FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Scattered Multi-field Volumes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Stefan Hehr
Matrikelnummer 0306434

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Eduard Gröller
Mitwirkung: Dr. Raphael Fuchs

Wien, 02.02.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Stefan Hehr, Sandgasse 16/1, 4020 Linz, Österreich

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 02.02.2011

Danksagung

Mein Dank geht zunächst an meine Eltern, die in den vergangenen Jahren immer hinter mir standen und mich finanziell aus der Ferne unterstützt haben. Desweiteren möchte ich meiner Schwester Annette für ihre Tipps zum Schreiben englischer Texte und für die aufmunternden Telefonate mit ihr danken.

Ein Dankeschön geht an meine beiden Betreuer Dr. Raphael Fuchs und Dipl.-Ing. Jürgen Waser, die immer äußerst geduldig auf meine Fragen eingingen. Der Erste im Fachgebiet der Computergraphik und zum Thema "Wie schreibe ich eine Diplomarbeit", der Zweite in allen Fragen des Programmierens und der Softwareentwicklung. Einen ruhigen Arbeitsplatz fand ich im VRVis mit vielen netten KollegInnen und interessanten Vorträgen.

Weiters möchte ich all meinen Freunden danken, die ich monatelang mit meinem Diplomarbeitsgerede gequält habe. Und schließlich geht ein ganz herzliches Dankeschön an meine Sonja, die mir letztenendes immer wieder den Rücken gestärkt und mich nach Rückschlägen stets von Neuem zu motivieren verstand. Eifrig und mit nahezu endloser Geduld las sie mir in den letzten Wochen und Monaten stundenlang Korrektur.

Danke Euch allen.

Kurzfassung

Diese Arbeit thematisiert den Umgang, im Konkreten die Speicherung, die GPU-Handhabung, aber vor allem die Darstellung und Visualisierung von komplexen Volumendatensätzen. Diese können aus unterschiedlichen Primitiven zusammengesetzt sein wie zum Beispiel aus Dreiecken, Voxel oder Partikel. Die vorliegenden Daten müssen dabei nicht räumlich voneinander getrennt sein, sie können sich auch überlappen, gegebenenfalls auch miteinander interagieren. Ein vorstellbares Szenario wäre eine überflutete Landschaft, wobei das Gelände in Form eines Dreiecksnetzes, das Wasser als Volumen und eventueller Niederschlag als Partikel vorliegen.

Nach umfassender Recherche im Bereich der Datenstrukturen (vornehmlich Bäumen) soll eine geeignete Struktur gewählt und praktisch umgesetzt werden. Diese Struktur soll nicht nur ermöglichen, große Datenmengen zu beinhalten, sondern diese auch nach Bedarf schnell und mit möglichst wenig Aufwand in den Speicher der Grafikkarte schieben zu können. Die Verwaltung und Zurverfügungstellung der Daten ist der eine Teil, der andere ist die Auswertung und die abschließende Ausgabe, d.h. das Traversieren durch die Baumstruktur der Daten. Diese Visualisierung erfolgt durch Ray-Tracing. Die Umsetzung eines erweiterbaren Ray-Tracers auf der Grafikkarte ist ebenfalls Bestandteil dieser Arbeit.

Aufgabe dieser Arbeit ist es, verschiedene Ideen zu sammeln, wie die verschiedenen Datensätze gehandhabt werden können, um eine möglichst schnelle Anzeige zu ermöglichen. Die Ansteuerung des hybriden Ray-Tracers ist vor allem in der Handhabung des Datenartwechsels herausfordernd. Zu diesem Zweck wurden drei Methoden angedacht und die Vielversprechenste implementiert.

Der praktische Teil wurde in C++ und NVIDIA CUDA implementiert. Er beinhaltet, nocheinmal zusammengefasst, die Handhabung von verschiedenen Datensätzen vom Einlesen über das Vorbearbeiten, sprich das Einpacken in eine geeignete Datenstruktur, bis hin zur Verfügungstellung auf der Grafikkarte und schließlich die Implementierung eines GPU-Ray-Tracers. Besonders erwähnenswert ist, dass dabei gänzlich auf OpenGL oder DirectX verzichtet werden kann. Das aufwändige Ray-Tracing wird direkt auf der Grafikkarte parallelisiert. Die Einarbeitung in den Stand der Technik und in vorangegangene Arbeiten im Bereich der Datenstrukturen und des Ray-Tracings bilden den theoretischen Teil dieser Diplomarbeit. Die Ergebnisse werden durch Bilder von ausgewählten Szenarien visualisiert und anhand von Geschwindigkeitsmessungen evaluiert.

Abstract

The main topic of this thesis is the handling of complex data sets. To be more precise, it considers the storage, the handling on the GPU and the visualization of extensive and complex data. Different sets can contain different types of data like triangles, voxels and particles. In addition they do not have to be separated in space, which means they can overlap and therefore interact with each other. Considering a flooded landscape, the terrain would be made out of triangles, the water out of voxels and the rain can be visualized by using particles.

After extensive research in the field of data structures (mainly trees), one structure should be chosen and implemented. This data structure should be good enough to handle not only small sets of data, but also big ones and should allow to move them efficiently to the memory of the graphical unit. To store and to manage data sets is only one part of the whole process, the other is to visualize them by traversing the tree data-structure. The visualization is done by ray tracing. Thus the implementation of an expandable ray tracer on the graphical unit is also part of this thesis.

The main task of this work is to gather several ideas how to handle different data sets of different kinds of primitives efficiently. The ray tracing on the graphical unit should be done as fast as possible. Switching between the different datatypes is the main challenge of this hybrid ray tracer. Three different methods were thought about for this purpose and finally the most promising one was chosen and implemented.

The practical part of this work was implemented in C++ and NVIDIA CUDA. It includes the handling of different data sets beginning with reading in. The next step is to process data sets, put them into a proper data structure and move them to the graphical unit and finally the implementation of an expandable GPU ray tracer. Notably this process needs neither OpenGL nor DirectX, the whole ray tracer is parallelized on the GPU. The research in the state of the art in the field and previous works of data structures together are making up the theoretical part of this master thesis. The results are visualized by pictures of selected scenarios and evaluated by measurements of speed.

Contents

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Basics and Terminology . . . 2
- 1.2 Requirement Analysis . . . 6
  - 1.2.1 Hardware . . . 7
  - 1.2.2 Software . . . 7
  - 1.2.3 Project . . . 7

2 State of the Art 9
- 2.1 Ray Tracing and Ray Casting . . . 9
- 2.2 Acceleration Structures . . . 10
  - 2.2.1 Bounding Volumes and Bounding Volume Hierarchies . . . 10
  - 2.2.2 Spatial Subdivision . . . 11
- 2.3 Methods of Tracing Rays through Data Structures . . . 15
- 2.4 Acceleration Techniques . . . 16
- 2.5 GPU and Parallelism . . . 16

3 Scenarios 17
- 3.1 Industrial Application . . . 17
- 3.2 Video Games . . . 17
- 3.3 Virtual and Augmented Reality . . . 17
- 3.4 Archeology . . . 18
- 3.5 Photorealism . . . 18
- 3.6 Architecture . . . 18
- 3.7 Visualization of Medical Data . . . 18
- 3.8 Fluid Simulation . . . 19

4 Algorithms 20
- 4.1 Geometry . . . 20
  - 4.1.1 Separated Objects . . . 21
  - 4.1.2 One Global Tree . . . 21
  - 4.1.3 Steering Geometry . . . 22
- 4.2 Optics . . . 25
- 4.3 Optimization . . . 26

4.3.1 Acceleration Structures . . . 27
4.3.2 Methods of Tracing Rays through Data Structures . . . 36
4.3.3 Tracing Particles . . . 40
4.3.4 Tracing Voxel Grids . . . 41

5 Implementation **42**
5.1 Framework . . . 42
5.2 Sourcecode and Classes . . . 43
5.2.1 Scene Item . . . 45
5.2.2 Model-Loader Node . . . 45
5.2.3 Tree Factory . . . 46
5.2.4 Acceleration Structure - Tree . . . 46
5.2.5 Steering Geometry . . . 48
5.2.6 Ray Renderer . . . 48
5.2.7 Cuda Ray Node . . . 49
5.2.8 GPU - CUDA Kernels . . . 50

6 Evaluation **53**
6.1 An Expressive Way of Testing Speed . . . 53
6.2 Basic Speed Tests . . . 53
6.2.1 Ray Intersections . . . 55
6.2.2 Memory Access Measurements . . . 56
6.3 Octree Settings . . . 57
6.3.1 Tree Height . . . 58
6.3.2 Polygon Threshold . . . 59
6.4 Traversal Methods . . . 61
6.5 Steering Scenarios . . . 64

7 Results **65**
7.1 Features . . . 66
7.2 Steering Scenarios . . . 70

8 Conclusion **75**

Bibliography **77**

List of Figures

1.1 Shaders . . . 1
1.2 A 'Whitted' Ray Tracer . . . 3
1.3 Viewing frustum . . . 5

2.1 Bounding Volume and Bounding Volume Hierarchies . . . 11
2.2 Spatial Subdivision . . . 12

4.1 Simple scenes . . . 20
4.2 Simple scenes with separate bounding volumes . . . 21
4.3 Simple scenes with global tree . . . 22
4.4 Simple scenes with steering geometry . . . 23
4.5 The steering-geometry approach: The scene *Head in Glass* . . . 23
4.6 The steering-geometry approach: Two rays shot through the scene *Head in Glass*. 24
4.7 Numerical problems in calculating intersection points . . . 26
4.8 A quadtree example . . . 28
4.9 A quadtree example. The tree nodes in preorder. . . . 29
4.10 Handling overlapping polygons . . . 30
4.11 The ropes of a simple quadtree . . . 31
4.12 Build-up ropes - five cases . . . 32
4.13 Using ropes . . . 35
4.14 RSA Method 1 . . . 38
4.15 RSA Method 2 . . . 39

5.1 The framework GUI . . . 43
5.2 A class diagram using Unified Modeling Language . . . 44
5.3 SceneItem . . . 45
5.4 ModelLoaderNode . . . 45
5.5 TreeFactory . . . 46
5.6 Tree classes . . . 47
5.7 SteeringGeometry . . . 48
5.8 RayRenderer . . . 48
5.9 CudaRayNode . . . 49
5.10 RayRenderer Kernel . . . 50
5.11 RayRenderer Functions . . . 50
5.12 RayRenderer Helpers . . . 51
5.13 RayRenderer Trace Kernel . . . 52

6.1 Octree Settings 57
6.2 Traversal Methods 61

7.1 An object rendered by using the implemented ray tracer 65
7.2 Features: Simple shading and steering polygons 66
7.3 Features: Shadow rays and self-shadowing 67
7.4 Features: Reflection 68
7.5 Features: Refraction and opacity 69
7.6 Scenarios: Industry 70
7.7 Scenarios: Video Games 71
7.8 Scenarios: Still life - a bottle and a glass 72
7.9 Scenarios: Still life - a head in a glass in front of different backgrounds 73
7.10 Scenarios: Still life - a head in a glass including visualized steering polygons . . . 74

1. Introduction

The graphics hardware improved enormously in the past few years. Nowadays it is not only possible to program parts of the graphics-hardware pipeline with so called *shader programs* (see figure 1.1), one can also use it to parallelize complex and expensive calculations. The high degree of parallelization opens possibilities, which can not be realized on conventional multi-core CPUs.

