern the generation, propagation, and detection of light, *radiometry* is the science of electromagnetic radiation measurement.

The basic quantity of radiometry is *radiance*, L, which most closely represents the colors that we see [60]. Therefore radiance is a very important

quantity in computer graphics, and radiance is what we must calculate and display to an observer in order to visualize a 3D model.

Technically speaking radiance is the amount of radiant energy arriving at or departing from an infinitesimal area per unit time with respect to a given direction. Meaning that radiance can be used to describe the intensity of light at a given point in space.

Radiance is a third derivative of *radiant energy*, Q. To give a better understanding of radiance the different derivatives of radiant energy will be described in the following.

Radiant Flux

First we would like to consider the flow of energy rather than isolated amounts. This is obvious since the eyes percept light waves continuously rather than on timely intervals. Radiant energy per unit time is called *radiant flux*, $\mathbf{\Phi}$:

$$\mathbf{\Phi} = \frac{dQ}{dt} \tag{3.6}$$

When looking at an object the eye registers the radiant flux departing the object in the direction of the eye. To describe a directional volume through which the energy can flow, we need to introduce the concept of a *solid angle*.

Solid Angle

In 2-dimensional space the more familiar *plane angle*, θ, is formed at the point $\boldsymbol{O}_\theta$ where two straight lines meet, see figure 3.5a. It is defined as the locus of all directions that may be occupied by either line as it is rotated about the vertex to bring it into directional coincidence with the other line. Drawing a unit circle centered at $\boldsymbol{O}_\theta$ the length of the circular arc intercepted by θ is a measure of the plane angle.

The solid angle is formed at a point $\boldsymbol{O}_\omega$ in 3-dimensional space. Consider a simply-connected curve (not passing through $\boldsymbol{O}_\omega$) perhaps forming the contour edge of a convex object, see figure 3.5b. The conical surface, containing all possible straight lines that extend from $\boldsymbol{O}_\omega$ to a point on the curve, forms the delimiter or bounding cone of the solid angle exactly as the straight lines in figure 3.5a forms the delimiter of the plane angle. The solid angle ω is then defined as the locus of all directions lying within the bounding cone. Drawing a unit sphere centered at $\boldsymbol{O}_\omega$ the spherical-surface area A_s intercepted by ω is a measure of the solid angle.

The magnitude of plane angles is radians [rad] or degrees [°]. The relationship between these two entities is:

$$1^\circ = \frac{180}{\pi}\,[rad]$$

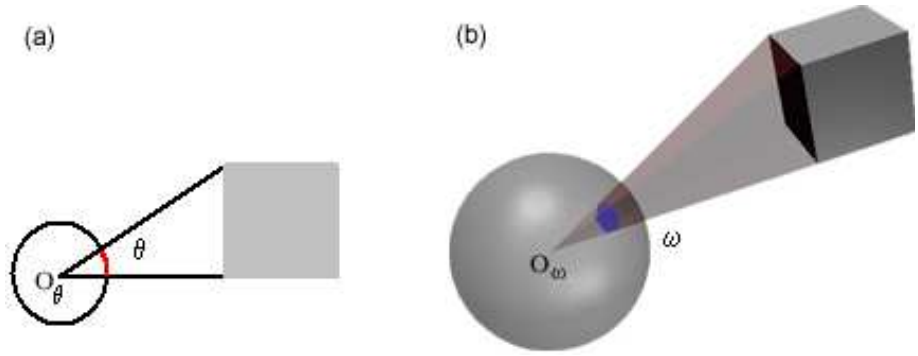


Figure 3.5: (a) A plane angle θ formed at the point $\boldsymbol{O}_\theta$. The length of the red circular arc, which is a part of the unit circle, is a measure of θ. (b) A solid angle ω formed at the point $\boldsymbol{O}_\omega$. The area of the blue patch on the unit sphere is a measure of ω.

The magnitude of a solid angle ω is steradians $[sr]$, which is the ratio of the intercepted spherical-surface area A_s to the square radius of the sphere r^2. As previously noted this magnitude is identical to the intercepted area on the unit sphere (where $r = 1$):

$$\omega = \frac{A_s}{r^2}\,[sr] \tag{3.7}$$

Meaning that the entire unit sphere contains $4\pi\,[sr]$.

Since computer graphics use ray optics, and therefore rays, to describe the propagation of light, the solid angle is mostly described by the direction around which it is defined. The directional element of radiance is defined by a differential solid angle subtended by a differential surface area, and therefore the bounding cone of the solid angle will approach a ray as the surface area gets infinitesimally small, and the ray approximation is thus reasonable. In this context it should be noted that a vector describing the direction of a is ray denoted by a bold omega ($\boldsymbol{\omega}$), while the solid angle itself is denoted ω and likewise the differential solid angle is denoted $d\omega$.

It is often convenient to express a solid angle in spherical coordinates. It eases the description of a differential (or elemental) solid angle $d\omega$, see figure 3.6, and the spherical coordinates converts to a unit direction vector as follows:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix} \tag{3.8}$$

Drawing a sphere of radius r centered at the point $\boldsymbol{O}$ where the solid angle is formed, the differential spherical-surface area dA_s is defined by the two angles θ and ϕ increased by the infinitesimal elements $d\theta$ and $d\phi$ respectively (as shown in fig. 3.6). The angular rotation given by θ moves on a great

circle and therefore the length of the circular (latitude) arc $d\theta$ is $rd\theta$. The angular rotation around the z-axis specified by ϕ does, however, not move on a great circle. Rather ϕ follows the arc of radius $r \sin\theta$ and the length of the circular (longitude) arc $d\phi$ is therefore $r \sin\theta \, d\phi$. The differential spherical-surface area is then given as $dA_s = r^2 d\theta \, \sin\theta \, d\phi$ resulting in the following definition of the differential solid angle:

$$d\omega = \frac{dA_s}{r^2} = \sin\theta \, d\theta \, d\phi \tag{3.9}$$

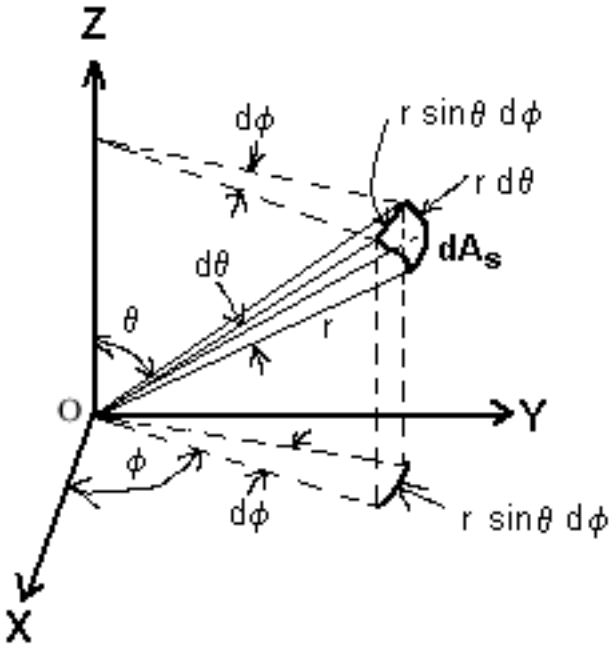


Figure 3.6: A differential (or elemental) solid angle. $d\omega = dA_s/r^2 = \sin\theta \, d\theta \, d\phi$

If we, for example, need to find the total radiance rather than directional radiance incident on a differential surface area, the general expression for any solid angle given in spherical coordinates is found as an integral over (3.9):

$$\omega = \int_{\phi} \int_{\theta} \sin\theta \, d\theta \, d\phi \tag{3.10}$$

Radiant Intensity

The differential solid angle $d\omega$ gives means by which we can describe an infinitesimal directional volume. An element of radiant flux per element of solid angle then describes a directional flow of radiation (or light). This is a second derivative of radiant energy which is called *radiant intensity*, I:

$$\frac{d^2Q}{dt\,d\omega} = \frac{d\Phi}{d\omega} = I(\boldsymbol{\omega}) \tag{3.11}$$

Radiant intensity is useful if we, for example, need to find the total flux incident on a surface from a certain direction. This is, however, rarely the case. Rather we would like to consider how the flux changes from point to point on a surface.

Radiant Flux Area Density

To consider the element of flux through an element of surface area, $d\Phi/dA$, we need another second derivative of radiant energy which is referred to as *radiant flux area density*. To distinguish between incident and exitant[3] radiation, radiant flux area density is separated into *radiant exitance*, M (also called radiosity, B), which is flux departing from the surface area, and *irradiance*, E, which is flux arriving at the surface area:

$$\frac{d^2Q}{dt\,dA} = \frac{d\Phi}{dA} = E(\boldsymbol{x}) \tag{3.12}$$

where $\boldsymbol{x}$ is a surface location.

Radiant flux area density, though useful in many cases, only considers the changes in flux from point to point. In order to describe the flux both in terms of direction and location, we must consider how a surface area is visible from a certain direction so before we can give the definition of radiance, we must look into the projection of a surface area on a given direction.

Projected Surface Area

Consider a differential surface area dA as depicted in figure 3.7 and recall that radiance describes the radiant flux arriving at or leaving a surface location in a certain direction.

The directional element is described by a differential solid angle $d\omega$ centered around a direction given in spherical coordinates (θ, ϕ), and it defines a spherical-surface area dA_s on the unit sphere as noted previously. The amount of energy per unit time that arrives at or leaves dA depends on the solid angle that dA occupies when viewed from dA_s. This quantity is the *projected surface area*, $dA_p = \cos\theta\, dA$, which is the projected area of dA normal to the (θ, ϕ) direction, see also figure 3.8.

Radiance

Being both directional and dependant on the surface location $\boldsymbol{x}$, radiance, L, has the following mathematical definition:

[3] "Exitent" was coined as an antonym of "incident" by J. C. Richmond in 1972, today the spelling has slid to "exit*a*nt".

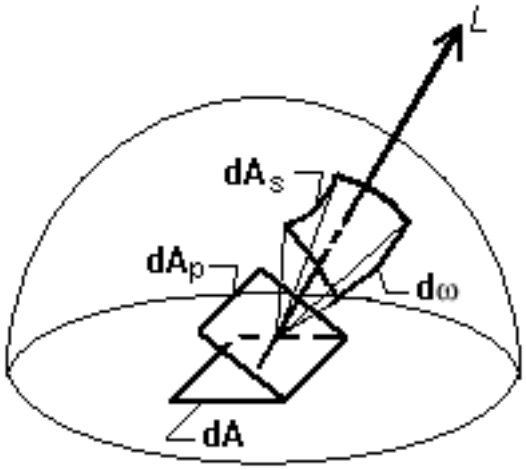


Figure 3.7: Pictorial description of directional radiation properties. dA is a differential surface area, dA_p is the projected area of dA normal to the direction of the differential solid angle $d\omega$.

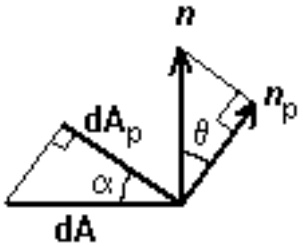


Figure 3.8: Two-dimensional view of a differential projected surface area, $dA_p = \cos\alpha\, dA = \cos(\frac{\pi}{2} - (\frac{\pi}{2} - \theta))\, dA = \cos\theta\, dA$.

$$\frac{d^3Q}{dt\, d\omega\, dA_p} = \frac{1}{\cos\theta}\frac{d^2\mathbf{\Phi}}{d\omega\, dA} = L(\boldsymbol{x}, \boldsymbol{\omega}) \tag{3.13}$$

Previously it has been mentioned a few times that radiance defines the amount of energy per unit time that arrives at dA from a certain direction, *or* leaves dA in a certain direction. Meaning that in a pictorial description we can turn the direction, around which the solid angle is defined, away from the surface (as seen in fig. 3.7) indicating that the radiance is exitant, or we could turn it towards the surface indicating that the radiance is incident. Exitant (or outgoing) radiance is denoted L_o while incident radiance is denoted L_i.

Radiant flux can only arrive to or leave the surface area dA from a direction (θ, ϕ) within the hemisphere above dA, therefore if we need to consider the total incident or exitant radiance at a certain surface location, we let $\int_\Omega d\omega$ denote integration over the hemispherical solid angle being special case of (3.10). Hence, for any function $f(\theta, \mathbf{\Phi})$:

$$\int_{\phi=0}^{2\pi}\int_{\theta=0}^{\pi/2} f(\theta,\phi)\sin\theta\, d\theta\, d\phi = \int_{\Omega} f(\theta,\phi)d\omega$$

which, for example, indicates that we can find the irradiance at a specific surface location as:

$$E(\boldsymbol{x}) = \int_{\Omega} L_i(\boldsymbol{x},\boldsymbol{\omega})d\omega = \int_{\phi=0}^{2\pi}\int_{\theta=0}^{\pi/2} L_i(\boldsymbol{x},\boldsymbol{\omega})\sin\theta\, d\theta\, d\phi \tag{3.14}$$

Irradiance is important since most often a ratio of the irradiance at a surface location will be reflected back into the scene resulting in indirect illumination if it reaches yet another surface. Indirect illumination is visually more important than we would think at first. The reflection and refraction of light resulting in indirect illumination is referred to as light scattering, which is the topic of section 3.4.

Before the light can scatter around in a scene it must be emitted from a light source, and before going into emission of light we must recall that light, in fact, consists of electromagnetic waves. The color of light, which radiance quantifies, is dependant on the wavelength of the radiant energy, Q, which we started out with.

Spectral Radiation

All the radiometric terms given above treat the *total* amount of incident or exitant radiation. Each term, in fact, has a spectral counterpart, such that for example:

$$Q = \int_0^{\infty} Q_\lambda d\lambda \tag{3.15}$$

where λ is the wavelength (recall figure 3.1).

In microscopic detail the *spectral radiant energy*, Q_λ, can be determined from the wavelength dependant number of photons and their energy:

$$Q_\lambda = n_\lambda e_\lambda = n_\lambda \frac{hc}{\lambda} \tag{3.16}$$

where $h \approx 6.6261 \cdot 10^{-34}\, Js$ is Planck's constant, and c is the speed of light (in vacuum $c = c_0 \approx 2.9979 \cdot 10^8\, m/s$).

To account for the spectral dependency of the radiometric terms, the values stored to represent eg. radiance or irradiance is a vector of three *color bands*. The three color components are red, green, and blue, and the model is known as the RGB color model. The RGB color model enables only a subset of the colors that exist in real life, but "the RGB model is defined by the

chromaticities of a CRT's phosphors" [38, p. 586], and therefore it models well the colors we can possibly show on a computer monitor. The subject of measuring and representing color is a scientific field known as *colorimetry*, see section 8.1 for further information.

Keeping in mind that each radiometric term can by treated at a single wavelength, and that the spectral counterpart of a radiometric term is denoted by a λ subscript, we can proceed to a description of light emission.

Light Emission

Throughout this section we have discussed how to measure radiance, which is a quantity representing the light that we see. Before radiance is incident on a surface it must be emitted from a light source. An important part of the illumination model that we wish to construct is, therefore, light emission.

To start with the simple case, we will consider a *blackbody*, which is a hypothetical object defined as a perfect absorber and emitter in each direction and at every wavelength. This means that a blackbody absorbs all incident radiance and radiates the same amount of energy as it absorbs. For example energy can be transferred to a metal by electricity in which case the emitted radiation will correspond to the electrical energy that is absorbed by the metal. This implies a relationship between radiance and temperature. Planck's law expresses the relationship between temperature and spectral radiant exitance of a blackbody in vacuum:

$$M_{b,\lambda} = \frac{2\pi C_1}{\lambda_0^5 (e^{C_2/(\lambda_0 T)} - 1)} \tag{3.17}$$

where T is the temperature of the object, $C_1 = hc_0^2 \approx 3.7418 \cdot 10^{-16} Wm^2$, and $C_2 = hc_0/k \approx 1.4388 \cdot 10^{-2} mK$, where $k \approx 1.3807 \cdot 10^{-16} J/K$ is Boltzmann's constant. Index of refraction $\eta = c_0/c$ should be included if the light source radiates into a medium where the speed of light c is not close to c_0, this is done by substitution of c_0 with $c = c_0/\eta$, and of λ_0 with $\lambda = \lambda_0/\eta$. The total radiant exitance of a blackbody, M_b, is related to temperature by the Stefan-Boltzmann law, which in a homogenous medium (constant refraction index) is given as follows:

$$M_b = \eta^2 \sigma T^4 \tag{3.18}$$

where $\sigma \approx 5.6704 \cdot 10^{-8} W/(m^2 K^4)$ is the Stefan-Boltzmann constant.

The Stefan-Boltzman law is useful if we know the temperature of a light source. In that case we can approximate the radiant exitance (under the assumption that the light source is a blackbody) at a surface location of the light source using (3.18).

The *emissivity* of an object specifies how well a real body radiates energy as compared to a blackbody. The blackbody is, however, a reasonable

approximation to light sources such as the sun. Further treatment of the subject is given in [121].

Sources such as light bulbs are more often described by the electrical power they absorb rather than the temperature they reach. Consider for example a light source with surface area A and power $\mathbf{\Phi}_s$ that emits light uniformly from all surface areas and in all directions, the constant emitted radiance, L_e, is given as:

$$\mathbf{\Phi}_s = \int_A \int_\Omega L_e cos\theta \, d\omega \, dA$$

and since L_e is constant and from the hemispherical version of (3.10) it follows that:

$$\begin{aligned} \mathbf{\Phi}_s &= \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\pi/2} \cos\theta \sin\theta d\theta \, d\phi \int_A dA \, L_e \\ &= 2\pi \left[\frac{-\cos^2\theta}{2} \right]_0^{\pi/2} AL_e \\ &= \pi A L_e \\ \Rightarrow L_e &= \frac{\mathbf{\Phi}_s}{\pi A} \end{aligned}$$

again it is assumed that the light bulb will emit all the energy that it absorbs. Still this simple equation is useful if we need to establish the radiance emitted by an indoor light source.

What has been outlined so far is a mathematical model that can describe how light in the form of electromagnetic waves radiate off objects and thereby illuminate a scene. The model gives a description of radiance, which is the radiometric quantity that most closely resembles the color that is perceived by the eye when looking at a surface area in 3D space.

Radiometry is the objective, quantitative way to measure electromagnetic radiation such as light. Since human sight to a large extend is subjective, the objective physical measures are not always sufficient to simulate human vision in a digital scene. Another scientific field called *photometry* is the answer to this shortcoming of radiometry. Photometry defines the same quantities as radiometry, but the units of measurement subjectively takes into account a standard human observer. While computer graphics most often work with radiometry some results from photometry can be added to the final result of an algorithm calculating radiance. We could describe it as a post processing adding extra visual realism to the final image. A short description of photometry will be given in the following section.

3.3 Photometry

While radiometry is the physical treatment of light, *photometry* deals with the quantification of the perception of light energy [30], that is, it includes the visual response of a standard observer, which radiometry does not.

In principle radiometry fully covers the area of photometry. While radiometry concentrates on electromagnetic waves in the optical radiation area see figure 3.1, photometry is measurements of visible light only. Photometry can be seen as a specification of this particular interval of wavelengths in the spectrum of electromagnetic radiation. The visible spectrum is, of course, particularly interesting to us since it is the area of waves that the eye is responsive to and that our brain therefore perceive.

A very simplified way to look at the eye would be as the human camera. Basically it is there to generate pictures or to collect light inputs for the brain to interpret into pictures. In the same way that a camera collects light which is later put on paper or into a digital medium.

Since the eye is a complicated receiver more sensitive to some wavelengths than others, it is not possible to use the same units as in radiometry. Instead the units of photometry are based on measurements of the human visual system. In 1924 the international commission on illumination CIE[4] created the photometric curve, see figure 3.9. This curve is based on test results and shows the photopic luminous efficiency of humans as a function of wavelength. The photopic luminous efficiency function show how well the eye adapts light from different wavelengths. Observers were asked to visually match the brightness of monochromatic light to wavelengths. The logarithm of the function is known as the relative visual brightness.

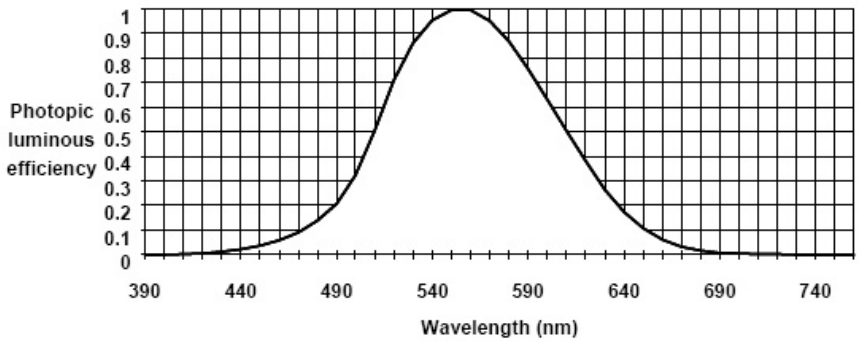


Figure 3.9: The CIE photometric curve (also called the photometric luminous efficiency function). The figure is identical to fig. 6 of [6].

The luminous efficiency function gives means by which we can make conversions between radiometric and photometric quantities. The following is a

[4]The abbreviation CIE originates in the French name: Commission Internationale d'Eclairage.

brief description of the different photometric quantities and their connection to the radiometric quantities, which were described in section 3.2.

Luminous Energy

Luminous energy is similar to radiant energy. Luminous energy is actually just radiant energy weighed photometrically. The weighing is related to the way in which the eye observes the radiated energy. For this purpose we use the photometric curve as mentioned before. The translation between radiant energy and photometric energy is done between luminous intensity and radiant intensity, see (3.19). To get the luminous energy we need to find the derivative of the luminous flux.

The meaning of luminous energy corresponding to the meaning of radiant energy only it is connected to observations of the eye instead of particles on a surface. Radiant energy is measured in joule which is the normal unit for energy. Luminous energy is measured in talbot.

All symbols used for photometry are the same as the corresponding radiometric symbol only a v subscript is added (for visible). Hence, luminous energy is Q_v.

Luminous Flux

The luminous flux or luminous power correlates to radiant flux. It is photometrically weighed radiant energy per unit time:

$$\mathbf{\Phi}_v = \frac{dQ_v}{dt}$$

Luminous flux is a subjective quantity describing the flow of light through space. We can connect luminous flux to spectral radiant flux by the following formula:

$$\mathbf{\Phi}_v = \int_\Lambda \mathbf{\Phi}_\lambda V(\lambda) d\lambda$$

where $V(\lambda)$ is the CIE photometric curve (fig. 3.9) representing the visual response of a standard observer, and Λ is the range of wavelengths that represents visible light.

Luminous Intensity

Luminous intensity is defined as:

$$I_v = \frac{d\mathbf{\Phi}_v}{d\omega}$$

where $d\omega$, as for radiant intensity, is a differential solid angle.

The relation between luminous intensity I_v and radiant intensity I, which indirectly also specifies the relation between the units watt (W) and lumen (lm), are again given by the CIE photometric curve (also called the photometric luminous efficiency function) as:

$$I_v(\boldsymbol{\omega}) = 683\frac{sr}{W}V(\lambda)I \tag{3.19}$$

where $V(\lambda)$ is the photometric luminous efficiency corresponding to the wavelength of the light, meaning that $V(\lambda)$ is a look up in figure 3.9.

Luminous Flux Area Density

Illuminance, obviously, corresponds to irradiance and is defined as:

$$E_v(\boldsymbol{x}) = \frac{d\boldsymbol{\Phi}_v}{dA}$$

Likewise, luminous exitance M_v corresponds to radiant exitance M.

Luminance

Luminance is the photometrical representation of light and corresponds, not surprisingly, to radiance:

$$L_v(\boldsymbol{x}, \boldsymbol{\omega}) = \frac{d^2\boldsymbol{\Phi}_v}{cos\theta\, d\omega\, dA}$$

Radiometry and photometry has many similarities and the units and quantities of the two fields correlate to each other. Table 3.1 shows this correlation. The table is put together from different sources. The description of physical quantities is from [97], and the table has been expanded with terms from [6].

Through the past few sections we have established a fair understanding of the different quantities that are used for representation and measurement of light. These measurements are what we wish to obtain from our digital scene and therefore the propagation of light from sources to reflecting objects must be present in our illumination model. How light scatters in an environment and illuminates objects that reflect light off (or refract it through) their surfaces is the topic of the next section.

3.4 Light Scattering

In section 3.2 the term *radiance*, closely resembling what the eyes perceive as color, was given a mathematical interpretation and was described as light intensity at a point in space. Emission of light dependant on the temperature or power (also called *wattage*) of a light source was treated.

Quantity	Radiometry	Photometry
Energy	Radiant energy Q $[J]$	Luminous energy Q_v $[talbot]$
Power	Radiant flux (Radiant power) $\mathbf{\Phi}$ $[W = J/s]$	Luminous flux (luminous power) $\mathbf{\Phi}_v$ $[lm]$
Power per unit solid angle	Radiant intensity I $[W/sr]$	Luminous intensity I_v $[cd = lm/sr]$
Power per unit area	Radiant flux area density (radiant exitance M or irradiance E) $[W/m^2]$	Luminous flux area density (luminous exitance M_v or illuminance E_v) $[lux = lm/m^2]$
Power per unit area per unit solid angle	Radiance L $[W/(m^2 sr)]$	Luminance L_v $[cd/m^2]$
Ratio of reflected to incident light	Reflectance ρ_r	Luminous reflectance
Ratio of refracted to incident light	Transmittance ρ_t	Luminous transmittance

Table 3.1: The relation between physical, radiometric, and photometric quantities. Symbols and units are stated for the radiometric and photometric quantities.

What was not described is how light, when emitted from a light source into a scene, scatters between and through the surfaces. The exitant light from a surface area is highly dependant on the incident light. Light scattering is the subject of this section.

Light incident on a surface has two possible ways of scattering; reflecting off the surface and/or refracting through the surface. *Reflection* is the process by which radiant flux ($\mathbf{\Phi}$), incident on a stationary surface, leaves that surface from the incident side without change in frequency; *reflectance* is the fraction of the incident flux that is reflected [88]. *Refraction* is the process by which radiant flux, incident on a surface, transmits into the medium underneath the surface; *transmittance* is the fraction of the incident flux that is refracted.

Unlike emitted flux (as described in sec. 3.2), reflected flux depends not only on the angle at which the incident flux impinges on the surface, but also on the direction being considered for the reflected flux [121]. Reflection at a surface area is therefore *bidirectional*, see figure 3.10a, and to gather all the reflected radiance in a given direction it is necessary to integrate over all incident directions of the hemisphere above dA, see figure 3.10b.

Nicodemus et al. [88] described a widely accepted theoretical framework and nomenclature for reflectance. Parts of this framework is presented in

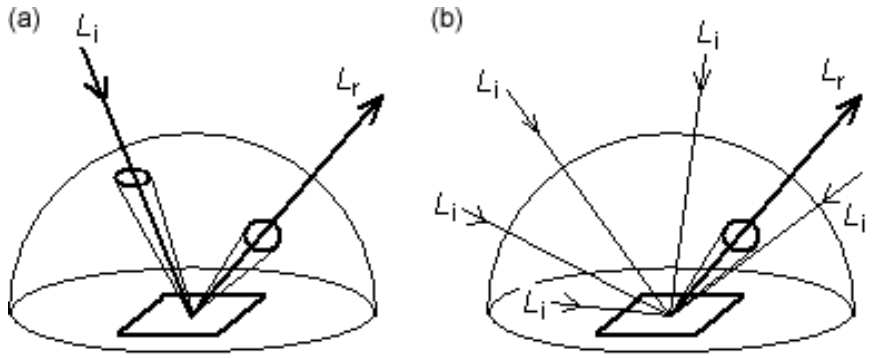


Figure 3.10: (a) Bidirectional reflection. (b) Hemispherical-directional reflection.

the following.

BSSRDF

Let $d\mathbf{\Phi}_i$ denote the portion of a radiant flux $d\mathbf{\Phi}_i$ arriving from within a differential solid angle $d\omega'$ at a differential surface area dA_i centered around the surface location $\boldsymbol{x}'$. Then the most general description of bidirectional reflectance within the domain of ray optics originates from the fact that the portion of reflected radiance dL_r, at the point $\boldsymbol{x}$ within the differential solid angle $d\omega$ coming from $d\mathbf{\Phi}_i$, is directly proportional to $d\mathbf{\Phi}_i$:

$$S = \frac{dL_r}{d\mathbf{\Phi}_i} \tag{3.20}$$

If we assume about the mechanism involved that there is some form of interaction between radiation and matter and that the flux incident at some location $\boldsymbol{x}'$ is scattered to some other location $\boldsymbol{x}$ as a result of this interaction, then the factor of proportionality S will, in general, have the following dependencies:

$$S = S(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}')$$

The function S is called the *Bidirectional Scattering Surface Reflectance Distribution Function* (BSSRDF). The BSSRDF is completely general, the only simplifying assumptions are those of ray optics described in section 3.1 and besides we ignore dependencies of the BSSRDF that are not of a geometrical nature.

It may be surprising that the surface location of reflection $\boldsymbol{x}$ not necessarily is identical to the location of incident flux $\boldsymbol{x}'$. The reason is that light incident on an object normally enters the material at one location, scatters around, and leaves at a different location, see figure 3.11. This visual effect

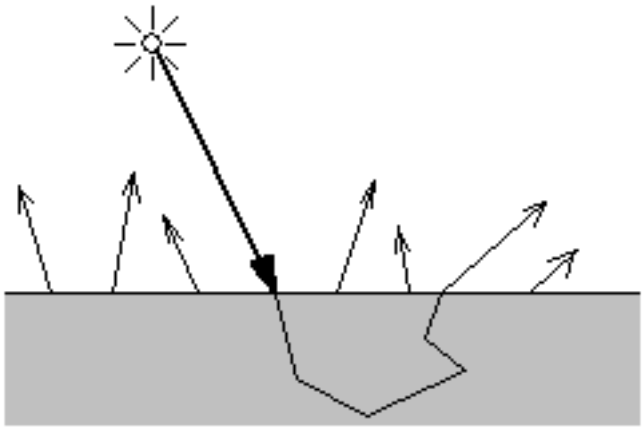


Figure 3.11: The surface location of reflection is not necessarily identical to the location of incident flux. This figure is a reconstruction of fig. 2.2 in [60].

is called *subsurface scattering* and it "is particularly noticeable for translucent materials such as marble and skin, but happens to some degree for all non-metallic materials" [60].

The surface locations are (in (u, v)-coordinates) each two parameters and the solid angles are each defined around a direction, which in spherical coordinates also amounts to two parameters per angle. All in all the BSSRDF is an eight-dimensional function, which is quite intractable to handle. We will return to the composition of approximate BSSRDFs when describing subsurface scattering towards the end of this section.

BRDF

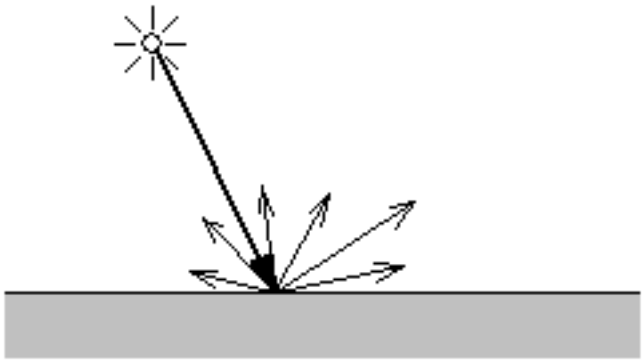


Figure 3.12: Light scattering under the assumption that incident flux is reflected in the same point. This figure is a reconstruction of fig. 2.3 in [60].

Though subsurface scattering is an important visual effect especially with respect to translucent materials, it is for most other materials a reasonable

assumption that incident flux is reflected in the same point, see figure 3.12. If we assume that $\boldsymbol{x} = \boldsymbol{x}'$ the surface is no longer scattering light internally and we call the factor of proportionality a *Bidirectional Reflectance Distribution Function* (BRDF). The BRDF is six dimensional and it is denoted $f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}')$. To remove the argument $\boldsymbol{x}'$ representing the center of the differential area on which the incident flux impinges, we must in general integrate over the entire area of incident flux A_i:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = \int_{A_i} S(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') dA_i \tag{3.21}$$

The incident flux $d\boldsymbol{\Phi}_i$ of (3.20) is dependant on both the incident differential area dA_i and the incident different solid angle $d\omega'$ to which it is confined. Therefore it is not entirely comparable to $d\boldsymbol{\Phi}$ of (3.12), rather the incident flux is dependant on the differential irradiance incident on dA_i from $d\omega'$, meaning that

$$d\boldsymbol{\Phi}_i = d^2\boldsymbol{\Phi} = dE_i \, dA_i \tag{3.22}$$

It then follows from (3.20) and (3.22) (still in general) that:

$$S(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') = \frac{dL_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}')}{d\boldsymbol{\Phi}_i} = \frac{d^2 L_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}')}{dE_i \, dA_i} \tag{3.23}$$

$$\Leftrightarrow f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = \int_{A_i} S(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') dA_i = \frac{dL_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}')}{dE_i} \tag{3.24}$$

which shows that setting $\boldsymbol{x} = \boldsymbol{x}'$ is a simplifying assumption that avoids the integral over A_i, and under that assumption (3.24) states that the portion of reflected radiance dL_r at the point $\boldsymbol{x}$ within the differential solid angle $d\omega$ is directly proportional to the incident irradiance dE_i confined within the differential solid angle $d\omega'$ from the direction (θ', ϕ').

To find a formula for dE_i it follows by substitution of (3.22) in (3.13) that:

$$\frac{1}{\cos\theta'} \frac{dE_i \, dA_i}{d\omega' \, dA_i} = L_i(\boldsymbol{x}, \boldsymbol{\omega}') \quad \Rightarrow \quad \frac{dE_i}{d\omega'} = L_i(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta' \tag{3.25}$$

which according to (3.24) indicates that:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = \frac{dL_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}')}{L_i(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta' \, d\omega'} \tag{3.26}$$

Suppose the unit normal $\boldsymbol{n}$ at the surface location $\boldsymbol{x}$ is known and that the direction of the incident solid angle is given as a normalized three-dimensional vector $\boldsymbol{\omega}'$, then the hemispherical-directional reflected radiance

(fig. 3.10b) can be computed using the following recursive formula derived from (3.26) by integration over the entire hemisphere:

$$\begin{aligned} L_r(\boldsymbol{x}, \boldsymbol{\omega}) &= \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta' \, d\omega' \\ &= \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') (\boldsymbol{\omega}' \cdot \boldsymbol{n}) \, d\omega' \end{aligned} \tag{3.27}$$

Intuitively the BRDF (f_r) specifies the portion of radiance incident at a surface location that is reflected in another direction. The BRDF at a single point on a surface is a four-dimensional function, which can be measured from real surfaces using, for example, a gonioreflectometer [126]. Many BRDF models exist; both empirical BRDFs based on sample data, physically based BRDFs, and even intuitive BRDFs based on simplicity and the quality of resulting images. Surface reflectance is important in the specification of different BRDFs.

Reflectance

Reflectance is defined as the ratio of reflected to incident flux:

$$\rho(\boldsymbol{x}) = \frac{d\boldsymbol{\Phi}_r}{d\boldsymbol{\Phi}_i} \tag{3.28}$$

By integration over the hemisphere above $\boldsymbol{x}$ in (3.13), we can achieve the following:

$$\frac{d\boldsymbol{\Phi}_i(\boldsymbol{x})}{dA} = \int_\Omega L_i(\boldsymbol{x}, \boldsymbol{\omega}') (\boldsymbol{\omega}' \cdot \boldsymbol{n}) \, d\omega' \tag{3.29}$$

Considering (3.27) and (3.13) we can also find the following equation for the differential reflected flux:

$$\frac{d\boldsymbol{\Phi}_r(\boldsymbol{x})}{dA} = \int_\Omega \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') (\boldsymbol{\omega}' \cdot \boldsymbol{n}) \, d\omega' \, (\boldsymbol{\omega} \cdot \boldsymbol{n}) \, d\omega \tag{3.30}$$

Holding (3.30) and (3.29) up against each other reveals the following formula for calculation of reflectance:

$$\rho(\boldsymbol{x}) = \frac{d\boldsymbol{\Phi}_r}{d\boldsymbol{\Phi}_i} = \frac{\int_\Omega \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') (\boldsymbol{\omega}' \cdot \boldsymbol{n}) (\boldsymbol{\omega} \cdot \boldsymbol{n}) \, d\omega' \, d\omega}{\int_\Omega L_i(\boldsymbol{x}, \boldsymbol{\omega}') (\boldsymbol{\omega}' \cdot \boldsymbol{n}) \, d\omega'} \tag{3.31}$$

To give a brief example of the usefulness of reflectance we can find the BRDF of diffuse surfaces, $f_{r,d}(\boldsymbol{x})$, which by definition of a diffuse material is indifferent to the directions of incident and exitant light. According to (3.31) we find:

$$\begin{aligned}
\rho_d(\boldsymbol{x}) &= \frac{f_{r,d}(\boldsymbol{x}) \int_\Omega \int_\Omega L_i(\boldsymbol{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \boldsymbol{n})(\boldsymbol{\omega} \cdot \boldsymbol{n})\, d\omega'\, d\omega}{\int_\Omega L_i(\boldsymbol{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \boldsymbol{n})\, d\omega'} \\
&= f_{r,d}(\boldsymbol{x}) \int_\Omega (\boldsymbol{\omega} \cdot \boldsymbol{n})\, d\omega \frac{\int_\Omega L_i(\boldsymbol{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \boldsymbol{n})\, d\omega'}{\int_\Omega L_i(\boldsymbol{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \boldsymbol{n})\, d\omega'} \\
&= f_{r,d}(\boldsymbol{x})\pi \\
\Rightarrow f_{r,d}(\boldsymbol{x}) &= \frac{\rho_d(\boldsymbol{x})}{\pi} \qquad (3.32)
\end{aligned}$$

Most often it is assumed that the reflectance is constant over the entire surface of the diffuse object in which case the BRDF is merely a constant given as $f_{r,d} = \rho_d/\pi$. Other more complex BRDFs will be described in later chapters of this part.

BTDF

While the BRDF, f_r, specifies the amount of radiance that is reflected, we could consider a similar function, f_t, specifying the amount of radiance that is refracted. f_t is called a *Bidirectional Transmittance Distribution Function* BTDF and it could be derived in a similar way only considering dL_t instead of dL_r in (3.20).

Measuring the BTDF from real surfaces is, however, harder than measuring the BRDF [126] (since the measurement must be done on the opposite side of the surface, that is, inside the medium). In computer graphics perfectly specular refraction (see section 3.1) is usually the only case that is considered, if a more complex distribution function is needed the BTDF is most often approximated by an empirical model tuned by a few handpicked parameters.

Reflectance is a ratio that can be used in order to establish the amount of reflected radiance, ρ. If a surface is perfectly specular no radiance will be absorbed and the amount of refracted radiance, that is, the transmittance, will be $1 - \rho$.

The reflectance of a perfectly specular surface is a special case of (3.31). To find this *specular reflectance* without solving (3.31), we need to borrow some theory from wave optics. The *Fresnel equations*, discovered in 1821 (see app. B), are based on the wave theory of light and they take *polarization* into account. Polarization can be explained as a superposition of two transverse waves, which oscillate in directions perpendicular to each other and to the direction of propagation [30]. The Fresnel equations find the amount of reflected light in a medium with index of refraction η_1 incident on a material with index of refraction η_2 at the angle θ_1 with the surface normal turning outwards, refracting light at the angle θ_2 with the surface normal turning inwards. The two components of polarized light are given as [60]:

$$\rho_{\parallel} = \frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2} \tag{3.33}$$

$$\rho_{\perp} = \frac{\eta_1 \cos\theta_1 - \eta_2 \cos\theta_2}{\eta_1 \cos\theta_1 + \eta_2 \cos\theta_2} \tag{3.34}$$

Light usually becomes polarized due to scattering [30]. Light emitted from light sources is mostly unpolarized, or so called *natural* light. For unpolarized light the Fresnel reflection coefficient, F_r, which is identical to the specular reflectance, becomes [60]:

$$F_r(\theta) = \frac{1}{2}(\rho_{\parallel}^2 + \rho_{\perp}^2) = \frac{d\mathbf{\Phi}_r}{d\mathbf{\Phi}_i} \tag{3.35}$$

Some computationally less expensive approximations to (3.35) have been proposed. One of them was derived by Schlick and has the following empirical form [118]:

$$F_r(\theta) \approx F_0 + (1 - F_0)(1 - \cos\theta)^5 \tag{3.36}$$

where F_0 is the value of (3.35) at normal incidence:

$$F_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2 \tag{3.37}$$

BSDF

Using the Fresnel equation, the BTDF is usually incorporated in the BRDF to account for perfectly specular refraction. A more general model must, however, specify both a BRDF and a BTDF over the entire unit sphere surrounding each surface location. Since BRDFs and BTDFs in most cases only are defined on hemispheres, the more general model amounts to four different functions[5], which more conveniently can be gathered into a single function, f_s, called a *Bidirectional Scattering Distribution Function* BSDF.

Participating Media

It is commonly assumed that all media in a scene are homogenous, and as stated previously, according to Hero's principle, light travels in straight lines in homogenous media. The result of these simplifying assumptions is that a rendering method usually can be content with evaluation of a distribution function, such as those described above, at the visible surface areas of a scene.

Suppose that we wish to render a cloud of smoke or dust, a glass of milk, a marble statue, or human skin. In all those cases (and many others) the

[5] Two BRDFs and two BTDFs - one of each for each side of the surface.

material to be rendered is inhomogenous and translucent, meaning that the light will refract into the medium and scatter around before it is refracted back into the medium from where it came at another surface location. In other words we need a different rendering method for inhomogenous media.

Rendering of a translucent material will often be combined with other rendering techniques that only take surface scattering into consideration, this is obvious since a scene often will contain both kinds of object materials. The inhomogenous media are therefore traditionally referred to as participating media. The material of participating media is often quite translucent, and since light does not move in straight lines inside it, we must take the entire volume of a participating medium into account (or at least a slab close to the border area) when an object of such material is to be rendered.

Radiance is still the quantity that we want to calculate. We must, however, take into consideration that the radiation is frequently weakened and scattered by interaction with matter when traversing through an inhomogenous medium. Let $\sigma_t(\boldsymbol{x}) = \kappa(\boldsymbol{x})\rho$ describe the *extinction coefficient*, which is composed of κ, the *mass absorption coefficient*, and ρ, the density of the material. The extinction coefficient $\sigma_t(\boldsymbol{x})$ defines the rate of radiation that is absorbed at location $\boldsymbol{x}$ in the medium. Now, after traversing a thickness ds in the direction of the light's propagation, the radiance will be absorbed by the medium according to the extinction coefficient [15]:

$$\frac{dL(s)}{ds} = -\kappa(s)\rho L(s) = -\sigma_t(s)L(s) \tag{3.38}$$

All the radiation that is absorbed is not necessarily transformed to other kinds of energy, some or all of it may as well be scattered back into the medium. In order to specify the out-scattered radiance we must specify the angular distribution of the scattered radiation, this is done by means of a phase function $p(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega'}) = p(\boldsymbol{x}, \boldsymbol{\omega} \cdot \boldsymbol{\omega'})$ which in the general case must agree with the following [15]:

$$\int_{\Omega_{4\pi}} p(\boldsymbol{\omega} \cdot \boldsymbol{\omega'}) \frac{d\omega'}{4\pi} = \varpi_0 \leq 1$$

where ϖ_0 represents the fraction of out-scattered radiance, while $1 - \varpi_0$ is the fraction that has been turned into other forms of energy (or radiation outside the visible spectrum). ϖ_0 is called the *albedo for single scattering*. In computer graphics it is most commonly assumed that the phase function is normalized to unity, that is, $\varpi_0 = 1$, which is the conservative case of perfect scattering.

The *scattering coefficient* $\sigma_s(\boldsymbol{x})$ is the rate of radiation that is scattered back into the medium at location $\boldsymbol{x}$. In the case of perfect scattering $\sigma_s = \sigma_t$. Otherwise the scattering coefficient is given as $\sigma_s = \varpi_0 \sigma_t$. In terms of thickness we can write the out-scattered radiance as:

$$\frac{dL(s)}{ds} = \sigma_t(s) \int_{\Omega_{4\pi}} p(s, \boldsymbol{\omega'}) L(s, \boldsymbol{\omega'}) d\omega' \tag{3.39}$$

Note how (3.39) to a certain extend resembles (3.27). For consistency with other texts, and since it may come in handy, we can also define an *absorbtion coefficient* as $\sigma_a = \sigma_t - \sigma_s = (1 - \varpi_0)\sigma_t$.

If the medium is not a light source itself it is called a *scattering medium*. In that case the out-scattered radiance is the total emission from a given point in the medium.

Suppose the radiance transfer is placed in a Cartesian coordinate system (which is generally the case), then the thickness $s(\boldsymbol{x}, \boldsymbol{\omega})$ is dependant on both the current location in the medium $\boldsymbol{x}$ and the incident direction $\boldsymbol{\omega}$. The total change in radiance per unit distance for a scattering media is then given as:

$$(\boldsymbol{\omega} \cdot \nabla) L(\boldsymbol{x}, \boldsymbol{\omega}) = -\sigma_t(\boldsymbol{x}) \left(L(\boldsymbol{x}, \boldsymbol{\omega}) - \int_{\Omega_{4\pi}} p(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega'}) L(\boldsymbol{x}, \boldsymbol{\omega'}) d\omega' \right) \tag{3.40}$$

Treatment of volumetric light sources such as fire or plasma is beyond the scope of this project, for the proper expansion of (3.40) see eg. [100], here we will restrict ourselves to scattering media only, and therefore (3.40) is sufficient for our purposes.

In order to derive a formal solution for (3.40) it is more convenient to keep the notion of thickness for a moment. The simpler form of (3.40) is the simple addition of (3.38) and (3.39):

$$\frac{dL(s)}{ds} = -\sigma_t(s) \left(L(s) - \int_{\Omega_{4\pi}} p(s, \boldsymbol{\omega'}) L(s, \boldsymbol{\omega'}) d\omega' \right) \tag{3.41}$$

The formal solution is presented in [15] (though in a more general form) as:

$$L(s) = L(0) e^{-\tau(s,0)} + \int_0^s e^{-\tau(s,s')} \sigma_t(s') \int_{\Omega_{4\pi}} p(s', \boldsymbol{\omega'}) L(s', \boldsymbol{\omega'}) d\omega' ds' \tag{3.42}$$

where $\tau(s, s')$ is the *optical thickness* of the material between the points s and s', which is given as:

$$\tau(s, s') = \int_{s'}^{s} \sigma_t(t) dt$$

Let the scattering participating medium be described geometrically by a finite volume $\mathcal{V} \subset \mathbb{R}^3$, and let $\partial\mathcal{V}$ denote the boundary of $\mathcal{V}$. Then rewriting (3.42) in terms of a Cartesian coordinate system results in the following formulation of the radiance at a location $\boldsymbol{x}$ in a direction $\boldsymbol{\omega}$:

$$L(\boldsymbol{x},\boldsymbol{\omega}) = \int_{\boldsymbol{x}_{\partial\mathcal{V}}}^{\boldsymbol{x}} e^{-\tau(\boldsymbol{x},\boldsymbol{x}')}\sigma_t(\boldsymbol{x}')\int_{\Omega_{4\pi}} p(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega}')L(\boldsymbol{x}',\boldsymbol{\omega}')d\omega' d\boldsymbol{x}' + e^{-\tau(\boldsymbol{x},\boldsymbol{x}_{\partial\mathcal{V}})}L(\boldsymbol{x},\boldsymbol{\omega}) \quad (3.43)$$

where $\boldsymbol{x}_{\partial\mathcal{V}} \in \partial\mathcal{V}$ is the closest surface point from $\boldsymbol{x}$ in direction $-\boldsymbol{\omega}$ ([100]) (recall that $\boldsymbol{\omega}$ is the direction in which the radiance is traveling).

In computer graphics (3.43) is referred to as *the volume rendering equation*, since "it is the equation that must be solved in order to render participating media" [60]. In our case it is not completely general since we have chosen to work with scattering media only, meaning that we left out the emission term. The simpler case introduced at the beginning of this section, where we consider the interaction between radiation and media at the surface only, of course, has a similar equation. This is simply called *the rendering equation* and it is the subject of section 3.5.

The volume rendering equation is rather complex and has only recently been approached in real-time. Some of the most recent real-time approximations are presented in [52]. One way to find a rough approximation is to construct a depth map of the scene and to use this for a simple thickness calculation.

Another approach is to simplify the equation by picking out a subset of materials that have some characteristics in common. Based on those characteristics some simplifying assumptions can often be made. A large part of the materials that we wish to simulate are quite dense. Materials such as marble, milk, leaves, and skin all have a well defined boundary ($\partial\mathcal{V}$). Rendering of those relatively dense materials can be approximated using the original assumption that we need only consider the surface area. This approach, in fact, indicates that we can use (3.43) to find an approximation of the BSSRDF (S in (3.20)).

As mentioned before the visual effect that we wish to simulate when using the BSSRDF is called subsurface scattering. This particular effect is especially visible in translucent materials with a well-defined boundary, and it is treated as the concluding part of this section on light scattering.

Subsurface Scattering

Why do we need to simulate subsurface scattering? (a) Though the effect is often subtle, it gives a powerful visual cue, and (b) translucent objects are more pleasing, interesting, and realistic [52]. Diffuse materials, such as cloth and skin, often tend to become rough and "unfriendly" in computer graphics when subsurface scattering is not taken into account.

After [45] in which (3.43) is approximated using a Monte Carlo method in combination with a standard ray tracer, the idea mentioned, namely that

the volume rendering equation can be used to define an approximate BSSRDF, has bread a series of articles [29, 64, 105, 65, 61] which resulted in an approximative BSSRDF model that has been used for interactive subsurface scattering in recent articles such as [75, 46, 81, 80, 13].

It would carry us too far to go through the theory presented in those articles. However, we can in short present some of the simplifying assumptions that are introduced and some of the results that they achieve.

First it is assumed that the phase function is independent of the single scattering albedo ϖ_0, meaning that the fraction of out-scattered radiance cannot change according to the phase function. Instead the single scattering albedo is replaced by:

$$\Lambda = \frac{\sigma_s}{\sigma_t}$$

and it is assumed that the phase function is normalized to unity $\varpi_0 = 1$. Under these assumptions $\sigma_s = \sigma_t - \sigma_a \neq \varpi_0 \sigma_t$. Furthermore it is assumed that the phase function is a function only of the phase angle, $p(\boldsymbol{\omega}, \boldsymbol{\omega}') = p(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')$, meaning that the phase function is the same throughout a translucent object. These assumptions are done to ease calculations involving the phase function.

> **Definition 6 (Isotropy)** *A medium is said to be* isotropic *at a point, if the radiance is independent of direction at that point. And if the radiance is the same at all points and in all directions, the medium is said to be* homogenous *and isotropic.* [15]

Next they look closer at the process of subsurface scattering. It is observed that "the light distribution in highly scattering media tends to become isotropic. This is true even if the initial light source distribution and phase function are highly anisotropic" [65]. Which means that multiple scattering beneath the surface of a translucent object can be represented by a diffusion approximation S_d, which in [65] is given as:

$$S_d(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') = \frac{1}{\pi} F_t(\eta, \boldsymbol{\omega}) R_d(\|\boldsymbol{x} - \boldsymbol{x}'\|) F_t(\eta, \boldsymbol{\omega}') \tag{3.44}$$

where $F_t = 1 - F_r$ is the unpolarized Fresnel transmittance (cf. (3.35)) and R_d is the diffuse reflectance, which is approximated by a dipole light source consisting of two point sources; one above the surface and one beneath. [75] provides a good overview of how R_d is calculated, while constants measured for different kinds of materials are provided in [65] which originally introduced the dipole light source.

Besides the diffusion approximation the BSSRDF given in this method also consists of a single scattering term $S^{(1)}(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}')$. The single scattering term defines how the radiance $L_i(\boldsymbol{x}', \boldsymbol{\omega}')$ incident on the translucent object at the surface location $\boldsymbol{x}'$ from the direction $\boldsymbol{\omega}'$ contributes to the outgoing radiance $L_o(\boldsymbol{x}, \boldsymbol{\omega})$ at the surface location $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$ by means of refraction through the translucent object according to Fresnel terms, absorption and the phase function.

The complete BSSRDF model (as presented in the above mentioned series of articles) is the sum of the diffusion approximation and the single scattering term:

$$S(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') = S_d(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') + S^{(1)}(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{x}', \boldsymbol{\omega}') \tag{3.45}$$

Most interactive approaches limit themselves to isotropic materials[6]. In that case $S \approx S_d$. Lensch et al. [75] show how the diffuse subsurface scattering reflectance R_d can be incorporated in a radiosity solution. Carr et al. [13] move this solution to the GPU. Hao et al. [46] incorporate S_d in a local illumination model, and finally Mertens et al. [81, 80] use a clever hierarchical boundary element method similar to hierarchical radiosity with S_d and report robust results. They all obtain interactive frame rates.

The intensions of this section (3.4) have been (a) to provide a good understanding of light scattering in general, and (b) to give a brief perspective on the current approaches to subsurface scattering. We have not given all the details, but we have provided the necessary references if specific details are needed. In part III we present our own simplified approach to subsurface scattering based on the theory of participating media.

This concludes the section on light scattering. In the next section we use the theory on light scattering to derive the rendering equation from its original enunciation.

3.5 The Rendering Equation

Most of the main computer graphics research areas were founded in the early eighties. Major global illumination techniques such as ray tracing, first presented for computer graphics by Turner Whitted in [137], and radiosity, from Goral et al. in [40], were more or less based on theoretical results from heat transfer, optics, and radiometry. A straightforward relationship between the different techniques was, however, not clear until Kajiya's enunciation of *the rendering equation* in [66], where he shows that ray tracing and radiosity basically seek to solve the same integral equation.

The rendering equation is based on thermal radiation heat transfer (see [121]) and fundamentally it derives from the same formula as has been de-

[6] An isotropic medium corresponds to a diffuse surface.

scribed in sections 3.2 and 3.4. In its original form the rendering equation was given as below, [66]:

$$L(\boldsymbol{x}, \boldsymbol{x}') = g(\boldsymbol{x}, \boldsymbol{x}') \left[\epsilon(\boldsymbol{x}, \boldsymbol{x}') + \int_S \rho(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}'') L(\boldsymbol{x}', \boldsymbol{x}'') d\boldsymbol{x}'' \right] \tag{3.46}$$

where $\boldsymbol{x}$, $\boldsymbol{x}'$, and $\boldsymbol{x}''$ are surface locations (possibly at different surfaces in the scene, $\boldsymbol{x}$ may be an eye point and $\boldsymbol{x}''$ a position on a light source). S is the union of all surfaces, meaning that the integral is taken over all points in the geometry of a scene. $g(\boldsymbol{x}, \boldsymbol{x}')$ is a visibility term specifying whether or not the point $\boldsymbol{x}'$ is visible from $\boldsymbol{x}$. $\epsilon(\boldsymbol{x}, \boldsymbol{x}') = L_e(\boldsymbol{x}, \boldsymbol{x}')$ is radiance emitted from the surface location $\boldsymbol{x}'$ reaching the surface location $\boldsymbol{x}$. As described in section 3.2 emission can result from, for example, a supply of electrical power, and it should be added to the radiance reflected at the point $\boldsymbol{x}'$.

In order to understand the transport of light from one surface location to another, we must first prove that radiance is invariant along straight paths.

Invariance of Radiance Along Straight Paths

According to Hero's principle light travels in straight lines in a homogenous medium (cf. section 3.1). Therefore it is evident that light traveling through a homogenous medium from one surface location $\boldsymbol{x}$ in a scene towards another $\boldsymbol{y}$ will reach the other point provided that no occluding object is situated between the two points.

Since light is traveling in a straight line between two surface locations, a differential solid angle $d\omega_{\boldsymbol{xy}}$ for radiance $L_o(\boldsymbol{x}, \boldsymbol{y})$ through a differential surface area $dA_{\boldsymbol{x}}$ centered at $\boldsymbol{x}$ can be subtended by a differential surface area $dA_{\boldsymbol{y}}$ centered at $\boldsymbol{y}$. The subscript o denotes that the radiance is $o$utgoing, and the radiance leaving $\boldsymbol{x}$ and arriving at $\boldsymbol{y}$ can thus be described as:

$$L_o(\boldsymbol{x}, \boldsymbol{y}) = \frac{d^2 \boldsymbol{\Phi}_o}{\cos\theta_{\boldsymbol{x}} \, d\omega_{\boldsymbol{xy}} \, dA_{\boldsymbol{x}}} \tag{3.47}$$

Letting the subscript i denote the radiance incident on a surface, we can, conversely, define the radiance arriving at $\boldsymbol{y}$ from $\boldsymbol{x}$ as:

$$L_i(\boldsymbol{y}, \boldsymbol{x}) = \frac{d^2 \boldsymbol{\Phi}_i}{\cos\theta_{\boldsymbol{y}} d\omega_{\boldsymbol{yx}} \, dA_{\boldsymbol{y}}} \tag{3.48}$$

Recall from section 3.2 that knowledge of the area subtending a solid angle gives us a possibility to determine the solid angle by projection of the area on a sphere centered at the point where the solid angle is formed.

Consider the solid angle ω formed at $\boldsymbol{x}$ subtended by a surface area $A_{\boldsymbol{y}}$ centered around $\boldsymbol{y}$, figure 3.13 pictures the differential case. Let r be the distance between $\boldsymbol{x}$ and $\boldsymbol{y}$, then according to (3.7) $\omega = A_s/r^2$, where A_s is the area of $A_{\boldsymbol{y}}$ projected to the sphere s of radius r centered at $\boldsymbol{x}$. If $A_{\boldsymbol{y}}$ is

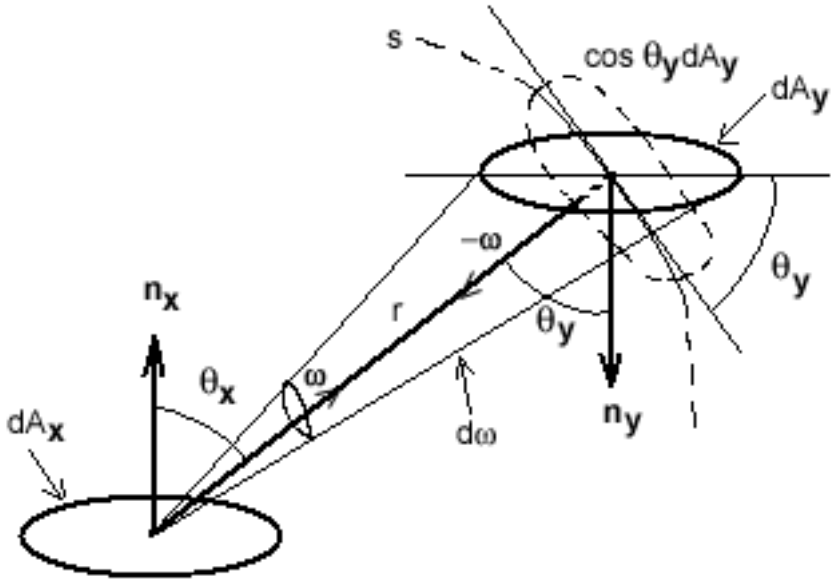


Figure 3.13: A solid angle formed at $\boldsymbol{x}$ subtended by an arbitrarily oriented differential surface area $dA_{\boldsymbol{y}}$ centered at $\boldsymbol{y}$.

small, a reasonable approximation to A_s is the projection of $A_{\boldsymbol{y}}$ on the plane tangent to s and normal to the direction $\boldsymbol{\omega} = \frac{\boldsymbol{y}-\boldsymbol{x}}{\|\boldsymbol{y}-\boldsymbol{x}\|}$ around which the solid angle is defined. Meaning that:

$$\omega = \frac{A_s}{r^2} \approx \frac{A_{\boldsymbol{y},p}}{r^2}$$

If we let $A_{\boldsymbol{y}}$ approach the infinitesimal surface area $dA_{\boldsymbol{y}}$, the differential solid angle (as specified in (3.9)) will, at the limit, approach:

$$d\omega = \frac{dA_s}{r^2} = \frac{dA_{\boldsymbol{y},p}}{r^2}$$

As shown in figure 3.8 we can further specify the differential projected area:

$$d\omega = \frac{dA_{\boldsymbol{y},p}}{r^2} = \frac{\cos\theta_{\boldsymbol{y}}\, dA_{\boldsymbol{y}}}{r^2} = \frac{(-\boldsymbol{\omega}\cdot\boldsymbol{n}_{\boldsymbol{y}})\, dA_{\boldsymbol{y}}}{r^2} \tag{3.49}$$

where $\theta_{\boldsymbol{y}}$ is the angle between the differential surface area $dA_{\boldsymbol{y}}$ and the plane tangent to s and normal to $\boldsymbol{\omega}$, which is equivalent to the angle between $-\boldsymbol{\omega}$ and the unit normal $\boldsymbol{n}_{\boldsymbol{y}}$ to $A_{\boldsymbol{y}}$ at $\boldsymbol{y}$.

Returning to the transport of light between two surface locations, we can substitute (3.49) in (3.47) and (3.48) receiving the following:

$$L_o(\boldsymbol{x},\boldsymbol{y}) \quad = \quad \frac{r^2\, d^2\boldsymbol{\Phi}_o}{\cos\theta_{\boldsymbol{x}}\cos\theta_{\boldsymbol{y}}\, dA_{\boldsymbol{y}}\, dA_{\boldsymbol{x}}}$$

$$L_i(\boldsymbol{y}, \boldsymbol{x}) \quad = \quad \frac{r^2 \, d^2 \boldsymbol{\Phi}_i}{\cos\theta_{\boldsymbol{y}} \cos\theta_{\boldsymbol{x}} \, dA_{\boldsymbol{x}} \, dA_{\boldsymbol{y}}}$$

According to the first law of thermodynamics (which is also known as the law of conservation of energy, see eg. [11]) the total energy of a system and its surroundings is *constant*. Meaning that all the energy outgoing at $\boldsymbol{x}$ that will reach $\boldsymbol{y}$ (which by Hero's principle will be all light that is not occluded) without being absorbed by participating media, will also be incident at $\boldsymbol{y}$. If light is occluded $L_o(\boldsymbol{x}, \boldsymbol{y}) = L_i(\boldsymbol{y}, \boldsymbol{x}) = 0$. Therefore it is indeed the case that in a vacuum:

$$L_o(\boldsymbol{x}, \boldsymbol{y}) = L_i(\boldsymbol{y}, \boldsymbol{x}) \tag{3.50}$$

□

Returning to the original problem; we seek to simulate human vision in a digital scene and show the result on a monitor. Basically this means that we want to find the total radiance incident at the differential surface area representing the eye. The total radiance incident at the eye centered around $\boldsymbol{x}$ is either radiance $L_i(\boldsymbol{x}, \boldsymbol{x}')$ transported from all surface locations in the scene to the eye, or the radiance $L_i(\boldsymbol{x}, \boldsymbol{\omega})$ incident at the eye from all directions given by the hemisphere over the eye.

The rendering equation describes the problem in terms of surface locations only, while the original definition of radiance describes the problem in terms of one surface location and directional solid angles on the hemisphere above it.

To show consistency between the rendering equation and the theory of radiance and light scattering that has been presented previously in this chapter, we will in the remainder of this section show why the rendering equation is an alternative formulation of the hemispherical problem that was presented with respect to reflection in section 3.4.

The invariance of radiance along straight lines gives a cue about why it is sometimes convenient to consider the radiance in terms of surface locations and an integral over all surface areas rather than an integral over the hemisphere as presented in (3.27).

$L(\boldsymbol{x}, \boldsymbol{x}')$ is referred to as transport radiance and mathematically we can define it as:

$$\frac{d^3 Q}{dt \, d\boldsymbol{x} \, d\boldsymbol{x}'} = \frac{d^2 \boldsymbol{\Phi}}{d\boldsymbol{x} \, d\boldsymbol{x}'} = L(\boldsymbol{x}, \boldsymbol{x}') \tag{3.51}$$

Combining (3.51) with (3.13) we will according to (3.49) retrieve the following connection between transport radiance and directional radiance:

$$L(\boldsymbol{x}, \boldsymbol{\omega}) = \frac{1}{\cos\theta} \frac{d^2 \boldsymbol{\Phi}}{d\omega \, d\boldsymbol{x}} = \frac{r^2}{\cos\theta \cos\theta'} \frac{d^2 \boldsymbol{\Phi}}{d\boldsymbol{x} \, d\boldsymbol{x}'} = \frac{r^2}{\cos\theta \cos\theta'} L(\boldsymbol{x}, \boldsymbol{x}') \tag{3.52}$$

Suppose we are considering the outgoing radiance from the differential surface area centered around $\boldsymbol{x}'$. Inserting the results from (3.49) and (3.52) in (3.46) we retrieve the following version of the rendering equation:

$$\begin{aligned} L_i(\boldsymbol{x}, -\boldsymbol{\omega})\frac{\cos\theta\cos\theta'}{r^2} &= g(\boldsymbol{x},\boldsymbol{x}')\Bigg[L_e(\boldsymbol{x}',\boldsymbol{\omega})\frac{\cos\theta\cos\theta'}{r^2} \\ &\quad + \int_\Omega \rho(\boldsymbol{x},\boldsymbol{x}',\boldsymbol{x}'')L_i(\boldsymbol{x}',\boldsymbol{\omega}')\frac{\cos\theta'\cos\theta''}{r^2}\frac{r^2}{\cos\theta''}d\omega'\Bigg] \\ \Rightarrow L_i(\boldsymbol{x}, -\boldsymbol{\omega}) &= g(\boldsymbol{x},\boldsymbol{x}')\Bigg[L_e(\boldsymbol{x}',\boldsymbol{\omega}) + \int_\Omega f_r(\boldsymbol{x}',\boldsymbol{\omega},\boldsymbol{\omega}')L_i(\boldsymbol{x}',\boldsymbol{\omega}')\cos\theta'\,d\omega'\Bigg] \end{aligned}$$

where $f_r(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega}')\cos\theta\cos\theta'/r^2 = \rho(\boldsymbol{x},\boldsymbol{x}',\boldsymbol{x}'')$.

Now, if we wish to find the exitant (or outgoing) radiance at location $\boldsymbol{x}'$ rather than the radiance incident at location $\boldsymbol{x}$, all radiance will be calculated in the same point and in that case we need not worry about the visibility term ($g(\boldsymbol{x},\boldsymbol{x}) = 1$). Then it follows that:

$$\begin{aligned} L_o(\boldsymbol{x}',\boldsymbol{\omega}) &= L_e(\boldsymbol{x}',\boldsymbol{\omega}) + \int_\Omega f_r(\boldsymbol{x}',\boldsymbol{\omega},\boldsymbol{\omega}')L_i(\boldsymbol{x}',\boldsymbol{\omega}')(\boldsymbol{n}\cdot\boldsymbol{\omega}')\,d\omega' \\ &= L_e(\boldsymbol{x}',\boldsymbol{\omega}) + L_r(\boldsymbol{x}',\boldsymbol{\omega}) \end{aligned} \tag{3.53}$$

where it should be noted that L_r is exactly the same as the reflected radiance described in (3.27).

Comparing transport radiance and directional radiance, we know from radiometry that directional radiance most closely resembles what the eye perceive as color, the *hemispherical formulation* of the rendering equation (3.53) is therefore better to use than Kajiya's original formulation (3.46). Still it might be useful to calculate the reflected radiance by integration over surface areas rather than the hemisphere. Therefore the rendering equation has an *area formulation* as well. The area formulation is derived by substitution of (3.49) in the integral representing the reflected radiance:

$$L_o(\boldsymbol{x},\boldsymbol{\omega}) = L_e(\boldsymbol{x},\boldsymbol{\omega}) + \int_A f_r(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega}')L_i(\boldsymbol{x},\boldsymbol{\omega}')g(\boldsymbol{x},\boldsymbol{y})\frac{(\boldsymbol{n_x}\cdot\boldsymbol{\omega}')(\boldsymbol{n_y}\cdot-\boldsymbol{\omega}')}{r^2_{\boldsymbol{xy}}}\,dA_{\boldsymbol{y}}$$

For consistency with other texts (eg. [30]) we will rename the visibility function to $V(\boldsymbol{x},\boldsymbol{y}) = g(\boldsymbol{x},\boldsymbol{y})$ and define a geometry term $G(\boldsymbol{x},\boldsymbol{y})$:

$$G(\boldsymbol{x},\boldsymbol{y}) = \frac{(\boldsymbol{n_x}\cdot\boldsymbol{\omega}')(\boldsymbol{n_y}\cdot-\boldsymbol{\omega}')}{r^2_{\boldsymbol{xy}}}$$

The resulting area formulation of the rendering equation is then:

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}) = L_e(\boldsymbol{x}, \boldsymbol{\omega}) + \int_A f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') V(\boldsymbol{x}, \boldsymbol{y}) G(\boldsymbol{x}, \boldsymbol{y})\, dA_{\boldsymbol{y}} \quad (3.54)$$

Having described the rendering equation all there is left to do is to solve it. Consider, however, the recursive nature of the reflected radiance term, this makes the rendering equation impossible to solve analytically. The following section on light transport theory will investigate the recursive reflection term and introduce a few integral operators for ease of notation.

3.6 Light Transport

During this chapter we have described a mathematical model through which the illumination of an environment can be simulated. Radiance is the quantity that we are interested in and the rendering equation states how radiance is calculated at an arbitrary location in an arbitrary direction.

Evaluation of the rendering equation is, however, not straight forward. In the remainder of this section we shall take a closer look. Suppose we are looking at a surface location $\boldsymbol{x}$. According to the invariance of radiance along straight paths (3.50) the radiance reaching the eye, that is, the color we see, is equivalent to the exitant radiance at $\boldsymbol{x}$ in the direction towards the eye. This quantity is exactly what the rendering equation (3.53) finds. The emission term L_e is only non-zero if the object is a light source. Calculation of L_e is described in section 3.2. The reflected radiance term L_r finds the hemispherical-directional contribution (see figure 3.10b), which is the sum of all radiance incident over the hemisphere. Now, according to the invariance of radiance along straight paths, each radiance incident from the direction $\boldsymbol{\omega}'$ on the hemisphere is equivalent to the exitant radiance at the first point intersected if we trace a ray backwards in the direction $\boldsymbol{\omega}'$. This exitant radiance is again found by evaluation of the rendering equation and so on and so forth. The recursion becomes apparent, since $L_i(\boldsymbol{x}, \boldsymbol{\omega}')$ in the rendering equation is substituted by $L_o(\boldsymbol{y}, -\boldsymbol{\omega}')$, which is again substituted by the rendering equation evaluated at $\boldsymbol{y}$ in the direction $-\boldsymbol{\omega}'$.

