

**Faculty of Computer Science
University of A Coruña**

**FINAL DEGREE PROJECT
TECHNICAL ENGINEERING IN COMPUTER SYSTEMS**

Global Illumination for Point-Based Rendering

Student: Luis Omar Alvarez Mures
Advisors: Emilio José Padrón González
Alberto Jaspe Villanueva
Date: July 28, 2012

Título: Iluminación global para render baseado en puntos

Clase: Proxecto de desenvolvemento en investigación

Autor: Luis Omar Alvarez Mures

Directores: Alberto Jaspe Villanueva
Emilio José Padrón González

Titor: Emilio José Padrón González

Tribunal: Emilio José Padrón González
Guillermo López Taboada
José Rodrigo Sanjurjo Amado

Data de lectura: 9 de xullo, 2012

Cualificación:

Dedicated to Rebeca and my parents.

Acknowledgements¹

My sincerest thank you for the help provided:

To Mr. Alberto Jaspe Villanueva.

To Mr. Emilio José Padrón González.

¹In alphabetical order

Resumo:

O render baseado en puntos ten múltiples vantaxes respecto do clásico que utiliza mallas de polígonos, sobre todo no caso no que os polígonos poidan chegar a ser máis pequenos ca un pixel.

O obxectivo deste proxecto é deseñar e implementar un motor de render baseado en puntos que utilice raytracing como método de iluminación global. Ao ser un campo todavía pouco explorado, o proxecto presenta unha importante compoñente investigadora, xa que implica a búsqueda de solucións que nos permitan resolver os distintos problemas que xurden nas diferentes fases dun *pipeline* gráfico para o render de puntos. Tres son as principais metas que se propuxeron acadar con este proxecto:

- Desenvolver un motor de *ray tracing* desde cero.
- Investigar como as técnicas de iluminación global máis habituais poden ser aplicadas en render baseado en puntos.
- Comprobar o potencial do punto como primitiva gráfica dun motor de render.

O motor proposto segue un deseño totalmente modular, con varias etapas que permiten desacoplar as distintas operacións a realizar para obter o render dunha escena 3D. Así, primeiro precisaremos gardar a información que o motor necesita para renderizar unha escea (cousas coma a posición da cámara, orientación, etc.). Para iso utilízase un arquivo de configuración XML que contén todos estos datos, así como calquera outra información relevante para o proceso de render.

Seguidamente precísase gardar os datos da nube de puntos. Para iso é necesario saber a posición, vector normal, etc. de cada punto, para o que utilizamos un arquivo de texto ASCII que contén toda a información da escena de entrada e que ademais soporta control de versións, permitindo cambios no formato. Tamén ofreceremos unha interfaz integrada en Blender que permite ao usuario crear datasets en Blender e renderizarlos co noso motor. Isto dota ó proxecto dunha gran flexibilidade, xa que é capaz de renderizar datasets ou calquer escea en Blender.

Despois deste paso, precísase elixir o modelo de cámara a usar. No noso motor implementamos dúas opcións diferentes. Así, é posible elixir entre unha cámara axonométrica ou en perspectiva (dependendo do modo de render elixido). Por outra banda, tamén é posible elixir entre obter o render mediante a proxección da escena na cámara ou ben usando raytracing. Aínda que o motor implementa ambas alternativas, concentrarémonos no raytracing, posto que é a opción que máis posibilidades e prestacións ofrece, sendo o caso a estudar no proxecto.

É importante reseñar que debido á natureza das nubes de puntos, que tenden a ter unha gran cantidade de datos, para a aceleración do proceso de raytracing o motor permite utilizar unha estrutura de aceleración espacial que permita organizar as nubes de puntos para optimizar determinadas operacións cos mesmos (intersección raio-escena, búsqueda de veciños, etc.). Concretamente, neste proxecto implementouse un k-d tree como estrutura de aceleración. O uso do k-d tree permite obter unha importante mellora no rendemento do noso motor.

Respecto ao tradicional render baseado en polígonos, o render con puntos presenta algunhas desvantaxes, como por exemplo a natureza adimensional dos mesmos. Isto quere dicir que o punto non ten superficie, volumen ou normal, o que introduce novos problemas en moitas das operacións comúns dun motor de render: interseccións, iluminación, etc. Concretamente, no caso do cálculo da iluminación é imprescindible dispor dos vectores normais ás distintas superficies da escena, neste caso representadas por puntos. Se non se dispón destas normais, é necesario estimalas dalgún xeito. Para resolver este problema o proxecto utiliza un método de mínimos cadrados que se usa para estimar a normal nun punto dependendo de donde estén os seus veciños.

A derradeira etapa do noso motor, pero non menos importante de acordo aos obxectivos do proxecto, é a iluminación da escena. O motor ofrece dúas opcións. Por un lado pódese utilizar un método de *hard shadows*, menos custoso computacionalmente pero con resultados pouco realistas. Por outra banda, o motor implementa o método de *Monte Carlo*, modelo de iluminación global que trata de simular o comportamento real da luz nun contorno, ofrecendo resultados máis realistas, aínda que significativamente máis custosos.

Posto que o motor fai un importante uso de distintas operacións e funcións matemáticas ao longo de todas as súas fases, o proxecto tamén incorpora unha librería matemática que se encarga destas tarefas.

Abstract:

The objective of this project is designing and implementing a point-based rendering engine that utilizes ray tracing for global illumination computation.

Point-based rendering has several advantages compared to the more classic polygon-based rendering that uses polygon meshes, specially when polygons can be smaller than a pixel.

In this project we explore how to represent point models without ray tracing techniques (with a projective camera) and using ray tracing that is the true focus of this project.

It is also important to mention that because of the nature of point clouds, datasets tend to be really large. That is why a k-d tree is used to accelerate the ray tracing process. A k-d tree is a space partitioning structure for data organization (in our case, point data). This acceleration structure reduces greatly rendering times, and without it the rendering process would take too long and be too computationally expensive.

Point-based rendering also has some disadvantages, like for example the zero-dimensional nature of points. This means that a point has no surface, volume or normal vector. This results in problems in the rendering process. In this project we explore several solutions for these problems.

To facilitate the usage of the project to the end user, and to also make testing easier; a plugin compatible with Blender will be programmed. It allows the rendering of scenes created in Blender with our project and facilitates the use of the engine thanks to the integration with the interface of Blender.

Lista de palabras clave:

Motor, render, nubes de puntos, k-d tree, iluminación global, ray tracing, Monte Carlo.

Keywords:

Engine, render, point clouds, k-d tree, global illumination, ray tracing, Monte Carlo.

Hardware y software utilizado:

- Intel Core i5.
- C++, Xcode, Git, libpng, tinyXML, openMP.

Contents

1	Introduction	1
1.1	BDE	2
1.1.1	Motivation and context	2
1.1.2	Objectives	3
2	Planification and methodology	7
2.1	Agile software development	7
2.1.1	Agile manifesto	7
2.1.2	Description	8
2.1.3	Application	9
3	CG basics	13
3.1	Rendering	13
3.1.1	Rendering process	15
3.1.2	Models	15
3.2	Illumination	16
3.3	Transformations	17
3.4	Surfels	18
3.4.1	Sphere approach	19
3.4.2	Disc approach	20
3.5	Camera model	21

3.5.1	Parameters	22
3.5.2	Inner workings	23
4	Structure	27
4.1	Analysis	27
4.2	State of the art	28
4.2.1	Point-based rendering	29
4.2.2	Global Illumination in point-based rendering	30
4.3	Design	31
5	I/O and Blender integration	33
5.1	I/O	33
5.1.1	Design	33
5.1.2	Point cloud input formats	34
5.1.3	XML config file	35
5.1.4	Image output	37
5.2	Blender integration	37
5.2.1	Loading the plugin	38
5.2.2	Using the plugin	38
5.2.3	Normal visualization plugin	39
5.2.4	Point cloud visualization plugin	39
6	k-d tree	41
6.1	Design	41
6.2	Tree representation	42
6.3	Tree construction	42
6.4	Tree traversal	46
6.5	Performance tests	47
7	Normal estimation	51

7.1	State of the art	51
7.2	MLS	52
7.3	Algorithm: design and implementation	53
8	Ray tracing	57
8.1	Introduction	57
8.2	Design	58
8.3	Ray-scene intersection	61
8.4	Shadows	62
8.5	Surfel overlapping	63
8.6	Supersampling anti-aliasing	65
8.7	Performance tests	66
9	Global illumination	69
9.1	Shading	70
9.2	Hard shadows	70
9.2.1	Diffuse reflection	72
9.2.2	Specular reflection	72
9.3	Monte Carlo	73
9.3.1	State of the art	74
9.3.2	Design	75
9.3.3	The rendering equation	76
9.3.4	Monte Carlo integration	77
9.3.5	Monte Carlo sampling	78
9.3.6	Brute force Monte Carlo	80
9.4	Results	82
10	Conclusions and future lines of work	89
10.1	Conclusions	89

10.2 Future lines of work	90
-------------------------------------	----

List of Figures

1.1	Modeling types comparison	1
2.1	Gantt chart	11
3.1	Rendering types comparison	14
3.2	Model types comparison	16
3.3	Point cloud	19
3.4	Sphere approach	19
3.5	Disc approach	20
3.6	Pinhole camera	21
3.7	Camera orientation	22
3.8	Hither and Yon	23
3.9	Viewing frustum	24
3.10	Ray casting	25
4.1	BDE pipeline	28
4.2	Class diagram	32
5.1	I/O class diagram	33
5.2	Data file example	35
5.3	XML config file example	36
5.4	Example image output	37
5.5	BDE panel	38

5.6	BDE normal plugin	40
6.1	k-d tree	42
6.2	k-d tree	43
6.3	No splits available	45
6.4	Tree traversal	46
6.5	Surfels and rays versus time	47
6.6	k-d tree comparison	48
7.1	MLS	53
7.2	Scene class diagram	55
8.1	Ray tracing	58
8.2	Shadow rays	60
8.3	Shadow rays	62
8.4	Surfel overlapping	63
8.5	Surfel overlapping	64
8.6	Weight average diagram	64
8.7	Multithread performance comparison	66
9.1	Ray tracing with hard shadows	71
9.2	Diffuse reflection	72
9.3	Specular reflection	73
9.4	Monte Carlo class diagram	75
9.5	Material class diagram	75
9.6	Rendering equation diagram	76
9.7	Polar coordinates	79
9.8	Indirect bounces	82
9.9	Monte Carlo performance	83
9.10	Sampling comparison	84

9.11	Render result 1	84
9.12	Render result 2	85
9.13	Render result 3	85
9.14	Render result 4	86
9.15	Render result 5	86
9.16	Render result 6	87
9.17	Render result 7	87

List of Tables

2.1	Cost estimate	10
6.1	k-d tree comparison data	49

Chapter 1

Introduction

Computer Graphics are graphics that are created using a computer. Advances in computer graphics have had an impact on several media types such as for example, the movies and video games industry.

Images are normally captured by optical devices; such as cameras, mirrors, lenses, etc. Rendering is the process of generating an image from a model, the *scene*, by means of a computer. A scene contains information like geometry, lighting, etc. about a virtual scene.

Typically, polygonal modeling has been the approach followed for modeling objects in the scene, approximating their surfaces using polygons. Point-based graphics focuses on points as the fundamental representation of the surfaces instead of polygons (see Figure 1.1).

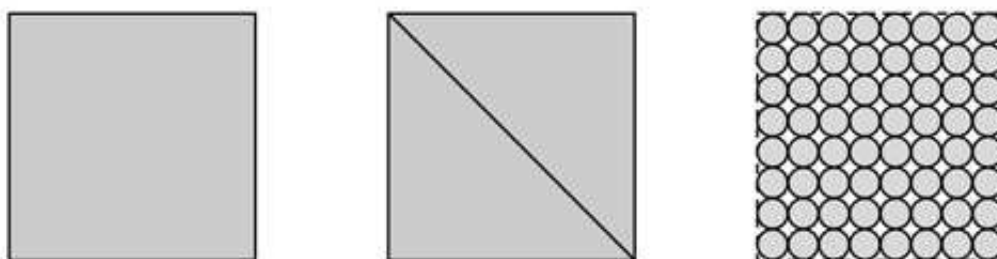


Figure 1.1: From left to right; mathematical representation of a plane, representation using two polygons (triangles) and a point-based representation.

Once a representation of the object is stored in a model, a lighting simulation algorithm is required to illuminate the scene. Global Illumination is a group of algorithms that were created to add more realistic lighting to 3D scenes and take into account not only the light that reaches an object directly, but also the light that is reflected on other surfaces and reaches the object indirectly. The objective is to achieve a physically-based simulation of light transport in the scene.

Computing global illumination means to solve an integral equation, so some of the main global illumination approaches are based on Monte Carlo integration, a method that is often used in simulations of physical and mathematical systems. It performs a numerical integration using random numbers relying on repeated random sampling to obtain a result. Usually ray tracing is used hand in hand with Monte Carlo integration, because it can be used to simulate how rays of light move through the scene.

All of these concepts will be explained more in depth in their corresponding chapters.

1.1 BDE: a point-based global illumination rendering engine

BDE stands for **Black Diamond Engine**, this is the name that we have given to our point-based rendering engine. In this chapter we will briefly introduce the motivation that led us to choose this project, the objectives we have set and the structure of this document.

1.1.1 Project motivation and context

As performance of the 3D hardware has grown exponentially, polygons get smaller and cover a few pixels. Overhead for triangle setup is magnified and is leading to a high rendering cost per pixel. With points we are simplifying the rendering pipeline, using a simpler and more efficient rendering primitive than triangles. Points are a more natural choice when we want to represent organic models that may have very small features and very complex contours.

Apart from the advances in rendering hardware, there has also been a lot of progress in 3D acquisition technology. Modern 3D scanning devices such as laser range scanners acquire huge amounts of point data. A usual approach to visualize this data with polygons is to process this range data and create consistent triangle meshes. This is complicated and time consuming task. With point based rendering this process is not necessary as it uses point data as the basic input primitive.

The big difference is that points do not have connectivity or topology, that is why we could need to interpolate the surface that they are representing, indeed, we need to do this to compute an accurate illumination of the scene.

Point based rendering has evolved in the last years greatly, a lot of researchers have been exploring solutions for the efficient representation and rendering of point-based geometry. Multiple conferences and workshops organized in the last years have had point-based graphics among their topics (e.g. the first Symposium on Point-Based Graphics took place in Zürich, Switzerland, in 2004), as well as scientific journals and books. The huge number of submissions for these events shows that there is a lot of interest and future in this field.

In 2004, Michael Bunnell from NVIDIA published a chapter in GPU Gems 2 [PFS04] in which he documented his method for computing ambient occlusion¹ without using ray tracing, the usual approach, but using a point cloud to approximate the computation of illumination. This was a great idea and soon Pixar² investigated how to implement this same ideas in their RenderMan engine.

Pixar has used point-based rendering in different movies, such as “Surf’s Up” and “Pirates of the Caribbean: Dead Man’s Chest” for computing diffuse global illumination (color bleeding). The method has been used in production of more than a dozen feature films, for example for rendering Davy Jones and his crew in two of the “Pirates of the Caribbean” movies [Chr05]. Their approach is not a purely point-based rendering engine, they just use point clouds to store some information about a polygon-based scene. Pure point-based rendering is still in an early stage and is uncharted territory in most commercial engines.

1.1.2 Project objectives

The aim of this project is the design and implementation of a point-based rendering engine with global illumination using ray tracing. The engine will implement the researched solutions for the main problems that arise in the development of the different stages of a point rendering pipeline. This will allow us to provide a functional engine that covers some of the most important aspects in the rendering pipeline. Additionally we have integrated our engine with a popular open-source 3D software used in rendering and modeling called Blender [Fou12].

From a research and development standpoint, this project gives us an overview of what

¹A diffuse shadowing effect.

²Pixar is a computer animation film studio well known for the production of multiple CG animated feature films. Pixar’s films have been created using its PhotoRealistic RenderMan software, their own implementation of the RenderMan rendering application.

stages in a rendering pipeline for point-based graphics present a bigger challenge as of today, being this a field that it is still in an early stage in the Computer Graphics world.

Specifically, the main objectives of this project are:

- Build a raytracer from scratch.
- Research how traditional global illumination techniques can be applied to point-based rendering.
- How does using points as a primitive affect the classical rendering pipeline.

Because points are zero-dimensional [GP07]; i.e. they do not have volume, area or any other higher-dimensional analogue, to ray trace them we need to give the points an area or the rays would't intersect with anything in the scene. Two different approaches have been used in the project to deal with this issue. One uses spheres as the representation of the points (surfel), whereas the other uses discs. Additionally, normal vectors are usually needed to compute global illumination, so it is necessary to estimate them if they are not available. To solve this issue we will treat it as a Least Squares minimization problem, that will estimate the best normal for a point depending on its neighbors.

A lot of math is involved in all of these steps. Because of this, the project will also have a library that will be in charge of all the math that is needed in the engine. Once implemented it will rarely change so it will save compiling time to have a separate library for this task already pre-compiled.

Document structure

Because of the research nature of the project, we have several clearly defined stages along the way. The document has been structured in light of this. In most chapters there is a previous analysis step, an algorithmic solution to the problem described in the chapter and the theoretical foundations that the solution has been based on.

Chapter 2. Planification and methodology

We start by describing the methodology used for the development of the project. This chapter also lays out the Gantt chart that illustrates the project schedule.

Chapter 3. Computer Graphics basics

This chapter briefs the basic concepts about computer graphics. How does the camera work, what are 3D transformations and what is a surfel are all questions answered in this chapter.

Chapter 4. Structure of a point-based global illumination engine

This chapter describes the structure of BDE and presents the class diagram of the engine. We also include a state of the art about point-based rendering and global illumination.

Chapter 5. I/O and Blender integration

BDE needs to be able to load and store a lot of different information to render a scene (things like camera position, orientation, etc.) and the scene itself (the point cloud data) has to be stored somehow (point position, normal vector if available, etc.).

This chapter explains everything related to input and output. How does the engine store point cloud data in an ASCII text file. How is the XML configuration file containing all information related to the camera and render process structured. What type of image output does the engine support.

Since we provide an interface integrated in Blender that will allow the user to create the dataset in Blender and render it with our engine, this chapter also describes what plugins for blender we have made available and how to use them. This adds the project a lot of flexibility, as it is capable of rendering provided datasets or your own Blender creation.

Chapter 6. Acceleration structure: k-d tree

Since ray tracing tends to be computationally expensive due to the great amount of ray-scene intersection tests that need to be computed, and working with point clouds usually means dealing with a huge amount of data, a naive approach without acceleration structures yields really bad performance results. Because of this we have used a k-d tree as an acceleration structure to significantly reduce rendering times. A k-d tree [PH10] is a space-partitioning data structure for organizing “data” in a k-dimensional space. K-d trees are a special case of binary space partitioning trees. The k-d tree will use up a little more CPU time to read the dataset (building the k-d tree), but in the end the CPU time that saves later makes up for this fact.

The chapter explains how does a k-d tree work and describes the specific implementation that we have used in our project. An analysis of the k-d tree performance under different circumstances is also presented in this chapter.

Chapter 7. Normal estimation

Surface normals are needed to compute illumination, so we have to estimate them if they are not provided with our point cloud data. In this chapter we first explore different ways of calculating surfel normals in the state of the art section, then we describe the theoretical foundations of the method chosen for our engine and lastly we explain how we have implemented it.

Chapter 8. Ray tracing

We explain how ray tracing works, methods for ray-sphere and ray-disc intersections, how are shadows calculated, our antialiasing technique and finally we offer some performance tests that show us if multithreading affects positively rendering times.

Chapter 9. Global illumination

This chapter provides an in-depth explanation about the illumination techniques used in this project. It explains how we use Monte Carlo integration [Caf98] to solve the global illumination problem that the rendering equation presents. Furthermore, a state of the art of Monte Carlo global illumination methods is commented, as well as some basic radiometry concepts are introduced.

Chapter 10. Conclusions and future lines of work

In this chapter we explore what we have learned with the project, analyze whether the objectives have been met and take a look at possible lines of future work for the project.

Chapter 2

Planification and methodology

This chapter explains what software development method was used in the project. We are strong believers in agile software development, above all for research oriented projects like this. In the next sections, some basic concepts of this development philosophy are introduced, as well as how it was applied in our project. Furthermore, a Gantt chart with the different steps and periods of the development is presented so that the reader can see how long each element of the project took. Since this is a project that requires a lot of research, it is difficult to plan ahead; because we do not know how long we are going to spend on a certain element. Finally we will give a cost table with the estimated project cost.

2.1 Agile software development

Agile software development [Coc01] is a combination of development methods that use iterative and incremental development, where requirements and solutions mature through collaboration between self-organizing, cross-functional teams. It encourages adaptative planning, evolutionary development and delivery, a time-boxed iterative approach, and promotes quick and flexible response to change.

2.1.1 Agile manifesto

In February of 2001, several developers met at Snowbird, Utah resort, to debate different lightweight development methods. They published the *Manifesto for Agile Software Development* [Bec01] to define the approach that is now called agile software development.