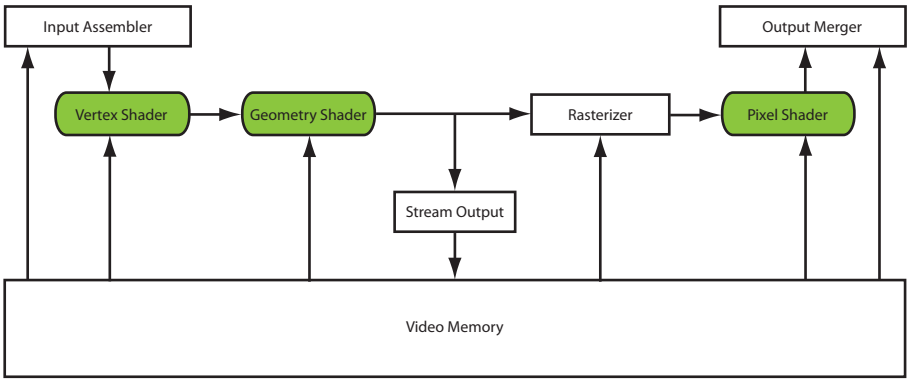


Figure 1.1: Shaders

To analyze how powerful this free programmable graphic hardware is, it would be interesting to implement a ray-tracing system trying to reach real-time performance, also with bigger scenes including different kinds of geometric data. Every data can have a different type of source, e.g., a result of a simulation process like fluids or simply read from file like a transparent box consisting of quads. The task was to put them together in a final step and render them.

Such a system could help in industrial applications, when objects have to be examined accurately in less time or to build up a fast preview. It also has its future in video games [FGD+06], or in other real-time and/or multi-field applications. In chapter 3 the following keywords and topics will be discussed in detail:

- Virtual and Augmented Reality
- Visualizing Medical Data
- Archeology - Preservation and Presentation of Cultural Heritage
- Architecture

- Photorealism
- Other Media

For several months a framework was implemented, which should be able to simulate data and to visualize the results. To develop a system, which matches the requirements mentioned above, is time-consuming and complex. A lot of time was spent to write a ray tracer as described by [Gla89], not only for one type of primitive, but for several types. To test different approaches to trace rays, it should be easy to modify, too.

For speed-up an adequate acceleration data-structure for the primitives had to be found and the most promising one was implemented. Read different model files, store them, handle them, especially move them to the graphical unit was also a time-consuming work. Costly in terms of time are two other points which have to be mentioned: first of all working with and debugging a system still in development, which CUDA definitely is, and secondly building up a new framework from scratch in a team was another challenge and often led to long discussions and changes.

This thesis gives a detailed overview of the work which was done in the last months and its results. It is targeted at readers, who do not have to be experts in computer graphics. All basics and terms, which are needed, are described in the following section.

1.1 Basics and Terminology

In this section some terms are explained, which are used later on in this work. There are also a few definitions, which are important to understand the algorithms below (see chapter 4).

- **The Rendering Equation**
  The rendering equation was described by Kajiya in the year 1986 [Kay86]. It is a solution to the problem of global illumination. The original rendering equation is:

$$I(x, x') = g(x, x')[\epsilon(x, x') + \int_S \rho(x, x', x'') * I(x', x'')dx''] \tag{1.1}$$

  where:

$I(x, x')$	is the intensity of light passing from point *x'* towards point *x*.
$g(x, x')$	is a geometric term depending on the position of point *x* to point *x'*. If an object is in between, this term is set to 0.
$\epsilon(x, x')$	is the intensity of light which is emitted from point *x'* towards point *x* if point *x'* is a light source.
$\rho(x, x', x'')$	is the intensity of light outgoing from a point *x"* and being reflected in point *x'* in the direction of point *x*.
$I(x', x'')$	is the intensity of light passing from point *x"* towards point *x*.

Simply said the rendering equation returns the outgoing light at a given point *x*. This light includes the self emitted light in addition to all gathered light which is coming from other points *x'* and is reflected towards *x*.

- **Ray Tracing and Ray Casting**

  A ray tracer is a method to visualize a scene in a photo-realistic manner. Turner Whitted introduced a basic ray tracer in 1979 [Whi79]. Figure 1.2 shows a simple one. For every pixel a ray is shot into the scene, which is called a *primary ray*. If the ray hits an object, it stops. An extra ray is sent in the direction of the light source or the light sources to see if the point of intersection lies in shadow or not (*shadow ray*). If the hit object mirrors the ray, its direction may also be reflected and traced further into the scene (*secondary rays*).

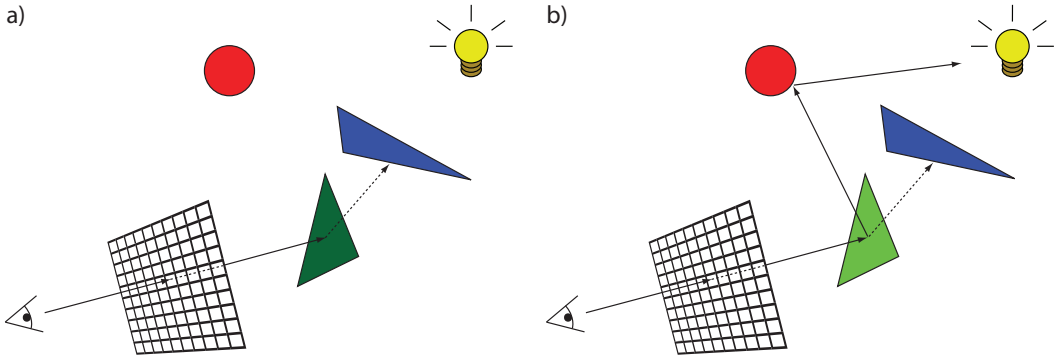


Figure 1.2: A 'Whitted' Ray Tracer. a) the intersection point between the eye ray and the green triangle is not illuminated b) the intersection point is illuminated indirectly

A ray tracer is able to simulate many visual effects which create the illusion of real life images such as caustics, shadows, refraction, reflection and other so called *scene features*. All of these effects are useful to imitate (global) illumination that looks real in the given scenario. The distribution of light is not as simple as shown in figure 1.2a), where the green triangle is illuminated only if there is no other object lying between the intersection point and the light source. *Global illumination* means that the light in a scene can be propagated for example by a reflection of another object (mirror) as shown in figure 1.2b). This must be considered for a real looking image. If only a single ray is shot per pixel, it is called *ray casting* or *first hit ray tracing*.

- **Rays**

  A ray can be defined by two components - its *origin* and its *direction*. To move a point p an arbitrary distance t along a normalized direction vector $\hat{d}$, equation 1.2 can be used to calculate point p':

$$p' = p + t * \hat{d} \tag{1.2}$$

- **Eye**
  Figure 1.2 shows that the origin of the eye is the point in which the ray starts. In this work the term *eye point* is used to describe that point.

- **Color Handling**
  Colors can be stored in four channels. Whereas the first three channels are assigned for *red*, *green* and *blue*, the last one is containing the *opacity*. While tracing rays color has to be added each time the ray intersects an object. Adding color was described by Marc Levoy [Lev90] and can be done by using the following equations:

$$C_{out} = C_{in} + (1 - \alpha_{in}) * C * \alpha \tag{1.3}$$

$$\alpha_{out} = \alpha_{in} + (1 - \alpha_{in}) * \alpha \tag{1.4}$$

  where:

C_{out}	is the resulting accumulated color,
α_{out}	is the resulting accumulated opacity,
C_{in}	is the current accumulated color,
α_{in}	is the current accumulated opacity,
C	is the color, which has to be added and
α	is the opacity, which has to be added.

  The color of the ray can be calculated by dividing the finally resulting color channels by alpha.

- **Scene Features**

  There are some features that can be added to objects to increase the photo realism of a scene. A few of them are listed and explained below. By using the rasterization technique the following effects can only be realized - if at all - by rough approximations. A ray tracer can calculate them all in a natural way.

  - *Illumination*

    Shadows, self shadows and illumination, which was already explained in the rendering equation above, are more or less simple to implement. An extra ray from a point of intersection has to be shot to the light source(s). If an object is in-between, the point lies in shadow and can receive light only indirectly.

  - *Reflection*
    To reflect a ray at a point of intersection, the reflection vector (and the surface normal, if it is not given) has to be calculated. The equation of reflection is given in [Gla89] on page 132. The following form of the reflection equation (1.5) is taken from the book of Hearn and Baker [HB04] (page 600):

$$r = u - dot(2u, n)n \tag{1.5}$$

    where:

r is the reflection vector,
u is the incoming ray,
n is the surface normal.

– *Refraction and caustics*
If non-opaque surfaces are hit, a ray can enter the object at this point and can change its direction. This is called refraction. Some examples of non-opaque surfaces are glass, prisms (causing caustics) and water. To calculate the change of direction of the ray, equation 1.6 is used (see Hearn and Baker [HB04] on page 600):

$$t = \frac{\eta_i}{\eta_r} u - (cos\Theta_r - \frac{\eta_i}{\eta_r} cos\Theta_i) n \tag{1.6}$$

where:

t	is the refracted vector,
u	is the incoming ray,
n	is the surface normal,
η_i, η_r	are the refraction indices in the incident material and the refracting material,
Θ_r	can be calculated by Snell's law: $cos\Theta_r = \sqrt{1 - (\frac{\eta_i}{\eta_r})^2 (1 - cos^2\Theta_i)}$
Θ_i	is the angle between the incoming ray and the normal vector *n*

- **Rasterization**
Rasterization is a technique to visualize a scene. In a simple process every single triangle is handled separately and can be shown on the display.

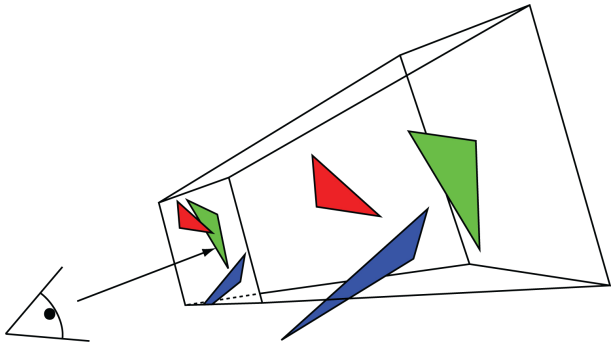


Figure 1.3: Viewing frustum

Figure 1.3 shows the so called viewing frustum, a truncated and four sided pyramid. To visualize the triangles they are projected onto the smaller plane to the left, which is called the near plane. The bigger plane on the right side is called far plane. Every triangle which lies behind the far plane is cut away. The projection of triangles leads to some smaller problems. For example they have to be clipped at the borders of the near plane (see figure 1.3, the blue triangle). To determine the color of each screen pixel the clipped triangles

have to be rasterized.

- **Primitives**

  Primitives are not only polygons like triangles, quads and rectangles, but also spheres, particles and voxels, the 3D analogon of pixels.

- **Data Structures and Acceleration Structures**

  Primitives can be stored in so called *data structures* or *acceleration structures*. The use of these structures is to decrease the number of intersection tests. Primitives are grouped and assigned to, for example, a bounding volume like an axis-aligned bounding box (AABB) enclosing all these primitives.

- **Trees**

  Global definitions for all trees mentioned in this thesis:

  - Each tree has only one root node.
  - Each tree is *acyclic*, except rope trees.
  - The *height* of a tree is equal to the length of the longest path from root to leaf.
  - The *depth* of a node is equal to the length of the path from the node to the root.

- **Real-time**

  Real-time is usually defined as at least 60 frames per second.

- **Scattered Multi-field Volumes**

  Scattered Volumes: Various volumes can be positioned arbitrarily in the scene.
  Multi-field Volumes: These volumes do not have to be spatially separated. They can overlap each other, so that one point in space has several values [FH07]. An example is data from MRT and CT.

1.2 Requirement Analysis

Before taking a look at the state of the art, a short outline of the problems and requirements of the practical part of this work are given. In chapter 4 some of the following issues will be treated in detail.

1.2.1 Hardware

Hardware specific requirements of the system will not be a big topic in this work. A usual modern hardware configuration should be enough. The ray tracer was tested and implemented on an Intel(R) Core(TM)2 Duo CPU with 4GB RAM and an NVIDIA GeForce 8600 GTS with 512 MB Memory. All numbers and results are, if not mentioned separately, measured on this hardware system. To run the ray tracer a system requires only a graphic card supporting NVIDIA CUDA 2.3.

1.2.2 Software

The requirements of the software, which is needed for this project, are low. A simple software pipeline, consisting of input, processing and output is sufficient. A possibility to outsource parts of the processing step to the graphical unit is needed. The CPU handles the input (like loading or creating data), preprocessing (like building up acceleration structures), steering GPU calls and finally the output on screen. More information about the implementation of the framework and its graphical user interface (GUI) is given in chapter 5.