To handle the rendering equation more concisely, we can introduce a few linear integral operators[7]. Operator notation is a convenient way to describe light transport. We can define a *scattering operator* $\mathcal{S}$ ([17]), which represents the integral in (3.27) finding reflected radiance from incident radiance. $\mathcal{S}$ transforms a certain (incident) radiance distribution, L_i, over all points and directions to another distribution, L_r, that gives the reflected radiance values after one reflection [30]. This means that $\mathcal{S}$ depends on all the different BRDFs defined for different materials in a scene.

[7] Integral operators are a special case of the *transformers* introduced in array theory (cf. sec. 2.1).

Having the scattering operator $\mathcal{S}$ at our disposal we can rewrite the hemispherical version of the rendering equation using the functional approach described in section 2.1, which results in the following concise form:

$$L_o = L_e + \mathcal{S}L_i \tag{3.55}$$

The result of $\mathcal{S}$ applied to a radiance distribution L_i over all points and directions ($A \times \Omega$) is effectively a hemispherical-directional reflectance distribution function (HDRDF) applied to all points and directions.

Because of (3.50) another operator called the *propagation operator* $\mathcal{P}$, transforming an exitant radiance distribution to an incident radiance distribution, can be defined. In light of the propagation operator the rendering equation can be written as:

$$L_o = L_e + \mathcal{S}\mathcal{P}L_o \tag{3.56}$$

where it shows immediately that the rendering equation is recursive.

For convenience we define a new operator representing the composition of scattering and propagation $\mathcal{T}_o = \mathcal{S}\mathcal{P}$ called the *exitant transport operator*. We can now expand the rendering equation as a Neumann series:

$$\begin{aligned} L_o &= L_e + \mathcal{T}_o L_o \\ &= L_e + \mathcal{T}_o(L_e + \mathcal{T}_o L_o) \\ &= L_e + \mathcal{T}_o(L_e + \mathcal{T}_o(L_e + \mathcal{T}_o L_o)) \\ &= L_e + \mathcal{T}_o L_e + \mathcal{T}_o^2 L_e + \mathcal{T}_o^3 L_o \\ &= \sum_{n=0}^{\infty} \mathcal{T}_o^n L_e \end{aligned} \tag{3.57}$$

where the superscript n denotes that the outgoing radiance has been scattered n times. The path of light through a scene expressed by the propagation, scattering, and transport operators is illustrated in figure 3.14.

The physical interpretation of (3.57) is that we can split the rendering equation up into components of radiance that has scattered once, twice, thrice, etc. [66] (each scattering is often referred to as a light bounce). This observation is the most important point in this section. It indicates that we can describe the illumination of a scene by means of the light's path through the scene from the light source to the eye point.

Light Transport Notation

When light interacts with matter the resulting scattering is highly dependant on the material properties. Since we know that the rendering equation is the sum of contributions from different light paths (cf. (3.57)), classification of

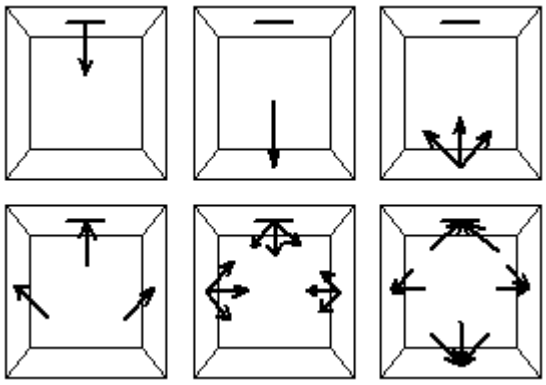


Figure 3.14: A simple illustration of light transport. From top left to bottom right: L_e, $\mathcal{P}\,L_e$, $\mathcal{SP}L_e$ or $\mathcal{T}_oL_e$, $\mathcal{PT}_oL_e$, $\mathcal{T}_o\mathcal{T}_oL_e$ or $\mathcal{T}_o^2L_e$, and $\mathcal{PT}_o^2L_e$. The sequence could be continued indefinitely.

light paths according to the materials they have interacted with along the way would indeed be useful. Paul S. Heckbert [48] introduced a notation, now called the *light transport notation*, where:

- `L` is a light source
- `E` is the eye (or the virtual camera)
- `S` is a specular surface
- `D` is a diffuse surface

Juxtaposition of these symbols defines a light path, for example, `LDE` denotes radiance from a light source reflected diffusely once before reaching the eye, and `LDSE` is a diffuse surface lit by a light source and seen by reflection on a specular surface (such as a mirror).

A class of light paths is defined using the following symbols in combination with the letters given above [60, p. 31]:

- `(k)+` one or more `k` events
- `(k)*` zero or more `k` events
- `(k)?` zero or one `k` event
- `(k|k')` a `k` or a `k'` event

meaning that `L(S|D)*E` is a class in which all light paths fit.

The advantage of light transport notation is that some visual effects are often due to a certain class of light paths. If a method for the simulation of one specific visual effect is found, we can split `L(S|D)*E` into subclasses for

different visual effects and use different methods for their solutions. Only we must make sure never to add the same light path into the integral more than once. An example is a method for mirror effects. Mirrors are simulated by the light paths given as `L(S|D)*SE`. If a specific method is used for mirror effects other rendering methods used for the same scene are only allowed to add light paths of the class `L(S|D)*DE` to the integral of the rendering equation.

Light transport notation also gives a concise way to describe which part of the rendering equation that a specific rendering method can solve. Classic ray tracing, for example, only solves the paths given as `LD?S*E`, while classic radiosity solves the paths `LD*E` only. Ray tracing and radiosity are described in chapter 4.

Reciprocity of the BRDF

According to Nicodemus et al. [88, p. 40] (and many later sources) it is a fundamental property of the BRDF that, "by Helmholtz reciprocity, which holds in the absence of polarization and magnetic fields", the BRDF is independent of the direction in which light flows. That is, we may swap the incident and exitant directions and the BRDF remains unchanged:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = f_r(\boldsymbol{x}, \boldsymbol{\omega}', \boldsymbol{\omega}) \tag{3.58}$$

This property of the BRDF is quite important since it allows us to solve the rendering equation by tracing the radiance backwards through the scene. The propagation and scattering is, according to the invariance along straight paths and the reciprocity respectively, the same as that of figure 3.14 only, when tracing the radiance backwards, we start at the eye point instead of the light source. This gives rise to a dual understanding of the light transport problem which is briefly introduced in the remainder of this section.

Importance

If we trace radiance backwards through the scene, we cannot know the exact value of the radiance in the same way as we can calculate the initial emission term before tracing radiance from the light source. Instead we know that the radiance reaching the eye is 100% of interest to us. In [99] Pattanaik and Mudur introduced the term *importance* W, which has been given several definitions in the literature, it is, however, commonly defined as "the fraction of radiance that contributes (directly or indirectly) to the region of interest" [17, p. 4], and we shall use it as such. Meaning that the radiance reaching the eye has the importance $W_e(\boldsymbol{x}_{\text{eye}}, \boldsymbol{\omega}) = 1$.

Importance has the exact same properties as radiance, only it propagates in the opposite direction and solves the directional dual problem. Importance also has an exitant (W_o) and an incident (W_i) representation, and according

to the invariance of radiance along straight paths it is given, if $\boldsymbol{x}$ and $\boldsymbol{y}$ are mutually visible, that:

$$W_i(\boldsymbol{x}, \boldsymbol{\omega}) = W_o(\boldsymbol{y}, -\boldsymbol{\omega}) \tag{3.59}$$

Suppose we define a set $S \subset A \times \Omega$, where A denotes all the surface points in a scene and Ω denotes the hemisphere, such that each element $(\boldsymbol{x}, \boldsymbol{\omega}_{\text{eye}}) \in S$ is a point $\boldsymbol{x}$ seen directly from the eye. $\boldsymbol{\omega}_{\text{eye}}$ is the direction towards the eye at the point $\boldsymbol{x}$. Then according to (3.59) $L_o(S)$ fully contributes to the region of interest, and we can write:

$$W_e(\boldsymbol{x}, \boldsymbol{\omega}) = \begin{cases} 1 & \text{if } (\boldsymbol{x}, \boldsymbol{\omega}) \in S \\ 0 & \text{if } (\boldsymbol{x}, \boldsymbol{\omega}) \notin S \end{cases} \tag{3.60}$$

Then, in light of (3.58), (3.59), and (3.60), we can define an equation for calculation of incident importance, which is quite similar to the rendering equation. This equation is called the *response function* (or potential function):

$$W_i(\boldsymbol{x}, \boldsymbol{\omega}) = W_e(\boldsymbol{x}, \boldsymbol{\omega}) + \int_{\Omega_x} f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') W_o(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta' d\omega' \tag{3.61}$$

where $\boldsymbol{n}$ is the surface normal at $\boldsymbol{x}$, and Ω_x denotes all the directions over the hemisphere from where radiance was incident at the point $\boldsymbol{x}$.

In exactly the same way as we did previously with the rendering equation, we can introduce an *incident transport operator* $\mathcal{T}_i = \mathcal{PS}$ and write (3.61) in the following way:

$$W_i = W_e + \mathcal{T}_i W_i \tag{3.62}$$

Though we will not go into the theory of adjoint operators, we will briefly mention that if we let $\mathcal{O}^*$ denote the adjoint of $\mathcal{O}$, then $\mathcal{T}_i = \mathcal{PS} = \mathcal{P}^*\mathcal{S}^* = (\mathcal{SP})^* = \mathcal{T}_o^*$, which shows that the incident transport operator is the adjoint of the exitant transport operator. For more information on adjoint operators in rendering we refer to [17] and [30].

Importance is interesting since it gives us the opportunity to start a rendering algorithm both at the light source and at the eye point, and to let importance and radiance meet midway. We shall exploit this notion in the rendering method that we develop in part III.

This concludes the theory that we wish to present on light transport. In the following section we shall briefly describe the local and global illumination models in terms of the theory that has been presented in this chapter.

3.7 Local vs. Global Illumination

As described introductorily in chapter 1 a local illumination model treats each point in a scene as an isolated case where exitant radiance is dependant on the locations of light sources and on the location of the eye only. In terms of the rendering equation:

$$L_{\text{local}}(\boldsymbol{x}, \boldsymbol{\omega}_{\text{eye}}) = L_e(\boldsymbol{x}, \boldsymbol{\omega}_{\text{eye}}) + \sum_{i=0}^{n-1} f_r(\boldsymbol{x}, \boldsymbol{\omega}_{\text{eye}}, \boldsymbol{\omega}_i') L_e(\boldsymbol{y}_i, \omega_i')(\boldsymbol{n} \cdot \boldsymbol{\omega}_i')$$

where n is the number of light sources, $\boldsymbol{y}_i$ is the position of the i'th light source, $\boldsymbol{\omega}_i'$ is the direction towards the i'th light source, and $\boldsymbol{n}$ is as always the surface normal at the location $\boldsymbol{x}$.

The assumption that each point is independent of the surrounding geometry results in some serious short-comings of local illumination.

In terms of light transport local illumination handles the light paths `L(S|D)?E` only. Also note that local illumination approximate light sources as single points in space. To get a more correct result the light sources ought to be modeled as surface areas, and then the area formulation of the rendering equation should have been used instead. The result is that the visibility term has been left out in local illumination, which has the effect that scenes lit with a local illumination model do not feature shadows.

Local illumination models are useful for simple shading only. Their advantage is that they are computationally very inexpensive. Until recently local illumination supplemented by an approximative shadow algorithm was the de facto standard in most applications featuring real-time rendering.

Global illumination is the term used for models that approximate the rendering equation more closely than local illumination does. The rendering methods for *full* global illumination simulates all classes of light paths exactly once: `L(S|D)*E`. Most global illumination methods, however, only simulate a subset of those light paths.

As mentioned before classic ray tracing simulates the paths `LD?S*E`, while classic radiosity simulates `LD*E`. Later approaches expand these techniques or combines them, in the latter case they are called hybrid methods.

A hybrid method for simulation of full global illumination was exactly what led Heckbert to his very useful light transport notation, see [48]. Heckbert's method has a light pass shooting photons into the scene and depositing the photon's power in a radiosity texture, this models the paths `LS*D`. Besides he has an eye pass tracing importance in order to model the paths `DS*E`. Now, progressive refinement of the radiosity textures, that is, progressive reemission of photons from the brightest surface, will repeat the paths `(S*D)` an arbitrary number of times. Combination of the progressively refined light pass and the eye pass will thereby account for all light paths: `L(S*D)*S*E = L(S|D)*E` [48].

Methods such as Heckbert's "adaptive radiosity textures" are also referred to as multi-pass methods. In fact, Heckbert's method closely resembles the widely used multi-pass method called photon mapping, which was developed by Henrik Wann Jensen in [53, 63, 54, 62, 57, 55, 56, 58, 59, 60]. There are important differences, however, and photon mapping will be described in more detail in chapter 4.

The next section will take a superficial look at some of the numerical methods that can be employed in order to solve the problem of infinite recursions, which is inherent in global illumination.

3.8 Solving Recursive Integrals

A Fredholm integral equation of the second kind is of the following form [98]:

$$y(x) = f(x) + \lambda \int_a^b K(x,t)y(t)dt$$

If we compare this form with (3.53) or (3.54), it is obvious that the rendering equation is indeed a Fredholm equation of the second kind, and as such it cannot be solved analytically, except for trivial cases. The option we have left is to use numerical algorithms for the solution of the rendering equation. There are two basic approaches: finite element and point sampling (or Monte Carlo) techniques, all others are derived from them [124].

The basic idea of finite element techniques is to establish a system of linear equations, which has a solution approximating the original integral. More technically the exact solution of the Fredholm equation lives in an infinite-dimensional space. If we project the function (y in the formulation above, L_o in the rendering equation) into a finite-dimensional subspace spanned by some orthogonal basis functions, then an approximative solution for the equation can be derived, see [124] for further details.

Radiosity [40, 18, 94] is based on the finite element approach. The radiosity method (named after the radiometric quantity radiosity, B, which is equivalent to the exitant radiance, M) takes advantage of the fact that reflected radiance and radiosity are *usually* piecewise smooth functions over a surface. If this is the case, they can be approximated by smooth finite-dimensional functions. The problem is that only diffuse materials have these properties and therefore the fundamental assumption in radiosity is that all materials in a scene are perfectly diffuse. This induces serious limitations to the radiosity approach. The advantage, on the other hand, is that the resulting illumination looks smooth and pleasing.

Monte Carlo methods basically approximate the integrand by taking a large number of point samples distributed according to a *probability density function* (PDF). The *phenomenon average*[8] is the average of the sample

[8] A long-term average of data from an underlying random phenomenon, see [104]

points found according to the PDF. Due to the Law of Large Numbers the probability that the phenomenon average is equal to the exact value of the integral converges to unity as the number of samples increases towards infinity, see [30] or [124] for further details.

Distribution ray tracing[9] [20, 133] is based on Monte Carlo techniques. The simple form of distribution ray tracing uses a cosine lobe for sampling over the hemisphere, and rejection sampling if reflection is confined to a certain solid angle by the BRDF at the current surface point. Though Cook et al. [20] in 1984 only had a verbal presentation of the rendering equation[10], their algorithm includes how the following should be done at each position seen from the eye point:

- Sampling of the light sources for shadow calculations.
- Sampling around the mirror direction for reflections.
- Sampling around the direction of transmitted light for refractions.

The algorithm stops at a single level of distributed reflections and refractions to avoid the combinatorial explosion, meaning that it models the light paths `L(S|D)?(S|D)?E`. Ward et al. [133] presents an idea called irradiance cashing, which introduces the assumption of smooth indirect illumination in order to make multiple diffuse interreflections more computationally tractable in distribution ray tracing.

Monte Carlo methods are conceptually simpler than finite element methods, but the large amount of samples needed often results in very slow convergence. Most often the sample points will be too few and variance will be visible as high-frequency noise in the image. Unfortunately the eye is more easily distracted by high-frequency noise [60] as compared to the smoothed images rendered using finite element methods. On the other hand, the systematic errors that can occur in finite element methods (because of the reduction of dimensions in the integral), will not occur in images rendered using Monte Carlo techniques.

In this section we have given a short introduction to the two basic numerical algorithms that can be employed in order to solve recursive integral equations. Besides we have given two examples to illustrate how these algorithms are applicable to the problem of solving the rendering equation. Knowledge of the two basic numerical techniques, supported by the rest of the theory described in this chapter, should, hopefully, make it easier to classify and understand the different rendering techniques that we will present throughout the remainder of this part.

[9] Distribution ray tracing was first called distribut*ed* ray tracing. In [48] Heckbert argued that the term distributed ray tracing was confusing because of the parallel hardware connotations of ray tracing. Therefore the term was changed.

[10] Recall that the rendering equation was not known in the area of computer graphics until Kajiya introduced it in 1986 in [66].

Chapter 4

Traditional Approaches to Realistic Image Synthesis

All art is but imitation of nature.

Lucius Annaeus Seneca (4 BC. - 65 AD.)

In this chapter we will discuss rendering methods that strive for visual realism. In order to render images of a virtual scene in photorealistic quality the rendering method will have to employ a global illumination model (see section 3.7).

As written in section 3.8 there are, in principle, only two fundamentally different methods for rendering of photorealistic images: Ray tracing and radiosity. Both methods have advantages and disadvantages. Neither classic ray tracing nor classic radiosity fully solves the global illumination problem (cf. sec. 3.6), therefore three paths of development have emerged since the ray tracing and radiosity were originally developed. The options are to expand one of the classical approaches or to make a hybrid.

The development of ray tracing expansions has primarily worked in the direction of *path tracing* introduced by Kajiya in [66]. Subsequently multi-pass techniques such as photon mapping have emerged with great success. Expansions of radiosity have worked in the direction towards *stochastic radiosity* [87]. In stochastic radiosity sampling is used to solve the system of linear equations approximating the rendering equation more efficiently. This makes stochastic radiosity a hybrid of the two basic numerical methods for solving the rendering equation (cf. sec. 3.8), still it is initially based on a finite element method. Good examples of hybrid methods are Heckbert's adaptive radiosity textures (cf. sec. 3.8) and a method called *instant radiosity* presented by Keller in [69].

Describing all the methods presented since Whitted and Goral et al. introduced ray tracing and radiosity respectively is far too extensive for this report. A few methods, however, have been implemented for this project and strongly inspired the rendering ideas that we present in part III. Some of our ideas and experiments are even based directly on these methods. In this chapter we, therefore, give a brief introduction to the following rendering methods for realistic image synthesis:

- Classic radiosity is described in section 4.1. Since radiosity has only had a peripheral influence on our final method, we will not go into details on this subject. However, we do have a simple implementation of radiosity in order to create reference images for our own method. The background necessary for this implementation will be described.

- Ray tracing was the original method of choice for this project. Much time has been invested in a complete understanding of classic ray tracing. Some of the ideas that we first came about, strived to optimize ray tracing in various ways (cf. chap. 11). Recall that one of the original objectives of this project was to move methods for realistic image synthesis closer to real-time rendering and vise versa. Ray tracing is given a thorough description in section 4.2.

- Monte Carlo ray tracing is necessary if we want to expand classic ray

tracing so that it can simulate full global illumination. Monte Carlo ray tracing is also an important step towards a good understanding of photon mapping. Section 4.3 presents Monte Carlo ray tracing in short. The Monte Carlo techniques described in this section are also important for some of our own ideas described in part III.

- Photon mapping is a great technique for speeding up Monte Carlo ray tracing. Recent articles have even shown that photon mapping can be simulated in real-time, see especially [74]. Section 4.4 describes photon mapping, which also inspired the method we call direct radiance mapping (see chap. 12).

4.1 Radiosity

When light enters a scene it will bounce around between the objects present. Faster than the eyes can percept it the light will reach a steady state where most of the scene is illuminated. Some objects in the scene will be visible because the light reaches them directly from the source while other objects will be illuminated by indirect lighting, meaning that light has been reflected from other objects. In the real world the indirect light has a surprisingly large influence on the illumination of an environment. A good example is given in [2, p. 277]:

> At night, go into a room and close the blinds and drapes and turn a light on. The reason you can see anything not in line of sight of the light source is because the light bounces off objects in the room.

When light reaches its steady state, the objects in a scene can all be modeled as light sources sending out radiance corresponding to the light that is reflected from them. For a perfectly specular material the light will be reflected in a certain direction according to the law of reflection (see sec. 3.1), but for a perfectly diffuse material the reflected light will be distributed uniformly over the hemisphere above the incident surface location.

As briefly mentioned in section 3.8 the fundamental assumption in radiosity is that all surfaces in the scene are diffuse. The reason for this assumption is that light reflected diffusely is quite smooth and usually does not change suddenly over a surface area. When this is the case, the rendering equation can be given a quite exact approximation by a finite element method (cf. sec. 3.8).

The nature of radiosity leads to both advantages and disadvantages. Radiosity is perfect for the simulation of color bleeding. The easiest way to explain color bleeding is through an example: If a red wall is close to, for example, a white wall you will faintly be able to see red on the white wall,

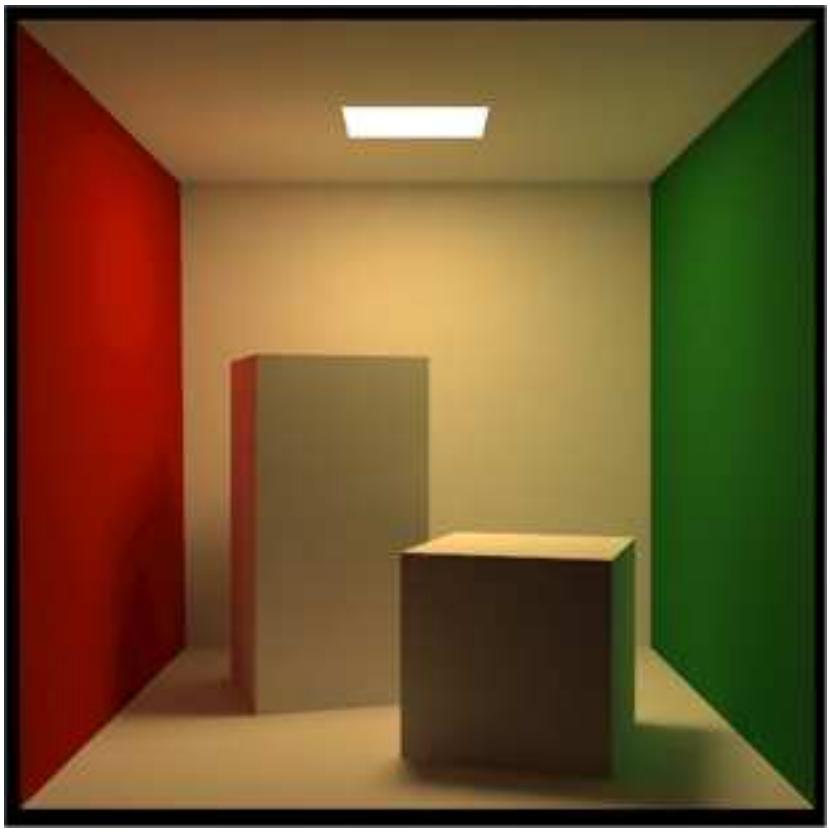


Figure 4.1: The Cornell box is a benchmark scene that physically exists at the Cornell University Program of Computer Graphics. The Cornell box is useful for testing whether different rendering techniques can model diffuse interreflections. The visual effects that should be noticed in this scene are soft shadows, color bleeding (visible as a red or green shade on otherwise white surfaces, especially at the sides of the boxes turning towards the the red and green walls respectively), and the smoothness of the illumination in general. This particular rendering of the Cornell box is a reference image posted at the web page of Cornell University [95].

since the light reflected from the red wall is red. An example of color bleeding can be seen in figure 4.1.

An advantage of radiosity is that it is independent of the camera (or eye) position. This means that as long as all objects and light sources in a scene remain static no recalculation is necessary. Hence, you can move around freely in the environment. Radiosity scenes are therefore sometimes referred to as walkthroughs [90].

Unfortunately it takes quite some time to calculate the radiosity solution. This means that radiosity, like ray tracing, is ill-suited for real-time applications. Furthermore the basic assumption of traditional radiosity excludes rendering of specular objects. To improve on this shortcoming a concept known as *extended form factors*, introduced by Rushmeier in [114, 115],

showed that the system of linear equations can be altered in order to allow ray tracing for the specular elements in a scene and radiosity for the diffuse elements. This enables hybrid methods that can seek to combine the best from both methods.

The Radiosity Equation

As already mentioned all reflecting surfaces in the scene are assumed to be diffuse. This implies that all surfaces emit a constant energy, simulating a constant reflectance of light from a given surface. In other words one surface reflects the same light with the same intensity in all directions. The task is now to calculate the interactions between the reflections.

First step is to divide all objects in the scene into smaller (often rectangular) areas, or patches, where the energy is assumed constant within each patch. The number of patches decides how detailed the light settings will be. In areas where the light intensity is changing fast it is good to have many patches where in constant or slow changing areas many patches are more or less a waste of calculations. One optimization scheme for radiosity is therefore to subdivide patches in areas where the illumination changes suddenly (this would normally be in corners or around spot lights). The starting point would then, of course, be a rough mesh, meaning large patches.

For each patch we then need to find the energy present. The radiosity B (which is equivalent to radiant exitance M) of a patch is the total rate of energy leaving the surface area that it covers. The total energy leaving a patch will be the sum of emitted and reflected light. To determine B we need to use a geometrical unit called a form factor. The calculation of the form factor is the core of the radiosity algorithm. We will go into further detail about the form factor later.

Recall from section 3.2 that the definition of radiosity is:

$$B(\boldsymbol{x}) = \frac{d\boldsymbol{\Phi}}{dA}$$

The assumption that all surfaces are perfectly diffuse makes radiance independent of direction. This means that we can simplify the area formulation of the rendering equation (3.54) by replacement of radiance with radiosity, resulting in the following equation:

$$\begin{aligned} B(\boldsymbol{x}) &= B_e(\boldsymbol{x}) + \int_S f_{r,d}(\boldsymbol{x}) B(\boldsymbol{x}') G(\boldsymbol{x}, \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y}) dA_{\boldsymbol{y}} \\ &= B_e(\boldsymbol{x}) + \frac{\rho_d(\boldsymbol{x})}{\pi} \int_S B(\boldsymbol{x}') G(\boldsymbol{x}, \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y}) dA_{\boldsymbol{y}} \end{aligned} \tag{4.1}$$

where B_e is emitted radiosity and $f_{r,d}(\boldsymbol{x}) = \rho_d(\boldsymbol{x})/\pi$ is the BRDF of a perfectly diffuse material (see (3.32)). ρ_d is the diffuse reflectance indicating how much energy that is reflected back into the scene.

In order to solve (4.1) using a finite element method it is assumed that radiosity is constant over each patch. In that case the radiosity of one patch can be used as a basis element for a system of linear equations. The true radiosity is, in fact, only rarely piecewise constant, but due to linear interpolation across each patch the difference between the approximation B' and the true version B is, in practise, rarely visible.

The interchange of energy between two patches P_i and P_j is a function of the geometrical relationships between them. This includes for example the distance between them and their relative orientation. The more parallel and the closer to each other the two patches are, the higher will the energy exchange between them be.

We can find the radiosity of a patch P_i due to all other patches in the scene, P_j, $j = 1, \ldots, n$, where n is the number of patches in the scene, as [30]:

$$B'_i = B_{e,i} + \rho_i \sum_{j=1}^{n} F_{ij} B'_j \tag{4.2}$$

where B'_k (due to the assumption of constant radiosity over each patch) is the *average* radiosity of patch k, ρ_i is the diffuse reflectance of patch i, $B_{e,i}$ is emitted radiosity of the patch i, and F_{ij} are the patch-to-patch form factors, given as [60]:

$$F_{ij} = \frac{1}{A_i} \int_{S_i} \int_{S_j} \frac{G(\boldsymbol{x}, \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y})}{\pi} dA_j dA_i \tag{4.3}$$

The form factors represent the fraction of energy leaving patch j and arriving at patch i. S_i is the set of points in patch i and S_j is the set of points in patch j. The following three properties of form factors are derived in [30, p. 150]:

1. The form factors are *zero* or *positive* in a scene consisting of closed opaque objects.

2. If the scene is closed:

$$\sum_{j=1}^{n} F_{ij} = 1$$

   If the scene is not closed:

$$\sum_{j=1}^{n} F_{ij} < 1$$

3. The form factors satisfy the following *reciprocity relation*:

$$A_i F_{ij} = A_j F_{ji}$$

Being the most complex element in the radiosity equation (4.1), the calculation of form factors usually determines the rendering time when using radiosity. If we exploit the reciprocity relation given above (property 3) there is an opportunity to halve the number of form factors that we must compute.

The simplest way to calculate an approximated form factor is to assume that the energy transfer between two patches is constant no matter which point $\boldsymbol{x}$ we choose on the patch i, and which point y we choose on the patch j. In that case calculation of the patch-to-patch form factor simplifies to:

$$F_{ij} \approx \frac{1}{A_i} \frac{G(\boldsymbol{x}, \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y})}{\pi} A_j A_i = \frac{G(\boldsymbol{x}, \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y})}{\pi} A_j \tag{4.4}$$

where the center of patch i is chosen as $\boldsymbol{x}$ and the center of patch j is chosen as $\boldsymbol{y}$.

This is one of the approaches that is included in our simple radiosity implementation. Another approach calculates form factors using the *hemicube* method, which is described in the following.

The Hemicube method

The hemicube method is presented in [18] and shortly reproduced here. In order to calculate the form factor F_{ij} between two patches P_i and P_j an imaginary cube is constructed around the center of the receiving patch (P_i). The upper half of the surface of the cube is called the hemicube. The faces of the hemicube are divided into square pixels at a given resolution. The idea is then to project the patch P_j onto the five planar surfaces, see figure 4.2.

Recall, however, that the energy received at the patch P_i is dependent on the location and orientation of the patch P_j. Therefore the contribution of each square pixel to the form factor is also dependent on the pixel location and orientation. For each pixel p_q we can therefore define a delta form factor ΔF_q, and the sum of all the delta form factors will result in the patch-to-patch form factor:

$$F_{ij} = \sum_{q=1}^{R} \Delta F_q \tag{4.5}$$

where R is the number of hemicube pixels covered by the projection of P_j onto the hemicube.

Visibility between the patches P_i and P_j must be taken into account. This is done by determining in advance the patch visible through each pixel p_q. For example a picture can be taken for each face of the hemicube drawing the patches in individual id colors.

After visibility has been determined, the geometry term $G = \cos\theta_i \cos\theta_j / r^2$ (see sec. 3.5) is the only thing left in order to calculate the form factor (cf.

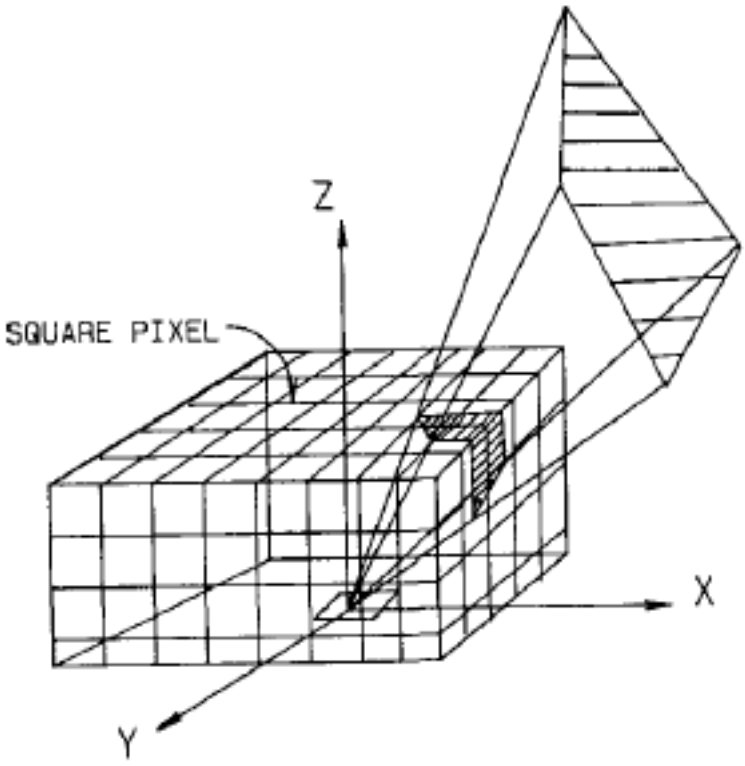


Figure 4.2: The hemicube. This figure is identical to figure 6 in [18]. (Courtesy of M. F. Cohen and D. P. Greenberg.)

(4.3)). The geometry term is calculated quite easily on the hemicube, we must, however, distinguish between the top of the hemicube and the sides of the hemicube. On the top:

$$\begin{aligned} r &= \sqrt{x^2+y^2+1} \\ \cos\theta_i &= \cos\theta_j = \cos\theta \\ \cos\theta &= \frac{1}{\sqrt{x^2+y^2+1}} \end{aligned}$$

meaning that

$$\Delta F_{\text{top}} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2}\Delta A = \frac{1}{\pi(x^2+y^2+1)^2}\Delta A \tag{4.6}$$

where ΔA is the area of a square pixel on the hemicube. On the side of the hemicube:

$$r = \sqrt{y^2+z^2+1}$$

$$\begin{aligned}\cos\theta_i &= \frac{z}{\sqrt{y^2+z^2+1}} \\ \cos\theta_j &= \frac{1}{\sqrt{y^2+z^2+1}}\end{aligned}$$

meaning that

$$\Delta F_{\text{side}} = \frac{\cos\theta_i \cos\theta_j}{\pi r^2}\Delta A = \frac{z}{\pi(y^2+z^2+1)^2}\Delta A \tag{4.7}$$

The hemicube method enables an implementation of radiosity where pictures are taken for each face of a hemicube in order to determine form factors between the receiving patch and all other patches in the scene. Taking pictures makes it possible to exploit the GPU for radiosity computations, which is a great advantage with respect to processing time. The hemicube method is the second and last form factors calculation method that is included in this project.

Progressive Refinement

After a method for the calculation of form factors has been chosen, a progressive refinement method is employed in order to solve the system of linear equations given in (4.2). The solution for this system is the radiosity approximation of the illumination that we wish to find.

Progressive refinement is an iterative process where each iteration represents the propagation of light from the patch that is currently the brightest. This corresponds to application of the integral operator $\mathcal{T}_i$ (cf. sec. 3.6). For each iteration $\mathcal{T}_i$ is applied to the radiosity of the brightest patch in the scene, that is, the brightest patch distributes its energy to every other patch in the scene according to the form factors between them. An iteration thereby simulates diffuse reflection off the brightest patch.

After each iteration the radiosity distributed in the scene is a little closer to the steady state. We can stop the iteration when we see fit, and since all object materials are perfectly diffuse, the radiosity of a patch is also the radiance reaching the eye from that patch (if the patch is not occluded). This is also why radiosity solutions are independent of the viewer's position in the scene.

Having a set of arrays: Final radiosities `B`, 'unshot' radiosities `deltaB`, emitted radiosities `B_e`, areas `A`, and reflectances `R`, holding values for each patch, and moreover a two-dimensional array of form factors `F` holding values for each combination of two patches, the C-like pseudo code for progressive refinement could be given as below:

```
for all i
{
  B[i] = B_e[i];
```

```
  deltaB[i] = B_e[i];
}

while not convergence
{
  pick i, such that deltaB[i] * A[i] is largest (the brightest patch);

  if patch i has not been visited before
  {
    position hemicube over patch i;
    calculate form factors F[i][j], j = 1,...,n;
  }

  for every patch j
  {
    temp_deltaB = deltaB[i] * R[j] * F[i][j];
    deltaB[j] += temp_deltaB;
    B[j] += temp_deltaB;
  }
  deltaB[i] = 0;
}
```

using a few array theoretic functions and the transformer EACH (see definition 2) we can rewrite the pseudo code in a slightly more compact form:

```
B := B_e;
deltaB := B_e;

while not convergence
{
  maxPatch := first sublist [EACH (max (B*A) =), grid] (B*A);

  if maxPatch has not been visited before
  {
    position hemicube over maxPatch;
    calculate the form factors of the array (maxPatch pick rows F);
  }

  [B, deltaB] := [B, deltaB]
    + [pass, pass] ((maxPatch pick deltaB) * R * (maxPatch pick rows F));

  place [0 maxPatch, deltaB];
}
```

where `X := Y` denotes that each element of the array `X` is assigned the value found at the corresponding position in the array `Y`. A description of the array theoretic functions: `first`, `sublist`, `max`, `grid`, `pass`, `pick`, `rows`, and `place` can be found in [102], the meaning should, however, emerge by comparison to the first pseudo code. The point in having an array theoretic formulation is that the for-loops disappear, which reveals the parallel nature of the algorithm. This concludes our description of radiosity. The next subject is classic ray tracing.

4.2 Ray Tracing

The objective of this section is to provide a general understanding of the most basic concepts constituting the global illumination algorithm called ray tracing.

The Basic Concept

The earliest surviving work on geometrical optics, written by Euclid in ancient times [32], initially gives the following description of ray tracing [33]:

> Let it be hypothesized that straight lines drawn out from the eye travel a distance of large magnitudes,
>
> and that the figure enclosed by the sight-lines is a cone having its vertex at the eye and its base at the limits of the things seen,

See also figure 4.3.

The first recursive ray tracing algorithm for computer graphics was (as mentioned before) introduced by Turner Whitted in 1980 [137]. Subsequently many others have tried to improve and develop this approach further, still using the same basic principles.

Consider a point light source spreading light in all directions from a single point in space. If the light is divided into rays expected to follow straight lines in space until they intersect an object (as Euclid hypothesized), each ray can be represented by a point of origin, a direction vector, and a parameter describing how far the light has traversed along the line. Looking at each ray individually (tracing each ray), gives a method to calculate how the light source will illuminate a scene.

Each time a ray intersects an object some light will be absorbed, some reflected and some might be refracted. This means calculation of new rays bouncing off the object or passing through it. All rays ought to be followed until they intersect a Lambertian surface (perfectly diffuse surface)[1], disappear into the background or reach the eye point.

It is easy to imagine that the illumination of a scene with several objects result in a tremendous amount of rays. On top of that most of the rays are not even useful to the viewer since they never reach the eye point. In fact an infinite number of rays would be needed to illuminate the scene perfectly. This is one of the reasons why the process is reversed. Instead of tracing rays that depart from the light source, rays are traced with departure from the eye point (which the Greeks in ancient times thought was reality), meaning that it is only necessary to trace rays that actually have an influence on the visible area. In that way we can remove the useless rays that do not reach the eye point anyway. This approach - calculating the rays from the

[1] This is the case in classic ray tracing, since diffuse interreflections are not modeled.

Figure 4.3: Euclid's "The Optics" is the earliest surviving work on geometrical optics. There were a number of medieval Latin translations, which became of new importance in the fifteenth century for the theory of linear perspective. This technique is beautifully illustrated in the miniature of a street scene in this elegant manuscript from the library of the Duke of Urbino. Text: [32]. Image: [128].

eye point instead of the light source - is called *forward* ray tracing. Other approaches use rays calculated from the light source or from both, these are called backward ray tracing and bi-directional ray tracing respectively. Both approaches are more advanced than forward ray tracing, therefore those will not be considered at this point. Forward ray tracing is pictured in figure 4.4.

Since an eye point, like the point light source, is also just a point in space an infinite number of rays could be traced from it in any given direction. To find the rays that are needed in order to fill the picture of the scene, a view plane (cf. sec. 2.3) is generated. The ray tracer will cast one or more rays from the eye point through each pixel of the view plane. When a ray intersects

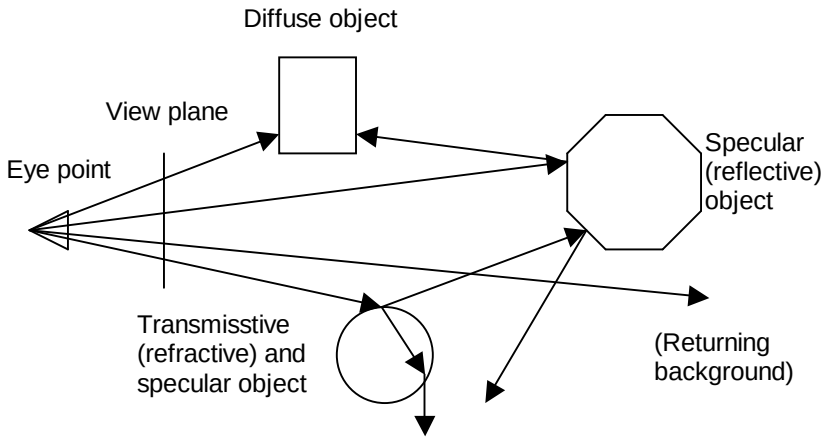


Figure 4.4: Rays are sent from the eye point through a view plane into the scene. Some rays will never intersect anything, others will intersect objects. Perfectly diffuse objects will stop the ray while specular and transmissive objects will generate new rays that continue traveling through the scene. In the end all contributions will be added together generating the image sent to the screen.

an object, new rays might be generated to calculate the contribution of light in the intersected point from the directions of reflection and refraction depending on the material properties of the object. These rays might again intersect new objects that will repeat the same procedure.

The recursion stops when a ray is absorbed by a Lambertian surface, or traced into the background or the light source. The recursion adds together all light contributions until the final color of the pixel in the viewing plane is found. In theory it is possible for rays to keep intersecting with different objects and spawning new rays into infinity. To avoid that possibility a maximum number of intersections is chosen and called the maximum recursion depth. Every time a ray intersects an object one is added to the recursion depth. When a ray reaches the max recursion depth no more rays are spawned from this ray and black is returned, meaning that no further light is added to the final pixel shade.

Since ray tracing generates a tree structure where each new intersection gives a reflection and a transmission leaf, it is ideal for recursive programming [135]. The radiance L_{pixel} incident on a pixel in the view plane is found using the following equation:

$$L_{\text{pixel}} = \int_{\text{viewplane}} L(\boldsymbol{p}, \boldsymbol{\omega}_{\text{eye}}) h(\boldsymbol{p}) d\boldsymbol{p} \tag{4.8}$$

with $\boldsymbol{p}$ being a point on the image plane and $h(\boldsymbol{p})$ a weighting or filtering function [30]. $L(\boldsymbol{p}, \boldsymbol{\omega}_{\text{eye}})$ is the radiance at the point $\boldsymbol{p}$ incident from the first point intersected if a ray is traced from the eye point through $\boldsymbol{p}$, this radiance is normally found recursively:

$$L(\boldsymbol{x}, \boldsymbol{\omega}) = L_{\text{local}}(\boldsymbol{x}, \boldsymbol{\omega}) + k_{\text{reflec}} L(\boldsymbol{x}, \boldsymbol{\omega}_r) + k_{\text{trans}} L(\boldsymbol{x}, \boldsymbol{\omega}_t) \tag{4.9}$$

where $\boldsymbol{\omega}_r$ and $\boldsymbol{\omega}_t$ are the directions of reflection and refraction respectively. k_{reflec} and k_{trans} are material specific parameters that describe the quantity of reflected and transmitted light from the object respectively, hence $k_{\text{reflec}} + k_{\text{trans}} \leq 1$. If the index of refraction η_1 for the material in which the ray is traveling and the index of refraction η_2 for the material that is intersected are known, the quantity of reflected and transmitted light can be found using the Fresnel reflectance described in section 3.4: $k_{\text{reflec}} = F_r(\cos\theta, \eta_1, \eta_2)$ and $k_{\text{trans}} = 1 - F_r(\cos\theta, \eta_1, \eta_2)$, where $\cos\theta = (\boldsymbol{n} \cdot \boldsymbol{\omega})$, and $\boldsymbol{n}$ is the unit surface normal at $\boldsymbol{x}$.

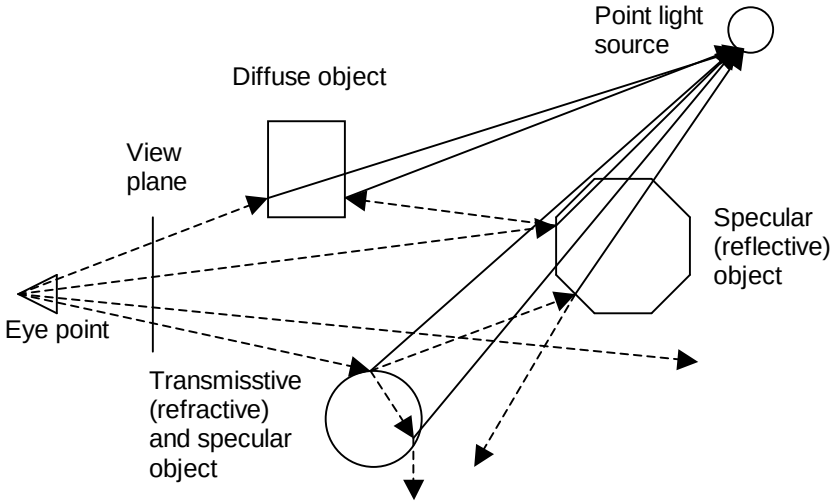


Figure 4.5: Shadow rays are cast from each intersection point in a straight line towards all light sources in the scene (one ray for each light source). If an object is intersected on the way to the light source a shadow will be generated. In this case the specular object will cast a shadow on a part of the transmissive object. Shadows are generated by leaving out direct illumination in the calculation.

Tracing shadow rays from each point of intersection, the ray tracer can easily calculate simple shadows, see figure 4.5. A ray is cast from each intersection point directly towards each light source, if there is an obstacle between a light source and the point of intersection the local (or direct) illumination term L_{local} is left out for that particular light source. Shadow rays generate hard shadows (shadows with a sharp edge). Hard shadows are not as convincing as soft shadows, which are generated eg. by radiosity (see fig. 4.1).

Parts of ray tracing

Ray tracing comprises several steps. This section will identify each step and in the following sections a more in depth description will be given.

The C-style pseudo code for a simple ray tracer is given below. It is build on the ray tracer implemented in this project. Note that the classic ray tracer supports point light sources only.

Pseudo code for a simple ray tracer:

```
void trace_rays() {
  3Dvector shade; // RGB color
  3Dvector viewPlaneNormal, viewPlaneU, viewPlaneV, eyePoint;
  float focalDistance;
  int windowWidth, windowHeight, currentPosX, currentPosY;

  for currentPosX from 0 to windowWidth
    for currentPosY from 0 to windowHeight
    {
      // Generate ray

      2Dvector coords = (currentPosX / windowWidth - 0.5,
                         currentPosY / windowHeight - 0.5);

      3Dvector rayDirection = viewPlaneNormal * focalDistance +
                              viewPlaneU * coords[0] +
                              viewPlaneV * coords[1];

      r = Ray(eyePoint, normalize(rayDirection));

      shade = trace(r)

      Send shaded color to screen;
    }
}

3Dvector trace(Ray r) {
  3Dvector backgroundColor; //RGB color

  if r.intersectionCount > MAX_RECURSION_DEPTH
    shade = BLACK;
  else
  {
    for i from 0 to numberOfObjects
       test for intersection of i'th object with r;

    3Dvector intersectedPos = intersection closest to ray origin;

    if r intersects an object
    {
       3Dvector result = 0;

       Ray shadowray = spawn ray in direction towards the light source;
       trace(shadowray);

       if shadowray.did_hit()
         result += local illumination eg. by Phong using the normal,
```

```
                ray direction, and direction to each light source;

      // Recursive reflection
      if object diffuseness < 1 and object reflectivity > 0
      {
         Ray reflect = spawn ray in reflected direction;
         result += object reflectitvity * trace(reflect);
      }

      // Recursive transmission
      if object transmissivity > 0
      {
         Ray transmit = spawn ray in transmitted direction;
         result += object transmissivity * trace(transmit);
      }

      return (color of material on intersected object * result);
   }
   else return backgroundColor;
  }
}
```

Looking at the pseudo code the different steps of ray tracing are revealed. First of all the ray tracer must be able to generate rays. When a ray is generated there must be detection of intersections, then new rays must be spawn considering reflection, refraction and shadows. In the end shading of the different intersection points must be calculated. This leaves the following list of steps that each ray must go through:

- Ray casting
- Intersection
- Reflection
- Refraction
- Shading

Ray Casting

The first step in ray tracing is to generate rays coming from the eye point through a point in the view plane. The view plane represents the picture that will become visible to the user on the screen. Depending on the demands for image quality one or more rays can be sent through each pixel, as indicated by the integral in (4.8). Since speed of the process very much depends on the number of rays cast less than one ray per pixel can be considered if interpolation between them is used afterwards.

To create a ray we need a point, a direction vector, and a distance parameter. Initially the point will be the eye point later it will be the intersection

point. The direction vectors of the first rays generated are found by drawing a line between the eye point and the point representing a pixel in the viewing plane (simply by subtracting one point from the other). All direction vectors should be normalized. For a ray spawned at a point of intersection the direction depends on whether the ray is the result of a reflection or a refraction, see figure 4.6.

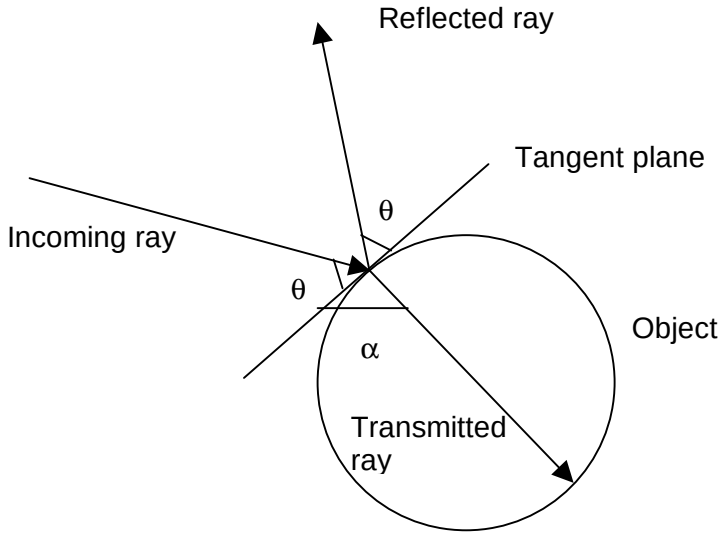


Figure 4.6: New rays are generated from the intersection point (unless the object is completely diffuse). For the reflective ray the incoming angle θ matches the outgoing angle. The angle of the transmitted ray α depends on material properties of the object.

The following equation generates a ray from the eye point. Rays spawned by reflection or transmission are calculated otherwise (cf. sec. 3.1).

Let the eye point be given as $\boldsymbol{e} = (e_0, e_1, e_2)$ and a direction as:

$$\boldsymbol{d} = [\boldsymbol{n}\ \boldsymbol{u}\ \boldsymbol{v}] \begin{bmatrix} f \\ u \\ v \end{bmatrix}$$

where $\boldsymbol{n}$, $\boldsymbol{u}$, and $\boldsymbol{v}$ is an orthonormal frame for the view plane and $[f\ u\ v]^T$ is the focal distance and (u, v)-coordinates specifying a relative position on the view plane. The ray is then given as the parametric equation of a straight line:

$$\boldsymbol{r}(t) = \boldsymbol{e} + t\boldsymbol{d}$$

where t is the distance (or time) parameter describing how far the light has traversed along the line. Tracing a ray, that is, finding the intersected point on the closest visible surface along a ray originating at point $\boldsymbol{x}$ and pointing in the direction $\boldsymbol{\omega}$, can then be described as follows [30]:

$$\begin{aligned} r(\boldsymbol{x}, \boldsymbol{\omega}) &= \{\boldsymbol{y} : \boldsymbol{y} = \boldsymbol{x} + t_{\text{intersection}}\boldsymbol{\omega}\} \\ t_{\text{intersection}} &= \min\{t : t > 0 \wedge \boldsymbol{x} + t\boldsymbol{\omega} \in A\} \end{aligned}$$

where all the surfaces in the scene are represented by the set A. This is called the *ray casting operation.*

Intersection

Intersection between rays and objects is the heaviest part of the ray tracing algorithm. When a ray is traced into the scene there are several things that must be considered. First step is to find out which objects, if any, the given ray intersects. In the brute force manner every ray is checked for collision with every object in the scene, meaning that intersection between each line representing a ray and every object is calculated. If a ray intersects more than one object it is necessary to determine which object is closest, this is done using the distance parameter t (the object within the shortest distance is, of course, the closest and the one intersected).

It is obvious that brute force testing is quite computationally expensive, especially if the objects have a complex structure. To help this, objects are usually defined as combinations of primitives for which we can more easily compute intersections. Besides each object can be surrounded by a bounding box, which is a box fitted to surround the object tightly. When rays are traced through the scene they first check for intersections with the bounding boxes, which is less complex than calculation of intersection with each triangle in the scene. In this way all the objects residing in a box that is not hit by a ray can be left out of the intersection calculations, see figure 4.7.

Since objects are constructed from primitives, we only have to implement collision detection with the primitives represented in our scene. In principle the only primitives we need are a triangle and the shape of the primitive used as bounding volume (in most cases a box). Other primitives that are frequently used in computer graphics are spheres, cones, planes and toruses (see fig. 7.1). The equations that provide intersection between a line and a sphere, a triangle, a box, or a plane are provided later in this section.

When the ray intersects, the next step is to generate a reflected ray and a transmitted ray, depending on material properties. If the object is perfectly diffuse there will be no reflected ray or transmitted ray, otherwise the contribution of light from the reflected and transmitted rays depend on the reflectivity, k_{reflec}, and transmissivity, k_{trans}, parameters of the object material, or more correctly it depends on the Fresnel reflectance given by the refraction indices of the different media through which the ray travels.

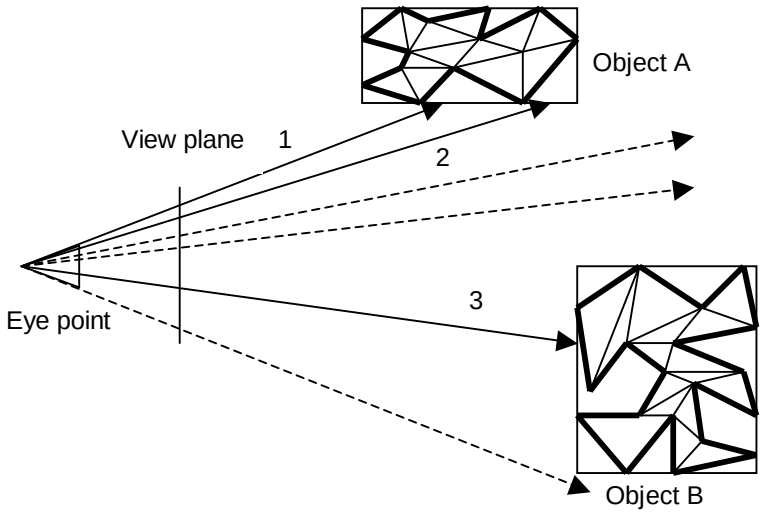


Figure 4.7: Bounding boxes are used to reduce search time for ray intersections. First step is to find intersections between rays and bounding boxes. When an intersection between a ray and a bounding box is identified intersections between the ray and the triangles inside the bounding box can be tested. In this way ray 1 only have to test for intersections with the triangles of object A and ray 3 only for intersections with object B. Ray 2 will have to test for intersections with A as well even though it will not hit any triangles, but only a minimum of rays will be as unfortunate as ray 2. The rest of the rays can be left out, since they do not intersect any bounding boxes, hence, they merely return background color values.

If a ray hits a Lambertian surface or is traced into the background, the recursion stops. This is the end point of the recursive pattern. When an end point is reached an RGB color value is returned according to the color of the Lambertian surface or the background color. Moreover the end point will return no color (black) if the maximum recursion depth is reached. All contributions are added together all the way up to the view plane representing the pixels that are sent to the output window on the screen. Each time a contribution from a reflected or transmitted ray is added to the total contributions the values are multiplied by a percentage factor, representing the reflection or transmission factor of the object. The result is that the higher the recursion depth the less the contribution to the final pixel shade.

Ray/Sphere Intersection

Consider a ray represented by a straight line $\boldsymbol{r}(t)$:

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$$

where t is a parameter indicating the distance traversed from the origin $\boldsymbol{o}$ along the line in direction $\boldsymbol{d}$. $\boldsymbol{r}(t)$ is then a position in space that coincides with the line. A sphere with the center positioned at $\boldsymbol{c} = (c_x, c_y, c_z)$ and with a radius r is represented by the following equation:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$$

or

$$||(x, y, z) - (c_x, c_y, c_z)|| - r = 0 \quad \Rightarrow \quad ||\boldsymbol{r}(t) - \boldsymbol{c}|| - r = 0$$

This simplifies to [2, p. 569].

$$t^2 + 2tb + c = 0$$

where $b = \boldsymbol{d} \cdot (\boldsymbol{o} - \boldsymbol{c})$ and $c = (\boldsymbol{o} - \boldsymbol{c}) \cdot (\boldsymbol{o} - \boldsymbol{c}) - r^2$.

Solving the quadratic equation above gives the following points of intersection between the ray and the sphere, if $b^2 \geq c$, else no intersections exist:

$$t = -b \pm \sqrt{b^2 - c}$$

The ray tracer is only concerned with the first point intersected along the line (the smallest t value of intersection greater than zero). Therefore, if the ray origin is outside the sphere, we choose the intersection $t = -b - \sqrt{b^2 - c}$, otherwise $t = -b + \sqrt{b^2 - c}$ is the correct point of intersection. In case the ray is tangent to the sphere there will be only one point of intersection $t = -b$.

Ray/Triangle Intersection

Concerning polygons we have chosen, in the standard ray tracer, to implement ray/triangle intersection only. Most scenes include only polygons of three and four vertices (triangles and quadrilaterals), and a polygon of four vertices easily simplifies to two triangles.

A point, $\boldsymbol{t}(u, v)$ on a triangle, spanned by the vertices $\boldsymbol{v_0}$, $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$, is given by the following equation:

$$\boldsymbol{t}(u, v) = (1 - u - v)\boldsymbol{v_0} + u\boldsymbol{v_1} + v\boldsymbol{v_2}$$

where $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. u, v, and $w = 1 - u - v$ are called barycentric coordinates for the triangle (see eg. [42]).

Solving the equation $\boldsymbol{r}(t) = \boldsymbol{t}(u, v)$ gives the point of intersection, if any. Denoting $\boldsymbol{e}_1 = \boldsymbol{v}_1 - \boldsymbol{v}_0$, $\boldsymbol{e}_2 = \boldsymbol{v}_2 - \boldsymbol{v}_0$, and $\boldsymbol{s} = \boldsymbol{o} - \boldsymbol{v}_0$ the solution finding t, u, and v is [2, p. 580]:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(\boldsymbol{d} \times \boldsymbol{e}_2) \cdot \boldsymbol{e}_1} \begin{pmatrix} (\boldsymbol{s} \times \boldsymbol{e}_1) \cdot \boldsymbol{e}_2 \\ (\boldsymbol{d} \times \boldsymbol{e}_2) \cdot \boldsymbol{s} \\ (\boldsymbol{s} \times \boldsymbol{e}_1) \cdot \boldsymbol{d} \end{pmatrix}$$

Ray/AABB Overlap

AABB is short for Axis Aligned Bounding Box. When testing rays against bounding volumes it is only necessary to determine whether there is an intersection or not, the exact intersection point is not required, since the ray will be tested against the primitives contained within the bounding volume subsequently. This simplifies the calculation considerably.

Since light only travels in one direction along the line representing the ray, we are looking at the interval $[0, \infty[$ or, choosing a large floating point value to represent infinity, $t \in [0, 10^6] = [t_{\min}, t_{\max}]$.

Consider an AABB described by a two points $\boldsymbol{p}_{\min}$ and $\boldsymbol{p}_{\max}$ and a line segment representing the ray that we wish to test for overlap also described by two points $\boldsymbol{r}_{\min}(t_{\min})$ and $\boldsymbol{r}_{\max}(t_{\max})$. Let $\boldsymbol{h} = (h_x, h_y, h_z) = (\boldsymbol{p}_{\max} - \boldsymbol{p}_{\min})/2$ denote the half vector of the bounding box, let $\boldsymbol{c} = (c_x, c_y, c_z) = (\boldsymbol{r}_{\max}(t_{\max}) + \boldsymbol{r}_{\min}(t_{\min}))/2$ denote center of the line segment, and let $\boldsymbol{w} = (w_x, w_y, w_z) = \boldsymbol{r}_{\max}(t_{\max}) - \boldsymbol{c}$ denote the half vector of the line segment, then the following test will decide whether the ray overlaps the axis aligned bounding box or not [2]:

$$\begin{array}{lrl} & |c_x| & > v_x + h_x \\ \wedge & |c_y| & > v_y + h_y \\ \wedge & |c_z| & > v_z + h_z \\ \wedge & |c_y w_z - c_z w_y| & > h_y v_z + h_z v_y \end{array}$$

$$\begin{aligned} \wedge \quad & |c_x w_z - c_z w_x| > h_x v_z + h_z v_x \\ \wedge \quad & |c_x w_y - c_y w_x| > h_x v_y + h_y v_x \end{aligned}$$

where $(v_x, v_y, v_z) = (|w_x|, |w_y|, |w_z|)$. If this holds true the ray overlaps the bounding box, otherwise the ray and the AABB are disjoint.

Ray/Plane Intersection

Though planes are usually not present in real-time computer graphics, since rendering objects of infinite size is inefficient, we will include them in our ray tracer as the rays are traced into infinity anyway.

The intersection point between a line $\boldsymbol{r}(t)$ representing a ray and a plane, described by a normal $\boldsymbol{n}$ and a point $\boldsymbol{p}$ in the plane, is found solving the following equation:

$$\boldsymbol{n} \cdot (\boldsymbol{r}(t) - \boldsymbol{p}) = 0 \quad \Rightarrow \quad t = -\frac{\boldsymbol{n} \cdot \boldsymbol{p} + \boldsymbol{n} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{d}} \quad , \quad \boldsymbol{n} \cdot \boldsymbol{d} \neq 0$$

If $\boldsymbol{n} \cdot \boldsymbol{d} = 0$ the ray and the plane are parallel.

Reflection

The reflected ray is a new ray generated at a point of intersection. The ray has the exact same properties as any other ray. As mentioned before a ray has a vector describing its direction and a parameter t telling how far the light has traversed along the line. The direction of a reflected ray is determined by the law of reflection (see sec. 3.1). You could say that the reflected ray is a mirrored image of the incoming ray through the normal $\boldsymbol{n}$ of the intersected surface in the plane of incidence, see figures 4.6 and 3.3.

(3.2) can be used to find the direction of the reflected ray $\boldsymbol{d}_r$. For convenience we repeat the equation below:

$$\boldsymbol{d}_r = 2(\boldsymbol{d}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{d}'$$

where $\boldsymbol{d}' = -\boldsymbol{d}$ is the direction from the point of intersection towards the ray origin.

Refraction (or Transmission)

The transmitted ray is the ray refracted through a transparent material. The direction of the transmitted ray depends on object material properties and the angle between the normal $\boldsymbol{n}$ and the direction $\boldsymbol{d}$ of the incident ray. The transmissivity of the material states the percentage of the incoming light that is transmitted. Furthermore the ratio η_1/η_2 between the index of refraction η_1 for the material in which the incident ray was travelling and the index of

refraction η_2 for the material which the ray is refracted into, has an influence on the direction of the refracted ray.

The direction $\boldsymbol{d}_t$ of a transmitted ray can be found using equation (3.5), which is repeated below:

$$\boldsymbol{d}_t = \frac{\eta_1}{\eta_2}((\boldsymbol{d}' \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{d}') - \boldsymbol{n}\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (\boldsymbol{d}' \cdot \boldsymbol{n})^2)}$$

where again $\boldsymbol{d}' = -\boldsymbol{d}$ is the direction from the point of intersection towards the ray origin.

Shading

The final shade of each pixel is found when the recursive algorithm unravels, meaning that each pixel shade is found adding up the contribution from all the rays that was spawned at each point of intersection when that particular ray was traced into the scene from the eye point through a pixel in the viewplane. Local illumination is calculated at each point of intersection using some kind of BRDF (eg. Phong).

Sending the final pixel color to the screen is carried out using a graphics library such as OpenGL or Direct3D.

The next section is a short presentation of Monte Carlo ray tracing. Monte Carlo ray tracing is a necessary step in the direction towards photon mapping. Monte Carlo techniques are also very useful in their own right, they are unfortunately often also quite computationally expensive.

4.3 Monte Carlo Ray Tracing

As shortly described in section 3.8, point sampling is one of the two basic techniques that can be used in order to solve a recursive integral such as the rendering equation. Mathematical techniques that use statistical sampling to simulate phenomenon or evaluate values of functions are called Monte Carlo techniques [30].

Suppose we want to evaluate the following integral using a Monte Carlo method:

$$I = \int_{\Omega_x} f(x)dx$$

where x is a possibly multi-dimensional variable with domain Ω_x. Then a Monte Carlo estimator $\langle I \rangle$ will approximate I by taking a lot of random samples, $g(x_i)$, $x \in \Omega_x$, $i = 1, \ldots, N$, and averaging their contributions [126]:

$$\langle I \rangle = \frac{1}{N}\sum_{i=1}^{N} g(x_i) = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

where $p(x_i)$ is a *probability density function* (PDF) according to which we can draw samples. If all N random variables have the same PDF they are called *independent identically distributed* (IID) variables. In [30, p. 57] it is shown that if all N variables are IID variables, the expected value (or the mean) of the estimator $E[\langle I \rangle]$ equals the desired integral I. The variance of this estimator is given as:

$$\sigma^2 = \frac{1}{N} \int \left(\frac{f(x)}{p(x)} - I \right)^2 p(x) dx$$

meaning that the variance decreases linearly as N increases. The standard deviation σ is the square root of the variance, which means that in order to halve the error in the estimator $E[\langle I \rangle]$ we must quadruple the number of samples. This is the reason for slow convergence of Monte Carlo ray tracing.

General properties of the PDF are:

$$\begin{aligned} \forall x : p(x) &\geq 0 \\ \int_{-\infty}^{\infty} p(x) dx &= 1 \end{aligned}$$

The remaining problem, before we are able to use the Monte Carlo technique for approximation of the rendering equation, is that we must be able to sample according to the PDF that we see fit. In [30, p. 64] it is explained how this is done: "A sample can be generated according to a given distribution $p(x)$ by applying the inverse cumulative distribution function of $p(x)$ to a uniformly generated random variable" ξ over the interval[2] $[0, 1[$. The *cumulative distribution function* (CDF) is defined as [30]:

$$P(y) = Pr(x \leq y) = \int_{-\infty}^{y} p(x) dx$$

Looking at the integral calculating reflected radiance in (3.53) it would be convenient to be able to sample according to a cosine PDF:

$$p(\theta, \phi) = \frac{cos\theta}{\pi}$$

where θ and ϕ are spherical coordinates.

The cumulative distribution function $P(\theta, \phi)$ is derived in [30, p. 66]:

$$P(\theta, \phi) = \frac{\phi}{2\pi}(1 - \cos^2 \theta)$$

[2]We use the notation $[a, b[$, instead of $[a, b)$ to denote that b is not included in the interval.

which is separable with respect to θ and ϕ, resulting in the following formula for computing the spherical coordinates of a direction sampled according to the cosine PDF:

$$(\theta, \phi) = \left(\cos^{-1}(\sqrt{\xi_0}), 2\pi\xi_1\right) \tag{4.10}$$

where $\xi_i \in [0, 1[$, $i = 0, 1$. (With respect to the derivation of (4.10) from the CDF it should be noted in the case of θ that ξ_1 replaces $1 - \xi$, which is possible since $\xi \in [0, 1[$.)

A few other distributions useful for rendering are given in [124, p. 33].

Being able to sample according to the cosine lobe by calculation of new directions using (4.10), we can for example expand the ray tracing described in the previous section by adding a radiance term L_{indirect} to (4.9). L_{indirect} simulates multiple diffuse reflections:

$$\begin{aligned} L(\boldsymbol{x}, \boldsymbol{\omega}) &= L_{\text{local}}(\boldsymbol{x}, \boldsymbol{\omega}) + k_{\text{reflec}} L(\boldsymbol{x}, \boldsymbol{\omega}_r) + k_{\text{trans}} L(\boldsymbol{x}, \boldsymbol{\omega}_t) \\ &\quad + (1 - k_{\text{reflec}} - k_{\text{trans}}) L_{\text{indirect}}(\boldsymbol{x}, \boldsymbol{\omega}) \end{aligned} \tag{4.11}$$

In order to calculate the indirect illumination term N sample rays are traced with directions $\boldsymbol{\omega}_i$, $i = 1, \ldots, N$ found using (4.10). L_{indirect} is estimated using the following Monte Carlo estimator:

$$\langle L_{\text{indirect}} \rangle = \frac{\rho_d}{N} \sum L(\boldsymbol{x}, \boldsymbol{\omega}_i) \tag{4.12}$$

where ρ_d is the diffuse reflectance. This estimator follows since $p(x) = \cos\theta/\pi = (\boldsymbol{\omega} \cdot \boldsymbol{n})/\pi$, and since:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = f_r = \frac{\rho_d}{\pi}$$

when the material is perfectly diffuse, with a constant diffuse reflectance.

The above expansion of classic ray tracing effectively makes ray tracing able to model diffuse interreflections. The convergence is, however, slow. Many sample rays must be distributed and traced recursively in order to obtain a decent evaluation of the indirect illumination term. Photon mapping, which is the subject of the next section, presents a method that significantly can speed up the calculation of the indirect illumination term in Monte Carlo ray tracing.

Another application of of the Monte Carlo technique is that it enables complex light sources (that is, light sources with an areal extension) in ray tracing. In the case of complex light sources the direct illumination term (L_{local} in (4.11)) should be evaluated using a Monte Carlo estimator sampling several rays towards random positions on each light source.

In order to have a renderer that can simulate full global illumination we choose to implement photon mapping, which is the subject of the following section.

4.4 Photon Mapping

Recall Heckbert's idea of adaptive radiosity textures (section 3.7). Heckbert describes a two-pass method in which photons are emitted from light sources, stored in "radiosity textures", and re-emitted from the brightest surface in a progressive refinement manner to account for the light paths: `L(S*D)*`, and rays are emitted from the eye point to account for the paths `DS*E`.

Photon mapping is a similar kind of two-pass method. The most important difference is that when a photon intersects a diffuse surface it is stored in a map which is *independent* of the scene geometry. Instead of the progressive refinement approach for modeling multiple diffuse reflections, each photon is traced through the scene using Russian roulette (see the following section on photon tracing). Since the resulting photon map is independent of geometry we can not use textures for radiance estimates when ray tracing the scene from the eye point. Instead the rendering equation must be approximated by a *radiance estimate*, which is based on the density of photons over a given surface area. All in all photon mapping comprises the following steps:

- Photon tracing
    1. Photon emission.
    2. Photon scattering.
    3. Photon storing
- Forward ray tracing
- The radiance estimate

Each of these steps will be addressed in the following. Our presentation basically follows [60]. Photon mapping in its purest form craves an immense number of photons in order to give a good radiance estimate. After the presentation of the steps given above, we will therefore briefly describe how photon mapping can be combined with other techniques (eg. distribution ray tracing) in order to lower the number of photons needed for an acceptable result. In combination with other methods photon mapping is speeded up significantly. The low processing time combined with the independence of geometry and the compatibility with rendering of participating media currently makes photon mapping one of the most powerful techniques simulating full global illumination.