The conclusions that we can reach from the manifesto's items are described below:

- **Individuals and Interactions:** In agile development self-organization and motivation are really important. Other values promoted by the manifesto are co-location¹ and pair programming².
- **Working software:** Working software will be utilized for more purposes than presenting documents to the client.
- **Customer collaboration:** The software requirements cannot be fully realized from the beginning of the software development cycle, so being in touch with the customer is really important.
- **Responding to change:** Agile development is keen on fast responses to change and continuous development.

More principles are mentioned in the manifesto, some of them are:

- Customer satisfaction by rapid delivery of useful software.
- Welcome changes even late in the development.
- Working software is the principal measure of progress.
- Maintaining a constant pace.
- Cooperation between business people and developers.
- Attention to technical excellence.
- Simplicity.

2.1.2 Description

Agile methods break down task into small increments with minimal planning and normally long-term planning is not directly involved. *Iterations* are short timeframes that typically last from one to four weeks. A team works in each iteration through a full software development cycle; including planning, requirements analysis, design, coding, etc. This minimizes risk and facilitates adaptation to change. An iteration may not add enough new functionalities to warrant a market release, but the objective is to have an available release at the end of each iteration.

¹The act of placing multiple individuals within a single location.

²Two programmers work together at one workstation.

Team composition does not depend on corporate hierarchies or corporate roles of team members. They normally have the responsibility of completing tasks that deliver the required functionalities that an iteration requires. How to meet an iteration's objectives is decided individually.

Agile methods encourage face-to-face communication instead of written documents if possible. Most teams work in an open office (the *bullpen*), which makes this type of communication easier.

Each agile team contains a customer representative, that ensures that customer needs and company goals are aligned.

Most agile methods encourage a routine that includes daily face-to-face communication among team members. In a brief session team members tell each other what they achieved the previous day, what they are going to do today and the problems that have appeared.

As agile development emphasizes on working software as the primary measure of progress and has a clear preference in face-to-face communication this results in less written documentation than other methods.

2.1.3 Application

We have used in our project several of the premises of agile software development. We have used approximately weekly or biweekly iterations focused on completing simple objectives that would yield working software. The work in each iteration has been constant and diligent, focusing always on completing the task at hand, but not avoiding changes if it was deemed necessary.

The meetings with my two advisors were kept on face-to-face basis when possible. In each meeting we would comment what had been achieved since the last one, laid out what the objectives were for the next iteration and talked about the roadblocks found on the task at hand. Whenever communication in person was not possible, e-mails were exchanged regularly if tasks were completed or any other event arised.

To keep the code accesible by everyone in the team, up to date and well organized **Git** [Tor05] version control software was used. Git is free software used to manage source code and keep distributed revision control. Every Git working directory is a repository with complete history and revision tracking capabilities. In this project it was esential to have a system like this in place, because of the research nature of the project. Ideas needed to be tried without wasting time on keeping tabs on the added code, Git makes this really easy and not problematic thanks to history and revision tracking.

Resource	Units	Hours	Cost per hour	Total
Software				0 €
Computer	1			1000 €
Analyst-programmer	1	800	15 €	12000 €
				13000 €

Table 2.1: Table showing the estimated cost of the project.

Although because of the research nature of the project it was not possible to estimate how much time we would spend in each iteration, we provide a Gantt chart that depicts how the project iterations were performed after the fact (see Figure 2.1).

We have also estimated the cost of the project as we can see in Table 2.1. It only includes human resources and hardware costs because all the software used was free.

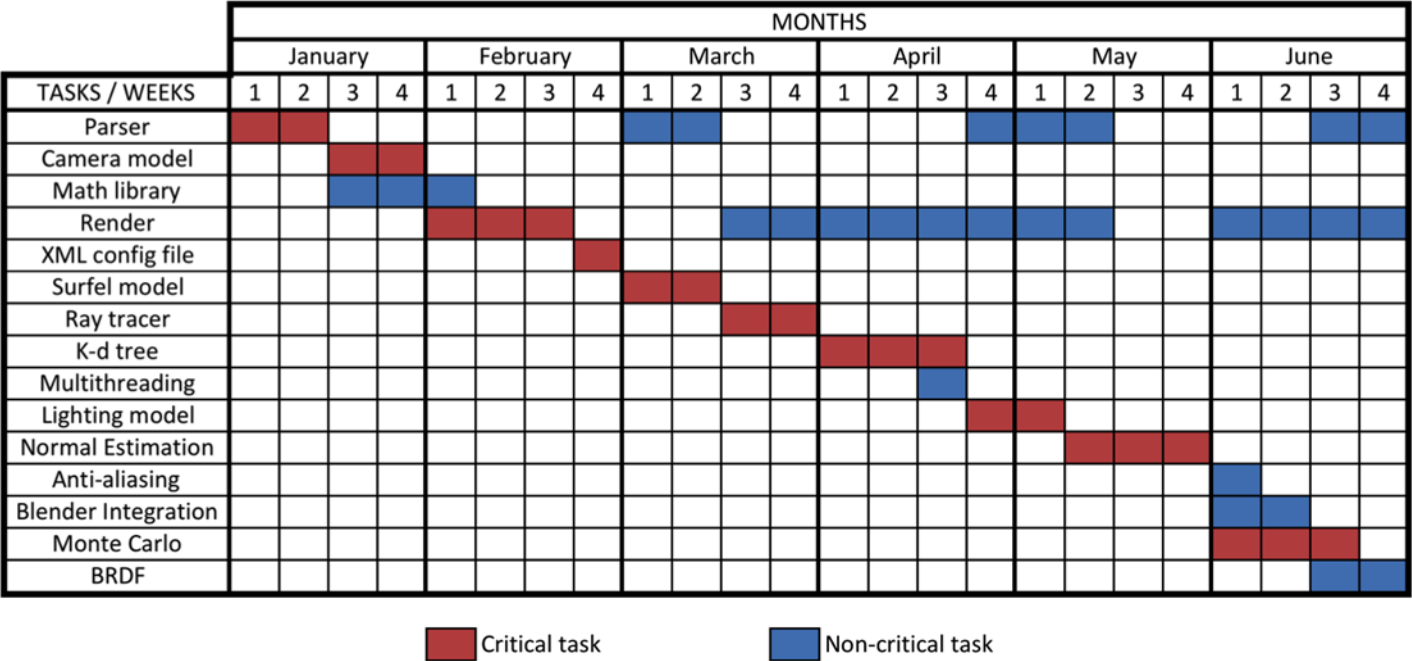


Figure 2.1: Gantt chart of the project.

Chapter 3

Computer Graphics basics

In this chapter, some concepts in 3D rendering are introduced with special attention to the main components of a point-based rendering engine. We first describe how the ray casting process that generates the frame-buffer with the final render works. This means that we have to explain the camera model. We will also depict how we represent points in space by means of surfels.

Only a brief description of these concepts is presented, to get more familiarized with Computer Graphics the book [FvDFH12] is recommended.

3.1 Rendering

Traditionally Computer Graphics has been defined as the computer science discipline that is dedicated to synthesizing images algorithmically with computers. Nowadays we can find even more related topics like hyper realistic photography, animation techniques, virtual reality, etc. To generate images from three-dimensional scenes a process called **render** is used. It is in charge of modeling objects and their properties, illumination and the camera that will capture everything.

As has been said before, rendering is the computational process of generating an image from a model. From this definition multiple interpretations are possible, from creating a 3D animation film to a bar chart in a spreadsheet, all of them are equally valid. Although the term is normally used when the model we are using is of a spatial nature and more specifically three-dimensional [Jas12].

Several classifications of rendering techniques could be made, but for this introduction we will use two of them. On one side, if we use the type or style of the image we want to

achieve, we will have:

- **Non-photorealistic rendering**, uses other types of effects that render the scene with artistic style, intended to look like a painting or drawing.
- **Photorealistic rendering**, tries to be as faithful as possible to reality. This type of render can be subdivided in:
 - **Physically-based rendering**: Tries to compute an authentic simulation of light transport through the virtual scene and its interaction with materials and objects. The precision of this simulation will depend on the mathematical and physical models chosen.
 - **Faked**: Utilizes algorithmic tricks, not trying to pretend an ad-hoc simulation of light, usually to reduce render times.



Figure 3.1: A photorealistic render on the left, versus a non-photorealistic render on the right.

On the other side, if we compare the interaction capabilities of the rendering application with the user, we can divide rendering techniques in:

- **Offline rendering**: Where the process that generates the image is too slow to respond instantaneously to the user interactions. A time scale of seconds and up to several days can be considered “slow”. This happens for example when generating photograms for a movie.
- **Online rendering**: Where processing time would be short enough to respond to user interactions so that the user has the sensation of continuity. A time scale of milliseconds would be needed to achieve this effect. Normally it is measured in *frames per second* (FPS). A good example of this technique are videogames.

Most of the rest of this document is devoted to photorealistic and offline rendering techniques, since that is the kind of rendering BDE is focused on.

For more information about physically-based rendering the reader can consult [PH10] and about real-time rendering [AMHH08].

3.1.1 Rendering process

There are several techniques for generating images from a model. All of them start with a camera position, the geometry of the objects is projected in that direction one way or another, calculating the color of each resulting pixel. The two families of techniques are the ones based on *scanline rendering*, where lists of geometry are traversed with horizontal scan lines. Intermediate calculations determine what object is closer to the camera, how do the lights in the scene affect the objects, etc.

And the ones based on *raycasting*, or ray emission, that are used by the majority of offline rendering engines. These rays are “semi-infinite” segments defined by an origin position and a direction, that emulate a ray of light.

Raytracing [Gla89] is probably the most famous algorithm based on raycasting. Of recursive nature, its principal loop traces a ray per each pixel of the image that is going to be generated, that tries to simulate the inverse path that the light follows. When each of these primary rays intersect with something in the scene, they cast new rays from the intersection point to calculate things like shadows or reflections.

The ponderation of each of these rays will generate a unique color for each primary ray, that will be shown in the generated image. This topic is covered in more depth in Chapter 8.

3.1.2 Models

Modeling describes the process of forming the shape of a virtual object in a computer. There are several types of models (see Figure 3.2):

- **Polygon-based:** Describe the surface of an object as a set of polygons. Triangles are the most commonly used polygons, since they are flat and trivially convex. Most of the existing graphics hardware is optimized for this primitive.
- **Voxel-based:** Divides space in a regular 3D grid where the *voxel* is the smallest unit of volume. Each cell is either filled or not and depending on that the pixels are shaded. Memory increases as precision is augmented.

- **Point-based:** Objects are represented by point samples of their surface, that is why they are usually called *surfels*. Each point has a position and some information about the surface that it belongs to. Compared to the traditional triangle-based approach, this primitive needs its own rendering techniques and its own pipeline, since different challenges must be faced. This is our case study and because of that, this primitive will be further explained in Section 3.4.

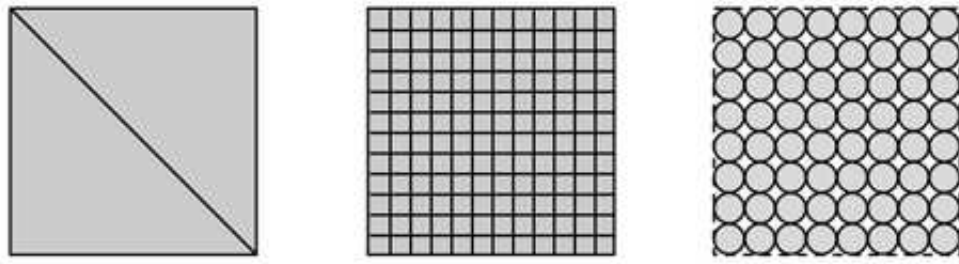


Figure 3.2: From left to right; a polygon-based, a voxel-based and a point-based model of a plane (2D representation).

3.2 Illumination

In rendering, the problem of “illumination” is really complex, because as in the real world, the generated image depends directly on the rays of light that reach the camera sensor (in this case, the pixel matrix). As the models that the algorithm uses are more demanding, the image obtained will be more realistic, and the complexity of the computations and scene description will increase.

In this simulation, the calculation of how light is transmitted over 3D space and scattered in the scene is the most important and complex factor. The set of techniques and methods that deal with this problem is normally denominated *Light Transport Theory*.

RGB is an additive color model in which red, green and blue are combined to form a wide variety of different colors. The name of the model is based on the initials of the additive primary colors, red, green and blue. It is used to describe the color of the pixels that represent the surface of an object in the resulting image. To know the RGB color that an opaque surface reflects, normally a *bidirectional reflectance distribution function* (BRDF) is used.

Furthermore, it is necessary to know the light sources in the scene and how they work. Some of the most used models for light sources are:

- **Point lights**, points in space that emit light in every single direction.
- **Ambient lights**, that reside in the scene volume and can be set fixed (ambient) or stored in directional ambient maps (HDRI).
- **Emitted by objects in the scene**, from their surface.

Finally, the model chosen for the materials of the objects in the scene is also important. Thus, the aspect of a figure in the scene can change between crystal, stone, etc. depending on the material. This is achieved by modifying the properties of the light that arrives at an object and then is reflected back to the rest of the scene.

Because of the complex and continuous nature of light, any realistic light transport model chosen for rendering involves solving an integral equation. As solving this integrals analytically is not a viable option, we will need to make use of numerical methods and probably one of the best suited for this job is *Monte Carlo* integration. Please refer to Chapter 9 for more information about this method.

3.3 3D Euclidean space and transformations

In 3D computer graphics we normally work with a three-dimensional space of Euclidean geometry, the term “Euclidean” is used to distinguish between these spaces and the curved spaces of non-Euclidean geometry. The most common types of operations in Euclidean geometry can be represented with a transformation matrix if homogeneous coordinates are used. Because of this a transformation $\mathbf{T}$ can be used for several purposes.

Using basic linear algebra concepts, a 4×4 matrix can be used to express the linear transformation of a point or vector. A transformation will then be represented by the elements of the 4×4 matrix. They can also be used to perform some transformations that are non-linear on an Euclidean space. For this reason transformation matrices are widely used in computer graphics.

Generally a transformation is a mapping from points to points or vectors to vectors [PH10], for example:

$$p' = \mathbf{T}(p) \quad v' = \mathbf{T}(v)$$

To transform the points or vectors we just have to perform the appropriate matrix multiplications. This also allows the composition of transformations, we just have to multiply

the transformation matrices.

Most commonly used transformations are:

- **Rotation**, will rotate a point or a vector by a given angle either around an arbitrary axis or the x , y or z axis.
- **Scaling**, takes a point or a vector and will scale its x , y and z components by a factor.
- **Translation**, only affects points and will translate coordinates x , y and z a set amount.
- **Look-at**, this is a useful transformation for placing a camera in the scene. It can be used to give the camera a position and a point at which the camera is looking at.

To illustrate how the transformations are represented by a 4×4 matrix, equation 3.1 shows the generic transformation matrix for a translation:

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.1)$$

This transformation is applied to a point $P = (x, y, z)$ in the following way:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix}$$

3.4 Surfels and points

A point cloud is a set of vertices or points in a three-dimensional coordinate system. These vertices are usually positioned in 3D space and have a set of coordinates (x, y, z) . These sets of points normally are representative of the external surface of an object.

Point clouds are normally created by 3D scanners, these devices capture automatically a large number of points on the surface of an object (see Figure 3.3), and yield a point cloud file as a result.

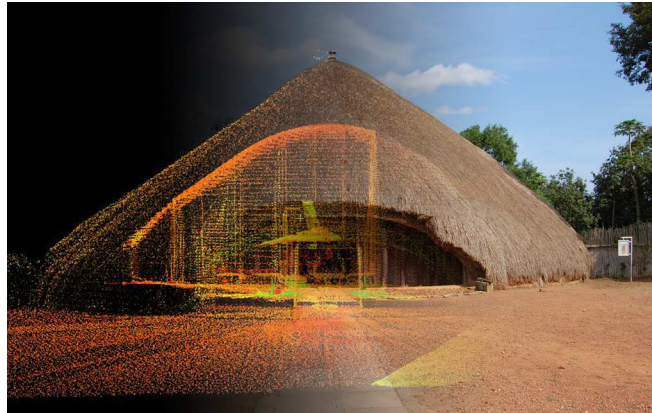


Figure 3.3: Photo overlaid atop laser scan data from a project held in early 2009 at Kasubi Tombs.

The points from the range data represent a surface, but a point is zero-dimensional; this means that it does not have volume, area, length or any other higher dimensional equivalent. Usually point clouds are not directly usable in most 3D applications because of this reason. That is why we need surfels (surface elements), that describe the surface in a small neighborhood.

At this point we need to choose how the surface will be represented by the point. In our project we will have two approaches. We will either represent the surfel as a sphere or a disc.

3.4.1 Sphere approach

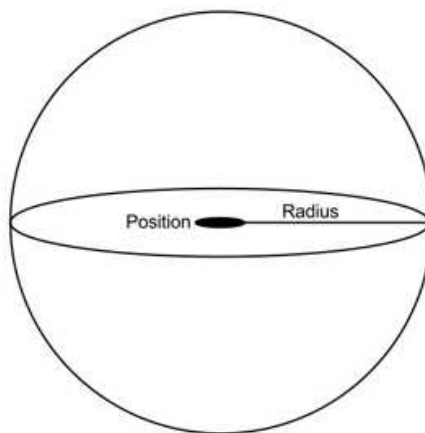


Figure 3.4: Visual representation of the surfel as a sphere.

If we choose to represent the surfel as a sphere, the surfel can have the following parameters (see Figure 3.4):

Position: x , y and z coordinates in world space.

Radius: This will be the radius of the sphere that will represent the surfel. The engine will also be capable of calculating the radius automatically.

Color: The surfel color in RGB parameters if we are going to use the lighting from the scanned data.

Material: We will specify the material's parameters in case we want the engine to compute global lighting.

3.4.2 Disc approach

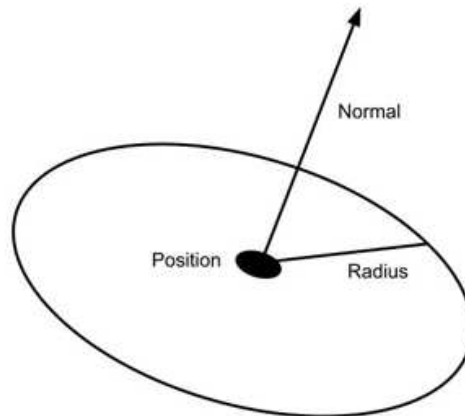


Figure 3.5: Visual representation of the surfel as a disc.

If on the other hand we choose to represent the surfel as a disc, the surfel can have the following parameters (see Figure 3.5):

Position: x , y and z coordinates in world space.

Normal: A vector that represents the surface normal of the surfel. It can be provided or the engine can estimate it.

Radius: This will be the radius of the sphere that will represent the surfel. The engine will also be capable of calculating the radius automatically.

Color: The surfel color in RGB parameters if we are going to use the lighting from the scanned data.

Material: We will specify the material's parameters in case we want the engine to compute global lighting.

3.5 Camera model

Almost everyone nowadays has used a camera and knows its basic purpose: you want to record an image of the world (usually pressing a button) and the image is then recorded on a film. One of the simplest cameras in the real world is the *pinhole camera*. Pinhole cameras use a light tight space with a hole at one end. When this hole is uncovered light enters through the hole and reaches a piece of photographic paper on the other end of the box (see Figure 3.6). In this day and age, cameras are more complex than this simple camera model, but this is a good starting point to explain how our simulation works.

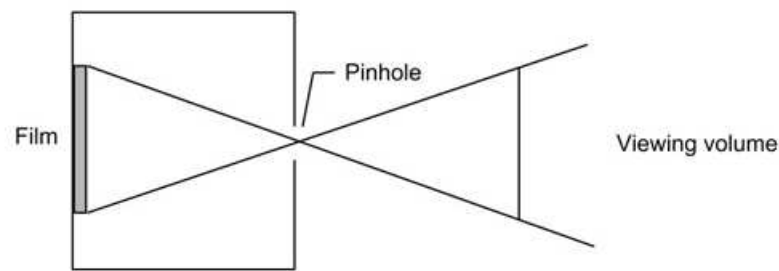


Figure 3.6: Diagram explaining how a pinhole camera works.

The most important function of the camera is defining the part of the scene that will be recorded on the photographic film. Connecting the pinhole to the edges of the film creates a double pyramid that extends into the scene. Objects that are not inside this pyramid will not be imaged on the film. As cameras are now more complex, we will refer to the region that can be imaged as *viewing volume*.

If we were to use the pinhole as the film, the viewing volume would not change. When the film or image is in front of the pinhole, the pinhole is frequently referred to as the *eye*. In our simulated camera where the film is in front of the eye, we will display the amount of light traveling from the image plane to the eye. This is where *rays* will be generated to check how much light is contributed to each point in the image (*frame-buffer*¹).

¹The portion of memory (buffer) reserved to maintain temporally an image (frame) awaiting to be sent

The camera model used in this project supports several settings. First we explain how this settings affect the camera model, after we detail how the camera model interacts with the ray tracing process. Readers can consult [PH10] if they want to delve deeper in the topic.