1.2.3 Project

The choice of a proper acceleration structure or perhaps different acceleration structures needs a close analysis of the requirements. These requirements can be categorized into two main issues: Requirements for Rendering (RR) and Requirements for Performance (RP). The goal is to find a solution, which satisfies both of them.

1.2.3.1 Requirements for Rendering

The rendering of the scene is done by tracing rays, not only through volumes consisting of voxels, but also through other geometrical primitives like triangles and particles. This requires the possibility to easily access the primitives from an arbitrary spatial block. Casting rays in multi-field data scenes implies some special requirements. To cull currently invisible parts of the scene, it is desirable to have the ability to discard them fast. This can be done easily by using a tree as an acceleration structure like an octree or Kd-tree, where it is simple to cut off single branches and push them to the graphical unit separately.

1.2.3.2 Requirements for Performance

As mentioned before, handling spatially clustered parts of data is a very important issue for the performance of a ray tracer. Some questions can be collected and grouped into the categories *Data-handling* and *Parallelism*:

- **Time**
  How to keep time complexity low?
  What are the most expensive procedures?
- **Data-handling**
  How to upload the needed data efficiently?
  How to avoid the host-device bottleneck?
- **Parallelism**
  How to portion data?
  Which procedures can be executed in parallel?

2. State of the Art

In the following sections an overview of all relevant issues is given. The first part will be about tracing rays. After this a few types of data structures holding the geometrical data of the scene will be discussed. The next section will show methods to traverse these structures while tracing rays. Some interesting works about acceleration techniques are part of this thesis. The last section treats ray tracing by using the GPU, parallelism and related work in this field.

2.1 Ray Tracing and Ray Casting

As already described in section 1.1, a ray tracer can be a really extensive and difficult challenge. The implementation of such a ray tracer is a complex project. First it has to be determined, which features are needed. Then the options, the ray tracer should offer and which are less important, have to be sorted out. Paul S. Heckbert grouped these options in four categories (see Glassner [Gla89] page 264ff):

- **Geometry**
  This category includes the extensibility of the set of primitives as well as simple and complex operations on primitives like *read*, *transform* and *deform*.
- **Optics**
  Which optical effects should be simulated, for example shadows, reflection, refraction, fog and many more.
- **Optimization**
  First of all *Optimization* concerns the optimization of algorithms. Other topics are the choice of acceleration structures to reduce the number of intersection tests and the finding of the cheapest intersection tests. Common optimizations like changing hardware or programming language are also part of this category.
- **System**
  This category deals with modularity, debugging, interaction, animation and distributed computing.

Based on these four suggested categories the practical work was planned and grouped (see chapter 4). The following section 2.2, for example, is dedicated to the category *Optimization*.

2.2 Acceleration Structures

A native ray-tracing algorithm needs too much time because it has to intersect every element with the ray, starting from the eye point. A spatial subdivision technique (acceleration structure) is required to speed up the traversal. The choice of a proper acceleration structure is not an easy task. There are many issues to be considered and some structures are more or less useful for the objects in the scene [TS05] [Hav00] [Bou05]. For example fluids are usually not stored as polygons, but as particles. Simple geometrical shapes like boxes, cubes or pyramids can be exactly represented by using polygons. Spheres can be approximated by polygons, too, but much more polygons are necessary to represent them.

For the storage of data (e.g., results of MRT or CT), polygons are not useful as the scanned data does not provide geometrical properties. Hence for each cell in a 3D grid of an arbitrary resolution a scalar value has to be stored. So if there is data, which does not fill a whole volume, only its hull has to be stored. This hull can be represented simply by polygons. Due to the knowledge of theses facts the following acceleration structures should be considered.

2.2.1 Bounding Volumes and Bounding Volume Hierarchies

As the shapes of objects are often complex, the ray intersection can be very expensive. An idea to reduce these costs is to wrap the objects in simple primitives like spheres or boxes which can be intersected more efficiently. So a first fast intersection test with this bounding volume can exclude many rays which do not hit the object.

The big disadvantage of spheres as bounding volumes is, that if one dimension of the bounded object is for example very large in relation to the other dimensions, a simple sphere can contain much empty space. For such inhomogeneously dimensioned objects another bounding volume has to be chosen. Bounding volumes can be used in other approaches to speed up as well.

The next logical step is to extend them to a hierarchy of bounding volumes (BVH). Each parent volume contains some child volumes lying spatially completely inside of their parent volume. This means that every ray, which misses the parent volume, misses all children as well. Or the other way around, if a ray intersects a parent, only its children have to be checked. If there is more than one of them intersected, the closest one has to be found (see figure 2.1). Child volumes do not have to be disjoint completely, i.e., they can overlap each other [GS87]. Niels Thrane and Lars Ole Simonsen compare different acceleration structures including BVH and their usage on GPU [TS05].

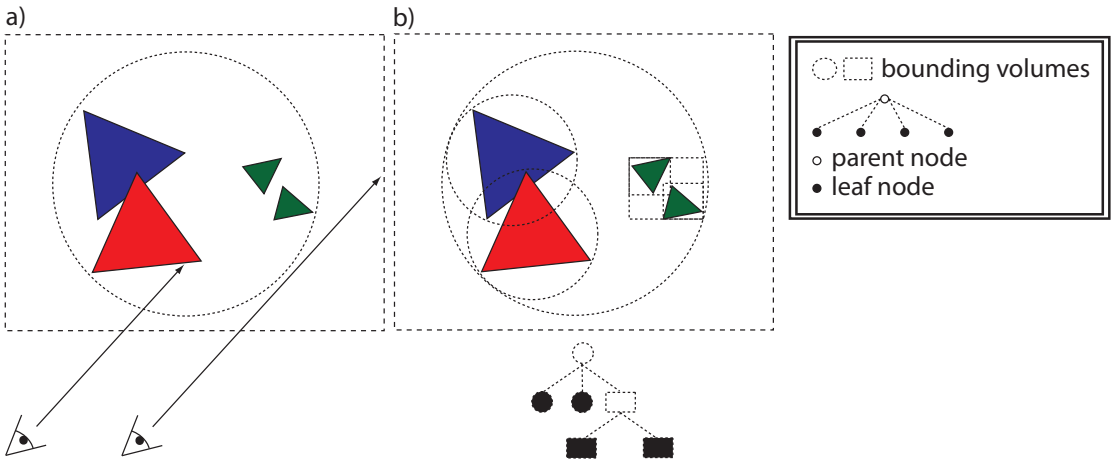


Figure 2.1: a) Bounding Volume (Sphere) b) Bounding Volume Hierarchies

2.2.2 Spatial Subdivision

Another type of acceleration structures are *spatial subdivisions*. The spatial region of a scene is subdivided into smaller parts or cells. So the scene and its included objects are divided and ordered in space. The structure containing the information about the subdivisions also includes information about the distances between the objects. The leafs of the subdivision structures are pointing to the included object data. Further information about this acceleration technique can be found in the literature [Gla84], [TS05], [Kap87] and [FTI86].

Spatial subdivision structures are usually built by dividing space, by splitting at planes, which usually are axis-aligned, but not necessarily. While tracing rays through a subdivision structure only a few events can occure:

- The ray intersects an elementary cell or leaf node and the assigned object. The intersection point is lying inside of the elementary cell (see figure 2.2): In this case a valid intersection point has been found.
- More than one object is found. So the algorithm has to find the closest one.
- No intersection is found or the intersection point is lying outside of the elementary cell.

The method of tracing rays through a spatial subdivision is called *Ray Shooting Algorithm* (RSA). The main difference to the BVH approach is that the elementary cells are spatially strictly separated, they do not overlap each other.

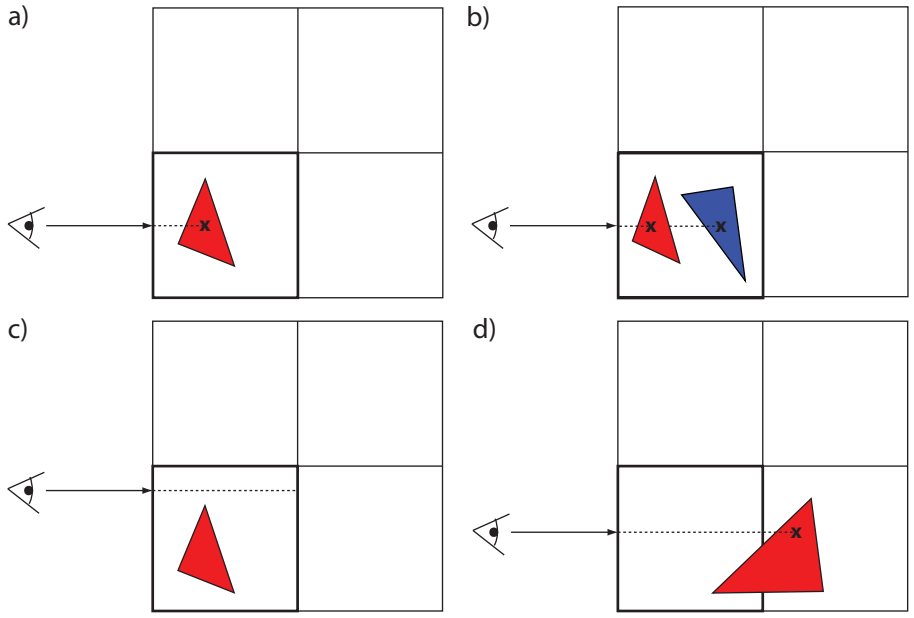


Figure 2.2: Spatial Subdivision - a) Object found b) Two objects have to be sorted c) Ray misses the object d) Intersection point is lying outside, in a neighboring elementary cell

2.2.2.1 Grids

Grids have been well discussed already. This work does not go into detail. To visualize the grid volumes, only a simple traversal algorithm (front-to-back and back-to-front) was implemented. Thus only a short overview of grids is given in the following section.

A **Uniform Grid** is the first spatial subdivision technique to be explained. The scene is split into cells with the same elementary size called *voxels* (volume elements), analogous to *pixels* in 2D (picture elements). The uniform grid in the context of RSA was first introduced by Fujimoto [FTI86]. The ray traversal algorithm for a uniform grid has been improved by several researchers, including Ping-Kang Hsiung and Robert H. Thibadeau [HT92] and Robert Endl and Manfred Sommer [ES94].

Traversing a uniform grid can be done by using a 3D DDA, the three dimensional discrete differential analyzer. One disadvantage of this simple algorithm is that each voxel has to be checked and some value has to be stored and sent to the graphical memory, no matter how insignificant this value might be. If a ray is shot, for example, through a voxelized sphere at most two voxels would be hit, which contain significant values. All other voxels, before, inside and behind the volume are visited in vain. For medical data, as produced by an MRT, a uniform grid can be suitable, because each voxel contains data. The uniform grid was analyzed by John G. Clearly and Geoff Wyvill [CW88].

Non-Uniform Grids are very similar to the uniform grids. Both split the space axis-aligned. The only exception is that the cells do not have to be of the same size (uniform). So the space can be split along an arbitrary axis as well.

The advantage of non-uniform grids is obvious. An area of many empty cells of a uniform grid can be gathered to one cell and skipped in a single step. On the other hand it is also obvious, that it costs much more storage and memory as the cell-split information has to be stored and read. Another disadvantage is, that the algorithm to traverse the scene has to be more complex than the simple 3D DDA already described before [Gig90].

As a last grid technique in this short overview about grids, we give some information about **Hierarchical Grids**: The idea is to insert grids into other grids. According to the object a finer grid can be set if more details have to be stored. The algorithm traversing a hierarchical grid is more complex than the simple 3D DDA.

2.2.2.2 Kd-tree

Before introducing the Kd-tree, a description of its next relative, the Binary Space Partitioning tree (BSP), is given. There are two different sorts of BSP. The first type is the axis-aligned one, which subdivides space into two same sized cells along a perpendicular plane. The plane is placed in the 'middle', i.e., it subdivides space into two half spaces. The second type of BSP tree is polygon-aligned, which means that an arbitrary splitting plane is chosen, which can include a polygon. This plane does not have to be axis-aligned.