Photon Tracing

Photons are traced as ordinary rays. They are traced from the light source as in backward ray tracing. The difference lies in how they are scatted and stored in the photon map when intersecting a material. Photon mapping uses a stochastic method rather than the recursive method used in ray tracing.

Photon Emission

Photon mapping can simulate any type of light source. The reason is the stochastic method that photon mapping employs. Each photon is emitted from a random location on a light source in a random direction. One method to use for finding random directions is *rejection sampling*, which uses repeated evaluation of random numbers until a certain property is present. An example is an isotropic point light source emitting light uniformly in all directions. The rejection sampling pick random samples in the unit cube and if the sample is also inside the unit sphere it is normalized and used as a direction for an emitted photon.

Having three random variables $\xi_i \in [0,1]$, $i = 0, 1, 2$ the following C-like pseudo code implements photon emission:

```
int n_e = 0;  // number of photons emitted

while(not enough photons)
{
  do
  {
    x = 2*xi_0 - 1;
    y = 2*xi_1 - 1;
    z = 2*xi_2 - 1;
  }
  while(x*x + y*y + z*z > 1);

  3Dvector d = normalize(3Dvector(x, y, z));
  3Dvector p = random location on light source;
  trace photon from p in direction d;
  n_e++;
}
scale power of photons with 1/n_e;
```

As noted, the power of the photons should be scaled according to the number of photons that have been emitted, which is most probably not the same as the number of photons stored in the photon map.

Photon Scattering

The basic idea of photon scattering is to importance sample a single direction of reflection according to the BRDF (or BTDF in case of transmission), as opposed to distribution ray tracing where numerous rays are spawned at each intersection. The photon may also be absorbed. In that case the recursion stops.

When a photon intersects an object, the type of interaction is decided by the material properties. One way to decide the type of intersection is as follows. Consider a material having diffuse reflectance $(\rho_{d,r}, \rho_{d,g}, \rho_{d,b})$ and specular reflectance $(\rho_{s,r}, \rho_{s,g}, \rho_{s,b})$, where r, g, and b denotes the three color

bands: red, green, and blue. The type of interaction is chosen using Russian roulette (first introduced to computer graphics by Arvo and Kirk [5]). First the average diffuse reflectance $\rho_{d,\text{avg}}$ and the average specular reflectance $\rho_{s,\text{avg}}$ is determined:

$$\begin{aligned} \rho_{d,\text{avg}} &= \frac{\rho_{d,r} + \rho_{d,g} + \rho_{d,b}}{3} \\ \rho_{s,\text{avg}} &= \frac{\rho_{s,r} + \rho_{s,g} + \rho_{s,b}}{3} \end{aligned}$$

Then having a uniformly distributed random variable $\xi \in [0, 1]$, the Russian roulette works as follows:

$$\begin{aligned} \xi \in [0, \rho_{d,\text{avg}}] &\longrightarrow \text{diffuse reflection} \\ \xi \in]\rho_{d,\text{avg}}, \rho_{s,\text{avg}} + \rho_{d,\text{avg}}] &\longrightarrow \text{specular reflection} \\ \xi \in]\rho_{s,\text{avg}} + \rho_{d,\text{avg}}, 1] &\longrightarrow \text{absorption} \end{aligned}$$

(here we use the notation $]a, b]$, instead of $(a, b]$, to denote that a is not included in the interval.)

If the photon is reflected, the power of the reflected photon is scaled according to the spectral reflectance value:

$$\begin{aligned} \Phi_r &= \Phi_{i,r}\rho_r/\rho_{\text{avg}} \\ \Phi_g &= \Phi_{i,g}\rho_g/\rho_{\text{avg}} \\ \Phi_b &= \Phi_{i,b}\rho_b/\rho_{\text{avg}} \end{aligned}$$

where the i subscript denotes the spectral power of the incident photon. ρ and ρ_{avg} are either diffuse or specular reflectance values, which kind to choose depends on the result of the Russian roulette.

The reflected photon will be traced in a direction sampled according to the BRDF. If the reflection is specular another Russian roulette can be used to choose whether the photon is reflected or transmitted:

$$\begin{aligned} \xi \in [0, F_r] &\longrightarrow \text{specular reflection} \\ \xi \in]F_r, 1] &\longrightarrow \text{specular refraction} \end{aligned}$$

where F_r is the Fresnel reflectance (see (3.35), (3.36), and (3.37)).

In case of specular refraction the direction is sampled according to a BTDF rather than a BRDF.

If the material is *perfectly* specular, the photon will be traced in the direction given by (3.2) or (3.5) depending on the result of the Russian roulette.

If diffuse reflection is the result of the Russian roulette, and if the material is *perfectly* diffuse, the reflected direction "is found by picking a random direction on the hemisphere above the intersection point with a probability proportional to the cosine of the angle with the normal" [60, p. 61]. Having two uniformly distributed random variables $\xi_i \in [0, 1[,\ i = 0, 1$, the direction is found in spherical coordinates according to (4.10):

$$(\theta, \phi) = \left(\cos^{-1}(\sqrt{\xi_0}), 2\pi\xi_1\right)$$

The reason why the cosine lobe is used instead of eg. rejection sampling is that the reflected samples will be weighted by the cosine term that also appears in the rendering equation.

Photon Storing

Whenever a photon intersects a diffuse surface (not necessarily perfectly diffuse) it is stored in the photon map. The radiance estimate that can be obtained from a photon map is much too incorrect for the single reflected direction of perfectly specular materials, therefore classic forward ray tracing is used for specular reflections (recall the second pass from the eye accounting for the paths `DS*E`).

When a photon is stored it represents flux incident on a surface. All the photons are stored in a kd-tree, see section 2.5. For a more thorough description of the kd-tree for photon mapping we refer to [60, chap. 6] and to [44] in which they show improved results with respect to the kd-tree. Other spatial data structures could be used as well, as mentioned in section 2.5 Günther et al [44] also report good results with a uniform grid of voxels for the photon map data structure

Forward Ray Tracing

The pass from the eye happens exactly as described in section 4.2. The only thing that should be noted is that L_{local} in (4.9) should be replaced by L_{global}, since the radiance estimate described in the following makes it possible to simulate full global illumination.

The Radiance Estimate

In order to calculate an estimate of L_{global} using the photon map, the rendering equation must be rewritten in terms of flux incident on a surface. (3.24), from which the rendering equation was derived, states the following, if we integrate on both sides of the equation:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) = \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') dE_i$$

Substitution of dE_i according to (3.22) results in the following:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) = \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') \frac{d\boldsymbol{\Phi}_i(\boldsymbol{x}, \boldsymbol{\omega}')}{dA_i} \tag{4.13}$$

which is exactly a way to calculate the recursive part of the rendering equation using the incident flux.

A simple way to approximate the integral in (4.13) is to expand a sphere around $\boldsymbol{x}$ until it contains n photons. Each photon p has the power $\Delta\boldsymbol{\Phi}_p(\boldsymbol{x}, \boldsymbol{\omega}_p)$, where it is assumed that the photon intersects the surface at $\boldsymbol{x}$, and they are incident on the surface area ΔA given as the sphere projected to the surface containing $\boldsymbol{x}$. This results in the following:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) \approx \sum_{p=0}^{n-1} f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}_p) \frac{\Delta\boldsymbol{\Phi}_p(\boldsymbol{x}, \boldsymbol{\omega}_p)}{\Delta A} \tag{4.14}$$

Assuming that the surface containing $\boldsymbol{x}$ is locally planar around $\boldsymbol{x}$, the projection of the sphere to the surface will be a circle of area: $\Delta A = \pi r^2$, where r is the radius of the sphere expanded around $\boldsymbol{x}$. It is shown in [60] that this approximation converges to $L(\boldsymbol{x}, \boldsymbol{\omega})$ when n goes to infinity.

Acquiring the n nearest photons from the photon map is not necessarily computationally cheap. The more photons the better the radiance estimate, and the more expensive is the photon map query. In [73] it is shown that several small photon maps are more efficient than one or two large maps, both with respect to construction and query. It is proposed to make a small individual photon map for each surface in a scene. This, however, results in dependency on scene geometry, which is otherwise one of the advertised advantages of photon mapping.

To see how the nearest photons in the photon map are found when the data structure is a balanced kd-tree, we refer to figure 6.3 in [60, p. 73] or figure 10 in [59].

Splitting Up the Rendering Equation

Using the radiance estimate directly for visualization of the illumination in a scene, will produce quite a lot of low-frequency noise, unless a sufficient amount of photons has been traced. A sufficient amount of photons usually means hundreds of thousands, and sometimes millions. Tracing millions of photons is obviously time consuming, therefore photon mapping is often, and should be, combined with other illumination methods.

In order to justify the combination of different illumination methods, we must split up the rendering equation and make sure that no light paths are included in the integral more than once.

The optimal approach is, of course, to use the method best suited for each visual effect that we wish to include in our illumination model. Direct

illumination and specular and glossy reflections are efficiently modeled by standard Monte Carlo ray tracing, there is no need to simulate these using photon mapping. Since photons emitted from light sources are quickly concentrated in regions with caustics, photon mapping is well suited for simulating caustics. Even when the photon map is visualized directly using the radiance estimate. The best simulation of multiple diffuse reflections (in the context of ray tracing) is a combination of Monte Carlo ray tracing and radiance estimates in a photon map. This method for multiple diffuse reflections is called final gathering, since it gathers radiance from the photon map indirectly.

If we expand the light transport notation with a `G` denoting glossy surfaces, light paths for each of the four parts that the rendering equation should be split into are given as:

- Direct illumination: `L(S|G|D)?E`
- Specular and glossy reflections: `LS*(G|D)(S|G)+E` and `LS+SE`
- Caustics: `LS+(G|D)E`
- Multiple diffuse reflections: `L(S|G|D)*(G|D)S*D(S|G)*E`

Note that these light paths sum to all possible light paths `L(S|G|D)*E`, and that they include all paths only once. Another approach is to classify a glossy material as either specular or diffuse in advance. In that case the paths given above simplifies to:

- Direct illumination: `L(S|D)?E`
- Specular and glossy reflections: `LS*(S|D)S+E`
- Caustics: `LS+DE`
- Multiple diffuse reflections: `L(S|D)*DS*DS*E`

where `S` not necessarily denotes a perfectly specular material an `D` not necessarily denotes a perfectly diffuse material. In order to split up the rendering equation (3.53) accordingly, we split the BRDF as follows:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = f_{r,S}(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') + f_{r,D}(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}')$$

and the incident radiance in the integral of the reflected radiance term as follows:

$$L_i(\boldsymbol{x}, \boldsymbol{\omega}) = L_{i,l}(\boldsymbol{x}, \boldsymbol{\omega}) + L_{i,c}(\boldsymbol{x}, \boldsymbol{\omega}) + L_{i,d}(\boldsymbol{x}, \boldsymbol{\omega})$$

where

- $L_{i,l}(\boldsymbol{x}, \boldsymbol{\omega})$ is direct illumination.
- $L_{i,c}(\boldsymbol{x}, \boldsymbol{\omega})$ is caustics.
- $L_{i,d}(\boldsymbol{x}, \boldsymbol{\omega})$ is indirect illumination reflected diffusely at least once.

In combination these two classifications result in the following version of the reflected radiance term:

$$\begin{aligned} L_r(\boldsymbol{x},\boldsymbol{\omega}) &= \int_\Omega f_r(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega'})L_i(\boldsymbol{x},\boldsymbol{\omega})(\boldsymbol{\omega'}\cdot\boldsymbol{n})d\omega' \\ &= \int_\Omega f_r(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega'})L_{i,l}(\boldsymbol{x},\boldsymbol{\omega})(\boldsymbol{\omega'}\cdot\boldsymbol{n})d\omega' \\ &\quad + \int_\Omega f_{r,S}(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega'})(L_{i,c}(\boldsymbol{x},\boldsymbol{\omega}) + L_{i,d}(\boldsymbol{x},\boldsymbol{\omega}))(\boldsymbol{\omega'}\cdot\boldsymbol{n})d\omega' \\ &\quad + \int_\Omega f_{r,D}(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega'})L_{i,c}(\boldsymbol{x},\boldsymbol{\omega})(\boldsymbol{\omega'}\cdot\boldsymbol{n})d\omega' \\ &\quad + \int_\Omega f_{r,D}(\boldsymbol{x},\boldsymbol{\omega},\boldsymbol{\omega'})L_{i,d}(\boldsymbol{x},\boldsymbol{\omega})(\boldsymbol{\omega'}\cdot\boldsymbol{n})d\omega' \end{aligned}$$

The rest of this section will describe a few possible ways to evaluate each of these integrals.

Direct Illumination

The direct illumination (light paths `L(S|D)?E`) is the simplest term to calculate since it simulates a single reflection only; the direct illumination is incident on a surface location directly from the light sources. If the light sources are isotropic point sources the direct illumination term can be modeled using classic ray tracing. In case of more complex light sources Monte Carlo ray tracing can be used to sample the light sources at each surface location. Rasterization methods (see chap. 5) are a fast option for calculation of direct illumination. Simulating complex light sources is, however, more difficult using rasterization methods. The visual effect resulting from complex light sources is soft shadows. The point light source is a hypothetical object that does not appear in real life, therefore soft shadows is a much more realistic effect than hard shadows resulting from point light sources. This indicates that it is often worthwhile to put a little extra effort in direct illumination calculations, instead of just using some simple shading algorithm (such as Phong [106]).

Specular and Glossy Reflections

Perfectly specular reflections and glossy reflections classified as specular are handled by the term called specular and glossy reflections (light paths `LS*(S|D)S+E`). Currently the best known technique handling such reflections is Monte Carlo ray tracing, where the distribution of rays could be optimized by importance sampling of $f_{r,S}$. Approximative methods based on rasterization is another option (see chap. 6). One popular method is environment mapping, which is also used for real-time rendering in this project.

Caustics

As mentioned before caustics can be simulated by a direct visualization of the photon map. In fact this approach is concurrently the most successful approach to simulation of caustics. In order to evaluate the caustics term (light paths `LS+DE`) individually, only photons that have taken the paths `LS+D` are stored in a separate *caustics photon map*. Radiance estimates using this map is an acceptable evaluation of the caustics term. Only few rasterization methods exist for caustics, one is presented in [130].

Multiple Diffuse Reflections

The multiple diffuse reflections term (light paths `L(S|D)*DS*DS*E`) simulates the effects that we also see in radiosity (an example is fig. 4.1) and then some. No doubt this term is the part of the rendering equation that is most costly to evaluate. Monte Carlo ray tracing could be used to evaluate this term, this would, however, be unnecessarily time consuming. A separate photon map containing only photons that have taken the paths `L(S|D)*DS*D` could be used for the evaluation of the term, this, however, often results in low-frequency noise. The combination of the two methods is an efficient alternative, which is called *final gathering*.

Final gathering evaluates the multiple diffuse reflections term using a *global photon map* containing all stored photons (light paths `L(S|D)*D`). The global photon map is not directly visualized, instead the light paths `DS*DS*E` are simulated by Monte Carlo ray tracing. This means that a ray traced from the eye reaching a diffuse surface will evaluate the multiple diffuse reflections term by sample rays distributed over the hemisphere according to $f_{r,D}$. When a sample ray reach another diffuse surface a radiance estimate from the global photon will be used to simulate the remaining part of the light path. Compared to the approach that directly visualizes a photon map containing only photons reflected diffusely at least once, this approach dramatically decreases the amount of photons needed for the radiance estimate in order to get a decent result. The sampling, however, increases the time needed for the computation.

In part III we describe a rasterization approach that partly simulates the multiple diffuse reflections term. Our method simulate the light paths `LS*DDS*E` using a map storing direct illumination from the light source (and possibly some specular reflections, that is, light paths `LS*D`), therefore it is called *direct radiance mapping*.

The description of the different terms have each referred to rasterization methods that can give an approximate evaluation of the integrals. In order to appreciate such approaches the traditional approach to real-time rendering, which is based on rasterization, will be presented in the next chapter. The extensions (such as those referred to in this section) that brings the ras-

terization approach closer to a better evaluation of the rendering equation will be described in chapter 6.

Chapter 5

Traditional Approaches to Real-Time Rendering

The art of life is a constant readjustment to our surroundings.

Kakuzo Okakura

While photorealistic rendering has its main focus on correctness of the scene and for the rendering to become as close to the real world as possible, the absolute main issue in real-time rendering is the processing time needed for each rendering. Methods for real-time rendering are very much limited due to this issue. Processing time is also the main reason why real-time rendering traditionally is based on local illumination and rasterization rather than global illumination methods such as ray tracing or radiosity. Photorealistic effects in real-time are often based on a very simplified version of the mathematics and physical laws behind the lighting calculations. Sometimes real-time methods even do not consider the physics behind a visual effect, instead they simply fake the effect using whatever model that makes the result look right. This approach is not necessarily bad, but in this project we have emphasis on physical realism. In this chapter we will therefore seek to describe how traditional methods for real-time rendering works and what their limitations are with respect to visual realism. In chapter 6 we will describe how different expansions of the real-time rendering concept approaches a better evaluation of the rendering equation.

The first section of this chapter (5.1) will in short describe the pipeline of real-time rendering.

Having the pipeline at hand, it is described in section 5.2 how objects are usually lighted and shaded in real-time and at which stage in the pipeline this usually takes place.

In section 5.3 it is described how textures are used to indulge visual realism into a rasterized image. When an object is textured it means that an image is 'glued' onto the object.

This concludes a relatively short chapter on traditional real-time rendering. The reason why we do not go into specific details on algorithms and techniques for a complete implementation of the rendering pipeline is that most of the techniques necessary for traditional real-time rendering are implemented in hardware today. This means that everything needed in order to implement traditional real-time rendering is a graphics library such as OpenGL or Direct3D. The graphics libraries exploit a GPU if its available and otherwise they simulate the standard techniques in software.

5.1 The Graphics Rendering Pipeline

Initially we can divide the rendering pipeline into three *conceptual stages*: An application stage, a geometry stage, and a rasterizer stage [2], see figure 5.1. Each of these stages may itself constitute a pipeline. In the following we will describe each of the conceptual stages.

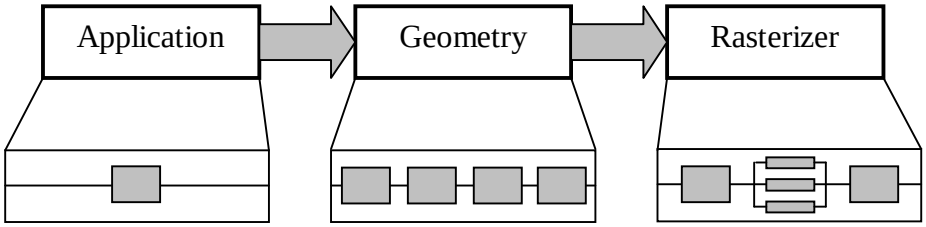


Figure 5.1: The conceptual rendering pipeline. This figure is a reproduction of figure 2.2 in [2].

The Application Stage

The application stage denotes the part of the rendering pipeline that is not build upon hardware implementation. At this stage objects are defined and the scene is composed. The camera and light sources are placed in the scene. The application stage is where user input (from keyboard, mouse, etc.) is handled. Animations are usually also carried out at this stage and object collision may be detected and handled. Extra acceleration algorithms, such as occlusion culling, are also implemented here.

When composing the scene the objects are most often left as they are in their local coordinate system (unless they are morphed or animated). Instead of changing the object data each frame, a transformation matrix is usually associated with each object and send along with it down the pipeline.

At the end of the application stage the scene geometry and the associated transformations should be ready for rendering. This and other rendering primitives (eg. normals, colors, etc.) are fed to the next stage of the pipeline. It should be noted that operations on vertices which are dependent on the rest of the geometry in the scene should be carried out in the application stage. When the geometry passes on to the next stage it is generally assumed that each vertex in the scene can be processed independently. This property makes it much easier for graphics hardware to process the data in parallel.

The Geometry Stage

While the virtual scene travels down the pipeline its geometrical contents will be transformed to reside in different coordinate spaces (they were mentioned briefly in sec. 2.3). When working with real-time rendering it is sometimes important to have a good understanding of the different coordinate spaces. If we need to change things at some point in the pipeline, it is crucial that we know which space we are currently working in. Figure 5.2 shows the different transforms and coordinate spaces that the geometry will pass through.

Usually those transforms are all that happens at the geometry stage. With the emergence of programmable GPU hardware it may, however, well be the case that other operations are carried out before or in-between the

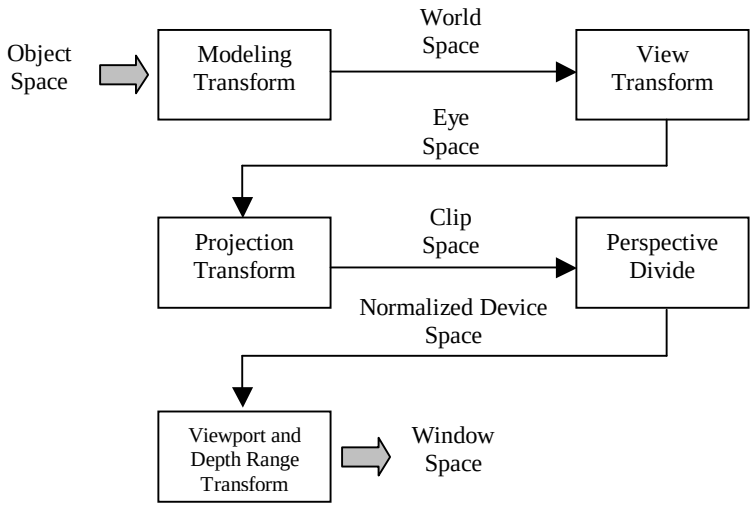


Figure 5.2: The different transforms that will be applied to the geometry and the different coordinate spaces that it will reside in while passing through the geometry stage of the rendering pipeline. This figure is a reproduction of figure 4-1 in [36]

transforms. The part of the geometry pipeline that has been made programmable is the transition from object space to clip space in figure 5.2. If we choose to re-program the vertex processor (which this part of the pipeline is called in GPU terminology), we must keep in mind that modeling, view, and projection transformations are replaced by the vertex program we create. Therefore, if we want the pipeline to behave as usual after adding our modifications to the vertex processor, we must remember to carry out these transformations in our vertex program. The different transformations are described in the following.

Modeling Transform

Each vertex is given in homogenous coordinates (cf. sec. 2.3). The modeling transform is any kind of transformation that the application stage has found useful. An object might have been modeled around the origin of its local coordinate system and the modeling transform can specify which position and orientation the object should have in world space. In this way the modeling transform can also specify movement of objects according to user input (see chap. 9).

View Transform

The view transform is given by the location and orientation of the virtual camera. Having an orthonormal frame $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{N}$ around the eye point $\boldsymbol{E}$ specifying the position and orientation of the view plane, we can describe the rotation matrix $\boldsymbol{R}$:

$$\boldsymbol{R} = \begin{pmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and the translation matrix:

$$\boldsymbol{T} = \begin{pmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

the concatenation of which will describe the view transform:

$$\boldsymbol{T}_{\text{view}} = \boldsymbol{RT}$$

Often the camera parameters provided are an eye point $\boldsymbol{E}$, a 'look-at' point $\boldsymbol{L}$, and a camera up vector $\boldsymbol{v}_{\text{up}}$. If this is the case the orthonormal frame can be found as follows:

$$\begin{aligned} \boldsymbol{N} &= \frac{\boldsymbol{E} - \boldsymbol{L}}{\|\boldsymbol{E} - \boldsymbol{L}\|} \\ \boldsymbol{V} &= \frac{\boldsymbol{v}_{\text{up}}}{\|\boldsymbol{v}_{\text{up}}\|} \\ \boldsymbol{U} &= \boldsymbol{N} \times \boldsymbol{V} \end{aligned}$$

where $\boldsymbol{N}$ points away from the 'look-at' point. The result is a right-handed coordinate system in which the viewer is looking along the negative z-axis, the y-axis points up, and the x-axis to the right.

Projection Transform

The purpose of the projection transform is to transform the view frustum (including its contents) into a unit cube in projective three space. Suppose the view frustum is given by the following six-tuple (l, r, b, t, n, f), where $(l, b, -n)$ is the minimum corner of the view plane, $(r, t, -n)$ is the maximum corner of the view plane, and f is the distance from the eye point to the far plane. Then the projection to the unit cube is given as [2]:

$$\boldsymbol{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The reason why n and f are distances from the eye point rather than z-values in the eye coordinate system, is to make them more intuitive. If n' and f' specified z-values in the eye coordinate system it would be the case that $n' > f'$. Instead it is here, more intuitively, the case that $0 < n < f$.

The projection transform presented here is similar to the one used in OpenGL. It should be noted, however, that some graphics APIs (eg. Direct3D) use a different orientation of their eye coordinate system and if that is the case, the perspective transformations should be changed accordingly.

Perspective Divide

The clip coordinates resulting from the projection transform are still given in homogenous coordinates (x, y, z, w). In order to obtain the actual point or vector, x, y, and z are simply divided by w as described in section 2.3. After this final projection the resulting coordinates are called *normalized device coordinates* [36]. In normalized device coordinates all visible geometry will be located within the unit cube.

Viewport and Depth Range Transform

Finally x and y of the normalized device coordinates are transformed into a coordinate system measured in pixels. This is called *window space* and the transformation is called the *viewport transform*. The z coordinate of the normalized device coordinates is scaled into the range of the depth buffer (or Z-buffer) for use in depth buffering. This is called the *depth range transform* [36]. When the virtual scene is given in window space, with an associated depth value for each pixel in case of depth buffering, everything needed for the rasterizer stage is available.

The Rasterizer Stage

After the geometry stage the rasterizer receives all the polygons, lines, and points, and their associated projected vertices, colors, and texture coordinates. Rasterization is then the process of determining the set of pixels covered by each of the geometrical primitives [36]. A resulting pixel location and its associated depth value and interpolated parameters, such as color, and texture coordinates, are referred to as a *fragment*.

In other words the rasterization breaks each geometric primitive into pixel-sized fragments for each pixel that the primitive covers [36]. When a primitive is rasterized the fragment parameters, such as texture coordinates, are interpolated between the vertices (usually linearly).

The rasterizer is also responsible for resolving visibility. The depth buffer is used for this purpose. A *depth test* ensures that the foremost object is displayed when two objects overlap the same pixel. Suppose z_1 denotes the depth value of the object that was first drawn in the pixel and z_2 denotes

another object overlapping the same pixel, then the default depth test is, of course, $z_2 < z_1$. Graphics APIs, however, makes it possible to choose a different depth test, which is sometimes very useful. The depth test is called a raster operation.

Another commonly used raster operation is the stencil buffer, which is a buffer usually storing an integer value for each fragment. This integer value can be used to mask out parts of the final image. The exact applicability of the stencil buffer and other raster operations will be described in context when we find use for them. Though each of the operations will not be described, the pipeline of raster operations is given in figure 5.3 as a teaser for the options that are available on modern GPUs.

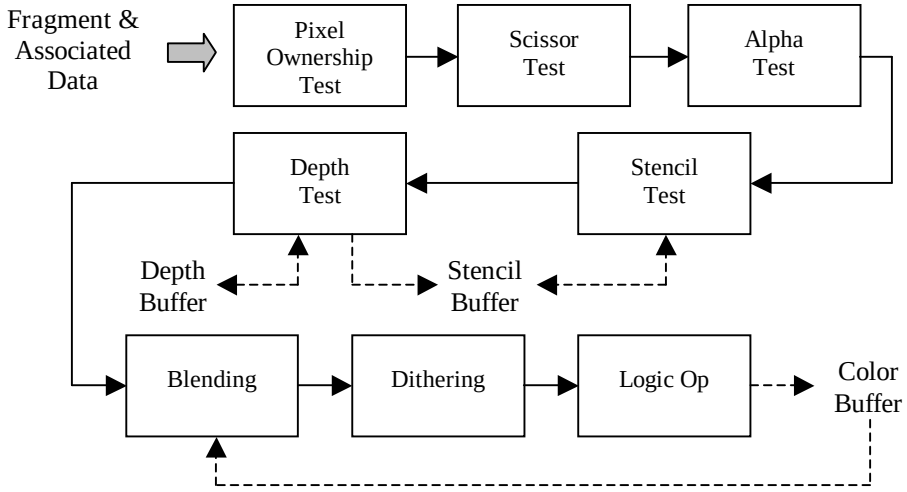


Figure 5.3: Standard OpenGL and Direct3D raster operations. This figure is a reproduction of figure 1-5 in [36].

On fourth generation GPUs (and newer) the fragment processor is also programmable. A *fragment program* is what we use to re-program the fragment processor. The only part of the fragment processing that is replaced by the fragment program is texturing. Otherwise the fragment program only expands the calculations that are carried out in the pipeline. All we must do at the end of a fragment program is to send the color that we found for the current fragment further down the pipeline through the raster operations.

It has been mentioned that a color is associated with each vertex and each fragment. This color is originally provided by the application along with the vertex position, normal, etc. The color may pass unaltered from the vertex processor to the fragment processor. This is called *flat shading*. From there on the color is usually just applied to each fragment that a primitive is covering. The result of a flat shading model is not too exciting, since the entire face of each polygon will have exactly the same color. If the application has provided different vertices for the same primitive object with

different colors, the flat shading model will merely choose the color of the first vertex encountered for the entire primitive. There are fortunately other shading options, some of those will be described in the following section.

5.2 Lighting and Shading

The color at each position of each primitive object is exactly what all the theory given in chapter 3 seek to find. Traditionally it has, however, not been possible to calculate illumination in real-time using the more correct physical model.

Some important models for shaded display were developed long before the rendering equation was presented in [66]. An obvious alternative to the flat shading model is linear interpolation between the vertex colors of each triangle. This method is called *Gouraud shading* named after Henri Gouraud who presented this technique in [41].

Having a better shading model we still need to find the color of each vertex between which we want to interpolate. The simplest shading we can use is the *polyhedral model*, which finds the shade of an object according to the simplest possible form of the rendering equation. Though the polyhedral model is much older than the rendering equation, we can still derive it from the rendering equation and thereby learn about the simplifying assumptions that it includes.

Suppose we only treat diffuse (or Lambertian) surfaces, meaning that the BRDF in each point is given as a constant reflection coefficient $f_{r,d}(\boldsymbol{x}) = \rho_d(\boldsymbol{x})/\pi = k_d(\boldsymbol{x})$. Furthermore we only consider a single point light source, which has no areal extension, meaning that $L_e = 0$ on all surfaces. The light emitted from the point source is given as $L_{e,0} = 1$, and only local illumination is considered. According to all these simplifications we can rewrite the rendering equation as follows:

$$\begin{aligned} L_o(\boldsymbol{x}, \boldsymbol{\omega}) &= L_e(\boldsymbol{x}, \boldsymbol{\omega}) + \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega'}) L_i(\boldsymbol{x}, \boldsymbol{\omega'}) \cos\theta \, d\omega' \\ &= \int_\Omega f_{r,d}(\boldsymbol{x}) L_i(\boldsymbol{x}, \boldsymbol{\omega'}) \cos\theta \, d\omega' \\ &= k_d(\boldsymbol{x}) L_{e,0} \cos\theta \\ &= k_d(\boldsymbol{x}) \cos\theta = k_d(\boldsymbol{x})(\boldsymbol{n} \cdot \boldsymbol{\omega'}) \end{aligned}$$

where θ is the angle between the surface normal, $\boldsymbol{n}$, and the direction towards the light, $\boldsymbol{\omega'}$. This is exactly the formulation of the polyhedral model, and though it is extremely simple, it can create a sensation of depth in an image. Figure 5.4 compares the flat shading model described previously with the polyhedral shading model and the polyhedral model combined with Gouraud shading.

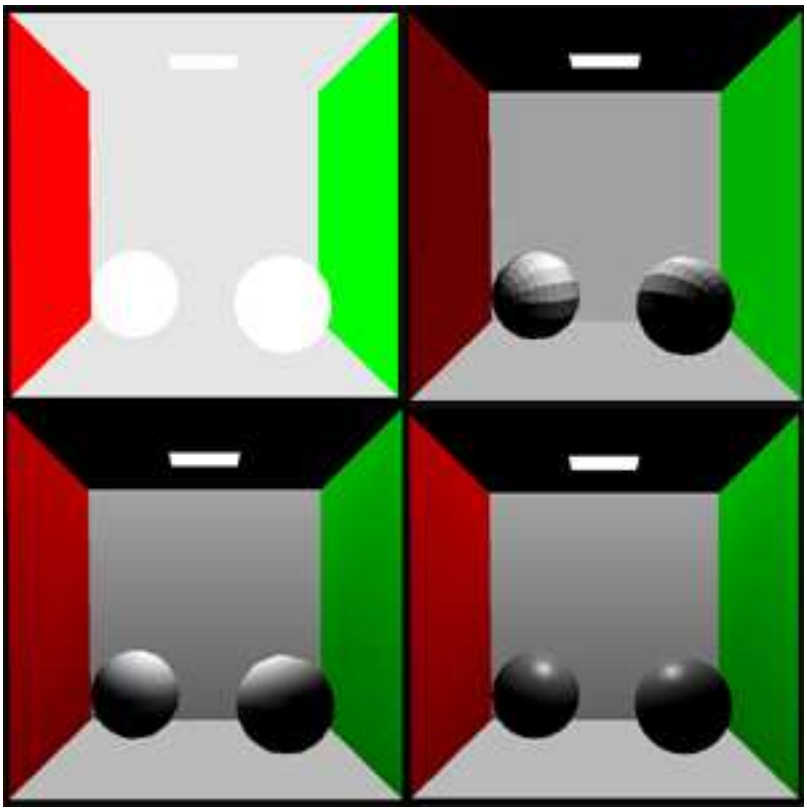


Figure 5.4: Screen shots of the Cornell box using different real-time shading models. The two boxes have been replaced by two spheres to make the shading more readily discernable. From top left to bottom right: (a) Flat shading model. (b) Polyhedral shading model. (c) Polyhedral model and Gouraud shading. (d) Phong highlighting.

In order to expand the polyhedral model to include specular surfaces as well as Lambertian surfaces, Phong published a very famous model in 1975, see [106], which (sometimes in a slightly modified form) is still used for real-time graphics. Quite appropriately it has subsequently been called the Phong model. The best way to describe the Phong model with respect to the rendering equation is to describe the BRDF that he proposed[1]:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = k_s \frac{(\cos\phi)^m}{\cos\theta} + k_d$$

where k_s is a constant specular reflectance coefficient, m is called the *shininess* and $\cos\phi = \boldsymbol{\omega} \cdot \boldsymbol{\omega}_s$, where $\boldsymbol{\omega}$ is direction from the surface location $\boldsymbol{x}$ towards the viewer and $\boldsymbol{\omega}_s$ is the direction of perfect specular reflection (found using (3.2)).

[1] The original article ([106]) did not propose a BRDF, but merely described the model. The fundamental theory for BRDFs were not introduced to computer graphics until years later, even though both theory and nomenclature were available in 1977, see [88].

When the Phong model is combined with Gouraud shading, the result is referred to as *Phong highlighting*. Two white spheres in a Cornell box, half specular ($k_s = 0.5$), half diffuse ($k_d = 0.5$), results in the image presented in figure 5.4d.

Because Gouraud shading interpolates the shade linearly between the vertices, the highlight often becomes coarse. Moreover details in the shading across a surface are sometimes missed if the triangles are large such as the walls in the Cornell box. In order to solve such problems the shading must be evaluated for each fragment rather than for each vertex, which means that all the polygons must be so small that that their shading of each vertex corresponds to shading for each fragment or we must alter the fragment processing in the graphics pipeline. The latter option is often referred to as *fragment shading*, and as noted in the previous section a fourth generation GPU is necessary if we need to program the fragment processor. Evaluating the Phong model for each fragment is referred to as *Phong shading*. The difference between Phong highlighting and Phong shading is presented in figure 5.5.

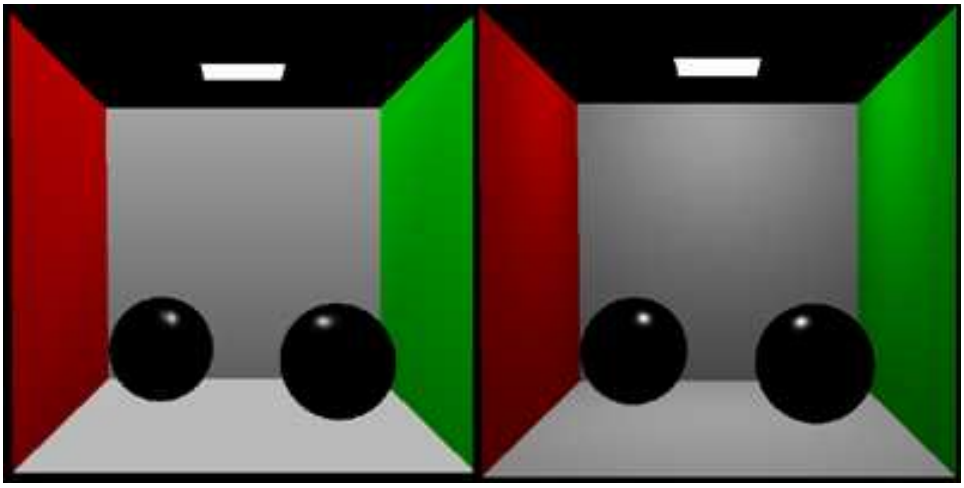


Figure 5.5: The two spheres in the Cornell box are perfectly specular ($k_s = 1$ and $k_d = 0$). From left to right: (a) Phong highlighting. (b) Phong shading.

Unfortunately the Phong model is not energy conserving and neither does it fulfil the reciprocity property of a BRDF, which is a serious disadvantage with respect to light transport (described in 3.6), and it means that the Phong model does not capture the behavior of most real objects [30].

Mainly for efficiency reasons Blinn proposed a popular variation of the Phong model in [9]. In the Blinn-Phong BRDF $\cos\phi = \boldsymbol{n} \cdot \boldsymbol{h}$, where

$$\boldsymbol{h} = \frac{\boldsymbol{\omega} + \boldsymbol{\omega}'}{\|\boldsymbol{\omega} + \boldsymbol{\omega}'\|}$$

is the half-vector between the direction to the viewer $\boldsymbol{\omega}$ and the direction to the light source $\boldsymbol{\omega}'$.

A modified version of the Blinn-Phong model, in fact, preserves the necessary properties of a BRDF. The modified Blinn-Phong BRDF is given as:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = k_s(\boldsymbol{n} \cdot \boldsymbol{h})^m + k_d$$

The relationship between the Phong model and its Blinn-Phong variant was found in [37] as $(\boldsymbol{n} \cdot \boldsymbol{\omega}_s)^m \approx (\boldsymbol{n} \cdot \boldsymbol{h})^{4m}$.

Since all the models described in this section only deal with local illumination, they only simulate the light paths `LDE`, in case of the polyhedral model, or `L(S|D)E`, in case of Phong or Blinn-Phong. To compensate for the missing indirect light an *ambient term*, L_a, which is merely a material constant, is often added to the shade of an object.

All these lighting and shading models were kept primitive and simple to make sure that they would be useful for real-time rendering, and today they still are. The Blinn-Phong model combined with the Gouraud shading (also referred to as Phong highlighting) is the de facto standard in real-time graphics libraries such as OpenGL. Many expansions seeking to obtain a more physically correct lighting and shading in real-time have been proposed since the Phong model, this is the subject of chapter 6.

In the following section we will take a brief look at texturing, since it is an important part of real-time rendering and because textures are useful for many other things than 'gluing' pictures on 3D objects.

5.3 Texture Mapping

Textures are used frequently in almost any kind of professional real-time application. They are a cheap way to enhance a Phong highlighted or Phong shaded scene so that it become more visually interesting. Textures often give a scene the final touch of realism, they are able to add details that would otherwise take an immense amount of triangles to render. In contrast to global illumination techniques (such as those described in chapter 4), texture mapping can be "grafted onto a standard [real-time] rendering method without adding too much to the processing cost" [134, p. 215].

Previously we have mentioned that texture mapping as a way to 'glue' images on 3D objects, and that is indeed the most common functionality of textures. The image is usually two-dimensional and we must make it fit the surface of a 3D object. Points on the surface of an object can be described in two dimensions if they are given in parameter space coordinates, more commonly called (u, v)-coordinates. The texture image is placed in the parameter subspace $[0, 1[\times[0, 1[$. The (u, v)-coordinates are usually provided at the application stage of the rendering pipeline. Graphics librariries (eg. OpenGL) usually also provide automatic texture coordinate generation. The

process of finding parameter space values from locations in space is called *mapping*, hence the name *texture mapping*.

For a texture to be applied to an object, (u, v)-coordinates should be attached to each vertex of the object specifying the part of the texture that ought to be drawn at surface location on the object. Unfortunately it is quite difficult to cover an arbitrary surface cleanly with a two-dimensional image. The texture will most likely be stretched or compressed in places to fit the surface [2]. A number of options are therefore given before a texture is applied. First we must choose how to handle the case where the (u, v)-coordinates reach outside the texture image defined in $[0, 1[\times[0, 1[$. The following options are usually available: *Repeat*, *mirror*, *clamp to edge*, and *clamp to border*. The difference between clamp to edge and clamp to border is that in the first case the edge of the texture is repeated when (u, v)-coordinates reach outside $[0, 1[\times[0, 1[$, while in the second case a separate border color can be defined, which is used for (u, v)-coordinates outside $[0, 1[\times[0, 1[$.

The next thing to worry about is whether the texture fit the number of pixels that the object covers on the screen. The number of pixels that an object covers can change significantly during the course of a real-time application. If the object is sufficiently close to the camera it may fill out all the pixels on the screen, while it may also be so far away that it covers a single pixel only. The first case is called *magnification*, the second is *minification*.

Two filtering methods are usually available in the case of magnification. Those are *nearest neighbor* (or a box filter) and *bilinear interpolation*. Other filters could be used as well. Corresponding filters are available for minification. Anyhow minification seems to cause more aliasing problems than magnification, and it is, therefore, often useful to have a *mipmap* for minification.

Mipmaps were first introduced by Lance Williams in [140]. Mip is an acronym from the latin phrase "Multum In Parvo" meaning "many things in a small place". A mipmap is a pyramidal data structure where the original texture image is the base of the pyramid (level 0). Averaging each 2×2 area of the image into a new texel[2] value creates the next level up. This process continues until the top of the pyramid contains only one texel (it is assumed that the dimensions of the base image are each a power of two). Now an appropriate level of the mipmap can be chosen to represent the texture in the case of minification.

The presentation of texture mapping provided in this section has been very superficial. The reason is that many things done with textures mostly concern anti-aliasing after the image has been stuck on an object, and at the beginning of this project we chose not to work on anti-aliasing in detail.

[2]Texel is short for texture element. A texel in a texture corresponds to a pixel on a screen.

Moreover the ideas that we present in part III do not rely on textures to be stuck on objects, rather they use textures as containers for data that we find useful in a fragment program.

The purpose of this chapter was to provide a very basic introduction to the traditional real-time rendering approach. The rendering method is called rasterization and it is, in fact, older than both ray tracing and radiosity. We have presented it here in its modern form, and described the different stages of the rendering pipeline. The parts of the otherwise hardware implemented pipeline that have been made programmable, have been pointed out. The next chapter describes some of the methods that are used to expand the rasterization approach in order to push real-time rendering closer to realistic image synthesis. These methods often depend on texture mapping, raster operations and the programmable parts of the pipeline (vertex and fragment programs).

Chapter 6

Approximating the Rendering Equation in Real-Time

The closer you get to your goal, the harder it will be to reach it - in fact, you may NEVER reach it.

Paige Waehner: *Exercise for Beginners - Setting Goals*

One objective in this project is to approximate, in real-time, as many as possible of the visual effects that result from a global illumination model. This is not a new objective, many researchers have worked on it before. Many cunning methods and clever algorithms have been developed over the years. Still, not all visual effects are available in real-time, at least not simultaneously and certainly not with the same accuracy as if one of the methods for realistic image synthesis had been used.

To solve the recursive integral of global illumination in real-time (having at most 65 milliseconds to spend for each image) we must either simplify the calculations significantly, or otherwise come up with a simple alternative that visually gives the same result (we may legitimately call this approach 'cheating'). Some of the methods presented in the literature for real-time approximation of global illumination are described in this chapter. We will emphasize on the methods that in some way relate to our idea of direct radiance mapping (see chap. 12), and on methods that can easily be combined with direct radiance mapping in order to compensate for the parts of the rendering equation that it does not solve.

Most articles addressing the subject of this chapter focus on one or a smaller group of visual effects rather than full global illumination. The reason is that one effect alone often is so expensive that it puts heavy restrictions on the resources left for other methods.

The sections of this chapter are divided into different methods. Sections 6.1, 6.2, and 6.3 describe those methods that we combine with direct radiance mapping in the real-time renderer of this project, they are described sufficiently for implementation.

A few methods that could advantageously be combined with our method are presented in sections 6.4 and 6.5. There has not been time for their implementation in this project, and, hence, they are not described in detail. We merely bring the readers attention to the applicabilities of these methods.

The remaining sections (6.6, 6.7, 6.8, and 6.9) are methods described for comparison with direct radiance mapping. Those methods can not be directly combined with our method, rather they could each replace it or some of their conceptual ideas could be used in combination with direct radiance mapping. We can think of them as competitors to the direct radiance mapping method. Therefore we will rather focus on advantages and disadvantages of these methods than on their theoretical foundation.

A description of different methods for subsurface scattering was given in section 3.4. This subject is not addressed any further in this chapter.

6.1 Stenciled Shadow Volumes

Consider the correctness of the images in figures 5.4 and 5.5. One thing we quickly notice to be wrong, is that there are no shadows below the spheres.

The reason is that the shade of each vertex (and fragment) is calculated locally, meaning that other geometry such as occluding objects are not considered when the shade is found.

Looking at the derivation of the polyhedral shading model, the simplifying assumption that loose the shadows is that of local illumination. We can split this assumption in two: Direct illumination only and mutual visibility. The assumption of only direct illumination has the following result on the reflected radiance.

$$\begin{aligned} L_r(\boldsymbol{x}, \boldsymbol{\omega}) &= \int_{\Omega} f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta \, d\omega' \\ &= \int_{\Omega_e} f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta \, d\omega' \end{aligned}$$

where Ω_e denotes the directions over the hemisphere in which a light source is visible. If we consider area light sources, and use the area formulation of the rendering equation instead of the hemispherical formulation the reflected radiance term is given as follows:

$$\begin{aligned} L_r(\boldsymbol{x}, \boldsymbol{\omega}) &= \int_{A_e} f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') V(\boldsymbol{x}, \boldsymbol{y}) G(\boldsymbol{x}, \boldsymbol{y}) \, dA_{\boldsymbol{y}} \\ &= \int_{A} f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_e(\boldsymbol{y}, -\boldsymbol{\omega}') V(\boldsymbol{x}, \boldsymbol{y}) G(\boldsymbol{x}, \boldsymbol{y}) \, dA_{\boldsymbol{y}} \end{aligned}$$

here A_e is the surface area of all light sources in the scene and A is the union of all surface areas in the scene.

The last part of the local illumination assumption is mutual visibility between all surfaces, which means that $V(\boldsymbol{x}, \boldsymbol{y}) = 1$ for all $\boldsymbol{x}$ and $\boldsymbol{y}$. Real-time shadow algorithms seek to avoid this simplifying assumption of mutual visibility. This section presents a fairly general shadow algorithm called *stenciled shadow volumes*.

The original shadow volume method, presented by Franklin Crow in [22], assumes a single light source being either a point light or a directional light. Directional light is a light source removed infinitely far away, which results in light coming from one direction only throughout the scene. In the case of a point light placed at $\boldsymbol{y}$, the rendering equation is only slightly more complex than the polyhedral model:

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}) = f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega_y}) V(\boldsymbol{x}, \boldsymbol{y}) L_{e,0} \, (\boldsymbol{n} \cdot \boldsymbol{\omega_y})$$

where $\boldsymbol{n}$ is the surface normal at $\boldsymbol{x}$ and $\boldsymbol{\omega_y}$ is the direction from $\boldsymbol{x}$ towards the light source placed at $\boldsymbol{y}$.

In case of a directional light the rendering equation is given as follows:

$$L_o(\boldsymbol{x}, \boldsymbol{\omega}) = f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}')V(\boldsymbol{x}, \boldsymbol{\omega}')L_{e,0}\left(\boldsymbol{n} \cdot \boldsymbol{\omega_y}\right)$$

where $\boldsymbol{\omega}' = -\boldsymbol{\omega}_{\text{light}}$, if $\boldsymbol{\omega}_{\text{light}}$ is the direction of the directional light.

The idea proposed by Crow is to let the polygons facing the light source and their contour edges extruded away from the light source define the shadow volume of an object. Surfaces contained within the volume will be in shadow, while those outside are not. The shadow volume is itself invisible, however, if we draw the shadow volume each front facing shadow polygon will tell that everything behind is in shadow, while a back facing polygon will cancel the effect of a front facing one.

Crow's description of shadow volumes (1977) did not find a real-time implementation until 14 years later in [49]. The reason being the difficulties in drawing an invisible shadow volume, which still would be able to specify whether a pixel was in shadow or not. The problem was solved with the emergence of a stencil buffer.

As described in section 5.1 the stencil buffer stores an integer for each pixel in the window. It is possible to use different kinds of masks and stencil functions in order to specify the values to be stored in the buffer.

Suppose we have a shadow volume for each object, then the stenciled shadow volumes algorithm is given as follows:

1. Clear the color, depth, and stencil buffers.

2. Enable writes to the color and depth buffers. Set the depth test to enable drawing when the depth value is *less than* the previous value.

3. Draw the scene with ambient lighting only.

4. Disable writes to the color and depth buffers. Enable writes to the stencil buffer.

5. Draw the front facing shadow polygons. Increment when the depth test passes.

6. Draw the back facing shadow polygons. Decrement when the depth test passes.

7. Disable writes to the stencil buffer. Enable writes to the color buffer.

8. Set the stencil test to enable drawing when the stencil value equals zero. Set the depth test to enable drawing when the depth value is *equal to* the previous value.

9. Draw the scene in full illumination.

Note how the stencil buffer first counts how many times a shadow volume is entered (5.) and afterwards counts how many times a volume is left behind (6.). If the result of the count is greater than zero, the surface area we are looking at must still be in shadow, while if the stencil value is zero after the count has finished the surface area must be free of shadow.

The algorithm can easily be expanded to account for several light sources:

1. Clear the color and depth buffers.
2. Enable writes to the color and depth buffers. Set the depth test to enable drawing when the depth value is *less than* the previous value.
3. Draw the scene with ambient lighting only.
4. For each light l:
   (a) Enable writes to the stencil buffer. Clear the stencil buffer.
   (b) Disable writes to the color and depth buffers. Set the depth test to enable drawing when the depth value is *less than* the previous value.
   (c) Draw the front facing shadow polygons. Increment when the depth test passes.
   (d) Draw the back facing shadow polygons. Decrement when the depth test passes.
   (e) Disable writes to the stencil buffer. Enable writes to the color buffer.
   (f) Set the stencil test to enable drawing when the stencil value equals zero. Set the depth test to enable drawing when the depth value is *equal to* the previous value. Enable additive blending.
   (g) Draw the scene illuminated by light source l only.

In the above we assume that shadow volumes are available. Though it is quite simple to extrude silhouette edges away from the light source to obtain shadow volumes, it is also quite expensive. One relatively efficient way, presented in [2, 12], is to draw each edge as a quadrilateral (or quad) of no width. Two polygons meet where each edge quad is drawn, two vertices of the quad should have normals from one polygon and two vertices should have normals from the other polygon. When drawing the shadow polygons to the stencil buffer, a vertex program can move vertices away from the light source if their normal points away from the direction towards the light. This approach will extrude silhouette edges for shadow volumes.

Compared to other shadow algorithms stenciled shadow volumes are, as mentioned, fairly general. Some shadow algorithms, such as *projection shadows*, can only cast shadows on planar surfaces. For some algorithms,

eg. *shadow textures*, it is necessary to specify which objects that are occluders and which objects that are receivers. An alternative to shadow volumes is *shadow maps* introduced by Lance Williams in [139]. In short the shadow map is a picture from the light source and everything that is not seen in this map is in shadow. The shadow map can, as shadow volumes, cast shadows from arbitrary geometry onto arbitrary geometry. There are different problems associated with each method. The shadow map mostly suffers from aliasing problems, while shadow volumes suffer from problems of a more geometrical nature. Most of the problems in shadow volumes are addressed in [78, 34] and we will not go further into them at this point.

The limitations of shadow volumes are still the simplifying assumptions of only direct illumination and only point or directional light sources. The first assumption is fundamental to all shadow algorithms. In order to remove this assumption we must calculate the mutual visibility between all surface locations in a scene, which is quite infeasible for a shadow algorithm that in its simple form cannot simulate light sources with an areal extension. The latter assumption has been addressed lately by several articles. The point and directional light sources result in hard shadow edges, since they have no areal extension that can make a source partly visible. The visual effect resulting from area light sources is known as soft shadows.

The extension that makes the stenciled shadow volumes algorithm able to simulate soft shadows, is not straight forward. One method, described in [51], uses an extended shadow map called a D-buffer to store umbra and penumbra volumes[1]. Another approach, presented in [7, 8, 1], calculates penumbra wedges and use them for approximation of soft shadows. Unfortunately we have not had time for an implementation of real-time soft shadows during this project. A good presentation of shadow volumes and the penumbra wedges extension is given in [35]. A general overview of the different algorithms can be found in [2].

This section concerned the direct illumination approximation of the rendering equation. Direct illumination include shadows and it is, therefore, better than the local illumination models presented in section 5.2. The next step towards visual realism is specular reflections, which is the subject of the two following sections.

6.2 Planar Reflections Using the Stencil Buffer

Recall from section 4.4 how the rendering equation can be split up into four different terms: Direct illumination, specular and glossy reflections, caustics, and multiple diffuse reflections. Having direct illumination, the next step towards global illumination is to simulate specular reflections. To start out

[1] *Umbra* denotes an area in full shadow. *Penumbra* denotes an area partly in shadow.

simple, we first consider single specular reflections for planar surfaces (light paths LD?S_pE, where S_p denotes specular reflections off planar surfaces only).

In global illumination specular reflection is an effect available through ray tracing. This section will introduce a rasterization approach to specular reflection, making it available in real time.

As described section 3.1 the angle of reflection equals the angle of incidence. In case of a planar reflector, the reflection of a scene is in other words an inversion of the scene in the plane representing the reflection surface. Figure 6.1 illustrates how the reflection can be represented as a rendering of the scene from an inverted virtual viewpoint.

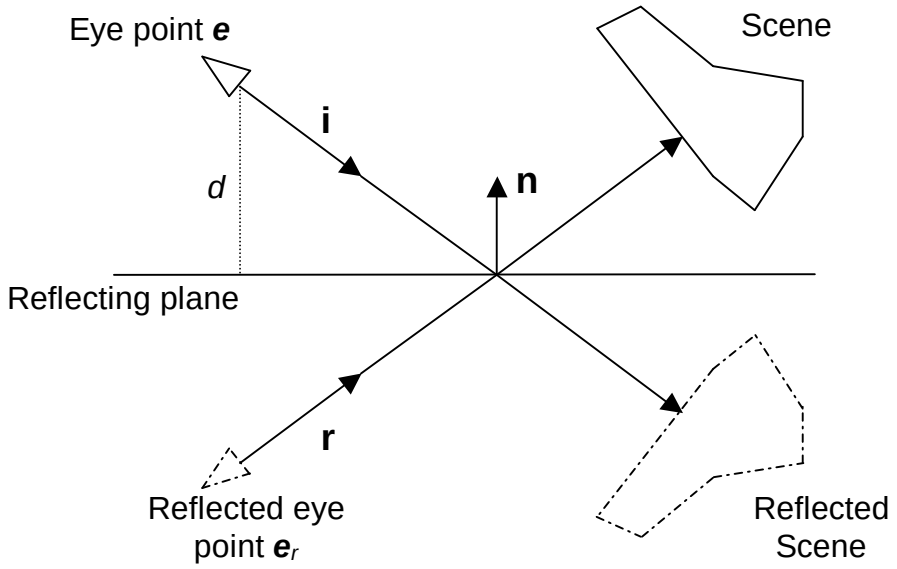


Figure 6.1: A scene reflected in a plane (planar reflection). The reflection can be seen as a reflection of the scene geometry in the reflecting plane or as a rendering of the scene from a reflected eye point. This figure is a reproduction of figure 6.1 in [90].

The position and viewing direction of the reflected eye point $\boldsymbol{e}_r$ and the reflected viewing direction $\boldsymbol{r}$ can be found from the position of the original eye point $\boldsymbol{e}$ and the original viewing direction $\boldsymbol{i}$, using the following two formulas:

$$\begin{aligned} \boldsymbol{e}_r &= \boldsymbol{e} + 2d(-\boldsymbol{n}) \\ \boldsymbol{r} &= \boldsymbol{i} - 2(\boldsymbol{n} \cdot \boldsymbol{i})\boldsymbol{n} \end{aligned}$$

where $\boldsymbol{n}$ must be a *unit* surface normal of the planar reflector. Here $\boldsymbol{r}$ is found using (3.2). Note, however, that signs have been changed since $\boldsymbol{i}$ points towards the surface not away from it.

One way to implement planar reflections is simply to render the scene from the reflected eye point in the reflected viewing direction. Afterwards

the result is shown in the reflecting surface.

Another method, which is used more frequently, makes use of the stencil buffer. This is also the method implemented for this project. The general idea is to let the stencil buffer clip out the part of a scene that should be reflective and then render the reflected scene in this area of the screen. By use of the stencil buffer we will only need to render the visible part of the reflected scene instead of the entire reflected scene. An example of how this can be done is to render the scene in one pass and stencil mark the pixels of the reflecting object. Then render the reflected scene into the marked area in the second pass [90].

The step by step algorithm of planar reflection using the stencil buffer appears in [70] as follows:

1. Clear the color, depth, and stencil buffers.
2. Enable writes to the color and depth buffers. Disable writes to the stencil buffer.
3. Render the scene excluding reflecting surfaces.
4. For each reflecting surface:
   (a) Disable writes to the color buffer. Set up stenciling to write the value 1 into the stencil buffer when the depth test passes.
   (b) Draw the polygons of the reflecting surface. Thereby all planar reflectors that are not occluded will be 'tagged' in the stencil buffer.
   (c) While the color buffer is still disabled, set the depth range to write the farthest value possible for all updated pixels and set the depth test to always pass. Also, set the stencil test to only update pixels tagged with the stencil value 1.
   (d) Draw the polygons of the planar reflector once again. This will clear the depth buffer for all visible pixels of the planar reflector.
   (e) Reset the depth test and the depth range. Re-enable the color buffer.
   (f) Establish a clip plane to render objects on the reflective side of the mirror plane only.
   (g) While still only updating pixels marked in the stencil buffer, render the scene mirrored in the plane of the reflecting surface.

The stencil buffer marks the area in which reflections occur in the scene, in this way only the reflective part needs to be considered when calculating the reflecting image. This prevents unnecessary calculations. The stenciled

planar reflections have been implemented for our demonstration application described in part III.

Planar reflections can also be simulated using texture mapping. This procedure is straight forward and uses two passes. First pass renders the reflected scene into a texture and second pass maps it to the reflecting object [79].

An advantage of using the stencil buffer procedure is that we will only need to update the pixels of the reflecting object. When using texture mapping, we can have problems with magnification and minification artifacts if the eye point is too close or too far away [90]. These artifacts will not appear in the stencil buffer method, since the scene is actually rendered.

If two reflective surfaces are facing each other reflections inside the reflections should be generated. In principle this reflection should go on for ever. A reflection loop like this is referred to as recursive reflections, which can be rendered using real time approaches as well. The planar reflection algorithm is recursively repeated a number of times. This number depends on the max recursion depth that is a parameter for how many reflections inside reflections there should appear. The procedure is described in [90, 91]. Meaning that we can simulate the light paths $\texttt{LD?S}_p\texttt{*E}$ using this approach and direct illumination, which brings us even closer to a full implementation of the specular term in real-time. The following section further explores this opportunity by describing a method for simulation of specular reflections on curved surfaces.

6.3 Cube Environment Mapping

In order to have a better approximation of the rendering equation, we must be able to account for specular reflections on curved surfaces as well as on planar surfaces. Unfortunately the technique for curved surfaces is not well suited for planar surfaces and vice versa, therefore we need both approaches. The light paths we wish to simulate in this section are $\texttt{LD?S}_c\texttt{E}$, and we can even expand this approach to include multiple specular reflection (light paths $\texttt{LD?S}_c\texttt{+E}$).

Environment mapping was introduced in 1976 by Blinn and Newell in [10]. It is a way to make curved objects appear to reflect the surroundings based on images saved in textures. The concept is to create an environment map describing the surroundings of the reflective object. The reflection is then simulated through lookups in the map. Environment mapping has become one of the most popular methods for generating perfectly specular reflections in real time.

Environment mapping has certain limitations. The map is created from a single point in space (usually the center point of the reflective object), meaning that the geometry of the reflection is only accurate if the surface point

is at the object center or if the reflected environment is infinitely distant. This means that all the surroundings, in fact, are reflected at the center of the reflecting object. In practice it means that objects located close to the reflecting surface will be reflected incorrectly, if the object close by is large, the effect will not be particularly noticeable. Shadows reflected on surfaces close to the reflector seem to cause the biggest problems. Another limitation to environment mapping is that the reflecting object not will be reflected in itself.

Different environment mapping methods exists. Sphere mapping, cube mapping, and dual paraboloid mapping are all supported directly by graphic APIs and graphics accelerators. Good descriptions of these can be found in [90]. In this project we have used cube environment mapping, which therefore will be the only one that we describe. The main principles of cube environment mapping will now be presented.

Creating an environment map is like drawing the surroundings of a reflecting object to a texture. The approach for environment cube mapping is to find a map for each of the six sides of a cube surrounding the reflecting object. The resulting texture maps picture the parts of the surroundings that are visible from the center of the reflective object in all directions. Figure 6.2 shows the concept of the environment cube map.

The cube environment map is view independent; if the eye point changes position only recalculation of the reflection vectors is needed to provide the correct reflection. The texture must be re-calculated though if the surroundings or the position of the reflector change.

In order to reflect a scene dynamically we must generate the textures for the environment each frame. The following steps are necessary in order to render the environment dynamically [142]:

1. Set up a camera with a 90° field of view at the center of the reflecting object's location.

2. Point the camera along $+x$ and render the (approximate) scene around your object into the first face of the cube map.

3. Repeat the process, rendering along $-x$, $\pm y$ and $\pm z$ into each face of the cube map.

When rendering the cube map the reflected direction ($\boldsymbol{r}$) of each fragment is used as texture coordinates for the look-up in the environment map. We can also create a single level of refraction if we use the refracted direction as texture coordinates.

Including multiple specular reflections in this approach is surprisingly simple and inexpensive with respect to processing time. We merely include the environment maps found in the previous frame, when rendering other specular objects in the images for the different faces of the environment

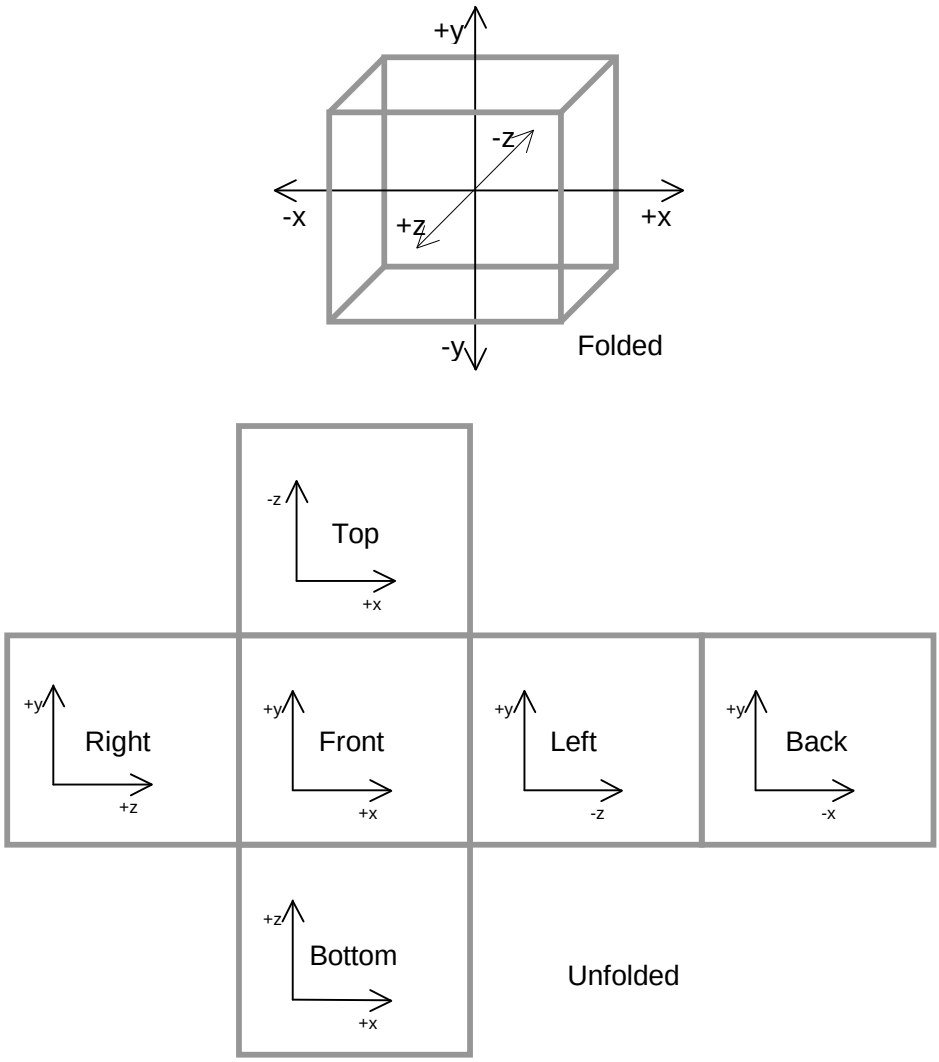


Figure 6.2: The concept of environment mapping. Each side of the cube is rendered and stored in a texture. The cube sides are referred to as cube faces.

map. This effectively provides an extra specular recursion for each frame that we render without much extra effort.

Adding the light paths that we can simulate using environment mapping to the light paths we can simulate using stenciled planar reflections (see the previous section), it is all in all possible to approximate a specular term using real-time rendering methods. The limitation is that we find reflections off perfectly specular materials only. This can be compensated for by use of blending, since the glossy effects that should be simulated by the specular reflections term of the rendering equation only include those glossy surfaces we have classified as closer to specular objects than diffuse objects.

The specular term found simulate the light paths `LD?S+E`. If we let direct illumination simulate `LD?E`, the resulting rendering method will simulate the paths `LD?S*E`. The specular term found here is not entirely comparable to the specular and glossy reflections term described in photon mapping (cf. sec. 4.4), which simulate the paths `LS*(S|D)S+E`. The missing part is reflection of caustics, which we cannot accomplish unless we employ a method that can simulate caustics in the first place. In the following section we will discuss a few methods that could be used to simulate caustics in real-time.

6.4 Real-Time Caustics

Though there has not been time for a real-time implementation of caustics during this project, we would like to point out some of the methods that have been presented in literature, since they can be used in combination with the methods mentioned previously in this chapter and with our own method called direct radiance mapping as well.

The only rasterization approach to real-time caustics that we are aware of is presented in [130]. The idea is to "discretize the surface of a specular reflector into small regions and treat each as a pinhole camera that projects the incoming light onto the surrounding objects". Unfortunately the approach is prone to aliasing artifacts and the results are not too convincing. Nevertheless it is one possible way to render caustics in real-time.

All other approaches to real-time caustics that we have come across seek to simulate photon mapping in real-time. Even if some approaches simulate the entire photon mapping algorithm, the caustics can (as shown previously) be separated from the rest of the photon mapping method and used individually in combination with other methods.

Purcell et al. [108] describe how photon mapping can be implemented in graphics hardware. They use a filtered direct evaluation of a global photon map which result in visible low frequency noise with respect to indirect illumination. Their processing times are unfortunately measured in seconds per frame (not fps). Donner and Jensen [28] have subsequently described how the processing time of the GPU computations can be improved using adaptive refinement. Using the method for caustics alone and combining it with other methods for the remaining computations, would probably result in better frame rates.

Real-time caustics can be achieved using distributed photon mapping. Günther et al. [44] show how caustics can be rendered using a cluster of 9 to 36 CPUs. This is, however, less interesting with respect to real-time rendering on a single CPU, which is the objective of this project. On the other hand they also present some interesting optimizations for photon mapping on a single machine. For example they use a variant of selective photon emission, which was originally introduced in [27], and they use alternative

data structures for storage of the photon map, see section 2.5.

Finally an interesting method for simulation of photon mapping in real-time is presented by Larsen and Christensen in [74]. The method is also described in section 6.7, but pick out the caustics method for now. The caustics calculations use a pbuffer[2] to count the number of caustic photons ending up in each fragment. Using this pbuffer as a texture for a fragment program, makes the fragment program able to determine the contribution of the caustics term in each fragment. A simple filter is applied which takes the caustics term of nearby pixels into account. The results are promising and the implementation is quite straight forward, even though an efficient ray tracer is needed for photon tracing.

In the following section, we will describe a few complex BRDFs that can enhance the shading methods normally used for real-time applications.

6.5 Complex BRDFs

Even the modified Blinn-Phong BRDF (see sec. 5.2) is not very close to a physically correct BRDF. The emergence of programmable graphics hardware has given a good opportunity for implementation of more complex BRDFs in real-time shading. In this section we shortly present a physically based BRDF and an empirical BRDF.

The Cook-Torrance Model

The Cook-Torrance BRDF is a physically based model, and it is a significant improvement compared to the modified Blinn-Phong model. It is quite expensive though. The math is given as follows [21]:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = \frac{F(\beta)}{\pi} \frac{D(\theta_h) G}{(\boldsymbol{n} \cdot \boldsymbol{\omega})(\boldsymbol{n} \cdot \boldsymbol{\omega}')} + k_d \tag{6.1}$$

where $\cos\beta = \boldsymbol{h} \cdot \boldsymbol{\omega} = \boldsymbol{h} \cdot \boldsymbol{\omega}'$ and $\cos\theta_h = \boldsymbol{n} \cdot \boldsymbol{h}$. $\boldsymbol{h}$ is the half-vector.