3.5.1 Camera parameters

The first parameter that the camera model supports is **camera position**. This parameter states where the camera will be positioned in world space coordinates (x, y, z) .

Next supported parameter is the **camera orientation**. This parameter is a point in the world at which the camera is looking at.

We also need to know how to orient the camera along the viewing direction implied by the first two parameters. The parameter **camera up vector** gives us that orientation.

We can see how this parameters are organized in camera space in Figure 3.7.

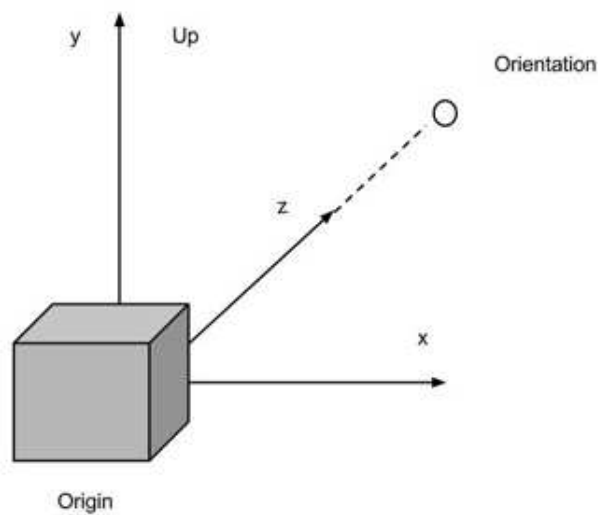


Figure 3.7: Diagram explaining the camera coordinates.

to the monitor.

Other important parameters in a camera model are the clipping planes. **Hither** dictates where the near clipping plane of the camera is located and **yon** indicates where the far clipping plane is situated as we can see in Figure 3.8. The camera's clipping planes give us the range of space along the z axis that will be visible in images. Objects that are in front of the hither plane or beyond the yon plane will not be visible.

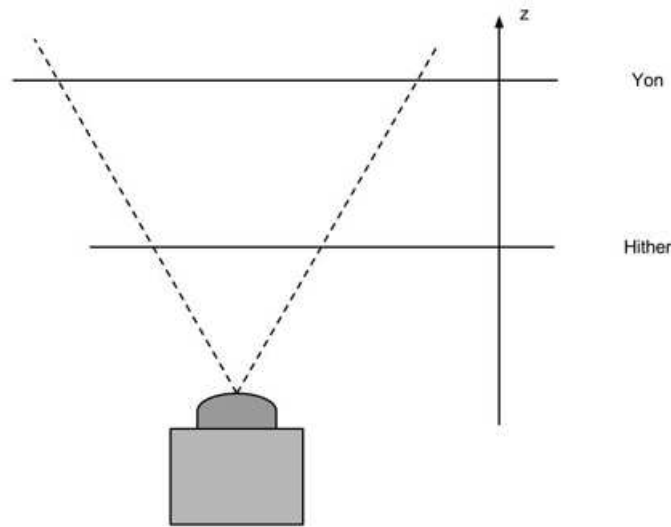


Figure 3.8: Representation of hither and yon planes.

The last two parameters that the camera supports are the **angles of view**. The angle of view describes the angular extent of the scene captured by the camera horizontally and vertically.

The clipping planes and the angles of view define the viewing volume in our model, also known as the *viewing frustum*.

3.5.2 Camera inner workings

Once we have the camera parameters we need to place the camera in the scene, and for this we will use a *look-at transformation*. All of the above parameters are given in world space coordinates, this tool gives us a transformation between world and camera coordinates.

Before we start building the frame-buffer, we will first transform all of the scene's points and normals (if provided) so calculating the rays that we are going to trace and the rest

of the computations are simplified.

As in certain contexts *frustum culling* can be useful, the engine offers the option to filter the *viewing frustum* (see Figure 3.9), although in global illumination this may not be useful, as there can be objects that influence the viewing frustum and filtering them would lead to illumination errors. The viewing frustum is the region of space that will appear in the image generated by the engine. This process of eliminating objects that are outside the viewing frustum is called *viewing frustum culling*.

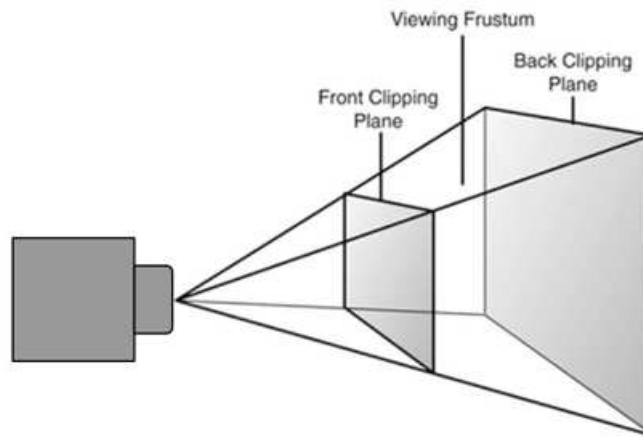


Figure 3.9: Diagram showing the viewing frustum.

After the optional filtering we have to choose if we are going to use ray tracing or just a projective camera. Our projective camera uses an orthographic model, while our ray tracing technique uses a perspective camera model.

If we use a projective camera then we will use an orthographic transformation that will project points to the far viewing plane. This type of transformation does not give the effect of foreshortening², it keeps lines parallel and preserves relative distance between objects. This transformation leaves x and y coordinates unchanged, but maps z values at the hither plane to 0 and z values at the yon plane to 1.

Once we have the adequate coordinates we just have to map them to screen space depending on the resolution of the render. If two points are mapped to the same pixel we have to see which one is closer in the z axis, because that will be the point that the pixel is going to represent.

If on the contrary we chose to use ray tracing instead of a projective camera, we have to calculate how to cast rays so that the scene will be projected on the near viewing plane.

²Objects becoming smaller on the image plane as they get further away.

This type of camera includes the foreshortening effect we mentioned before. This projection does not preserve distances or angles, and parallel lines no longer remain parallel.

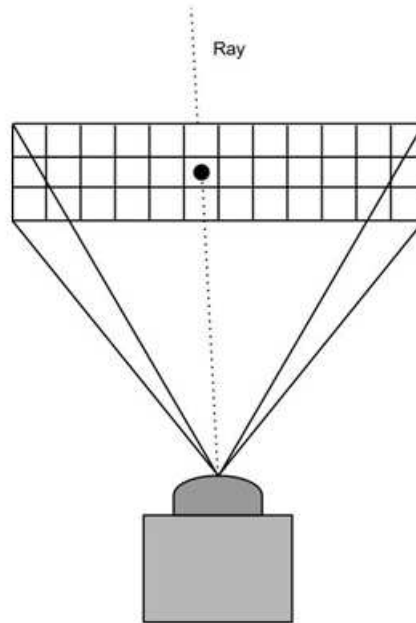


Figure 3.10: Visual representation of the process of ray casting.

To achieve this effect using ray tracing, we create rays that start at $(0,0,0)$ in camera coordinates. The direction of this rays is obtained calculating where the pixel is situated in camera coordinates, and then subtracting this point minus $(0,0,0)$. Finally we normalize the resulting vector (see Figure 3.10).

When we have one ray for each pixel, we are ready to use ray tracing to create the frame-buffer. We trace each ray through the scene and see which points intersect with the ray. Once we know the points that intersect with the ray, we need to choose the one that is closer to the near viewing plane. This will be the point that the pixel will represent.

This process implies that we have to check for intersections between each ray and the complete scene. These computations are costly when the scenes are big and to accelerate the ray-intersection process we need an acceleration structure. These structures reject chunks of the scene that the ray misses and just check for intersections in parts of the scene that the ray traverses. In our project we have chosen to use a *k-d tree* as we will see in Chapter 6.

Chapter 4

Structure of a point-based global illumination engine

In this chapter, the pipeline of BDE is shown. Thus, a high level description of the pipeline is done, which serves as a summary of the analysis phase for this project. The different stages of the pipeline are thoroughly described in the following chapters. Furthermore, this chapter includes a brief state of the art on point-based rendering and global illumination. Finally, a class diagram depicts the main design aspects of BDE.

4.1 Analysis

Figure 4.1 depicts the rendering pipeline of BDE. The first stage that the engine goes through, deals with all the input data that BDE needs to render a scene. This includes reading a configuration file in XML format, that contains information about the camera, render settings, etc. Once we know where the dataset with the point data is located, we are ready to read the file that contains the information. We can use the engine to read data imported with the BDE plugin for Blender or raw data in a format supported by the engine. The last stage of the rendering pipeline, the image creation with the resulting render, completes the I/O stage in BDE.

After creating the scene with the data read, there are a couple of optional rendering stages. We could choose to use a k-d tree for accelerating the ray tracing process or the normal estimation, in that case we would first build the k-d tree. If we decide not to use a k-d tree then this stage would not be necessary. We could want to estimate the normals if they are not provided, that is another optional phase. If we do not want to calculate the normals, we could still render the scene; but using normals provided by Blender or as a

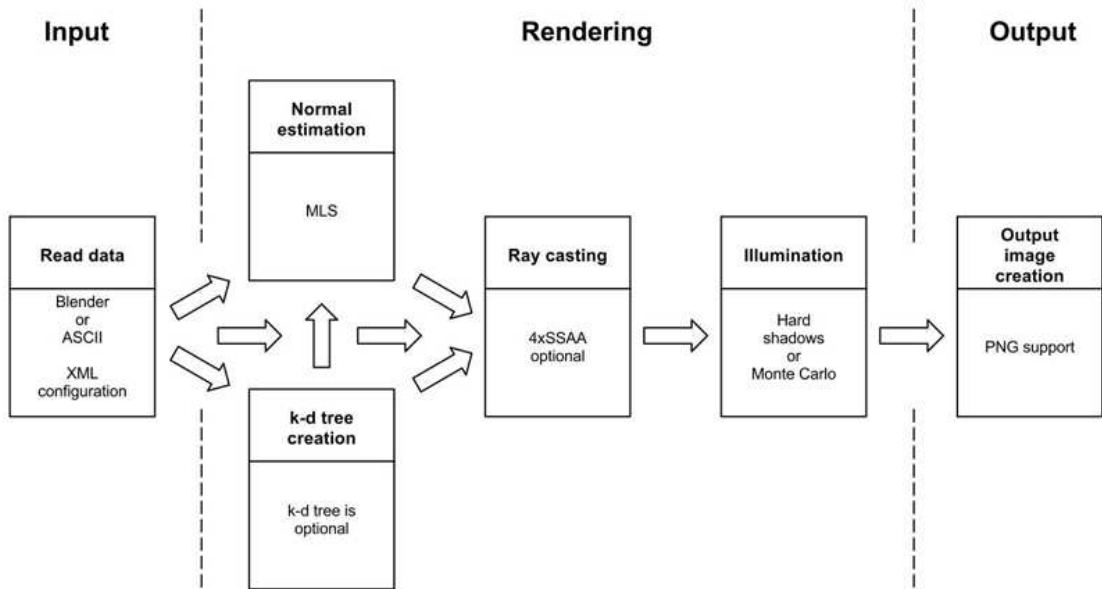


Figure 4.1: Representation of the rendering pipeline that BDE uses.

last resort use a sphere rendering technique that utilizes the surface normals of the sphere, but affects the quality of the render negatively.

Either way, after those two optional phases we need to cast the rays that we are going to need for ray tracing. In this stage we can choose to use 4x super sampling anti-aliasing (that is, casting four rays per pixel instead of one) or just render the scene normally. Right after creating the rays, we start testing them for intersections with the scene. Once this phase is completed, the pixels in the frame-buffer are shaded depending on what the rays intersected with.

In the illumination stage we have two options as well, we can choose to use a hard shadows technique to calculate the lighting or a brute force Monte Carlo approach detailed in subsection 9.3.6. It is necessary to mention that although the better way to compute illumination is the Monte Carlo technique, this is a computationally expensive process, unlike the hard shadows approach.

4.2 State of the art

Before we go any further, we have done a little survey about the current state of the art on point-based rendering and global illumination applied to point-based rendering.

Additional research of the existing proposals for the most relevant stages of BDE has been included in the corresponding chapters (Sections 7.1 and 9.3.1).

4.2.1 Point-based rendering

Point clouds are getting used more and more in movie production. One of the rendering engines that partially uses point clouds is Pixar's RenderMan [Pix12]. PRMan tessellates all surfaces into micropolygons. The algorithm can be extended to write out 3D data for each micropolygon and then each micropolygon is approximated as a microdisk (a surfel).

Point clouds can also be used to bake data [GP07]. The term *baking* is used to describe the computation and storage of data for later reuse. In the past 2D textures that required a (u, v) parameterization of the surfaces were used. Some surfaces did not have an inherent parameterization and had to be manually assigned one, this could be tedious and unintuitive. Instead point clouds have a natural, intuitive and simple parameterization; 3D position (x, y, z) . This are situations where storing data in point clouds to then obtain a render based on it can be an advantage. It is interesting to mention that PRMan does not use ray tracing to render the scene.

Another widely used alternative is Pointshop3D [ZPKG06]. It allows interactive shape and appearance editing of 3D point-sampled geometry. One key ingredient of the software is a new concept for interactive point cloud parameterization allowing for distortion minimal and aliasing-free texture mapping. It supports a great deal of different techniques to alter the appearance of 3D point models, including cleaning, texturing, sculpting, carving, filtering and resampling.

Currently the framework only allows one directional light in the scene. This is more of a point cloud editing tool than a point-based rendering engine. But it is a excellent piece of software and is a widely used tool.

We were not able to find any other popular or professional point-rendering engines, and were not able to find any that use ray tracing. This shows that the idea of this project is on the cutting edge and there are no available tools that have the same functionality.

There is a lot of research that has explored point-rendering techniques, but the use of point clouds is not yet mainstream. Among these point rendering researched techniques you can find *splatting* [LMR07] or ray tracing of point models. The most stablshed algorithm is surface splatting, although more advanced effects such as shadows, reflections, refractions, etc. are difficult to achieve using splatting. That is were ray tracing comes in handy, where high quality effects are needed.

4.2.2 Global Illumination in point-based rendering

The approaches for computing global illumination using point clouds that we have found in the literature are not based on physically-based methods such as ray tracing, so most of them do not produce the most accurate results. They compute an approximation based on the point cloud [GP07], which introduces an error; though the speed of the approximation makes up for the incorrectness. They group far away points in groups and compute ambient occlusion using spherical harmonics for representing the clusters of points from different directions.

It is important to note that this technique is not related to ours, as it does not use point-based models. They just use the point clouds to store information about polygon-based models, for example illumination data.

They first bake a point cloud of diffuse direct illumination. Then they read those data points, create an octree and represent the power for each octree node using 27 spherical harmonics. For multiple bounces they bake a direct illumination point cloud with also the diffuse colors. After this, they just compute the indirect illumination and multiply by the diffuse color and repeat the process for as many bounces as there are.

This methods just yield an approximation, since our engine is focused on computing global illumination based on ray tracing and Monte Carlo, it is slower, but allows us to achieve higher quality renders with physically-accurate illumination.

Another possibility for global illumination on point clouds is photon mapping [Jen96]. The photon mapping method consists of three stages: photon-tracing, photon map sorting and rendering.

In the photon-tracing phase, photons are emitted from the light sources and then traced through the scene using Monte Carlo Simulation. If a photon hits a diffuse surface is stored in a *global photon map*, a point cloud that contains data about the photon. The specular hits are stored in a *caustic photon map*. Finally, if the scene contains participating media, this hits will be stored in a *volume photon map*.

In the second step they are sorted into a k-d tree. The radiosity is also estimated at all or some of the positions. The result of this step is a new point cloud with radiosity data.

In the final stage, they use rays to sample the hemisphere above each point. At each ray hit point we look up the precomputed radiosity of the nearest photon with a adequate surface normal.

Although this approach would yield pretty good results, the photon map needs to be stored in memory; which is problematic when rendering complex scenes with huge photon

maps. This in BDE is not a problem, because a photon map does not need to be created to compute global illumination.

4.3 Design

Now that we have exposed the conceptual approach, we show the class diagram of the engine (see Figure 4.2). It will be further explained in the next chapters.

The class in charge of starting the application is **BDE**. Once the application is started, we need to read the data files, **Parser** serves that purpose. All BDE settings are stored in **BDESettings** and the scene description in the **Scene**, **Surfel**, **Light** and **Material** classes. The **BRDF** class makes it possible to extend our engine with more accurate and complex light transport models that calculate the reflection of light on a material. **Scene** after this step can be used to calculate **Surfel** normals or create a **KDTreeAccel** class to accelerate the rendering process.

The next step is starting the ray tracing rendering process, that is done using the **Render** class, that relies on **Ray**, **Camera** and **MonteCarlo** to render the scene provided.

To support all of this operations, we will have a math library. It uses **Point** for calculations related to point math, **Vector** for mathematical operations related to vectors and **Transform** in conjunction with **Matrix4x4** to apply transformations to points and vectors.

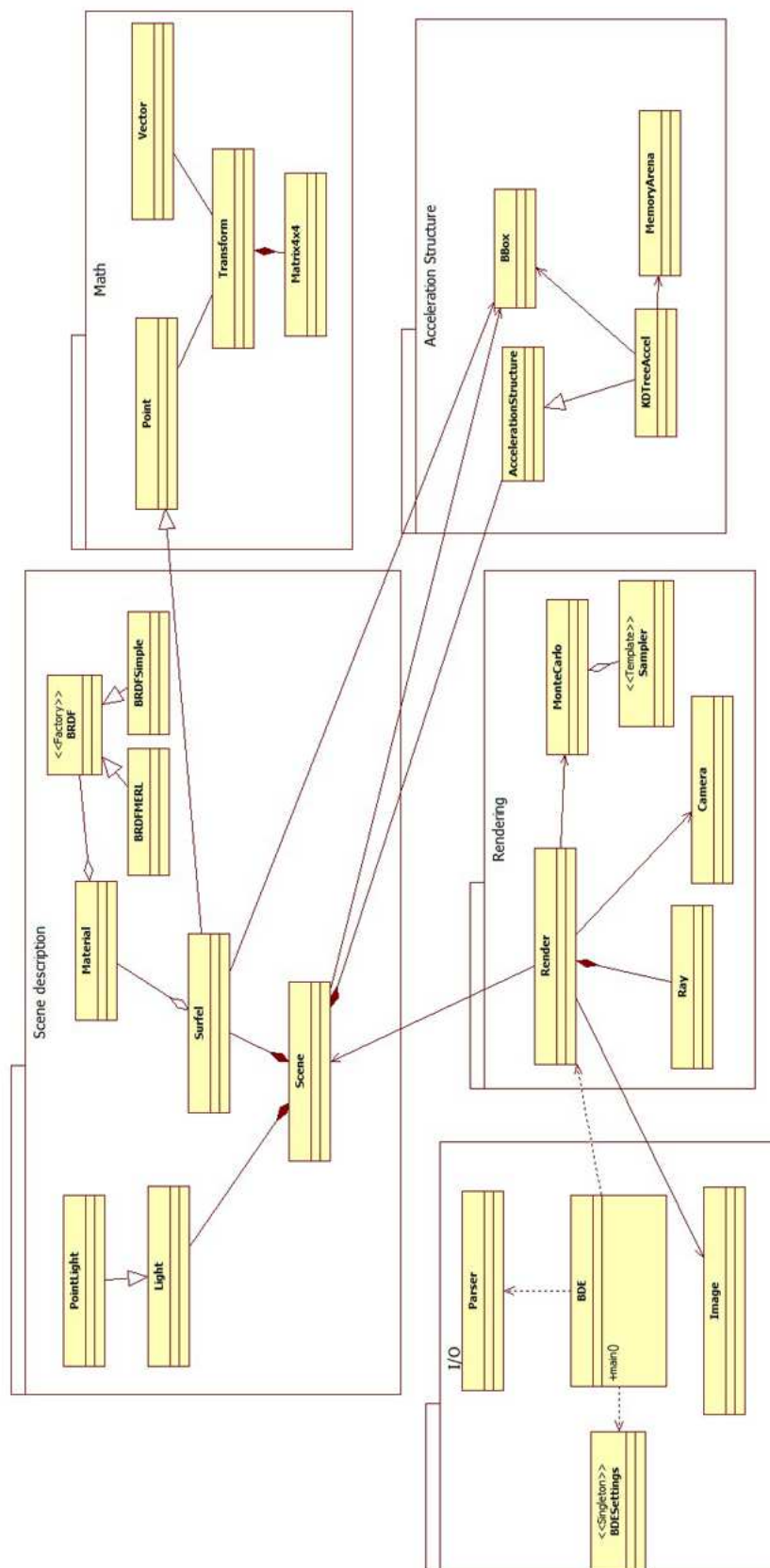


Figure 4.2: BDE class diagram.

Chapter 5

I/O and Blender integration

This chapter explains how input and output works in the engine and also describes how the engine interacts with Blender.