Using axis-aligned splitting planes leads to advantages during ray tracing. The ray-intersection tests can be replaced by a simple position check, on which side of the splitting plane the eye point lies. This method is less time consuming than the ray-intersection test. In addition, it reduces the computation costs of the signed distance by about three times [Hav00]. Often the order of axes is changed regularly, e.g., according to the x-, y- and z-dimension [Kap85].

The Kd-tree was introduced by Bentley in 1975 [Ben75]. In contrast to an ordinary BSP tree, Kd-trees can have splitting planes that do not have to produce two equal sized cells. So the axis-aligned plane can be placed arbitrarily along a dimension. A Kd-tree can be called a generalization of a BSP tree.

In a Kd-tree there are three different types of nodes. One *root node*, *interior nodes* and *leaf nodes*. If a leaf contains at least one object like a set of polygon(s), it is called *full leaf*; otherwise it is called *empty leaf*. An interior node always has two children representing the two parts of the split space. There are many papers examining different heuristics where and how a splitting plane has to be placed. The most famous one is called *SAH* (Surface Area Heuristics [MB90] [PGSS06]).

Another interesting paper was written by Tim Foley and Jeremy Sugerman [FS05], where the

use of a Kd-tree as an acceleration structure for a GPU ray tracer is examined. Kun Zhou et al. published a work about the construction of Kd-trees on graphical hardware in real-time [ZHWG08].

2.2.2.3 Octree

The last spatial subdivision mentioned is the so called octree - the 3D expansion of the quadtree in 2D. For this work this is the most important one and will be explained in more detail in chapter 4. There are interesting papers, which introduce and deal with octrees [Gla84] [Kap85] [Gla89] [Sam89] [Kno08].

As a difference to the already introduced BSP and Kd-tree, an interior cell of an octree does not have only two, but eight children called *octants*. Like in a BSP leaf, nodes can be called *full* or *empty*, if at least one object is assigned or not.

While going through an octree, the most time consuming work is to determine, which of the eight children of an interior node is intersected. If more than one is affected, the order must be determined in increasing distance to the eye point. The ray can intersect any number of children from 1 to 4, whereas rays, which lie on the border between children, are explicitly assigned to one child. This step is the most important and most time consuming part of a ray traversal algorithm which uses an octree. Vlastimil Havran [Hav99] and Aaron Knoll [Kno08] compare some different ray traversal algorithms with each other.

Octrees are very similar to BSP trees, and they have some disadvantages which are the same as for uniform grids. Space is equally subdivided and when traversed, the ray tracer has to check and pass many empty cells even if, for example, only one of them contains objects. An octree can be realized by a BSP or its generalization. Whereas octrees only need one level, BSP or Kd-trees need three to achieve a comparable result. Aside of having more levels, an octree is a special kind of a Kd-tree.

An expansion of the octree is the Octree-R [WSC+95] which is the generalization of the octree. The splitting planes do not have to be in the middle; they can be chosen arbitrarily, and so the children cell sizes can differ from each other. The Octree-R is related to the octree like the Kd-tree is related to the BSP tree. This expansion can speed up the traversal of an octree, but due to the factors per cell and dimension, which have to be stored, more memory is needed.

An approach to speed up octrees are **Ropes** [MB90] [KB00] [HBv98]. Ropes are additional connections between neighboring nodes, which simplify the traversal of an octree. Ropes are calculated in a preprocessing step; there is one for each face of an octant. They reduce the number of cell intersections, but instead the exit face must be found. Ropes are discussed in detail in chapter 4.

2.3 Methods of Tracing Rays through Data Structures

This section contains an overview of tracing methods. Some methods fit better to certain objects or scenarios than others. Thus switching them during ray tracing can lead to a better performance. Vlastimil Havran calls this methods *RSAs* (Ray Shooting Algorithms) in his PhD thesis [Hav00].

To shoot rays through an octree, a closer look at the octree itself has to be taken. If each leaf has the same depth (the tree is called complete), the RSA would be less complex because it is not necessary to test if a node is already a leaf. A node can be addressed by a simple calculation. But complete trees have a big disadvantage: Empty regions have to be stored and moved to the graphical unit without containing information. It also consumes much time to find the closest child (seen from the eye point). Each child has to be intersected with the ray and sorted in ascending order according to their distance to the eye point.

There are several RSAs which can be used to traverse an octree. In chapter 4 a more detailed description will be given, as this is an essential part of this work. A short overview of important terms, RSAs and previous work follows:

- **Basic Tree Traversal**

  First some basic terms of tree traversal are mentioned [FP02].

  - *Point Location*
    Find the leaf cell containing a given point.
  - *Region Location*
    Find the smallest cell that contains a given region.
  - *Neighbor Searches*
    Find the neighbor of a given node or cell touching it in a given direction.

- **3D DDA**

  Kelvin Sung describes an algorithm traversing an octree based on a 3D DDA [Sun91]. His algorithm works on a uniform voxel space and therefore requires a complete octree to calculate the address by a hash function.

- **Bottom-Up Based RSA Approach**
  A bottom-up approach first finds the closest intersection node of the object. To obtain the next nodes, a process called *neighbor finding* is used [Sam89].

- **Top-Down Based RSA Approach**
  Starting from the root node each node is intersected until the path leading to the next leaf node is found [RUL00].

- **Stack-Based and Stackless RSA Approaches**
  Using a traversal stack can simplify the ray shoot algorithm. In a stack, nodes, which are intersected but not yet used, can be stored because at least one node is located in front of it. These nodes can have child nodes again, which must be handled before its parent's siblings.

  If no stack is used, some calculations must be done multiple times, which leads to multiple memory accesses [PGSS07]. This means, stack-based methods can be an advantage, if enough memory is available. Another idea is to use stacks not for the whole tree, but only for some parts [HSHH07].

- **Ropes**

  Breaking the acyclic form of an octree and adding ropes makes it easy to find neighbor nodes by looking them up [MB90] [KB00] [HBv98].

2.4 Acceleration Techniques

The most complex and time consuming tasks of a ray tracer are the intersection tests [Bou05]. If an acceleration structure is used, mainly *Ray-Polygon* and *Ray-Box* intersections, but also *Ray-Sphere*, *Ray-Plane* and *Ray-Line* intersections (e.g., for debugging) have to be performed.

Martin Eisemann et al. [EGMM07] introduce a method for the intersection between a ray and an axis-aligned bounding box (*AABB*). They test in two dimensional space by using ray slopes. In his paper several methods of Ray-AABB intersections are compared to each other. Williams et al. [WBMS05] show an implementation of a Ray-AABB method which does not have numerical problems with slopes near zero along the axes.

2.5 GPU and Parallelism

To build an image, a ray tracer has to send many rays. Each of them can be seen as an independent single task - the values of its neighbors are not important for one ray. Thus it makes sense to parallelize the ray shooting [Lef93]. Modern hardware allows to execute this on the graphical unit [PBMH02] [AK10]. Recent attempts try to achieve real-time performance [GPSS07] [AL09] [GL10]. Kun Zhou et al. [ZHWG08] are using the GPU to build a Kd-tree in real-time, whereas Iliyan Georgiev et al. [GRHS08] try to realize a whole real-time ray tracer.

3. Scenarios

As mentioned in chapter 1, a multi-field hybrid ray tracer - the actual result of this work - can be useful in many fields of application. Pictures of possible scenes can be found in chapter 7.

3.1 Industrial Application

In this scenario an application example from the industrial sector will be described. A work piece built up from plans in a 3D design software is loaded. As the data is not scanned, this model consists of polygons. If visualizing some fluids in or around this piece, it can be combined with some particles. The polygons of the work piece can border them. Making the work piece or some polygons of it transparent gives the possibility to look inside easily.

3.2 Video Games

Video gaming is on its way to be one of the most popular field of application for a real-time ray tracer, because worlds can be rendered very attractively and realistically which is often desired by gaming communities. Ray tracing supports a lot of visual effects that can be a real eye candy for users. For example, a game world including some mobile objects on a terrain using effects, like shadowing, clouds, transparency, aso. can be rendered. This scenario includes less multi-field data but more effects which can be realized by a ray tracer.

3.3 Virtual and Augmented Reality

A hybrid multi-field ray tracer can also give lots of possibilities in virtual and augmented reality applications, where tracked or scanned objects have to be combined with artificially generated ones. Simulating the world as it is in a photo-realistic manner is desired in this field.

3.4 Archeology

Archaeological discoveries, often damaged by decomposition, can be reconstructed virtually. Data of scanned fragments, stored, e.g., in a 3D grid, can be combined with a polygon mesh easily, indeed not fully automatically but with the assistance of a specialist. Reconstruction and visualization of archaeological discoveries can help to give an insight into the past and help to preserve our cultural heritage.

3.5 Photorealism

Photorealism in general is highly demanding, especially, for example, in creating pictures for commercials. A fast real-time ray tracer can also be used for animation movies which are getting more and more popular. Considering a still life including transparent objects like bottles or glasses, effects like reflection and refraction let them look real. Liquids inside the objects can be implemented by adding particles or voxel volumes.

3.6 Architecture

Like in the industrial sector, a fast preview of objects can be of interest. Architects can use the ray tracer to visualize buildings with an emphasis on, for example, shadows in and around the objects. Shadows can affect the growth of plants around buildings and the other way round plants can affect the light situation inside buildings. Clouds represented as particles can also have an influence.

3.7 Visualization of Medical Data

Another very interesting field for the use of this ray tracer is a scenario from the medical sector. There are several different scanning methods like computed tomography (CT) or magnetic resonance imaging (MRI). Each method is used to take data of human body parts. The results of CT and MRI are three-dimensional images consisting of voxels. Using the steering geometry approach, which is introduced in this thesis (see chapter 4.1.3), these voxels can be combined with polygons representating bones.

3.8 Fluid Simulation

Finally we present a scenario, which can affect many decisions: Handling and simulating floodings, especially in cities. Many buildings composed of few polygons are located in floodwater. Again, the fluid (water) consists of particles. In this scenario effects are not very important. The challenge in this field of application is the real-time rendering of huge scenes.

4. Algorithms

In this section some thoughts and ideas about alternative approaches of ray tracing different types of data sets, data structures and optimizations of algorithms are mentioned. The implemented algorithms will be shown and discussed. Results and evaluation numbers will be presented in chapter 6. The following subsections are related to the first three categories for a ray-tracer software, described by Andrew S. Glassner [Gla89] (see chapter 2). These categories are *Geometry*, *Optics* and *Optimization*.

4.1 Geometry

To handle different types of data, they have to be stored in proper structures to move them easily to the graphical unit. Questions are: What has to be uploaded and how can it be stored? It is not necessary or perhaps not possible to move all data to the graphical hardware, especially when it does not support enough memory. Thus it must be partitioned and only those parts moved, which are necessary at the moment.

The task is to handle different data types while tracing along rays. The ray tracer has to know which type of primitive or object it has to trace at the moment. Three different approaches are considered to solve this task. Figure 4.1 shows two different scenes, which are used to explain them.