D simulates micro-facets for the material and is given as:

$$D(\theta_h) = \frac{1}{\alpha^2 \cos^4 \theta_h} e^{-(\tan\theta_h/\alpha)^2}$$

where α is a material parameter (it might be referred to as a measure of roughness).

G captures masking and self-shadowing by the micro-facets:

$$G = \min\left(1, \frac{2(\boldsymbol{n} \cdot \boldsymbol{h})(\boldsymbol{n} \cdot \boldsymbol{\omega}')}{\boldsymbol{\omega}' \cdot \boldsymbol{h}}, \frac{2(\boldsymbol{n} \cdot \boldsymbol{h})(\boldsymbol{n} \cdot \boldsymbol{\omega})}{\boldsymbol{\omega}' \cdot \boldsymbol{h}}\right)$$

[2] pbuffer is short for pixel buffer and it is an off screen buffer for storing rendered images which need not appear on the screen.

The Ward Model

The Ward model is an empirical model, meaning that it primarily intends to fit experimental data. It is less expensive than Cook-Torrance and it includes an intuitive set of material parameters: ρ_d, the diffuse reflectance, ρ_s, the specular reflectance, and α, the surface roughness. The BRDF is given as follows [132]:

$$f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') = \frac{\rho_d}{\pi} + \rho_s \frac{e^{-(\tan\theta_h/\alpha)^2}}{4\pi\alpha^2\sqrt{(\boldsymbol{n}\cdot\boldsymbol{\omega})(\boldsymbol{n}\cdot\boldsymbol{\omega}')}} \tag{6.2}$$

as before $\cos\theta_h = \boldsymbol{n}\cdot\boldsymbol{h}$ and $\boldsymbol{h}$ is the half-vector.

This concludes the sections concerning methods that can be combined with direct radiance mapping. In the following we will describe competing methods, which all seek to evaluate the multiple diffuse reflections term. First we will look at a traditional technique for real-time 'pseudo-global' illumination called *light mapping*.

6.6 Light Mapping

One obvious advantage of radiosity compared to other global illumination techniques is that it computes lighting throughout the scene, and that this lighting is static as long as the objects do not move. The reason is the assumptions of diffuse surfaces only. Diffuse surfaces reflect light equally in all directions over the hemisphere, and the resulting view-independence has the effect that the contribution of light to a surface can be captured in a texture and attached to the surface.

The concept of lighting information attached to a surface using a texture is referred to as light mapping. After the light maps have been created they can be used for real-time rendering nearly without expenses. In diffuse environments it will even be difficult to notice that the lighting does not change accordingly when objects are moved around. The illusion, however, quickly breaks if we start moving the light source around, or if we place a blue ball in front of the light source and notice that the color bleeding does not change.

Still, light mapping is an inexpensive and widely used technique which can simulate multiple diffuse reflections in real-time. In [109] it is described how radiosity was calculated on graphics hardware and stored in light maps for real-time use in a large scale game from Atari called "Shadow Ops".

In order to use radiosity stored in light maps in combination with the methods described previously in this chapter, we must choose either to include the direct illumination term in the radiosity calculation, which will result in obvious flaws when objects are moved, since the shadows will not move accordingly. Alternatively we can adjust the radiosity algorithm to

make sure that the light maps only include light reflected diffusely at least once. In this way the light map can be combined with the direct illumination term using blending.

Other global illumination techniques could be used with light mapping as well as radiosity can. We must always make sure, though, that only the multiple diffuse reflections term is computed, if we plan to combine the light mapping with other methods. The reason why it is clever to use light maps only for the multiple diffuse reflections term is that the indirect illumination changes quite slowly over the surfaces and therefore it will be less noticeable that such lighting is static.

To summarize light mapping can create sophisticated lighting at minimum cost for static scenes illuminated with static light sources. An optimization of the concept is to store only light reflected diffusely at least once in the light map, and then combine the light mapping technique with other methods for different parts of the rendering equation. Still, if we need our application to simulate dynamic indirect lighting, or if we do not have time for a global illumination preprocessing stage, we have to look for different approaches to an evaluation of the multiple diffuse reflections term. A brilliant reference for further information on light mapping techniques is [16].

The next section describes real-time simulation of photon mapping, which we also had a brief look at with respect to real-time caustics in section 6.4.

6.7 Real-Time Photon Mapping Simulation

A cunning real-time implementation of photon mapping is presented in [72, 74]. The part of this implementation that simulates the multiple diffuse reflections term is, of course, a competitor to our own method for computation of the same term. The real-time photon mapping approach could equally well be combined with the methods described in sections 6.1 through 6.5.

The real-time simulation of photon mapping first seek out the different surfaces in a scene. According to the method described in [73], a global photon map is created for each surface (this speed up photon map construction and radiance estimates).

A variant of the selective photon emission described in [27], is used to emit only intelligently selected photons each frame. Using selective photon emission, the photons are emitted in groups. The method presented in [74] emits one photon for each frame from each group. The path of the previous photon emitted in each group is stored. If the path taken by a photon in a group is not the same as the previous path, the entire group of photons is marked for redistribution.

For each surface a texture is generated holding radiance estimates from the photon map related to that particular surface. These textures are referred to as *approximated illumination maps*.

A hardware optimized final gathering method is presented, where a number of locations on the surfaces of the scene are chosen (eg. by the modeler). The approximated illumination maps are applied to each surface and then a picture is taken from each chosen location to simulate final gathering rays. The resulting pictures are averaged using hardware optimized mipmapping (the topmost image in the mipmap is the average). The result is a number of pre-determined locations distributed over the surfaces throughout the scene where the multiple diffuse reflections term is known - we could call it an indirect illumination field.

One way to visualize this indirect illumination field is simply to store the indirect illumination in coarse textures, which can then be applied to the surfaces where the indirect illumination was calculated. This approach is feasible since diffusely reflected indirect illumination usually changes slowly over a surface.

Compared to light mapping this method handles dynamic scenes illuminated by dynamic light sources. Furthermore it potentially simulates all kinds of light paths in the multiple diffuse reflections term (`L(S|D)*DS*DS*E`). The approximate final gathering unfortunately smooth out the result[3], but this is relative since it depends on the number of locations we choose on the different surfaces. The resulting illumination in a Cornell box using this method is almost indistinguishable from the global illumination reference (see [72]).

An additional work load that must be added to an implementation of this method is an efficient ray tracer, which must be available for the photon tracing. Efficient ray tracing involves spatial data structures (eg. BSP trees) which put some restrictions on the scene geometry. If objects are animated or if they morph during rendering, the spatial data structures must be reconstructed, this process may be quite expensive with respect to processing time. Also dynamic objects should be pointed out in advance and each of them should have an individual data structure, since reconstruction of a global spatial data structure each frame is too expensive.

Another disadvantage is the final gathering method where many low resolution pictures are taken at pre-determined locations in the scene geometry. This approach is not easily scaled to large and complex scenes, the number of approximated illumination maps that must be processed will escalate.

We have now described light maps, which pre-compute a static case of global illumination and store the result in a texture for each surface. Besides we have described a method for dynamic simulation of global illumination in the form of photon mapping. The next section will describe an approach that seeks to take the best of both worlds and combine pre-computation with

[3]This is unfortunate since smoothing the indirect illumination has the result that only close to perfectly diffuse surfaces can be handled at this last bounce of diffusely reflected indirect illumination.

dynamic lighting.

6.8 Pre-computed Radiance Transfer

In radiosity form factors represent energy transfer between surfaces. It is assumed that all surfaces reflect light diffusely and that radiosity is constant across the surface of a patch, hence, we need not consider the dependence on direction and location. This is why form factors represent *energy* transfer. If we remove the assumption of diffuse surfaces only, the concept of form factors can be generalized to represent *irradiance* transfer (instead of energy transfer) between surfaces. In practice this means that an arbitrary BRDF must be included in the radiosity form factor.

In [122] it is shown how a collection of spherical harmonics basis functions can represent such generalized form factors, or irradiance transfer functions. The reason for using spherical harmonics is that "a finite number of terms can be used to approximate relatively smooth functions defined on the sphere" [122, p. 192].

The concept of *pre-computed radiance transfer*, described in [123], is to let a dense set of vectors or matrices over a surface represent its *radiance* transfer function. The radiance transfer function is a further generalization of the spherical harmonics based approach described in [122]. The radiance transfer functions are sufficiently general to be used for the evaluation of the rendering equation directly.

Though arbitrarily complex BRDFs can be simulated using radiance transfer functions. The pre-computed radiance transfer methods often limit themselves to low-frequency lighting only. The reason is that diffuse surfaces, and glossy surfaces with diffuse characteristics, typically require fewer spherical harmonics basis functions in the radiance estimate than specular surfaces do. Fewer basis functions means fewer computations and, hence, the method becomes less expensive and less prone to aliasing artifacts resulting from an insufficient order of the spherical harmonics basis.

The approach is then to pre-compute for a given surface the radiance transfer functions, which are independent of the incident radiance term. A summation over inner products between the transfer functions and incident radiance samples on a surface then gives an inexpensive, dynamic, approximate evaluation of the rendering equation.

Because of extensive pre-computations the method is restricted to use of rigid objects only. The 'simple' version of pre-computed radiance transfer only works for radiance transfer from a convex, rigid object onto itself. This is called *self-transfer*. In [123] a method is also described for *neighborhood-transfer*, where radiance transfer functions are also computed from a rigid body to its neighbor space. This expansion allows soft shadows, glossy reflections, and caustics on dynamic receivers.

The neighborhood-transfer is important if the method is to be useful in a complex scene composed of many moving objects. However, neighborhood-transfer has problems when multiple objects reflect light or cast shadows on the same receiver. Moreover the lighting must be fairly constant across the entire neighborhood of an object to provide accurate results. This indicates that pre-computed radiance transfer is a brilliant method if only a single (or a few) objects are present in a scene at the same time (a scenario could be a dynamic virtual character filling out the entire scene while talking to the user of a 3D application). In more dynamic environments the method unfortunately falls apart. And lastly the developer using the method must be prepared for long pre-computation times.

A very different method finding much inspiration in the same background as pre-computed radiance transfer, is the subject of the next section.

6.9 Environment Map Rendering

After it was shown in [122] how a spherical harmonics representation could be derived for arbitrary BRDFs, the concept has subsequently been combined with environment mapping in [110, 111]. The approach used here was *sphere* mapping (where the sphere map is a two-dimensional image of a sphere). The sphere environment map represents the irradiance transfer functions used for spherical harmonics approximation of an arbitrary BRDF.

In a different context Greger et al. [43] had shown that *irradiance volumes* spaced in a regular grid across a scene could be used for storage and interpolation of global illumination in a scene. Irradiance volumes is described as an extended version of irradiance, which is defined in all points and directions in space. This means that irradiance volumes are decoupled from scene geometry. (In our opinion a better description of an irradiance volume is that it captures radiance transfer in all directions at a certain point in the scene.) Irradiance volumes were originally presented for pre-computation with traditional global illumination techniques.

A brilliant idea, presented in [77, 93, 92], is to combine the environment mapping representation of transfer functions with the irradiance volumes approach to global illumination. Instead of sphere maps, cube environment maps are used for storage of radiance transfer functions. Those cube maps are spaced in a regular grid of $4 \times 4 \times 4$ across the scene. Each environment map is updated each frame using the same technique for environment maps as described in section 6.3.

Since the transfer functions are decoupled from geometry and regularly spaced in the scene, a radiance estimate at an arbitrary location $\boldsymbol{s}$ in the scene is found by tri-linear interpolation to $\boldsymbol{s}$ from the "irradiance volumes" placed at the corners of the grid cells.

When mapping this method to the GPU, different spherical harmonics

coefficients are calculated using the environment maps and stored in 3D textures which are uploaded to the GPU in order to compute radiance estimates in a fragment program.

Though frame rates are below 10 per second, the results of this method look promising. Even for rather complex scenes. According to Nijasure et al. there are only few limitations to their technique, and they propose solutions for all the problems they describe. One problem is that the $4 \times 4 \times 4$ grid dimension is not sufficient for large scenes. The solution they propose is to space the grid non-regularly in the view frustum. This may, however, result in a new problem, since the tri-linear interpolation may be incorrect if the grid is non-uniform.

They also note that their current implementation do not support arbitrary BRDFs, though the method does not exclude them. We must keep in mind though that specular surfaces will be prone to aliasing artifacts because of the general limitations of the spherical harmonics approximation, while glossy surfaces with diffuse characteristics can be simulated if some extra processing time is set off for the radiance estimate.

A third problem about the method is color or shadow leakage. Some leakage is caused by the tri-linear interpolation scheme. A method that can handle this problem is presented in [92], though it can not (yet) be implemented in hardware.

Problems not mentioned, are problems associated with the spherical harmonics representation of the radiance transfer. According to [122] and [123] some aliasing artifacts can occur if too few spherical harmonics coefficients are computed. Another issue is that spherical harmonics are defined over the entire sphere and therefore it is difficult to completely remove the radiance in one direction from an "irradiance volume" if it is positioned close to an occluding object. This problem can result in light leaking through walls.

All that said, the reported results are still quite impressive. This method will be hard to compete against. Let that conclude our description of existing methods for real-time simulation of global illumination.

Part II

Modeling Contents

All of part I is based on the geometrical representation of a scene and the calculation of illumination in it. This part will concentrate on the creation of a scene and how to alter the scene in order to create a dynamic application. Scenes are normally created in a modeling program; in this project we have used the modeling program called Blender, which is freeware. In the modeling program there are settings for light sources, materials, and camera, and also animation features for moving things around in the scene. The terms used for modeling can sometimes be different from those used in the theory, especially when it comes to materials and light. During this part we will try to introduce some of the typical terms for light and material settings and relate them to the theory. Blender will also be shortly introduced in this part.

The first chapter of this part - chapter 7 - will describe how to model 3D scenes that are composed of polygons, as these most often are the primitives used in real-time graphics. Though objects in a scene are normally created from polygons, there are several handy methods for creating the 'right' polygons. Sometimes objects are created from curves or primitives and then transformed into polygons by the application, we have, however, restricted ourselves in this project to a description of polygonal modeling only.

Just after the objects of a scene have been created they appear dull, this is until material or textures are applied to them. There are a bunch of parameters that can be adjusted to make the objects look exactly right, as well as different special effects that can be used to create realistic objects. Chapter 8 concerns material settings and textures.

To make a scene even more interesting, from a users point of view, we must make it come alive, meaning that some objects in the scene should move about. This is the subject of chapter 9. Objects following a series of movements over a predetermined number of frames are referred to as animated objects. There are differences between animation for movies and interactive applications such as games; those will also be discussed in chapter 9.

After going through different aspects of modeling we turn to the modeling application. Chapter 10 will introduce how scenes are created using Blender. The chapter will introduce those features that we have used for creation of our test scenes, and show how those features described in chapters 7, 8, and 9 are used in practice. Since we want to test our own renderer with the scenes that we create in Blender, we need to export our scene. How this is done will also be explained in chapter 10.

Chapter 7

Modeling 3D Scenes

We do not grow absolutely, chronologically. We grow sometimes in one dimension, and not in another; unevenly. We grow partially. We are relative.

Anaïs Nin

In this chapter we will give a brief introduction to some of the concepts behind 3D modeling. As neither of us has any professional angle to modeling, this chapter will not address any advanced methods, it will rather present the knowledge we have achieved while modeling during this project. We will consider polygonal modeling only. Subjects described in this chapter are presented from a Blender point of view, where Blender is the modeling application used to create our test scenes.

Objects in most real time applications are built from polygons connected in so called polygon *meshes*. A polygon mesh can be created automatically by the modeling application, from primitive objects, extrusions, or polygon sweeps carried out by the user, or they can be created one polygon at the time.

One simple way to create polygon meshes is by use of primitive objects. The primitive objects available in Blender are presented in figure 7.1. For example we can create a primitive person from a cube, a sphere and four cylinders, see figure 7.2.

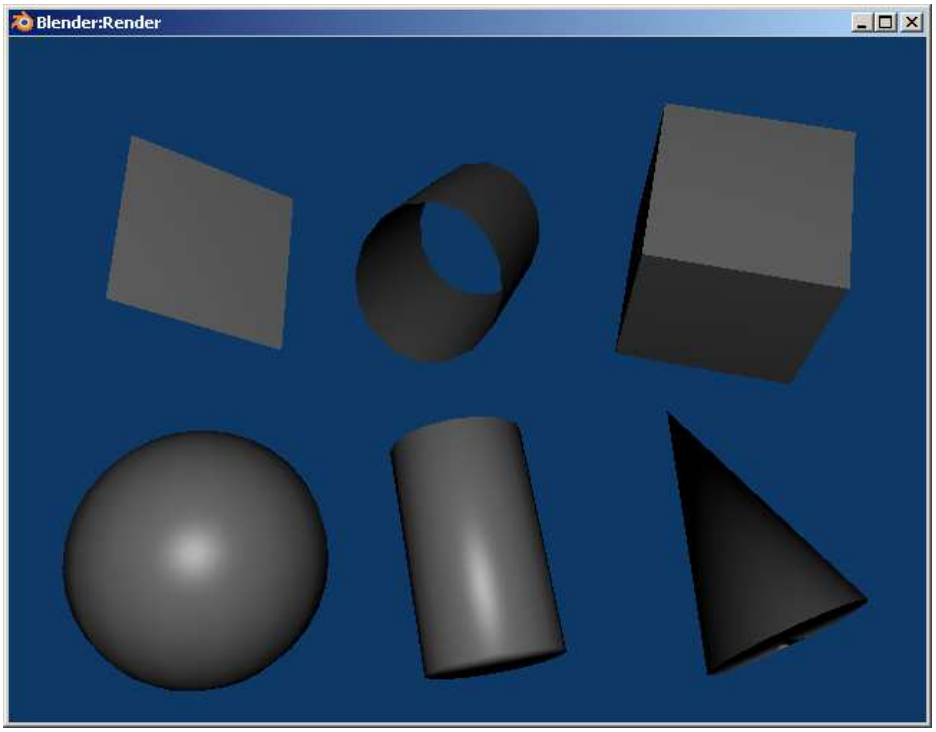


Figure 7.1: The primitives available in Blender. Top left to bottom right: A plane, a tube, and a box. A sphere, a cylinder, and a cone.

The shape of the primitives, which the person is composed of, have changed a little compared to those in figure 7.1. The cube is now flattened and the cylinders are stretched and slimmed to fit the shape of body, arms,

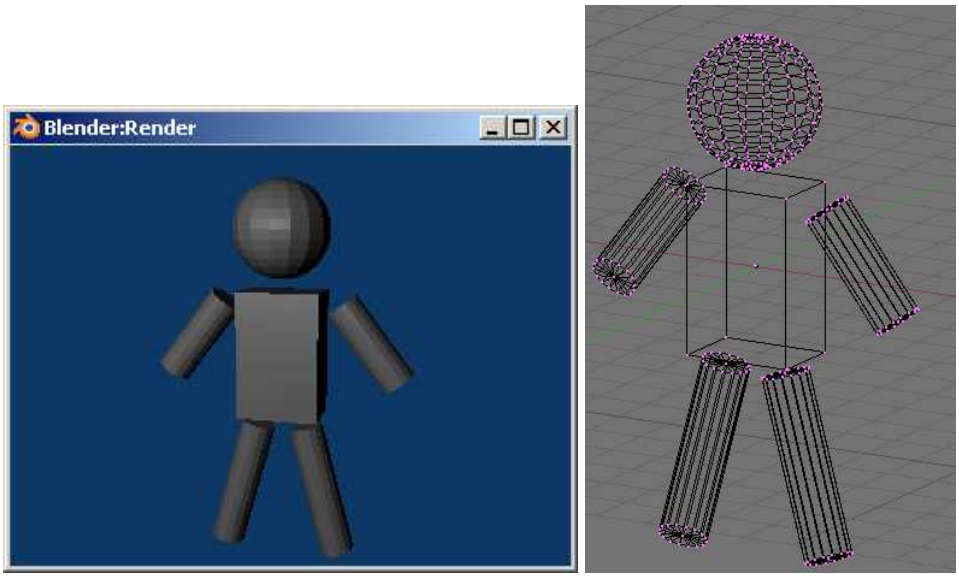


Figure 7.2: Primitive person created from a cube, a sphere and four cylinders.

and legs. All primitives can be stretched, rotated, translated, and scaled into arbitrary forms and positions. These transformations are carried out by transformation of each vertex in the polygon mesh using the transformation matrices presented in chapter 9.

To make our person look more realistic we could add more primitives, or we could start manipulating smaller groups of vertices or even single vertices if we desire. Faces can also be created between polygons to assemble the figure where holes appear, like above the shoulders. Figure 7.3 is a fast attempt to show how this concept is carried out on the left shoulder. Of course, more details are needed for this to look convincing.

Another approach for creation of polygon meshes is through extrusions and subdivisions. Starting from one primitive object we can extrude faces creating new vertices and in this way build up our mesh. This is perhaps a more abstract procedure, but it gives more freedom to the artist. Figure 7.4 gives an example.

The series of pictures in figure 7.4 should give an impression of this modeling concept. The figure shows an example of how it is possible to start the creation of the left side of a human torso. It is normally a good idea to create symmetric objects in halves, then later copy and mirror the mesh to create a fully symmetric line through the object. In this way you can be sure that, for example in the case of a human, both arms and legs are of equal size. The two halves can be assembled in the same manner as the shoulder was assembled with the torso and head in figure 7.3, or another option is to let Blender merge points lying at almost identical positions in space.

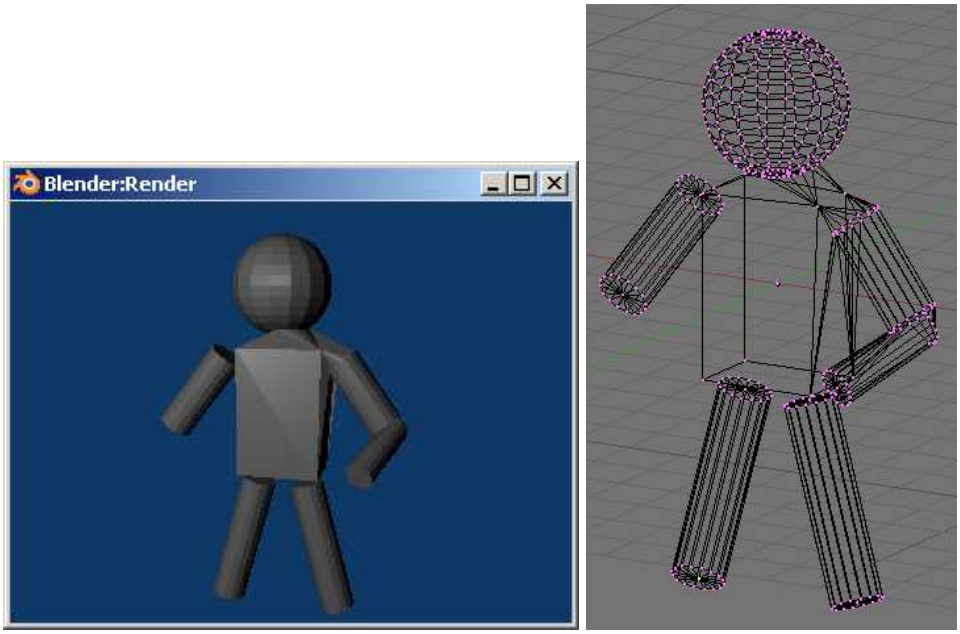


Figure 7.3: Assembling primitives. Here the assembly has been carried out for the left arm.

Sometimes it is more appropriate not to begin with a primitive object. For more complex objects or symmetric objects it can be convenient to begin with a curve that is either extruded or rotated around an axis (which is called a sweep). This approach can also be used when modeling after a two-dimensional sketch. In Blender it is possible to set a picture as the background of the modeling area. From this it is possible to draw the contour of an object or detailed parts of it and then extrude or rotate this contour line in a proper way to create the third dimension of the object. An approach that has been used for the creation of most objects in the test scenes is to draw the contour of objects by line segments, following a sketch in two dimensions, and then extrude the resulting curve in the third dimension. The resulting three dimensional object must be assembled with faces. The procedure is presented step by step in figure 7.5 (and is also described in chapter 10).

Another more advanced way of modeling is to use parametric curves and surfaces. This results in smooth objects, which often more closely resemble the shapes that we see in real life. Methods even exist for rendering of curves and surfaces directly, this is outside the scope of this report. If such a renderer is not available, objects created using curves and surfaces will, eventually, be represented as polygons before they are rendered. Modeling using parametric curves and surfaces will not be addressed any further, since we have not used them in the scenes created for this project. A good reference

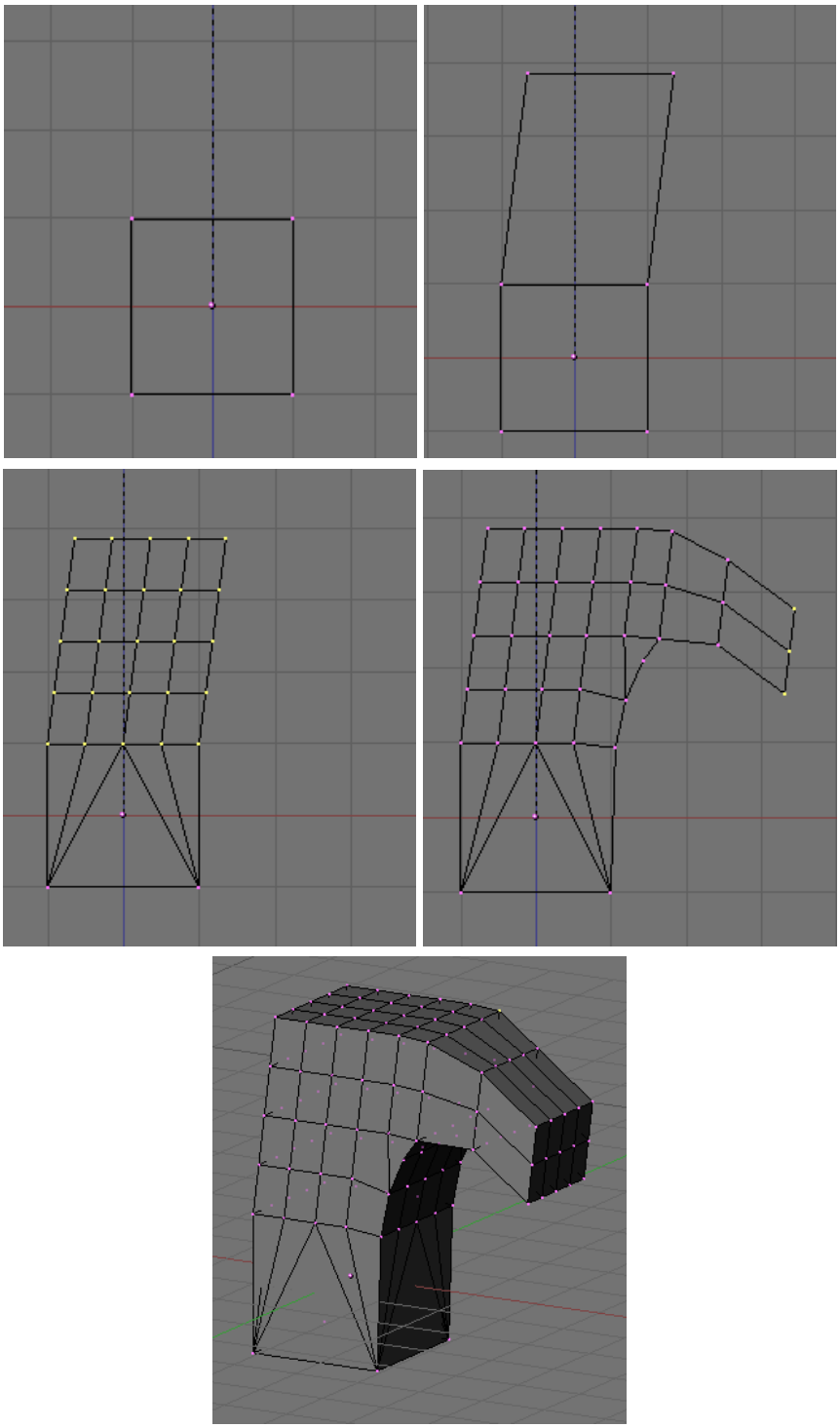


Figure 7.4: Object creation by extrusion and subdivision. From top left to bottom: (a) Beginning with a cube, (b) extrusion, (c) subdivision, (d) extrusion, and (e) a different angle.

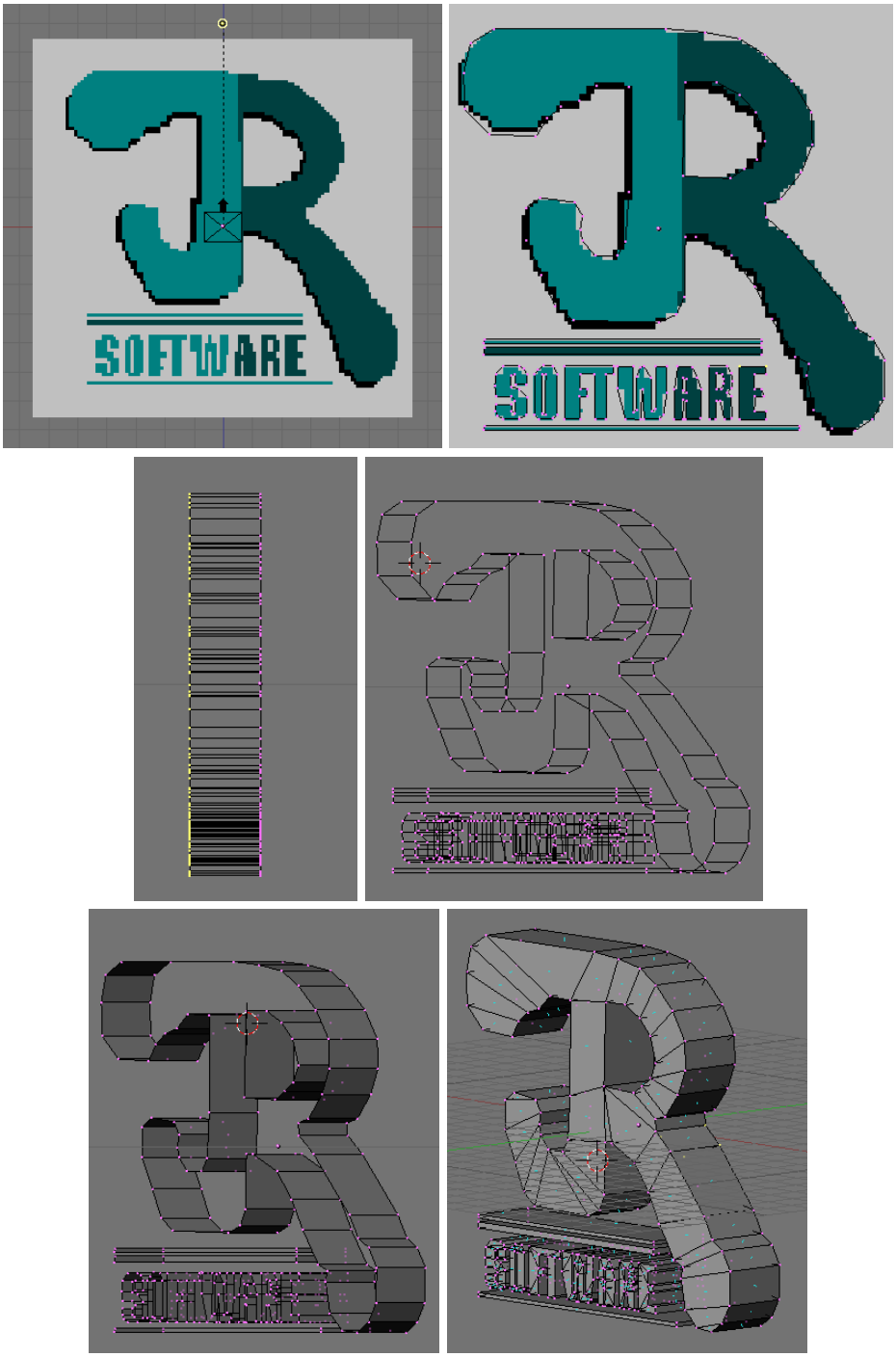


Figure 7.5: The line extrusion method. Contour of an object is drawn and then used together with extrusions or rotations to complete the 3D mesh.

on usage of curves and surfaces for real-time rendering is [134].

Above we have presented some different basic approaches to modeling polygonal objects. None of these approaches are more correct to use than others (at least not to our knowledge), in the end a combination of these and creativity is probably the best solution.

The methods we describe may not be customary procedure presented in text books as particular ways of handling modeling issues, rather they are composed of the experience with Blender obtained in this project and through different tutorials found on the internet and in [113].

When an object has been modeled the next step is to apply materials and sometimes also textures. This is the subject of the next chapter.

Chapter 8

Visual Appearance

I often think that the night is more alive and more richly colored than the day.

Vincent Van Gogh (1853-1890)

An important part of scene creation is to define material parameters for the objects. Without material settings objects will never become realistic. As a general description you could say that the material parameters simply decide how the light is reflected from the object. Still there are many issues in setting up the right material parameters. Often there are differences between the terms used in modeling applications and the terms used in theoretical texts. This chapter will treat the basic material terms and relate them to the theory presented in part I. Apart from material settings color textures also play an important role. This is especially true for real-time applications where textures are often used to achieve visual effects that are otherwise too time consuming to simulate physically. Textures will be addressed shortly at the end of this chapter.

Section 8.1 will address color settings in the RGB and HSV color spaces. Besides the objective of the section is to describe what the meaning of a material color is with respect to light transport and scattering.

The object color is one thing, but it is far from the only factor in definition of the material appearance. What is also important, is how we perceive the object material. Does the object appear soft, hard, shiny, etc.? The appearance of an object in this sense very much depends on how the light is reflected from it. Section 8.2 will discuss all material settings that are related to reflectance rather than color. The section will introduce some of the most commonly used terms for material settings in modeling environments and relate them to the different material parameters that we have encountered while describing BRDFs in part I.

Many global illumination effects are simulated in real-time using textures. Textures are great tools for giving objects the final touch of realism. A texture can in a way be seen as a detailed painting of the object. Some textures are more detailed than others. A texture can simply be a mix of two colors in a certain pattern, for example to simulate wood grains or it can be detailed for example for the face of a character. Section 8.3 shortly introduces textures and their applicabilities.

8.1 Colors and Human Perception

When a light source illuminates an object the color of the object is, in fact, a function $C(\lambda)$ given by the product between the surface reflectance and the light incident on the object at different wavelengths in the visible spectrum. Such a color function is also called the spectrum of the object. A brief description of human perception of color is needed in order to explain why three values are sufficient to model the continuous spectrum of an object

Figure 8.1 shows a model of the human eye. The eye is very complex and there are many other things to it than what is shown here. The figure only seeks to capture the most basic functionalities. In this report we can think

of the eye merely is a collector of light. The brain interprets the light that the eyes perceive and the results are the images that we see.

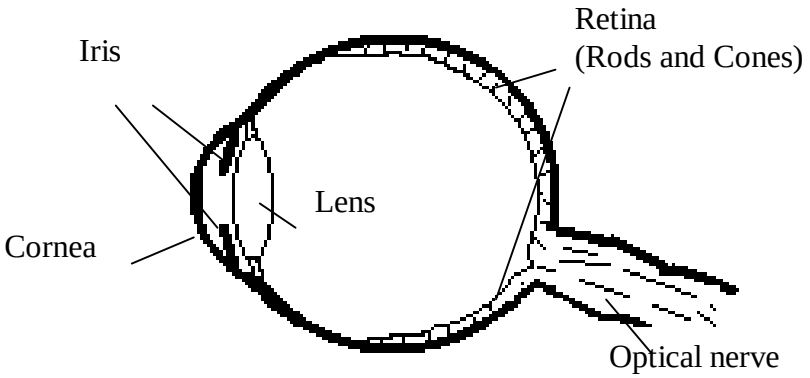


Figure 8.1: The eye alias the human visual system. (The figure is inspired by figure 1.11 in [3].)

The eye can be compared to the pinhole camera presented in section 2.3, it is more advanced though. The cornea is a transparent protection of the eye. The iris adjusts the amount of light that is allowed to enter the eye, this is important since the retina is very sensitive to light. The iris is the colored ring around the pupil and it is possible to observe its reaction to light. If there is much light a smaller amount is allowed to enter. This is the reason why pupils get smaller in bright day light and larger in dark surroundings. The lens is comparable to the camera lens, it gathers light from the surroundings and forms a 2 dimensional image on the backside of the eye called the retina. The retina is covered with small rods and cones that are able to translate the light into small impulses. These impulses are sent to the brain through the optical nerve. The functionality of the eye is therefore only to gather information and pass it on to the brain. From here it is the brain that must interpret what we see.

The rods and cones are the light sensors of the eye. The rods collect light at low levels of density, hence, they are responsible for night vision. The rods are good at collecting small densities of light, but are not good at determining the color and shape of an object. Cones are responsible for day vision and can handle colors and shapes well, but they will only work if sufficient light is present.

The cones are responsible for collection of colors. Exactly how this happens is still not fully understood, however, the eye has three different cone receptors and it is commonly accepted that the cones mainly absorbs three different colors: Red, green and blue [39]. These are referred to as the primary colors [38]. In other words the brain receive three signals from the eye and must interpret a color given by an otherwise continuous spectrum using these three signals only. “This is why three numbers can be used to represent

any spectrum seen" [2, p. 188].

The RGB color space represent colors by one value for each primary color. Moreover the RGB color model is defined according to the properties of a CRT (Cathode Ray Tube) monitor (as briefly mentioned in chapter 3), which makes it the obvious color model of choice in computer graphics.

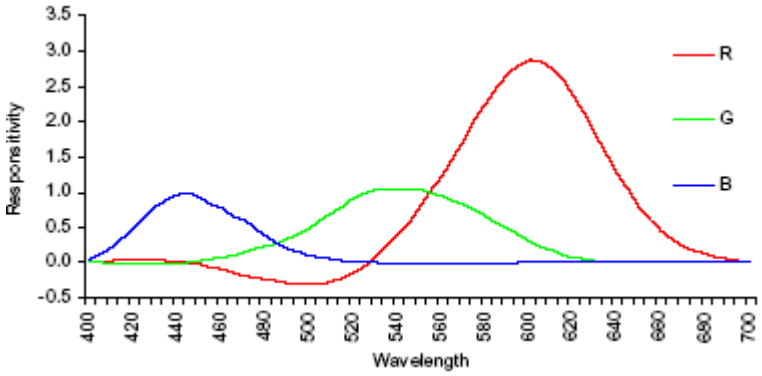


Figure 8.2: The $r(\lambda)$, $g(\lambda)$, and $b(\lambda)$ color matching curves. This figure is identical to figure 2.3 in [119]. (Courtesy of Mark Q. Shaw.)

Each primitive color $r(\lambda)$, $g(\lambda)$, and $b(\lambda)$ is itself a spectrum, see figure 8.2. When we specify R, G, and B material parameters of an object, we actually set the following values:

$$
\begin{aligned}
R &= \int_{380nm}^{780nm} C(\lambda) r(\lambda)\, d\lambda \\
G &= \int_{380nm}^{780nm} C(\lambda) g(\lambda)\, d\lambda \\
B &= \int_{380nm}^{780nm} C(\lambda) b(\lambda)\, d\lambda
\end{aligned}
$$

where $C(\lambda)$ is the color function of the material, and the integration is over the visible spectrum of electromagnetic waves.

The practical result of using the RGB color space is that we need not consider the spectral dependencies of radiometric values such as radiance, radiosity, irradiance, etc. Instead we represent each radiometric value as an (R, G, B)-vector, which effectively captures the integration since our eyes have only three different cone receptors.

Though the integration can be captured in this way, the approach is simplifying with respect to light transport and scattering. Since the integration is done "in advance" by use of RGB material color values, we can not follow individual light waves moving at certain wavelengths, and therefore we can

not model visual effects that appear in wave optics (such as fluoroscence, phosphorescence, interference, and diffraction).

Though convenient with respect to a computer monitor, a modeler should be aware that the RGB color space can not model all visible colors. In 1931 CIE (Commission Internationale d'Eclairage) defined three standard primary functions called $\overline{x}(\lambda)$, $\overline{y}(\lambda)$, and $\overline{z}(\lambda)$ from which all visible colors can be matched using only positive weights. They are meant to replace $r(\lambda)$, $g(\lambda)$, and $b(\lambda)$, to avoid their shortcomings. This has not happened in computer graphics, since the RGB model fits the monitors better.

There are many different color spaces or models representing colors in different ways. We can mention a few like CMYK, normally used in the printing industry and in ink standards, and YIQ used in connection with television. Some color models are based on perceptual terms such as *hue* and *saturation*. We will briefly describe the two color spaces RGB and HSV (hue, saturation, value), since they are both available in Blender.

The RGB color space can be visualized by the cube shown in figure 8.3. Colors in the RGB room are simply generated by adding the different values of red, green, and blue, which are defined as the primary colors.

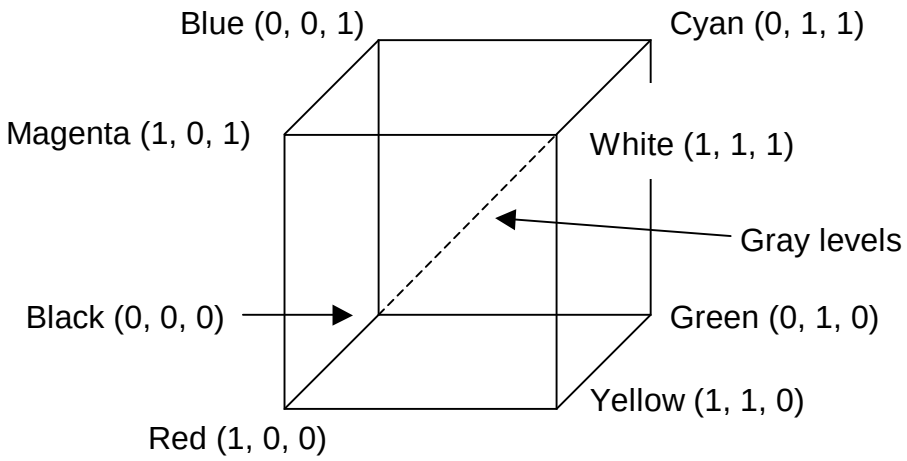


Figure 8.3: The RGB color space.

The HSV model represents colors by hue, saturation, and a value. Here we have a hexagonally shaped color space, corresponding to a cone subtended by an isometric view of the RGB color space, which means looking at the cube in figure 8.3 from the top downwards with the gray scale line as the line of sight. The HSV color space is shown in figure 8.4.

Setting the value parameter in the HSV room is like setting the value (V) between black and white, corresponding to movement along the gray scale line in RGB color space. Hue (H) is an angle selecting the pure color, and saturation (S) has the perceptual meaning that we would expect. This means that $S = 1$ specifies pure color according to H, while $S = 0$ is colorless

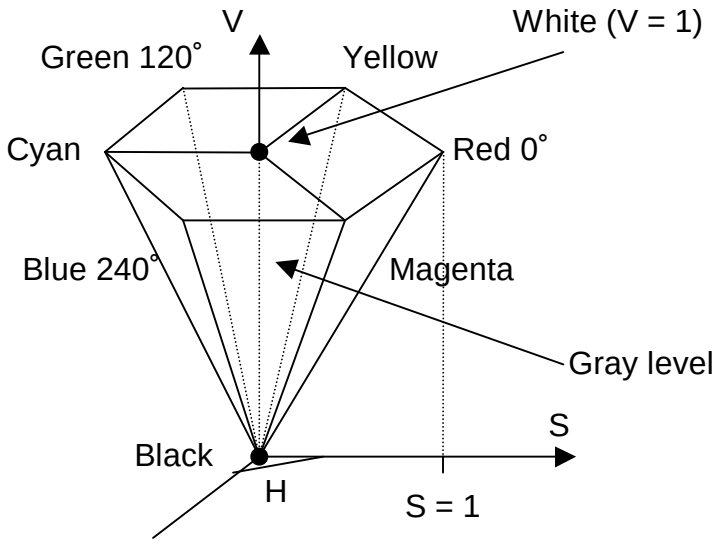


Figure 8.4: The HSV color space. This figure is similar to figure 13.30 in [38].

gray.

In this section it was described why the three component color theory is most often employed in computer graphics, and the impact of this theory on light transport and scattering was briefly mentioned. Figures 8.3 and 8.4 provide a helpful overview of the color spaces available in Blender. The next step is to describe other material settings which tell the BRDF how the reflected light is distributed over the hemisphere. This is the subject of the following section.

8.2 Material Parameters

An object can appear to be red and round, but this is not enough to describe it, for all that we know a red and round object could be anything from a red apple to a red metal ball. The missing parameter is what some might refer to as shininess. Is the object dull and disperse or is it shiny and reflective?

Besides setting the color of a material according to its reflectance at different wavelengths, there are also other parameters which are important. These parameters decide which directions light incident on a material from a certain direction are scattered back into the environment. In other words they decide how diffuse or how specular the reflectance of a material should be. Such parameters are, of course, related to the BRDF model used for shading.

Looking at the different BRDF models described in chapter 5, it shows that Phong shading and its variants are determined by three material parameters:

k_d The diffuse reflectance.

k_s The specular reflectance.

m The shininess.

In practice this means that the k_d corresponds to the general dull RGB color of an object. If the object is shiny and reflective it will have a highlight around the direction where a perfectly specular object would have reflected light directly towards the viewer. k_s defines the color of the light reflected specularly in the highlight. m sets the size of the highlight. A very shiny object (with large m) has a small concentrated highlight. To avoid overexposed light (meaning that the exitant radiance is larger than incident at the material) it should generally be the case that $k_d + k_s \leq 1$. In figure 8.5 the effect of change in the different parameters is shown.

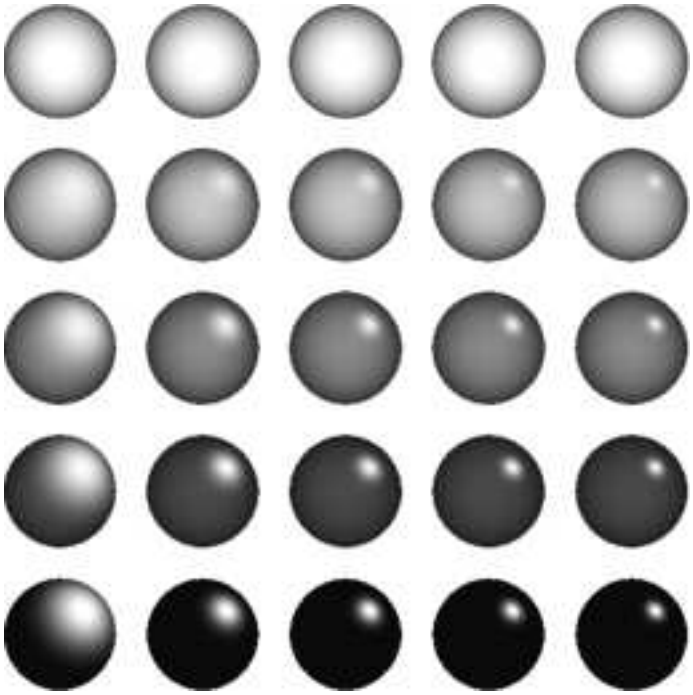


Figure 8.5: Changing material parameters associated with Phong variants. The k_d decreases and the k_s increases correspondingly in the vertical direction, going from top to bottom. In the horizontal direction we change the value of the shininess parameter, starting to the left with a small value and ending to the right with a high value.

Complex BRDF models such as the Cook-Torrance and the Ward models, described in section 6.5, mostly use a roughness parameter α rather than shininess. We will not go into detail on these material parameters, since we have mostly used Phong variants or the BRDF for perfectly diffuse materials where $f_{r,d} = k_d = \rho_d/\pi$.

Materials used for transmissive or translucent solid objects need a few extra parameters to be described adequately. A perfectly specular transmissive object, such as glass, will merely need to have an index of refraction specified. Recall from chapter 3 that translucent objects also need to have an absorption coefficient σ_a, an extinction coefficient σ_t, and a scattering coefficient σ_s specified. Besides we need to specify a phase function rather than a BRDF for translucent material. For simplicity we have chosen to work with perfectly diffuse translucent materials only in this project, which means that no material parameters, except for the diffuse object color, is needed for the phase function. The relationship between the three coefficients is that $\sigma_t = \sigma_a + \sigma_s$, which means that we need only specify two of them. Such two parameters can be measured, and they are given for different materials in literature (eg. in [65]).

When setting material properties in Blender two three component colors should be specified. They are specified either in the RGB or the HSV color space (see the previous section, and see also chap. 10). *Col* is the diffuse color of an object and *Spe* is the specular color of an object. Then Blender has a specular parameter $spec \in [0, 2]$, a shininess or roughness parameter $hard \in [1, 500]$, and a diffuse reflectance parameter $ref \in [0, 1]$. Blender allows overexposure of light. The following formulas translate the Blender material parameters to k_d, k_s, and m. Blender also allows complex BRDFs in which case m simply replaces α:

$$\begin{aligned} k_d &= Col \cdot ref \\ k_s &= \text{EACH}\,(1 \text{ min})\,(Spe \cdot spec) \\ m &= hard \end{aligned}$$

Blender has a translucency parameter which is explained in Blender as "the amount of diffuse shading of the backside". Supposedly it is used for simulation of subsurface scattering. Direct translation of this parameter to the absorption, scattering, and extinction coefficients is not immediately possible. More likely Blender translucency specifies a constant corresponding to $e^{-\tau}$, where τ is the optical depth of the object to which the material is applied (see sec. 3.4).

The index of refraction is called $IOR \in [1, 3]$ in Blender, and other parameters such as transparency A are also available. Transparency specifies the rate to which the object should be blended with the background. Blender also allows the user to specify an ambient term $Amb \in [0, 1]$ and an emission term $Emit \in [0, 1]$. The ambient color $Col \cdot Amb$ simulates multiple diffuse reflections by a constant and the emission term corresponds to a constant L_e emitted from the object to which the material is applied.

Real materials are not perfectly smooth and regularly colored over the entire surface. For example you will not be able to find any fully reflective

surfaces in the real world. Therefore it often adds extra realism to a surface if the material color changes across it. This is often added by the use of textures, which is the subject of the next section.

8.3 Textures

The visual result of adding color textures to a scene is best described by example. To have more complex showcases than a Cornell box, we have created a cave scene in Blender for this project. A screen shot from the cave scene with no textures is given in figure 8.6 and a screen shot of the same view including textures is given for comparison in figure 8.7. Note that the textures added in this scene are very simple and that it is possible to use far more complicated textures, this is, however, outside the scope of this report.

Figure 8.6: Screen shot from the cave scene with no textures (Blender render).

Unfortunately we have not found time for adding textures to all our Blender objects, and neither have we found the time to make a proper export

Figure 8.7: Screen shot from the cave scene including textures (Blender render).

of textures in order to use them in our own rendering. Though color textures can provide great visual effects, they are of less importance with respect to illumination. A color texture simply simulates that the RGB color of an object varies across the surface.

Textures are applied at a late stage in the rendering pipeline, see section 5.1, and they have many different applicabilities apart from color texturing, some of those are described in section 5.3. Often textures are used for storage of information, which is useful to have in a fragment program. This is the primary usage of textures in our rendering method, see part III. Another interesting application of textures is light maps, which is described in section 6.6.

We have now briefly introduced the most important material settings available in Blender and related them to theory in chapters 3 and 5. The next chapter will describe how objects are moved around in a scene. This is important since the ability to make things move around and come alive is the main reason why we want real-time rendering.

Chapter 9

Making Things Come Alive

24. *And God said, Let the earth bring forth the living creature after his kind, cattle, and creeping thing, and beast of the earth after his kind: and it was so.*

25. *And God made the beast of the earth after his kind, and cattle after their kind, and everything that creepeth upon the earth after his kind: and God saw that it was good.*

The Bible (King James Version): Gen.1 Verses 24 to 25

For objects to seem alive they need some sort of transformation. A scene containing one or more transforming objects is referred to as a dynamic scene. Not all objects in a dynamic scene need to transform, in fact many objects are often stationary and merely a part of the background scenery. Transformations can mean different things. A transforming object can simply be moving about in the scene, or it can change form. In some scenes all objects are moveable, in these cases there will often be a physical engine creating an artificial gravitational field to control the fall and collision of objects. In this chapter we will discuss different issues related to dynamic scenes.

The first section, 9.1, will describe the fundamental calculations behind transformation of objects. There are four basic transformations that an object, or part of an object, can go through: Translation, rotation, scaling, and shear, which are all affine transformations. From these basic transformations most movements of objects can be simulated.

After the first section where object transformation is described. Section 9.2 will explain how object transformations are controlled in a typical modeling application. Blender is our reference. We will also comment on the difference between animation for movies and animation for interactive applications such as games.

The last section (9.3) will describe how to navigate camera and objects in a scene interactively using a simple track ball. This section especially shows the applicabilities of the theory described in section 9.1.

9.1 Transformation

In section 2.3 it was described why homogenous coordinates are practical when we want to perform transformations in three dimensions. In this section we will introduce how rotation, scaling, shearing, and translation matrices are constructed and we will introduce the applicabilities of quaternions with respect to rotation.

The basic transformations: Translation, rotation, scaling, and shearing are all so called affine transformations, meaning that parallel lines in the transforming object are preserved. In rotation and translation the shape of the object does not change, such transformations are called rigid body transformations where both length and angles between points in the object are preserved. The following text basically follows chapter 3 in [2].

The transformation of a point is simply the inner product of the matrix and the point. The transformation matrix looks as follows:

$$\boldsymbol{X} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & t_x \\ a_{10} & a_{11} & a_{12} & t_y \\ a_{20} & a_{21} & a_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation, scaling, and shearing are done by altering different values in the 3×3 matrix:

$$\boldsymbol{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Scaling works by altering values in the diagonal of $\boldsymbol{A}$, where a_{00} scales in the x direction, a_{11} in the y direction, and a_{22} in the z direction. Translation is done by changing t_x, t_y and t_z. Each of the following matrices rotate an entity of ϕ radians about one of the three axes:

$$\boldsymbol{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 9.1 shows how a shearing affects an object by skewing it to one side.

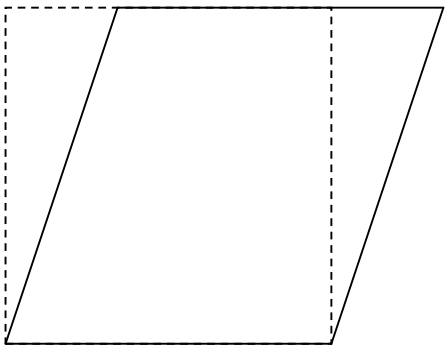


Figure 9.1: Shearing a box.

The shearing matrix is an identity matrix where one of the values that are not in the diagonal of $\boldsymbol{A}$ is non-zero. When one of these values is altered

it corresponds to shearing in one particular direction (positive or negative) on one particular axis.

Several transformations can be gathered in a single transformation matrix. If an object should be rotated a bit according to a transformation matrix $\boldsymbol{R}$ and then translated a bit according to a matrix $\boldsymbol{T}$, the final transformation matrix $\boldsymbol{X}$ is found by matrix multiplication of the translation matrix and the rotation matrix:

$$\boldsymbol{X} = \boldsymbol{T}\boldsymbol{R}$$

In this way more complex transformations can be created still using only one transformation matrix. Notice that $\boldsymbol{R}$ is applied first, but is written last. The order in which the matrices are multiplied has influence on the final outcome. This should be taken into consideration when creating the final transformation matrix.

The inverse transformation matrix is often useful, for example when switching between coordinate systems. The general way for computing the inverse of a matrix is given as explained in [2, p. 728] or any standard text book on linear algebra (eg. [31]). Another very useful way of computing inverses is in the case where $\boldsymbol{A}$ is an orthogonal matrix, which is always the case if the transformation is a concatenation of translations and rotations only. The inverse of an orthogonal matrix is given as $\boldsymbol{M}^{-1} = \boldsymbol{M}^T$, hence the inverse of $\boldsymbol{X}$ if $\boldsymbol{A}$ is orthogonal is given as:

$$\boldsymbol{X}_{\text{ortho}}^{-1} = \begin{pmatrix} a_{00} & a_{10} & a_{20} & -(\boldsymbol{t} \cdot (a_{00}, a_{10}, a_{20})) \\ a_{01} & a_{11} & a_{21} & -(\boldsymbol{t} \cdot (a_{01}, a_{11}, a_{21})) \\ a_{02} & a_{12} & a_{22} & -(\boldsymbol{t} \cdot (a_{02}, a_{12}, a_{22})) \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9.1}$$

Quaternions

A compact and useful way to represent rotations is by use of a mathematical conception called *quaternions*, introduced to computer graphics in [120]. We will not describe the mathematical background of quaternions here, some references are [2, 24, 42]. Rather we will shortly introduce their usage.

A quaternion is represented by a four-tuple $\hat{\boldsymbol{q}} = (\boldsymbol{q}_v, q_w) = (q_x, q_y, q_z, q_w)$, it should, however, not be confused with homogenous coordinates. Operations on quaternions are given as follows [2, p. 45]:

Multiplication	$\hat{\boldsymbol{q}}\hat{\boldsymbol{r}} = (\boldsymbol{q}_v \times \boldsymbol{r}_v + r_w \boldsymbol{q}_v + q_w \boldsymbol{r}_v, q_w r_w - \boldsymbol{q}_v \cdot \boldsymbol{r}_v)$
Addition	$\hat{\boldsymbol{q}} + \hat{\boldsymbol{r}} = (\boldsymbol{q}_v + \boldsymbol{r}_v, q_w + r_w)$
Conjugate	$\hat{\boldsymbol{q}}^* = (-\boldsymbol{q}_v, q_w)$
Norm	$n(\hat{\boldsymbol{q}}) = q_x^2 + q_y^2 + q_z^2 + q_w^2$
Identity	$\hat{\boldsymbol{i}} = (\boldsymbol{0}, 1)$

Suppose we have a point or a vector given in homogenous coordinates $\boldsymbol{p} = (p_x, p_y, p_z, p_w)$. Let the quaternion $\hat{\boldsymbol{p}}$ be given as each component of $\boldsymbol{p}$ inserted in $\hat{\boldsymbol{p}}$. Now, given a unit quaternion $\hat{\boldsymbol{q}} = (\sin\phi\,\boldsymbol{u}_q, \cos\phi)$, where $\boldsymbol{u}_q$ is a vector representing an arbitrary axis, then:

$$\hat{\boldsymbol{p}} \mapsto \hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^{-1} = \frac{\hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^*}{n(\hat{\boldsymbol{q}})} = \hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^* \tag{9.2}$$

rotates $\hat{\boldsymbol{p}}$ (which corresponds to $\boldsymbol{p}$) around the axis $\boldsymbol{u}_q$ by an angle 2ϕ [2]. Note that for a unit quaternion it is the case that $\hat{\boldsymbol{q}}^{-1} = \hat{\boldsymbol{q}}^*$. It can be shown that any rotation can be obtained in this manner, see eg. [42]. This indicates that quaternions are a compact and efficient way of representing rotation in three-dimensional space.

The rotation given above, (9.2), is also called an adjoint map of $\hat{\boldsymbol{p}}$. It can also be shown (eg. [42]) that this adjoint map has a corresponding 3×3 rotation matrix, which is orthogonal. Since we have only made use of unit quaternions in this project, we present the conversion from a unit quaternion $\hat{\boldsymbol{q}}$ to a 3×3 rotation matrix $\boldsymbol{M}^q$ below. For the general formula we refer to [42, 2, 120].

$$\boldsymbol{A}^q = \begin{pmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_wq_z) & 2(q_xq_z + q_wq_y) \\ 2(q_xq_y + q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_wq_x) \\ 2(q_xq_z - q_wq_y) & 2(q_yq_z + q_wq_x) & 1-2(q_x^2+q_y^2) \end{pmatrix} \tag{9.3}$$

Another interesting feature of a quaternion is the ease by which the rotation from one vector to another can be specified. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be two unit vectors denoting a direction in space. A unit rotation axis is then given as $\boldsymbol{u} = (\boldsymbol{s} \times \boldsymbol{t})/\|\boldsymbol{s} \times \boldsymbol{t}\|$. If 2ϕ denotes the angle between $\boldsymbol{s}$ and $\boldsymbol{t}$, then $\boldsymbol{s} \cdot \boldsymbol{t} = \cos(2\phi)$ and $\|\boldsymbol{s} \times \boldsymbol{t}\| = \sin(2\phi)$. "The quaternion that represents the rotation from $\boldsymbol{s}$ to $\boldsymbol{t}$ is then $\hat{\boldsymbol{q}} = (\sin\phi\,\boldsymbol{u}, \cos\phi)$" [2, p. 51].

A few trigonometric calculations show that the formula finding a unit quaternion specifying the rotation between two vectors $\boldsymbol{s}$ and $\boldsymbol{t}$ is given as [2]:

$$\hat{\boldsymbol{q}} = (\boldsymbol{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+\boldsymbol{s}\cdot\boldsymbol{t})}}(\boldsymbol{s} \times \boldsymbol{t}), \frac{\sqrt{2(1+\boldsymbol{s}\cdot\boldsymbol{t})}}{2} \right) \tag{9.4}$$

This ends our presentation of transformations. In the following sections we will give some examples of their use.

9.2 Animation and Motion Control

Building on the transformation matrices introduced in the previous section, we can create a series of frames where different transformation matrices are

used in order to give an object the appearance of continuous movement. Such a movement is referred to as animation. Animation can be a simple movement, like a ball bouncing up and down, or it can be more complex, like a person walking or a facial expressions. All movements over time qualify.

Animation is normally done in a modeling application by use of key frames. Since the object is moving over time they need to be redrawn in a slightly different position for each frame. Redrawing each moving object for each frame would be an overwhelming task. With key frames only key transforms of the objects moving are set, after that the application computes the transformations of the frames in-between.

A simple calculation of object positions in-between key frames is by linear interpolation. However to create realistic transformations it is sometimes necessary that the movement is not linear. An example could be a ball jumping up and down. If the ball is to follow the laws of physics, it will move slower around its highest point and faster at set off and just before hitting the ground. To create such effects we can use interpolation curves, which can be found using quaternion curves, see [120, 24]. Movements like this could also be simulated by a physics engine supporting gravity, which is why a good physics engine can save a lot of animation time.

In Blender key frames are controlled in the *action viewer*, where all key frames are placed. By this view key positions of objects can easily be moved or copied to different frames. Blender also supports interpolation curves in the *Ipo view*.

As we have seen in chapter 7, characters are often created from one mesh. If characters are supposed to move body parts only (for example an arm or a leg), just some of the mesh can be moved instead of the entire object. For simulating walking, for instance, parts of the character mesh must be moved in different directions at the same time. For purposes like this we can create a skeleton-like mesh defining bones that can be attached to parts of the mesh. Such a skeleton is called an *armature*. Whenever a bone in the armature moves vertices attached to it will follow. In this way we can modify only parts of the mesh influenced by the bones. An example of an armature can be seen in figure 9.2.

There is difference between animations for a movie and animations for a real-time application such as a game. Except for the fact that movie animations are much more detailed and spectacular, they are also predetermined. This means that the artist can concentrate on particular movements only. In a game, for example, movement of objects most often depends on how the user interacts with the scene. The fact that the exact movement of objects in a real-time application is unknown, means that animations must be split up into smaller pieces, each containing movement we know will be useful in many situations. Moreover animations like these must be able to follow each other in a fluent manner.

Small animation snippets, like those described above, are called cycles.

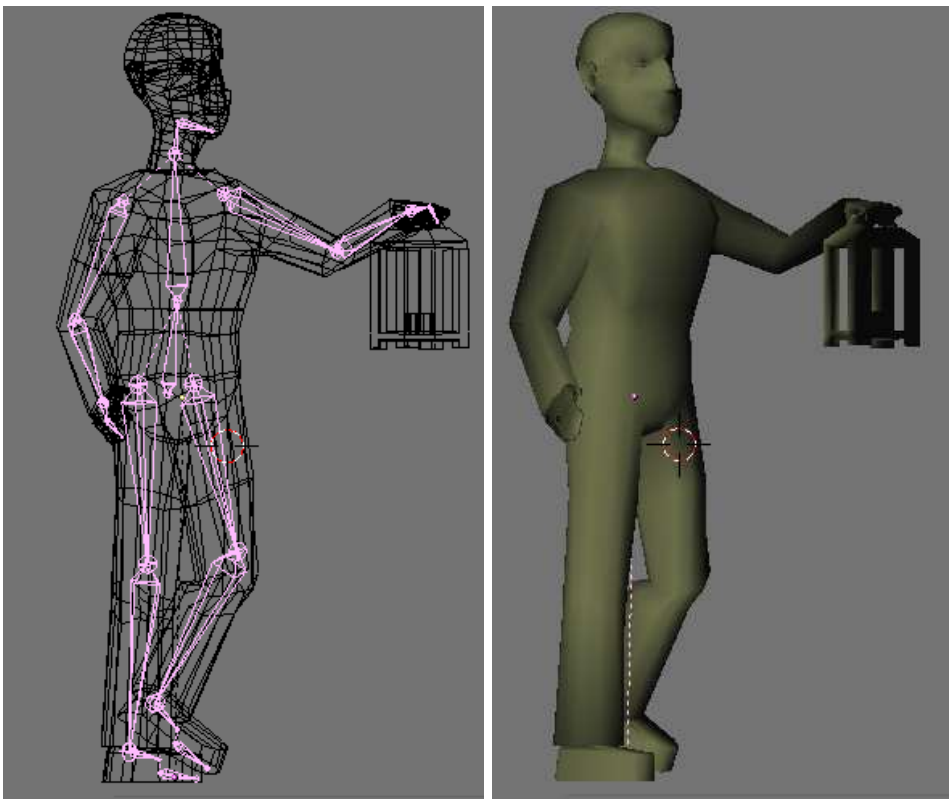


Figure 9.2: Armature of the male character in the cave scene. Notice how the mesh follows the bones.

In a game with a character controlled by a user, cycles such as a walk cycle or a hit cycle could be useful. In the example of a walk cycle the animation snippet must be able to 're-cycle', meaning that if several walk cycles follow one another, they will appear as if the character just kept walking.

To have a moving light source in our cave scene the character shown in figure 9.2 is supposed to walk about with a lantern, a walk cycle was created for this purpose and an idle cycle. Unfortunately there has not been time to export these animations to our own application. In order to show that a scene is truly dynamic, we must also be able to alter it dynamically. In real-time applications this usually happens by moving things around in the scene using some sort of input device (mouse, keyboard, etc.). Interactive control is the subject of the next section.

9.3 Interactive Control

In many real-time 3D applications a simple virtual track ball is connected to a mouse input device and used for camera navigation. This is also the

case for the application implemented during this project. The track ball we use was originally distributed during the DTU "Computer Graphics" course (02561). During this project we have modified the track ball from time to time and we have particularly adapted it to work for navigation of a chosen object (according to the current camera view) as well as it works for camera navigation. In the following we will first describe how the track ball works for interactive *camera* control, and second we will describe our expansion of it to include interactive *object* control.

The track ball is initialized by a center $\boldsymbol{L}$ around which the camera should rotate, and a distance z_{eye} which specifies how far away along the z-axis the eye point, or camera, should be placed. Most often the center is defined as the center of the scene in which the track ball is placed or the center of a particular object in the scene.

Internally the track ball has a translation vector $\boldsymbol{t} = (t_x, t_y, t_z)$ which is applied in view space, meaning that altering (t_x, t_y) results in a pan motion of the camera, while altering t_z results in a zoom motion. Pan motion is described by mouse motion when the right mouse button is down. Zoom is described by the mouse moving forwards or backwards when the middle mouse button is down.

The basis of the view space coordinate system is given by the current rotation of the camera. This rotation is specified by a quaternion $\hat{\boldsymbol{q}}_{\text{rot}}$. Moving the mouse while the left button is down will each frame provide a new mouse position to the track ball. This mouse position is projected to the sphere representing the virtual track ball. The sphere (or ball) is located at the center of the scene in view space, that is, the position the camera will always be pointing at. The new position found on the sphere will specify a new viewing direction $\boldsymbol{v}_2$. Suppose the previous viewing direction was stored as $\boldsymbol{v}_1$, then the quaternion $\hat{\boldsymbol{q}}_{\text{inc}}$ specifying the rotation from $\boldsymbol{v}_1$ to $\boldsymbol{v}_2$ is given by (9.4).

As described in section 5.1, eye point $\boldsymbol{E}$, 'look-at' point $\boldsymbol{L}$, and up vector $\boldsymbol{v}_{\text{up}}$ are sufficient to describe the camera orientation. If we choose the default orthonormal basis for the camera orientation $(\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z)$, the camera eye point, 'look-at' point, and up vector are given as:

$$\begin{aligned}
\hat{\boldsymbol{v}}_{\text{up}} &= \hat{\boldsymbol{q}}_{\text{rot}} \hat{\boldsymbol{e}}_y \hat{\boldsymbol{q}}^*_{\text{rot}} \\
\hat{\boldsymbol{L}} &= t_y \hat{\boldsymbol{v}}_{\text{up}} + t_x \hat{\boldsymbol{q}}_{\text{rot}} \hat{\boldsymbol{e}}_x \hat{\boldsymbol{q}}^*_{\text{rot}} \\
\hat{\boldsymbol{E}} &= \hat{\boldsymbol{q}}_{\text{rot}} ((z_{\text{eye}} + t_z) \hat{\boldsymbol{e}}_z) \hat{\boldsymbol{q}}^*_{\text{rot}} + \hat{\boldsymbol{L}}
\end{aligned}$$

where each resulting quaternion corresponds to a vector or a point in homogenous coordinates.

Now, all we need to do in order to rotate the camera incrementally according to the mouse motion, is to calculate $\hat{\boldsymbol{q}}_{\text{rot}} := \hat{\boldsymbol{q}}_{\text{rot}} \hat{\boldsymbol{q}}_{\text{inc}}$ for each frame. We can even let the camera spin according to a previous motion after the

left mouse button has been released by storing the incremental quaternion $\hat{\boldsymbol{q}}_{\text{inc}}$. The spin stops when $\hat{\boldsymbol{q}}_{\text{inc}}$ is reset to quaternion identity.

The extension for this track ball is to freeze the camera when the user picks an object (eg. by pressing 'p' when the mouse is located over an object), and then let the mouse control the selected object instead of the camera. What we want to specify with the mouse is the modeling transform (see fig. 5.1) of the selected object.

In order to move an object intuitively with the mouse, the motion should be controlled in view space, since this is the space where the user works. The task is now to find the modeling transform in view space.