5.1 Input and output

BDE supports a variety of input formats for the point cloud data, and also supports XML configuration files to store the render settings, camera settings, etc. The supported output image file format for rendered images is PNG. We will further explain all the supported formats in the next subsections.

5.1.1 Design

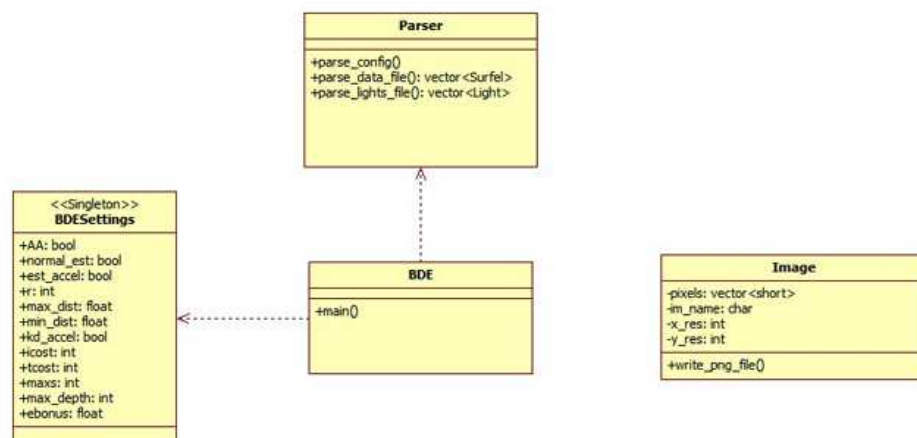


Figure 5.1: I/O class diagram.

In Figure 5.1 we can see the details of the input and output classes. The **Parser** class is in charge of parsing the XML config file, the point cloud file and the light source file. The settings are stored in **BDESettings**, so that they can be accessed whenever needed. Finally the **Image** class handles PNG image creation with the final render.

5.1.2 Point cloud input formats

The engine reads plain ASCII data files so we can intuitively see the data stored in the files. We can also make quick changes without having to rewrite the complete data file again without further problems. BDE depending on the information stored in the data file expects a certain file structure.

For distinguishing between different point cloud formats, that may evolve and include new data, we have implemented version support in the format. The data file version number will be the first element in the data file. Specifically the structure of a data file is:

Version: The data file version number.

Object transformations: The transformations that will be applied to the object. First, a line with the translation coordinates of the object: $x\ y\ z$. The next line specifies the rotation in the $x\ y\ z$ axes (value needs to be provided in radians). The following line will say how much to scale the object in $x\ y\ z$ axes. All of this parameters only have to be specified if a transformation wants to be applied to the object.

Material properties: Information about the material of the object. All of this parameters are optional too, they are just needed if we want to use ray tracing. If we wanted to just use rasterizing material properties are not needed. The first line are the ambient color's R G B parameters that range between 0 and 1. The adjacent line is the diffuse color in the same format. The following line represents the specular color also in the same format. The last line has the Phong exponent of the material and also the material's light emission parameter.

Number of points: This line contains the number of points in an object. This parameter is not necessary if we only have one object in the scene.

Surfel data: In the following lines the surfel data is stored. The surfel position in $x\ y\ z$ coordinates and the normal vector also in the same format. The normal vector is optional and does not go in the next line, it follows the position. Another optional parameter that is needed if we want to use a projective rendering approach is the point RGB color, stored in a float that contains 8 bits per channel and also goes in the same line.

We now present an example of how a point cloud data file is structured, this is one example that would have transformations, a material and finally surfel information with position and normal (see Figure 5.2).

v0.02	→	Version number
2.795753 -0.238215 -1.106424	→	Translation
-0.000000 1.483531 0.000000	→	Rotation
1.000000 1.000000 1.000000	→	Scaling
0.100000 0.100000 0.100000	→	Ambient color
0.052301 0.000000 1.000000	→	Diffuse color
1.000000 1.000000 1.000000	→	Specular color
50.000000 0.000000	→	Exponent and emission
289	→	Number of surfels
1.00000 1.00000 -0.00000 0.00000 0.00000 -1.0000	→	Position and normal
:		
:		
:		

Figure 5.2: Data file structure example.

If the user wants to use point lights, they are read from another ASCII file that only contains the light position in x y z coordinates. The engine expects one light per line.

The class **Parser** contains methods for reading the different types of point cloud files.

5.1.3 XML configuration file format

We chose to use XML as the encoding language for the configuration files, because it is human-readable, machine-readable and emphasizes simplicity, generality, and usability.

The structure of the config files is as follows:

XML declaration: Information about XML itself.

Scene tag: Information about the scene.

Camera element: This element contains information about the camera. It contains several attributes:

- p_x , p_y and p_z contain the camera position.
- d_x , d_y and d_z store the point at which the camera is looking.
- up_x , up_y and up_z contain the camera up vector.
- $hither$ and yon are the camera's near and far clipping planes.

- *angle_x* and *angle_y* store the camera's viewing angles.

Render element: The render element stores information related to the render. It has the following attributes:

- *width* and *height* are the horizontal and vertical resolution of the render.
- *surfel_radius* stores the surfel radius.
- *AA* lets us choose if we want to use x4 SSAA.
- *normal_est* and *est_accel* lets us choose if we want normal estimation and in case we do, if we want to accelerate it using the k-d tree.
- *r_func*, *max_dist* and *min_dist* control the behaviour of the normal estimation algorithm.
- *kd_accel* and *kd_nodes* let us choose if we want to use a k-d tree to accelerate the render and in case we do, establish how many surfels per leaf node we want to have.

Data element: This element tells the engine where the point cloud data is stored. It has only one attribute:

- *d_path* path to the point cloud data file.

As in the previous subsection, we now show an example XML configuration file (see Figure 5.3).

```
<?xml version="1.0" ?>
<scene>
  <camera p_x="0.876237" p_y="-11.071413" p_z="-5.343665"
    d_x="-0.142258" d_y="0.932764" d_z="0.331231" up_x="0.052929"
    up_y="0.341324" up_z="-0.938454" hither="0.100000"
    yon="100.000000" angle_x="0.857556" angle_y="0.503380"/>
  <render width="640" height="360" surfel_radius="0.140000"
    AA="false" normal_est="true" est_accel="false" r_func="1"
    max_dist="0.640000" min_dist="0.000100" kd_accel="true"
    kd_nodes="5"/>
  <data d_path="/Users/osurfer3/Dropbox/PFC/Datasets/
    custom_bde_datasets/blender.bde"/>
</scene>
```

Figure 5.3: XML configuration file example.

The class in charge of reading the XML files is **Parser**, it contains a method for reading the configuration files.

5.1.4 Image output

The supported image output is PNG. The PNG format used supports 8 bits per channel and three channels, one per RGB component. Any image viewer that supports PNG images will be able to read the images created by the engine.

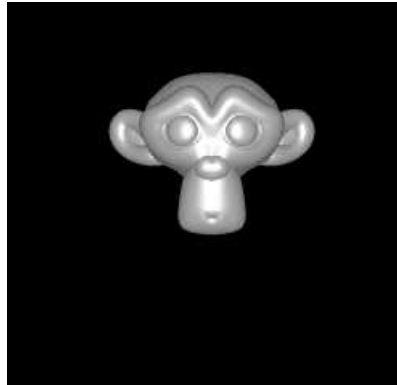


Figure 5.4: Example image, *Suzanne*, rendered in BDE.

The engine has an **Image** class in charge of creating the PNG image. It just needs a three dimensional float vector with the image pixel values and once we have that we can use the following code to create the PNG image:

```
Image *im = new Image(image_name , width , height , pixel_data );  
im->write_png_file (image_name );  
delete im;
```

The float vector needed has three dimensions, two represent the width and height of the image, and the last dimension stores the RGB values of the corresponding pixel.

5.2 Blender integration

For ease of use and to have a visual interface so camera placement, light source placement, object placement, etc. are way more intuitive; we have created a Blender plugin to make the engine compatible with this popular free software tool.

The plugin is programmed in Python and just needs to be loaded from Blender. Once we load the plugin, we will get a new Black Diamond Engine panel, in which we can configure BDE's options and when we are done press the Render button and we will render the scene using BDE instead of blender.

5.2.1 Loading the plugin

To load the plugin in blender we just need to:

1. Select the **Text Editor** among window types or press **Shift+F11**.
2. Select **Text** → **Open Text Block** and double click on the script file.
3. Click on **Run Script** button.

Once we have completed these steps we will see the BDE panel (see Figure 5.5).

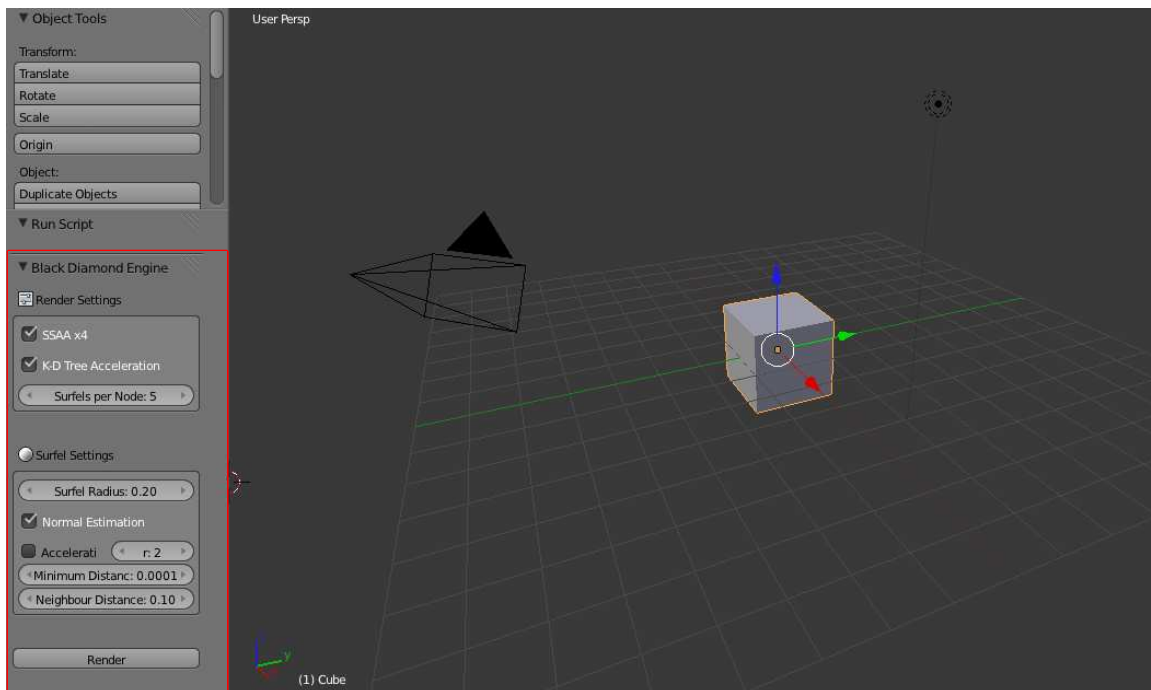


Figure 5.5: BDE plugin in Blender.

5.2.2 Using the plugin

Once we have our scene setup in Blender, we just need to adjust BDE's settings. The plugin will make sure that every object is rendered in BDE. The plugin supports:

- Meshes.
- Materials.
- Objects that emit light.

- Point lights.
- Cameras.
- Transformations (translation, rotation and scaling).

Everything will be placed exactly in the same way than in the blender scene. The plugin also extracts information about:

- Resolution in Render Presets.
- Internal camera data.
- Surface normals.

The settings in the BDE panel are pretty much self-explanatory or have been already explained in the above section, but if there is any doubt about what they do; just hovering over one will display a tooltip with a simple explanation about what it does.

The plugin will also warn the user if a material is not set for an object or if there is any other problem, with an error message.

5.2.3 Normal visualization plugin

We also provide a plugin for Blender that shows the surfel normals stored in a file. The plugin reads the file and draws the normals in blender. This tool can be used to see in Blender the normals estimated by BDE and watch them from different perspectives. This is a useful tool for testing and debugging purposes, because you can see really well if there are any anomalies in the normal calculation process.

The normals will be represented by a black line coming out of the corresponding surfel (see Figure 5.6).

5.2.4 Point cloud visualization plugin

The last plugin we offer allows us to load a simple point cloud file in Blender and represent those points as UV spheres in Blender.

This plugin is also a useful testing tool, it allows us to see the point cloud data in real time and visualize it from multiple positions without having to change camera parameters or rendering anything.

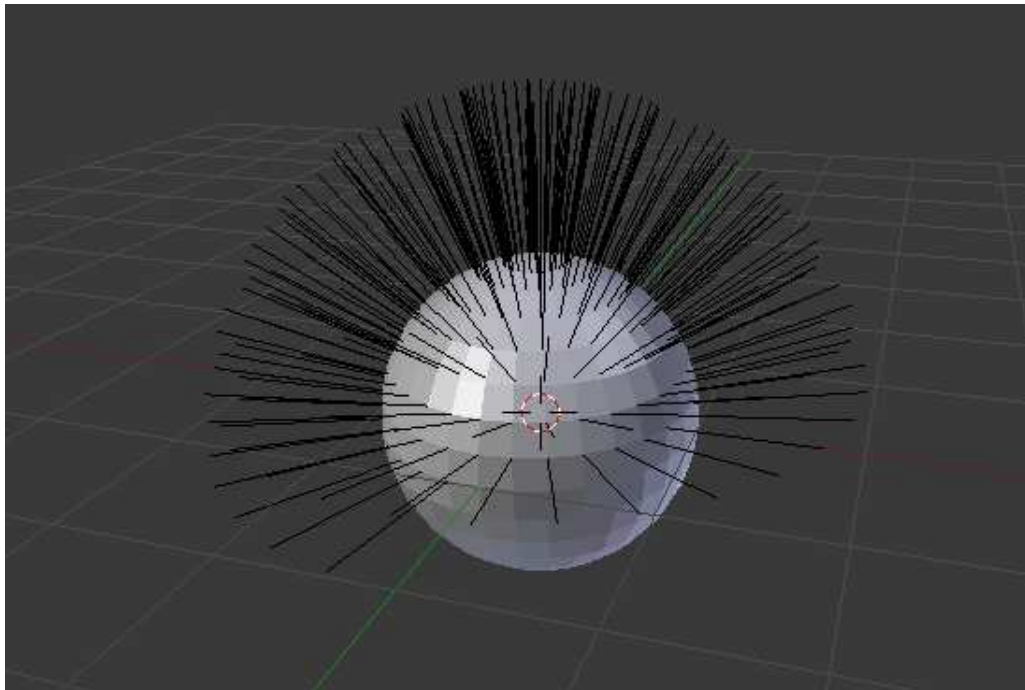


Figure 5.6: Normal visualization plugin in Blender.

Chapter 6

Acceleration structure: k-d tree

Usually, spacial acceleration structures are used in rendering to improve the performance of certain operations. In our case, the operations that we need to accelerate the most are ray-scene intersections. The acceleration of ray-scene intersections is key in global illumination as the number of ray-scene intersection tests increase exponentially. Our k-d tree is based on the implementation found in [PH10].

Binary space partitioning trees adaptatively subdivide space into regions of smaller size. A BSP tree starts out with a bounding box that encloses the whole scene. If the number of points in the scene is over a threshold, then the bounding box is split by a plane (see a 2D example in Figure 6.1). A k-d tree requires the splitting plane to be perpendicular to one of the coordinate axes; this makes both tree traversal and construction more efficient [PH10, Sam06].

6.1 Design

As we can see in Figure 6.2, the **KdTreeAccel** class handles the k-d tree creation and traversal. **MemoryArena** is in charge of making sure that the k-d tree makes efficient use of memory, so it encapsulates memory management, an essential aspect as far as memory is concerned. Finally, the **BBox** class is responsible for creating bounding boxes for several types of objects, such as k-d tree nodes or other primitives in the scene.

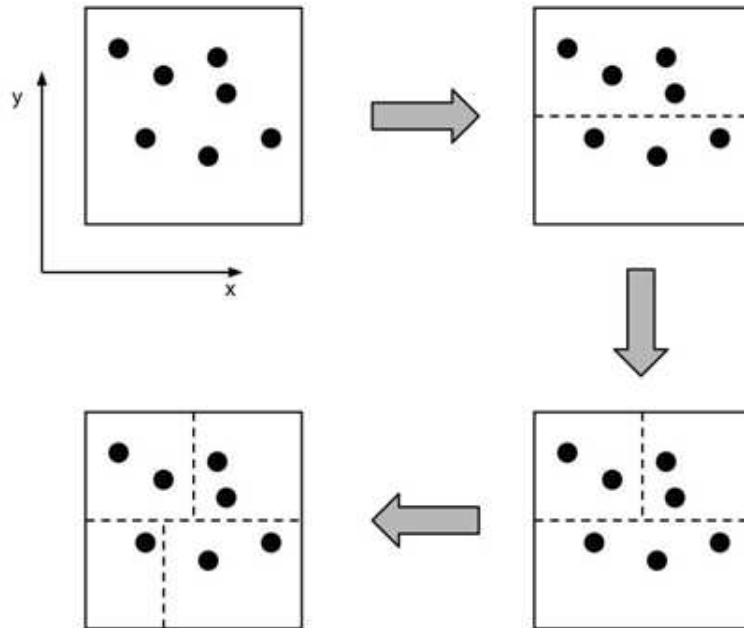


Figure 6.1: Bounding box splitting process.

6.2 Tree representation

The k-d tree is a binary tree, in which each interior node has two children and where the tree leaves store the points that overlap them. The interior nodes of the tree must provide at least the following information:

Split axis: Either x , y or z .

Split position: The position of the plane in which the axis was split.

Children: Information that indicates how to reach the child nodes.

Each leaf node just needs to know which points overlap it.

6.3 Tree construction

The k-d tree is constructed with a recursive top down algorithm. At each stage we have an axis aligned region and a number of points that overlaps that region. Then we either split the region and then make it an interior node, or we create a leaf node with the overlapping points, ending the recursion.

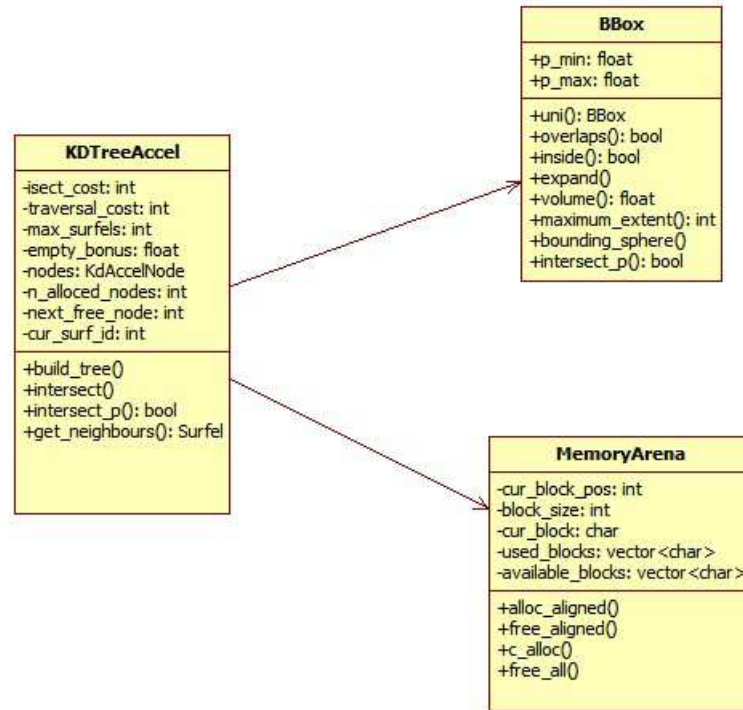


Figure 6.2: k-d tree class diagram.

We also need to obtain a maximum tree depth if none is provided. Although the building process normally ends with a reasonable depth, it is important to limit the depth of the tree so that in extreme cases the memory used will not grow indefinitely. A value of $8 + 1.3 \cdot \log(N)$ has empirically been proven to give a reasonable maximum depth for a variety of scenes in [PH10].

We create a leaf node (that stops the recursion) either if we have a small number of points in the region or if we reach the maximum depth.

If the node is an internal one, it's necessary to choose a splitting plane, classify the points with respect to the plane and recurse.

The implementation chooses a split depending on a cost model that calculates the computational expense of doing ray-intersection tests. The goal is to minimize the total cost, so the cost for each node is minimized. The cost is calculated for a series of candidate splitting planes and the one that gives the lowest cost is chosen.

The cost model intention is clear: at any node of the tree we could just create a leaf node from all the points in the region. In that case, a ray that will pass through this region will

be tested against all this points with a computational cost of

$$\sum_{i=1}^N t_i(i)$$

where N is the number of points in the region and $t_i(i)$ is the time to compute a intersection with the i th point.