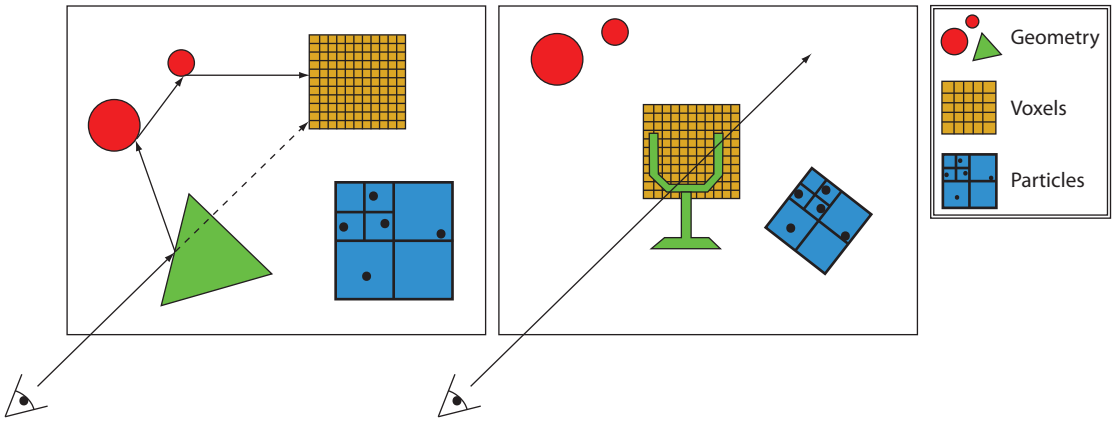


Figure 4.1: Simple scenes

On the left side there are different objects spatially strictly separated. The ray, which is shot from the eye, hits the green triangle, the red circles (both geometrical objects) and finally intersects the brown colored voxel grid. The scene on the right side shows a more sophisticated case. The objects are overlapping each other, therefore, switching the method of tracing rays is much more challenging. The simple green triangle on the left scene were replaced by a more complex object. This object is like glass with transparent triangles, so that the ray can enter it (ignoring refraction). The voxel grid contains, for example, a liquid, which should be present only in the glass, but the dimension of the volume exceeds the bowl.

4.1.1 Separated Objects

As shown in figure 4.2 each object has either its own boundary volume or a complete bounding volume hierarchy (particles). For every new ray, which will be shot, the tracer must intersect all of these bounding volumes and take the closest one to go into detail. Figure 4.2 shows that the ray starting from the eye intersects the bounding sphere of the green triangle and the voxel grid on the upper right. In the scene on the right side the ray also intersects these two bounding spheres. From the shown eye position the order of intersections is correct, the ray will hit the boundary of the glass first. After hitting, a new ray will be shot from the inside of the glass which means also inside both boundary volumes. The ray tracer has to decide which one is next.

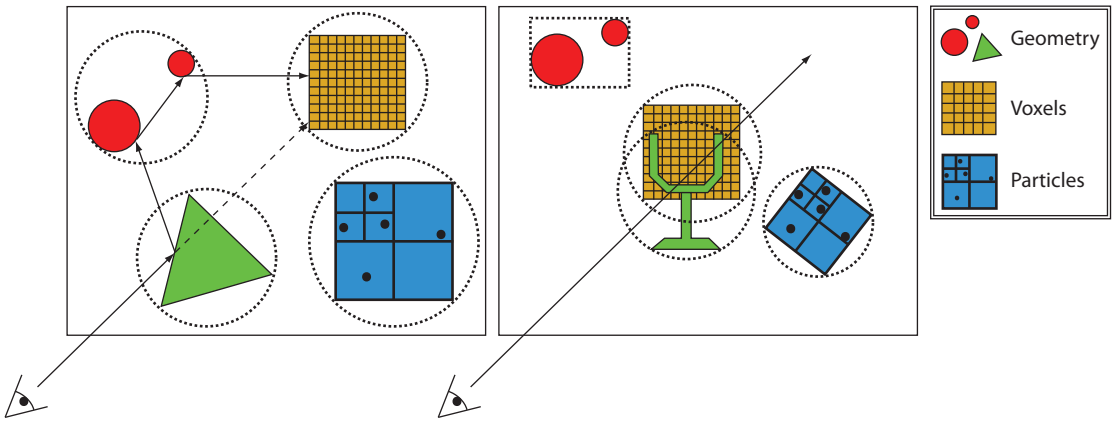


Figure 4.2: Simple scenes with separate bounding volumes

4.1.2 One Global Tree

Another approach is to hold the objects of the scene in one tree. This solves the problem of tracing scenes shown on the left side of figure 4.3 as well. Each node of the tree includes a few properties of the type of objects it contains. So switching is no problem. If objects are spatially not separated they have to be split into smaller parts. This method works, but rebuilding the

tree with consideration of possible animations would be time consuming and complex. If, for example, a polygon moves into a voxel grid, it must be voxelized. Even if this operation is possible, it is very costly.

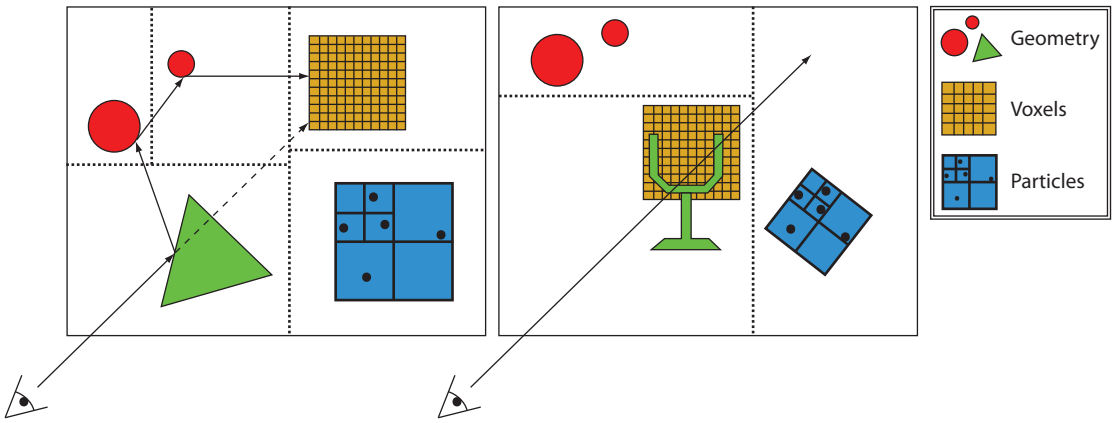


Figure 4.3: Simple scenes with global tree

4.1.3 Steering Geometry

The third approach, the *steering-geometry approach*, which is introduced next, was implemented and presents the core of this work. The scene is expanded by a set of polygons which isolate the different sets of data. In this work these polygons are triangles, so called *steering triangles*. The number of these triangles must be as small as possible, so they can be updated fast and the number of intersections between the ray and the steering triangles can be minimized. Each triangle contains information about the region in front of it and on its back. If a new ray starts, it has to be intersected with all of these steering triangles.

Figure 4.4 shows the already known scene from above. It is obvious that the scene on the left side, where all different sets are spatially separated, is easy to handle for this method. With a few steering triangles the voxel and particles can be split from the polygons. The figure on the right side shows how steering triangles can handle a complex scene. A few steering triangles, shown as dashed lines, isolate the different sets from each other (the red and green polygon sets, the blue particles and the brown voxel grid).

Depending on what the scene designer imagines, the steering triangles must be adapted. Here the volume should be inside of the glass. The steering triangles must be set manually for now. It would be another interesting project to implement a graphical tool to support designers. The next idea is to set the polygons located inside of the bowl as steering triangles. This leads to an amount of steering triangles which would be too high. Thus the inside of the glass has to be approximated like shown in figure 4.5.

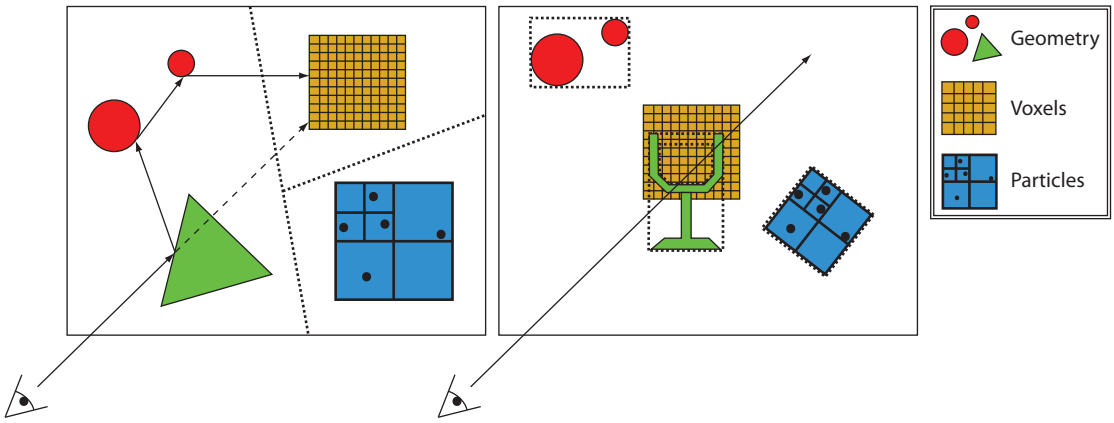


Figure 4.4: Simple scenes with steering geometry

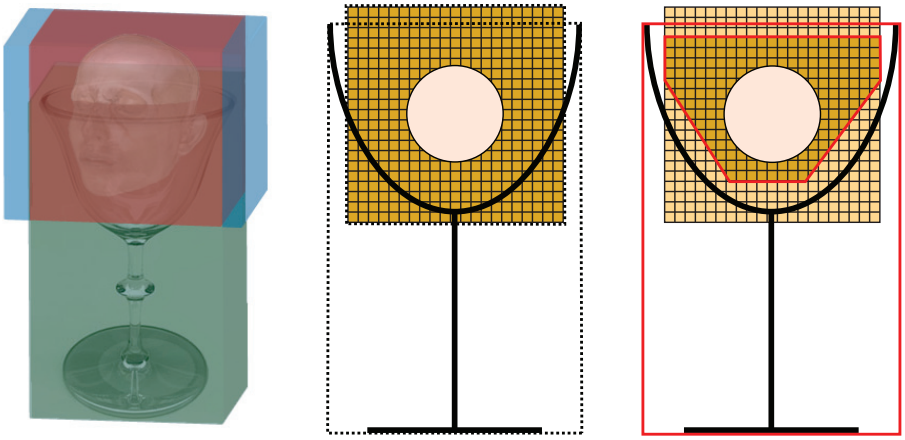


Figure 4.5: The steering-geometry approach: Scene: *Head in Glass*. Left: A photo montage. Middle: The photo montage as an abstracted scene. Right: The adapted steering triangles are colored in red.

In figure 4.6 two rays and their way through the glass scene are visualized. Both rays intersect four steering triangles. Whereas ray 1 enters the glass at its bowl and leaves it at the top without hitting another polygon of the glass the second one does it vice versa.

The traversal is quite simple: First the next pair of steering triangles (i.e., the next two intersected triangles), which is called *steering-triangle interval* has to be found. Then the appropriate tracing algorithm for the current area is activated. Ray 1 intersects in its first steering-triangle interval a glass polygon. If there is an intersection, which does not lie in this interval, it can be skipped. Ray 2 shows why: The first intersected polygon is located behind the voxel set, it must be handled later. Algorithm 1 presents the steering procedure in pseudo code.

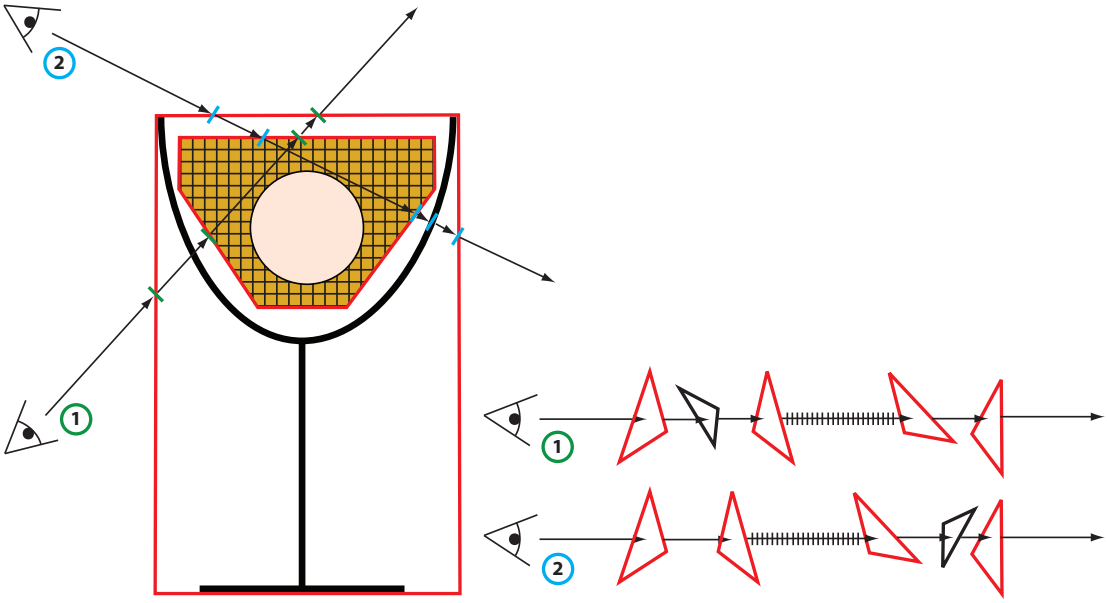


Figure 4.6: The steering-geometry approach: Two rays shot through the scene *Head in Glass*. The steering triangles are colored in red.