When the track ball is frozen we store the old view transformation matrix specified by $\boldsymbol{E}$, $\boldsymbol{L}$, and $\boldsymbol{v}_{\text{up}}$ (how to find the matrix from these three is described in section 5.1). If we let $\boldsymbol{M}_{\text{view}}$ denote the view transform, then a transformation $\boldsymbol{X}$ carried out in view space is given in world space as:

$$\boldsymbol{X}_{\text{world}} = \boldsymbol{M}_{\text{view}}^{-1} \boldsymbol{X}_{\text{view}} \boldsymbol{M}_{\text{view}} \tag{9.5}$$

Luckily the view space transformation consist of translation and rotation of the camera only, therefore we can find $\boldsymbol{M}_{\text{view}}^{-1}$ using (9.1).

Setting the center of the track ball to the center of the object in world space $\boldsymbol{C}_{\text{world}}$, we can specify the translation of the object in view space $\boldsymbol{T}_{\text{view}}$ according to pan and zoom of the track ball:

$$\boldsymbol{T}_{\text{view}} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where pan becomes moving the object parallel to the view plane and zoom becomes moving the object along the direction from the current camera position to the object center. This results in a quite intuitive translation of the object according to mouse movement. The next step is to rotate the object.

In order to rotate the object intuitively in view space we must first move it to the origin of the view space coordinate system. Having the center of the object in world space coordinates $\boldsymbol{C}_{\text{world}}$, we can transform it to view space coordinates as follows:

$$\boldsymbol{C}_{\text{view}} = \boldsymbol{M}_{\text{view}} \boldsymbol{C}_{\text{world}}$$

The translation is then simple because the origin of the view space coordinate system is now in $(0, 0, 0)$ relative to $\boldsymbol{C}_{\text{view}}$. Translation of the object to the origin of view space is:

$$\boldsymbol{T}_{\text{view},-\boldsymbol{C}} = \begin{pmatrix} 1 & 0 & 0 & -\boldsymbol{C}_{\text{view},x} \\ 0 & 1 & 0 & -\boldsymbol{C}_{\text{view},y} \\ 0 & 0 & 1 & -\boldsymbol{C}_{\text{view},z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and translation back to the previous object position is:

$$\boldsymbol{T}_{\text{view},\boldsymbol{C}} = \begin{pmatrix} 1 & 0 & 0 & \boldsymbol{C}_{\text{view},x} \\ 0 & 1 & 0 & \boldsymbol{C}_{\text{view},y} \\ 0 & 0 & 1 & \boldsymbol{C}_{\text{view},z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since we are looking at the object along the z-axis in view space, the local coordinate system around which we want to rotate the object will have a z-axis pointing in the opposite direction. If the object was positioned exactly in the 'look-at' point, the basis of the *object* coordinate system would be exactly opposite the basis of the *view space* coordinate system. Therefore, when we rotate the track ball, a reasonable approximation to a rotation of the selected object instead of the camera is given in view space as the conversion of $\hat{\boldsymbol{q}}_{\text{rot}}^{-1} = \hat{\boldsymbol{q}}_{\text{rot}}^{*}$ to a rotation matrix $\boldsymbol{M}^{q_{\text{rot}}^{*}}$ according to (9.3).

The rotation should as mentioned be performed at the origin of view space. The final transformation of the object in view space is, therefore, given as:

$$\boldsymbol{X}_{\text{view}} = \boldsymbol{T}_{\text{view}}\boldsymbol{T}_{\text{view},\boldsymbol{C}}\boldsymbol{M}^{q_{\text{rot}}^{*}}\boldsymbol{T}_{\text{view},-\boldsymbol{C}} \tag{9.6}$$

Inserting (9.6) in (9.5) results in the transform of the object in world space. If the object had no modeling transform to begin with. We can let $\boldsymbol{X}_{\text{world}}$ specify the new modeling transform of the object. For this project we will always assume that the object has no other modeling transform when this track ball motion control is applied.

This chapter has introductorily shown the impact of interaction and animation on real-time graphics. In the following chapter we will give a brief tutorial on Blender modeling, since one part of this project has been to show the work flow from modeling to rendering.

Chapter 10

Modeling in Blender®

"Well" said Owl, "the customary procedure in such cases is as follows."
"What does Crustimoney Proseedcake mean?" said Pooh. "For I am a Bear of Very Little Brain, and long words bother me."
"It means the Thing to Do."
"As long at it means that, I don't mind," said Pooh humbly.

A. A. Milne (1926): *Winnie-the-Pooh*

After introducing some of the different aspects in modeling we turn to the modeling application. As mentioned before we have used a modeling application called Blender to create most of our test scenes. One of the objectives of this project is also to create a platform for game creators with little or no financial means. Being free of charge Blender plays an important role in this objective. In this chapter we will give an introduction to the basic functionalities in Blender. It is described how actions related to the above chapters such as object modeling, material and texture settings, and animation are carried out. Blender has many functionalities in addition to those described here, for knowledge about these and more advanced features we refer to `www.blender.org`, which holds many excellent tutorials. A few of the additional functionalities will be mentioned at the end of the chapter without any details.

During the project a newer version of Blender was released. This chapter builds on Blender 2.33a which is currently the newest version available, but many of the scenes were originally created using Blender 2.24. We have found no problems in using the newer version of Blender for the last part of the project though we might have missed some of the improvements that could have eased our work.

First step is to give an overview of the menus and windows in Blender. Section 10.1 will introduce all the most important menus and views of the Blender workspace. This section will also describe how to change the view in order to navigate in the scene, and how to set up the camera.

Section 10.2 will describe how models are created from primitives, or polygon by polygon, and transformed.

After creating objects the next step is to give them material properties in order to make them look more realistic. Section 10.3 will describe the basic material and texture functionalities in Blender.

Following the order of chapters in this part, the next section (10.4) will address animation in Blender.

When a scene has been created in Blender, we want to export it to our own render application. This is the subject of section 10.5, which will describe how to use the export script that we have found for export from Blender to the .x3d file format. The import of .x3d files to our own application will also briefly be mentioned.

The last section (10.6) of this chapter will mention some of the additional features available in Blender. No details will be provided.

10.1 Blender Navigation

Opening Blender for the first time can be quite confusing. The user interface (see figure 10.1) has a vast amount of buttons and panels.

For this section the most important window will be the one in the middle

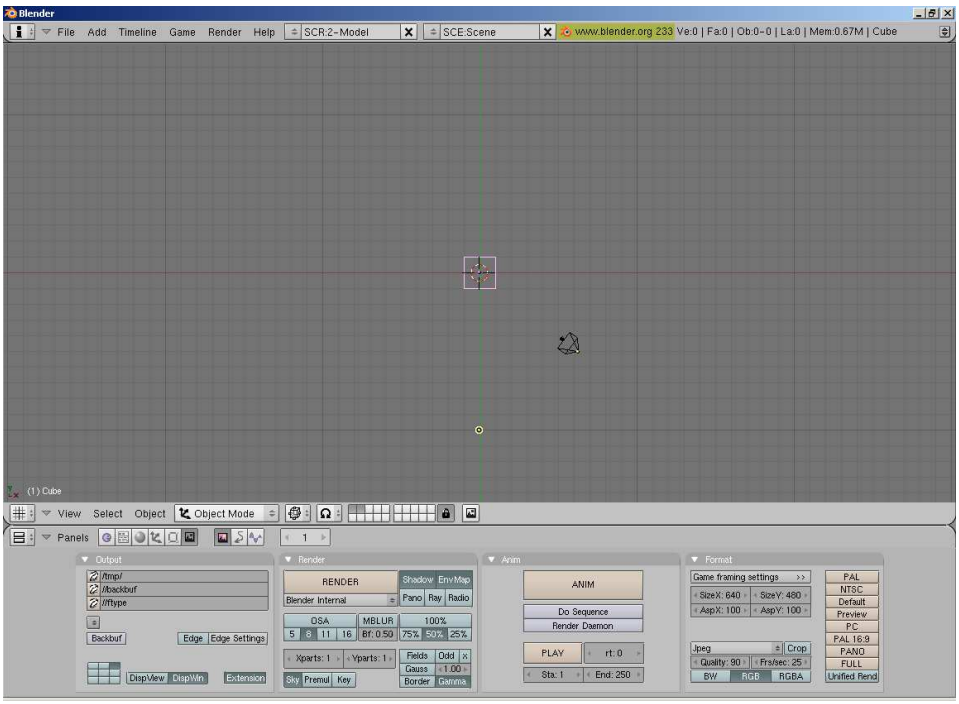


Figure 10.1: The opening view of Blender 2.33a.

showing the scene. A new Blender scene always contain three objects; a cube, the pink quad in the middle (the color tells us that the object is selected), a camera, represented by the pyramidal object with the arrow representing the up-vector and a light source, which is the yellow ring with a dot in the middle. To see the scene press the `RENDER` button below.

One important thing to mention is that the active window of Blender is the one where the cursor is currently over. This might be a new experience for MS Windows users, where windows will most commonly have to be selected. For LINUX and UNIX users nothing is new.

The Blender interface very much makes use of shortcut keys. When reading Blender instructions (in tutorials and the like), shortcut keys are referred to as the key name followed by `KEY`. So for example `AKEY` means pressing `A`. For navigation purposes the numeric keypad is used, keys here are addressed with `PAD` in front of the name of the key to press instead of `KEY` behind it. Pressing 5 on the numeric keypad would, hence, be referred to as `PAD5`. As with the keyboard short cuts, there are short names for the mouse as well: `LMB` means left mouse button, `RMB` is right mouse button and `MMB` is middle mouse button. Lots of mice only have two buttons, in such cases the `MMB` equals `LMB` while holding down `ALT` (`ALT+LMB`).

There are two ways of navigating your blender scene. You can either use the mouse or the numerical keypad. Table 10.1 describes the options Blender

gives for scene navigation.

Key	Functionality
PAD/	Show local view of selected objects (hide the rest of the scene)
PAD*	Copy the rotation of the selected object to the current 3D window
PAD-	Zoom out
PAD+	Zoom in
PAD.	Center and zoom in on selected object
PAD5	Switch between perspective and orthogonal view
PAD9	Force complete recalculation and redrawing of scene and animations
PAD0	Show camera view of the current active camera
CTRL-PAD0	Set current camera. Note that any object can be set as camera
ALT-PAD0	Return to previous camera (only works with camera objects)
PAD7	Top view
SHIFT-PAD7	Down view (opposite top view)
PAD1	Front view
SHIFT-PAD1	Back view
PAD3	Right view
SHIFT-PAD3	Left view
PAD2	Rotate downwards
PAD8	Rotate upwards
PAD4	Rotate left
PAD6	Rotate right
SHIFT-PAD2	Translate down
SHIFT-PAD8	Translate up
SHIFT-PAD4	Translate left
SHIFT-PAD6	Translate right
MMB	Rotation (track ball or axis aligned)[1]
SHIFT-MMB	Translation
CTRL-MMB	Zoom[2]

Table 10.1: Shortcuts for navigation in Blender.

Blender has fourteen different views that each can be chosen through

[1] Whether it should be track ball or axis aligned rotation is set in the user menu. The user menu is found when dragging down the top menu under "Views & Controls". Other interesting settings are found in here as well.

[2] If you have a scroll mouse the scroll button will usually be attached to the zoom functionality as well.

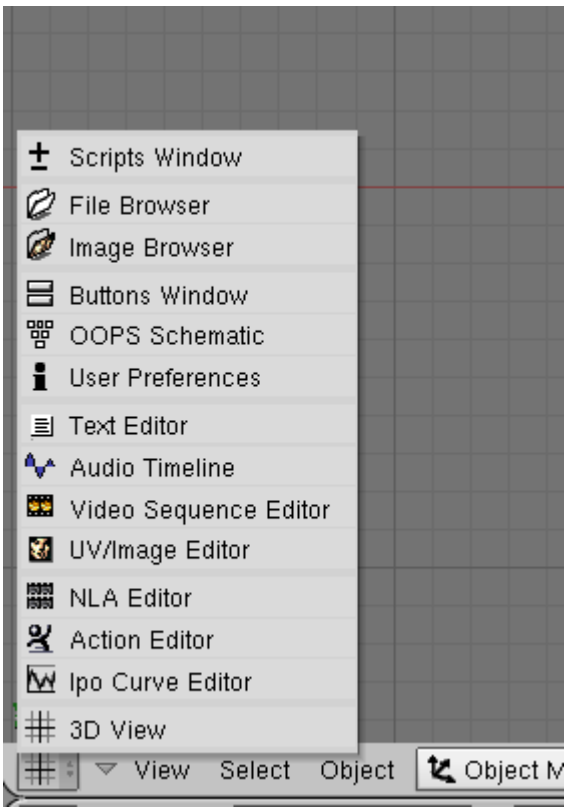


Figure 10.2: The fourteen different views available in blender.

the leftmost icon in the statusbar of a window. Note that there are three windows available when opening Blender, each with their own statusbar (the last window is hidden above the menubar in the top, to make it visible simply drag down the menubar). In each of these windows it is possible to change the view to something else if desired. The fourteen views available are shown in figure 10.2. Not all views are relevant to this project, those that are will be mentioned eventually.

Sometimes it is useful to have more views at the same time. This is easily achieved by right clicking at the edge of a window. A split screen popup will appear (figure 10.3a). Pressing "Split Area" will divide the view in two. This procedure can be followed several times if more views are desired. When the multiple views are no longer useful, we can join areas again by right clicking on the border between them and selecting "Join Areas" in the popup menu (figure 10.3b). The view that has last been in use, will fill out the entire area. A fast way to expand the working field shortly, without joining views, is by use of the full screen mode reached through `CTRL+UPARROW`, when pressed the active window will take up the entire screen, when pressed again the application will return to its previous state.

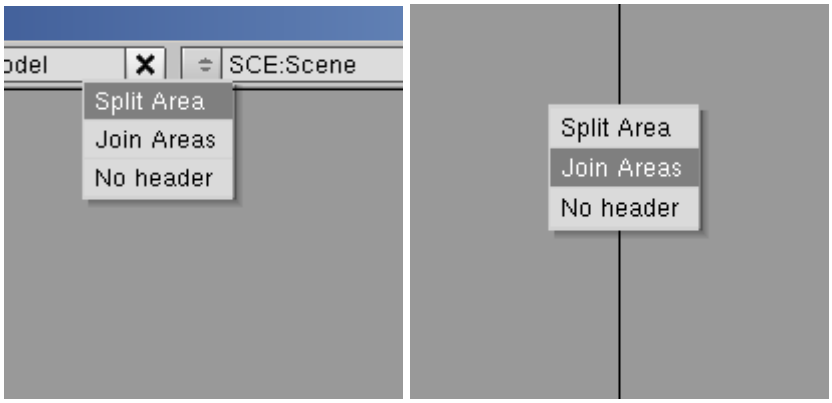


Figure 10.3: From left to right: (a) Splitting areas. (b) Joining areas.

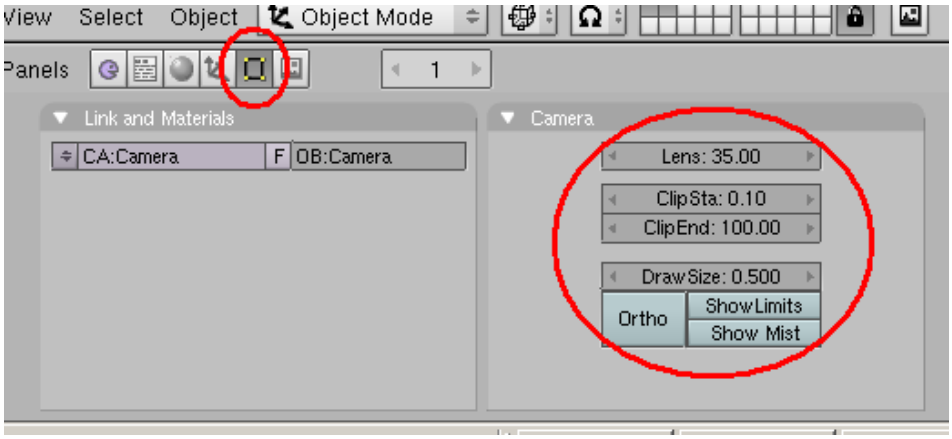


Figure 10.4: Camera settings. Remember to have the camera selected and to select the edit panel.

The scene is made visible through the camera and is illuminated by the light source. The camera can be moved about and rotated like any other object in the scene, how this is done is described in the next section. When a camera is selected (objects are selected with `RMB`) the camera parameters can be set. To set up the camera go to the edit panel by pressing `F9` or selecting the icon with a quad and four dots marking its corners (see figure 10.4).

The camera settings are available in the camera panel (marked on figure 10.4). Here you are able to set the lens angle ("Lens"), which determs the field of view. Near and far clipping planes ("ClipSta" and "ClipEnd" respectively) can also be set. "DrawSize" is simply the size of the camera object on the screen. "Ortho" enables orthographic rendering, while "Show Limits" and "Show Mist" makes the boundaries of the visible area show in the modeling view.

Normally one camera is sufficient, but it is possible to have several cameras placed in different locations with the right settings ready. Rendering then happens through the camera currently active. To set another camera active, simply select the camera and press CTRL+PAD0. Then the current camera is the selected one. Actually all objects can be chosen as a camera like this, even though they are not a camera object. Be aware, though, that camera settings only are available if the object set as current camera is a camera object.

Other small features and short cut keys that are nice to know are presented in tables 10.2 and 10.3.

This section has described the crux of the tools needed for Blender navigation. Many functionalities, buttons, panels, and views have not been described here. To describe all the different functionalities would be going too far. More excellent tutorials and information can be found at www.blender.org. This is where the modeling application also can be acquired. The following section will give a short tutorial on modeling in Blender related to the creation of models for this project.

10.2 Modeling in Blender

The techniques used for modeling in this project have already been described in chapter 7. The purpose of this chapter is to describe how these techniques are carried out in Blender.

The initial Blender scene already contains a camera, a light source, and a cube. The cube is selected, which is indicated by its purple color. The ability to select objects is, of course, crucial. Table 10.4 will explain the different possibilities of object selection.

The first modeling method in chapter 7 suggests use of primitives. The easiest way to add new primitives is to press SPACE in the modeling view. By this action you get the add menu, select the "Add→Mesh" submenu and pick the primitive you would like to add (see figure 10.5). The new primitive will be placed where the center is currently placed.

Objects can be moved around in the scene arbitrarily, or rotated, or scaled. The different possible object transformations can be seen in table 10.5. All transformations will be carried out on the selected objects.

After the shortcut key has been pressed the object will follow the mouse until LMB is pressed. If RMB is pressed the transformation will be canceled. To control transformations better hold down the CTRL key before the mouse is moved. Now the object will only transform in a unit grid. Appropriate units

[3]Paint selection provides the user with a brush like area to select objects. The area can be resized using the zoom functions or a scroll button. This selection feature is only available in edit mode (discussed later). To end paint selection mode press RMB.

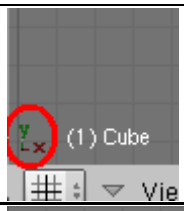
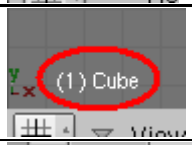
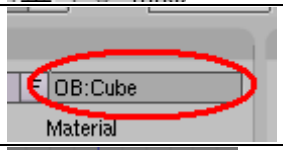
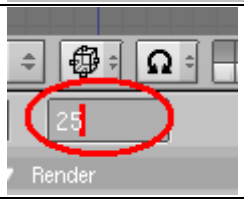
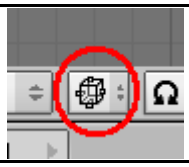
Feature	Description
Deleting a selected object (in object mode).	XKEY
To start all over.	CTRL+XKEY
The most used menus are available through one shortcut key.	SPACE
The coordinate system vectors are always shown in the lower left corner of the modeling view.	
The name of the currently selected object is shown in the lower left corner of the modeling view.	
An object can be named in the edit menu (see figure 10.4). Double click in the box.	
Most numbers can be set specifically by SHIFT+LMB. Press ENTER or LMB to confirm changes. The example to the right shows how the current frame is altered.	
There are four different centers (this can be the source of many mysterious errors. For example when rotating objects). Center can be chosen from the lower tool bar.	
How the scene is presented in the modeling view can be chosen in the lower menu bar.	

Table 10.2: Additional Blender features and shortcut keys. Continued in table 10.3 on the following page.

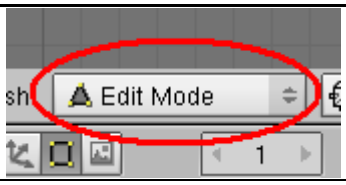
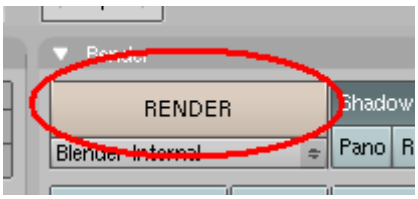
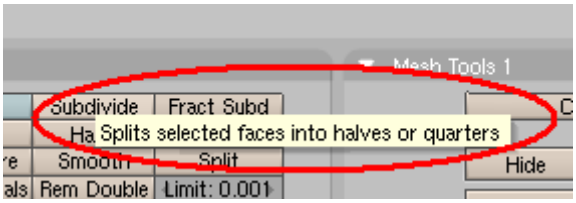
Modes can be selected through the mode selection bar.	
To view the scene output we can use Blender's render engine. The rendering settings appear by pressing `F10`. To render an image from the current camera press the `RENDER` button in the middle menu.	
Undo is not really available, instead auto save is enabled. Auto save settings are found by dragging down the top menu bar and select auto save.	
If there is doubt about what a button does, a small help popup will appear in many cases if the mouse is held over the button for a short time (the example to the right shows the help text for subdivision).	

Table 10.3: Additional Blender features and shortcut keys. Continued from table 10.2 on the previous page.

Selection Type	**Shortcut Key**
Select object	`RMB`
Select additional object	`SHIFT+RMB`
Deselect object	`SHIFT+RMB` (on selected object)
Deselect all objects	`AKEY`
Select all objects	`AKEY` (when none selected)
Area drag selection	`BKEY` then `LMB` (hold in and drag)
Area drag de-selection	`BKEY` then `RMB` (hold in and drag)
Paint selection[3]	`BKEY` twice - `LMB` selects
Paint de-selection[3]	`BKEY` twice - `SHIFT+RMB` de-selects

Table 10.4: Selecting objects in Blender.

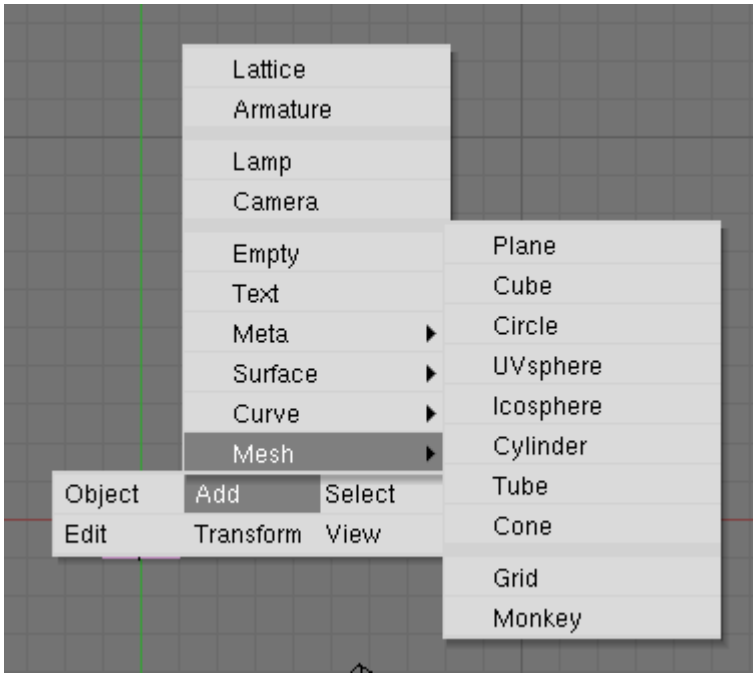


Figure 10.5: The easiest way to add a new primitive. The menu pops up when `SPACE` is pressed.

Object Transformation	**Shortcut key**
Translation	`GKEY`
Rotation	`RKEY`
Scaling	`SKEY`
Mirror	`MKEY`

Table 10.5: Blender transformations

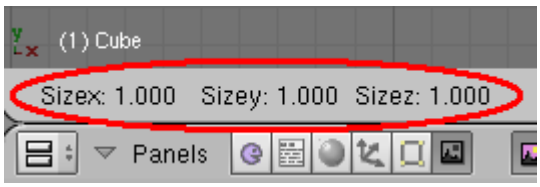


Figure 10.6: Units of transformation. This example is from scaling.

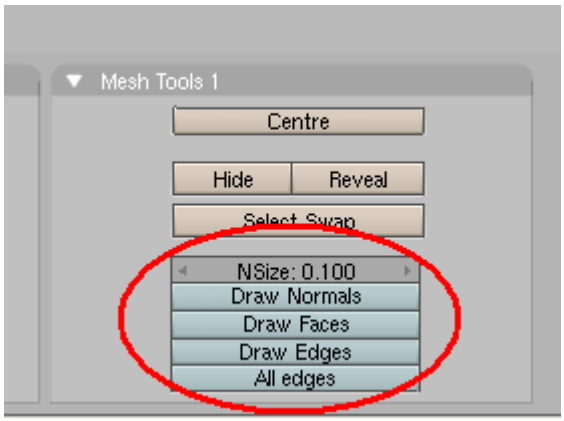


Figure 10.7: Settings for what should be visible in edit mode (NSize is the length of the normals).

are shown in the lower left corner (see figure 10.6). Rotations will always be in the same plane as the view is in.

It is also possible to restrict transformations like translation and scaling to follow an axis. After pressing the wanted transformation press the `XKEY`, `YKEY`, or `ZKEY` to select which axis to follow.

To manipulate single (or more) vertices you will have to enter edit mode. To enter edit mode press `TAB`. Now all vertices are visible. Vertices can be transformed and select in the same way as objects. When vertices are manipulated triangles between them will automatically follow.

In edit mode it is possible to view faces and normals. Settings for what to be visible can be found under the edit menu (the same button as for setting up the camera, see figure 10.4 in section 10.1), these settings can be seen in figure 10.7.

A small remark about the center of the object marked by a small dot (pink when the object is selected and yellow when deselected): When vertices are transformed in edit mode, the center stays where it is, while outside edit mode it follows the transformations. Therefore it is advisable not to translate the entire object in edit mode unless you want to move the center. Another important thing is that the center will stay, representing the object even if all vertices are removed. In other words an object cannot be deleted completely

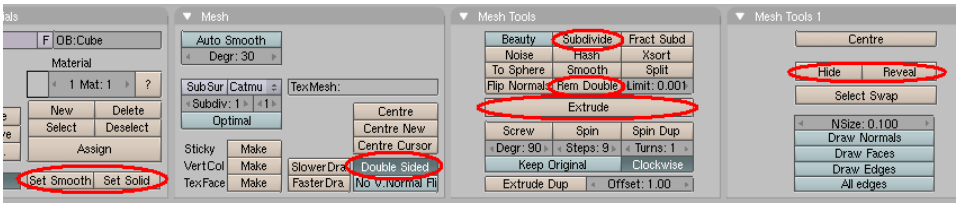


Figure 10.8: Features available in edit mode.

in edit mode, always do this in object mode if the object should be entirely removed.

Apart from the transformations available for objects, there are some extra possibilities in edit mode for manipulating vertices also found under the edit menu. Some worth noticing are: The "Set Smooth" option which smoothes the object ("Set Solid" reverses this), "Subdivide" which subdivide selected vertices, "Extrude" generating an extrusion from selected surfaces (shortcut key is the `EKEY`) and "Rem Double" adding together selected vertices lying close to each other. Another good feature is the "Hide"/"Reveal" functionality. Using these buttons selected vertices can be temporarily hidden, which makes it easier to manipulate vertices in more complex meshes. Note also the "Double Sided" button, which is selected by default. Double sided means that faces are visible from both sides. This can be a problem, since, for example, shadow volume calculations only works for one-sided faces, which can create troubles when using objects outside Blender. Besides it is crucial that all faces have the correct orientation if we want do back face culling, which we always want to do if possible. A good advice is therefore to deselect this feature. All features mentioned in this paragraph are outlined in figure 10.8.

Another method, which was described in chapter 7, is to model after a two-dimensional sketch. Loading a picture into the background of Blender is done as follows: Select "Background Image..." in the view menu in the lower selection bar (figure 10.9). The dialog shown in figure 10.10 will appear. "Use Background Image" reveals the other buttons and enables the use of a background image. To select an image press the folder icon to the right of "Image:". "Blend" determines transparency of the image (0 equals no transparency and 1 makes the image invisible). "Size" sets the size of the image "X Offset" and "Y Offset" places the image in the scene.

The image remains at the same position from all angles, meaning that no matter which angle you choose to see the scene the image will not transform. It is like a wall paper in the background. The image will only be visible when the view is perpendicular to one of the axes (`PAD7`, `PAD1`, `PAD3`, `CTRL+PAD7`, `CTRL+PAD1`, or `CTRL+PAD3`).

In the following we will describe an example of modeling according to a background image. This is the method we have used for most of the objects created for our test scene. We will follow the example pictured in figure 7.5.

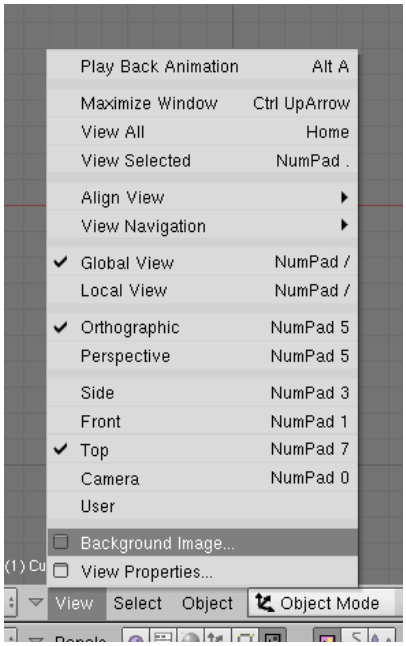


Figure 10.9: Select "Background Image..." to load an image into the Blender background.

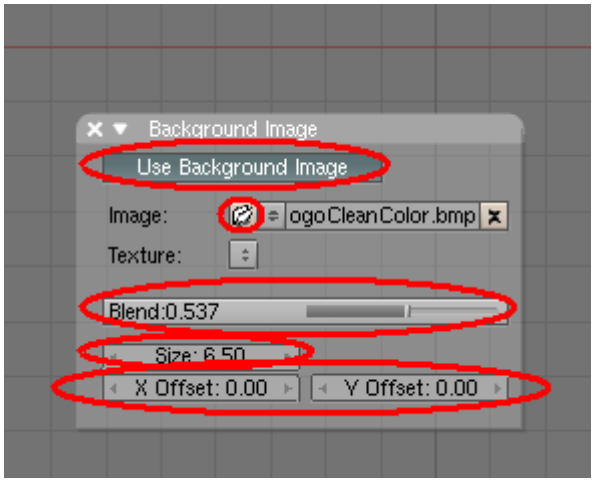


Figure 10.10: The "Background Image" dialog.

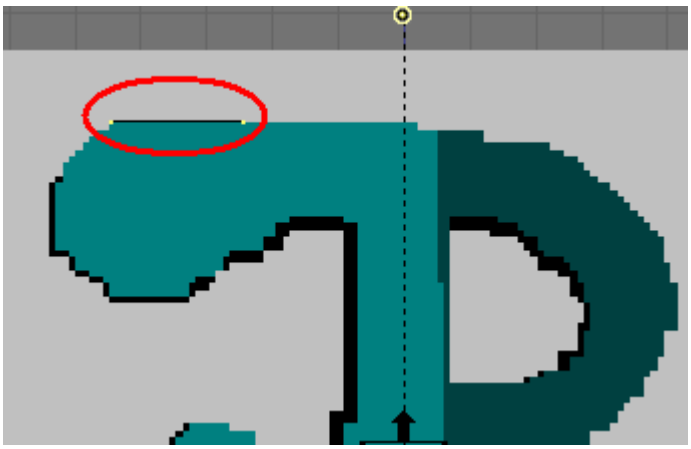


Figure 10.11: A line following a line in the background image.

The background image (which is the logo for our demonstration application) is loaded as described above. To start modeling add a plane from the SPACE pop up menu (see section 10.1).

Now, enter edit mode (TAB), select all vertices in the plane (AKEY), and move the plane to a favored position from where you want to start modeling (press GKEY to translate). When one of the lines of the plane is aligned with one of the lines of the background image erase the two vertices not connected to that line (deselect all vertices by pressing GKEY and select the vertices with SHIFT+RMB and press XKEY). We now have a line following a line in the background image that we want to model after, see figure 10.11.

Then we can draw the contour of the object presented in the background image by extruding or translating vertices one at a time (GKEY or EKEY, selection by RMB). When we want to connect the last to vertices and close the curve, we select both and press the FKEY. The same technique could be used for assembly of faces of the primitive man in figure 7.3. After drawing contours for each part of the image the result is as seen in the top right screen shot of figure 7.5.

In this example we have no more use of the background image so we remove it and turn to the right view (PAD3). Now, all that we can see is one straight line with vertices randomly scattered since our object is all flat. To give it the third dimension select all vertices (press AKEY). Press EKEY to extrude all vertices at once and hold down CTRL while dragging the mouse. We now have a three dimensional model of the JR logo. Turn the scene a little and see for your self (PAD4 or PAD6), or see the result in figure 7.5.

When pressing the ZKEY we can see the faces of the object. Blender has only created faces in the extruded direction. To make the object solid we must add the appropriate faces by selecting the vertices between which we want to create a face and press FKEY.

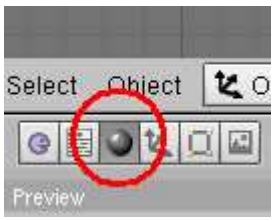


Figure 10.12: This button picks the material menu, shortcut key `F5`.

Faces can only be drawn between three or four vertices at a time so this easily becomes the most time consuming process when modeling. Another important thing to keep in mind is the double sided feature that Blender has toggled on as default. Objects being double sided (shaded on both sides) inside Blender are fine but when they are exported they usually only have one side, meaning that some of them might turn the wrong way. This results in transparent faces making the object full of holes. The best way to avoid this is to toggle the double sided feature off right away in the edit mode under the edit buttons menu, see figure 10.8.

We constantly must keep track of the normals otherwise they might point in the wrong direction. To keep track of the normals select "Draw Normals" in the rightmost menu (again see 10.8). Whenever the normal points inwards we should flip it. Do this by selecting the vertices that represent the corners of the face which has a normal turning inwards and click the "Flip Normals" button.

This ends our presentation of object creation and manipulation in Blender. Other possibilities are also available such as line sweeping and curves. These are not described in this report, for more information we refer, again, to `www.blender.org`.

When an object has been created, it should have material applied to it. Setting materials in Blender is described in the next section.

10.3 Material Settings in Blender

After creating the objects for a scene, we would like to give the scene some life by making the objects look real and colorful. This is done through materials. Translation between Blender material parameters and usual Phong material parameters is described in section 8.2. The translation is, however, not completely consistent, therefore we can not guarantee that materials will look exactly the same in our render engine as they do in Blender.

Materials for an object are created through the shading menu, which is selected through the small sphere in the lower menu bar (see figure 10.12). The menu is also available by the shortcut key `F5`.

For each new object a new material must be added. This is done simply

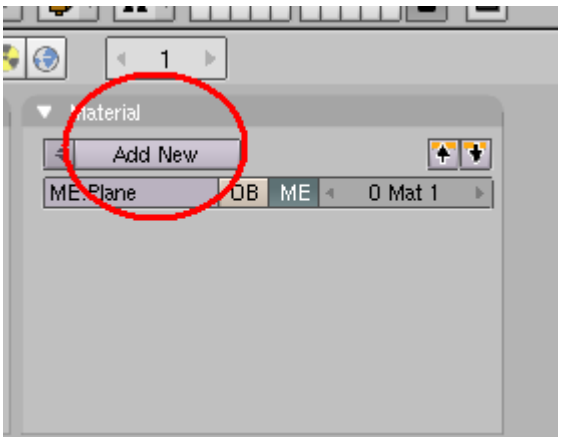


Figure 10.13: Adding a new material.

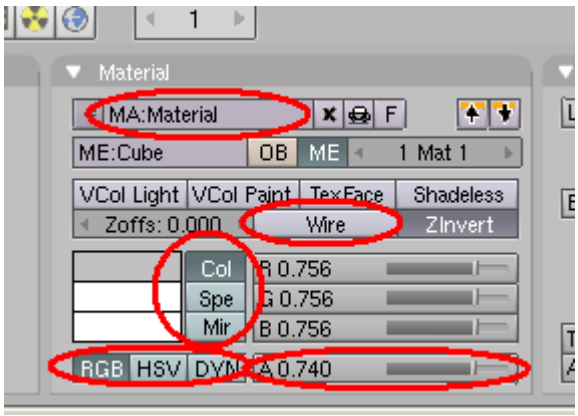


Figure 10.14: Main Blender material settings.

by pressing add new in the material menu, see figure 10.13.

When a new material is added a whole bunch of new options appear. In the following we will describe some of them.

Like objects, materials can be named under the material settings in the MA field, where most other functionalities concerning color and appearance are found. Here the diffuse color "Col" and the specular color "Spe" of an object are set. Note that it is possible to set colors using either the RGB or the HSV color space as described in 8.1. The "DYN" button activates physical settings related to the material and the "Wire" button enables rendering in wire frame mode. "A" is transparency settings. See figure 10.14.

In the "Shaders" menu all settings regarding ray tracing are found. Here there is also a slider for reflectance which is further outlined under the mirror transparency menu. Last we have the texture settings. Most changes in the material settings are made visible in the preview menus in the left side part of

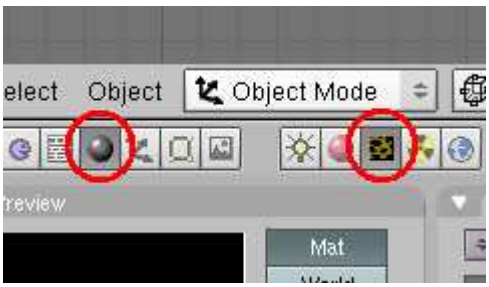


Figure 10.15: The texture menu. The right set of menu selection icons appear only when the first material icon in the permanent set of menu selection icons have been chosen.

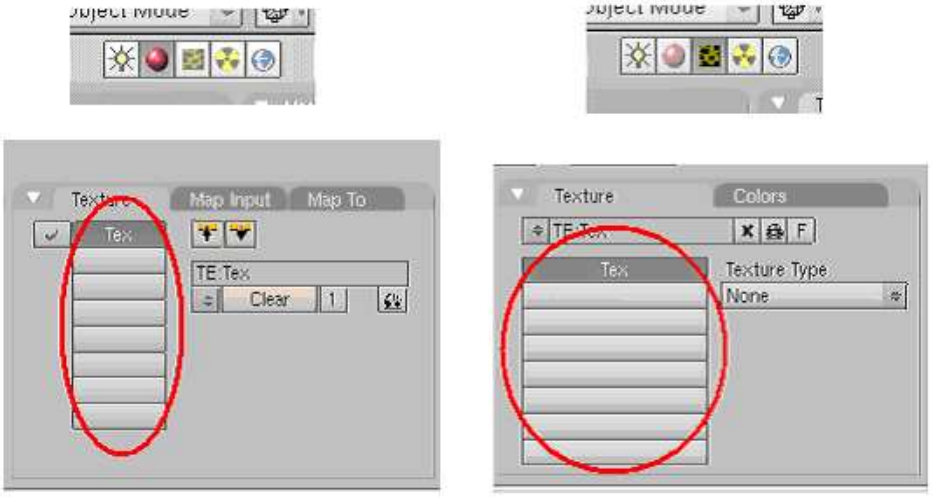


Figure 10.16: Textures are set up in the "Texture" menu. Each texture refers by name to textures added under the material menus. From left to right: (a) The texture menu of the material view. (b) The texture icon menu.

the screen. Since we are using our own renderer these are not too important to us.

The rightmost menu is for textures, that is, for associating textures with a material. This is done by pressing the "Add New" button. When this button is pressed more menus will appear. They hold settings for how the texture is mapped to an object to which the material is applied. How the texture should look, is set up in yet another set of menus, which are found in the menu bar above the extra set of menu icons that appear when the material setting have been chosen. Instead of the red sphere choose the leopard skin. See figure 10.15.

Textures are arranged according to their name. In the texture menu of the material view (fig. 10.16a), we have a list of textures that are attached to the material. The same list is found in the texture icon menu (fig. 10.16b).

Textures can be pre-generated pictures giving a higher level of detail to

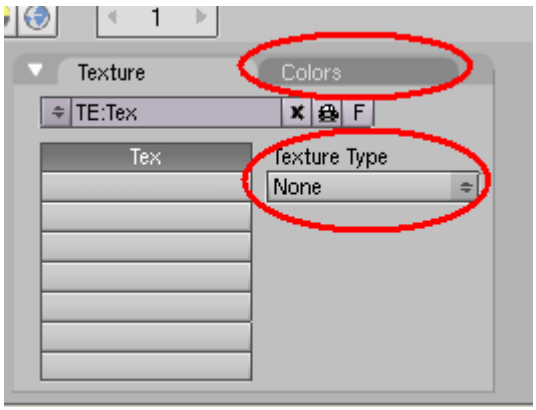


Figure 10.17: Texture patterns are selected in the "Texture Type" selection bar. The color of the non-transparent part of a texture pattern is the diffuse color of the material.

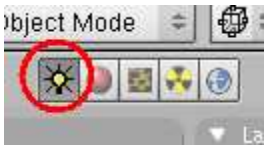


Figure 10.18: The light setup menu (only available when a light source is selected).

an object. They can also be computer generated patterns to simulate some kind of material, for example marble or wood. Here the texture consists of a partly transparent pattern, where the color of the non-transparent part is the diffuse color set in the colors menu. The "Texture Type" selection bar, shown in figure 10.17, makes different texture types available. Through this we can, for example, choose an image for the texture. When a texture type has been selected new menus will appear for individual settings.

Textures are not exported/imported to our render engine, so we will not go into any details.

A few more settings should be mentioned. First of all light settings are important for the appearance of the scene. Light settings are found through the light bulb in the icon selection bar, but are only available when a light source is selected (see figure 10.18). In the initial scene we have a light source (the circle with a dot in the middle) more light sources can be added like primitives through the popup menu activated by pressing `SPACE` (see figure 10.5 in section 10.2).

When a light source is selected and we have chosen the light setup menu we can modify the light source. Under light settings we can select the type of light source that we want (point light, area light, spot light, sun (ie. directional light), etc.), we can set distance, color, energy (intensity), and even textures can be attached. All light sources are exported/imported to our render engine. The export and import only distinguish between spot

Figure 10.19: Background is set in the world settings menu.

Figure 10.20: Render menu icon.

lights and point lights, the rest will be exported as the one of these that best fit.

The background of our rendered image can be set in the world settings chosen by the icon which is a globe (figure 10.19). World settings are simply settings for the background, which can be set to anything from a sky simulation to a single color or an image. The background settings are not exported.

The last button in the icon menu bar is radiosity settings, which are not used in this project.

To see the results from our modeling efforts we turn to the rendering menus. These are found in the stationary icon selection bar (the one that is always visible). The render menus are selected through the small landscape icon (figure 10.20).

When pressing the big render button in the middle of the render menu window (also seen in fig. 10.1), Blender will generate a picture from the current camera view. Different settings can be toggled on and off, such as shadows, ray tracing, radiosity, etc. The output picture is shown immediately after calculation has been carried out in a popup menu. The size and format of the output picture is set in the rightmost menu called format. The rendering image can be saved by pressing `F3`. The rendering functionality is good for previews of the scene. As it is now, it is hard to use the render output for comparisons with the results of our render engine, since the export/import scripts change too many things with regards to material and light settings.

After setting materials, we can start animating the scene. Blender animation is the subject of the next section.

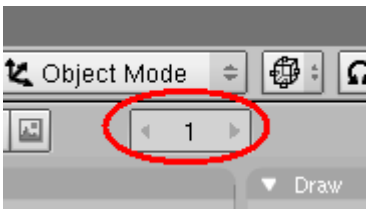


Figure 10.21: The frame counter.

10.4 Blender Animation

To create a dynamic scene we can animate our objects. The basic animation concepts are described in chapter 9, this section will follow chapter 9 and describe how animation is carried out in Blender.

Objects can be animated in Blender using key frames. The frame counter is found in the lower menu bar, see figure 10.21.

Before creating any key frames it is a good idea to bring up the key viewer. For this purpose we split the modeling view in two (as described in section 10.1, right-click on the edge of the window and select split). We want to make one of our windows a key viewer, to do this; go to the window type selection icon in the lower left corner of the window and select the "NLA Editor" (see figure 10.22).

To make the first key frame go to the modeling view and select the object that should be animated. When the object is placed in its first position, we start applying transformations and rotations by pressing `CTRL+AKEY` followed by "Ok". Now we are ready to create our first key frame. This is done by pressing the `IKEY` and selecting "LocRotSize" to save position size and rotation of the object. A small yellow box will now appear on a line for the selected object in frame one. This indicates that a key frame has been saved, See figure 10.23.

Now, move the frame counter to the frame where the next key frame should be. In this frame change the object to the next position, press `IKEY` and save another key frame. Moving through the different frames it is seen that the object will change transformation between the two key frames. To make the animation longer or more detailed save more key frames in-between the other frames. Key frames can be moved about, deleted, and selected in the NLA editor exactly as in the modeling view. Zoom functionalities can also help creating a better overview of the key frames, these works in the same way as in modeling (see section 10.1). To stretch the view in one direction hold down `CTRL+MMB` and drag the mouse, either horizontally (x direction) or vertically (y direction).

The course of the transformation between the two key frames can be changed in the "Ipo Curve Editor" found in the window type selection menu just like the "NLA Editor" (see figure 10.22). The Ipo curve editor shows

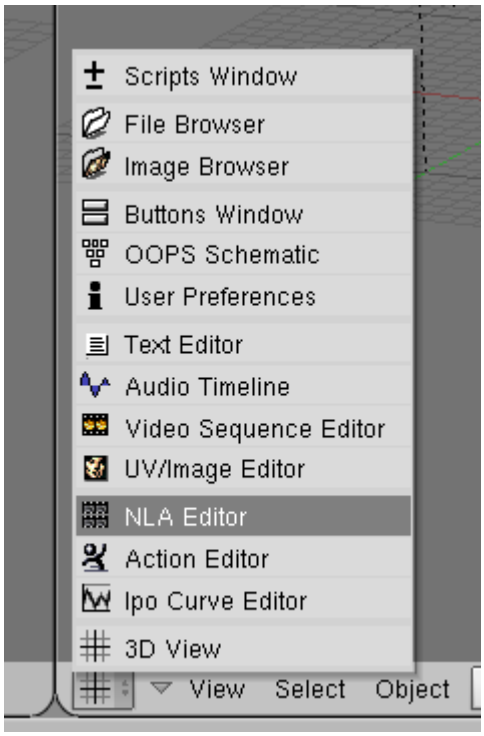


Figure 10.22: Select the "NLA Editor" to get an overview of the generated key frames.

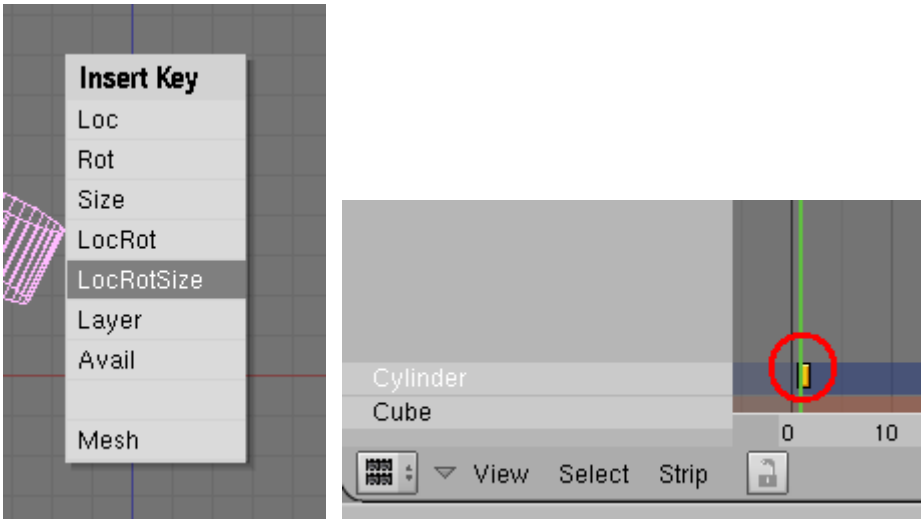


Figure 10.23: Saving a key frame by pressing the `IKEY`. Here a key frame is saved for a cylinder.

a curve for each transformation of the selected object. As we have saved keys for location, rotation, and scaling we have a curve for each of these transformation types on each axis (a total of nine curves). Figure 10.24 shows how the Ipo curve editor looks like in our case.

The curve overview may be a bit confusing since many curves sometimes cross each other. To concentrate at one transformation at the time, we can click different curves on and off to the right using the Blender selection and deselection functionalities. To select a curve press the text and not on the colored square in front of the text. Pressing the colored square selects the curve instead. When a curve is selected we can transform it as we transform objects in the modeling view. To change the form of the curve press `TAB`. It is now possible to alter single points on the curve and the curve slope.

As presented in chapter 9 we can create an armature for more complex meshes. To create an armature press `SPACE`, then select "Add" and then select "Armature" (see figure 10.25).

When you drag the mouse a bone will appear drawn from the first point indicating the beginning of this bone chain. Each time `LMB` is pressed one bone ends and a new bone begins. The bones are linked together in a chain. Around each link point we can move the armature freely; otherwise it has only stiff lines of connection. The links of the armature can be translated as other vertices in the modeling view. A bone is selected by selection of the two links that it is connecting. Bones can be rotated and scaled. A typical armature for a human is shown in figure 10.26.

Sometimes it is appropriate to connect links without actually having a bone between them (these are the dotted lines in figure 10.26), this is done by parenting. To make one bone parent of another, we must be in edit mode (Blender automatically changes to edit mode when we choose to add an armature). In the edit menu selected through the quad in the icon menu bar (see figure 10.4 in section 10.1) we can set the parent of all selected bones. When a bone is made child of another bone it will automatically follow changes of that bone. Figure 10.27 shows the parenting of the armature in figure 10.26. Notice that the pelvis is not a child of any other bone, this means that when we move the pelvis the rest of the armature will follow without transforming.

The names of the bones are quite important in the parenting process. The name of a bone is set in the BO field by `LMB` clicking, exactly as when naming objects, materials, and textures. To name the bones it is a good idea to select them one by one, only selected bones appear in the armature bones menu.

When the armature is done it should be fitted to the object that you want to use it for. Think of the armature as the skeleton of the object. Also try to keep the center of the armature in the same place as the center of the mesh, the center is best placed between the feet close to the ground (this is the center of rotation of the entire mesh a.k.a. the pelvis).

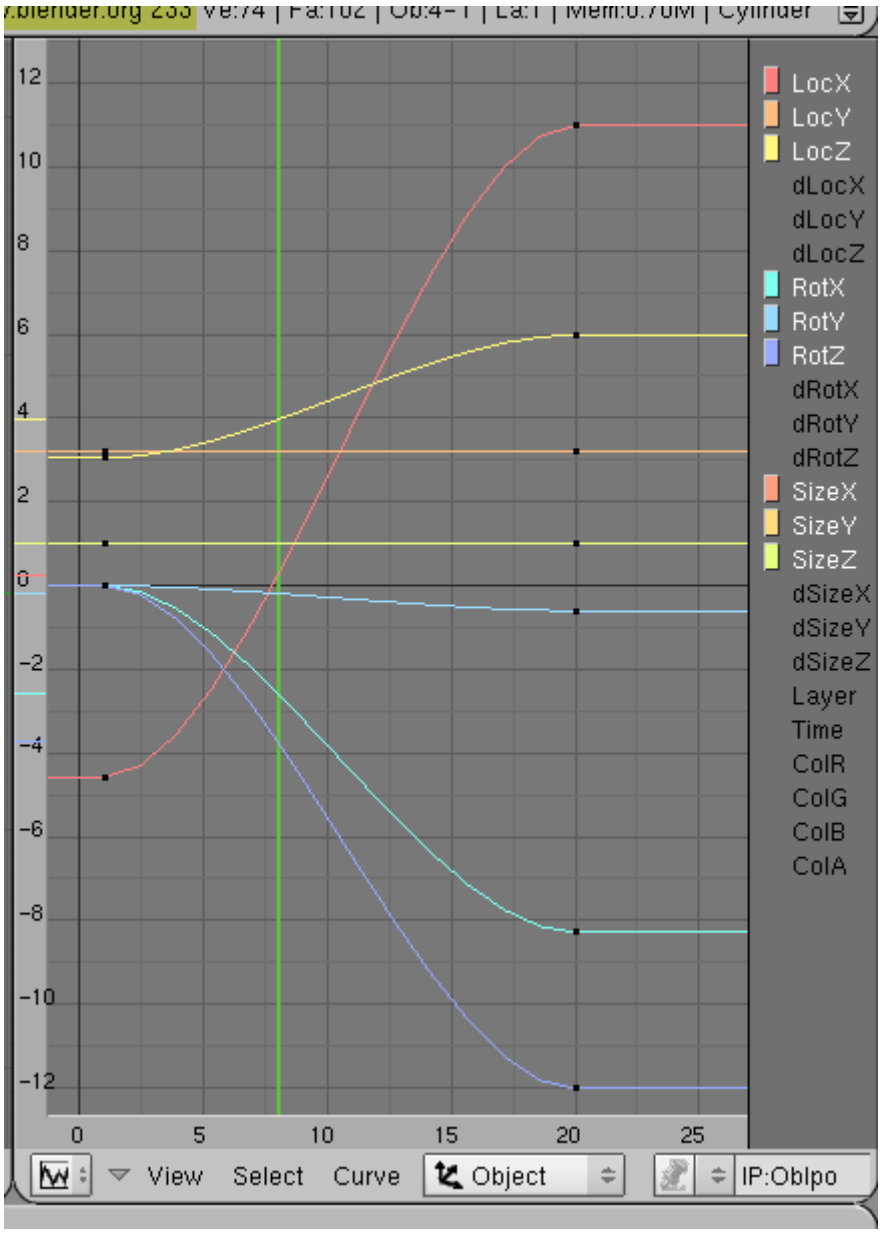


Figure 10.24: The "Ipo Curve Editor". Here we have saved key points for location, rotation, and scaling of a cylinder in frame 1 and frame 20 (the view has been changed using zoom and stretch functionalities). The vertical green line indicates the current frame shown in the modeling view.

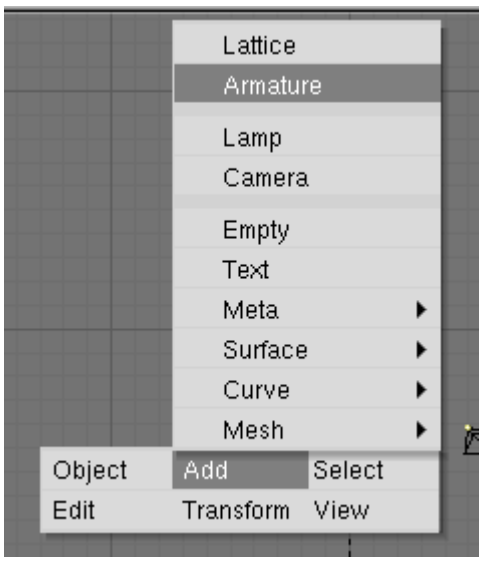


Figure 10.25: Creating an armature (NB. armatures can not be added in edit mode).

It is now time to attach the armature to the object mesh. To do this right you must specify which vertices to be affected by which bones. This is done by grouping the vertices. Here it is absolutely necessary that you are able to remember the names for the different bones in the armature. Select the object that you want to use the armature for and enter edit mode (`TAB`). In the edit menus under links and materials create a new vertex group (see figure 10.28). It is important that the vertex group is given the same name as that of the bone it should be attached to. Now, select the vertices you want to be influenced by that bone and press "Assign". If you are not happy with your selection later on you remove vertices from the group again by selecting them and pressing "Remove". To see which vertices that are currently in the group deselect all vertices in the modeling view (`AKEY`) and press the "Select" button. Now only the vertices in the current group are selected.

Try to have each vertex represented in one group only, although it is allowed to have them influenced by more than one. In any case it is most important that all vertices belong to some group; otherwise they will stay in their original position when the armature starts to transform the mesh.

When all vertices are assigned to a group we can assign the mesh to the armature. Make sure that you are not in edit mode, then select both armature and object mesh and press `CTRL+P` and select "Use Armature". Now the armature and the mesh have been connected.

To test the armature we must be in pose mode. We can enter pose mode when the armature is selected by pressing `CTRL+TAB`. Then the bones in the armature appear blue. Use the standard transformation keys to alter the bones of the armature, the mesh associated with it should follow the bones.

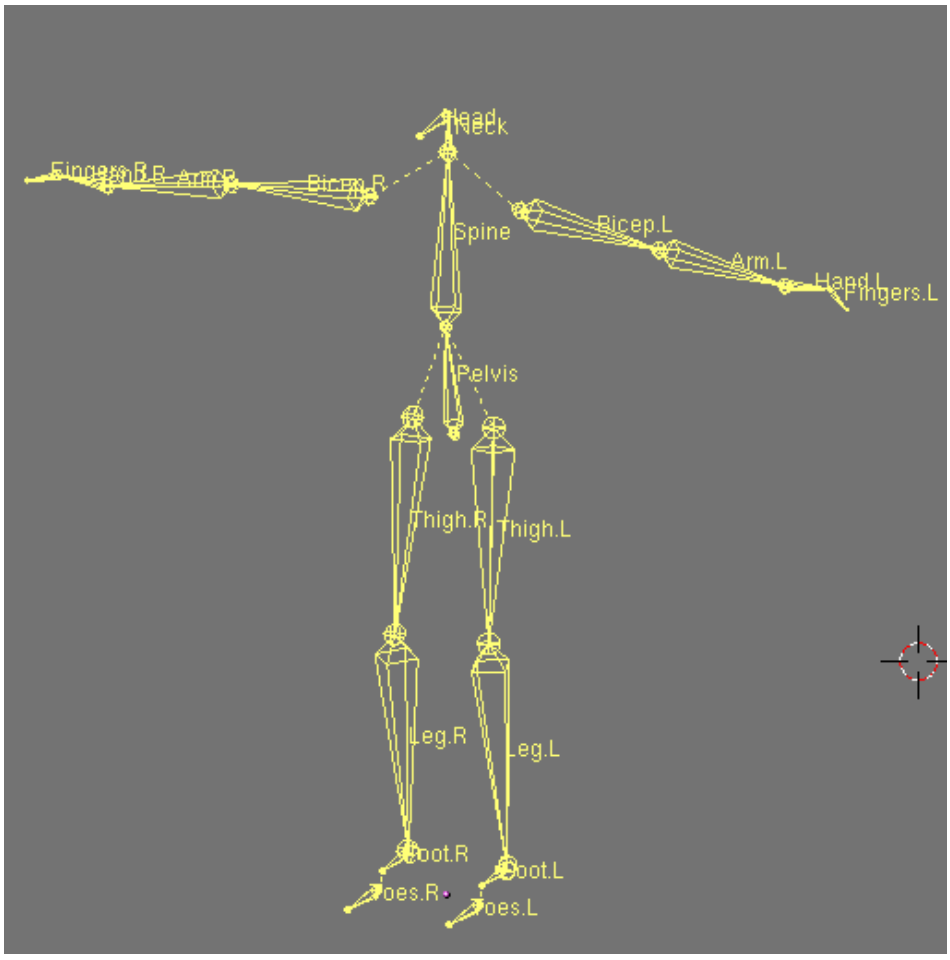


Figure 10.26: Armature example. This armature is used for the human character in the cave scene.

Figure 10.27: The "Armature Bones" menu shows the parenting hierarchy of the armature in figure 10.26. The pelvis bone is not a child of any other bones, hence, it is the top of the hierarchy.

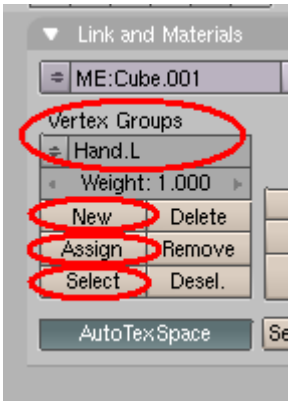


Figure 10.28: Creating a vertex group. The name of the group must be the same as the bone that should influence it. In this menu vertices are assigned, removed, selected, and deselected from the current vertex group as well.

Some common errors are: Vertices not changing position with the armature because they do not belong to any group, and vertices from strange places in the mesh that respond to transformations of the wrong bones because they are assigned to a wrong group. Such errors only reveal them selves by testing the armature in pose mode.

At all times, in pose mode, it is possible to clear transformations and return the mesh and armature to the their starting positions by pressing `ALT+R`, `ALT+S`, and `ALT+G` to clear rotation, scaling, and translation respectively.

As with animation described earlier in this section, we can save key frames for the armature. The only difference is that we are now saving actions, so instead of the "NLA Editor" we use the "Action Editor", see figure 10.29.

In the action editor we see the same yellow boxes representing key frames, there is a key frame for each bone in the armature though (the armature must be selected to show any key frames in the action editor).

In games we often make use of animation cycles that can be used in sequences. Cycles are done most easily by copying the key frames from frame one to the last frame in the cycle and then fill the space in-between with other key frames. In this way we are sure to create a cycle. It is possible to have many different cycles in blender. To begin a new cycle select "Add New" in the lower menu bar of the action editor view (see figure 10.30).

To view animations turn to the render menus (the small landscape icon in the icon selection bar, see figure 10.20 in section 10.3). Instead of the "Render" menu use the "Anim" menu. Set the number of frames to the same number of frames used in the cycle and press "ANIM". Blender now renders all the frames needed to play the animation. When all frames have been rendered press "PLAY" to view the animation. The animation will appear in a popup window and replay until the window is closed. Figure 10.31

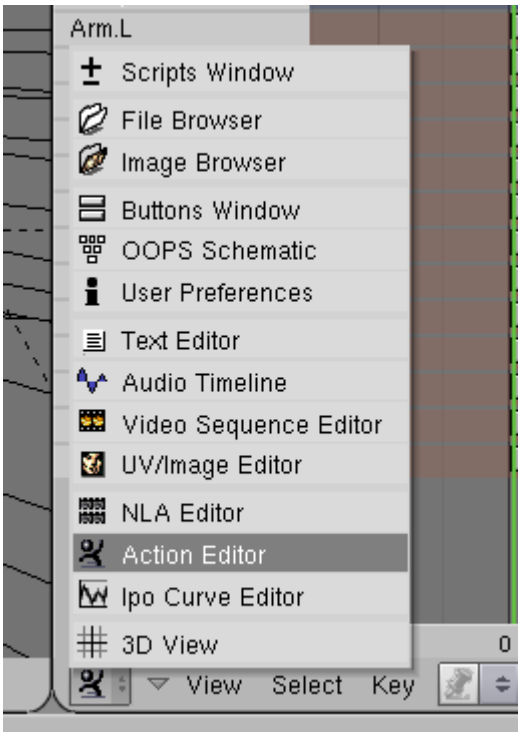


Figure 10.29: Selecting the "Action Editor" to view key actions instead of key transformations.

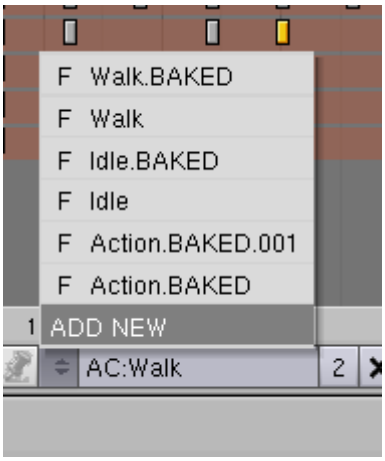


Figure 10.30: Multiple cycles can be created. To begin a new cycle select "Add New" in the lower menu bar of the action editor view.

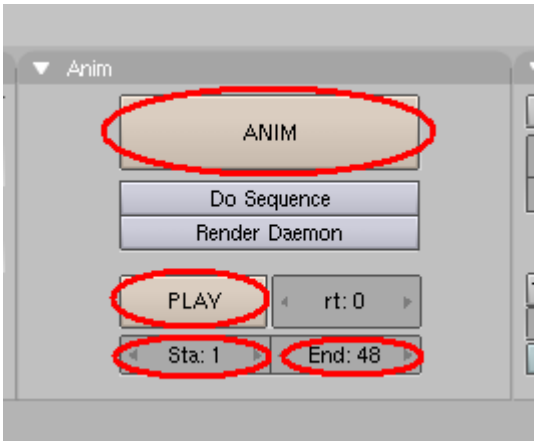


Figure 10.31: The "Anim" menu.

demonstrates the "Anim" menu.

The next section describes the options we have when we want to export our blender models and import them to our own renderer.

10.5 Export Scripts and Import Libraries

All data in Blender can be accessed via a Blender Python interface. Python[1] is an interpreted object-oriented programming language suitable for scripting among other things. As any other tool used for this project (except for MS Windows) Python is freeware.

To render the scenes created in Blender with our own render engine, we must export them to a file format that can subsequently be imported in our own application. The simplest way is to export all data to a file defining a C++ function, which initializes all the data directly to our renderer at compile time. The export script we wrote for this purpose is simply called `export.py`, source code is available on the attached CD-ROM, see appendix A.

In order to run such an export script in Blender first split the modeling view (see figure 10.3, section 10.1). Press `SHIFT+F11` over one of the new windows to switch to text editor mode (or choose "Text Editor" in the window type menu, see eg. figure 10.29), then press `ALT+SHIFT+FKEY` while the mouse is over the text editor and choose "Open..." in the popup menu (see figure 10.32). Browse to find `export.py` and open it in the text editor.

The first line of the export script is `FILENAME = "path"`, where `path` is the full path of some .cpp file name. Set the path and file name as you see fit. The chosen file will be the resulting export. Now, simple press `ALT+PKEY`

[1] URL: `http://www.python.org/`

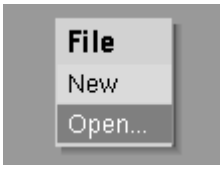


Figure 10.32: Opening a script file.

while the mouse is over the text editor. Some info will be printed to the console window Blender always keeps open beside the main window. The .cpp file given as `path` will then contain a .cpp file initializing a world for our rendering application.

The code export works well for small scenes and it's a good way to get started. However, when the scenes increase in size, the compile time of the exported code escalates intolerably. It is there necessary to use an intermediary file format. Blender supports export to different formats (under file→export), but we have chosen to use an XML (eXtensible Markup Language) based file format for 3D models called X3D [143, 144]. This eased the work we had to do with respect to import, since there exist open source tools for parsing of XML, and besides an X3D importer written by Bent Dalgaard Larsen was distributed in the DTU course "Computer Graphics" (02501). This importer is called *BMesh* and it makes use of the open source `C` library for XML parsing called *Expat*, written by James Clark[2].

A script for exports from Blender to X3D also exists, it is written by Adrian Cheater and the script file is called `x3d_export.py`[3]. The script is, however, at the development stage (June 12, 2004 - version 0.16). Nevertheless it presented a nice framework, which we have modified and made more robust for this project. To export a Blender scene or part of a scene with `x3d_export.py` follow the same steps as above until the script has been opened. No path need to be set in this export script. Instead when `ALT+PKEY` is pressed the popup menu shown in figure 10.33 will appear.

After choosing whether to export the entire scene or only selected objects, the Python script will query the file name of the export. Beware that this export script only works properly if you are using Blender 2.32 or newer, and if you have Python 2.3 or newer installed. It should also be noted that because of the early development stage of the script errors may easily occur and limitations to the export exist.

The import called BMesh is also a frame work at an early development stage. Therefore the import has many deficiencies. On the other hand the code is 'friendly' and allows expansions without causing too many troubles. For this reason we have been able to connect the BMesh import to our render

[2]URL: `http://www.libexpat.org/`
[3]URL: `http://www.bitbucket.ca/~acheater/blender/`

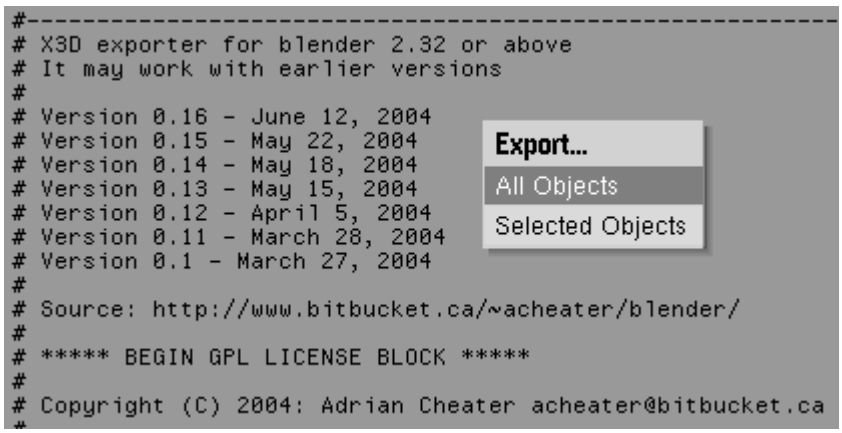


Figure 10.33: Exporting Blender models to X3D.

engine. Still there is room for many optimizations in the export/import part of this project.

One example of an export/import shortcoming is that the import support textures, while the export does not provide the necessary texture images. Another example is that normals are exported, but not imported by the BMesh. Instead they are re-calculated in some way that does not always generate better results.

Though we could have made many other choices with respect to export and import, we chose those options that gave us as much freedom as possible with respect to changes in the implementation. In return we had to cope with some of the limitations that naturally follows with programs at an early development stage.

Here ends out discussion on export scripts and import libraries. The next section concludes this chapter by mentioning some of the Blender features that have otherwise not been described in this chapter.

10.6 Additional Blender Features

The functionalities that have been described in this chapter constitute only a subset of Blender's capabilities. We have mainly described the functionalities that have been used to create test scenes for this project, and scenes that we had hoped to use as test scenes.

Blender is continuously developing, and as far as we are concerned only for the better. This chapter builds on version 2.33, when we started the project the newest version was 2.32 and already version 2.34 is available at `blender.org`. Fortunately functionalities are fairly compatible between the versions and much is done to describe the new features that newer versions present. Manuals and tutorials for older versions are also still available for

download, and since Blender is free of charge many amateurs write tutorials which they make available on the net.

The most extensive extra feature for Blender not mentioned until now is the build in game engine. It is actually possible to create your own 3D game using Blender only. The game engine provides a physical engine and a number of function calls to create a game from a scene. A good description of how to create games in Blender is found in [113]. Of course information about this can be found on the internet too.