Another option is to split the region. If we do that, the cost will be

$$t_t + p_B \sum_{i=1}^{N_B} t_i(b_i) + p_A \sum_{i=1}^{N_A} t_i(a_i)$$

where t_t is the time that traversing the interior node and determining which of the children the ray passes through takes, p_B and p_A are the probabilities that the ray passes through each one of the two regions below and above the splitting plane, being b_i and a_i the indices of points below and above the splitting plane, and N_B and N_A the number of points that overlap the regions below and above the splitting plane, respectively. The splitting plane affects both the probabilities as well as the series of points on each side of the splitting plane.

We will assume that $t_i(i)$ has the same value for all of the points for simplification purposes. The user has the option to set the intersection cost t_i and the traversal cost t_t . The ratio between this two parameters is what determines the way that the algorithm will behave.

It is also preferable giving a little more importance to choosing splitting planes where one of the child nodes has no points that overlap it, because rays that pass through this regions can advance to the next node without testing for intersections. After applying all these changes, the new costs are on one hand

$$t_i N$$

and on the other

$$t_t + (1 - b_e)(p_B N_B t_i + p_A N_A t_i)$$

where b_e is a “bonus” value that is worth zero unless one of the two regions has no points.

The probabilities P_B and P_A that a random ray that passes through B will also pass trough A is the division of their surface areas, s_A and s_B :

$$p(A | B) = \frac{s_A}{s_B}$$

Because we need the cost of the rays that pass through the interior node, we can use this result as is. So if we choose to split a region A into two subregions B and C, the probabilities that the ray also passes through each subregion are s_B/s_A and s_C/s_A , respectively.

Now that we have a way to calculate probabilities for the cost model, the last thing that we need solve is how to generate candidate positions for splitting and how to compute the cost for the candidates. The ideal place to split the regions is on one of the faces of the point's bounding boxes. Not taking the bounding box in account could generate artifacts when checking for intersections. To check for the best splitting plane we will project the bounding boxes onto each axis and choose the one with the lowest cost.

It may happen that no possible splits are available for a node (see Figure 6.3). In this case there would be no candidate position to split the node in any of the axes, because splitting the node would yield no benefits since each possible child node would still have the same number of points. When this happens there is no other option apart from giving up on it and making it a leaf node.

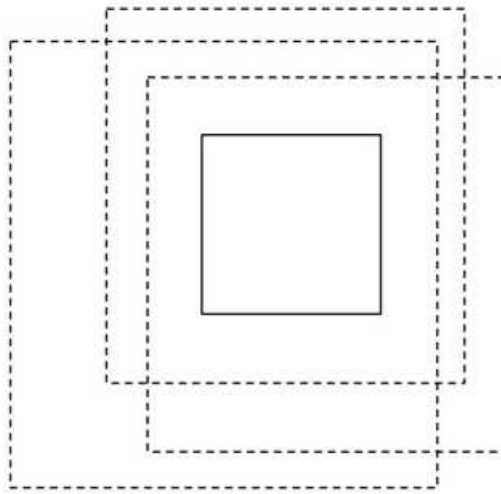


Figure 6.3: Multiple bounding boxes overlapping a k-d tree node.

Another possible case, is that the best split will have a higher cost than not splitting the node. If the cost is a lot worse and there are not a lot of points a leaf node is created. If not, the number of bad splits is stored, because sometimes it is best to split the node as at a later split may be able to find better splits given less number of points to consider.

6.4 Tree traversal

Intersecting the ray with the bounding box of the complete tree yields initial t_{min} and t_{max} values that are the ray's parametric ranges. If the ray misses the bounds, we now that the ray does not intersect with anything. If the ray intersects with the bounding box, we start the descent into the tree, starting at the root node. At each interior node it determines in which of the child nodes the ray enters first and then we process them in the correct order. This process ends when the closest intersection is found or when the ray finally exits the tree.

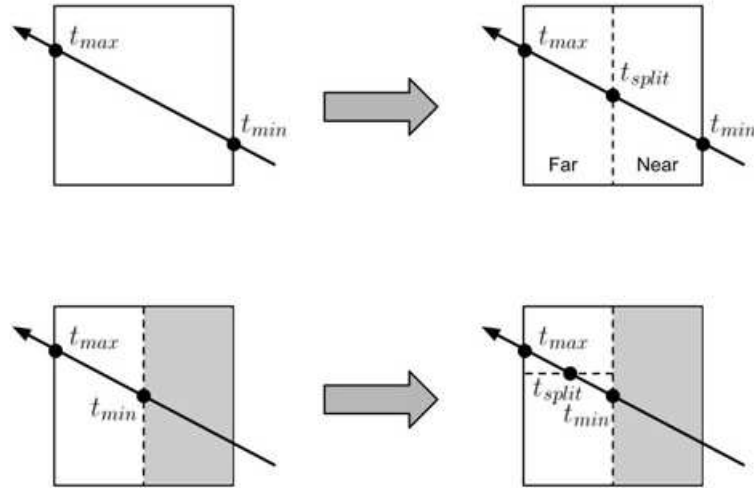


Figure 6.4: Diagram to explain visually the tree traversal process.

When the ray hits the root node, then we have to consider the two children of the root node. The ray first enters the child node labeled “near”, where the parametric range is $[t_{min}, t_{split}]$. If the node labeled “near” is a leaf we perform the intersection tests if necessary, if not we process its child nodes. If no hit was found in the node, or if we were to find a hit beyond $[t_{min}, t_{split}]$, then the node labeled “far” would be processed. This process continues until processing tree nodes in a depth-first front-to-back traversal is complete and the ray exits the tree or the closest intersection is found as we mentioned before (see Figure 6.4).

6.5 Performance tests

This section presents some performance tests that evaluate the effectiveness of the k-d tree acceleration structure. All the tests were performed on a single thread of a Intel Core i5-2557M CPU and 4 GB of 1333 MHz DDR3 RAM.

In the 3D graph of Figure 6.5, we represent surfels and rays versus time. In this graph we can see that the k-d tree does a marvelous job, even as we augment the surfels exponentially. It only shows a big increase in rendering times for the case with the most rays and over 500k surfels, and this is because to keep the k-d tree creation times around an acceptable value, for this last case we increased the number of surfels per node in the k-d tree from 10 to 100.

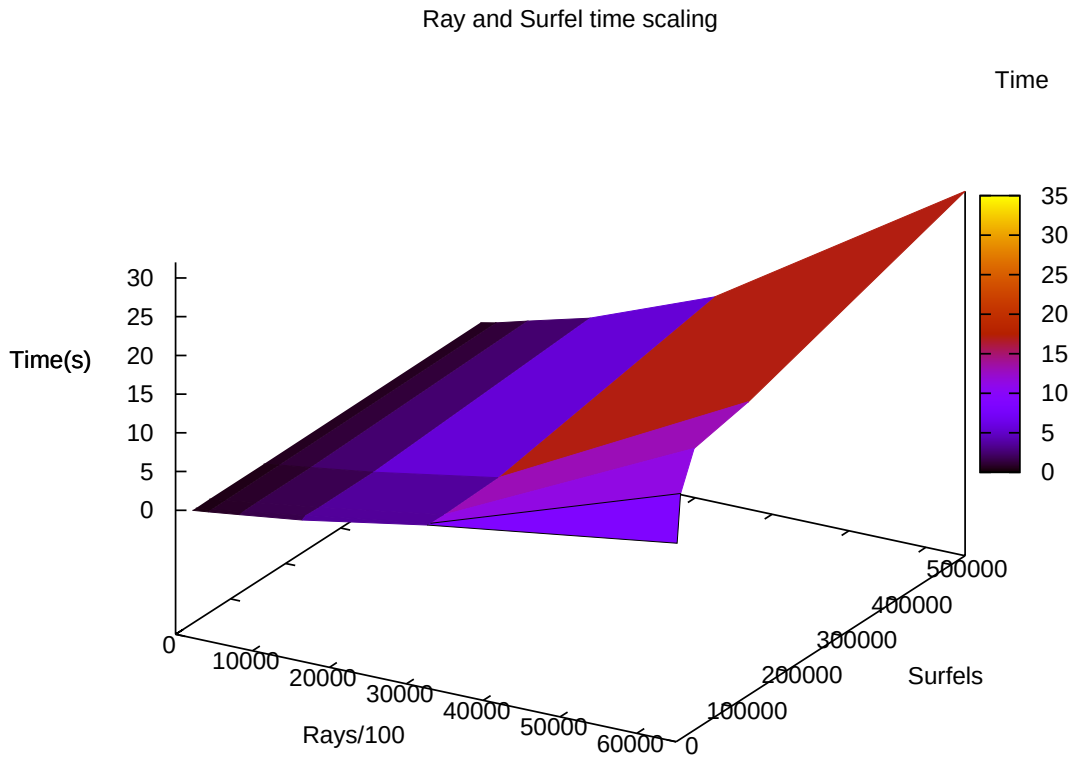


Figure 6.5: 3D plot that shows how surfels and rays affect the rendering times.

The lesson that we learn from this graph, is that the number of surfels does not affect the rendering time very much thanks to the k-d tree. What really adds up to the rendering time is the number of rays traced, this means that as you increase the resolution, rendering times go up. This shows us that theoretically how many surfels are in the scene should

not influence the rendering times, only the k-d tree creation time.

It is important to know that the number of surfels per node will ultimately dictate the number of intersection test that we have to perform, so the less the better.

In the next plot, we can see the performance improvement achieved with the k-d tree (see Figure 6.6).

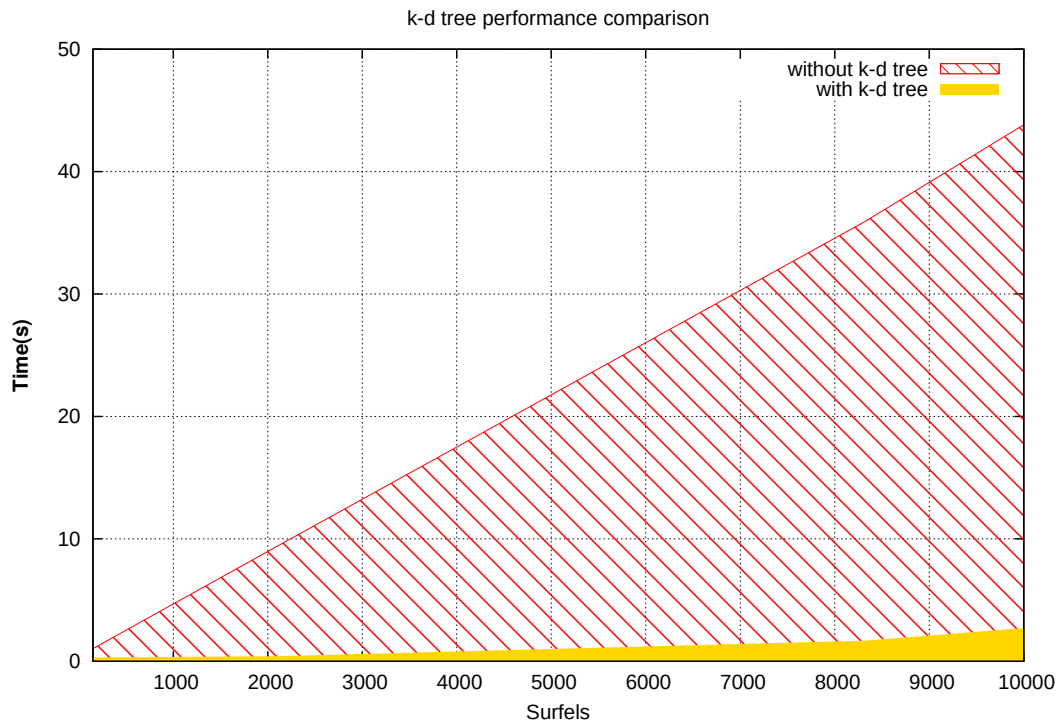


Figure 6.6: Plot showing the time it takes to render a scene with or without a k-d tree.

The time reduction is remarkable, with a k-d tree the time it takes to render the scene is really low, and the tree traversal time remains almost constant as you increase the number of surfels. Only the memory the k-d tree occupies and the construction time slightly increase. While not using the k-d tree leaves us with huge rendering times that augment steadily as you increase the number of surfels.

Surfels	Time (s)	
	with k-d tree	without k-d tree
148	0.2937	1.008
548	0.3268	2.737
2116	0.4045	9.458
8324	1.6801	35.942
33028	16.8858	152.141

Table 6.1: Table showing the data involved in Figure 6.6.

Chapter 7

Normal estimation

When using realistic illumination models, surface normal vectors of all the objects in the scene are needed. Although in polygon-based rendering this is trivial, when working with point clouds without topology it presents a problem. Therefore, if normal vectors of the surfels are not part of the input dataset (obtained from the modeling process) we will have to estimate them.

In this chapter we address the challenge that obtaining the normal vectors of a point cloud presents, this in essence is the same as estimating what points belong to the same surface. We present a brief state of the art and the approach included in BDE.

7.1 State of the art

There are two main techniques used for normal estimation in point clouds. A *numerical* approach using optimization [GP07] and a *combinatorial* method that uses a Delaunay/Voronoi property [NM99]. Numerical methods normally work well in the presence of noise. Combinatorial methods traditionally have not worked well with noise, but there have been some advances in that regard [DLS05].

We have chosen the numerical approach because we want to be able to use noisy datasets. The theory and implementation behind our method will be explained in the next sections.

Plane fitting methods. The normal in each point is estimated based on the plane obtained by applying a least squares method to the k -nearest neighbors of the corresponding point. This method can be improved giving more importance to the closest neighbors. Another improvement can be not using the same k for all the points. As a similar method

to this one is used in our engine and it is thoroughly explained in the next section, we will not delve deeper in this technique.

Big Delaunay ball method. For a group of points P , being V the Voronoi diagram and D the dual Delaunay triangulation of P . The voronoi cell for a point p will be V_p . The line that unites p and the furthest vertex in V_p (its *pole*) can be used to estimate the normal at p and even an orientation. This fact will not be true with noisy samples, that's where the Delaunay ball comes in.

A ball is called Delaunay if it has a center at a Voronoi vertex v and a radius $\|v - p\|$ where $v \in V_p$. The balls with poles situated in its center are called *polar balls*.

If p is in the boundary of a polar ball B the line that joins p and the center of B estimates the normal at p . It has been observed that if there is some noise, certain Delaunay balls will remain relatively big and can be considered polar balls. Then we can redefine the pole for a point p as the furthest vertex of the V_p whose Delaunay ball is big. The normal can be estimated from the segment through p and its pole.

This algorithm is called Big Delaunay Ball algorithm or BDB.

7.2 Moving Least Squares for normal estimation

If normals are not part of the dataset provided, we can try to estimate them with the following method that is based on a numerical optimization approach [GP07]. We assume that we need to calculate the normal $\mathbf{n}$ in a location $\mathbf{q}$. Then the points that are close to $\mathbf{q}$ will describe the surface around the point $\mathbf{q}$. So a tangent in $\mathbf{q}$ should be as close as possible to this surrounding points, and the normal will be perpendicular to this tangent plane. To determine a tangent plane in $\mathbf{q}$, we can formulate it as a least squares problem (see Figure 7.1).

We are searching for a plane that yields a normal $\|\mathbf{n}\| = 1$ passing through $\mathbf{q}$ that minimizes the squares $(\mathbf{n}^\top(\mathbf{q} - \mathbf{p}_i))^2$. But as we only want to consider points $\mathbf{p}_i$ that are close to $\mathbf{q}$, we will use a weight function θ . Once we incorporate the locally weighted function then $\mathbf{n}$ is defined by the next minimization problem:

$$\min_{\|\mathbf{n}\|=1} \sum_i (\mathbf{n}^\top(\mathbf{p}_i - \mathbf{q}))^2 \theta(\|\mathbf{p}_i - \mathbf{q}\|).$$

This would be a non-linear optimization problem, because of the constraint $\|\mathbf{n}\| = 1$. To be able to get a computable solution, we use the matrices of the outer product $(\mathbf{p}_i - \mathbf{q}) \otimes$

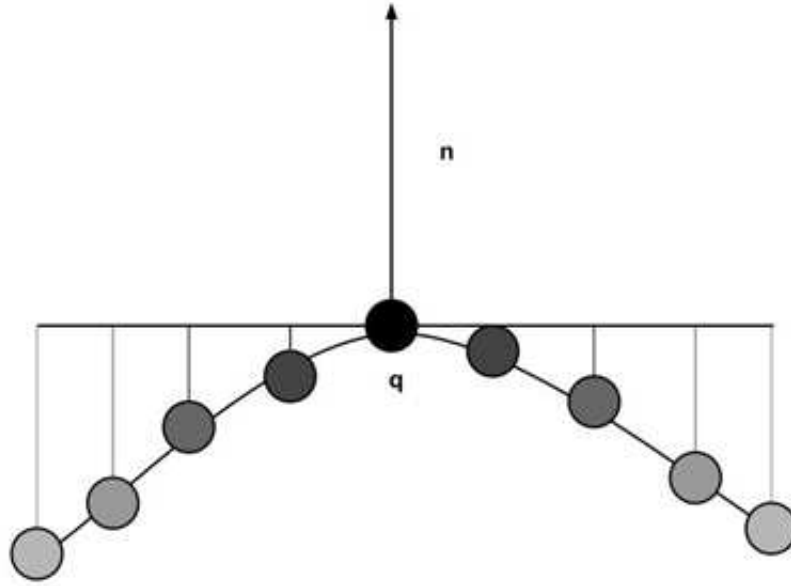


Figure 7.1: A unit normal that is orthogonal to the plane that minimizes the square distances to the points is calculated.

$(\mathbf{p}_i - \mathbf{q})^\top$. Then we can rewrite the function that has to be minimized as

$$\mathbf{n}^\top \left(\sum_i (\mathbf{p}_i - \mathbf{q}) \otimes (\mathbf{p}_i - \mathbf{q})^\top \theta(\|\mathbf{p}_i - \mathbf{q}\|) \right) \mathbf{n}$$

and then compute the eigenvalues/eigenvectors of this resulting matrix, the eigenvector corresponding to the lowest eigenvalue will be our best estimation for the normal.

The chosen weight function is $\theta(\delta) = \delta^{-r}$ that for a $r > 0$ will give more importance to the neighbors closer to the point in which we want to calculate $\mathbf{n}$.

It is important to note that this method gives us a normal direction, not an orientation. The orientation will have to be given or estimated.

7.3 Algorithm: design and implementation

The class in charge of normal estimation is **Scene** (see Figure 7.2). There are two methods that take care of estimating normals: `get_normals_accel()` and `get_normals()`. The former uses k-d tree acceleration for neighbour search, whereas the latter does not.

The algorithm still acts the same way in both methods, the only thing that changes is

how we obtain the neighbours.

For all surfels in the scene we obtain their neighbours inside a maximum distance. Then we calculate the sum of outer product matrices (resulting from the distance vector between point and neighbour) multiplied by the corresponding weight θ . There is one outer product sumatory matrix per point.

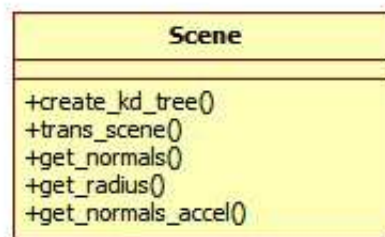
Once we are done and we have the resulting matrix M , we need to solve the equation $|M - \lambda I| = 0$. This is a third degree polynomial in λ . The roots of this polynomial will be the eigenvalues of the matrix.

Once we have the three resulting eigenvalues, we will use the smallest one to calculate the corresponding eigenvector that will be our estimated normal.

To accomplish this, we will solve $Mv = \lambda v$. This system of equations will provide us the eigenvector v . This eigenvector will be the normal in the corresponding point.

Finally we will show the pseudocode for the normal estimation process:

```
for(int i = 0; i < cloud.size(); i++) {  
  
    for (int j = 0; j < neighbour.size(); j++)  
        M += DIST * theta;  
  
    det = determinant_calculation(M - lambda);  
  
    eigenvalues = find_real_roots(det);  
  
    eigenvalue = min(eigenvalues);  
  
    normal = solve_eigenvector_system(eigenvalue);  
  
}
```

Figure 7.2: Class diagram of the **Scene** class.

Chapter 8

Ray tracing

According to Figure 4.1, once the input point cloud dataset has been read and the normal vectors for all the surfels in the scene are available, we are ready to start tracing rays and check with what objects they intersect in the scene. To know what a ray intersects with, we have to run a series of disc or sphere intersection tests.

The k-d tree accelerates this process greatly, but as it can still be demanding depending on the resolution, we have used threadsafe code so it can make use of all the threads available in the system running the engine, this makes it possible to exploit current systems with multi-core processors.

In this chapter we first include a brief introduction about ray tracing. Then we lay out the design of the classes involved in ray tracing. Next the process of ray-scene intersection is detailed. Once we know what a ray has intersected with in the scene we need a technique that lets us know if an object is shadowed, our technique is also explained in this chapter.

Some problems can arise with a render, like aliasing or surfel overlapping when the model is not optimal. To deal with overlapping problems we have devised a technique based on a weighted mean that we also detail in this chapter. Lots of techniques exist to solve aliasing problems, we describe the approach that we used that employs supersampling anti-aliasing. We finally offer some performance testing in which we can see how ray tracing performance varies when using multithreading.