The steering-geometry approach seems to be the most promising and interesting one. The algorithm is easy and efficient to implement, it is not too complicated and therefore error-prone. Due to the high cost of triangle-ray intersections the number of steering triangles has to be small. This implies that boundaries consisting of many polygons can only be approximated roughly.

Algorithm 1 Tracing rays using the steering-geometry approach

```
1: ray ⇐ primaryray
2: s0 ⇐ getNextSteeringTriangle(ray)
3: if s0 is null then
4:    exit() {terminate algorithm}
5: end if
6: while true do
7:    ray.origin ⇐ s0.intersectionPoint
8:    s1 ⇐ getNextSteeringTriangle(ray)
9:    if s1 is null then
10:      interval ⇐ ∞
11:   else
12:      interval ⇐ calcInterval(s0, s1)
13:   end if
14:   if s0.type is 'polygons' then
15:      ret ⇐ tracePolygons(ray)
16:      checkIfInsideSteeringInterval(ret.intersectionPoint, interval)
17:   else if s0.type is 'voxel' then
18:      ret ⇐ traceVoxel(ray, interval, stepSize)
19:   else if s0.type is 'particles' then
20:      ret ⇐ traceParticles(ray, interval)
21:   end if
22:   s0 ⇐ s1
23:   if checkExitCriteria() then
24:      exit() {terminate algorithm}
25:   end if
26: end while
```

4.2 Optics

Special optical effects like shadows, self occlusion, reflection and refraction are simple tasks for a modularly designed ray tracer. From each ray-intersection point a newly defined ray will be shot. Thus, for example, shadows with self occlusion can be simply realized by shooting an additional ray from the intersection point to the light source(s). The same can be done with reflections, the primary ray is just reflected at the intersection point using a given surface normal. If the material of the intersected object is not fully opaque, the ray can also be split-up or refracted.

These effects were implemented by using the equations already described in chapter 2. Setting a new ray direction and starting point causes numerical problems. If the calculation of the new ray origin has small rounding errors, the object can shade itself spuriously (as figure 4.7 shows). While setting the origin before the exact intersection point causes no problems, putting it behind

the point leads to wrong results. This effect of self-occlusion is called *shadow acne* and can be fixed by using an epsilon. The new origin is calculated as follows:

$$o = i - \epsilon * \hat{d} \tag{4.1}$$

where:

- o is the new origin,
- i is the intersection point,
- ϵ is a positive number near zero and
- $\hat{d}$ is the normalized direction vector.

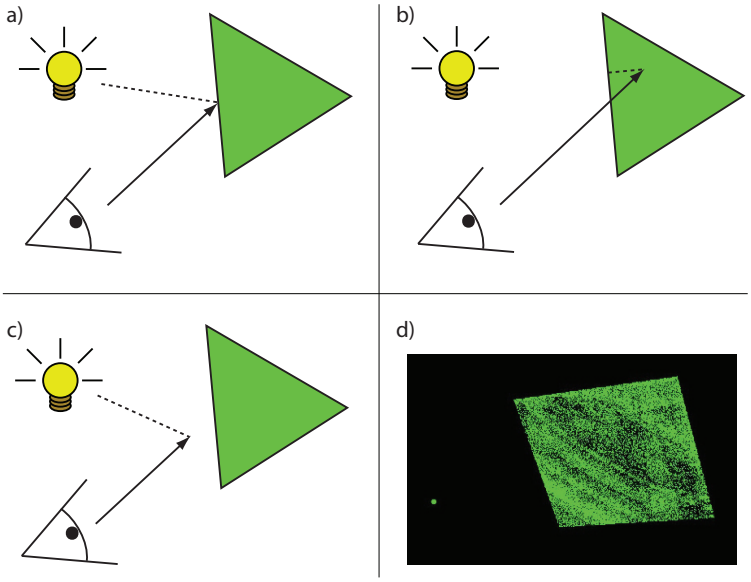


Figure 4.7: Numerical problems in calculating intersection points. a) Shadow ray from point of intersection. b) Numerical problems, object itself is between light source and point of intersection. c) Using equation 4.1 to avoid numerical problems. d) A green light source on the left side and a polygon with shadow acne on the right side.

4.3 Optimization

In this section some ideas and algorithms about testing and optimizing ray tracing are presented. These optimizations are very important. As a slow routine, e.g., for testing intersections or traversal through a data structure, can lower the frame rate drastically, most of the work for this ray tracer was invested in this field of work. This section has a few subsections. First of all

the tree structure used to hold objects, which consist of polygons, will be introduced. This tree was expanded by ropes with the aim to traverse faster through the objects. Afterwards some different approaches to traverse the data structure are discussed and finally some information about handling other sets of data are given.

4.3.1 Acceleration Structures

In chapter 2 different kinds of acceleration structures were already listed. In the following section the octree is revisited and described in more detail. It was implemented and expanded by ropes, which were also mentioned before. Building up the octree was quite complex, so this process is explained as well as the build-up of the ropes.

4.3.1.1 Octree

For convex boundary volumes containing objects, which are built out of polygons, an octree seems to be a suitable data structure for this project.

Before **building up** an octree some criteria have to be considered for the termination of the algorithm:

- The current node is not intersected by any polygon.
- A given depth is reached.
- A threshold is set. If the number of polygons is falling below this value, the algorithm can terminate (see chapter 6).

Building up the octree is made in two steps: First a pointer tree of objects, which is stored in main memory, has to be created. To move and finally use the data on the graphic hardware, it needs a second step, in which the octree has to be put into arrays. Figure 4.8 shows the first step. A simple polygonal object is located on top of figure 4.8 and its resulting pointer tree (*Quadtree Graph*) is shown at the bottom. The object consists of nine vertices which can be seen in table *Vertices* included in figure 4.8. The coordinates of the vertices are not relevant to describe the build-up of a tree, but their indices are. These indices are equivalent to the array position of the vertices.

While none of the listed termination conditions are satisfied, the space is split into four child cells. The bounding boxes of each of these cells have to be intersected with each triangle of the parent node. If a triangle intersects a bounding box it must be assigned to the dedicated child node. In figure 4.8 the eight triangles are listed in table *Triangles*. To avoid storing triangle

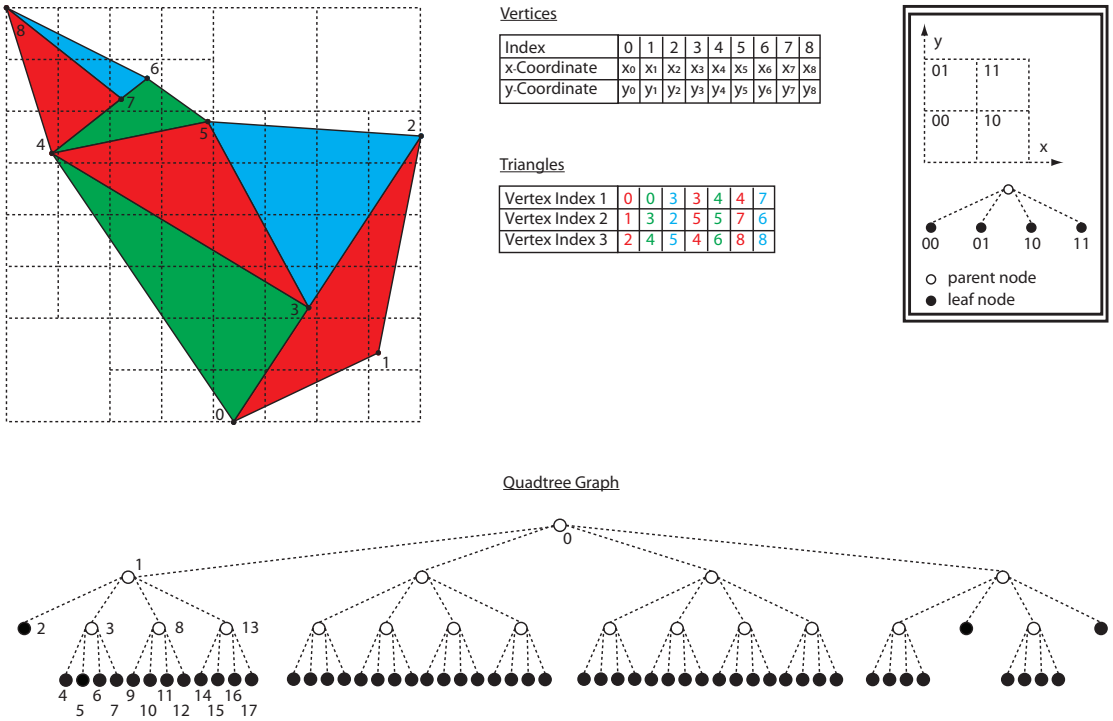


Figure 4.8: A quadtree. Top: A simple polygonal object with vertices and triangles stored in arrays. Bottom: The constructed quadtree shown as a graph.

vertices redundantly, pointers are used. These so called *indices* are integer values. A triple of indices in anticlockwise order defines a triangle.

As mentioned before, in a second step this pointer tree has to be flattened into an array of tree nodes. Each node includes the following information:

- a pointer to its parent node,
- pointers to its child nodes,
- a pointer to the start position in the polygon array (in the example of figure 4.9 it is called the *Quadtree Triangle Array*) and
- the number of polygons it contains.

An additional useful information is the type of a node (*root*, *interior node*, or *leaf*), which simplifies the traversal algorithm. The array of tree nodes is built in preorder. Therefore, the array positions of the children is always higher than the position of their parent. The reason of this order is obvious; it simplifies updates and cutting off entire branches.