Since all data are available through the Python interface to Blender, fairly complex games can be created through the combination of Blender and Python scripting. The Python interface also gives the game engine an extra dimension with respect to export of features to other applications. Python has access to the main loop of the game engine and can, therefore, access game parameters such as transformations and physical forces on the fly. These can be recorded as a kind of physics dependant key frames, in this way the Blender game engine can be used to create interactive animation sequences very quickly. Afterwards they can be exported and tested on an external renderer such as the one written for this project.

All in all we think that the combination of Blender and Python is a powerful tool, which is quite useful for modeling and animation of test scenes for external renderers. Blender is another step towards a game development platform which is free of charge. That ends the part of this project that concerns modeling of contents for 3D rendering.

Part III

Ideas, Results, and Experiments

The preceding chapters have presented little new except for the idea of an array based math engine (sec. 2.1), and few experiments apart from the expanded track ball (sec. 9.3) and the model-to-code export script (sec. 10.5). Yet we find that the theory given in part I is indispensable when we want to describe our own ideas, how they came to us, and why we discarded them or held on to them. The more practically oriented part II is a necessity for us in order to show the possibilities of a free of charge game development platform, and part II is important in order to supply our report with anything more interesting than a static standard scene (such as a still image of a Cornell box).

In other words the two previous parts provide the foundation from where ideas emerge and experiments come into mind. Building on this, we will now focus on the ideas that have appeared during the project. The main idea has become the Direct Radiance Mapping (DRM) method which we have referred to from time to time throughout the report and which will be describe in detail. Other ideas will also be presented and discussed, some have been implemented too. Direct Radiance Mapping was, however, the idea that we chose for further development towards the end of the project.

Chapter 11 will present the ideas that we have come up with during the project. We will discuss possibilities and drawbacks for each of them. This will often lead to the reason why we chose not to bring the idea any further in this project. A few ideas have been implemented on an experimental stage, in these cases the results will be discussed as well.

Going through the project idea by idea eventually leads to the concept that we have chosen to stick by: A method we have named Direct Radiance Mapping (DRM). The method builds upon basal radiance calculations and contains few restrictions and preconditions. DRM is based on the newest GPU technology, and the method will draw advantage when GPUs grow faster and more efficient. This means that the method may stand even stronger in the future. First of all we use DRM to calculate indirect illumination, but there are other applicabilities that will be tried out as well, an example is subsurface scattering. Chapter 12 will describe the method in detail.

DRM is mainly a method for simulation of indirect illumination, and as such it can easily be combined with other methods addressing different parts of the rendering equation (see chap. 6). Chapter 13 will mention all the additional methods that we have implemented for combination with DRM. We will also mention implemented methods that are not for real-time rendering. Those have mainly been implemented for comparison. The radiosity implementation was originally implemented as an exercise in the DTU course "Computer Graphics" (02561), therefore it does not use the same render engine as the rest of the rendering methods in our application. Still a radiosity implementation is very useful for comparison, and it has therefore been brought along and given a menu of its own in our viewer.

Apart from DRM we have put an effort in construction of a graphical user interface (GUI) called JR Viewer, which shows how the global illumination effects that we are creating works in real-time compared to standard global illumination rendering (based on chap. 4). We have implemented a ray tracer as well as a radiosity renderer capable of rendering some predefined scenes (Cornell box containing a few objects). These scenes can also be rendered using our different real-time global illumination effects. If our method was to be used in a more extensive real-time 3D environment, we would need to be able to render more complex scenes than a Cornell box. As a part of the project we have taught ourselves how to model in Blender (see chap. 10) and we have created a "cave scene" for testing our method. This scene is also available in the GUI. Chapter 14 will describe the capabilities of the GUI and what we wish to show with our different test scenes. Note that the GUI is a MS Windows application and, hence, does not run under other operating systems (Linux, Mac, etc.). This was a choice we made in order to have some extra windows features at our disposal (such as drop down menus).

Even though we have chosen not to emphasize on the process of software development, rather on the application of mathematical tools to solve a complex problem, implementation is inevitably a major part of the work that must be done if we want to experiment or if we want to verify the ideas that we put forth.

Chapter 15 will present all the different program parts and discuss their structure. All parts are bound together in our graphical user interface, which therefore can be seen as the main part. From this part branches like ray tracing, radiosity and rasterization emerges. These are, on the other hand, all just different ways of rendering the same scene, therefore we will see that most of the rendering methods make use of the same render engine, which is the core of the application.

Chapter 11

Ideas

The best way to have a good idea is to have lots of ideas.

Randall Jarrell: *Pictures from an institution* (pt. 1, ch. 4)

The starting point of this project was to create a decent global illumination renderer and a simple real-time renderer. The implementation of those is described in chapter 15. Then the plan was to move them towards each other little by little and see if we could make them meet midway.

Our choice of global illumination algorithm was ray tracing expanded by photon mapping. This seemed to be the most general global illumination method having acceptable performance with respect to processing time, the choice was, of course, also inspired by those who had previously been able to simulate photon mapping in real-time or at interactive rates, see section 6.7.

The first step was then to find optimization strategies for photon mapping which could lead it in a direction towards rasterization methods. It springs to mind when dealing with photon mapping that most of the time spend in the photon mapping algorithm is spend for ray tracing. Ray tracing is well explored with respect to efficiency and optimization schemes, nevertheless we had a few ideas for ray tracing optimizations some of which were based on graphics hardware. Some of the ideas for ray tracing optimizations developed into possible solutions for the entire global illumination problem. Others (those based on graphics hardware) brought ray tracing closer to rasterization. The different ideas are introduced in the following order:

1. Angular visibility between axis aligned bounding boxes (AABBs) for fewer intersection tests in ray tracing.
2. A topological network for radiance transfer between objects.
3. Displacement mapping for fast ray/object intersection.
4. A multi-agent approach to global illumination, where each object is an autonomous agent controlling its own shade.
5. An 'atmosphere' to limit the influence of each object in global illumination.
6. A line-of-sight algorithm for fewer intersection tests in ray tracing.
7. Gouraud interpolation of the first intersection point. A rasterization approach to the first level of the ray tracing algorithm.
8. Single pixel images each representing a ray. An attempt to let rasterization do full recursive ray tracing.

Keep in mind that these sections merely introduce the ideas. Some of them have not been implemented, but are presented here to give an insight in the process it was to find the final idea, and they give a feel of the choices we had to make before starting to pursue a specific idea. Our final idea of direct radiance mapping emerged when we started at the other end of the

rope, that is, when we tried to move rasterization closer to realistic image synthesis. Direct radiance mapping is described in the next chapter, but it is founded in the ideas presented in this chapter.

11.1 Angular Visibility Between AABBs

The first idea for ray tracing optimization is to set up an axis aligned bounding box (AABB) for each object in the scene. When light is reflected from an object we can use the angular visibility between the AABB of the intersected object and the AABB of each of the remaining objects in the scene. It can be tested whether the direction of the reflected ray lies within the angular visibility between two objects before the ray is tested for intersection with the receiving object.

Recall from section 4.2 that when a ray is traced from a position P in the direction $\boldsymbol{\omega}$ we must in traditional brute force ray tracing test *each* primitive object in the scene for intersections and pick the intersection closest to P.

In scenes containing many primitives spatial data structures are often necessary in order to occlude as large parts of the scene as possible before choosing which objects to test for intersections.

In the following we describe a very simple way to limit the number of intersections tests. Let $\boldsymbol{C}_{\text{ref}}$ be the center of the reflecting object from where a ray could be traced. Let $\boldsymbol{C}_{\text{rec}}$ be the center of a receiving object which a ray could be tested for intersection with. Then cosine of the angle (θ) between the direction of the ray $\boldsymbol{\omega}$ and the direction pointing from the center of the reflecting object to the center of the receiving object is given as:

$$\cos\theta = \boldsymbol{\omega} \cdot \frac{\boldsymbol{C}_{\text{rec}} - \boldsymbol{C}_{\text{ref}}}{\|\boldsymbol{C}_{\text{rec}} - \boldsymbol{C}_{\text{ref}}\|}$$

Here $\cos\theta \in [-1, 1]$ will give a measure stating how close the direction of the ray is to the vector direction 'connecting' the two objects. Suppose a threshold stating $\cos\theta_{\max}$ has been precalculated, then $\cos\theta > \cos\theta_{\max}$ returns whether it is worth testing for intersections or not. The total number of operations needed for this test is a dot product and a comparison test ($>$).

Let $\boldsymbol{B}_{\text{ref},i}$, $i = 0, \ldots, 7$ be the eight corners of the AABB for the reflecting object, see figure 11.1, and let correspondingly $\boldsymbol{B}_{\text{rec},i}$, $i = 0, \ldots, 7$ denote the AABB of the receiving object, then $\cos\theta_{\max}$ is found as the minimum of the cosines of the angles between $\overline{\boldsymbol{C}_{\text{ref}}\boldsymbol{C}_{\text{rec}}}$ and $\overline{\boldsymbol{B}_{\text{ref},i}\boldsymbol{B}_{\text{rec},i}}$. The concept is illustrated in figure 11.2.

Suppose we precalculate the direction of the connecting line $\boldsymbol{\omega}_{\boldsymbol{C}} = (\boldsymbol{C}_{\text{rec}} - \boldsymbol{C}_{\text{ref}})/\|\boldsymbol{C}_{\text{rec}} - \boldsymbol{C}_{\text{ref}}\|$ and the threshold value between each combination of two objects. Now, whenever a ray is reflected off an object in a direction $\boldsymbol{\omega}$, the simple test:

$$\boldsymbol{\omega} \cdot \boldsymbol{\omega}_{\boldsymbol{C}} > \cos\theta_{\max}$$

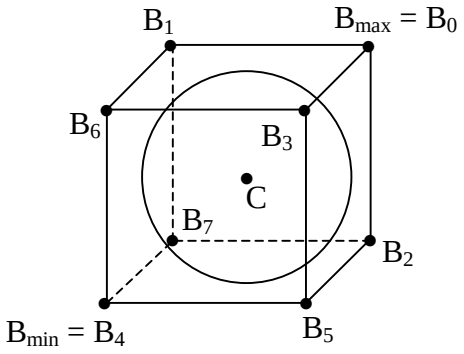


Figure 11.1: Center position and axis aligned bounding box (AABB) of an object.

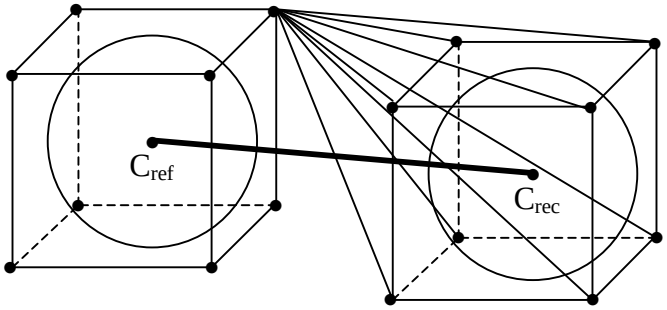


Figure 11.2: Precalculation of threshold values is done using the corners of each objects bounding box. Here the bold line represents $\overline{\boldsymbol{C}_{\mathrm{ref}}\boldsymbol{C}_{\mathrm{rec}}}$ and the first set of lines $\overline{\boldsymbol{B}_{\mathrm{ref},0}\boldsymbol{B}_{\mathrm{rec},i}}$ for threshold value determination are drawn. A similar set of lines for each of the remaining points in the AABB of the reflecting object should also be tested as threshold value.

can quickly reject an object before testing for intersection. We call $\cos\theta_{\max}$ the angular visibility between two objects. Because of the simplicity the idea is quite efficient for a limited amount of objects. A spatial data structure such as the BSP tree will probably be more efficient when the number of objects increase significantly. The reason is that the BSP tree optimally only will have to test the ray against $\log N$ planes, where N is the number of objects in the scene, while the angular visibility is tested for each object. On the other hand the BSP tree is difficult to change when first set, while the angular visibility between two objects easily can be modified in real-time.

Fortunately nothing prevents us from combining angular visibility with other spatial data structures. An interesting approach in a real-time environment consisting of complex rigid objects, could be to have an angular visibility 'network' between the objects and a BSP tree for the primitives in each object. This would have the effect that only some weights in the angular visibility network would need recalculation when objects are moving, while the BSP tree would only need to be moved not entirely recalculated,

since the objects were assumed to be rigid.

This was the first idea. The angular visibility network has been implemented, unfortunately we have not had the time to combine it with a BSP tree. The results of an angular visibility network are good in the simple Cornell box scene which is otherwise often too simple to benefit from spatial data structures. Improvements due to angular visibility tests alone are roughly 26% when ray tracing a Cornell box with a chrome (or mirror) sphere and a glass sphere (12 triangles and 2 spheres) including specular recursions, hard shadows, and phong shading. The resulting image is shown in figure 11.3.

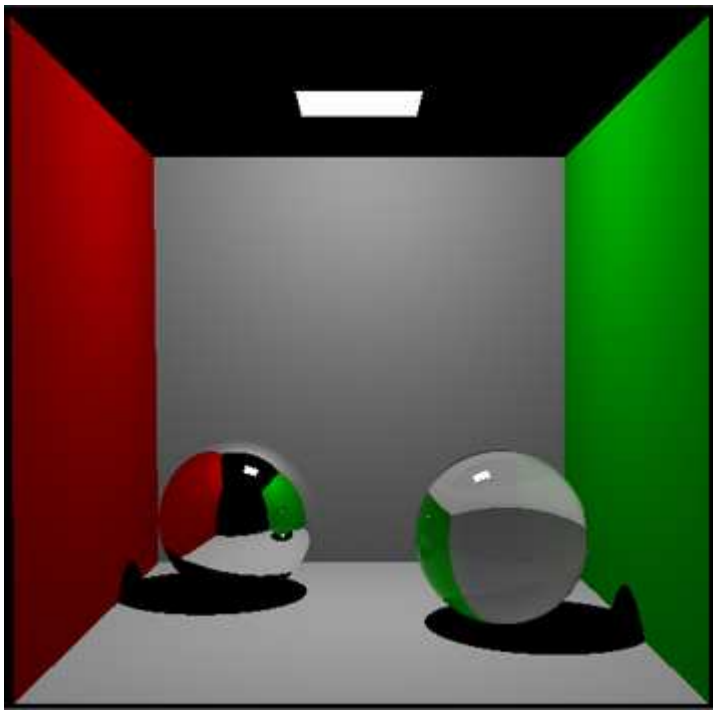


Figure 11.3: A Cornell box with a chrome (or mirror) sphere and a glass sphere (12 triangles and 2 spheres). The scene was used as a test of the efficiency of the angular visibility network. The specific render includes specular recursions, hard shadows, and phong shading

There is room for further improvements on the method. If an AABB contains a triangle only, it would, in fact, be much more efficient to find the angular visibility using the corners of the triangle instead of the corners of the bounding box.

The idea of an *angular visibility network* is interesting since the angular visibility could be a measure of the amount of energy which should be transferred between two objects. This idea is explored a little further in the next section.

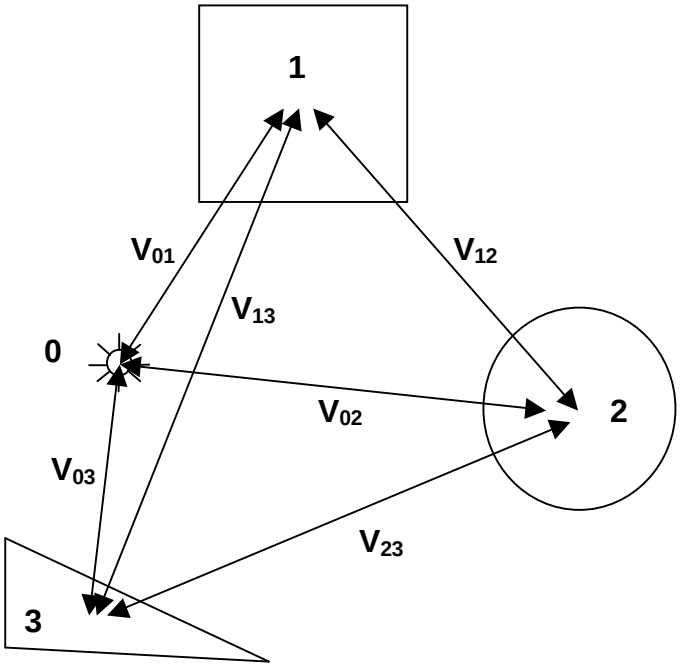


Figure 11.4: An example of a simple network. Each object (including the light source) is a node in the network. The weights of the edges between the objects are given as visibility. Note that $V_{ij} = V_{ji}$.

11.2 Topological Network

The second idea follows in the footsteps of the first one, since it might be possible to determine the radiance transfer or the flux area density transfer between two objects by use of angular visibility (among other things). The idea is to describe radiance as flux in a fully connected topological network with each object as a node.

Instead of tracing rays through the scene the idea is to distribute energy. Energy could be distributed for example according to angular visibility. Another option is the hemicube form factor ΔF_{ij} method (see sec. 4.1), which can determine the fraction of energy leaving patch i that will reach patch j. The form factor is on the other hand quite expensive to calculate.

Each edge in the network will represent energy exchange between two objects. The light sources are part of the network. Light is emitted from the light sources by propagation through the network. Suppose we decide a certain amount of photons to be emitted from each light source, then the visibility between the objects should determine how to distribute the photons to the different objects in the network. We could refer to the visibility V_{ij} between two objects as the weight of the edge between the objects in the network, see figure 11.4.

When an object receives a *signal* (a number of photons) from another

object the photons are distributed over the area of the receiving object which is visible from the reflecting (or emitting) object. Environment maps (see sec. 6.3) could be used to determine these areas.

After emission and propagation in the network the temporary energy of the object decreases, while the temporary energy of the objects that have received photons increases. Now, suppose we set some threshold for each object. When the temporary energy passes above the threshold, the object is marked for reflection of energy. In the next step of the algorithm all objects marked for reflection will propagate energy back into the network, while still storing the photons they received. Now, the recursion can continue until an equilibrium state is reached. At this point the final image has been reached.

Inspiration for this method was found in the idea of wave expansion neural networks [67, 68, 71], for those familiar with the research area of neural networks the analogy quickly becomes apparent.

Though the method seems close to progressive refinement as described in radiosity, the difference is that photons are distributed across the surface of an object. This means that we can let reflected energy originate in specific photons incident on an object and thereby photons propagated in the network can have a specific direction and origin. This means that we can model arbitrary BRDFs if necessary.

The three main difficulties in this method are visibility calculations, reflection (or refraction) of light from an object to itself, and reconfiguration of the network when things change dynamically.

To further expand the networking concept each object could itself contain a network where sample locations across the surface would be nodes. In this way the object would be able to interact with itself. As suggested in [61] in a different context, Turk's repulsion algorithm [127] could be used to sample points across the object surface.

Most of the concepts described in this section are merely suggestions, nevertheless, the network analogy was also the origin of our direct radiance mapping idea (see chap. 12). If we do not choose to have an internal network for each object, we could instead distribute the incident photons over the visible part of the bounding box. The following section presents an idea for precalculation of the intersection test that must be calculated when the photons are traced from the bounding box to the exact intersection with the receiving object.

11.3 Displacement Mapping

Complex objects are often difficult to ray trace because they consist of an immense number of triangles. Many methods exist to handle this problem, most of them are based on spatial data structures. Spatial data structures spend time finding the right triangle for the intersection calculation. As an alter-

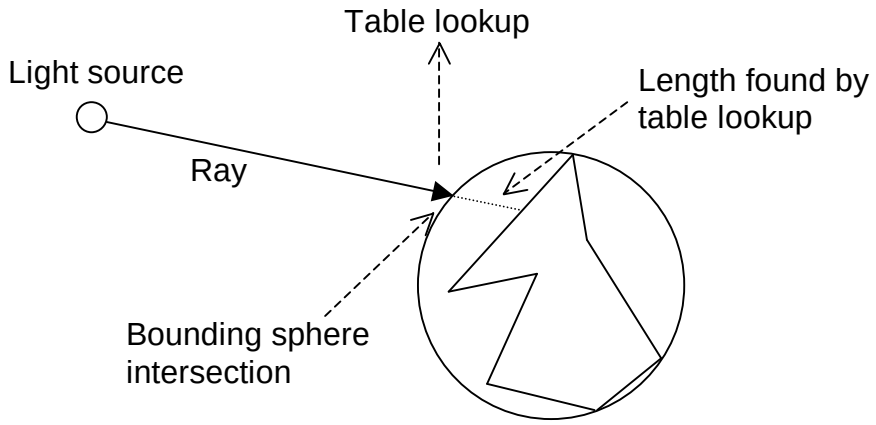


Figure 11.5: The displacement mapping concept.

native to spatial data structures we reinvented the concept of displacement mapping. Instead of calculating intersection with the complex object itself our idea was to calculate intersection with a bounding volume and then use a many dimensional texture to return the exact point of intersection. This would also be a good idea in continuation of the topological network idea, since rays (or photons) can be propagated in the network and distributed across the part of the bounding volume visible between the reflecting object and the incident object, and then traced to the correct intersection using the displacement map.

Looking into the ray tracing process it is noteworthy that any ray intersecting the bounding sphere surrounding a triangle mesh at a certain point in a certain direction, will intersect the exact same point on the same triangle inside the mesh, as long as the triangles do not deform, that is, if the object is rigid. This fact gives strong indications that pre-calculations are an option.

To pre-calculate the information stating the point in a mesh that a ray will intersect when it intersects the bounding sphere, we must span a configuration space across the entire bounding sphere the surface of which is two-dimensional in (u, v)-coordinates. Then two extra dimensions must be added to account for an incident direction (θ, ϕ) on the hemisphere giving a total of four dimensions. We could describe such a configuration space as a four-dimensional texture. The texture would hold the length along the ray that we must travel from the bounding sphere before the object is intersected, see figure 11.5. In the following we will describe how such a four-dimensional texture could be constructed using an array-based terminology.

Let $\boldsymbol{d} = (\theta, \phi) \in [0, \pi] \times [0, 2\pi]$ denote the direction of the incident ray, and let $\boldsymbol{P} = (u, v) \in \mathcal{S}$, where $\mathcal{S}$ is the (u, v)-space of the sphere surface, denote the point on the sphere that was intersected. Then points should be sampled uniformly on the sphere and over the incoming directions in order

to discretize the axes. The structural operation behind a Cartesian product, that is, the array theoretic transformer called OUTER (see definition 7), will then span the entire configuration space of the axes.

Definition 7 (OUTER) *Let A_1 and A_2 be arrays of data and let f be a binary first order function, then*

$$A_1 \text{ OUTER } f \ A_2 \tag{11.1}$$

is defined as the function f applied to all pairs defined by the Cartesian product of the arrays A_1 and A_2.

The discretization of each floating point axis will divide it into intervals. If we sample at midpoint of each interval, the configuration space will hold the midpoint of each cell in the four-dimensional texture. For each cell midpoint we find the distance to the point of intersection by ray tracing and assign to the texture value of that configuration.

We can now use a four-dimensional array (or texture) as a map $\mathcal{M}$ from direction $\boldsymbol{d}$ of the incident ray and position $\boldsymbol{P}$ of intersection on the bounding sphere to distance t that must be traversed along the ray before the intersection is found:

$$\mathcal{M} : [0, \pi] \times [0, 2\pi] \times \mathcal{S} \to \mathbb{R}$$

where

$$\mathcal{M} \mapsto \text{OUTER } \texttt{intersect}$$

which means that a sampling of the continuous space $[0, \pi] \times [0, 2\pi] \times \mathcal{S}$ to discrete axes and use of the structural operation behind an outer product with an intersection algorithm, will span a configuration space that leaves

$$t = \mathcal{M}(\mathbf{d}, \mathbf{P})$$

being a simple table look-up. Having (of course) complexity $\mathbf{O}(1)$.

After re-inventing the concept we found that it was already well explored and known as view-dependent displacement mapping, see [131]. The absolute largest drawback is the size of the array. On top of this the displacement map of a particular object would basically need recalculation each time transformations other than translations or rotations are carried out. Translation can be done without recalculation since the array concerns the bounding box and the object, not the surroundings as such. This is also true for rotation if the bounding volume is a sphere. (Uniform scaling may also be possible since the lengths in the map can scale accordingly.)

The idea of a table look-up for ray tracing carries on to some of our subsequent ideas. In the following section we turn to a more general concept, which could be applied in combination with many other ideas.

11.4 Multi-Agent Global Illumination

With a background in the area of artificial intelligence, we quickly came up with the idea of a multi-agent approach to global illumination. The idea is to regard each object, be that in a topological network or not, as an agent which must itself take care of its current shade at all positions across its surface. This distributed approach is interesting because of the parallelism that is inherent in global illumination.

Each object, which in this context is an autonomous agent, should run in its own program thread. Sensing is a well known conception in multi-agent systems, a good reference on multi-agent systems in general is [136]. Whenever the object sense the surrounding it will update visibility of other objects, for example by use of an environment map. The object will at any time be able to receive new packets of photons. They will, however, not be processed until the agent thread is idle. The agent will also try to pass on energy to those objects visible in its environment map. Each packet of photons will then be a message containing the information necessary for the receiving agent to distribute the photons across its surface and again pass on photons to visible objects.

The advantages of a distributed approach is that each object, in principle, can use different rendering algorithms adapted specifically for their nature. A very complex object may get away with a simpler shading method and thereby not decrease the frame rate. An important aspect of multi-agent systems for dynamic environments are anytime algorithms, meaning that the object must always have the best answer available at any time. In this way the rendering can be made more robust. A single object may take a long time to render but this will not stop the entire rendering process, that particular object will just update its shading a little too late (maybe every fifth frame).

From our point of view the idea is indeed interesting, it seems to bring up many possibilities that have otherwise been unthought of. The problem is whether there is an overhead in the administration of a pseudo parallel thread for each object and the handling of the messages send between the agents. The idea of individually controlled objects is, however, appealing to us. The next section presents a simple idea which could be useful, for example in the context of a multi-agent system.

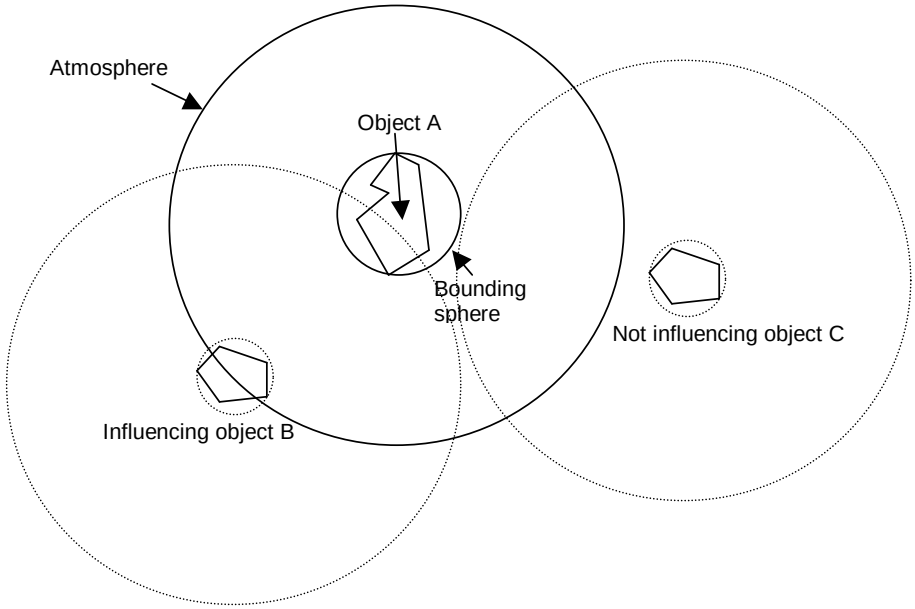


Figure 11.6: Object A is influenced by light reflected from object B, since this object interferes with its atmosphere. Light from object C has no influence, since this is outside the atmosphere of object A.

11.5 Object Atmosphere

Normally changes in indirect illumination caused by other objects becomes insignificant if the objects reach a certain distance. A very simple idea, which limits the amount of work that must be done, is to have a sphere of influence, or an 'atmosphere', around each object in order to limit the influential area of objects in calculations of indirect illumination, see figure 11.6. The same concept is known from light sources, which often are supplied with a sphere of influence. An object atmosphere could be useful eg. for the multi-agent approach.

This idea has not had much influence on our current implementation. Another idea is to use a line of sight algorithm for ray tracing optimization, this is the subject of the following section.

11.6 Line-of-Sight Algorithm

Suppose your scene is divided into a regular grid (often only two-dimensional). Then the purpose of a line-of-sight algorithm is to find the exact list of grid cells that a ray following the line of sight will pass through, see figure 11.7. If we use the line-of-sight algorithm for ray/object intersection all empty grid cells will be thrown away. When a grid is reached containing objects the algorithm will check (if the algorithm uses a two-dimensional grid) whether

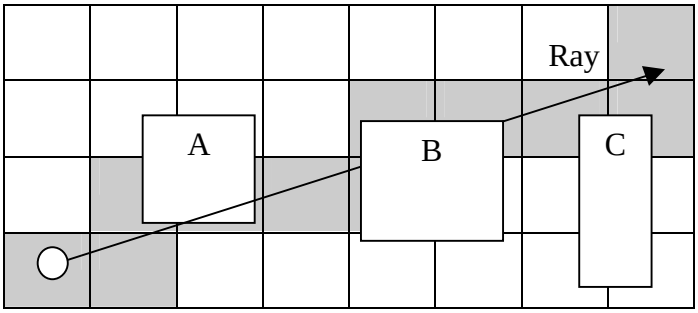


Figure 11.7: The concept of a line-of-sight algorithm (a). This is the top view. The list of grid cells is marked in gray. The ray has possible intersection with both A, B, and C in this case. The algorithm will consider one not empty grid cell at the time.

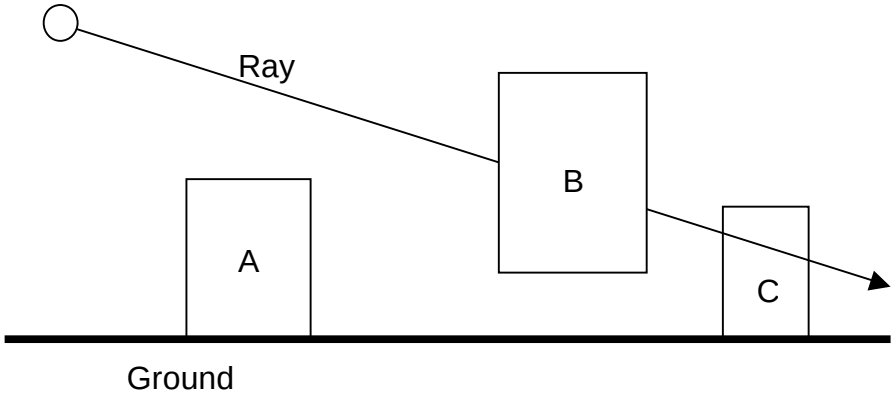


Figure 11.8: The concept of a line-of-sight algorithm (b). When looking closer at the side view (or at the height interval) the algorithm will find that object B is intersected by the ray first.

the object lies in the correct height interval or not, see figure 11.8.

The idea had its origin in an efficient line-of-sight algorithm for 3D landscapes described in [129]. This paper only describes how to find the list of grid cells. After trying out an implementation of a quad tree for regular grids, we chose to discard the idea since it was too expensive to alter the grid on the fly. With a better and more efficient implementation of a regular grid the idea might still be interesting.

While the idea described in this section still try to optimize the ray tracing procedure itself, we shall in the following sections try to move ray tracing closer to rasterization in different ways. This more closely follows up on the ideas stated previously in this chapter.

11.7 First Intersection in Hardware

The first idea for moving ray tracing closer to rasterization is to do the first level of ray tracing in hardware. The idea is quite obvious because rasterization can obtain images similar to ray traced images as long as specular recursions are not included. In other words we may as well let the rasterizer solve the part of the rendering equation which simulates direct illumination, while ray tracing takes care of specular recursions.

The most important part of this idea is that we cannot just render the direct illumination as usual with the rasterizer. This would give us no improvements with respect to ray tracing, since we would still need to start the ray tracing from the eye. What we need is to find the first intersection points using the rasterizer. The procedure is to make a simple vertex shader that scales the position of each vertex (in world coordinates) to the interval $[0, 1]$. This can be accomplished using an AABB of the scene to be rendered.

First give all primitive objects a specific color id and draw the scene (see figure 11.9a), then draw the position of each fragment using Gouraud shading and the simple vertex shader mentioned above (see figure 11.9b). Those two images result in the first level of ray tracing (intersection point and id of the intersected triangle). From there on the ray tracing can carry on as usual. For further optimization the depth buffer could be used to specify the t value of the intersection point for each ray ($\boldsymbol{r} = \boldsymbol{o} + t\boldsymbol{d}$) that would eliminate the pass to make image b.

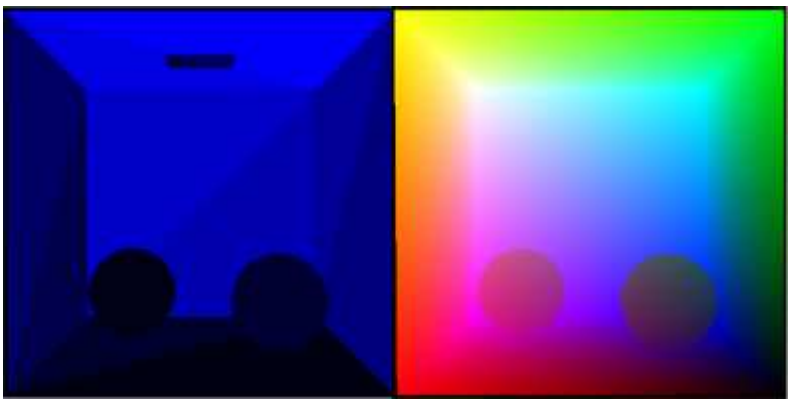


Figure 11.9: From left to right: (a) Color ids for each primitive in the scene. (b) Position of first intersection scaled to the interval $[0, 1]$.

Another improvement is that shadow volumes can replace shadow rays. The procedure is for each light source to render the shadow volumes to the stencil buffer as usual and then use the result to decide whether light from a particular light source should contribute to the shade of each ray representing a pixel.

Using these improvements and the simple angular visibility idea described

in section 11.1, traditional ray tracing is improved as follows: Rendering figure 11.10b takes 41% of the time we spend for the rendering of figure 11.10a. The slightly more complex scene of figure 11.11 takes approximately 26% of the reference time. The rendering times are given in table 11.1.

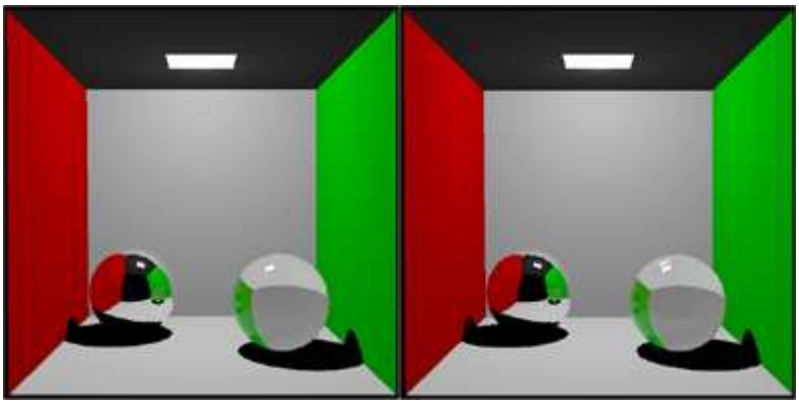


Figure 11.10: Renderings of the Cornell box with two spheres. From left to right: (a) Reference image using standard (one ray per pixel) ray tracing. (b) The same image rendered where the first intersection was found in hardware.

Method\Scene	**Two spheres**	**Orb on pedestal**
Traditional ray tracing	10.297 s	137.547 s
First intersection in HW	4.218 s	36.125 s

Table 11.1: Rendering times for the first intersection in hardware improvement compared to standard ray tracing. Rendering was performed on a 1GHz Pentium 3 machine with a GeForce4 MX440 SE.

As seen in figure 11.10, the inaccuracy of the color buffer (8 bits per color band) leads to some aliasing artifacts close to the lines and highlights of reflections and refractions. Using a fragment shader for the position image or a pbuffer with higher precision may eliminate such aliasing artifacts.

Opportunities for further improvements along the same line of thought are, for example, to do the BRDF calculations in a fragment shader, or to find the the first reflection and refraction vectors for the recursion to level two in hardware using a simple vertex shader. We did not spend time for these minor improvements since we hoped to find a more general approach, which would be able to run in real-time.

All the ideas described in this section are useful for the first level of ray tracing only. This means that highly specular scenes will not draw a significant advantage from the method. The improvement is also relative to how optimized the ray tracing processing is with respect to spatial data structures. A truly optimized ray tracer would benefit less from doing the first step in hardware. Even if the improvements are not significant the idea

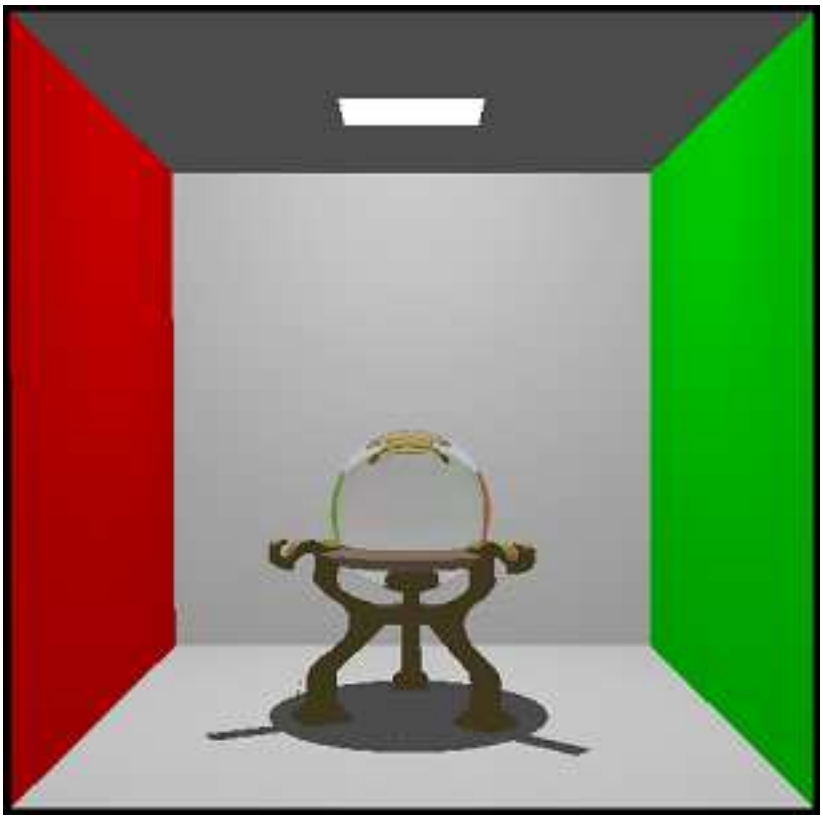


Figure 11.11: Rendering of the Cornell box with orb. A slightly more complex scene. Ambient light is included in this render, otherwise the pedestal under the orb would be entirely black (traditional ray tracing does not account for diffusely reflected indirect illumination).

of a first step of ray tracing fast enough for real-time is interesting and it will be shown in chapter 12 how this can be exploited. Before the idea of direct radiance mapping, our plan was to see if we could do the entire ray tracing process using rasterization methods. Our brief approach towards this end is described in the next section.

11.8 Single Pixel Images

An experiment we did was to trace the rays after the first level by movement of the camera and rendering of a size 1×1 image for each ray. The bottleneck is, of course, that each ray must travel all the way down the graphics pipeline, which takes approximately half a millisecond on a GeForce4 MX440 SE with a simple scene such as a Cornell box. This is hardly more efficient than a highly optimized spatial data structure, but on the other hand no data structure need reconstruction when objects start moving dynamically.

The idea would move the entire process of ray tracing to the GPU. However, it is not very elegant. In fact it utilizes the hardware in a cruel manner, since the entire scene must be clipped for each ray that we want to trace. If this approach was to be a success, the GPUs would have to start emphasizing on an ability to render a large amount low resolution images rather than an ability to process an increasing amount of fragments (high image resolutions) and an increasing amount of vertices (more triangles). Results for single image pixels are not significantly better on a GeForce 5950 and for more complex scenes the render time for each single pixel image increases significantly. This indicates that GPUs not likely will improve performance for many low resolution images. We therefore quickly realized that the idea had little future.

Many ideas have been mentioned in this chapter. Some more comprehensive than others. All of them originated in the movement from realistic image synthesis towards real-time rendering. None of them came close to real-time performance, at least none of those we had time to implement. In the following chapter we will in particular draw on the ideas of networking (sec. 11.2) and first intersection in hardware (sec. 11.7) when we start with a real-time renderer and move towards global illumination.

Chapter 12

Direct Radiance Mapping

Assess the advantages in taking advice, then structure your forces accordingly, to supplement extraordinary tactics. Forces are to be structured strategically, based on what is advantageous.

Sun Tzu (∼500 BC.): *The Art of War*

After a presentation of several ideas which were either half pursued or half discarded, we will present the method that we chose to work with in particular during the last part of our project. We have chosen to call the method Direct Radiance Mapping (DRM) and to our knowledge no-one else have tried it out before. Therefore we regard DRM to be an important part of our project.

Recall the expansions of the rasterization approach described in chapter 6. As shown in that chapter one of the most significant visual effects, which is present in global illumination, but difficult to model in real-time, is light reflected diffusely more than once. Direct radiance mapping is another method for simulating multiple diffuse reflections. Common for most methods trying to solve this problem is that some limitations are inflicted on the scene dynamics. The general idea in our approach is to apply as few limitations to the scene dynamics as possible, and then always solve the simple case before taking the next step towards full indirect illumination. This will especially be apparent in section 12.2.

In section 12.1 we outline the basic ideas and sources of inspiration behind direct radiance mapping. Next, in section 12.2, the method is described from a more practical point of view and the progress of the method to its current state is discussed. Abilities and limitations of direct radiance mapping will be discussed in section 12.3. In the final section of this chapter we will compare direct radiance mapping to the competing methods described in chapter 6.

12.1 The Concept

In this section we will concentrate on the conceptual idea of direct radiance mapping (DRM). What we describe here is an abstract formulation which shows the origin of DRM and its relation to the ideas mentioned in the previous chapter. Our implementation does not necessarily follow this abstract formulation, the next section will focus on the resulting method for implementation.

The basic idea is to take a picture of the scene from the point of view of the light source (as done in shadow mapping proposed by Lance Williams in [139]), let this picture represent direct illumination, then take a picture of the scene from the eye point, representing visible points in the scene. By going through each point visible in the eye point picture and calculating the light contribution from each point in the direct illumination picture we get the first bounce of indirect light. Figure 12.1 seeks to illustrate the process.

A more elaborate version of the concept, which tells more about where it came from and how it can open up further possibilities than a single bounce of indirect illumination, is explained in the following.

Considering the rendering equation as a problem of wave propagation

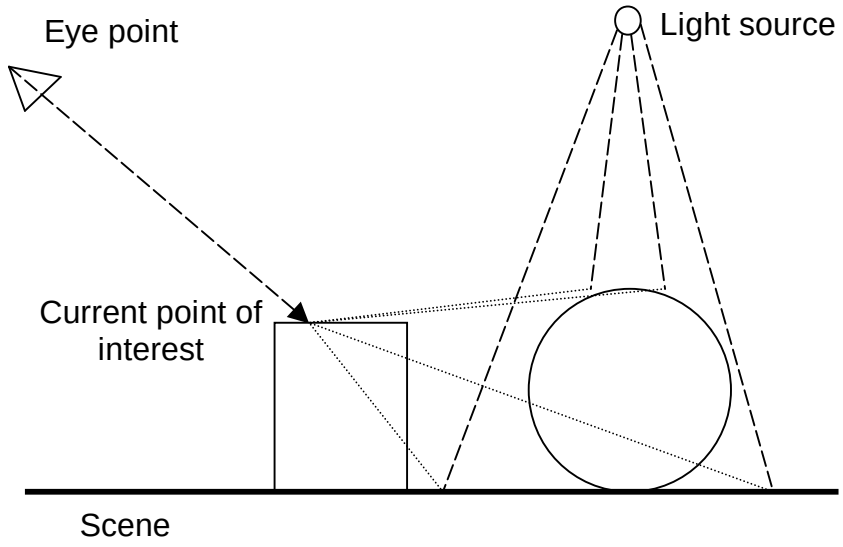


Figure 12.1: The basic idea of direct radiance mapping. For each point of interest we must calculate the light contribution from each directly illuminated point in the scene.

(in a similar way as propagation in the topological network was described in section 11.2) we can split it into propagation of radiance from the light sources and propagation of importance from the eye or the camera (recall the concept of importance from section 3.6).

The first step is then propagation of radiance from the light sources. We could denote this as follows: $L_i = \mathcal{P}L_e$. Incident radiance is determined by the point of intersection, the incident direction, and the incident power. These information and, hence, the entire field of direct incident radiance can be measured using images from each light source. The first point of intersection can be found as described in section 11.7. Incident directions follow from the position of the light source where an image is taken from. The direct incident power is given by the geometry term (G) between the light source and the point of incidence (visibility need not be considered, since everything seen from the light source will also be illuminated).

The purpose of storing these information representing the incident radiance field is not to estimate direct illumination. Better methods exist for that. Instead another image is taken this time from the eye point picturing the first step of importance propagation. As with the radiance propagation image the first points of intersection are also stored for importance propagation. The incident directions are given by the eye point and the importance of the locations seen directly from the eye is always one (see (3.60)).

The positions of direct incident radiance are now stored in a texture which we might refer to as a direct radiance map. It is quite similar to a photon map since each texel could be regarded as a photon. On the other hand it does not have a tree structure, and nor is it used for radiance estimates in the

same way as a photon map is. The positions of direct incident importance are stored in what we might refer to as a direct importance map. The texels could in this case be thought of as importons[1]. All in all we now have two maps; one of direct radiance and one of direct importance.

To propagate the light waves further we need only consider the light that by one way or another propagates to points of importance. One way to propagate waves is by means of random signals in a connected digraph (directed graph). If we let the edges of a graph, connecting 'importons' of the direct importance map with 'photons' of the direct radiance map, be given by an outer product, the result is a directed graph where each importon node is fully connected to all photon nodes (and oppositely each photon node will be fully connected to all importon nodes). Random signals in such a connected digraph would simulate the propagation of light if all importon nodes were fully visible to all photon nodes.

To start out with the simple case, we assume that all surface materials are Lambertian. In that case propagation of the signals can be performed completely at random, and we need merely gather radiance propagated from random photon nodes towards random importon nodes and weigh the signals by the normals. We even do not need to take the incident direction into consideration when signals are propagated in the network.

The simple case is too reductive with respect to photorealistic images. As always expansions come at large expenses. First we ought to introduce a BRDF at each photon node either sending signals towards certain directions or weighing signals according to incident direction and material properties. This is a feasible task which enables the possibility of materials that are not perfectly diffuse (Lambertian).

To model the indirect illumination properly, we should take the geometry in-between two nodes into account. This expansion from the simple case is the most expensive and difficult part of the global illumination problem. Visibility between two points could be checked in order to account for indirect shadows.

Specular reflection and refraction are also a part of the indirect illumination, but techniques such as environment mapping can deal with these cases reasonably well, see section 6.3.

The light propagation network is constructed in order to compute the indirect illumination reflected diffusely between surfaces. To summarize on the expansions; a BRDF and storage of extra material properties in the direct radiance map is necessary to model non-Lambertian surfaces, visibility check for each edge is necessary to include indirect shadows.

The process of light propagation can continue over several frames, which means that the longer a scene keeps static the better the indirect illumination

[1]The term "importon" was coined by Peter and Pietrik in [103] to denote photons emitted from the observer.

will become.

To simulate subsurface scattering each importon node in the direct importance map can be copied, jittered (at random in a direction pointing into the translucent material) and stored beneath the surface as another layer. The more translucent the material the more layers we need. The subsurface layer is connected to the direct importance map as the direct importance map is connected to the direct radiance map, in that way we need only propagate the radiance into the next layer (according to a BSSRDF) in order to simulate subsurface scattering.

A problematic theme in the method presented is the lack of light reflected diffusely more than twice, but we must keep in mind that each diffuse reflection will weaken the contribution of the signal considerably. Using environment mapping for reflection and refraction we can only account (under the simplifying assumptions of environment mapping) for the light paths given, in light transport notation, as `LS*DDS*E`. Calculations of direct illumination and specular reflections account for the paths `LD?S*E`. The sum of these paths exclude visual effects such as caustics resulting from the paths `LS+DE`. Some different methods introduced in chapter 6 propose ways to render caustics in real-time, and since none of the light paths for caustics are included in our method, such methods could be adopted for caustics and combined with the method we present to expand the subset of indirect illumination that we can simulate.

If additional indirect illumination is necessary, one approach is to distribute some regularly spaced points to different surfaces throughout the scene, these points should be relatively few so that they can be intermediary nodes fully connected to both the photon nodes and importon nodes. When light is propagated from the photon map it may reach the intermediary nodes and bounce around before it reaches the importon nodes. Such a network would capture additional indirect illumination, but surely at large expenses. An alternative to interconnection between the two maps and the intermediary nodes is to take a low resolution picture at each intermediary point and to connect the nodes only to the importon nodes. The signal to propagate from the intermediary nodes is then a mean of the illumination measured at the point through the picture taken.

Another approach to multiple bounces is to extend the idea of environment maps to include diffuse surfaces. This idea is more easily explained when the actual implementation of direct radiance mapping has been discussed.

Conclusively we may note that the method presented does not fully solve the global illumination problem, rather it solves a subset of the problem. The method can be combined with many existing methods for real-time rendering that do not simulate indirect illumination in order to achieve additional visual effects. The method can in an efficiency/correctness trade off achieve effects such as subsurface scattering and an (almost) arbitrary subset of

indirect illumination in a physically plausible way.

In the following section we will describe how the direct radiance mapping method is founded in the theory described in part I and moved towards a real-time simulation of global illumination step by step.

12.2 The Resulting Method

The abstract formulation of direct radiance mapping, or the conceptual idea, was given in the previous section. The idea of a network between direct photons and direct importons is, we think, good for development of the idea. From a practical point of view the abstract thoughts will have to take a different form. This section will describe the method on a form that is practical for implementation.

Local Illumination from an Isotropic Point Light Source

The general proposal in this method is to solve the simple case first and then increase complexity gradually. Therefore we first describe direct illumination before moving on to the direct radiance mapping idea. The simplest case assumes perfectly diffuse (or Lambertian) surfaces and only direct local illumination from an isotropic (see definition 6) point light source. Meaning that we partly solve the first piece of the rendering equation:

$$L_{o,0} = L_{e,0} + L_{r,0} \tag{12.1}$$

where $L_{e,0}$ is radiance emitted uniformly in all directions and $L_{r,0}$ is radiance reflected off perfectly diffuse surfaces. Hence, the equation finds a very simplified version of the outgoing radiance.

We can even calculate $L_{r,0}$ analytically. As derived in section 3.2, the radiance incident at a point $\boldsymbol{x}$ from a light source of area A uniformly emitting the power $\boldsymbol{\Phi}_s$, is:

$$L_{i,1} = L_{e,1} = \frac{\boldsymbol{\Phi}_s}{\pi A} \tag{12.2}$$

where the subscript 1 indicates an area light source. If we assume a point light source, we can not use the same formula. Rather we should consider the emitted light intensity at the point light source, since light intensity depends solely on the differential solid angle $d\omega$ describing a directional volume (see (3.11)):

$$I = \frac{d\boldsymbol{\Phi}}{d\omega}$$

An isotropic light source emits a constant light intensity, $I_{e,0}$, in all directions which gives us the following result by integration over the entire unit sphere around the point light:

$$\begin{aligned} \boldsymbol{\Phi}_s &= \int_{\Omega_{4\pi}} I_{e,0}\, d\omega = I_{e,0} \int_0^{2\pi}\int_0^{\pi} \sin\theta\, d\theta\, d\phi = I_{e,0}\, 4\pi \\ \Rightarrow I_{e,0} &= \frac{\boldsymbol{\Phi}_s}{4\pi} \end{aligned} \tag{12.3}$$

Recall the relation between a differential surface area dA and the differential solid angle $d\omega$ subtended by dA (see (3.49)):

$$d\omega = \frac{\cos\theta dA}{r^2}$$

where θ is the angle between the normal $\boldsymbol{n}$ at dA and the direction towards the incident light, and r is the distance between the point at which $d\omega$ is formed and the center of dA. The *irradiance* at a differential surface area dA centered around a surface location $\boldsymbol{x}$ caused by the flux through a differential solid angle $d\omega$ formed at the point $\boldsymbol{y}$ can then be calculated using the intensity emitted at $\boldsymbol{y}$:

$$E_i = \frac{d\boldsymbol{\Phi}}{dA} = \frac{d\boldsymbol{\Phi}}{dA}\frac{d\omega}{d\omega} = I\frac{d\omega}{dA} = I\frac{\cos\theta dA}{r^2 dA} = I\frac{\cos\theta}{r^2}$$

which under the assumption of local illumination, meaning that we do not consider occluding objects, and an isotropic point light source results in the following:

$$E_{i,0} = I_{e,0}\frac{\cos\theta}{r^2} = \frac{\boldsymbol{\Phi}_s}{4\pi}\frac{\cos\theta}{r^2}$$

Again recalling the theory of preceding chapters (see (3.24)), the radiance reflected at a point is proportional to the irradiance incident at the very same point:

$$f_r = \frac{dL_r}{dE_i}$$

which means that we, under the assumption of Lambertian surfaces, get the following result by integration over the incident irradiance:

$$L_{r,0} = \int f_r\, dE_i = \frac{\rho_d}{\pi}E_{i,0} = \frac{\rho_d \boldsymbol{\Phi}_s \cos\theta}{(2\pi r)^2} \tag{12.4}$$

Like a blackbody, a point light source has no true physical existence, but is merely a hypothetical object. Therefore a visual representation of the light source itself is senseless. This means that we can not, and should not, calculate the radiance emitted directly from a differential surface area on the light source $L_{e,0}$, since a point has no area. In other words:

$$L_{o,0} = L_{r,0}$$

which means that we have an analytical solution of the simplest case. This solution can be typed into a fragment program giving us a renderer that can easily run in real-time, though it can only render perfectly diffuse materials lit by isotropic point light sources.

If we want to draw area light sources and still use this simple model, we can approximate the light sources by a point light source at their center and draw their area using $L_{o,1} \approx L_{e,1} + L_{r,0}$. However, if we let a point light source approximate a square area light source, we should integrate over one hemisphere only, which results in the following radiance reflected from surfaces beneath the light source:

$$L_{r,0} = \frac{\rho_d \mathbf{\Phi}_s \cos\theta}{2(\pi r)^2} = \frac{\rho_d \mathbf{\Phi}_s (\boldsymbol{n} \cdot \boldsymbol{\omega}')}{2(\pi r)^2} \tag{12.5}$$

Though it is a coarse approximation, the square light source is *drawn* using some constant value for emitted radiance while the actual *light emission* comes from a point light source at the center of the fake square light source.

Direct illumination from an Isotropic Area Light Source

Expanding the simplest approach a little, we can find the direct illumination resulting from isotropic area light sources, if we assume that it is very distant from the scene:

$$L_{o,1} = L_{e,1} + L_{r,1}$$

The reflected radiance $L_{r,1}$ can be found using the area formulation of the rendering equation (see (3.54)):

$$L_{r,1} = \int_S f_r \, L_{i,1} G \, V dA$$

where the arguments have been left out for simplicity. S denotes the union of all the area light sources, f_r is the BRDF, $L_{i,1}$ is the radiance incident from isotropic area light sources, G is the geometry term, V is the visibility term, and dA is a differential area on the isotropic area light source. Since we assume isotropic light sources and that the they are located far away from the scene geometry, the integral can be simplified as follows:

$$L_{r,1} = f_r \, L_{i,1} G \, V A$$

Inserting the result given in (12.2), the geometry term, and the BRDF for Lambertian surfaces, we have:

$$L_{r,1} = f_r \frac{\mathbf{\Phi}_s}{\pi} G V = \frac{\rho_d \mathbf{\Phi}_s \cos\theta \cos\theta'}{(\pi r)^2} V \tag{12.6}$$

which is not very far from the point light source approximation in (12.4). The visibility term is still included. An assumption of local illumination would eliminate it by setting it to one. The correct value of it would be $V = A_{\text{visible}}/A$, where A_{visible} is the visible light source area as seen from the location where the radiance is reflected. The most common approximation to V is, however, a hard shadow method such as shadow volumes (see sec. 6.1), which approximates the light sources by point lights and thereby render the objects as fully in shadow or fully occluded with respect to each light source.

Figure 12.2a shows the result of $L_{o,1} \approx L_{e,1} + L_{r,0}$, that is, local illumination using a point light approximation of the area light source in the Cornell box. Figure 12.2b shows $L_{o,1} = L_{e,1} + L_{r,1}$ under the assumption of local illumination for comparison, and figure 12.2c shows the same including a shadow volume algorithm. The renderer uses a fragment shader.

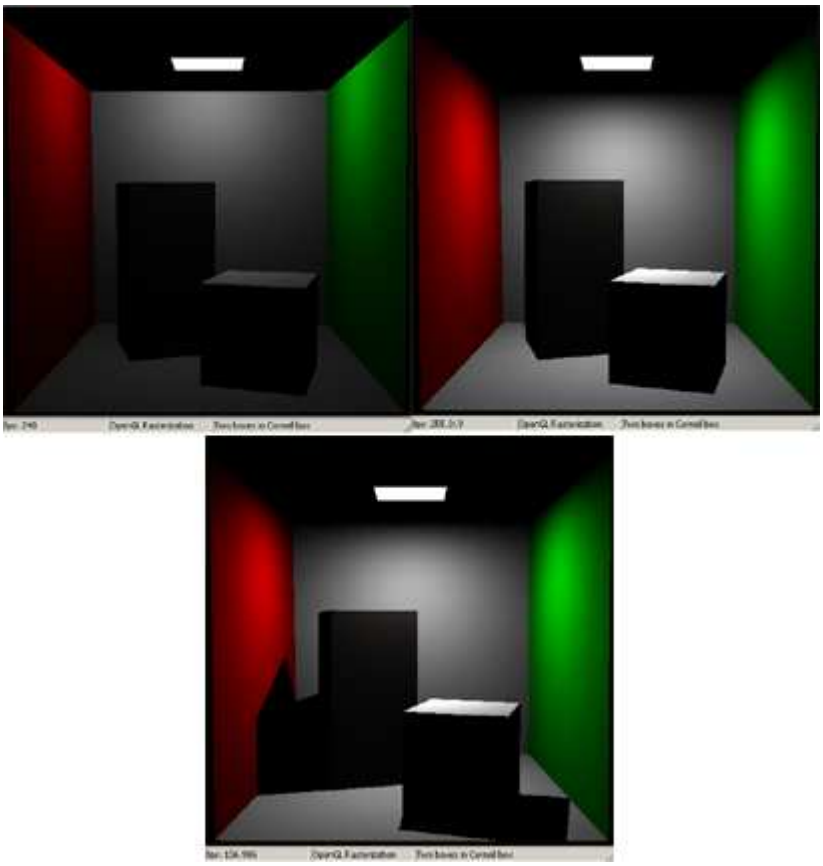


Figure 12.2: The Cornell box rendered using analytical local illumination solutions implemented in a fragment shader. From top left to bottom: (a) The square light is approximated by a point light at the center of the square. No visibility, only local illumination. (b) Area light source calculation, but still only local illumination. (c) Visibility term approximated by a point light at the center of the square.

Having a decent approximation of the direct illumination, it is time to expand our model a little. Thinking about radiosity we know that multiple diffuse reflections result in a soft lighting of the Cornell box and that we should be able to discover the visual phenomenon known as color bleeding. In light transport notation radiosity models the light paths `LD*E`. The direct illumination model merely calculates the contribution from the light paths `LDE`, even with a crude approximation of the visibility term only. Instead of solving the expensive radiosity equation in order to find all contributions of diffusely reflected light, we start out modestly and look at the contribution from the light paths `LDDE`. In the following we describe a version of the idea presented in section 12.1, which is practical for implementation.

Light Reflected Diffusely Twice

Assuming that all surfaces are perfectly diffuse they reflect the same amount of radiance in all directions. Therefore the radiance that we can picture from the light source, using a local illumination model, is also the light that is reflected in all other directions over the hemisphere at each surface location seen from the light source. In other words the direct radiance reflected off the surface locations seen from the light source gives us means by which we can calculate the contribution of the light reflected diffusely twice.

Suppose we take a picture from the light source capturing the direct radiance. Then we can let each fragment of the image represent the radiance reflected at a specific surface location. The contribution of all these fragments to another surface location is given as:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) = \int_\Omega f_r(\boldsymbol{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\boldsymbol{x}, \boldsymbol{\omega}') \cos\theta \, d\omega'$$

where $L_i(\boldsymbol{x}, \boldsymbol{\omega}') = L_{r,0}$ if we do not take visibility between surface locations into account, or $L_i(\boldsymbol{x}, \boldsymbol{\omega}') = L_{r,1}$. Knowing approximate values of $L_{r,0}$ or $L_{r,1}$ at each fragment of the image seen from the light source, we can approximate the integral by accumulation of the contribution from each fragment:

$$L_r(\boldsymbol{x}, \boldsymbol{\omega}) \approx \sum_M \frac{\rho_d}{\pi} L_{r,0}(\boldsymbol{n} \cdot \boldsymbol{\omega}') \, \Delta\omega'$$

where M denotes the set of all fragments in the picture taken from the light source, and $\Delta\omega'$ is the spherical-surface area, on the unit sphere centered at $\boldsymbol{x}$, which is intercepted by the solid angle subtended by the surface area pictured in a single fragment.

In practice we store the image in a texture representing direct radiance as seen from the light source, hence the name of the method: *Direct radiance mapping*. A texture simply storing direct radiance as seen from the light source does, however, not store sufficient information. We need to calculate the direction towards the incident light $\boldsymbol{\omega}'$ for each fragment. The solution is

to store another texture containing the corresponding intersection point for each fragment (exactly as done in sec. 11.7). We regard both these textures to be a part of the direct radiance map.

Though the images taken from the light sources need not be in very high resolution, it is still too expensive to calculate a summation over all texels for each fragment in the final picture from the eye point, and $\Delta\omega'$ is not found easily either. One possible solution is to sample the texture in order to keep up a reasonable frame rate.

Monte Carlo integration (see sections 3.8 and 4.3) is a possibility in order to solve the integral by sampling. Using the probability density function (PDF) $p(\boldsymbol{\omega}') = 1/A_s$, where $A_s = 2\pi$ is the area of the unit hemisphere, gives the following integral estimator:

$$\langle L_r(\boldsymbol{x}, \boldsymbol{\omega})\rangle = \frac{1}{N}\sum_{i=0}^{N}\frac{\rho_{r,i}}{\pi}\frac{L_{r,0,i}(\boldsymbol{n}\cdot\boldsymbol{\omega}_i')}{p(\boldsymbol{\omega}')} = \frac{2}{N}\sum_{i=0}^{N}\rho_{r,i}L_{r,0,i}(\boldsymbol{n}\cdot\boldsymbol{\omega}_i') \qquad (12.7)$$

where N is the number of samples. The sampling is unfortunately not straightforward, since no build-in random functions exist on the graphics card, meaning that we can not easily pick new samples from one fragment to another. Moreover the samples will have to be few, since concurrent fragment programs are quite susceptible to looping; all loops will be unrolled resulting in a slow fragment program (since long fragment programs are slow). The simple solution is to choose the same regularly spaced samples for all fragments, which results in a relatively good approximation in some points and a bad approximation in others. The almost unacceptable approximation in this approach is that we cannot know whether the sampling is done according to the PDF, which assumes a uniform sampling across the hemisphere over the fragment.

If, for example, an object is located close to the light source filling out the entire direct radiance map, then almost for sure the samples will not be spaced uniformly over the hemisphere. Rather the solid angle which the sample points subtend on the hemisphere will be very small. One unwanted effect resulting from this is that the indirect light will be heavily overexposed when an object comes close to the light source unless we find a PDF that can counteract against this problem.

Despite the problems, we constructed a fragment program using the same samples for each fragment. For the sake of simplicity we chose nine sample points spaced regularly throughout the direct radiance map. The result is shown in figure 12.3.

Instead of $p(\boldsymbol{\omega}') = 1/2\pi$ we could use a PDF denoting uniform sampling in the solid angle $\omega_{\mathcal{V},i}$ formed at the position of fragment i and subtended by the contour edge of a bounding volume $\mathcal{V}$ containing the nine regularly spaced sample points, see figure 12.4. At the application stage of the pipeline

Figure 12.3: The Cornell box including a nine points sampling of a direct radiance map for indirect illumination reflected diffusely exactly twice. For this image the hemispherical formulation of the rendering equation was used with the approximative regular sampling.

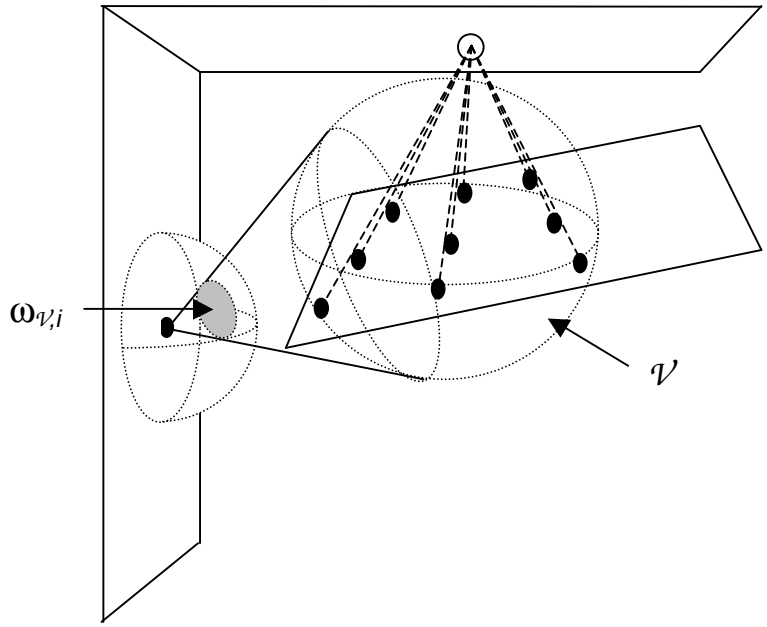


Figure 12.4: The solid angle $\omega_{\mathcal{V},i}$ formed at the position of fragment i and subtended by the contour edge of a bounding sphere $\mathcal{V}$ containing the nine regularly spaced sample points. The figure sketches the situation in figure 12.12, where color bleeding is overexposed.

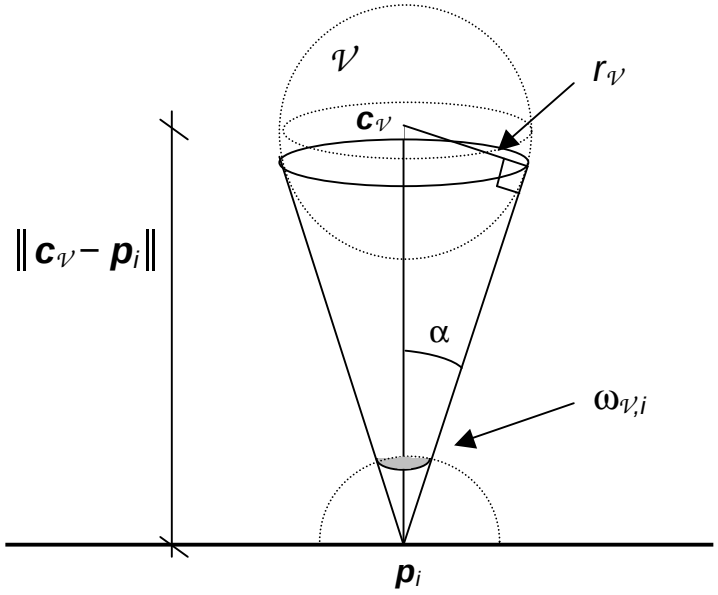


Figure 12.5: If the bounding volume $\mathcal{V}$ is a bounding sphere, the solid angle $\omega_{\mathcal{V},i}$ subtended by the sphere can be found using spherical coordinates.

(see sec. 5.1) we could find this bounding volume $\mathcal{V}$ and either approximate the solid angle (which may be different for each fragment) by the solid angle $\omega_{\mathcal{V},\boldsymbol{c}}$ formed at the center of the scene $\boldsymbol{c}$ and subtended by $\mathcal{V}$, or we could send the bounding volume down the pipeline and compute $\omega_{\mathcal{V},i}$ for each fragment. In any case a better PDF for (12.7) is given as:

$$p_i(\boldsymbol{\omega}') = \frac{1}{\omega_{\mathcal{V},i}} \approx \frac{1}{\omega_{\mathcal{V},\boldsymbol{c}}}$$

Suppose we let $\mathcal{V}$ be a bounding sphere given by a center $\boldsymbol{c}_{\mathcal{V}}$ and a radius $r_{\mathcal{V}}$. Then the solid angle formed at the position $\boldsymbol{p}_i$ of fragment i and subtended by $\mathcal{V}$ could be found in spherical coordinates as shown in figure 12.5. Recalling (3.10) from the description of solid angles in section 3.2, we can find the solid angle as follows:

$$\omega = \int_0^{2\pi} \int_0^{\alpha} \sin\theta \, d\theta \, d\phi = 2\pi(1 - \cos\alpha)$$

where α is the angle shown in figure 12.5, which means that:

$$\omega_{\mathcal{V},i} = 2\pi(1 - \frac{\sqrt{\|\boldsymbol{c}_{\mathcal{V}} - \boldsymbol{p}_i\|^2 - r_{\mathcal{V}}^2}}{\|\boldsymbol{c}_{\mathcal{V}} - \boldsymbol{p}_i\|})$$

If we let $\mathcal{V}$ be an AABB, we could approximate $\cos\alpha$ by the angular visibility between the point $\boldsymbol{p}_i$ and $\mathcal{V}$ (angular visibility was described in section 11.1).