8.1 Introduction

In computer graphics, ray tracing is a method that generates an image by tracing the path that light follows through pixels in an image plane and simulating its behaviour when it

interacts with virtual objects (see Figure 8.1).

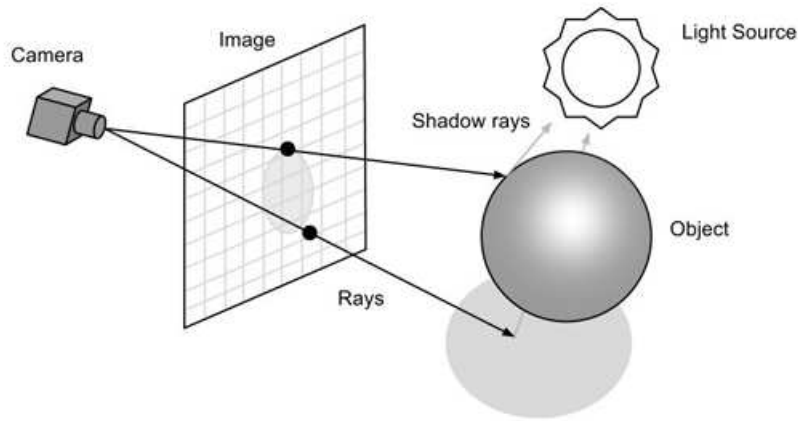


Figure 8.1: Image that shows how the ray tracing algorithm works.

This technique is capable of producing realistic images, usually more physically accurate than scanline rendering methods, but at a higher computational cost. This is possible due to the fact that the algorithm is capable of simulating light behaviour in a more natural way. Ray tracing can be used to simulate several optical effects, such as reflection, refraction, etc.

We normally trace a path (ray) from an imaginary eye (the camera) through each pixel of the image that is going to be created, and then calculate the colour that would be visible through it. Typically each ray must be tested for intersection at least with a subset of the objects present in the scene. Once you identify the nearest object, the algorithm estimates the incoming light at the point of intersection, the material properties of the object and combines these two to provide a final color for the corresponding pixel. Certain illumination algorithms (i.e. Monte Carlo integration) and reflective or translucent materials would require more rays to be cast across the scene. Ray tracing makes this really easy because of the nature of the algorithm, that is inspired in how light behaves physically. For more information on ray tracing methods please refer to [PH10].

8.2 Design

As can be seen in Figure 8.2 **Render** is the most important class in the rendering process, it is in charge of creating the rays with **Ray**, checking for intersections and shading the pixels. **Camera** handles all the operations related to the camera in **Scene**. This last class contains the scene description. A scene is composed of **Lights**, **Surfels** and their

corresponding **Materials**.

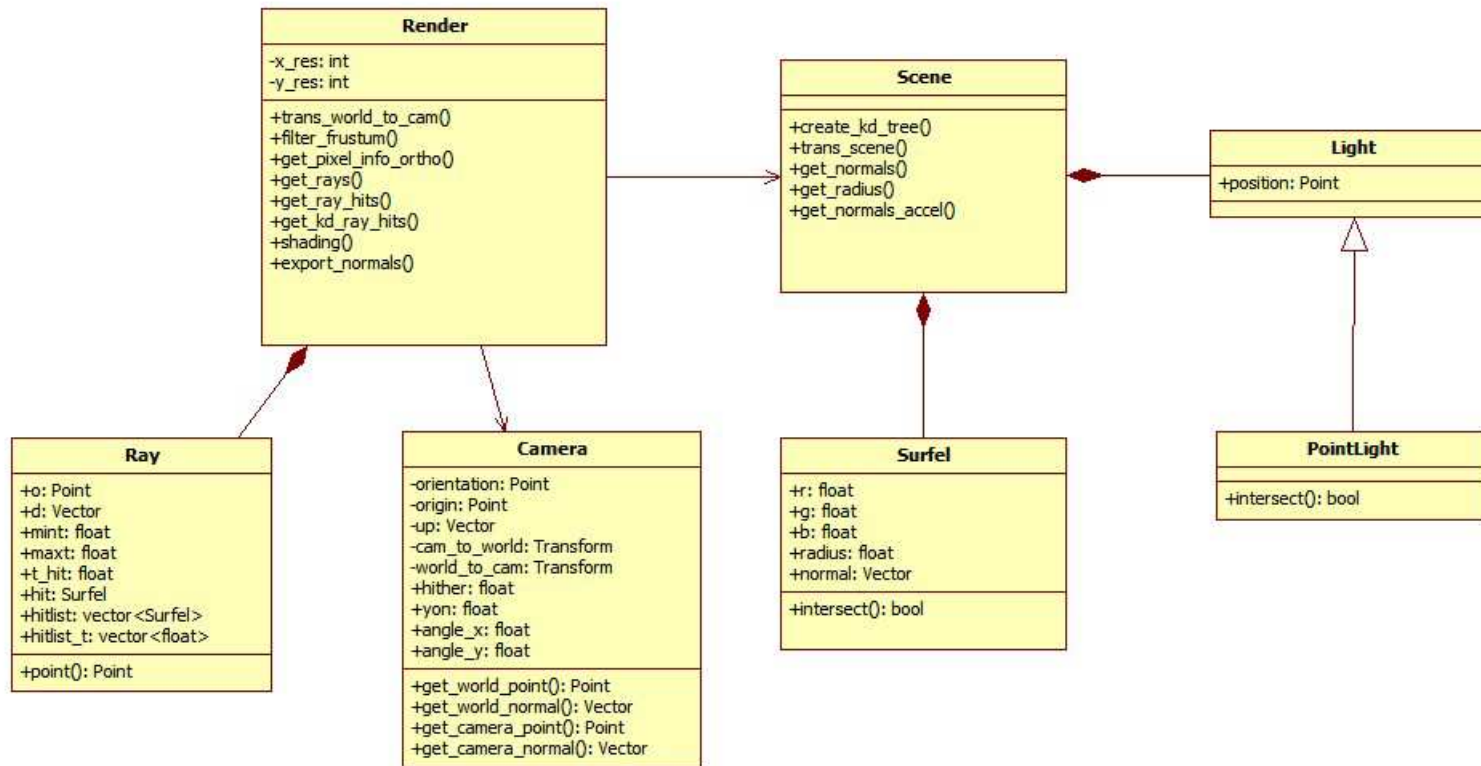


Figure 8.2: Class diagram of the elements involved in ray tracing.

8.3 Ray-scene intersection

When using ray tracing, we have to know what do rays intersect with in the scene. We have two types of intersections:

Ray-sphere intersection. Being d the ray direction, o the ray origin, $[t_{min}, t_{max}]$ the ray's parametric range, p the surfel's position and r the surfel's radius. To know the intersection point between the ray and the sphere with center p that will have the following equation:

$$(x - p_x)^2 + (y - p_y)^2 + (z - p_z)^2 = r^2$$

substituting the ray in the equation we just get a quadratic equation that we solve for t , after simplifying the equation we have

$$t = \frac{-B \pm \sqrt{B^2 - AC}}{A}$$

being

$$B = (o - p) \cdot d$$

$$C = (o - p)^2 - r^2$$

$$A = d^2 = 1$$

because the ray direction is a unit vector, $d^2 = 1$ and the quadratic equation ends up being

$$t = -B \pm \sqrt{B^2 - C}$$

and this equation is easily solved. If there are no real roots the ray will miss the sphere. If there are roots we will select the smallest one, because that will be the closest hit on the sphere. Once we have the closest hit we just have to check if it is inside the parametric range $[t_{min}, t_{max}]$ of the ray for it to be a valid hit.

Ray-disc intersection. Being d the ray direction, o the ray origin, $[t_{min}, t_{max}]$ the ray's parametric range, p the surfel's position, r the surfel's radius and n the surfel normal. To know the intersection point between a disc and the ray we will perform the following calculations.

First we will check if the ray intersects with an infinite plane orthogonal to the surfel's normal. We check for that intersection first writing the plane in vector notation

$$(p_0 - p) \cdot n = 0$$

and the ray also in that same notation

$$p_0 = o + td.$$

We substitute the ray in to the plane equation and get

$$t = \frac{(p - o) \cdot n}{d \cdot n}$$

then we solve for t . If there is no intersection, the above denominator will be 0. In any other case the ray intersects the plane once and the intersection will be t .

Once we know that the ray intersects with the plane, we just have to check that the distance to the position of the surfel is not greater than r . We do that using the following comparison

$$\|r(t) - p\| < r$$

being $r(t)$ the point obtained from substituting t in the ray's parametric equation.

8.4 Shadows

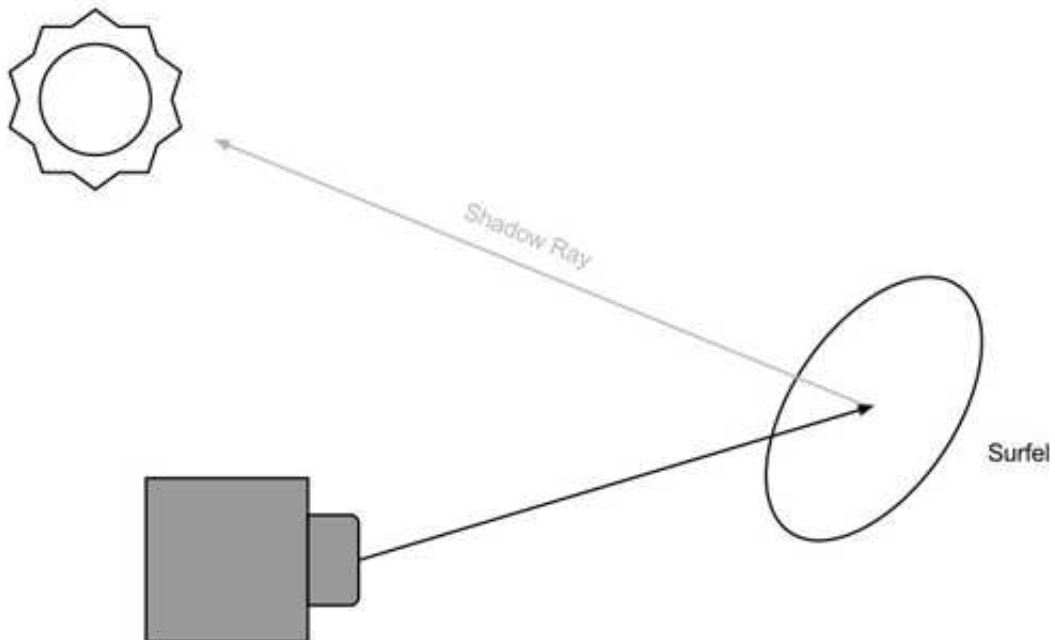


Figure 8.3: Diagram explaining how a shadow ray works. In this case there is nothing occluding the path between the object and the light source.

Once we know where the ray intersects an object, we need to know if that object receives light or is blocked by another one. To know this we trace a shadow ray from the object's hit point to the light source. If there is no hit in the parametric range of the ray, light is not blocked (see Figure 8.3). If there was any hit between the light source and the hit point then the path is obstructed by another object.

This lighting model does not consider indirect paths to the light sources, that phenomena will be considered in the global illumination model.

8.5 Surfel overlapping

Sometimes when surfels are too close together or have too much radius, they overlap each other. If the surface is not curved, it is not a problem; but when the surface is curved the overlapping produces a non visually pleasing effect (see Figure 8.4).

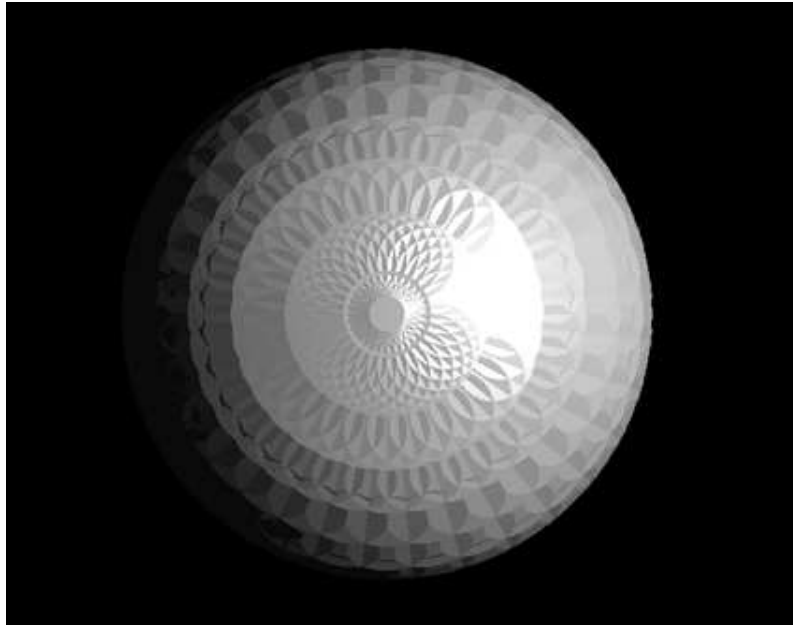


Figure 8.4: A sphere with lots of overlapped surfels.

This kind of effect can be countered using a weighted average when shading the pixel that represents the overlapped surfels (see Figure 8.5).

To achieve this effect, we calculate

$$\overline{C} = \frac{\sum_{i=1}^N C_i \cdot w_i}{\sum_{i=1}^N w_i}$$

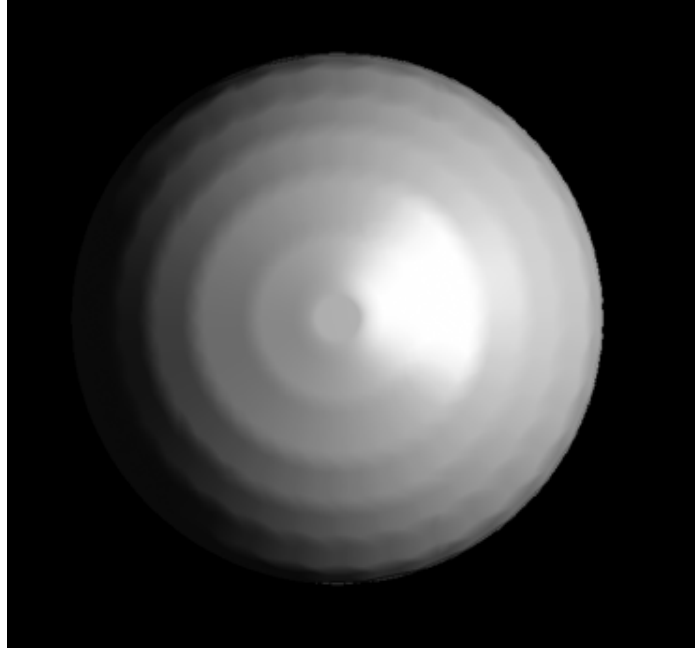


Figure 8.5: The same sphere of Figure 8.4 but rendered with our technique.

where N is the number of surfels that overlap, C_i is the color of each surfel, P the hit point, d_i the distance between P and the center of the surfel, r_i the radius of the surfel and

$$w_i = \left(1 - \frac{d_i}{r_i}\right).$$

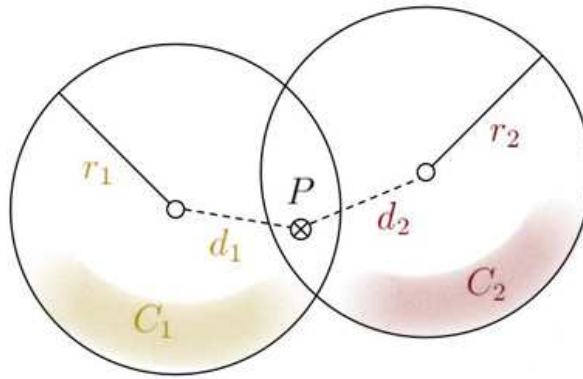


Figure 8.6: Visual representation of the terms in the weighted average.

To better understand the technique, Figure 8.6 offers a visual representation of all the terms involved in the calculation of the weighted mean.

$\overline{C}$ will be the final color of the corresponding pixel, this will make the surface smoother and decrease the artifacts caused by surfel overlapping. It should be noted that if the model has constant point density, this technique should not be necessary.

8.6 Supersampling anti-aliasing

There are a lot of techniques that we can use to reduce aliasing. We will describe our current anti-aliasing method that is based on supersampling.

Supersampling is the idea of taking more individual color samples for each pixel so we can reduce aliasing problems, like for example stair effect on borders. In our method, we will take four samples instead of one per each pixel and set the pixel color to the average of the four samples. This will be effectively *4x supersampling*, because we are computing four times more rays for each pixel.

We present the following pseudo code to show how we can do this. For each pixel we have to launch four rays, but each ray will only contribute a quarter.

```
for (int i = 0; i < resolution; i++){

    float rgb=0;

    if (sub_pixel_rays.intersect()) {

        shading(sub_pixel_rays);

        rgb += 0.25 * sub_pixel_rays.rgb;

    }

    return rgb;

}
```

8.7 Performance tests

The ray tracing process in the engine is programmed so that the use of multithreading is optional. If the user wants to use multithreading, he would just need an OpenMP compatible compiler. The engine will fully make use of all threads available, not only for the ray tracing process but other computationally expensive parts of the engine. The test was performed on a Intel Core i5-2557M CPU and 4 GB of 1333 MHz DDR3 RAM.

We now proceed to show a graph that will compare the use of multithreading to using a single thread in the ray tracing stage (see Figure 8.7), this is the most computationally expensive part of the rendering pipeline and where multithreading will shine.

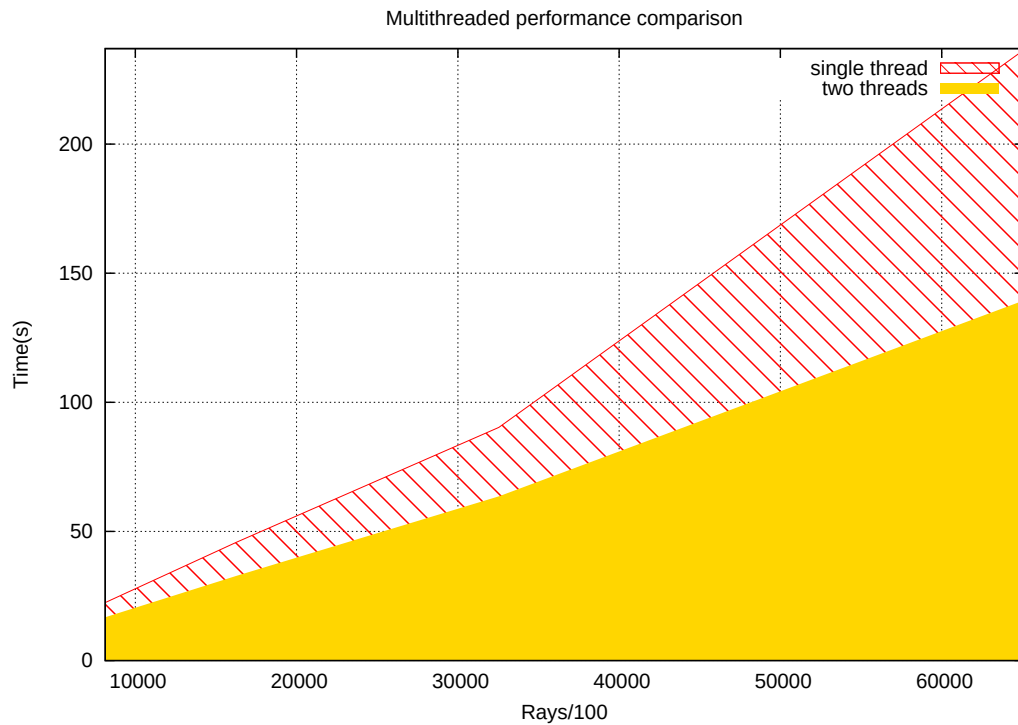


Figure 8.7: Plot that shows the time it takes to ray trace a scene with and without multithreading.

As we can observe in the plot, just with one more thread there is a big gap in the rendering times. As we increase the number of rays to trace, the difference between single and multithread deepens. Taking into account that the number of rays to trace using Monte Carlo augments exponentially, the use of multiple threads will be even more important and the performance gap will be huge.

As you increase the number of threads, with a high number of rays to trace; performance will improve greatly, there is no doubt that the use of multithreading is key in this stage

of the rendering pipeline.

Chapter 9

Global illumination

Although the computation of direct shadows in a scene may offer a certain realistic look, it is the use of physically-based global illumination models that allows us to achieve highly realistic 3D images. These kind of methods try to deal with the real behaviour of light in an environment, taking into account not only *direct illumination*¹ but also *indirect illumination*² to obtain a physically accurate result.

Reflections and shadows are part of global illumination, when we simulate them, objects have an effect upon one another (unlike when affected by only a direct light). Images rendered using global illumination techniques normally appear more realistic than using direct illumination algorithms. At the cost of being much more computationally expensive.