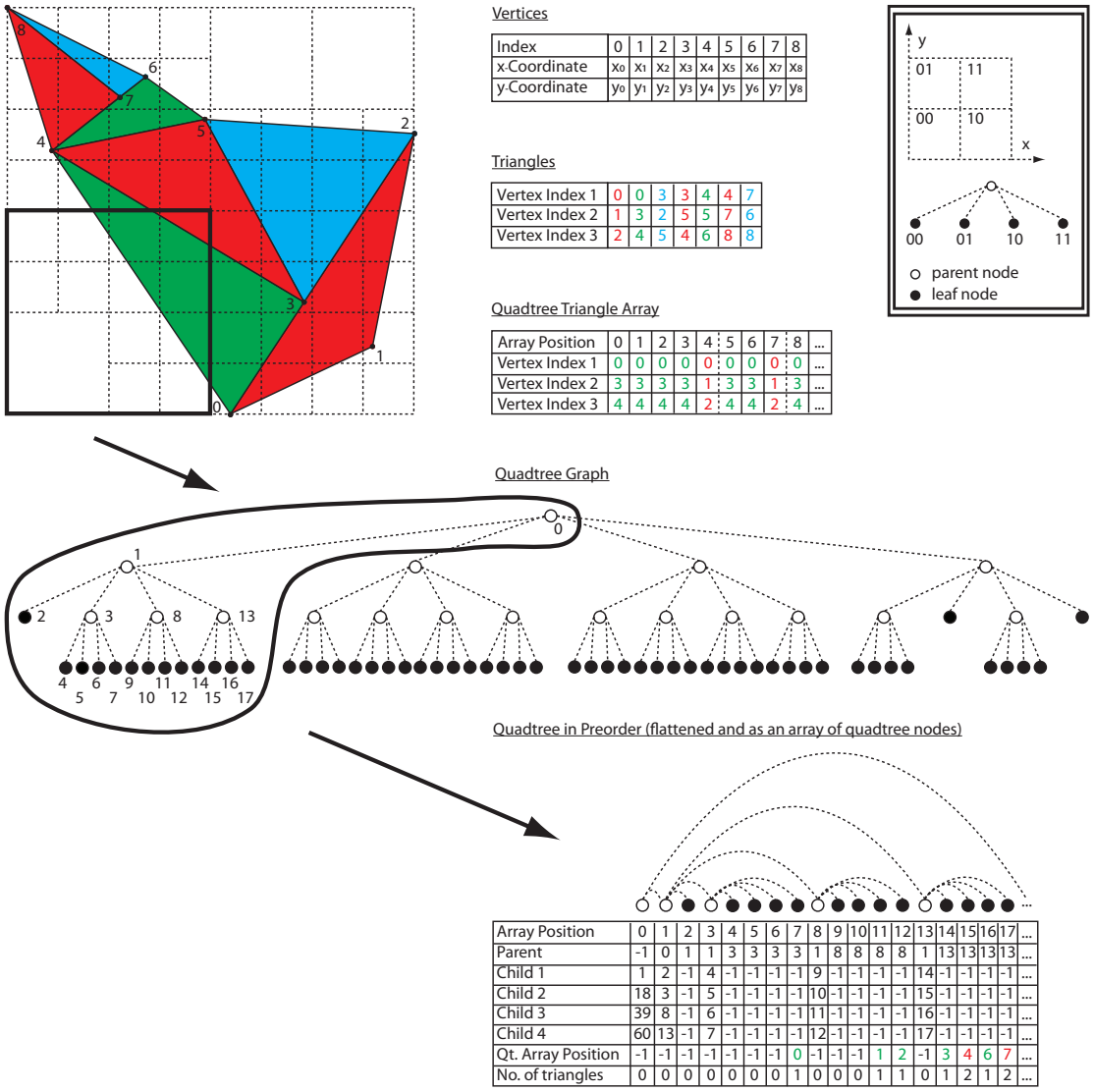


Figure 4.9: A part of the *Quadtree Graph* is flattened in preorder to store it into an array of quadtree nodes. The *Quadtree Triangle Array* contains the triangles which are assigned to the quadtree nodes.

In figure 4.9 a part of the quadtree is outlined. This black frame corresponds to the black marked part of the *Quadtree Graph* in the middle of figure 4.9. The nodes are numbered in preorder. The row of nodes (on bottom) is the result of the flattening. Beneath the array of quadtree nodes is shown. The *Array Positions* correspond to the numbers of the nodes above. The array of quadtree nodes contains, beside its array positions, the parent of each node, their children, the start positions in the *Quadtree Triangle Array* (*Qt. Array Position*), and the number of triangles which are assigned to each node (*No. of triangles*). To render the black part only, the array of

vertices, the array of quadtree triangles and the array of quadtree nodes are necessary. They have to be moved to the graphical device.

During testing of this octree structure a problem occurred. Due to the recursive algorithm and the redundant indices the memory often ran low. This problem can be solved by building smaller subtrees first and storing only the resulting arrays, instead of keeping the whole pointer tree inside the memory. There are different methods of traversing an octree. Before discussing them, another problem is mentioned.

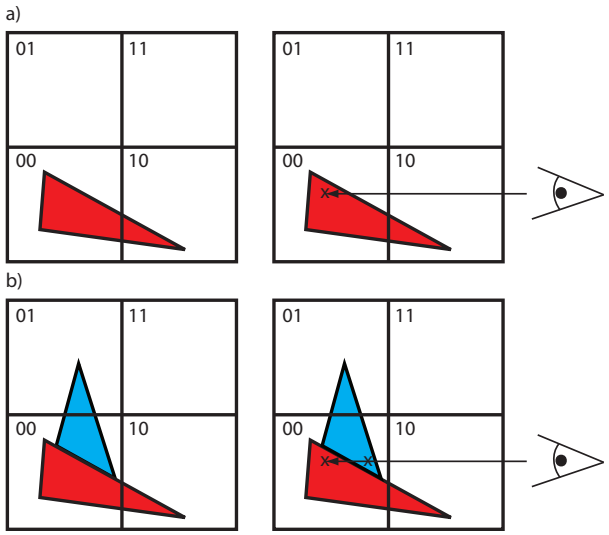


Figure 4.10: Handling overlapping polygons. a) The red triangle is assigned to multiple cells. b) The blue triangle has to be intersected first.

Usually polygons overlap multiple cells as shown in figure 4.10. They have to be assigned to each of these cells. This leads to an additional effort while performing the RSA. Each intersection point, which was found, has to be checked if it really lies in the current active cell. Figure 4.10 shows an example. The red triangle in figure 4.10a) overlaps the cells *00* and *01* and is assigned to each of them. The ray first enters cell *10* and afterwards cell *00*. When cell *10* is active the intersection test between the eye ray and the red triangle is performed and an intersection point is found. This intersection point lies inside cell *00* which is not active at the moment. Figure 4.10b) shows the problem of such intersection points. A blue triangle is added to the scene and located in front of the red one and is not assigned to the first active cell *10*. Thus the eye ray should hit the blue one first, but it does not, because the red one is intersected in cell *10*. To avoid these false intersections, each intersection point has to be checked if it is localized in the current active cell.

4.3.1.2 Ropes

Ropes extend an acceleration structure like an octree by additional connections. Nodes of trees do not have only directed parent-child connections, they also have additional connections to their neighbors. These neighbors can be at any height of the tree (except at a height of 0). So in terms of graph theory the former directed acyclic graph becomes a directed graph with cycles inside as shown in figure 4.11.

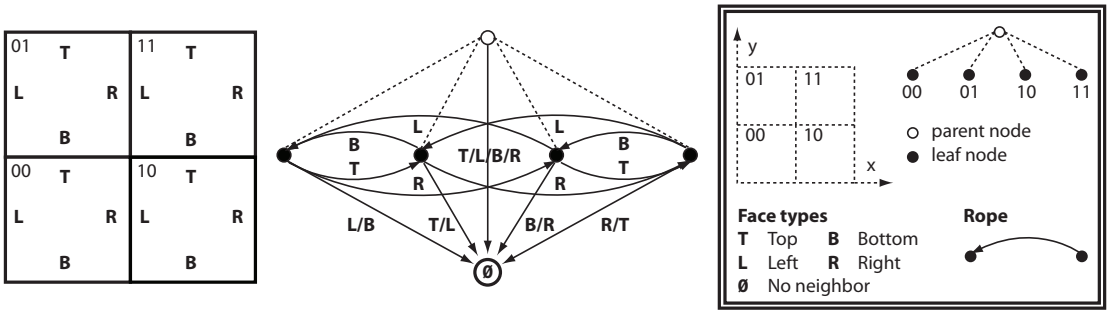


Figure 4.11: The ropes of a simple quadtree. For each face of a cell a connection is drawn to its neighboring cell.

Figure 4.11 shows a simple example of a quadtree and its related ropes. On the left side a simple quadtree is visualized including the order of the cell face's labels. Its corresponding quadtree graph is shown next to it by using dotted lines. The directed connections are the related ropes. Each node has four of them, one for each cell face. If there is no neighbor, because the cell lies at the outer border, the related connection points to nowhere. To visualize each rope of each node in the graph of figure 4.11, a special node called *No neighbor* is included. The root node always contains four connections to this special node. Multiple connections to the *No Neighbor* node are merged to a single one and named appropriately.

Ropes are an additional effort. To build them in a preprocessing step, means, to spend much time on calculations. The advantage of ropes is the simplification of the ray-tracing process. When neighbors are known, it is possible to traverse through the object without time consuming calculations and most notably without using stacks allocating a lot of GPU memory. However, most complex and expensive is the calculation of the exit face, which is needed to read the proper neighbor of the current node from memory. This task is, besides accessing the memory, the most costly one and therefore it has to be optimized.

Build-Up Ropes
Building up ropes is a non trivial task. The build-up algorithm has to perform a procedure to find the neighbor of each face for each cell. This is called *neighbor-finding procedure*. It has to deal with five different cases. Figure 4.12 shows these cases, again using a quadtree to simplify the explanation.

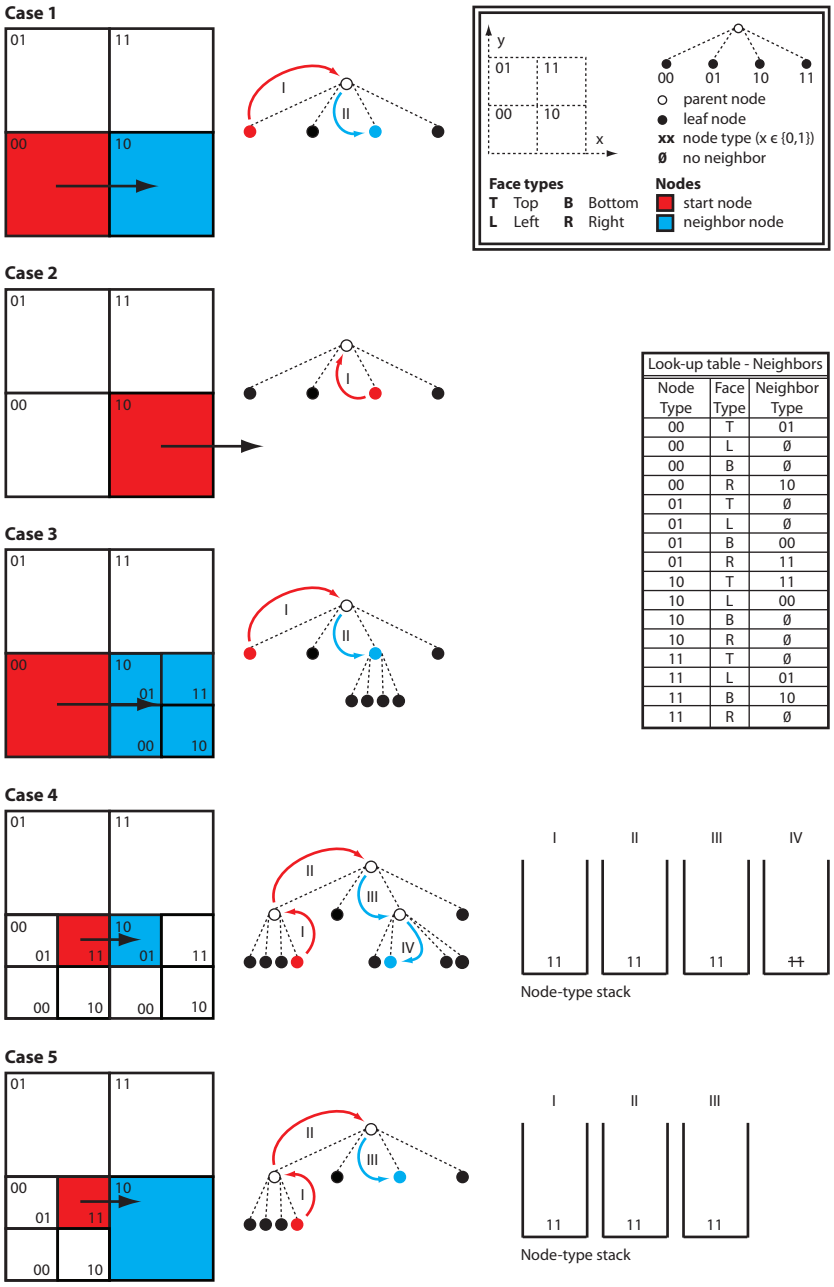


Figure 4.12: Five cases to find the neighbor (colored in blue) of a start node (colored in red). On the left side the five different cases are visualized by using a quadtree. In the middle their respective graphs are shown including colored arrows and roman numbers which indicate the order of node treatment. In *case 4* and *case 5* the *neighbor-finding procedure* requires a stack to store node types. The look-up table on the right side is needed to find the neighbor's node type, given the type of a start node and the type of the face which lies in between.