Another way to get an integral estimator of the reflected radiance is to use the area formulation of the rendering equation. If we let the PDF denote uniform sampling across the surfaces in the direct radiance map, $p(\boldsymbol{x}) = 1/A_M$, where A_M is the area of the surfaces in the map, the resulting estimator will be:

$$\begin{aligned} \langle L_r(\boldsymbol{x}, \boldsymbol{\omega}) \rangle &= \frac{1}{N} \sum_{i=0}^{N} \frac{\rho_{r,i}}{\pi} \frac{L_{r,0,i}(\boldsymbol{n_x} \cdot \boldsymbol{\omega}_i')(\boldsymbol{n_y} \cdot -\boldsymbol{\omega}_i')}{r_{\boldsymbol{xy}}^2 p(\boldsymbol{x})} \\ &= \frac{A_M}{N} \sum_{i=0}^{N} \frac{\rho_{r,i}}{\pi} \frac{L_{r,0,i}(\boldsymbol{n_x} \cdot \boldsymbol{\omega}_i')(\boldsymbol{n_y} \cdot -\boldsymbol{\omega}_i')}{r_{\boldsymbol{xy}}^2} \end{aligned} \tag{12.8}$$

It should be noticed that the normal is now needed in the direct radiance map as well as it is in each fragment that we render. This means that the map must contain a third texture holding the normal associated with the intersected surface location. Also recall from section 5.1 that a polygon close to the camera may fill out the entire image and a polygon very far away may fill out a single pixel only. This indicates that calculation of the area of the surfaces seen in the direct radiance map A_M not necessarily is done easily.

If we approximate A_M by a constant and place an object close to the light source, the indirect light will again be heavily overexposed, since the actual area seen from the light source is not very large. Besides this problem (12.8) also results in a more expensive fragment program than the one needed to evaluate (12.7). In any case figure 12.6 shows an image found using (12.8) and a constant approximation of A_M for comparison with figure 12.3.

An idea for a rough approximation of A_M is to take the value $\boldsymbol{v}_0$ of the texel at position $(0, 0)$ and the value $\boldsymbol{v}_1$ of the texel at position $(1, 1)$ (the first and last) in the texture holding first intersection points, and find the squared distance between the two points that they return:

$$A_M \approx (\boldsymbol{v}_1 - \boldsymbol{v}_0) \cdot (\boldsymbol{v}_1 - \boldsymbol{v}_0)$$

This could be done at the application stage (see sec. 5.1) and supplied to the fragment program as a uniform parameter each frame.

In both cases, the low number of samples makes the estimate quite fragile. The few sampling points clearly show themselves when we use the area formulation as seen in figure 12.6. The remaining indirect illumination is, however, more correct than in figure 12.3. When using the hemispherical formulation of the rendering equation, the sampling problems will show themselves in the overall indirect illumination (instead of at specific points). Putting an object exactly where a sample is taken will result in a significant change in the indirect illumination. This can cause flickering while objects are moving.

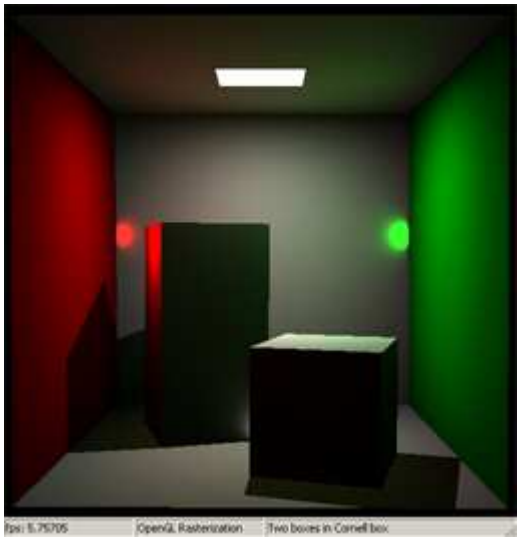


Figure 12.6: The Cornell box including a nine points sampling of a direct radiance map for indirect illumination reflected diffusely exactly twice. Here the area formulation of the rendering equation was used with the sampling.

The sampling is root to many problems, therefore we have thought of many ways to work around it. One idea was to use mipmapping (see sec. 5.3). Unfortunately we cannot use mipmapping in the same way as it is exploited in the method described in section 6.7. The reason is that the real-time photon mapping method has many small textures that can be mipmapped in a standard, hardware optimized mipmapping procedure, we have a few textures which we want to mipmap differently for each fragment. This is not immediately possible, and probably will not be unless mipmapping becomes a build in function available in a fragment program. Different possibilities of improvement will be discussed in chapter 16.

If we can not find a way to avoid sampling, we can at least have confidence that future fragment programs will be able to render more samples each frame. This will improve the method gradually as the graphics hardware improves.

Even when the graphics hardware has improved to enable more samples, or summation over the entire map, the calculation of either $\Delta\omega'$, A_M, or $\omega_{\mathcal{V}}$ must be done adequately. Unfortunately we have not had the time to explore the ideas for calculation of A_M and $\omega_{\mathcal{V}}$ which we proposed above. So far we will, therefore, settle for the sample images in figures 12.3 and 12.6 with respect to indirect illumination reflected twice, and move on to further expansions of the concept.

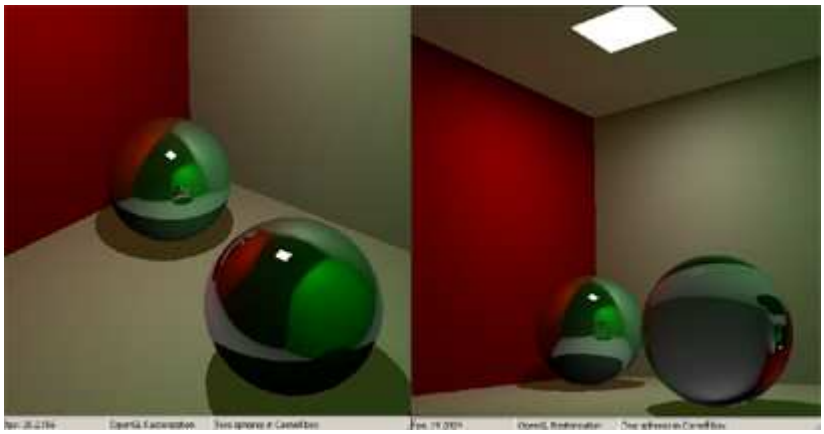


Figure 12.7: The Cornell box rendered using direct radiance mapping combined with recursive environment mapping. The two boxes have been replaced by spheres having the following materials. From left to right: (a) Two chrome (or mirror) spheres. (b) A chrome and a glass sphere.

Arbitrary BRDFs

Suppose we let the normals associated with the intersection points be a part of the direct radiance map, as is the case if we use the area formulation of the rendering equation for sampling as described above. Then storing the pure material color instead of the reflected radiance in our direct radiance texture, would make it possible (if enough processing power is available for each sample in each fragment) to model an arbitrary BRDF. Even more textures might be needed for the direct radiance map, such as one holding the direction of incidence if more than one light source exist, and one holding three material parameters for BRDF calculation at the first intersection point from the light source. This expansion is very expensive, at current hardware it quickly results in fragment programs exceeding the allowed number of instructions, nevertheless, it indicates that arbitrary BRDFs can be modeled using direct radiance mapping.

Perfectly Specular Reflections

As described in section 6.3 specular recursions can be handled in real-time using environment mapping. Combining this approach with the direct illumination, and the diffuse illumination reflected twice, we do, as previously mentioned, end up with a method able to model the light paths: `LS*DDS*E` and `LD?S*E`. Some results are shown in figure 12.7, where the boxes are replaced by two spheres.

Figure 12.8: A glossy effect is obtained by adding indirect illumination to otherwise perfectly specular objects. From left to right: (a) The planar mirror in the cave scene excluding the glossy effect. (b) The planar mirror in the cave scene including the glossy effect.

An Inexpensive Glossy Effect

An inexpensive way to make perfectly specular surfaces (such as the spheres in figure 12.7) look glossy, is to simulate that the object is diffuse with respect to indirect illumination. The direct illumination is replaced by the color found in the reflected direction, but the diffusely reflected indirect illumination is added to the object. Since no perfectly specular objects exist in real life, this method gives mirrors a natural feeling, see figure 12.8.

Multiple Bounces

An expansion of the method following the multi-agent line of thought (see sec. 11.4) would, as mentioned in section 12.1, be to let each object have an environment map of the radiance in its surroundings. The idea is then to add a few samples of indirect illumination from the environment map (sampled according to the BRDF) to the final shade of each fragment that we render. Including fragments rendered when the environment maps are updated. In this way the indirect illumination will include more bounces for each frame rendered, in a way analogous to the recursive environment mapping for perfectly specular surfaces. Unfortunately we had this idea too late to have time for an implementation. The method seems appealing except for the fact that the frame rate will decrease as the number of objects in the scene increases, because each new object will need another six images for update of an environment map each frame.

Another idea is to sample in the previous image taken from the eye point when drawing the texture of radiance from the light source, There are several

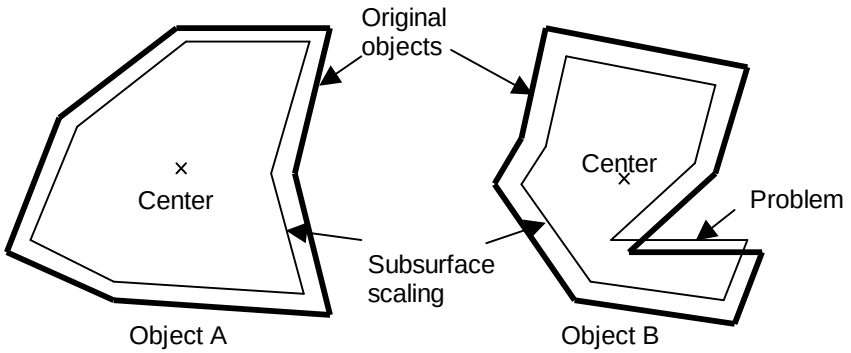


Figure 12.9: The scaling for real-time subsurface scattering. Since scaling is around the center problems can easily occur as illustrated through object B.

limitations to this approach though. First, only objects seen from the eye point or from a light source can have an influence on the illumination in a scene. Second, only reflections off perfectly diffuse surfaces would be possible. The reason is that if we use an arbitrary BRDF as described above we need a texture with pure material color instead of radiance, which is incompatible with the concept.

Subsurface Scattering

The original idea for an expansion of DRM to include subsurface scattering (see section 12.1) was to expand the network by randomly jittered nodes stored inside a translucent object. Since the network has no actual existence (only an abstract existence) in our implementation of the method we had to alter the concept a bit.

According to [60] the reciprocal of the extinction coefficient (described in section 3.4) corresponds to the average distance that a photon will move through a medium before interaction:

$$d = \frac{1}{\sigma_t}$$

Using this average distance it is our idea to scale a translucent object uniformly around its center to the size $1 - d$ and draw it inside the original object, see figure 12.9. Unfortunately there is a problem in this simple scaling, since the scaled object may end up outside the original object as seen in the figure. To solve this approximately, the stencil buffer is used to mark the original object and only the part of the scaled object which ends up inside the original object will be stored.

To continue the concept of direct radiance mapping, the radiance incident on the diminished version of the original object, which represents subsurface scattered light, is also found as an image from the light source and stored as a texture in the direct radiance map.

For simplicity we only model the first intersection below the surface of the object and we assume that this intersection takes place at the average distance. These approximations result in an optical depth which equals unity:

$$\tau(s, s') = \int_{s'}^{s} \sigma_t(t)dt \approx \sigma_t \Delta s \approx \sigma_t d = 1$$

If we do not take in-scattered light into account, but only direct radiance subsurface scattered once, the attenuated radiance reaching the diminished object will be given as follows:

$$L_{\text{subsurf}} = (1 - F_r)e^{-1}L_{i,1}$$

where F_r denotes the Fresnel reflectance.

The point is, now, that we can render a subsurface scattered illumination term for all translucent objects and add it to the direct and indirect illumination terms found previously. This is done by considering the radiance transported to the diminished object representing light scattered beneath the surface once. When looking at the surface of a translucent object the integral finding in-scattered light at the surface location can be approximated by a sampling of the subsurface scattered light.

The simplest way to find a contribution from the subsurface scattered light is to use a single sample. We let the center of a translucent object be projected in the direction towards the light source into a position in the direct radiance map. This position in the direct radiance map will usually also specify a position in the subsurface scattered illumination. Letting this single sample compute the contribution from the subsurface scattered illumination results in the following:

$$L_i = e^{-\sigma_t \Delta s}\, \sigma_s\, p(\theta)\, L_{\text{subsurf}}$$

where σ_t is the extinction coefficient, σ_s is the scattering coefficient, and $p(\theta) = 1/(4\pi)$ is the face function (we have chosen to work with isotropic scattering only). Δs is found as the distance between the subsurface scattered position and the surface position.

This extremely simple version of subsurface scattering has some reasonable results. Of course the illumination is mostly incorrect, but the visual cue certainly exists, see figure 12.10 for a comparison between a scene where the subsurface scattering is included and a scene where it is not.

It should be noted that the subsurface scattering requires two extra textures in the direct radiance map: One for subsurface radiance and one for subsurface positions.

At the expense of processing time, the sampling could be increased to have a better estimate of the subsurface scattered illumination.

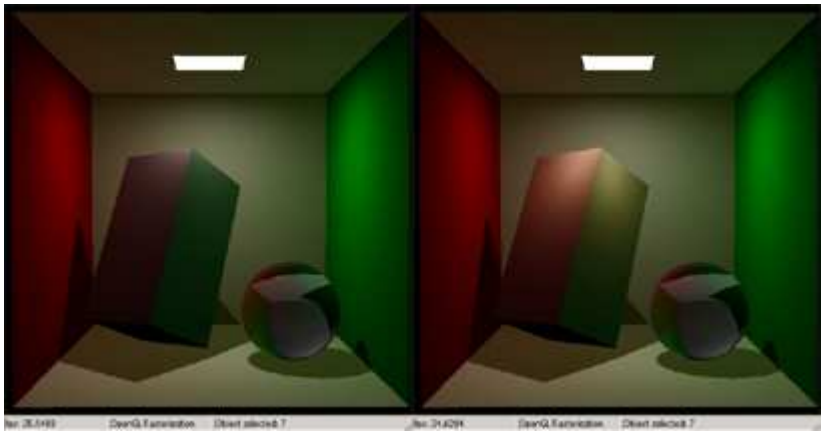


Figure 12.10: The Cornell box containing a translucent box and a glass sphere. From left to right: (a) Scene rendered with no subsurface scattered illumination. (b) Scene rendered including subsurface scattered illumination.

Subsurface scattering is the final expansion of direct radiance mapping that we have worked with. In the next section we will summarize the different abilities of the method and take a closer look at the limitations of the method.

12.3 Abilities and Limitations

In the previous section we found an analytical solution for the direct illumination term resulting from isotropic light sources. Following this we wanted to find the first bounce of indirect light using a direct radiance map composed of pictures taken from the light sources. To calculate indirect illumination we then needed for each fragment in the resulting image, taken from the eye point, to find the contribution of light from each fragment in the pictures taken from the light sources. Unfortunately this is beyond the capabilities of current GPUs both with respect to limited number of fragment program instructions and with respect to real-time frame rates. Therefore we simplified the direct radiance map significantly by using low resolution images (currently we use 32x32 pixels) for the textures in the direct radiance map. From these crude textures we are currently able to calculate the contribution of indirect light from only nine sample points spaced regularly across the direct radiance map. More samples result in unacceptable (that is, not real-time) frame rates on a NVIDIA GeForce FX 5950 graphics card.

That we use low resolution textures is, in our opinion, not necessarily a disadvantage. When the textures are used for simulation of multiple diffuse reflections the low resolution may have a smoothing effect which is often desirable when simulating diffuse reflections. The limited number of sample points is a major source of flickering and aliasing artifacts. In order to

increase the number of sample points we could distribute the calculation over multiple frames, so that only part of the light was calculated in each frame. This would give a low sampling rate approximation when things move dynamically and a continuously improving image while camera and objects stay static. The flickering will still be apparent since it is mostly visible when things move around dynamically.

Our limited implementation of the method approximates $\omega_{\mathcal{V}}$ and A_M by constants. This results in a heavy overexposure of indirect illumination when objects move close to the light source. We proposed approximate solutions for this problem in the previous section, but have not had the time to test these solutions. Therefore the overexposure will show itself when using the demonstration application (see figure 12.12). We see this not as a limitation to the method, but rather as a technicality which we can find a solution for.

In the following we will first list the abilities and afterwards problems and limitations of direct radiance mapping. Both have been indicated in the previous sections, therefore we will only sum them up shortly. The abilities are:

- Direct radiance mapping is a method for creating indirect illumination in real-time. In real-time we are able to calculate an approximation of the first bounce of diffusely reflected indirect illumination in each frame. This enables visual effects such as color bleeding.
- The method is independent of number of vertices in the scene, meaning that it is bounded by the number of fragments rather than the number of vertices when rendering.
- Since we recalculate all illumination terms for each frame, objects and light sources can move about freely in the scene.
- The method is independent of the geometry in the scene, that is, we do not have to consider the shape or complexity of an object.
- Our method is based on the GPU, meaning that CPU processing power is left for other purposes and that the method will improve as graphics cards improve.
- Many extensions have been proposed giving rich opportunity for better versions of the rendering method.

Although the above capacities sound appealing there is also a back side of the medal. We will now discuss the problems and limitations to our current implementation method. Along the way we will suggest possible solutions to the problems or describe what it would take to solve them. The problems and limitations are:

- Only first bounce is calculated
- We assume that all objects are diffuse while calculating the indirect illumination.
- Indirect light is calculated in a local illumination manner, in other words we do not take indirect shadows into account.
- The method is fragment bound.
- (The direct radiance map is low resolution.)
- Only a low sampling rate is affordable in the calculation of indirect illumination using the direct radiance map. And the same sample points are used in all fragments.
- $\omega_{\mathcal{V}}$ or A_M is approximated by a constant.

The limitations result in a number of artifacts in the scene. Some appear always others are not visible until the scene becomes dynamic. We will now go through these one by one and discuss them.

As long as only the first bounce is calculated some indirect illumination will, of course, be missing. The result is an image, which is a little darker than would have been the case if we had included multiple light bounces. Proposals for multiple bounces were given in the previous section.

The assumption that all objects are diffuse is *not* a fundamental necessity for the method. It is merely assumed in the implementation for efficiency reasons. Expansion to arbitrary BRDFs is also described in the previous section.

The fact that we do not consider visibility in the estimate of the diffusely reflected illumination term results in a number of artifacts. One problem is visualized in figure 12.11 showing a screen shot of the cave scene. Here the room behind the door is illuminated by indirect lighting, which is not correct, since the door should keep the room completely occluded. The error occurs because indirect light is calculated in a local illumination manner where objects blocking the path of the indirect illumination are not taken into consideration. To avoid this problem we must take indirect shadows into consideration in the calculations. A solution is not straightforward. The proposal of an environment map for each object, which can model multiple light bounces, will also include indirect shadows. This approach is maybe the best solution.

The method does not so much depend on the number of triangles or vertices in the scene, which allow us to create complicated scenes without loosing too much frame rate. In contrast the method is bound by the number of pixels that needs to be rendered, or in other words the method is fragment bound. This means that the higher screen resolution we use the lower

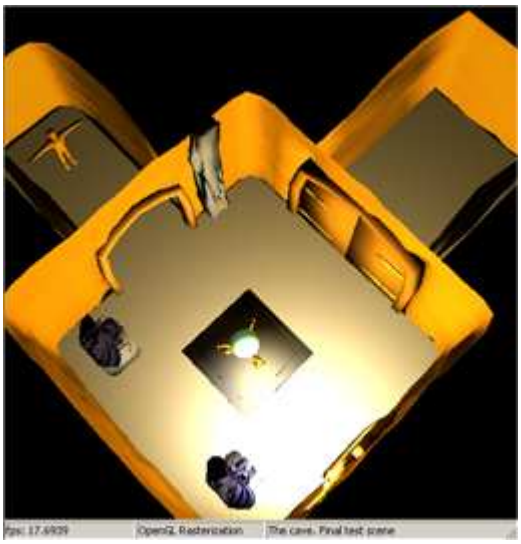


Figure 12.11: The cave scene. Notice that the corridor behind the door is illuminated by indirect light which is not correct.

frame rate we get. It should, however, be noted that GPU developers encourage fragment bound programs since the efficiency of fragment processors currently is developing at an incredible speed.

The direct radiance map is low resolution (for example 32x32 or 64x64), meaning that the diffusely reflected indirect illumination will be less detailed and, hence, less correct. On the other hand the sampling rate we can afford is so low that a very detailed texture will result in more problems than it solves. The few samples mean that smaller objects have a good chance of being missed in the indirection illumination calculations.

The nine regularly spaced point samples are too few, the problem clearly shows in figure 12.6 in the case where we use the area formulation of the rendering equation. It is more difficult to show the flickering that it causes when we use the hemispherical formulation. To see this we refer to the demonstration application on the attached CD-ROM, see appendix A.

A more correct sampling would choose different sample points instead of the same nine always. This would probably cause other problems though, since it would be harder to control the total amount of illumination in the scene and as long as we only use nine sample points we could risk constant flicker even when no objects are moving.

Figure 12.12 shows the problems resulting from the overexposure of indirect illuminations and in figure 12.12a the problem of missing indirect shadows is also apparent. As described in the previous section, the overexposure appears when objects are placed close to the light source. The problem is that we approximate $\omega_{\mathcal{V}}$ and A_M as constants. Better ways to calculate these were proposed in the previous section, we hope that they will

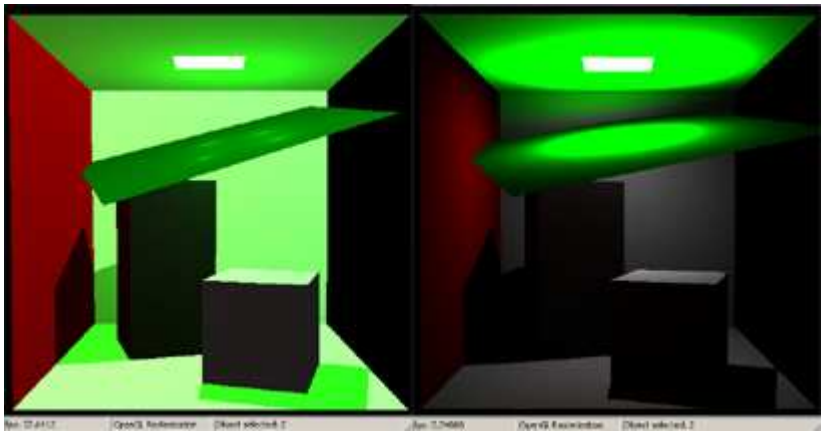


Figure 12.12: The problems resulting from approximation of $\omega_{\mathcal{V}}$ and A_M by a constant. When objects are placed close to the light source, indirect illumination gets overexposed. From left to right: (a) The green wall of the Cornell box moved close to the light source. The indirect illumination is approximated by the hemispherical formulation of the rendering equation. Here the missing indirect shadows are also apparent. (b) The same as (a) only using the area formulation of the rendering equation.

solve the problem in a reasonable way.

As the final part of this section we will briefly discuss our simplified approach to subsurface scattering.

Using the concepts of direct radiance mapping we have created a very minimal simulation of subsurface scattering. The outcome of this method is not quite precise enough for a detailed comparison with other methods (such as those described in section 3.4) creating subsurface scattering in real-time (or at interactive rates).

Very few experiments have been carried out with the method as we have mainly concentrated our efforts on indirect illumination. This leaves us with several corrections and improvements which could be contributed to the subsurface scattering implementation. A few will briefly be mentioned in the following while we describe some of the problems and limitations to the method for subsurface scattering which we presented in the previous section.

First of all we can only afford to use one sample point to calculate the subsurface scattering of an object. This is the main reasons why the subsurface scattered light often looks somewhat odd. More sample points would be beneficial but as it is now they are too expensive with regards to the frame rate, at least if we want to combine the subsurface scattering with the indirect illumination.

Another problem is the simplified way in which we create the inner object. Since we scale the object towards its center we risk that parts of it is cut away because it falls outside of the original object (cf. fig. 12.9). This means that from unfortunate angles the subsurface scattering effect disappears entirely, if the sample point is set where there is no inner object available. A better

way to scale the object would be to translate each vertex in a negative direction from its normal. In this way the inner object would never come outside the boundaries of the original object (unless it is translated all the way through the object and out through the surface on the opposite side).

Since the inner object is an exact replicate of the original object (only smaller) the distance to the surface will always be the same. It might give a more realistic effect if the distance could vary a little. An idea for generating a more scattered distance is to put several scaled objects inside each other and then pick a random one. In this way we could have an arbitrary number of layers with different distances to the surface to pick from randomly. To create effects even closer to the original physics we could establish bounces between the layers as well. However this would probably be an expensive procedure.

After a discussion of the most important drawbacks let us return to what we have accomplished. We are able to find the first bounce of indirect light in an arbitrary scene no matter how complex the objects are. The light is recalculated at a frame rate of 10-30 fps (see the lower left corner of the different screen shots, or try out the application). Moreover we have proposed a simple version of subsurface scattering, which gives reasonable results and many options for improvements taking the many simplifying assumptions into account. The next section will compare the method to other implementations that we have come across in the literature study of this project.

12.4 Comparison

Direct radiance mapping is mainly a method for real-time simulation of the part of the rendering equation, which in section 4.4 is described as the multiple diffuse reflections term. Other methods are able to do the same; some of these are described in chapter 6. This section will compare direct radiance mapping to each of the methods described in sections 6.6, 6.7, 6.8, and 6.9 with respect to the ability to simulate the multiple diffuse reflections term. The methods will be compared to direct radiance mapping one by one.

Light Maps vs. Direct Radiance Mapping

The light map method uses pre-calculated global illumination (most often radiosity) to apply indirect illumination in a scene. The illumination is static but in diffuse environments it provides a good visual effect even if it does not change according to the surroundings. The method is used in several newer games; an example is given in [109]. A basic assumption of light maps is that, as long as the light sources are static and not particularly remarkable, the fact that diffusely reflected indirect illumination is static is not noticeable. This is often true in a game scenario where swift movement is frequently

required and the user's attention is set on opponents. In such cases indirect lighting can be treated as an atmosphere in the background where static effects seem sufficient.

A disadvantage of light maps is the pre-calculation, which makes the indirect light unchangeable. This means that even if elements with significant influence on the scene are altered the light will stay the same. Also, if the light source is moved, we might be able to change the direct illumination but the indirect illumination stays the same. Indirect lighting has an influential effect on a scene, this can be seen in the JR Viewer by toggling it on and off in the different rendering algorithms and observing the difference. Defects like this can be minimized using additional light maps for different situations, but all cases cannot be covered. To fully provide correct indirect illumination in all cases, we must be able to update at any time. Direct radiance mapping is able to calculate direct illumination in every frame regardless of the number of moving objects and light sources.

The drawback in direct radiance mapping (besides flickering in some situations) is the missing indirect shadows and the fact that only the first bounce of indirect illumination is available. If these problems could be solved and the frame rate could be raised we would have a much more correct method for dynamical indirect illumination. The fact that direct radiance mapping would be able to update the indirect illumination in real-time can add new effects to the scene. For example changes in color bleeding caused by movement of objects.

Real-Time Photon Mapping Simulation vs. Direct Radiance Mapping

The method described in section 6.7 implements photon mapping in real-time - almost. This gives an excellent result regarding global illumination but unfortunately there are a number of other restrictions to the method. We need to predetermine points in the scene to simulate final gathering, this gives limitations in scene size and complexity. Dynamic objects must be predefined as they need specific calculations and in order to enable ray tracing capabilities in real-time we must do some preparations for spatial data structures such as BSP trees.

Compared to direct radiance mapping, which uses a very approximative version of diffuse reflections, we will find that the real-time photon mapping simulation method is much more accurate. In fact, it probably provides the most correct indirect illumination of all methods discussed here, which do not use extensive pre-computing. When it comes to accurate light calculation real-time photon mapping is better, but when it comes to scene dynamics it is not that flexible. Here the direct radiance method has huge advantages since it is independent of scene complexity and the number of moving objects.

Pre-computed Radiance Transfer vs. Direct Radiance Mapping

In pre-computed radiance transfer a large part of the calculations are carried out in advance. The result of the pre-computation is a method capable of simulating almost any type of visual effect on a single object, even in a frame rate useful for commercial applications such as computer games. However, the method still has some limitations; it must assume that objects are rigid, otherwise pre-computation of global illumination would not be possible, and as described in section 6.8 the method has problems the moment several dynamic objects need to interact in the same scene. These are both issues that do not affect direct radiance mapping since it is independent of the scene geometry.

In other words the most important drawbacks of pre-computed radiance transfer are the limitations regarding interaction between dynamic objects and the massive pre-computations needed. The advantage of pre-computed radiance transfer is the quality of the global illumination, which is only victim of the limitations in spherical harmonics basis functions. Pre-computed radiance transfer is good for simple scenes craving detailed illumination, such as a talking person. In comparison direct radiance mapping has almost opposite features. While there is no restrictions on scene dynamics, the diffusely reflected indirect illumination is crude and imprecise.

Environment Map Rendering vs. Direct Radiance Mapping

The method, which we have chosen to call environment map rendering[2] builds on the same philosophy with respect to scene dynamics as does direct radiance mapping. Therefore it has many of the same advantages as direct radiance mapping.

Environment map rendering (in the sense of Nijasure et al. [93]) is done by estimation of radiance transfer functions at the nodes of a regular grid spaced throughout the scene. The transfer functions are represented in a spherical harmonics basis and estimated using environment maps. For each location in the scene to be rendered an interpolation between the nearest transfer functions should decide the illumination at that specific location.

The problematic issues are the regular grid, the interpolation, and the usual limitations to spherical harmonics representation of transfer functions. First, the resolution of the regular grid depends on scene complexity. We can compare the nodes of the grid where environment maps are found to the sample points in direct radiance mapping. The interpolation can cause some color or shadow leakage through walls for example, which is again also connected to the spherical harmonics representation of the transfer functions.

[2]No particular name has been given to the method by its authors.

The inventors of the method suggest solutions for most of these problems (for more details see sec. 6.9).

All in all environment map rendering is a method having capabilities similar to direct radiance mapping. It also has some problems, but seemingly they are few. The frame rates are a bit lower (approximately 10 fps), but the result is more correct. The solution suggested for color and shadow leakage can not yet run in hardware. Nevertheless, the environment map rendering method presented by Nijasure et al. is a tough competitor to direct radiance mapping. Nijasure's method probably gets the upper hand.

Summary

The comparisons are summarized in table 12.1, which presents positive features of the different methods as we see them. The features listed should be seen with regards to the multiple diffuse reflections term only. For example it is possible to have dynamic objects in scenes using the light mapping method but the indirect light does not change with them.

Some markings are set in parentheses, this illustrate that there are certain conditions that must be fulfilled before it is true. The following list shortly sum up these restrictions.

- Real-time photon mapping simulation supports an arbitrary number of dynamic objects, but the dynamic objects must be pointed out in advance.

- In the description of real-time photon mapping it is stated that diffuse objects must be assumed. This is, however, only necessary in the last bounce of the indirect illumination, since interpolation between points distributed in the scene is used. At all light bounces except the second to last diffuse interaction an arbitrary BRDF can be employed.

- Pre-computation does occur in the real-time photon map simulation method, but only of BSP trees for speeding up the rendering process.

- Pre-computed radiance transfer supports both perfectly diffuse and glossy BRDFs. An inherent limitation to the spherical harmonics approach used in pre-computed radiance transfer, is that the closer the BRDF comes to a perfectly specular BRDF, the more spherical harmonics basis functions will the method need to model the reflection without serious aliasing artifacts. Therefore pre-computed radiance transfer only supports low-frequency lighting environments.

- Indirect shadows are supported in the environment map rendering method, but only if the method is implemented in software (not in hardware).

- Environment map rendering uses spherical harmonics in the same way as pre-computed radiance transfer, hence, it is subject to the same restrictions regarding BRDFs.

Looking at the table we find that methods building on pre-computation are the ones currently fast enough to be used in commercial applications. However, there are certain restrictions to them that are only fully solved using methods that do not need pre-calculations. On this observation we dare predict that other methods based on GPU intensive rendering will win in time (at least in some situations).

Another thing revealed from the table is certain drawbacks of the direct radiance mapping in features supported by other methods. In the report we have suggested solutions to many of these drawbacks, others might be solved by combination of our method with some of the others. Particularly interesting could be a combination of direct radiance mapping and environment map rendering as the two methods builds on the same philosophy carried out a bit differently. Although the table reveals fewer capabilities in some areas for direct radiance mapping compared to the other methods we still have faith in the idea, as it is at an early state of development.

This concludes our presentation of direct radiance mapping. The next chapter will shortly describe the different traditional rendering methods that we have implemented for reference.

Feature	Light Mapping	Real-Time Photon Map Simulation	Pre-computed Radiance Transfer	Environment Map Rendering	Direct Radiance Mapping
Support of single dynamic object		X	X	X	X
Arbitrary number of dynamic objects		(X)		X	X
Support of dynamic light sources		X	X	X	X
Support of deformable objects				X	X
Independent of geometry with respect to object shading		X	X	X	X
Independent of geometry with respect to processing speed	X		X	X	X
Independent of the number of light sources	X	X	X	X	
Independent of scene size	X				X
Support of indirect shadows	X	X	X	(X)	
Arbitrary BRDFs	X	(X)	(X)	(X)	X
Multiple light bounces	X	X	X	X	
Runtime sampling not required	X		X	X	
Pre-calculation not required		(X)		X	X
Sufficient frame rate for commercial applications (eg. games)	X		X		

Table 12.1: List of positive features for each of the different methods as we see them.

Chapter 13

Other Implemented Rendering Methods

Virtue is never isolated; it always has neighbors.

Confucius (552-479 BC.): *Analects* 4:25

Since it was the original idea for this project to combine photorealistic rendering and real-time rendering by having an implementation in each camp, we started out constructing a ray tracer and added different optimizations schemes of our own (described in chap. 11) before implementing our real-time renderer.

The result of this procedure is that we have implemented a fairly functional ray tracer as well as other global illumination methods, which we can use as reference for our results from direct radiance mapping (DRM). The drawback is perhaps that by putting focus in two places we have not been able to go as deep into one subject as we could have. On the other hand we feel that working with traditional photorealistic rendering has given us a better fundament for the real-time implementation; in fact, we think that it is doubtful whether we would have come up with any of the ideas presented in chapters 11 and 12, if we had not followed this procedure.

In this chapter we will describe rendering features implemented beside DRM. Section 13.1 will mention the rendering methods that have been implemented for photorealistic rendering, and section 13.2 will point out the methods that have been implemented for combination with DRM in our real-time renderer.

13.1 Photorealistic Rendering Methods

Radiosity, ray tracing, and photon mapping have all been implemented and are available in the application accompanying this report. Each of these methods have been implemented in a very simple version, since they are mainly there for reference. In the following we will point out which parts of each method that have been implemented.

Radiosity

Radiosity is a simple implementation, which only include two approaches: Radiosity using the simple analytical approach to the form factor calculation given in (4.4), in this case we do not consider visibility between patches, and radiosity using the hemicube method, where five pictures are taken for determination of each form factor.

A progressive refinement approach similar to the one described in section 4.1 is used to solve the system of linear equations. Our radiosity implementation features no expansions and is merely there for reference when we simulate the multiple diffuse reflections term using other methods. Some typical screen shots of our radiosity implementation are shown in figure 13.1.

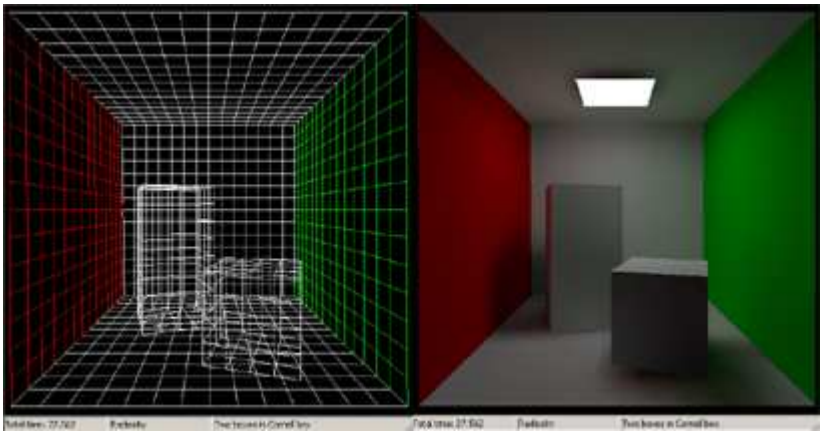


Figure 13.1: Screen shot of the scene patches for radiosity and the result of the hemicube radiosity solution (as implemented for this project) applied to the Cornell box.

Ray Tracing

A naïve one-ray-per-pixel ray tracer has been implemented. It follows the same concepts as described in section 4.2. The idea of angular visibility, described in section 11.1, is always used with the ray tracer, since it improves on rendering times without compromising image quality. Other spatial data structures have not been implemented. Hardware optimization as described in section 11.7 has been implemented as well. It can be toggled on and off, since it reduces image quality a little.

Soft shadows have been implemented using Monte Carlo ray tracing as described in section 4.3. A number of shadow rays (we use 50) are traced towards positions on a square area light source sampled using Quasi-Monte Carlo Halton sequences (see eg. [141]). The fraction of these rays reaching the light source denotes the visibility term.

The three simplest BRDFs have been implemented for shading in the ray tracer. The BRDFs are called Phong, Blinn-Phong, and Modified Blinn-Phong. All of them are described in section 5.2.

Photon Mapping

Photon mapping has been implemented as described in section 4.4, except we have not found time for an implementation of final gathering. Instead we divide the photon map into four different maps, one for each term that we wish to simulate. The four photon maps are used for direct illumination, caustics, indirect illumination, and shadows respectively.

The caustics photon map is used as described in 4.4. The indirect illumination map is used for a direct visualization of the indirect illumination. This is not optimal since a direct visualization of a photon map always results in low frequency noise. Our implementation is like this because we

had in mind to filter the direct visualization of the indirect illumination using a neural network, our project moved in another direction though. The shadow photon map is used for optimization of the soft shadow Monte Carlo ray tracing operation. If no shadow photons are found nearby, we need not calculate the visibility term we can merely set it to one.

If final gathering was to be implemented we would need to have a global photon map, which would be a combination of the direct, caustics, and indirect maps.

Some screen shots of our ray tracing solution including some photon mapping are given in figure 13.2.

These were, in short, the global illumination methods implemented and available through the JR Viewer (see chap. 14). In the next section we point out the different real-time rendering methods that we have implemented for combination with direct radiance mapping.

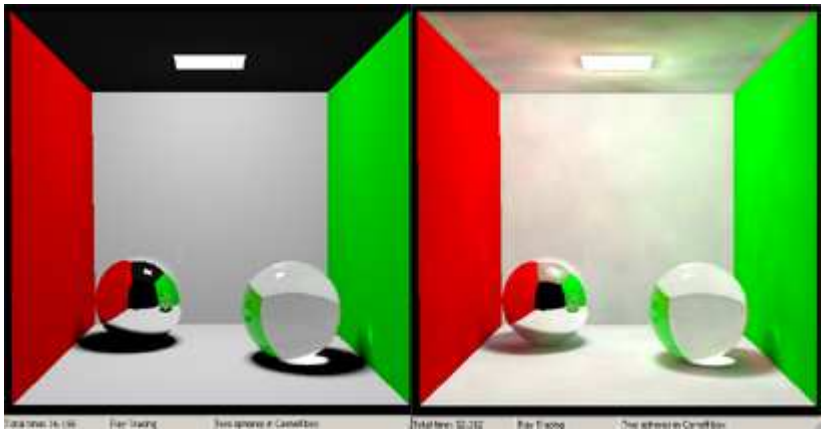


Figure 13.2: Screen shots of our ray tracing solution applied to the Cornell box. From left to right: (a) Ray tracing including soft shadows and a caustics photon map. (b) The same as (a), but also including a direct visualization of the photon map simulating multiple diffuse reflections. Except for the noise the secondary caustics should be noticed below the glass sphere.

13.2 Other Real-Time Rendering Methods

As described in chapter 6 many different real-time methods can be combined in order to simulate different rendering equation terms. The only rule to comply with is that no class of light paths can be included more than once. In this section we shortly point out the different methods that we have made available in our implementation for combination with direct radiance mapping. We will also mention a few of the methods that advantageously could have been combined with direct radiance mapping if we had had extra time for the implementation.

The basic real-time renderer implements the simple rasterization approach described in chapter 5. The standard shading is Phong highlighting. The expansions are as follows: First curved and planar reflections have been implemented as described in sections 6.2 and 6.3. It is an option in the implementation to use recursive curved reflections. Second fragment shading has been implemented for direct illumination as an alternative to the standard Phong highlighting. Shadow volumes have been implemented as described in section 6.1. Finally refractions have been implemented for the special case of a sphere.

Real-Time Glass Ball Rendering

Refractions are found using an environment map in the same way as for curved reflections only the refracted direction should be used for texture coordinates rather than the reflected direction. In the special case of a sphere the light refracts twice; once at the first intersection point and once when the refracted ray has passed through the sphere and is again refracted back into the environment.

The first refracted direction $\boldsymbol{\omega}_{t,1}$ is found using the refraction formula (3.5). The refraction formula is available as a hardware instruction on most GPUs. Now, to find the second refracted direction $\boldsymbol{\omega}_{t,2}$ we must find the normal, $\boldsymbol{n}_2$, at the second point of intersection. Considering figure 13.3 we quickly realize that $\boldsymbol{n}_2$ is the normal, $\boldsymbol{n}_1$, at the first point of intersection reflected in the direction opposite to the first refracted direction $-\boldsymbol{\omega}_{t,1}$.

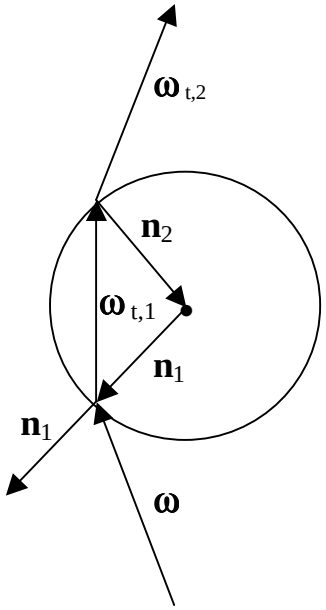


Figure 13.3: Double refraction through a sphere.

Using the reflection formula (3.2) we get:

$$\boldsymbol{n}_2 = 2(\boldsymbol{n}_1 \cdot -\boldsymbol{\omega}_{t,1})(-\boldsymbol{\omega}_{t,1}) - \boldsymbol{n}_1$$

and since the environment map assumes that all directions used for texture coordinates originate at the center of the object, the second refracted direction can be found by the refraction formula used between $-\boldsymbol{\omega}_{t,1}$ and $\boldsymbol{n}_2$.

The refraction should afterwards be blended with the reflection according to the Fresnel reflectance. If we use the simplified formula given in (3.36) and insert $\eta_1 = 1$ for air outside the sphere and $\eta_2 = 1.5$ corresponding to the refraction index of glass, we get the following simple Fresnel formula:

$$F_{r,\text{glass}}(\theta) = 0.04 + 0.96(1 - \cos\theta)^5$$

where θ is the angle of incidence.

Sphere Caustics

Creating caustics in real-time for arbitrary objects is a difficult task. The most promising method we have found is the one presented in [74], which is also described shortly in section 6.4. Caustics for a simple primitive like a sphere, resulting from light emitted from a point light source, can be approximated in a straightforward manner. To create caustics in real-time for a sphere we could create a cone with an axis through the center of the sphere. In a plane also through the center of the sphere and perpendicular to a line between the sphere center and the light source, we could mirror the light cone. The caustic caused by the light will then appear where the cone intersects with another surface. The concept is illustrated in figure 13.4.

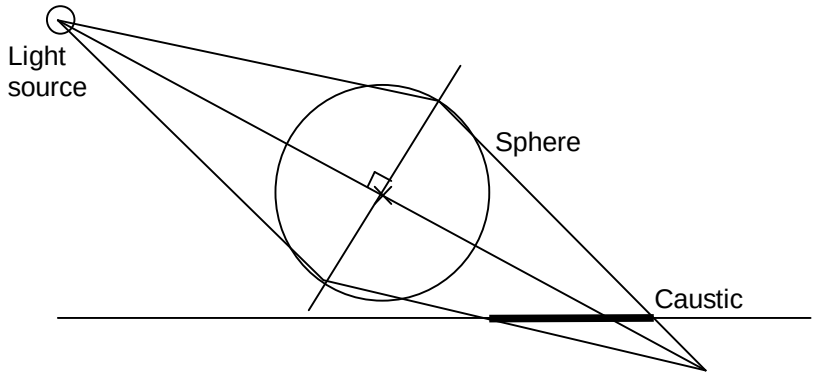


Figure 13.4: A fast approximation of a sphere caustic.

The caustics area can be marked in the stencil buffer and illumination attenuated according to Fresnel reflectance can be added to the caustics area.

Similar simple assumptions about light traveling through other primitive objects could be created. However, the method will not work for arbitrary objects and it gives only a crude approximation. To make real caustic calculations, rays must be traced inside the object. In [74] it is described how this can be done in real-time. Unfortunately there has been no time in this project for an implementation of real-time caustics (not even the crude sphere caustic presented here).

The remaining real-time rendering methods that have been implemented were described in chapter 12. In the following chapter we will give a presentation of the GUI that has been developed for the application implementing the different rendering methods.

Chapter 14

Graphical User Interface

Un croquis vaut mieux qu'un long discours.

Napoleon Bonaparte (1769-1821)

Many projects concerning global illumination in real-time limit themselves to a proof of concept only using predefined test scenes. This is, of course, quite alright, but if the method developed is to be applicable for a commercial application, it must be able to cope with more complex scenarios. To take the project a little step further we have enabled a way to create scenes in a modeling application and then export them to a graphical user interface (GUI) where our illumination methods can be tested on the scene. This is a part of the second objective for this project in which we want to show the work flow from modeling to real-time rendering using our own methods

Section 14.1 will describe the test scenes created for the project. Each object in each scene will be identified and their purpose will be explained. About half of the test scenes are written directly in code since they to a certain extent are benchmark scenes (Cornell boxes containing different simple objects, the Cornell box data can be found in [95]). The rest of the scenes are created in Blender. How to use Blender is explained in chapter 10.

The scenes are available through the windows GUI. How to use the GUI and all its features is covered in section 14.2.

14.1 Test Scenes

To demonstrate our implementations of rendering methods and to compare them to each other we have created a number of test scenes available through a small demonstration application: The JR Viewer. In this section we present the test scenes one by one in order to explain what each of them is useful for.

The scenes are available through the file dropdown menu (see next section). The following scenes can be selected:

- Cornell Box - Empty
- Cornell Box - Two Boxes
- Cornell Box - Two Spheres
- Cornell Box - Sphere and Box
- (Cornell Box - Orb)
- (Cornell Box - Arthur)
- Cave Scene

We will now present the scenes one by one.

Cornell Box - Empty

The Cornell box is a computer graphics benchmark scene invented at the Cornell University and used very frequently. The box is physically built and detailed material parameters are measured from it. The data can be found in [95]. It is an almost quadratic box with one side open towards the camera. In this way an almost quadratic room is created. The floor, back wall and ceiling are white (not purely white, just white), while the left wall is red and the right wall is green (figure 4.1 in chapter 4 shows a Cornell box with two boxes inside it). The empty Cornell box is the simplest test scene we have and all rendering methods work for this scene.

Cornell Box - Two Boxes

Traditionally the next level of complexity for Cornell boxes is to place two boxes in them, as a representation of simple objects. With the two boxes we can better illustrate color bleeding and shadows. Subsurface scattering is enabled for the tall box in the scene for demonstration of our crude real-time implementation of this effect. The Cornell box with two boxes is the only scene besides the empty Cornell box that is available for radiosity rendering. This is because we transferred the radiosity implementation, as previously mentioned, from a DTU course (02561) merely to have radiosity for reference. The two boxes scene is therefore well suited for color bleeding comparison.

Cornell Box - Two Spheres

In this scene the two boxes are replaced with two other simple objects: Spheres. The spheres have different material parameters. The left sphere is a chrome (or mirror) sphere while the right sphere is a glass sphere. This scene is made to demonstrate reflective and refractive rendering methods created for the project. When working in the rasterization menu (see next chapter) both spheres will have chrome material if refractions are not enabled.

Cornell Box - Sphere and Box

The Cornell box containing a sphere and a box combines the box with subsurface scattering and the refractive sphere. This gives an opportunity to test (almost) all the different rendering methods in the same scene. In this way we can also challenge the renderer and see if it can keep up the frame rate when all the different methods are applied at the same time.

Cornell Box - Orb

This Cornell box, containing the orb, is the first scene which has not been implemented in code, but exported from Blender. The orb represents a

slightly more complex object. The orb itself has glass material parameters while the pedestal beneath it has varying material parameters. The idea of this scene is to have a few more triangles in the scene. In that way we can quickly show the limitations of our ray tracer. The orb was exported from Blender using our 'homemade' export directly to code, therefore ray tracing is also possible in the orb scene. Unfortunately the orb has currently not been inserted in the application and is therefore not available.

Cornell Box - Arthur

Yet a little more complex is our gargoyle (Arthur), who is also created in Blender. Arthur is composed of 4861 vertices resulting in 6024 faces, and he has subsurface scattering enabled. This serves the purpose of showing that the subsurface scattering method can be used on more complex objects as well. The larger number of vertices can also indicate that our method is indifferent to how many vertices we have in the scene. Arthur is exported using the `xml_export.py` script and imported using the BMesh, this import is not yet connected to our ray tracer, and therefore this scene currently cannot be ray traced. Arthur is also not available in a Cornell box at the moment.

Cave Scene

The cave scene was constructed as an example of a scenario that we could meet in a commercial application. The scene has a little of everything. There is a plane mirror, Arthur the gargoyle, and a gargoyle twin (named Conrad). Only one of the gargoyles has subsurface scattering material parameters to show the difference. To better examine the indirect lighting there is an open and a closed corridor. The surrounding cave has a warm color which bleeds onto the floor that is otherwise gray. All objects in the cave scene can be moved around freely. There is also a person in the cave scene, who has even been animated to move around with a lantern holding a light source. Exporting the animation and making it work in the JR Viewer is a future plan.

Looking back at chapters 11 and 12 most of the different scenes can be found in the various figures containing screen shots. Having an overview of the different scenes, we can proceed to a description of the JR Viewer itself.

14.2 JR Viewer

To demonstrate our implementations we have created a small application called JR Viewer. The application is created for MS Windows, meaning that it only runs on Microsoft platforms. The illumination algorithms are on the other hand platform independent. The program structure implemented

behind the application is described in chapter 15. Figure 14.1 shows a screen shot of the JR Viewer. In this section we shortly present the functionalities of JR Viewer.

Figure 14.1: JR Viewer.

When the application is started, the only menu point available is "File". In general the menu points of the application are only available when functionalities for the chosen scene are available. For example all menu point are available for the Cornell box with two boxes, but for the Cornell box with two spheres the radiosity menu is unavailable. In this way we hope to avoid problems. Many of the methods depend on graphics hardware, in cases where the needed hardware is not available, effects that can not be rendered are grayed as well.

All scenes loaded have a track ball enabled, so that the scene can be examined from different view points. It is also possible to select objects by holding the mouse over them and pressing `P` to select. When an object is selected the track ball will work on the selected object instead of the scene. The math behind the track ball was presented in section 9.3. Note that track ball functionalities for objects only work under rasterization, track ball functionalities for camera navigation work both in radiosity and in rasterization. When rendering with ray tracing the camera is locked at a sensible camera position in the scene and objects are placed at their original positions. The track ball functionalities were also described in section 9.3.

At the bottom of the application window there is a status bar. The first field of the status bar shows the rendering time, or the frame rate when this is appropriate. The second field shows the rendering method and the third

field (in general) shows the current scene.

In the following we will present the dropdown menus one by one.

The File Menu

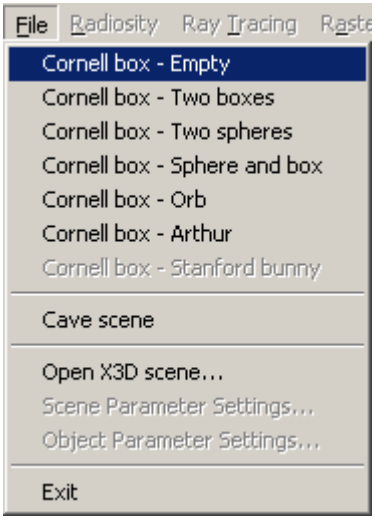


Figure 14.2: The File menu.

In the File menu, see figure 14.2, we find all the available test scenes. We have chosen to make the scenes described in the previous section directly available[1] so that a user more easily can make sample images at all times. Beneath these test scenes we have made an option for import of an arbitrary X3D scene. For reasons mentioned in the previous section, it is only possible to use the rasterization menu with such a scene.

When a new scene is loaded all settings are reset. This means that no matter the settings that were selected for the last scene and the rendering method that was used, all settings will we deselected and the rendering of the scene will appear as plane OpenGL rasterization when a new scene is loaded.

Two setting dialogs are also available in the file menu. There is a setting dialog for scene parameters and a setting dialog for object parameters. The scene parameter settings dialog is only available when a scene has been opened and the object parameter settings dialog is only available when an object has been picked. Figure 14.3 shows the scene parameters dialog and figure 14.4 shows the object parameters dialog. Parameter can be set to exact values by the text fields. Object parameters have associated slide bars for interactive changes to the parameters. Otherwise parameters should be

[1] Some of the scenes may be grayed because we did not find time to insert them in the final frame work.

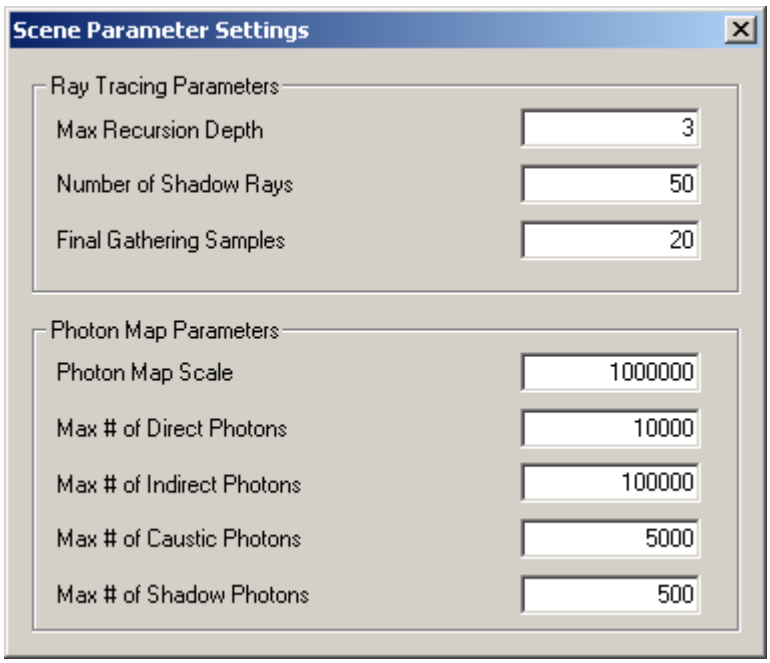


Figure 14.3: Screen shot of the scene parameter dialog.

updated by the current number when <RETURN> is pressed. To leave the text fields without updating <ESC> should be pressed. Unfortunately we have not had the time to couple the parameter settings to our render engine, but the windows exist for future use.

Since the parameter settings still do not quite couple to the render engine, they will not be described any further at this point.

Using the last file menu option we can exit the application.

The Radiosity Menu

Through the radiosity menu (figure 14.5) we can select a few radiosity rendering methods. The radiosity menu will only work with two scenes: The empty Cornell box and the Cornell box with two boxes. The radiosity rendering is merely present in this application for comparison with the other implemented methods.

As described in section 13.1, we can chose between two different methods for radiosity solutions: The analytical method and the hemicube method. Also available is a view of the patches used in the calculations.

Camera movement is possible after the radiosity solution has been calculated. Objects can not be moved without recalculation of the radiosity solution, therefore object selection has been disabled under the radiosity menu.

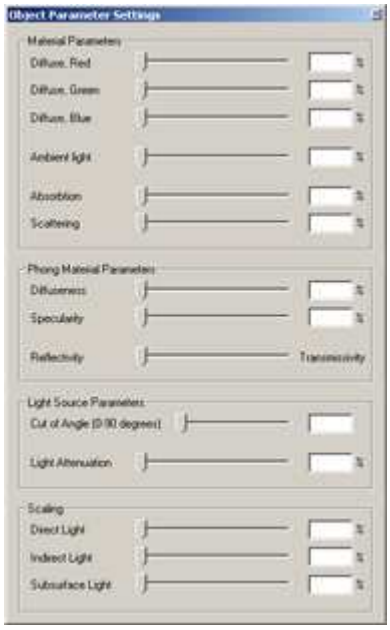


Figure 14.4: The object parameter dialog.

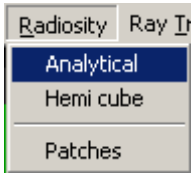


Figure 14.5: The Radiosity menu.

The Ray Tracing Menu

The ray tracing menu (figure 14.6) enables ray tracing of the simple scenes. Different rendering equation terms, shading methods, and shadow methods can be applied in the ray tracing process. These are chosen from the menu. The menu presents a number of effects that can be toggled on or off with a mouse click. Some effects rule out others, for example it makes no sense to render both hard and soft shadows, hence if soft shadows are selected, hard shadows are automatically deselected. Setting points that rule each other out are isolated from the other points by lines (the first points do not rule out each other). Some menu points have no effect on some scenes, for example it gives no effect to enable specular recursions if none of the objects in the scene are specular. When appropriate settings have been selected the menu option "Go..." should be selected for ray tracing of the scene.

Since no interactivity is available after ray tracing the resulting image is stretched over a quad as a texture in order to make it scalable. The original

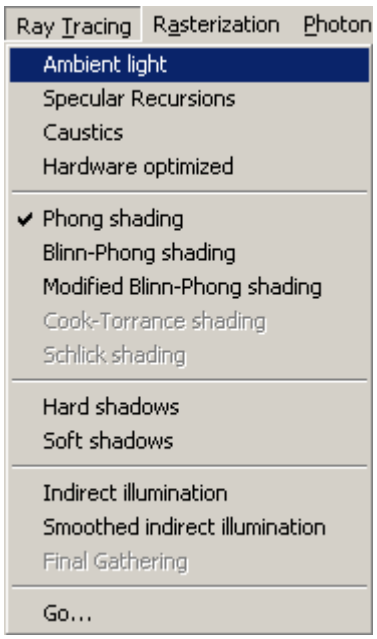


Figure 14.6: The Ray Tracing menu.

texture is 512x512 pixels. In ray tracing we have chosen to have the camera fixed at a sensible position in the scene. The scenes in which the ray tracing menu is available were pointed out in the previous section.

In chapter 13 we described the different rendering methods that are available with pointers back through the report. In the following we will shortly describe the different menu options for ray tracing and photon mapping:

- "Ambient light" adds a constant ambient term to the rendering equation in order to give a crude simulation of multiple diffuse reflections.
- "Specular recursions" spawn reflected and refracted rays if the material parameters indicate that it is necessary.
- "Caustics" estimate the caustics term of the rendering equation by use of a caustics photon map.
- "Hardware optimized" toggles calculation of the first intersection in hardware using the idea described in section 11.7.
- "Phong shading" uses Phong shading for the direct illumination term. Phong shading deselects the other available shading options.
- "Blinn-Phong shading" uses Blinn-Phong shading for the direct illumination term and deselects the other available shading options.

- "Modified Blinn-Phong shading" uses modified Blinn-Phong shading for the direct illumination term and deselects the other available shading options.
- "Hard shadows" enables shadow rays.
- "Soft shadows" enables a Quasi-Monte Carlo estimation of the visibility term.
- "Indirect illumination" enables a direct visualization of a photon map containing indirect illumintation reflected diffusely at least once.
- "Smoothed indirect illumination" is a cheap version of the indirect illumination option. Therefore these two options exclude each other.
- "Final gathering" is a menu option for future implementations.

The Rasterization Menu

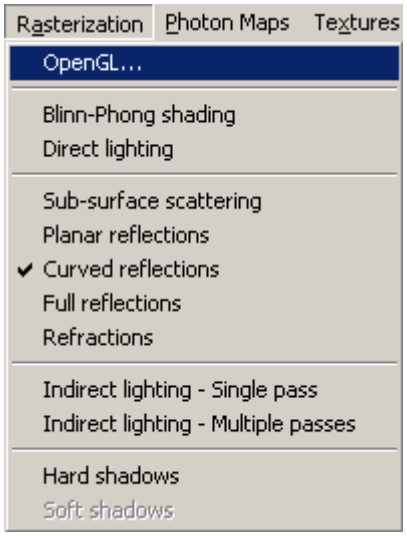


Figure 14.7: The Rasterization menu.

In the rasterization menu, see figure 14.7, we find the available real-time rendering methods. Any scene can be rendered using these menu points. When a new scene is loaded it will automatically be rendered using traditional real-time rendering (as described in chapter 5) implemented in OpenGL. At this point no menu options are selected. In contrast to the other menus all settings in the rasterization menu are changed right away when they are clicked. In the rasterization menu direct radiance mapping is responsible for the indirect lighting options. As in the ray tracing menu some settings rule

out others and some settings are irrelevant for some scenes. To shift to rastization after having used another rendering method (such as ray tracing), select the top menu option "OpenGL...".

The following list describes the different options one by one.

- "Blinn-Phong shading" switches from the standard Phong highlighting to Blinn-Phong shading implemented in a fragment shader.
- "Direct lighting" enables an analytical solution of the direct illumination term which is described in chapter 12. The Blinn-Phong shading and the direct lighting options both simulate the direct illumination term, and therefore they exclude each other.
- "Sub-surface scattering" enables our crude simulation of subsurface scattering as described in chapter 12.
- "Planar reflections" allows real-time planar reflections.
- "Curved reflections" use environment mapping for real-time curved reflections.
- "Full reflections" makes sure that the correct lighting is used in the curved reflections and it also makes the curved reflections recursive.
- "Refractions" enables the specialized refractions described in section 13.2.
- "Indirect lighting - single pass" enables direct radiance mapping recalculated for each frame.
- "Indirect lighting - multiple passes" calculates direct radiance mapping over several frames. This implementation exists in an early version of the application, but unfortunately we did not find the time to incorporate it in the current frame work.
- "Hard shadows" enables shadow volumes.
- "Soft shadows" are in the future works department.

The Photon Maps Menu

Several effects under the ray tracing menu uses photon mapping. To give an idea of the technology behind the scene, we have made a visualization of the different photon maps available in the Photon Maps menu, see figure 14.8. Any combination of photon maps is allowed. The following photon maps are available:

- "Direct light" shows the direct photons, which are photons stored when the surface intersected first is a diffuse surface.

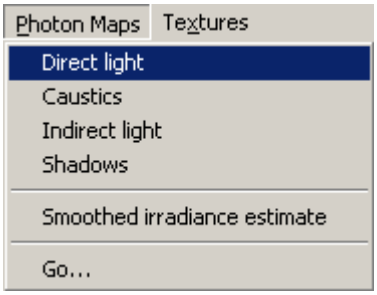


Figure 14.8: The Photon Maps menu.

- "Caustics" shows the photon map storing photons that have followed the light paths `LS+D`.
- "Indirect light" shows the remaining photons, which have been stored neither in the photon map for direct illumination nor in the photon map for caustics.
- "Shadows" store a special kind of photons called *shadow photons*, they are presented in [62]. When a direct photon is stored a shadow photon is simply traced in the same direction as the direct photon, but from the intersection point and onwards through the scene storing shadow photons on all diffuse surfaces that it encounter on its way.

The "Go..." option traces the photons and visualizes the photon map. A last option which has also been placed in this menu is "Smoothed irradiance estimate" which shows a cheap direct visualization of the photon map storing indirect light. This is the estimate used in the option "Smoothed indirect illumination" of the Ray Tracing menu.

The Textures Menu

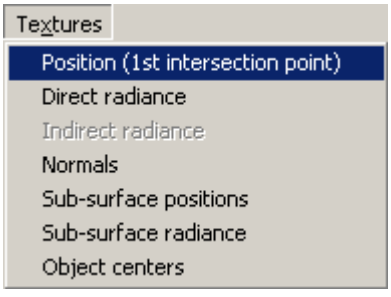


Figure 14.9: The Textures menu.

The direct radiance map consists of different textures. A visualization of these textures is available in the Textures menu (figure 14.9). The textures

are presented in the following list. All the textures are used for the direct radiance mapping method and are not applied as color textures anywhere in the scene.

- "Position" is a texture providing the first intersection points seen from the light source.
- "Direct radiance" gives the direct illumination stored in the direct radiance map.
- "Normals" is a texture providing the normals at each intersection point seen from the light source.
- "Sub-surface positions" corresponds to the Position texture except it pictures intersection points found inside translucent objects.
- "Sub-surface radiance" finds the radiance reaching the points given in the Sub-surface positions texture after attenuation while traveling through the translucent object.
- "Object centers" provides the object centers according to which the subsurface scattering is sampled. This texture is not absolutely necessary since the object centers can be provided to the direct radiance mapping method as uniform variables.

This concludes the presentation of our graphical user interface (GUI). We encourage use of the application itself which is available on the CD-ROM accompanying this report, see appendix A. In the following chapter we will describe the structure of the implementation behind the GUI.

Chapter 15

Implementation

Progress lies not in enhancing what is, but in advancing toward what will be.

Kahlil Gibran (1883-1931): *A Handful of Sand on the Shore*

Much implementation has been done during this project, in this chapter we present the design of our final application, which is accessed through the GUI described in the previous chapter. Implementation has been done using the following freely available tools:

- **OpenGL** open source graphics library (including extensions)
- **BCC55** Borland C++ commandline compiler version 5.5
- **makegen** for creating makefiles
- **expat** for XML parsing
- **GLUT** OpenGL Utility Toolkit
- **CG** C for Graphics
- **Python** for export scripting

The purpose of this chapter is to provide the reader with an overview of the program parts. Thereby we hope to reveal all the aspects of the application. After giving an overview the chapter will present each program part in a more detailed manner. The actual code will not be presented in this chapter but is available on the attached CD-ROM, see appendix A.

Section 15.1 describes the overall program structure. After that the sections of this chapter are divided into different kinds of menu options. Section 15.2 describes status options, section 15.3 describes render options, and section 15.4 describes scene options. Finally section 15.5 gives a design diagram of the render engine, which is used for several different menu options.

15.1 Program Structure

Through the JR Viewer we give access to the different rendering methods that have been implemented. The source code for the program is divided into different libraries each treating its own part of the project. Each source file starting with a capital letter implements an object, otherwise it implements a tool for the objects or the application in general. The main file (which is called `main.cpp`) contains the WinMain function for the JR Viewer. The structure of the program is quite similar to the structure of the source files and libraries, which are divided mainly according to the menus in the GUI. Appendix C shows the structure of the source files and the libraries for the program. The structure of the program is presened in figure 15.1.

There are three core modules in for the application. The WinMain function placed in the main file (`main.cpp`), which controls the GUI, and thereby also the application. The render engine (placed in the engine library, see app. C) where all the geometry is contained and where the most basic rendering

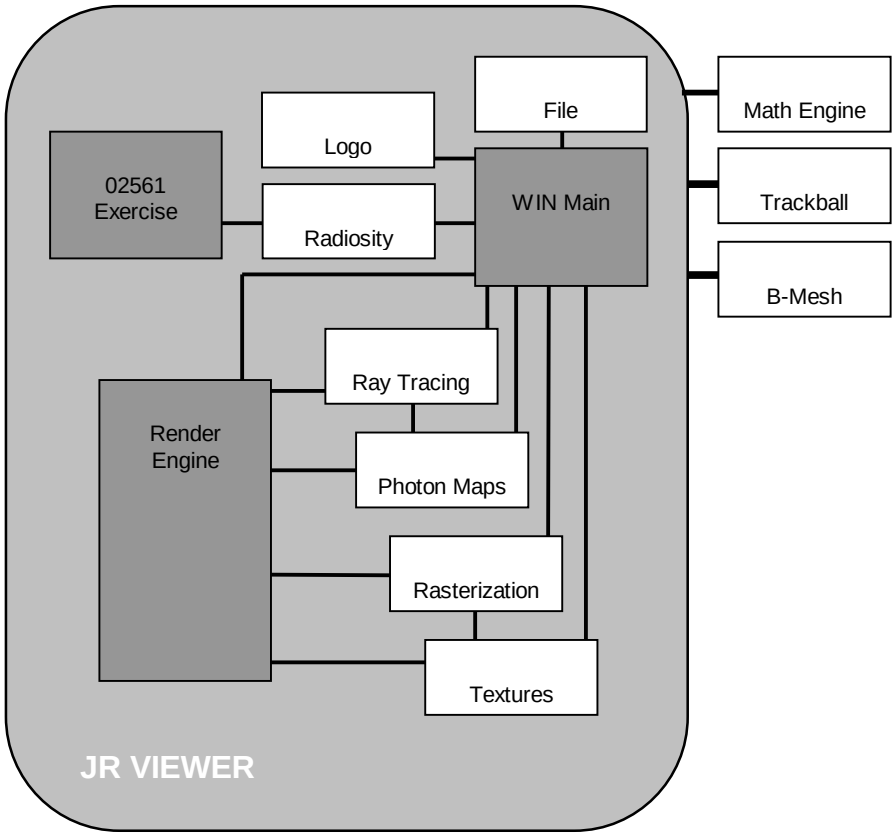


Figure 15.1: The program structure. The names in the boxes refer partly to the menu options, partly to the libraries and partly to controlling files.

is done. 02561 Exercise (from the radiosity library) that takes care of the radiosity rendering.

The rest of the boxes inside the JR Viewer container in figure 15.1 each refer to a menu of the application except for the logo box which is the logo used as the starting picture. The boxes outside the JR Viewer container are general tools that can be used by all of the files in the JR Viewer.

The JR Viewer is a MS Windows application. Windows is a multitasking operating system; this means that multiple applications or programs can run at the same time in what is called threads [47]. A great tutorial for getting started with windows programming can be found at [83].

To enable actions, messages are sent from windows to the application, it is then up to the application to respond to these messages. The function handling messages is the callback function. The callback function is merely a long list of cases with a case for each relevant message (irrelevant messages are simply ignored).

There are a number of callback functions in the application. The most

important is perhaps the one handling the menu options available in the application. Menu options are gathered under the `WM_COMMAND` message. If such a message is sent to the application callback function, it will go through the different menu option ID's collected in another case list. The IDs for each menu option is generated in the header file `resource.h` and linked to the menu in the resource file `menu.rc`.

Most of the menu options simply set different parameters or flags to let the render engine know which effects to include in the rendering of a scene. We will refer to such menu options as *status options*. The flags for the render engine are initiated in `menuflags.h`. Besides there are the menu options that actually switch the current rendering method. These menu options activates rendering according to the flags selected. Finally we have the menu options loading new scenes into the application. Whenever an old scene is left all settings are reset, meaning that all menus are deselected and all status options are reset. Each time a new scene is loaded we will have to gray or un-gray (disable or enable) the menus that we want to be available.