Radiosity, ray tracing, beam tracing, cone tracing, path tracing, Metropolis light transport, ambient occlusion, photon mapping, and image based lighting are algorithms used in global illumination, some of them can be used together to generate more accurate images.

In this chapter we present how our engine applies ray tracing to simulate light transport in a 3D virtual scene. As commented in the previous chapter, ray tracing is a method especially well fitted for simulating light transport, since lots of different effects may be incrementally implemented. Our engine includes a simple local illumination algorithm, as well as a Monte Carlo based global illumination approach. Quality and performance results are discussed.

¹Light that impacts the object directly from a light source.

²Light from the source that is reflected by other surfaces in the scene and after that gets to the object.

9.1 Shading

In computer graphics, shading is in charge of altering the color of an object in 3D space to create a realistic effect, mainly depending on the distance and angle to light sources. Shading is performed during the rendering process, that's why the class that takes care of shading is **Render**. Shading is calculated for each pixel based on what a ray intersected with, the lights in the scene, etc. It provides a pixel color that will later be used to create the rendered image.

BDE supports two types of shading. We can either use Monte Carlo or hard shadows with Phong shading. The first technique is vastly more time consuming depending on the number of samples chosen to solve the lighting integral. The second is way faster and yields pretty good results. Although the best quality will be provided by Monte Carlo, this high quality comes at a cost, the render will take a lot more time.

9.2 Ray tracing with hard shadows and Phong shading

If we were to use this technique, the first step is checking which of the camera rays intersect with something in the scene. If the ray intersects with an object, then we have to shade the corresponding pixel. If it does not intersect with anything, we just leave the ambient color.

```
for(int i = 0; i < rays.size(); i++){  
  
    if(ray_intersects()) {  
        pixel = shading();  
    } else {  
        pixel = ambient_color();  
    }  
  
}
```

To compute the shading, we first need to know if there is an object blocking the path to the light as we saw in Section 8.4.

```
for(int i = 0; i < lights.size(); i++){  
  
    if(light_occluded()) continue;  
  
    compute_diffuse_and_specular_reflection();  
  
}
```

As we see in the pseudo code, if the light source is unoccluded, we have to compute the specular and diffuse contributions of the corresponding light source. These two steps are explained in the next two subsections. Figure 9.1 shows an example of how a render using this simple shading looks like.

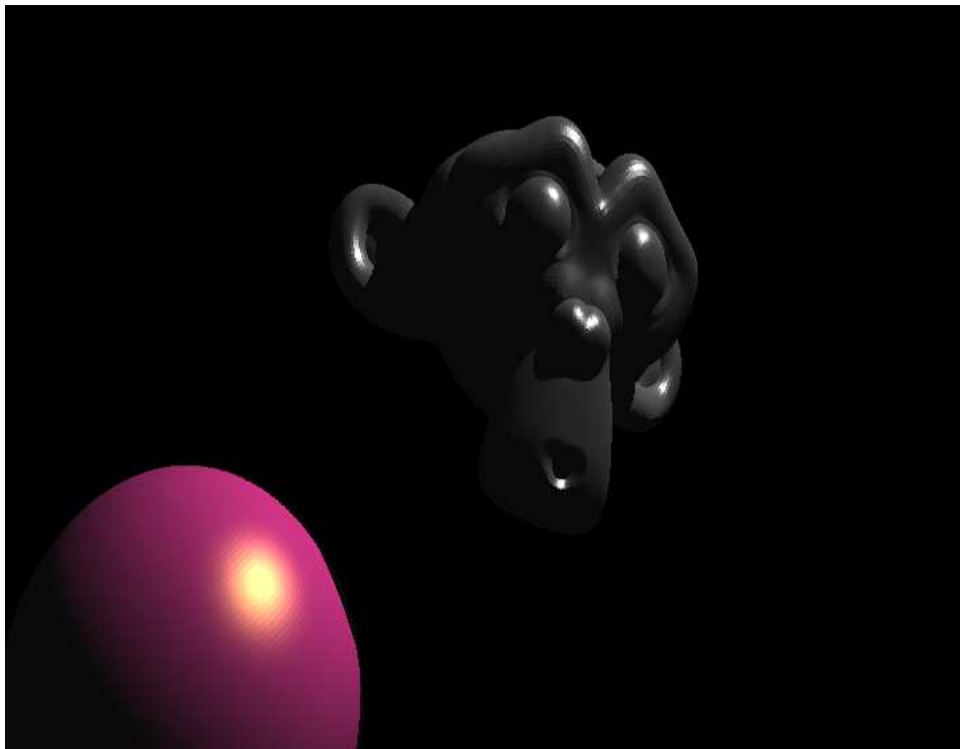


Figure 9.1: Example of a scene rendered with BDE and the technique described in this section.

9.2.1 Diffuse reflection

Diffuse surfaces scatter light equally in all directions (see Figure 9.2). Two examples of nearly-ideal diffuse surfaces are a chalkboard and matte paint.

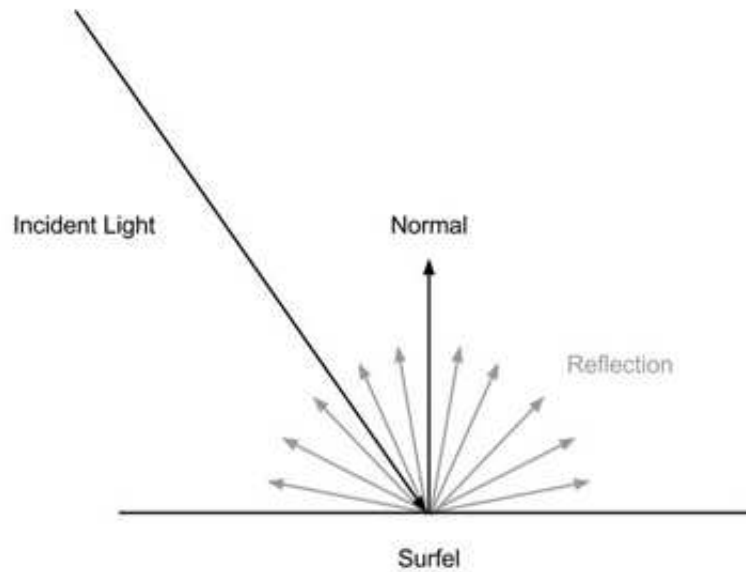


Figure 9.2: Diffuse reflection of light.

To obtain the factor that specifies the amount of light reflected we will use a Lambertian term:

$$\frac{l \cdot n}{|l| |n|}$$

Being l the vector resulting from subtracting the light position minus the hit point and n the surfel normal.

Once we have this term, we just have to multiply by the diffuse color of the surfel's material and we will have the diffuse component of the light reflection.

9.2.2 Specular reflection

Specular surfaces reflect light preferentially in a set of directions (see Figure 9.3). An example of specular surface would be a glossy material, with a mirror offering the perfect specular reflection.

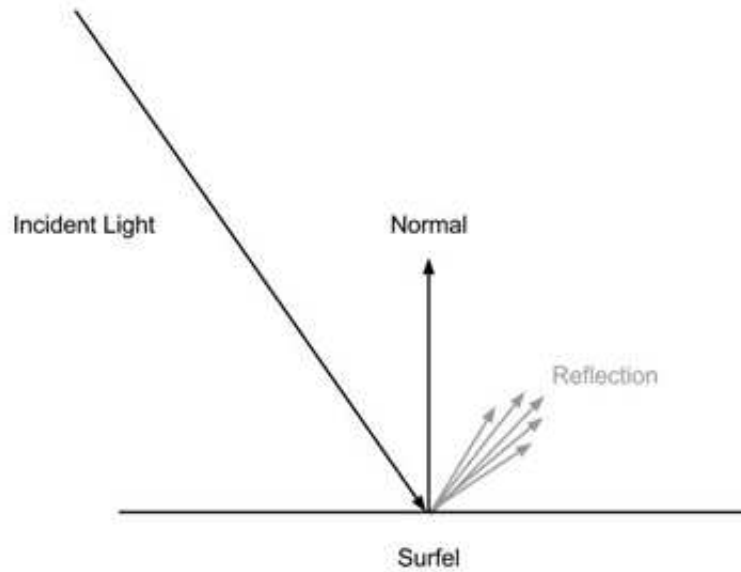


Figure 9.3: Specular reflection of light.

To obtain the factor that specifies the amount of light reflected we will use a Phong term:

$$\left(\frac{h \cdot n}{|h| |n|} \right)^f$$

Being n the surfel's normal, f the material's Phong exponent and h Blinn's halfway vector:

$$h = l + v$$

In which v is the view ray direction and l the same as before, the light position minus the surfel's hit point.

9.3 Monte Carlo

Illumination in the real world cannot be reproduced only with a direct illumination model, you need to account for indirect illumination as well. That is where ray tracing combined with Monte Carlo integration comes in. It makes it possible to compute global illumination with indirect lighting. Effects like color bleeding or contact shadows are naturally simulated by this model.

As we try to emulate with more precision reality, models will become more computationally expensive and we will get more realistic render results. The scene description will also be more complex, making the model even more computationally intensive.

The infinitesimal and continuous nature of light will result in the appearance of integrals in the mathematical description of the chosen model. As solving these integrals analytically is not a viable option, we will need to use numerical methods like Monte Carlo to solve them.

Monte Carlo is an algorithm that relies on repeated random or quasi-random sampling to compute results. This method is normally used when it is impossible to calculate a result with a deterministic algorithm.

The Monte Carlo method was created in the 1940s by John von Neumann, Stanislaw Ulam and Nicholas Metropolis, it was named after the Monte Carlo Casino a famous place where Ulam's uncle would gamble away his money.

9.3.1 State of the art

As realistic image synthesis has more applications in areas such as design, movies, architecture, etc. a common trend in all of them is demanding more realistic images of increasingly complex models. Monte Carlo based techniques are probably the best methods that can deal with this grade of complexity. Advances in algorithms and hardware has made Monte Carlo a natural choice for solving most problems. This has made an impact in classic techniques like radiosity that was the first choice for most graphics researchers a while back.

As we see in [AFH⁺01] we can apply Monte Carlo integration to calculate the direct lighting at a certain point from a light source in the presence of objects that can occlude it. We can also compute more advanced effects like indirect illumination. There are several monte carlo techniques that are widely used for this purposes, for example *path tracing* [LW93].

The algorithm integrates all the luminance arriving to a point on the surface of an object. The light arriving at this point will then be reduced by a surface reflectance function to determine how much of it will go towards the camera. Path tracing simulates naturally effects that have to be specifically added using other methods, like for example soft shadows, color bleeding, motion blur, caustics, etc.

Another popular algorithm is *Metropolis light transport* [CE05]. Is a variant of the Monte Carlo method Metropolis-Hastings. The algorithm constructs paths from the camera to the light source using path tracing, then introduces slight modifications to the path. The

advantage of this algorithm versus conventional path tracing is that when you find a path, this algorithm explores nearby paths, thus finding difficult paths that other algorithms could easily miss. Metropolis light transport is a method that sometimes converges to a solution of the rendering equation quicker than other algorithms, for example path tracing.

Photon mapping is also a popular method for computing global illumination. As we have described it in Subsection 4.2.2 we won't analyze it here.

9.3.2 Design

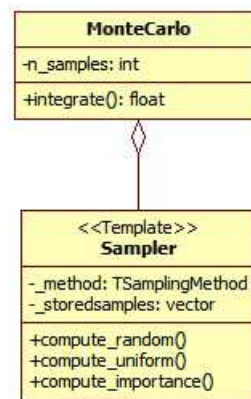


Figure 9.4: Monte Carlo class diagram.

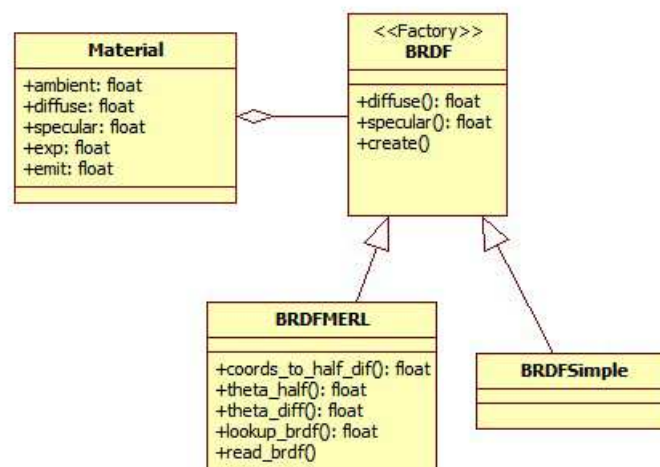


Figure 9.5: Material class diagram.

As we can see in Figure 9.4, **MonteCarlo** handles the task of integrating the incoming light in the hemisphere of a hitpoint. **Sampler** provides the necessary samples for **MonteCarlo** to perform its task.

In Figure 9.5 it can be seen that the **Material** class defines what the material properties of an object will be. Note that the **BRDF** class can be used to compute the reflectance of the material.

9.3.3 The rendering equation

In 1986 James Kajiya introduced the rendering equation [Kaj86], that even today is the basis of the majority of rendering engines. This is an integral equation in which the outgoing radiance at a certain point is the sum of all the emitted and reflected radiance in every direction (see Figure 9.3). This is supported by the energy conservation principle. We now show the rendering equation, that describes the total amount of light emitted from a point x along a particular viewing direction, given a function for incoming light and a BRDF:

$$L_o(x, \omega, \lambda, t) = L_e(x, \omega, \lambda, t) + \int_{\Omega} f_r(x, \omega', \omega, \lambda, t) L_i(x, \omega', \lambda, t) \cos(\widehat{\omega' n}) d\omega' \quad (9.1)$$

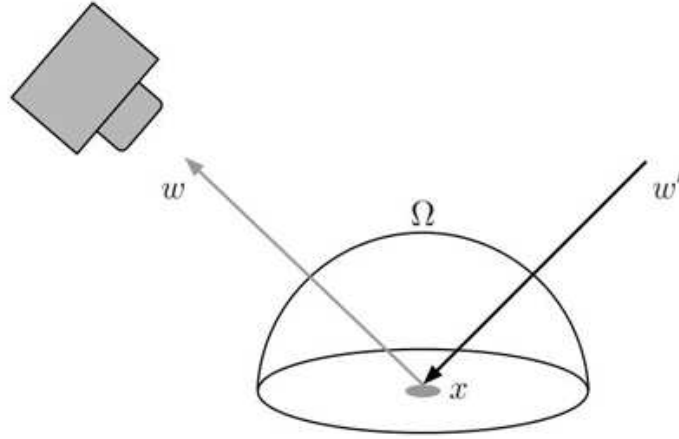


Figure 9.6: Visual aid to understand the rendering equation.

Where λ is a wavelength of light and t is time. $L_o(x, \omega, \lambda, t)$ is the total amount of light with wavelength λ directed outwards along direction ω at time t from a position x . $L_e(x, \omega, \lambda, t)$ is the emitted light (by the material), whereas $L_i(x, \omega', \lambda, t)$ is the incoming light reaching x from direction ω' , and $\cos(\widehat{\omega' n})$ represents the attenuation of incoming light due to the incident angle. $\int_{\Omega} \dots d\omega'$ is the integral over a hemisphere of inward vectors.

$f_r(x, \omega', \omega, \lambda, t)$ is the BRDF, the proportion of light reflected.

This equation has some limitations, it does not capture all of the aspects of light reflection. Some of them are phosphorescence, fluorescence, interference and subsurface scattering.

There are countless alternatives to solve the integral of Equation 9.1, like we have seen in Section 9.3.1. In this project we have used a brute-force approach based on Monte Carlo integration as described in the following subsections.

9.3.4 Monte Carlo integration

We now proceed to show how Monte Carlo integration is performed mathematically as we can see in [PH10, AFH⁺01].

Suppose we want to solve the integral $\int_a^b f(x)dx$. Given a uniform random variable $X_i \in [a, b]$, the Monte Carlo method states that the estimated value for the estimator is

$$F_N = \frac{b-a}{N} \sum_{i=1}^N f(X_i).$$

The estimator $E[F_N]$ is equal to the integral as we can see next. Note that the PDF³ corresponding to X_i has to equal $1/(b-a)$ because p has to be a constant and integrated to 1 for the domain $[a, b]$. Then we have

$$\begin{aligned} E[F_N] &= E \left[\frac{b-a}{N} \sum_{i=1}^N f(X_i) \right] \\ &= \frac{b-a}{N} \sum_{i=1}^N E[f(X_i)] \\ &= \frac{b-a}{N} \sum_{i=1}^N \int_a^b f(x) p(x) dx \\ &= \frac{1}{N} \sum_{i=1}^N \int_a^b f(x) dx \\ &= \int_a^b f(x) dx \end{aligned}$$

The restriction of using uniform random variables can be generalized to any other distribution function. This is relevant because it will let us choose alternative sampling methods to look for better convergence to the solution. If the random variable X_i follows a random

³Probability density function.

PDF $p(x)$, then we can utilize the estimator

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}$$

to solve the integral. The only restriction of $p(X_i)$ is that it cannot be equal to zero $|f(x) > 0|$.

Extending the Monte Carlo estimator to additional dimensions is trivial, in 3D the samples X_i just have to be chosen with three coordinates from a tridimensional PDF. For example, to solve

$$\int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{z_0}^{z_1} f(x, y, z) dx$$

we simply will use samples $X_i = (x_i, y_i, z_i)$ from the cube defined by (x_0, y_0, z_0) and (x_1, y_1, z_1) . Then the constant value of the PDF $p(x)$ will be

$$\frac{1}{(x_1 - x_0)} \frac{1}{(y_1 - y_0)} \frac{1}{(z_1 - z_0)}$$

and the estimator

$$\frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N} \sum_{i=1}^N f(X_i).$$

It is important to highlight that the number of samples N can be chosen arbitrarily, it will be independent of the number of dimensions that we are working with. Mathematically, this is one of the biggest advantages of the Monte Carlo method in rendering applications versus other numerical integration methods, where the number of samples required augments exponentially with extra dimensions.

The convergence of the Monte Carlo method is $O(\sqrt{N})$. Even not being specially faster than other methods, this value is also dimension independent, which is a big advantage.

9.3.5 Monte Carlo sampling

When we talk about sampling to solve the integral that the rendering equation presents with Monte Carlo, the PDF of the classical method is transformed in a distribution of tridimensional points over the surface of a imaginary hemisphere that surrounds the point in which we want to calculate lighting. The type of sampling used is one of the most important things we have to choose, it has a big impact in the generated solution and therefore in the final aspect of the rendered image.

This samples will be used to create rays with origin in the point that we want to calculate

(the center of the hemisphere) and direction set by the sample. This ray will interact with the scene looking for geometry, imitating a ray of light, etc. like a “probe”.

To generate the samples over the hemisphere, usually we use the polar coordinates (θ, ϕ) that have to be mapped to 3D space using $p(p_u, p_v, p_w)$. We now show how the mapping is done (see Figure 9.7):

$$p(\theta, \phi) = p(p_u, p_v, p_w)$$

$$(x \in [0, 1], y \in [0, 1]) \rightarrow (\phi = 2\pi x, \theta = \cos^{-1}[(1 - y)^{1/(m+1)}])$$

$$\mathbf{p} = \sin \theta \cos \phi \mathbf{u} + \sin \theta \sin \phi \mathbf{v} + \cos \theta \mathbf{w}$$

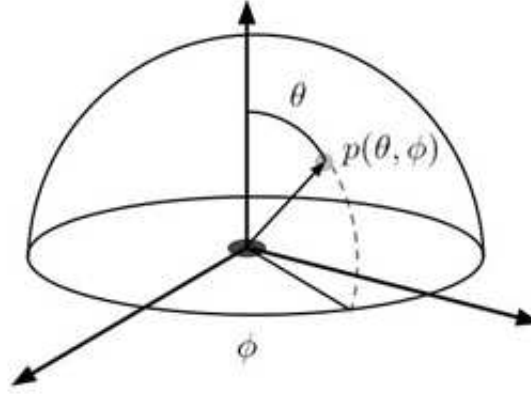


Figure 9.7: Visual representation of polar coordinates mapping.

Now we can generate the samples in one of two ways:

- Generating samples in a 2D square space and rejecting the ones that are not inside the hemisphere disc. After one is accepted, we would just have to “lift” it up to the third dimension using cosine modulation depending on the distance to the center.
- We could also generate the samples using polar coordinates and then transform them to 3D cartesian coordinates after the fact. This could be compared to working with latitudes and longitudes in the earth globe.