To give an idea of the data flow in the program we will in the following give an example of each type of menu option. Note that radiosity falls a bit out of these categories since it uses its own render engine from the DTU "Computer Graphics" course (02561) as previously mentioned.

15.2 Status Options

Take the "Phong shading" effect in the Ray Tracing menu as an example of a status option. We will now describe what happens step by step when the "Phong shading" menu option is selected. Figure 15.2 shows a diagram of the events.

When "Phong shading" is selected from the ray tracing menu a `WM_COMMAND` is sent from Windows to the application.

1. The `WM_COMMAND` has an associated ID of the Phong shading menu option called `ID_RAY_PHONG`. This is received by the callback function.

2. In the case of `ID_RAY_PHONG` we check if Phong shading is already selected. If this is not the case we select it and deselect all the other shading methods available for ray tracing.

3. We then make sure that the shadow algorithms are not grayed since it is possible to render shadows in combination with the Phong shading algorithm. This would not be possible without a shading algorithm since no direct light is then simulated.

4. Last but foremost the flag `BRDF_PHONG` is set telling the render engine that it must now ray trace the scene using our Phong implementation.

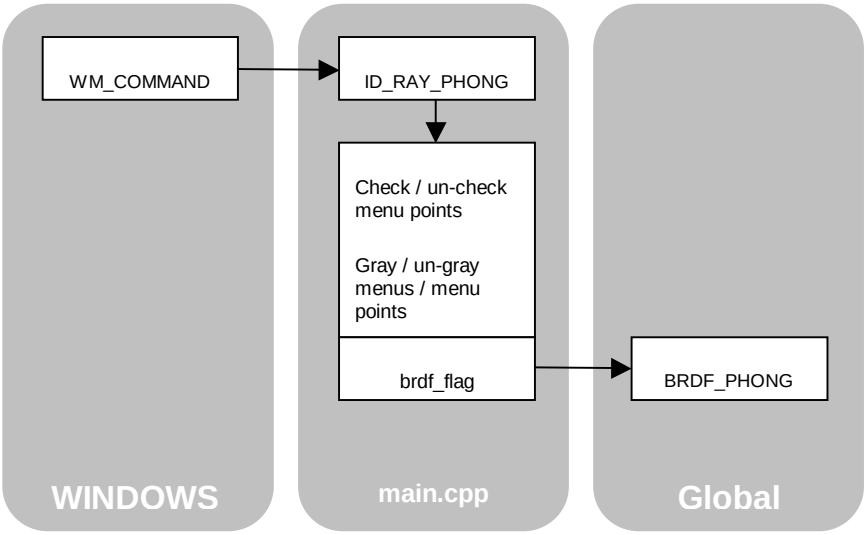


Figure 15.2: The diagram for menu options that toggles render features.

This is in principle what happens when a status option is selected in the JR Viewer. Table 15.1 names all the menu options in the JR Viewer that classifies as status options.

15.3 Render Options

The status options are used by the render engine. The render engine is triggered when we choose a menu option that demands rendering. We refer to these options as *render options*. Table 15.2 shows menu options that falls into the category of render options.

Again we could mention the radiosity menu options in this list, but they are left out since they use their own isolated render engine.

We can continue the example from before by going through the process of pressing "Go..." in the ray tracing menu. When "Go..." is pressed the rendering starts. The current rendering method is determined by a variable called `menu`. `menu` is set to a value corresponding to the menu where a render option was selected - in this case it corresponds to the Ray Tracing menu. `menu` is an index for an array of display functions. Each menu has its own display function (for rendering the scene using the render engine) and its own reshape function (for reshaping the window).

In the case of ray tracing we use the `menu` value to call first the `reshape_raytracing` function, which prepares and initializes the ray tracing, then the `display_raytracing` function, which does the actual ray tracing of the scene using the render engine. The reshape and display functions for ray tracing are declared in `raytracing.h` and implemented in `raytracing.cpp`

Ray Tracing	→Ambient Light
Ray Tracing	→Specular Recursions
Ray Tracing	→Caustics
Ray Tracing	→Hardware optimized
Ray Tracing	→Phong shading
Ray Tracing	→Blinn-Phong shading
Ray Tracing	→Modified Blinn-Phong shading
Ray Tracing	→Hard shadows
Ray Tracing	→Soft shadows
Ray Tracing	→Indirect illumination
Ray Tracing	→Smoothed indirect illumination
Rasterization	→Blinn-Phong shading
Rasterization	→Direct lighting
Rasterization	→Sub-surface scattering
Rasterization	→Planar reflections
Rasterization	→Curved reflections
Rasterization	→Full reflections
Rasterization	→Refractions
Rasterization	→Indirect light - single pass
Rasterization	→Hard shadows
Rasterization	→Soft shadows
Photon Maps	→Direct light
Photon Maps	→Caustics
Photon Maps	→Indirect light
Photon Maps	→Shadows
Photon Maps	→Smoothed irradiance estimates

Table 15.1: A list of all the status options in the JR Viewer, that is, all the menu options that can be selected or deselected with influence on the rendering method.

Ray tracing	→Go...
Rasterization	→OpenGL...
Photon Maps	→Go...
Textures	→Position
Textures	→Direct radiance
Textures	→Normals
Textures	→Sub-surface positions
Textures	→Sub-surface radiance
Textures	→Object centers

Table 15.2: Menu options that triggers the rendering engine.

(in the raytracing library). The render engine makes use of the flags set by the status options, we will return to this in a little while. `reshape_raytracing` is called once when "Go..." is pressed, and it is only called subsequently if the window is reshaped. `display_raytracing` is called in the main loop whenever the application has nothing else to deal with.

The process is described in the following sequence and illustrated in figure 15.3. In other words the following is what happens when `WM_COMMAND` is called with `ID_RAY_GO`.

1. The `WM_COMMAND` has an associated ID of the "Go..." menu option called `ID_RAY_GO`. This is received by the callback function.

2. A message is sent to the status bar telling that ray tracing is now the current rendering method.

3. A shade option is calculated using the status flags. It is used for material calculation purposes in the render engine.

4. The `menu` parameter is set to a value corresponding to the Ray Tracing menu.

5. Initialize OpenGL for rendering. This is always done when a render option is selected to make sure that we start from scratch.

6. `reshape_raytracing` is called for initialization of the ray tracing.

7. Since the `menu` variable has been altered the main loop will now call the `display_raytracing` whenever the application is idle. Therefore the `display_raytracing` function carries out the ray tracing of the scene using the render engine.

This example is for ray tracing, but the process is nearly the same no matter what rendering method we choose. In the case of radiosity though all processing is carried out in the separate renderer for radiosity.

15.4 Scene Options

To make all this useful we need something to render. We load scenes through the file menu. When a file is loaded all previous menu settings are reset. New scenes are always rendered using the OpenGL menu option under rasterization as this is available for all scenes. Table 15.3 contains all menu option in this category. We call those menu options *scene options*.

As an example of loading a scene we look closer at the Cornell box containing two spheres. The ID for this scene is `ID_FILE_TWOSPHERECB`, when this case entry is triggered the old world will be erased and the function

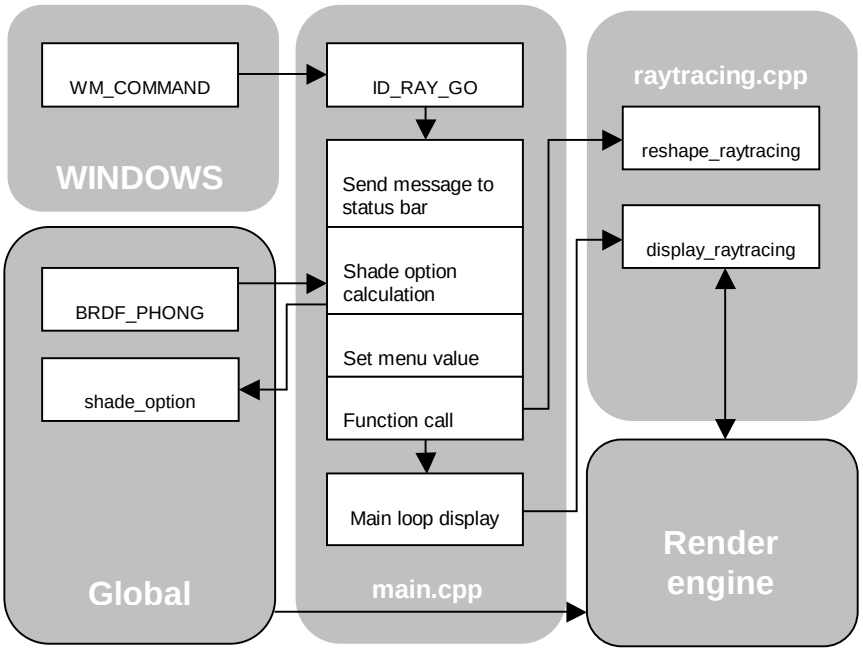


Figure 15.3: Rendering using the JR Viewer.

File→Cornell box - Empty
File→Cornell box - Two boxes
File→Cornell box - Two spheres
File→Cornell box - Sphere and box
File→Cornell box - Orb
File→Cornell box - Arthur
File→Cave scene
File→Load scene...

Table 15.3: The scene options.

`cornell_2spheres` will load the new scene into a new world. `cornell_2spheres` is declared in `file.h` and implemented in `file.cpp` (contained in the file library). This function loads a Cornell box containing two spheres into the application. After this the network of angular visibility between the objects (cf. sec. 11.1) is initialized. Then different relevant menus and menu options are un-grayed as all menus are grayed and all settings are reset when a new scene is to be loaded. Finally we send a message to the status bar with the name of the scene and a boolean variable is set which indicates whether the loading of the scene was successful or not. If this flag is set false the start up logo will be loaded instead of the chosen scene.

The following list describes the procedure step by step and figure 15.4 shows a diagram of what happens.

1. The `WM_COMMAND` has an associated ID of the Cornell box - two spheres menu option called `ID_FILE_TWOSPHERECB`. This is received by the callback function.
2. The function `cornell_2spheres` is called for loading of a scene containing two spheres in a Cornell box.
3. Angular visibility network is initialized in the World object of the render engine.
4. All menus except radiosity, which is not available for this scene, are activated.
5. As the scene contain a refractive object caustic menu options are enabled in the ray tracing and photon maps menus. Scene parameter dialog menu option is un-grayed as well, since a scene is now available.
6. The scene name is sent to the status bar.
7. It is reported whether the loading was successful or not.
8. If the load was successful OpenGL rendering is initialized and the `menu` variable is set to the rasterization menu.

15.5 The Render Engine

The above text and diagrams should explain the different aspects of the implemented application. One important step is still missing though; the rendering process carried out by the render engine. A design diagram of the render engine is presented in figure 15.5

When a new scene is loaded as described above, a new World object holding scene geometry, material, and light sources is created for the render

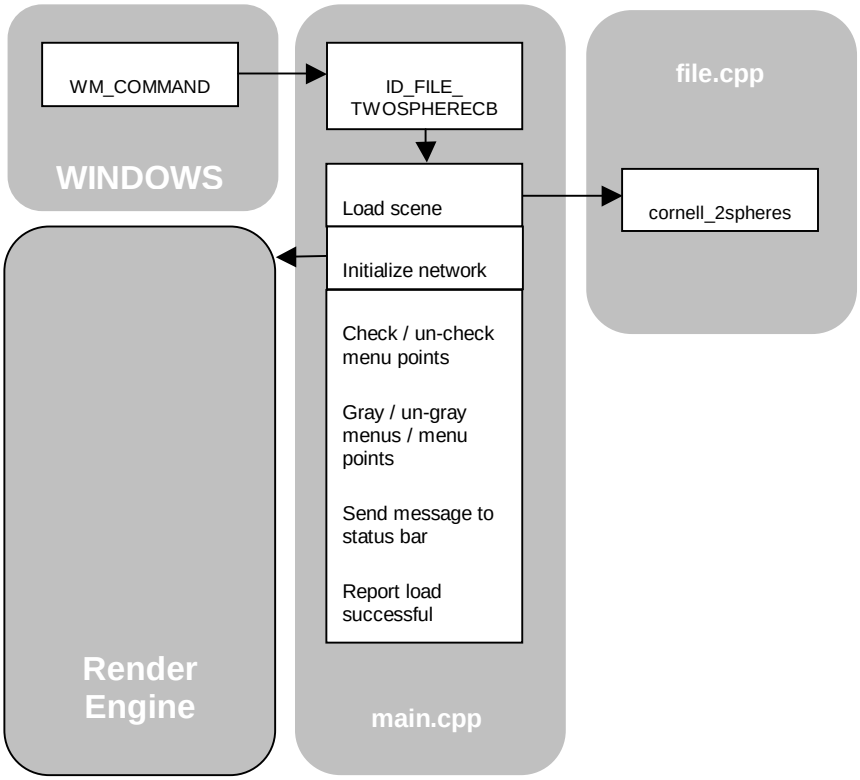


Figure 15.4: The process of loading a scene.

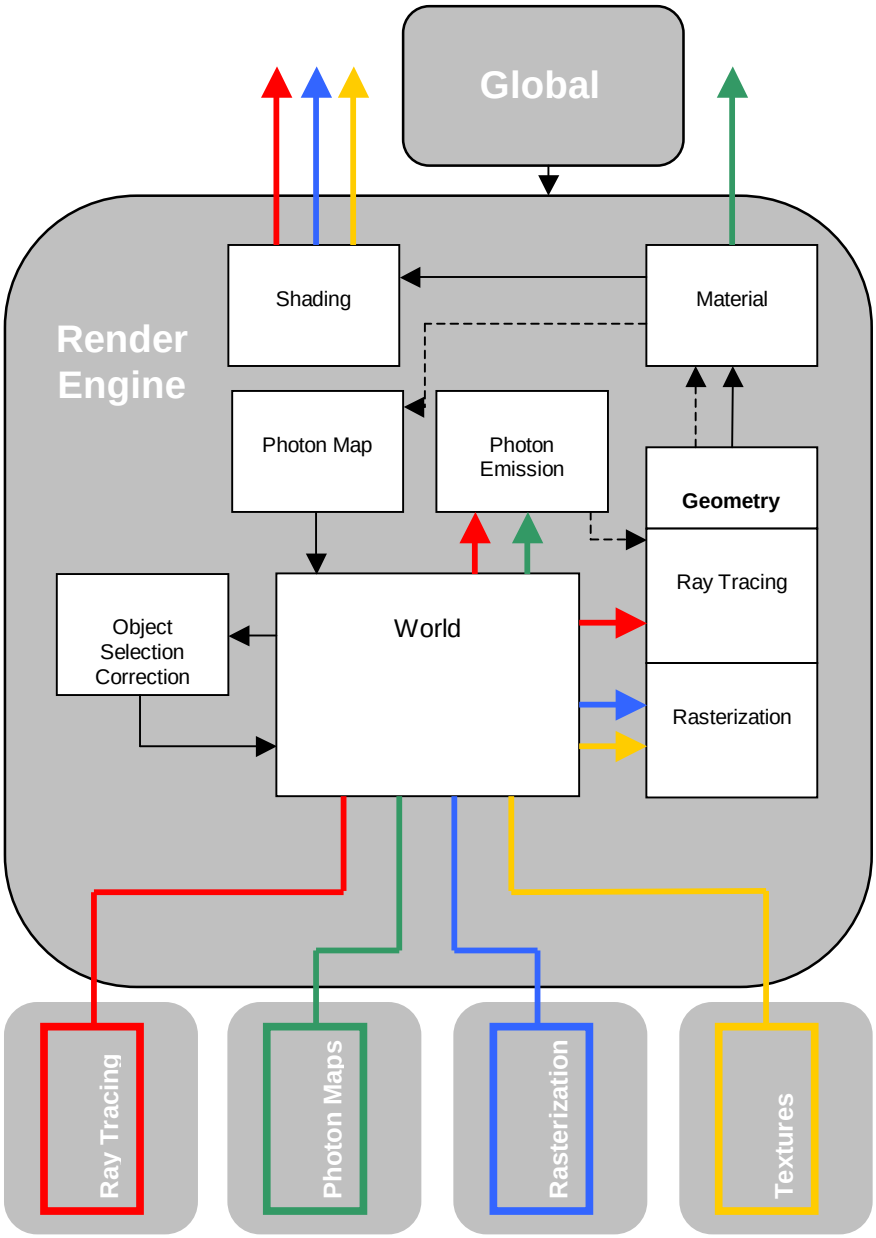


Figure 15.5: A design diagram for the render engine.

engine. Depending on the rendering method chosen by a render option, the render engine will be invoked differently. As indicated in figure 15.5 ray tracing will trace single rays through the scene by intersection calculations, while rasterization will rasterize the scene geometry once. The shading box in figure 15.5 should be regarded differently depending on the rendering method.

Ray tracing will shade each ray independently by use of the render engine. To speed up the ray tracing process by avoiding time consuming if-sentences in the main loop of the ray tracing algorithm, we have chosen to create functions for each combination of selected status options. The different functions are assembled in an array and the function to use is by `shade_option`, which as previously mentioned was calculated using the different flags set in the ray tracing menu. The array of different shading functions is implemented in `Material.h` and in `Material.cpp`.

Rasterization will rather use the render engine to obtain geometry, material, and light settings before rasterizing the scene using vertex and fragment programs. The shading box could therefore be regarded as a fragment program in the rasterization case. Textures are created using the render engine in the same way as when rasterization is done. Where we in the rasterization approach use the textures for later passes, we display them directly when the render options in the Textures menu have been chosen.

Photon mapping is coupled to the ray tracing procedure as indicated by the stippled route through the diagram in figure 15.5. If we store the photons, which are stored in a photon map, in an OpenGL display list as well while emitting photons. We can visualize the photon maps as indicated by the green arrows in the diagram.

This chapter has described the superficial flow of data in our program. For exact implementation details we refer to the source code on the attached CD-ROM (see appendix A). This concludes the description of our experiments, ideas, and implementations. What remains are some conclusive words given in part IV.

Part IV

Conclusions

Chapter 16

Discussion

. . . as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns - the ones we don't know we don't know.

Donald Rumsfeld, US Secretary of Defence (2003)

One purpose of this thesis was to move realistic image synthesis closer to real-time computer graphics. After thorough studies of illumination methods and how to replicate the interaction of light with matter we set out to generate our own methods for simulation of global illumination effects in real-time. A number of ideas appeared and one was chosen as the most promising. This idea, referred to as direct radiance mapping, has been discussed in detail in chapter 12.

Another objective was to produce the entire work flow from the creation of scene meshes to the illumination of them in real-time, using a dynamic application. This objective has been obliged through revision of Blender and the building of a demonstration application (the JR Viewer). The functionalities of Blender were described in chapter 10 and the demonstration application was presented in chapter 14.

In this chapter we will concentrate on the parts that yet remains to be discussed in this report. Section 16.1 will address future aspects of the project and what we would have done if more time had been available. The applicabilities of the project ideas, results, and experiments will be discussed in section 16.2.

16.1 Future experiments

The idea of direct radiance mapping came to us at a fairly late time during the project, since it was the outcome of in-depth theoretical studies of global illumination and the practical angle gained through implementation of different illumination methods. The same studies and implementations formed a number of other ideas as well presented in chapter 11. It could be interesting to pursue some of the ideas a little further. The more conceptual ideas such as the topological network described in section 11.2 and the multi-agent approach described in section 11.4 are the ideas that we find most interesting for future development. These two approaches could be combined and perhaps they may even be useful in a combination with direct radiance mapping.

A modest idea such as angular visibility, described in section 11.1, can in fact find many small applicabilities here and there. Recall, for example, that we assume in our direct illumination calculation that light sources are very distant from the scene. This means that the angle used for the cosine term in the radiance calculation is found according to the direction towards the center of the light source. The result is that the side of the large box in the Cornell box (see fig. 12.2) is entirely black though the area light source would in fact illuminate it from a part of its area. The angular visibility could be used to find the size of this area and thereby a better estimate of the direct illumination resulting from an isotropic area light approximated by a point light would be available.

Direct radiance mapping in itself still contains unexplored territories. There are many interesting expansions and experiments that could be worked with. The method still has many flaws and ideas for removing some of these are presented in chapter 12. To better compare the concept of direct radiance mapping with ordinary global illumination methods and to generate a proof of concept implementation, it could also be interesting to implement the conceptual method (sec. 12.1) in software with an actual network for light propagation instead of the instruction limited implementation that we have now on the GPU.

In the nearest future a trimming of the method would be appropriate removing some of the most significant drawbacks like flickering and overexposure of indirect illumination and missing indirect shadows, some proposals for their solutions were given in chapter 12.

New GPUs, already available at the market, have functionalities which our method can benefit from significantly. Important functionalities of 6th generation GPUs are the feature called render to multiple targets, which would make the many passes for rendering low-resolution texture a single pass or two only, and true looping which can significantly improve on the processing time consumed by our fragment programs for indirect illumination. Render-to-texture is another improvement that we even could have tried on the hardware we currently have available. Unfortunately we did not find time for this. Wishful thinking from our side is to hope for new GPUs with a mipmapping instruction available in fragment programs.

The demonstration application of the project also has its limitations. As the basic application is merely made to demonstrate our project accomplishments and to show that we are able to use scenes that we have modeled externally in a modeling tool, it is important to make the export and import scripts more robust and cooperative. This is crucial for using our implementations in other connections. Therefore a future assignment is to improve on those.

A number of global illumination algorithms have been implemented for comparison purposes. These implementations have also been important in our learning process and therefore more such implementations are welcome in the future.

For comparison purposes as well, it could be enlightening to implement some of the competitive methods described in chapter 6. As mentioned in section 12.4 some of the ideas used in these methods might also be combined with direct radiance mapping. Implementing them would therefore be an opportunity to study the method in details and find out how combinations could be carried out best.

In chapter 12 we mention and idea of an environment map for each object in the scene. We find that this idea, though dependent on the number of objects in the scene, could be very interesting to pursue. It seems as a good choice for calculation of multiple bounces in the estimate of diffusely reflected

indirect illumination.

Looking at the incorrect bright spots in figure 12.6, we may shortly bring up an old-timer - Lambert after whom Lambertian surfaces are named. In his *Photometria* (1760) Lambert presented the "five-times" rule of thumb [2, p. 187]:

> If the distance from the light source is five times or more that of the light's width, then the inverse square law is a reasonable approximation and so can be used. Closer than this and the solid angle the light covers will vary noticeably different from an inverse square relationship.

Thinking of the sample points in our scene as light sources of a size equal to eg. A_M/N, where N is the number of sample points, we may be able to avoid the bright spots by Lambert's "five-times" rule of thumb.

Though mipmaps are discussed here and there in the preceding chapters, we think that many opportunities yet lie in mipmapping combined with direct radiance mapping. An idea is to use a mipmap for the light source images. Then we should use the level in the mipmap that corresponds best to the number of sample points that we can afford. This might make direct radiance mapping much more robust. Another thing is that we could combine shadow maps with this idea. Then the shadow map can use a high resolution texture while DRM uses a low resolution texture at a higher level in the mipmap. For this to work, it is necessary that we must be able to choose specifically in the fragment program, which level in the mipmap that we want to use. This functionality is fortunately available on modern graphics hardware.

With respect to modeling it would be nice to examine the Blender Render in-depth. Then we could compare Blender renders to our own renderings. Also we could compare our real-time results to the real-time results available through Blender's game engine.

As a development of our work flow concept we would also like to develop the JR Viewer further by adding different features continually. Examples of features are: Connection of several scenes, animations and dynamic light effects, and maybe even a physics engine. By adding such features we would gradually come closer to a truly dynamic application and eventually we would have all the tools needed for a free of charge game development platform.

This section holds many ideas for future experiments and more could probably be found. For example, we could work on some of the subjects that have largely been neglected in this project; Scene graphs, spatial data structures, anti-aliasing, real-time soft shadows, etc. This concludes our discussion on future work. In the following section we will discuss the applicabilities of this project.

16.2 Applicability

Direct radiance mapping in its current form is not sufficient for use in commercial applications it is still on an experimental stage. To make it suitable for commercial applications there are many problems that must be dealt with. First of all the frame rate must be higher and the flickering and overexposure of indirect illumination must be corrected. With more time invested in improvements of the method and the frequently increasing speed of graphics hardware we believe that there is hope for direct radiance mapping. Still the method very much depends on hardware and since it may take a while before GPUs become fast enough for the method, it will take even longer before the average consumer will have such hardware available. One thing speaking in favor of direct radiance mapping, in this respect, is that it can be applied at different levels of complexity (the sample points can be adjusted or spread over several frames).

One of the goals of this project was to create a platform for developing real-time dynamic applications such as computer games. During the project we have managed to establish the beginning of a free of charge game development platform. This is done using various sources including our own software. All tools that have been used, from compiler to modeling program, are free of charge and available on the internet. Although we have not created a complete game we have taken the first small step by generating an application able to present our own modeled scenes using our own render engine. Of course there is a long way to a finished product, but we believe to have provided a proof of concept showing that it is possible to get far with little expenses on software.

The application that we have built is used mainly for presentation purposes and for comparing different rendering methods. As it is possible to load arbitrary scenes in X3D format others can use the application for demonstration purposes as well. The application can of course be used by others to compare their rendering methods with the traditional methods as well as ours.

In the final chapter we will sum up on the report and give our final remarks.

Chapter 17

Conclusion

We have a habit in writing articles published in scientific journals to make the work as finished as possible, to cover up all the tracks, to not worry about blind alleys or describe how you had the wrong idea at first, and so on. So there isn't any place to publish, in a dignified manner, what you actually did in order to get to do the work.

Richard Buckminster Fuller (1895-1983)

This thesis encompasses two main objectives. First, we wanted to explore how close we could bring realistic image synthesis to real-time execution rates, and secondly, we wanted to make a platform for development of real-time applications.

The report is divided into four parts. The first part mainly serves the first objective, the second part mainly serves the second objective, and part III serves both objectives by a presentation of the ideas, results, and experiments that we have had during this project. Part IV (this part) discusses and concludes the report.

The first part builds the theoretic foundation for the rest of the project. The chapters of this part have served several purposes. Chapter 2, for example, presents an array-based math engine which can be used for an efficient implementation of vector and matrix math. Chapter 3 is important in order to understand the fundamental illumination model behind the global illumination that we would like to simulate in real-time. Having a good understanding of the global illumination model enable us to discover the abilities and limitations of a method before taking it too far.

The theoretical angle to a subject is rarely enough to give a full understanding. Therefore we introduced different traditional methods for both realistic image synthesis (chapter 4) and real-time rendering (chapter 5). Which parts of these methods that we have implemented are described in chapter 13.

During the implementation of these methods we came up with several ideas, some for improvement of specific methods, others for more conceptual approaches to rendering of global illumination. The ideas are described in chapter 11 and listed below.

1. Angular visibility between axis aligned bounding boxes (AABBs) for fewer intersection tests in ray tracing.
2. A topological network for radiance transfer between objects.
3. Displacement mapping for fast ray/object intersection.
4. A multi-agent approach to global illumination, where each object is an autonomous agent controlling its own shade.
5. An 'atmosphere' to limit the influence of each object in global illumination.
6. A line-of-sight algorithm for fewer intersection tests in ray tracing.
7. Gouraud interpolation of the first intersection point. A rasterization approach to the first level of the ray tracing algorithm.
8. Single pixel images each representing a ray. An attempt to let rasterization do full recursive ray tracing.

A combination of the different traditional methods we had implemented and the ideas in chapter 11, ended up in a specific idea called direct radiance mapping, which found its inspiration in the idea of angular visibility and made extensive use of the idea concerning Gouraud interpolation of the first intersection point.

Chapter 12 basically describes direct radiance mapping. First the conceptual idea is described (in sec. 12.1) to give an insight in the thoughts that led to the idea. The conceptual formulation of the idea may not be practical for implementation purposes, but it gave us many ideas for solutions to the problems and limitations that we found in the method.

The direct radiance mapping method is described in section 12.2. Building on the theory of part I, a practical method for implementation of direct radiance mapping is described mathematically step by step. Direct radiance mapping was originally an idea for simulation of a single bounce of diffusely reflected indirect illumination.

To have possibilities of both expansions of direct radiance mapping and comparison of direct radiance mapping to other methods simulating the same part of the global illumination model, a number of other methods for real-time global illumination were explored. These are described in chapter 6. The study of methods presented by others is, of course, important to the project both for ideas and for comparison purposes.

A number of extensions to direct radiance mapping have been done. Those include use of stenciled planar reflections and environment mapping for specular reflections, and use of stenciled shadow volumes for hard shadows. We also propose a simple version of subsurface scattering as an extension to direct radiance mapping. We can therefore conclude that direct radiance mapping is rich in possibilities for expansions, which is important when we think of the many current problems and limitations. Limitations such as lack of multiple bounces and missing indirect shadows are worst since they are not solved in the original method. Lesser problems have also been discovered but we can explain them through our mathematical description and give possible solutions for them.

The general comparison between direct radiance mapping and other methods for global illumination effects in real-time revealed some shortcomings of the method. However, it also revealed several advantages particularly in connection with freedom of the scene mesh, compared to some of the other methods. A scheme holding the different methods up against each other can be found in chapter 12, table 12.1. Although there are still features missing we have suggested solutions to most problems mentioned, and as the method must be classified as in an early state of development we still find hope for it. All options are still open.

Part II takes a more practical angle addressing the second objective of the project, which was to create a platform for development of real-time applications. The chapters of the part address different modeling issues.

The last chapter (10) introduces Blender; a free of charge modeling application. Through Blender we can generate models for later use in a dynamic application. This should be seen as an initializing step towards creation of commercial dynamic applications such as computer or console games.

Later in part III we describe the JR Viewer, our demonstration application, which can be seen as a beginning implementation of a dynamic application. This is justified since we are able to render a scene containing our own modeled scenes using our own implementations in a separate application. The JR Viewer even allow us to move objects about in a simple manner using the track ball and picking functionalities. The track ball was developed to enable interactive movement of objects in section 9.3.

The fact that the work flow has been created alone from freeware tools can be seen as a proof of concept, that it is possible to get started as a developer with little or no financial means.

From the discussion chapter and the chapter concerning direct radiance mapping we can conclude that much work is still to be done. Our method is in an early stage of development and several ideas for improvements are suggested throughout the report, just waiting to be implemented and tested.

Direct radiance mapping depends very much on hardware. The development in this field is currently rapid and pointing in directions beneficial to our method. We believe, without doubt, that already the next generation of GPUs will enable features that can improve our method significantly in terms of frame rate and sample points. This is not only build on intuition but also on promised features like more instructions possible and better looping in fragment programs as well as render to multiple targets, which would all be very useful for direct radiance mapping.

We believe that global illumination continuously will move towards a real-time implementation with little or no limitations. Whether or not direct radiance mapping will be useful in this process, only time can show.

Appendix A

Contents of Attached CD-ROM

The compact disc accompanying this report contains all the tools used for the project. These are also available on the internet in the newest versions (Blender as an example has been updated twice during this project alone). The CD also contains many of the articles and other writings presented in the reference list. Most writings are in .pdf format and named after the title of the paper. Finally our application and all the program code are included. The JR Viewer is made executable from the root of the CD. To execute run the file `JRView.bat` or `program JRView.exe`.

The following list presents the contents on the CD.

Library	Contents
..\. (root)	A script running an executable version of the JR Viewer demonstration application (`JRView.bat`)
..\lit	All literature available in digital form that is referred to in the report. The literature is sorted by author in subdirectories.
..\lit\TUTMAN	Tutorials and manuals for different tools and programming languages used in the project.
..\modeling	Contains the installation files for the different Blender versions available for us through the project. The descriptions in part II fits version 2.33a. Newest version is always available on www.blender.org.
..\modeling\export	The export script used for exporting files from Blender in X3D format.
..\modeling\manual	The Blender manual from the web site www.blender.org in pdf format.
..\modeling\scenes	Different scenes created during the project including those used in the JR Viewer. All in the blend format used by Blender.
..\program	Contains the code files. The structure is presented in appendix C.
..\report	The report in pdf and ps format as report01.pdf and report01.ps.
..\tools\reader	Installation file for Acrobat Reader
..\tools\CG	Installation file for the CG library
..\tools\compiler	Installation file for the Borland C++ Command Line Compiler 5.5 compiler used in this project and the `makegen` application used to generate make files.
..\tools\latex	A tutorial for LaTeX in the file tnssitl.pdf
..\tools\OpenGL	dll files needed to run OpenGL.

Appendix B

Historical Remarks

The nature of light has been studied and investigated ever since man began to philosophize.

Presumably the Pythagoreans (∼582–500 BC.) of ancient Greece put forth the "particle" theory of light [138] in which it is hypothesized that the eyes send rays to objects and that these rays give us information about the object's shape and color [96].

Though we know, today, that light is emitted from light sources, not the eyes, the mathematical model described by the ancient philosophers is basically the same as when we trace rays from the eyes in backward ray tracing.

The first to theorize that the medium of sight (light) is from the object and is received by the visual perception organ (eye) is Empedocles (∼480 BC.) [96]. However, many years should pass before this theory was generally accepted.

Euclid's Optics (∼300 BC.) is the oldest surviving work on geometrical optics (or ray optics) [32]. He notes that *light travels in straight lines*. Furthermore the principles of *perspective* are established and the *law of reflection* is given [33], presented below in its modern enunciation [116]:

> The reflected ray lies in the plane of incidence;
> the angle of reflection equals the angle of incidence.

Approximately one and a half centuries later Hero (or Heron) of Alexandria (∼150 BC.) showed geometrically that the actual path taken by a ray of light reflected from a plane mirror is shorter than any other reflected path that might be drawn between the source and the point of observation [4]. The principle of the path of minimum distance is later known as *Hero's Principle*. This result leads towards Fermat's Principle that should later become one of the four fundamental postulates of ray optics [116].

Refraction when light passes from air into water or glass was observed and studied by Cleomedes (50 AD.) and Ptolemy (130 AD.). In the work of

the latter it is suggested that the angle of refraction is proportional to the angle of incidence [4]. Later this shall result in Snell's law.

Through logic and experimentation the Arabian scientist and scholar Ibn al-Haitham[1] known to the Europeans as Alhazen (965–1039 AD.) finally discounted the theory that vision issues forth from the eye. He used a forerunner of the camera obscura (a pinhole camera) as a model for the eye and investigated both spherical and parabolic mirrors. He was aware of *spherical aberration* and also investigated the *magnification* produced by lenses and *atmospheric refraction*. His work influenced all later investigations on light [138, 4, 82].

Quite a few centuries should now pass before the next break through in the theory of light would emerge. Optics had again become an exciting area of research with the invention of telescopes and microscopes in the early seventeenth century. In 1611 Johannes Kepler discovered total internal reflection and described the small angle approximation to the law of refraction [30]. A decade later, in 1621, Willebrord Snell discovered the relationship between the angle of incidence and the angle of refraction when light passes from one transparent medium to another [4]. This relationship is called *Snell's law*, which led to the *law of refraction*. The same law was, in 1637, independently discovered and formulated in terms of sines by René Descartes. It was published in his Discourse on Method containing a scientific treatise on optics [26]. In its modern form the law of refraction is given as [116]:

> The refracted ray lies in the plane of incidence; the angle of refraction θ_2 is related to the angle of incidence θ_1 by Snell's law,
>
> $$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

Pierre de Fermat's Principle of "least time", according to which a ray of light follows the path which takes it to its destination in shortest time, is enunciated in 1657 [4]. From this principle both the laws of reflection and refraction can be deduced. Today *Fermat's Principle* has a slightly different presentation [116]:

> Optical rays traveling between two points, A and B, follow a path such that the time of travel (or the optical path length) between the two points is an extremum relative to neighboring paths.

The extremum may be a minimum, a maximum, or a point of inflection. It is, however, usually a minimum, in which case Fermat's original formulation is true. In a homogenous medium the index of refraction (n) is the same

[1] Full name: Abu Ali Mohamed ibn al-Hasan Ign al-Haytham.

everywhere, and so is the speed of light. The path of minimum light is then also the path of minimum distance, which leads to the following formulation of Hero's Principle: In a homogenous medium light travels in straight lines [116]. Decidedly Hero's Principle is a special case of Fermat's Principle.

In 1665, Grimaldi and Hooke conclude upon their observations of *diffraction*, the phenomenon where light "bends" around obstructing objects, that light is a fluid that exhibits wave-like motion. Hooke proposed *a wave theory of light* to explain this behavior [30]. Around the same time Isaac Newton describes his theory of color using a prism to show that white light is a combination of different colors [4] (the phenomenon is called *dispersion*).

Ole Rømer deduces an approximate value of *the speed of light* in 1676. This is done from detailed observations of the eclipses of the moons of Jupiter. From Rømer's data, a value of about $2 \cdot 10^8\ m/s$ is obtainable [4].

A major milestone for a wave theory of light is the "Traite de Lumiere" published by Huygens in 1690. He considered that light is transmitted through an all-pervading aether that is made up of small elastic particles, each of which can act as a secondary wavelet [4]. Huygens also discovered the phenomenon of *polarization* during his experiments [30]. His theory was, though able to explain many of the known propagation characteristics of light, put aside when Newton published his "Opticks" (1704) in which he put forward the view that light is corpuscular (particles traveling in straight lines).

A century should pass before Thomas Young (1801) provided the necessary support for a wave theory of light by demonstration of the *principle of interference* through his double-slit experiment.

In 1816, Augustin Jean Fresnel presents a rigorous treatment of diffraction and interference phenomena showing that they can be explained in terms of a wave theory of light. Later (1821) he presented the laws which enable the intensity and polarization of reflected and refracted light to be calculated, [4, 30]. Modern text books present the Fresnel equations as follows [60]:

$$\begin{aligned} \rho_{\parallel} &= \frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2} \\ \rho_{\perp} &= \frac{\eta_1 \cos\theta_1 - \eta_2 \cos\theta_2}{\eta_1 \cos\theta_1 + \eta_2 \cos\theta_2} \end{aligned}$$

$$F_r(\theta) = \frac{1}{2}(\rho_{\parallel}^2 + \rho_{\perp}^2)$$

Using the wave theory Helmholtz (1821–1894) records the curves for the basic mixing colors: ultramarine blue, green, and red. These three colors and their mixing colors are ascribed to three kinds of color sensitive receptors in the eye. This theory becomes known as the Young-Helmholtz Three Component Theory [96].

Maxwell puts forth his famous theory of the electromagnetic field in 1865. Taking earlier experimental and theoretical work of Coulomb, Poisson, Ørsted, Ampere, Faraday, and others; he completed the theory [76]. From his studies of the equations describing electric and magnetic fields, it was found that the speed of an electromagnetic wave should, within experimental error, be the same as the speed of light. Maxwell concluded that light is a form of electromagnetic wave [4].

Heinrich Hertz validated Maxwells theory experimentally [76] and in 1887 he accidentally discovered the *photoelectric effect*, which is the process whereby electrons are liberated from materials under the action of radiant energy [30]. An effect that can not be described by the wave model.

Nevertheless light can, at this point, be described as the visible part of the spectrum of electromagnetic radiation. Electromagnetic waves range from radio waves, which can have wavelengths of kilometers, to cosmic rays with wavelengths of less than 10^{-10} centimeter. The range of visible wavelengths is approximately $0.4 - 0.7\mu m$, and it is included in the area of electromagnetic waves detected as heat radiation [121]. The research area of thermal radiation heat transfer is therefore of interest to the theory of light.

In 1879 Joseph Stefan published an experimental investigation of the spectral distribution of heat transfer from bodies of different temperatures. His results were a number of curves showing that the emitted power increases with increasing temperature, and that the maximum point of a distribution is found at shorter wavelengths when the temperature increases [11].

A few years later (1884) Ludwig Boltzmann was able to deduce theoretically that the emissive power varies with the temperature power four. Similarly Willy Wien[2] deduced the displacement law a decade later (1896):

$$\lambda_{\max} T = \text{constant}$$

Since heat and light is electromagnetic waves, many attempts to find a connection between the experimental results of Stefan and the theoretical results of Boltzmann and Wien were conducted using Maxwell's equations. However, it was only possible to find functional expressions that could confirm Stefan's experimental results at very long or at very short wavelengths [11]. In 1901 Max Karl Ernst Ludwig Planck provided the missing concept. He proposed the existence of a light quantum, based on this quantum theory he was able to explain the spectrum of radiation emitted from a blackbody [107]. Planck found it necessary to introduce a universal constant described as the quantum of action, now known as Planck's constant ($h \approx 6.63 \cdot 10^{-34} Js$) [4].

Planck's theory remained mystifying until Albert Einstein in 1905 explained the photoelectric effect on the basis that light is quantized, the

[2]Willy was a convenient nickname since his full name was Wilhelm Carl Werner Otto Fritz Franz Wien.

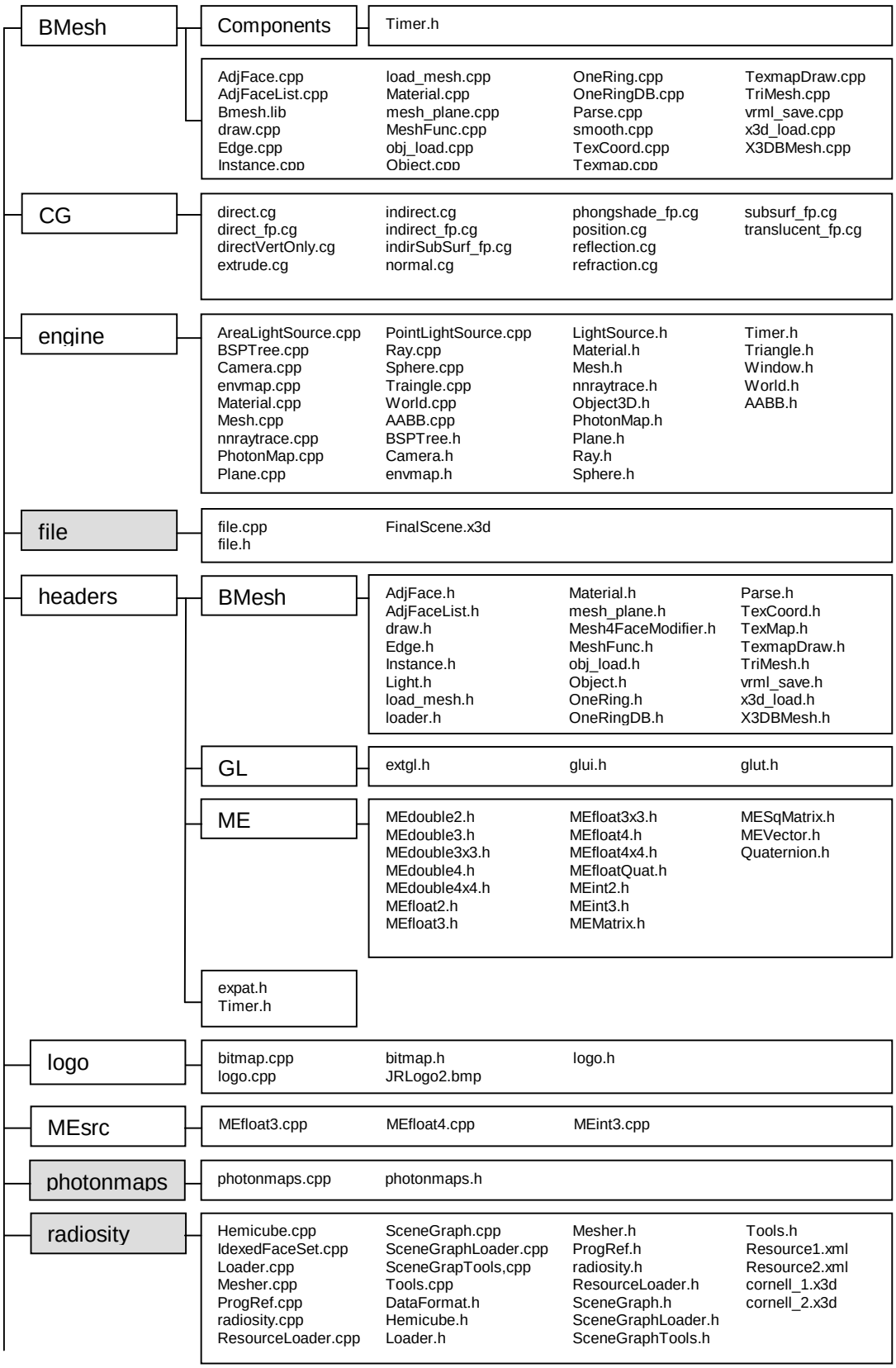
quanta would subsequently become known as photons [4]. During the next two decades scientists recast all of physics to be consistent with Planck's theory [82]. However, the wave model of light was still necessary to describe phenomena such as interference and diffraction - Niels Bohr called this the complementary nature of light [60].

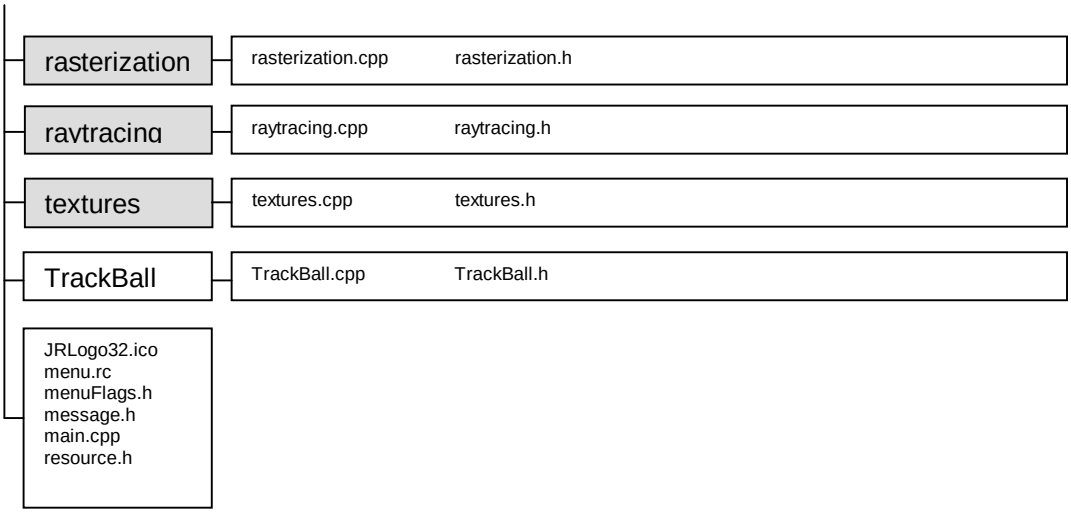
In 1913, Bohr put forth a theory explaining the observation that atoms absorb and emit light at particular frequencies that are characteristic of the atom [4]. His theory was based on *quantum mechanics*. By considering submicroscopic phenomena, researchers such as Bohr, Born, Heisenberg, Schrödinger, Pauli, De Broglie, Dirac, and others, have been able to explain the complementary nature of light [30].

To summarize four major models for the explanation of light have been put forth through times. Those are ray optics, wave optics, electromagnetic optics, and quantum (or photon) optics. Each model has a set of postulates on which all possible results of that model depend. The more advanced models include the simpler ones and therefore the postulates of ray optics for example follows naturally from the postulates of wave optics. A thorough explanation is provided in [116].

Appendix C

Structure of Source Files and Libraries


BMesh
Components
Timer.h
AdjFace.cpp
AdjFaceList.cpp
Bmesh.lib
draw.cpp
Edge.cpp
Instance.cpp
load_mesh.cpp
Material.cpp
mesh_plane.cpp
MeshFunc.cpp
obj_load.cpp
Object.cpp
OneRing.cpp
OneRingDB.cpp
Parse.cpp
smooth.cpp
TexCoord.cpp
Texmap.cpp
TexmapDraw.cpp
TriMesh.cpp
vrml_save.cpp
x3d_load.cpp
X3DBMesh.cpp
CG
direct.cg
direct_fp.cg
directVertOnly.cg
extrude.cg
indirect.cg
indirect_fp.cg
indirSubSurf_fp.cg
normal.cg
phongshade_fp.cg
position.cg
reflection.cg
refraction.cg
subsurf_fp.cg
translucent_fp.cg
engine
AreaLightSource.cpp
BSPTree.cpp
Camera.cpp
envmap.cpp
Material.cpp
Mesh.cpp
nnraytrace.cpp
PhotonMap.cpp
Plane.cpp
PointLightSource.cpp
Ray.cpp
Sphere.cpp
Traingle.cpp
World.cpp
AABB.cpp
BSPTree.h
Camera.h
envmap.h
LightSource.h
Material.h
Mesh.h
nnraytrace.h
Object3D.h
PhotonMap.h
Plane.h
Ray.h
Sphere.h
Timer.h
Triangle.h
Window.h
World.h
AABB.h
file
file.cpp
file.h
FinalScene.x3d
headers
BMesh
AdjFace.h
AdjFaceList.h
draw.h
Edge.h
Instance.h
Light.h
load_mesh.h
loader.h
Material.h
mesh_plane.h
Mesh4FaceModifier.h
MeshFunc.h
obj_load.h
Object.h
OneRing.h
OneRingDB.h
Parse.h
TexCoord.h
TexMap.h
TexmapDraw.h
TriMesh.h
vrml_save.h
x3d_load.h
X3DBMesh.h
GL
extgl.h
glui.h
glut.h
ME
MEdouble2.h
MEdouble3.h
MEdouble3x3.h
MEdouble4.h
MEdouble4x4.h
MEfloat2.h
MEfloat3.h
MEfloat3x3.h
MEfloat4.h
MEfloat4x4.h
MEfloatQuat.h
MEint2.h
MEint3.h
MEMatrix.h
MESqMatrix.h
MEVector.h
Quaternion.h
expat.h
Timer.h
logo
bitmap.cpp
logo.cpp
bitmap.h
JRLogo2.bmp
logo.h
MEsrc
MEfloat3.cpp
MEfloat4.cpp
MEint3.cpp
photonmaps
photonmaps.cpp
photonmaps.h
radiosity
Hemicube.cpp
IdexedFaceSet.cpp
Loader.cpp
Mesher.cpp
ProgRef.cpp
radiosity.cpp
ResourceLoader.cpp
SceneGraph.cpp
SceneGraphLoader.cpp
SceneGrapTools,cpp
Tools.cpp
DataFormat.h
Hemicube.h
Loader.h
Mesher.h
ProgRef.h
radiosity.h
ResourceLoader.h
SceneGraph.h
SceneGraphLoader.h
SceneGraphTools.h
Tools.h
Resource1.xml
Resource2.xml
cornell_1.x3d
cornell_2.x3d


rasterization
rasterization.cpp
rasterization.h
raytracing
raytracing.cpp
raytracing.h
textures
textures.cpp
textures.h
TrackBall
TrackBall.cpp
TrackBall.h
JRLogo32.ico
menu.rc
menuFlags.h
message.h
main.cpp
resource.h

Bibliography

[1] Tomas Akanine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Rendering Techniques '02 (Proc. of the Thirteenth Eurographics Workshop on Rendering)*, pages 297–306, 2002.

[2] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering.* A K Peters, Natick, Massachusetts, second edition, 2002.

[3] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL^{TM}.* Addison-Wesley, third edition, 2003.

[4] A brief history of optics. http://members.aol.com/WSRNet/D1/hist.htm. Accessed 3rd of June 2004.

[5] James Arvo and David B. Kirk. Particle transport and image synthesis. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):63–66, August 1990.

[6] Ian Ashdown. Photometry and radiometry: A tour guide for computer graphics enthusiasts. www.helios32.com/Measuring Light.pdf, October 2002.

[7] Ulf Assarsson. *A Real-Time Soft Shadow Volume Algorithm.* PhD thesis, Chalmers University of Technology, 2003.

[8] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In M Doggett, W. Heidrich, W. Mark, and A. Schilling, editors, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 33–40. Eurographics Association, 2003.

[9] James F. Blinn. Models of light reflections for computer synthesized pictures. *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 192–198, July 1977.

[10] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[11] Erik Both and Gunnar Christensen. *Termodynamik*. Den private Ingeniørfond, third edition, 2002.

[12] Chris Brennan. Shadow volume extrusion using a vertex shader. In Wolfgang F. Engel, editor, *ShaderX: Vertex and Pixel Shader Programming Tips and Tricks*. Wordware, May 2002.

[13] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 51–59. Eurographics Association, 2003.

[14] Jens Michael Carstensen, editor. *Image Analysis, Vision, and Computer Graphics*. IMM, Technical University of Denmark, Lyngby, 2001.

[15] Subrahmanyan Chandrasekhar. *Radiative Transfer*. Dover Publications, Inc., New York, 1960. Unabridged and slightly revised edition of the work first published in 1950.

[16] Keshav Channa. Light mapping - theory and implementation. http://www.flipcode.com/articles/article_lightmapping.shtml, July 2003. Accessed 31st of August 2004.

[17] Per H. Christensen. Adjoints and importance in rendering: An overview. *IEEE Transactions on Visualization and Computer Graphics*, 9(3), 2003.

[18] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):31–40, July 1985.

[19] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, 1993.

[20] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):137–145, July 1984.

[21] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Computer Graphics*, 1(1):7–24, 1982.

[22] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(3):242–248, July 1977.

[23] Haskell Brooks Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland Publishing Company, Amsterdam, 1958.

[24] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, Department of Computer Science, University of Copenhagen, July 1998.

[25] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwartzkopf. *Computational Geometry: Algorithms and Applications.* Springer, Berlin, second edition, 2000.

[26] René Descartes. *Discourse on Method, Optics, Geometry, and Meteorology.* Hackett Publishing Co, Inc, revised edition, May 2001.

[27] Kirill Dmitriev, Stefan Brabec, Karl Myszkowski, and Hans-Peter Seidel. Interactive global illumination using selective photon tracing. In *Rendering Techniques '02 (Proc. of the Thirteenth Eurographics Workshop on Rendering)*, pages 21–34, 2002.

[28] Craig Donner and Henrik Wann Jensen. Faster gpu computations using adaptive refinement. In *Proceedings of SIGGRAPH 2004, Technical Sketches*, August 2004.

[29] Julie Dorsey, Alan Edelman, Henrik Wann Jensen, Justin Legakis, and Hans Køhling Pedersen. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH 1999*, pages 411–420, 1999.

[30] Philip Dutré, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination.* A K Peters, Natick, Massachusetts, 2003.

[31] Jens Eising. *Lineær algebra.* Institut for Matematik, Danmarks Tekniske Universitet, 1997.

[32] Adventures in cybersound: Euclid. http://www.acmi.net.au/AIC/EUCLID_BIO.html. Accessed 14th of May 2004.

[33] Euclid. Optics. http://www.calstatela.edu/faculty/hmendel/Ancient Mathematics/Euclid/Optics/Optics.html. Accessed 14th of May 2004.

[34] Cass Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. NVIDIA White Paper, March 2002. http://developer.nvidia.com.

[35] Kasper Fauerby and Carsten Kjær. Real-time soft shadows in a game engine. Master's thesis, Department of Computer Science, University of Aarhus, December 2003.

[36] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, 2003.

[37] F. Fisher and A. Woo. R.e versus n.h specular highlights. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 388–400. Academic Press, 1994.

[38] Foley, van Dam, Feiner, and Hughes. *Computer Graphics: Principle and Practice.* The Systems Programming Series. Addison-Wesley, second edition, 1997.

[39] Aa. Frøslev-Nielsen. Lys, farver og farvers reproduktion. Technical report, Den Grafiske Højskole, 1996.

[40] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):213–222, July 1984.

[41] Henri Gouraud. Computer display of curved surfaces. Technical Report UTEC-CSc-71-113, Department of Computer Science, Unicersity of Utah, June 1971. Also in *IEEE Transactions on Computers*, vol. C-20, pp. 623–629, June 1971.

[42] Jens Gravesen. *Differential Geometry and Design of Shape and Motion.* Department of Mathematics, Technical University of Denmark, November 2002. Lecture notes for 01243.

[43] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The irradiance volume. *IEEE Computer Graphics & Applications*, 18(2):32–43, 1998.

[44] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime caustics using distributed photon mapping. In H. W. Jensen and A. Keller, editors, *Eurograhics Symposium on Rendering*, 2004.

[45] Pat Hanrahan and Wolfgang Krueger. Reflection from layered surfaces due to subsurface scattering. *Computer Graphics (Proc. SIGGRAPH '93)*, pages 165–174, August 1993.

[46] Xuejun Hao, Thomas Baby, and Amitabh Varshney. Interactive subsurface scattering for translucent meshes. In *Proceedings 2003 ACM Symposium on Interactive 3D Graphics*, April 2003.

[47] Kevin Hawkins and Dave Astle. *OpenGL Game Programming.* Prima Tech's Game Development Series. Premier Press, 2001.

[48] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):145–154, August 1990.

[49] Tim Heidmann. Real shadows real time. *IRIS Universe*, (18):28–31, 1991.

[50] Heinrich Hey. *Photorealistic and Hardware Accelerated Rendering of Complex Scenes.* Ph.D. dissertation, Technischen Universität Wien, Technisch-Naturwissenschaftliche Fakultät, May 2002.

[51] Bjarke Jakobsen, Niels J. Christensen, Bent D. Larsen, and Kim S. Petersen. Boundary correct real-time soft shadows. In *Computer Graphics International 2004 Proceedings*, pages 232–239, June 2004.

[52] Greg James and Simon Green. Real-time animated translucency. Game Developers Conference, 2004. http://developer.nvidia.com/object/gdc_2004_presentations.html.

[53] Henrik Wann Jensen. Global illumination via bidirektional monte carlo ray tracing. Master's thesis, Technical University of Denmark, 1993.

[54] Henrik Wann Jensen. Importance driven path tracing using the photon map. In P. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95 (Proc. of the Sixth Eurographics Workshop on Rendering)*, pages 326–335. Vienna: Springer-Verlag, June 1995.

[55] Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96 (Proc. of the Seventh Eurographics Workshop on Rendering)*, pages 21–30. Vienna: Springer-Verlag, 1996.

[56] Henrik Wann Jensen. *The Photon Map in Global Illumination.* PhD thesis, Technical University of Denmark, September 1996.

[57] Henrik Wann Jensen. Rendering caustics on non-lambertian surfaces. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 116–121, Toronto, May 1996. Canadian Information Processing Society, Canadian Human-Computer Communications Society.

[58] Henrik Wann Jensen. Parallel global illumination using photon mapping. SIGGRAPH 2000 Course Notes, New York: ACM Press, July 2000.

[59] Henrik Wann Jensen. A practical guide to global illumination using photon maps. SIGGRAPH 2000 Course Notes, New York: ACM Press, July 2000.

[60] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* A K Peters, Natick, Massachusetts, 2001.

[61] Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. In J. F. Hughes, editor, *Proceedings of SIGGRAPH 2002*, pages 576–581, July 2002.

[62] Henrik Wann Jensen and Niels Jørgen Christensen. Efficiently rendering shadows using the photon map. In Harold P. Santo, editor, *Compugraphics '95*, pages 285–291, December 1995.

[63] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, March 1995.

[64] Henrik Wann Jensen, Justin Legakis, and Julie Dorsey. Rendering of wet materials. In D. Lischinski and G. W. Larsoi, editors, *Rendering Techniques '99 (Proc. of the Tenth Eurographics Workshop on Rendering)*, pages 273–282. Vienna: Springer-Verlag, 1999.

[65] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of SIGGRAPH 2001*, pages 511–518, August 2001.

[66] James T. Kajiya. The rendering equation. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):143–150, August 1986.

[67] Ashraf A. Kassim and B. V. K. Vijaya Kumar. The wave expansion neural network. *Neurocomputing*, 16:237–258, 1997.

[68] Ashraf A. Kassim and B. V. K. Vijaya Kumar. Path planners based on the wave expansion neural network. *Robotics and Autonomous Systems*, 26:1–22, 1999.

[69] A. Keller. Instant radiosity. In Turner Whitted, editor, *Proceedings of SIGGRAPH 1997*, pages 49–56, Reading, 1997. MA: Addison-Wesley.

[70] Mark J. Kilgard. Improving shadows and reflections via the stencil buffer. NVIDIA White Paper, November 1999. http://developer.nvidia.com.

[71] Michail G. Lagoudakis. Hopfield neural network for dynamic path planning and obstacle avoidance. Semester project for CMPS 588 "Neural Networks", 1997. http://www.cs.duke.edu/∼mgl/acadpape.html.

[72] Bent Dalgaard Larsen. *Global Illumination for Real-Time Applications by Simulating Photon Mapping.* PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, 2004.

[73] Bent Dalgaard Larsen and Niels Jørgen Christensen. Optimizing photon mapping using multiple photon maps for irradiance estimates. In *WSCG'2003 Posters Proceedings*, pages 77–80, February 2003.

[74] Bent Dalgaard Larsen and Niels Jørgen Christensen. Simulating photon mapping for real-time applications. In H. W. Jensen and A. Keller, editors, *Eurographics Symposium on Rendering*, 2004.

[75] Hendrik P. A. Lensch, Michael Goesele, Philippe Bekaert, and Jan Kautz. Interactive rendering of translucent objects. In *Proc. IEEE Pacific Graphics 2002*, pages 214–224, 2002.

[76] Malcolm Longair. Light and colour. In Lamb and Bourriau, editors, *Colour: Art and Science*. Cambridge University Press, 1997.

[77] Rafal Mantiuk, Sumanta Pattanaik, and Karol Myszkowski. Cubemap data structure for interactive global illumination computation in dynamic diffuse environments. In *Proceedings of International Conference on Computer Vision and Graphics*, pages 530–538, September 2002.

[78] Morgan McGuire, John F. Hughes, Kevin T. Egan, Mark J. Kilgard, and Cass Everitt. Fast, practical and robust shadows. NVIDIA White Paper, November 2002. http://developer.nvidia.com.

[79] Tom McReynolds, David Blythe, Brad Grantham, Mark J. Kilgard, and Scott Nelson. Advanced graphics programming techniques using opengl. SIGGRAPH 1999 Course Notes, July 1999.

[80] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth, and Hans-Peter Seidel. Efficient rendering of local subsurface scattering. In Jon Rokne, Reinhard Klein, and Wenping Wang, editors, *11th Pacific Conference on Computer Graphics and Applications*, pages 51–58, Canmore, Canada, October 2003. IEEE.

[81] Tom Mertens, Jan Kautz, Philippe Bekaert, Hans-Peter Seidel, and Frank Van Reeth. Interactive rendering of translucent deformable objects. In Per Christensen and Daniel Cohen-Or, editors, *Eurographics Symposium on Rendering*, pages 130–140, June 2003.

[82] Light: History of light theories. Microsoft® Encarta® Online Encyclopedia, 2004. Reviewed by John H. Marburger. http://encarta.msn.com. Accessed 3rd of June 2004.

[83] Brook Miles. *theForger's Win32 API Tutorial*, 1998–2003. Version 2.0. http://winprog.org/tutorial/.

[84] Trenchard More, Jr. Notes on the development of a theory of arrays. Technical Report 320-3016, IBM Scientific Ctr., Philadelphia, Pa., 1973.

[85] Trenchard More, Jr. Notes on the axioms for a theory of arrays. Technical Report 320-3017, IBM Scientific Ctr., Philadelphia, Pa., 1973.

[86] Trenchard More, Jr. Axioms and theorems for a theory of arrays. *IBM Journal of Research and Development*, 17(2):135–175, 1973.

[87] L. Neumann, M. Feda, M. Kopp, and W. Purgathofer. A new stochastic radiosity method for highly complex scenes. In *Rendering Techniques '94 (Proc. of Fifth Eurographics Workshop on Rendering)*, pages 195–206, Darmstadt, Germany, June 1994.

[88] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometrical considerations and nomenclature for reflectance. Technical report, National Bureau of Standards (US), October 1977.

[89] Fred E. Nicodemus, editor. *Self-Study Manual on Optical Radiation Measurements.* National Institute of Standards and Technology, U.S. Department of Commerce, 1976–1985. NBS Technical notes, Series 910-1 through 8. Chapters were published as completed. http://physics.nist.gov/Divisions/Div844/manual/studymanual.html.

[90] Kasper Høy Nielsen. Real-time hardware-based photorealistic rendering. Master's thesis, Informatics and Mathematical Modeling, Technical University of Denmark, November 2000.

[91] Kasper Høy Nielsen and Niels Jørgen Christensen. Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. *Journal of WSCG*, 3:91–98, 2002.

[92] Mangesh Nijasure. Interactive global illumination on the graphics processing unit. Master's thesis, B. E. University of Bombay, November 2003.

[93] Mangesh Nijasure, Sumanta Pattanaik, and Vineet Goel. Interactive global illumination in dynamic environments using commodity graphics hardware. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, pages 450–454, 2003.

[94] T. Nishita and E. Nakamae. Continuous tone representation of 3-d objects taking account of shadows and interreflection. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):23–30, July 1985.

[95] Cornell University Program of Computer Graphics. Cornell box data. http://www.graphics.cornell.edu/online/box/data.html, April 1998. Accessed 25th of August 2004.

[96] History of color science and theory. http://www.total.net/~daxx/colour_theory_history.shtml. Accessed 3rd of June 2004.

[97] James M. Palmer. Radiometry and photometry faq. http://www.optics.arizona.edu/Palmer/rpfaq/rpfaq.htm, October 2003.

[98] Sybil P. Parker, editor. *McGraw-Hill Dictionary of Mathematics*. McGraw-Hill, 1997.

[99] S. N. Pattanaik and S. P. Mudur. The potential equation and importance in illumination computations. *Computer Graphics Forum*, 12(2):131–136, 1993.

[100] Mark Pauly, Thomas Kollig, and Alexander Keller. Metropolis light transport for participating media. In B. Peroche and H. Rushmeier, editors, *Rendering Techniques '00 (Proc. of the Eleventh Eurographics Workshop on Rendering)*, pages 11–22, New York, 2000. Springer.

[101] Allan Pedersen. An introduction to array theory and nial. Electric Power Engineering Department, Technical University of Denmark, September 1990.

[102] Allan Pedersen and Jens Ulrik Hansen. Q'nial stand-by. Electric Power Engineering Department, Technical University of Denmark, May 1988.

[103] Ingmar Peter and Georg Pietrek. Importance driven construction of photon maps. In G. Drettakis and N. Max, editors, *Rendering Techniques '98 (Proc. of the Ninth Eurographics Workshop on Rendering)*, pages 269–280, Vienna: Springer-Verlag, 1998.

[104] Joseph D. Petruccelli, Balgobin Nandram, and Minghui Chen. *Applied Statistics for Engineers and Scientists*. Prentice Hall, Upper Saddle River, New Jersey, 1999.

[105] Matt Pharr and Pat Hanrahan. Monte carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of SIGGRAPH 2000*, pages 75–84, July 2000.

[106] Bui Tuong Phong. Illumination for computer-generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[107] Max Planck. On the low of distribution of energy in the normal spectrum. *Ann. Physik*, 4(3):553–563, 1901. http://dbhs.wvusd.k12.ca.us/webdocs/Chem-History/Planck-1901/Planck-1901.html.

[108] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In M Doggett, W. Heidrich, W. Mark, and A. Schilling,

editors, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[109] Brian Ramage. Fast radiosity using pixel shaders. *Game Developer*, 11(7):20–39, August 2004.

[110] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH 2001*, pages 497–500, 2001.

[111] Ravi Ramamoorthi and Pat Hanrahan. Frequency space environment map rendering. In *Proceedings of SIGGRAPH 2002*, pages 517–526, 2002.

[112] Barnett Rich. *Review of Elementary Mathematics*. Schaum's Outlines, second edition, 1997. revised by Philip A. Schmidt.

[113] Ton Roosendaal and Carsten Wartmann, editors. *The Official Blender Gamekit: Interactive 3-D for Artists*. No Starch Press, San Francisco, 2003.

[114] Holly E. Rushmeier. Extending the radiosity method to transmitting and specularly reflecting surfaces. Master's thesis, Program of Computer Graphics, Cornell University, 1986.

[115] Holly E. Rushmeier and Kenneth E. Torrance. Extending the radiosity method to include specularly reflecting and translucent materials. *ACM Transactions on Graphics*, 9(1):1–27, January 1990.

[116] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of Photonics*. John Wiley & Sons, New York, 1991.

[117] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of SIGGRAPH 2000*, pages 229–238, July 2000.

[118] Christophe Schlick. An inexpensive BRDF model for physically based rendering. *Computer Graphics Forum (Proc. of Eurographics '94)*, 13(3):233–246, September 1994.

[119] Mark Q. Shaw. Evaluating the 1931 cie color matching functions. Master's thesis, Center of Imaging Science, Rochester Institute of Technology, Rochester, New York, June 1999.

[120] Ken Shoemake. Animating rotation with quaternion curves. *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 245–254, July 1985.

[121] Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer.* Taylor & Francis, New York, fourth edition, 2002.

[122] François X. Sillion, James R. Arvo, Stephen H. Westin, and Donald P. Greenberg. A global illumination solution for general reflectance distributions. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):187–196, August 1991.

[123] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics*, 21(3):527–536, July 2002.

[124] Philipp Slussalek. *Vision: An Architecture for Physically-Based Rendering.* Ph.D. dissertation, Der Teknischen Fakultät der Universität Erlangen-Nürnberg, 1995.

[125] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[126] Frank Suykens De Laet. *On Robust Monte Carlo Algorithms for Multi-Pass Global Illumination.* Ph.D. dissertation, Katholieke Universiteit Leuven, Dept. of Computer Science, September 2002.

[127] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26:55–64, July 1992.

[128] Urb. lat. 1329 fol. 1 recto math04 NS.19.

[129] Tom Vykruta. Simple and efficient line-of-sight for 3d landscapes. In Steve Rabin, editor, *AI Game Programming Wisdom.* Charles River Media, Inc., Hingham, Massachusetts, 2002.

[130] Michael Wand and Wolfgang Straßer. Real-time caustics. In P. Brunet and D. Fellner, editors, *Computer Graphics Forum*, volume 22(3), September 2003.

[131] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, July 2003.

[132] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):265–272, July 1992.

[133] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):85–92, August 1988.

[134] Alan Watt and Fabio Policarpo. *3D Games: Real-Time Rendering and Software Technology*, volume One of *ACM SIGGRAPH Series*. Addison-Wesley, 2001.

[135] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992.

[136] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, Massachusetts, 1999.

[137] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[138] Bill Williams. A history of light and lighting. http://www.mts.net/~william5/history/hol.htm, 1999. Edition 2.2. Accessed 3rd of June 2004.

[139] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

[140] Lance Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):1–11, July 1983.

[141] Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with hammersley and halton points. *Journal of Graphics Tools*, 2(2):9–24, 1997.

[142] Chris Wynn and Sim Dietrich. Cube maps. NVIDIA Presentation, March 2001. http://developer.nvidia.com.

[143] Information technology | computer graphics and image processing | extensible 3d (x3d). ISO/IEC FDIS 19775:200x.

[144] Information technology | computer graphics and image processing | extensible 3d (x3d) encodings. ISO/IEC 19776:200x.