Depending on the PDF we can distinguish between the following types of sampling [Sam, Jas12]:

- **Pseudorandom:** Completely random sampling, it generates a lot of high frequency noise in the results.
- **Deterministic:** It distributes the samples uniformly within the hemisphere. It can yield good results with less samples, but it can generate visible artifacts in the results.
- **Quasi Monte Carlo:** It uses distributions with quasi-random numbers, i.e. that maintaining a random seed are slightly modified for obtaining a more uniform distribution. With this we can get “low discrepancy” sets. Examples of this sampling method are: stratified sampling, Poisson discs, etc.
- **Importance:** It is a general technique used for estimating properties of a certain distribution, while only having samples generated from a different distribution instead of the distribution of interest. For example, if we had a BRDF that would have a clear disposition toward a certain set of directions; we would sample with more emphasis around those directions. This method clearly improves the quality of the results [Law04].
- **Adaptive:** This is a special type of sampling, in which initially we do not know the total number of samples and where the computation of the samples and evaluation have to be alternated. We normally start with a uniform distribution in grid form. Once evaluated, we look for discrepancies between neighboring samples. If several close samples in the grid have very different results, we find ourselves in a high frequency zone of the integral. In this case we subdivide the grid in a new level and reevaluate the new samples but only apportioning half of their initial weight to the solution. This process is repeated until a maximum depth is reached. Outcome depends on the scene, but it generally yields pretty good results.

We provide a comparison between the methods implemented in BDE, so that the reader can see what kind of results can be expected from different types of sampling (see Figure 9.10).

9.3.6 Brute force Monte Carlo

Once we now how the theory behind the Monte Carlo method works, we just need to integrate it into BDE. We did that in the class **MonteCarlo**. We now will explain how the Monte Carlo algorithm is implemented in our engine.

First we calculate the corresponding samples with the **Sampler** class:

```
static Sampler sampler;  
sampler.compute(hit_point, normal, num_samples);
```

Once we have the samples, we need to check how they interact with the scene and compute the contribution of each sample to the result (if they do not intersect with anything in the scene they just contribute an ambient color):

```
for (int i = 0; i < n_samples; i++) {  
  
    Ray ray = Ray(hit_point, sample);  
  
    if (ray_intersects()) rgb = sample.compute_contrib();  
    else rgb = ambient;  
  
    sum_rgb += rgb;  
  
}  
  
sum_rgb *= inv_samples;  
  
return sum_rgb;
```

To calculate the contribution if there are intersections, we check if the surfel emits light; if it does it means that the ray that corresponds to the sample is unoccluded and that light reaches the surfel. We then compute the diffuse and specular contributions with the BRDF on that point.

If the surfel does not emit light, we have to use Monte Carlo again to see if from that indirect bounce we could reach a light source. So we make a recursive call to the Monte Carlo function that will repeat the process until a maximum depth. In Figure 9.8 we can see how light can bounce in objects in the scene and reach our object indirectly.

```
if ( surfel.emits() ) {  
  
    rgb = brdf.diffuse() + brdf.specular();  
  
} else if ( level < max_depth ) {  
  
    MonteCarlo new_mc = MonteCarlo();  
    rgb = new_mc.integrate(s, ray, level+1);  
  
}
```

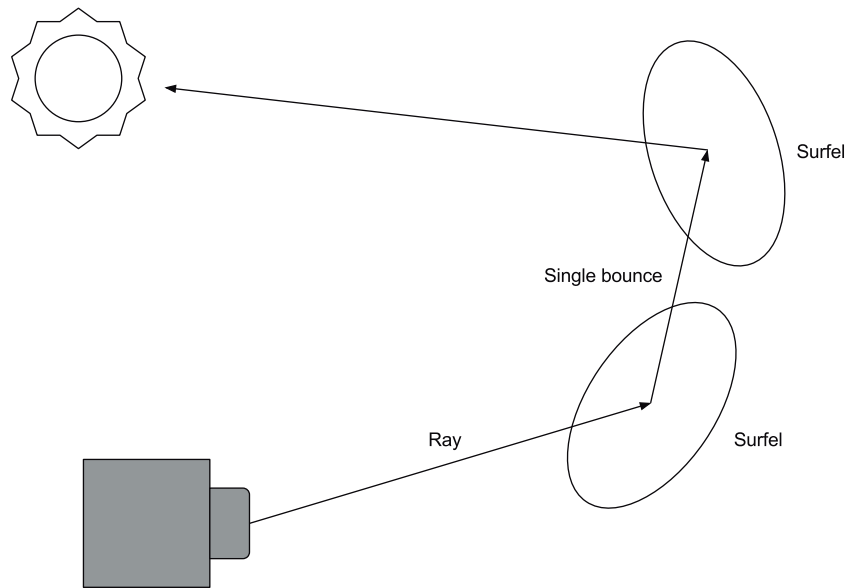


Figure 9.8: Example of a ray reaching a light source with a single bounce.

9.4 Results

In Figure 9.9 we can see how as we increase the number of samples rendering times increment exponentially. This performance test was run on a single thread of a Intel Core i5-2557M CPU and 4 GB of 1333 MHz DDR3 RAM. The test was run at a 640x360 resolution on a scene with 33000 surfels.

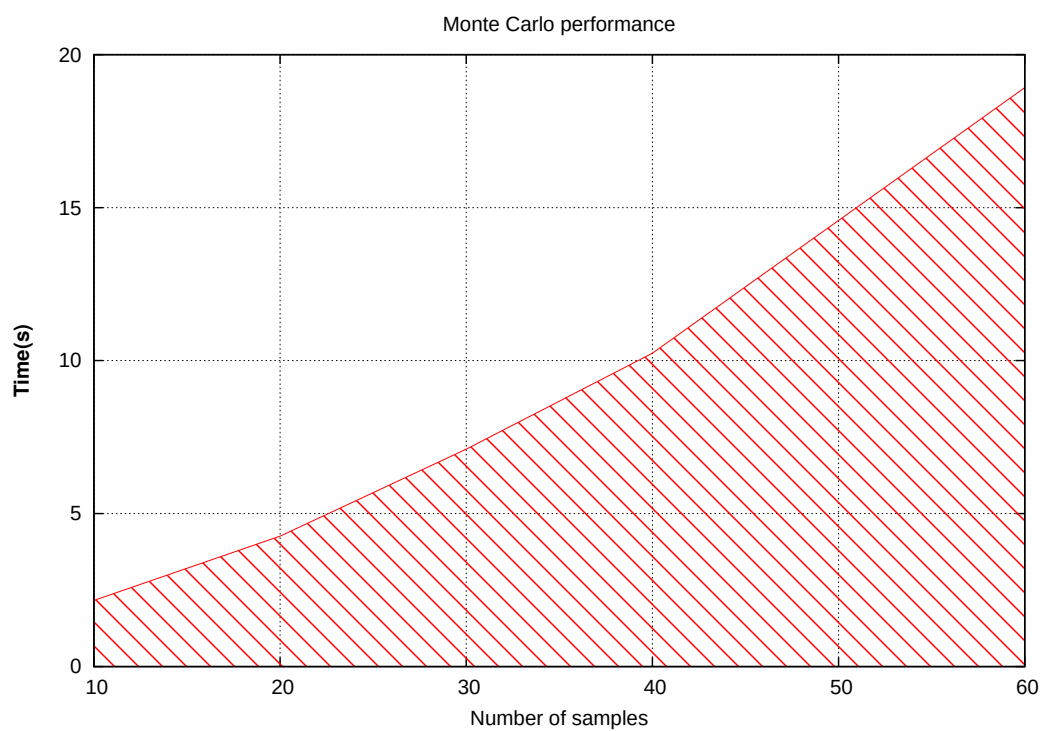


Figure 9.9: Graph that shows how performance varies depending on the number of samples.

We also offer several renders that showcase the kind of results that our global illumination approach can offer:

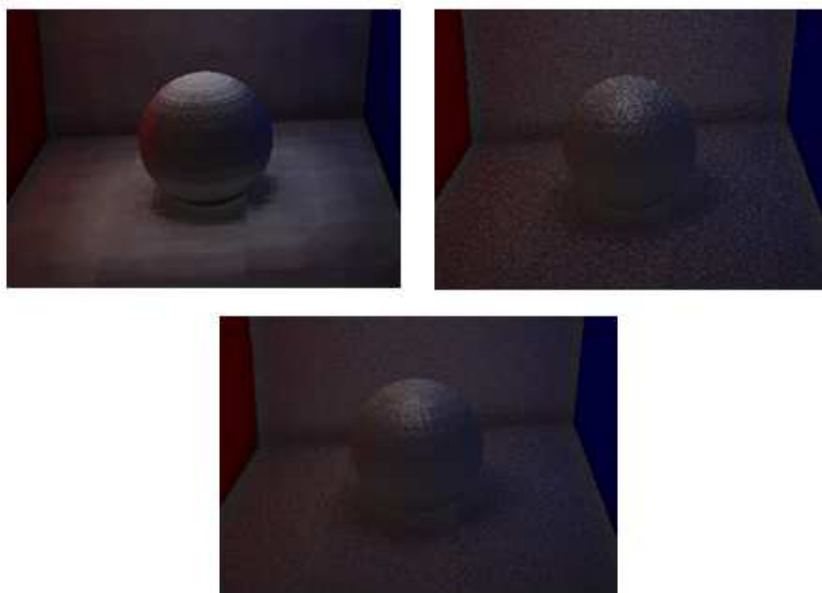


Figure 9.10: Our own version of the classic test scene “Cornell room” rendered with 100 samples. Comparison between deterministic, pseudorandom and QMC sampling methods from left to right.

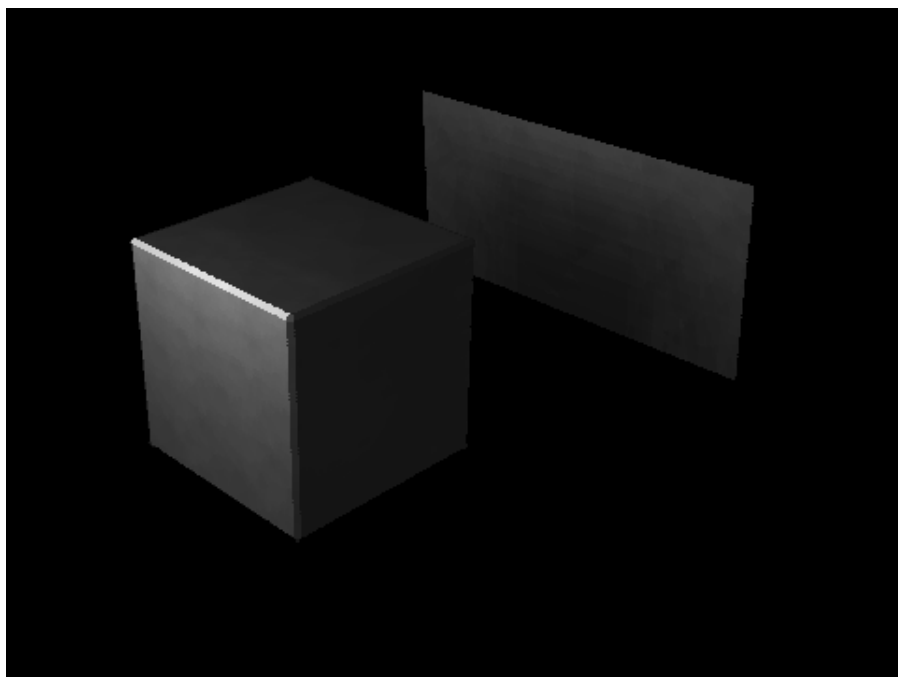


Figure 9.11: Monte Carlo with 700 deterministic samples. Render of a 33k surfel scene created in Blender.

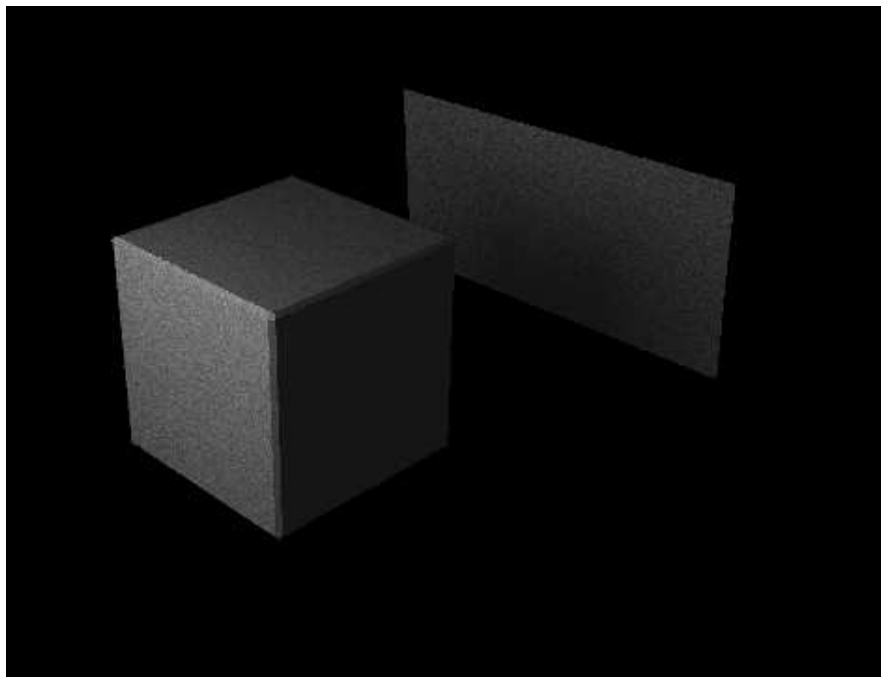


Figure 9.12: Monte Carlo with 700 QMC samples. Render of a 33k surfel scene created in Blender.



Figure 9.13: Monte Carlo with 500 QMC samples. Render of a 500k surfel scene created in Blender.



Figure 9.14: Monte Carlo with 500 deterministic samples. Render of a 500k surfel scene created in Blender.



Figure 9.15: Monte Carlo with 1000 random samples. Render of a custom “Cornell room” scene created in Blender.



Figure 9.16: Monte Carlo with 500 QMC samples. Render of a custom “Cornell room” scene created in Blender.



Figure 9.17: Hard shadows. Render of a 500k surfel scene created in Blender.

Chapter 10

Conclusions and future lines of work

10.1 Conclusions

The first conclusion that we have reached is the fact that all objectives that we laid out for the project were met:

- Build a raytracer from scratch.
- Research how traditional global illumination techniques can be applied to point-based rendering.
- How does using points as a primitive affect the classical rendering pipeline.

After having researched and dealt with the problems that have arisen in the design and implementation of a point-based rendering pipeline that includes physically-based illumination, several conclusions have been reached:

- **The importance of an acceleration structure:** The use of the k-d tree in our case has been key, as we could see in the performance tests in Chapter 6. The render times increase greatly when not using a k-d tree as the number of surfels grow in the scene. Instead when using a k-d tree they remain almost constant as you increase the number of surfels.
- **Rendering is a highly parallelizable workload:** The use of multithreading in our rendering algorithm reduces render times significantly even just using two threads, performance testing about this subject can be consulted in Chapter 8.

- **Constant model density:** This is something that is really important when using point clouds and trying to get realistic results. Models with non-constant density present problems when trying to obtain a high quality render; such as apparent holes in the objects or surfel overlapping. This problems can be overcome with a couple of techniques (see section 8.5), but it is best to use a model with constant point density if we want high quality results.
- **Normal vectors:** The surface normals of the surfels are one of the most important aspects when trying to achieve a high quality render. The normals can even be used as a measure of the quality of the renders, if the normals are not calculated correctly or we input a bad set of normals for the model, the render quality will be bad. If instead the normals are correct we will see good results. Ultimately normals play a really important role in the rendering process.
- **Real-time rendering of point clouds:** As we have seen in testing, it is probably possible to achieve rendering times that would be suitable for real-time rendering. Using a k-d tree and only a CPU we achieved milisecond render times. If we would employ a GPU instead of a CPU rendering times would be greatly improved even if hardware is not yet optimized for point-based rendering. Real-time rendering could only be achieved using the hard shadows technique, the Monte Carlo method increases rendering times greatly in its current state.
- **Multi-platform:** The engine can run on Mac OS X, Windows or Linux.

10.2 Future lines of work

Several lines of work come to mind when thinking about expanding BDE. Some of them are:

- **OpenCL:** As we have explained in the previous section, rendering is a highly parallelizable workload. OpenCL would allow us to use the parallel power of GPUs to accelerate the rendering process. This would lead to improvements in quality, because we would be able to use more samples while not increasing rendering times which would lead to higher quality images.
- **OpenGL:** We approached the fact that using a CPU we were able to obtain rendering times that were suitable for real-time rendering. The next step would be using OpenGL to accelerate the rendering times even more and display the results in real time.

-
- **Web visualization of point clouds:** Another possible line of work would be porting BDE so that it could be run on a web browser. This could widen even more the applications on the engine, visualizing point clouds directly on a browser would make even more open to everyone the engine.
 - **Improving illumination with Photon Mapping:** Because of the nature of Photon Mapping it would be a great match for BDE. This would yield a great improvement in rendering quality.
 - **Integrating BDE in more 3D software:** Implementing more plugins, i.e. for Maya, 3DS Max, etc.
 - **Preparing BDE for commercial use.**

Bibliography

- [AFH⁺01] James Arvo, Marcos Fajardo, Pat Hanrahan, Henrik Wann Jensen, Don Mitchell, Matt Pharr, and Peter Shirley. State of the art in monte carlo ray tracing for realistic image synthesis. *SIGGRAPH*, 2001. 74, 77
- [AGP⁺04] Marc Alexa, Markus Gross, Mark Pauly, Hanspeter Pfister, Marc Stamminger, and Matthias Zwicker. Point-based computer graphics siggraph 2004 course notes. *SIGGRAPH*, 2004.
- [AMHH08] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. ISBN 978-1-56-881424-7. AK Peters, third edition, 2008. 15
- [Bec01] Kent Beck. Manifesto for agile software development. *Agile Alliance*, 2001. 7
- [Caf98] R. E. Caflisch. Monte carlo and quasi-monte carlo methods. *Acta Numerica* 7, 1998. 6
- [CE05] David Cline and Parris Egbert. A practical introduction to metropolis light transport. *ACM Transactions on Graphics*, 2005. 74
- [Chr05] Per H. Christensen. Point-based approximate color bleeding. *Pixar Technical Memo 08-01*, 2005. 3
- [Coc01] Alistair Cockburn. *Agile Software Development*. ISBN 978-0-20-169969-2. Addison-Wesley Professional, first edition, 2001. 7
- [DLS05] Tamal K. Dey, Gang Li, and Jian Sun. Normal estimation for point clouds: A comparison study for a voronoi based method. *Eurographics Symposium on Point-Based Graphics*, 2005. 51
- [Fou12] Stichting Blender Foundation. Blender is the free open source 3d content creation suite. <http://www.blender.org/>, 2012. 3

-
- [FvDFH12] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. ISBN 978-0-20-184840-3. Addison-Wesley Professional, third edition, 2012. 13
- [Gla89] A. S. Glassner. *An introduction to raytracing*. Morgan Kauffman, third edition, 1989. 15
- [GP07] Markus Gross and Hanspeter Pfister. *Point-based Graphics*. ISBN 978-0-12-370604-1. Morgan Kaufmann Publishers, first edition, 2007. 4, 29, 30, 51, 52
- [Jas12] Alberto Jaspe. Técnicas de iluminación 3d basadas en el método de monte-carlo, 2012. 13, 79
- [Jen96] Henrik Wan Jensen. Global illumination using photon maps. *Rendering Techniques*, 1996. 30
- [Kaj86] James Kajiya. The rendering equation. *SIGGRAPH*, 1986. 76
- [Law04] Jason Lawrence. Efficient brdf importance sampling using factored representation. *SIGGRAPH*, 2004. 80
- [LBRF11] Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox. A large-scale hierarchical multi-view rgb-d object dataset. *IEEE International Conference on Robotics and Automation (ICRA)*, may 2011.
- [LMR07] Lars Linsen, Karsten Muller, and Paul Rosenthal. Splat-based ray tracing of point clouds. *WSCG*, 2007. 29
- [LW93] Eric Lafortune and Yves Willems. Bidirectional path tracing. *Proceedings of Computer Graphics*, 1993. 74
- [MPBM03] Wojciech Matusik, Hanspeter Pfister, Matt Brand, and Leonard McMillan. A data-driven reflectance model. *ACM Transactions on Graphics*, 22(3):759–769, jul 2003.
- [NM99] Amenta N. and Bern M. Surface reconstruction by voronoi filtering. *Disc. Comp. Geom.*, 1999. 51
- [PFS04] Matt Pharr, Randima Fernando, and Tim Sweeney. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, first edition, 2004. 3

-
- [PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. ISBN 978-0-12-375079-2. Morgan Kaufmann Publishers, second edition, 2010. 5, 15, 17, 22, 41, 43, 58, 77
- [Pix12] Pixar. Pixar’s renderman. <http://renderman.pixar.com/view/renderman>, 2012. 29
- [Pra12] Stephen Prata. *C++ Primer Plus*. ISBN 978-0-321-77640-2. Addison-Wesley, sixth edition, 2012.
- [Sam] Advanced computer graphics: sampling. Computer Science Department University of Freiburg. 79
- [Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. ISBN 978-0-12-369446-1. Morgan Kaufmann Publishers, sixth edition, 2006. 41
- [Tor05] Linus Torvalds. Git. <http://git-scm.com/>, 2005. 9
- [ZPKG06] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop3d. <http://graphics.ethz.ch/pointshop3d/>, 2006. 29