In case *1* the right neighbor of the red node is the direct child of its parent which type is 10. In the first working step (*I*) the type of the neighbor is determined by the the *Look-up table - Neighbors* on the right side of figure 4.12. In step *II* the child, which is of the determined type, is read from the parent's children list. In this example the parent is the root node of the tree.

Case *2* deals with the case of a start node lying at the border of the quadtree. A right neighbor node does not exist. Table *Look-up table - Neighbors* returns that there is no neighbor to the given type 10. The *neighbor-finding procedure* can be terminated in this case, because the parent of the start node is the root node of the tree.

Case *3* is similar to case *1*. The only difference is that the start node has more than one neighbor and the procedure has to be terminated after working step *II*. Therefore a stack is needed which can store types of nodes. Entries are pushed into this stack until a working step reaches the root node, and are handled (popped) on their path to the neighbor node. Using an empty stack as additional termination criterion, case *3* stops after step *II*, because no node type was pushed onto the stack until the root node is reached. In case *4* and case *5* this stack is needed too, and at least one element is pushed.

In case *4* four steps are necessary to find the neighbor of the red colored start node. The first one involves a look-up for which type is on the right side of a node of type 11. The result would be that there is no neighbor. As described in case *2* the procedure can be terminated returning 'no neighbor found', but this would be wrong in this case. Thus a new termination criterion has to be introduced. If the parent of a node is not the root, the type of the node has to be pushed onto the stack instead of returning 'no neighbor found'. Therefore the type of the start node is pushed onto the stack, which is shown at the right side of the graph of case *4*. In step *II* the root node is reached and the type of the right neighbor of the start node's parent, which is of type 00, can be read from the look-up table. It is of type 10. As described above the next node can be found by using the children list of the root node. The procedure can not stop at this node, because the stack is not empty. Thus it has to push the entry from the stack and finally finds the neighbor node in step *IV* by using the stored node type.

Case *5* is similar to case *4*, but the procedure has to be terminated after step *III*. Even though the stack is not empty, the neighbor node is already found, because the current node is a leaf node.

Algorithm 2 shows the *neighbor-finding procedure* which has to be performed once for each face of an octree cell. The first step is to create a stack which can store types of nodes (*node-type stack*). Starting with a given node it has to be found out if it has an easy to find neighbor (*EasyNeighbor*). If there is one, the algorithm can stop searching and return this node as the appropriate one (see case *1*). If there is no child, the node type must be put onto the *node-type stack* and the procedure has to determine if the parent of the parent has an easy to find neighbor in the appropriate direction and so on. If there is no such neighbor, 0 is returned. The cell must be at the border (see case *2*).

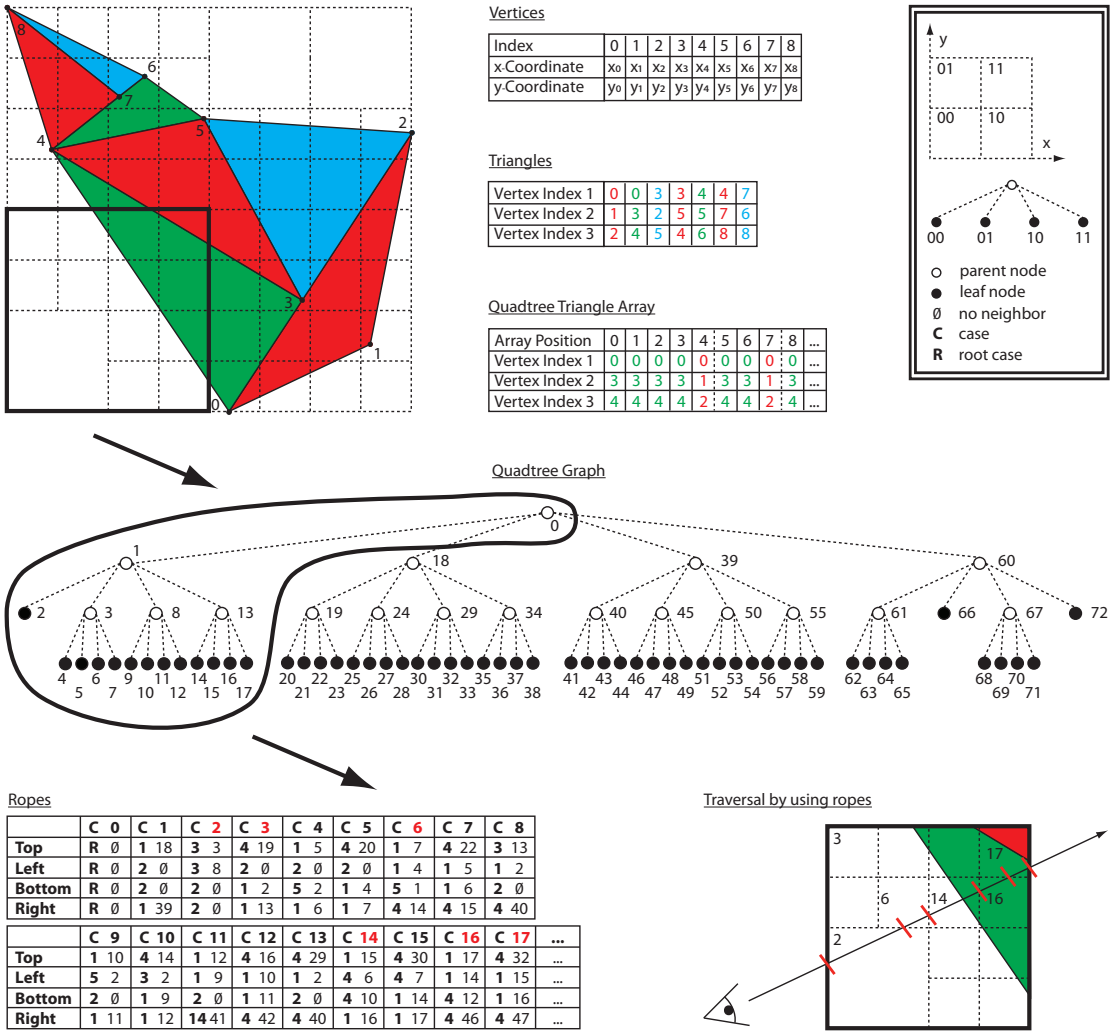
After building up, the *node-type stack* can now be processed. While the stack is not empty the

Algorithm 2 Build-up of ropes: The *neighbor-finding procedure*

```
1: node {the given current node}
2: face {the given face type between current node and the neighbor node}
3: nodeTypeStack {stack to store node types}
4: neighbor {return neighbor node}
5: type {node type}
6: buildStackCondition ⇐ true {to exit the build-up stack loop}
   {Build-up stack}
7: while buildStackCondition do
8:    if node.parent.notExists() then
9:       neighbor ⇐ null
10:      stack.deleteAllElements()
11:      buildStackCondition ⇐ false
12:   else
13:      if node.hasEasyNeighbor(face) then
14:         neighbor ⇐ node
15:         buildStackCondition ⇐ false
16:      else
17:         stack.push(node.type)
18:         node ⇐ parent
19:      end if
20:   end if
21: end while
   {Handle stack}
22: while stack.isNotEmpty() do
23:   type ⇐ stack.pop()
24:   neighbor ⇐ neighbor.getChildOfType(type)
25: end while
26: return neighbor
```

assumed neighbor is the child of the current node which has the type of the top entry of the stack. If there are children, one of them has to be the neighbor of the start node. Otherwise the current node itself will be returned as the proper neighbor (see case *4* and case *5*). To find the appropriate children of the current node, it is necessary to find out which position in the stack is related to the according neighbor node. If the stack is empty, the current node can be returned as neighbor node. If there are multiple neighbor nodes, the *neighbor-finding procedure* returns their parent node. Thus the traversal algorithm has to check if the connected node is a leaf. If not, the algorithm has to find the appropriate child.

Figure 4.13 shows an example of using ropes which have to be stored in the same order as the tree array to simplify access while walking through the object. The ropes of the black marked area are listed in the table in the lower left corner. To reproduce the build-up of the ropes the number of the appropriate build-up case is inserted in column *C*. Some of the node numbers are



	C 0	C 1	C 2	C 3	C 4	C 5	C 6	C 7	C 8
Top	R Ø	1 18	3 3	4 19	1 5	4 20	1 7	4 22	3 13
Left	R Ø	2 Ø	3 8	2 Ø	2 Ø	2 Ø	1 4	1 5	1 2
Bottom	R Ø	2 Ø	2 Ø	1 2	5 2	1 4	5 1	1 6	2 Ø
Right	R Ø	1 39	2 Ø	1 13	1 6	1 7	4 14	4 15	4 40

	C 9	C 10	C 11	C 12	C 13	C 14	C 15	C 16	C 17	...
Top	1 10	4 14	1 12	4 16	4 29	1 15	4 30	1 17	4 32	...
Left	5 2	3 2	1 9	1 10	1 2	4 6	4 7	1 14	1 15	...
Bottom	2 Ø	1 9	2 Ø	1 11	2 Ø	4 10	1 14	4 12	1 16	...
Right	1 11	1 12	14 41	4 42	4 40	1 16	1 17	4 46	4 47	...

Figure 4.13: Using ropes. The table in the lower left corner shows the ropes of the black marked area. On the right side is an arbitrary ray shot through the area. The ray visits node 2, 3, 6, 14, 16 and 17 before leaving the black area. These nodes are colored red in the ropes table.

colored red. These nodes are intersected by the ray in the example at the right side of figure 4.13. The *R* in column *C* denotes a trivial case of the root node without neighbors.

The quadtree traversal procedure by using ropes is simple. The start node is node number *2*. The ray leaves this node at its top, thus the look-up in the column of node number *2* line *Top* leads to node number *3*. As this node is not a leaf, the traversal procedure has to find the ray-intersecting child node, which is the closest one to the eye origin. It is node number *6*. The next visited node is node *14*, followed by node *16*. The final node of the black marked area is node number *17*.

4.3.2 Methods of Tracing Rays through Data Structures

As already discussed, ray-polygon intersections are very time consuming. Thus one has to find a proper acceleration structure. The next step is to design an algorithm to traverse the scene by using the chosen data structure. For this, important issues must be considered. GPU memory is not only needed to store object data (like vertices, materials, acceleration structures, ...), it also has to manage the locally used variables (storing and access). Especially algorithms using stacks need a lot of memory.

In chapter 2 some terms about ray shooting algorithms (*RSA*) were already explained. In the subsections below a closer look on the implemented RSA will be given. Except the first one each of these algorithms uses an octree as acceleration structure. The following list outlines different types of RSA:

- **Straightforward RSA Approach**
  The simplest way to shoot rays into a scene is to intersect with each primitive, for example, with all triangles in a scene. This algorithm is robust and easy to implement but temporally way too expensive. The operation, which costs most, is the intersection calculation. While working on the GPU (NVIDIA CUDA), this often leads to an unexpected problem - if an object or a scene has too many primitives to intersect, the kernel returns a time out. Thus there is no other choice but to find a better way to traverse the scene.

- **Top-Down Based RSA Approach**
  The top-down approach traverses the octree from the root node to the leaves to find the next active node. Beginning from the outer bounding volume the approach intersects the eye ray with each child until the leaf node is found which is the closest to the eye point. This node is now the active one and the intersection with its included objects can be performed. To find the neighbor node, the ray point is set to the exit point of the active cell and the algorithm has to start again at the bounding volume of the root. The ray tracer has to provide the ability to find a leaf cell which contains a given point. This is called *point location*. An advantage is that only little memory is needed but many intersections between the eye ray and interior tree cells have to be processed multiple times. The calculation of the new position of the ray point can lead to numerical problems. If this calculation is not precise enough, it could be positioned just in front of the next node's bounding box. The next intersection test would return the current active node as the new active node. This could happen multiple times. In the worst case the moving point would never leave the active node.

- **Bottom-Up Based RSA Approach**
  The bottom-up approaches try to find the next neighbor directly by using additional tree information or results of the last steps. The algorithms are usually more complex than the top-down approaches but faster.

- *Stackless RSA Approach*
  A simple extension of the top-down approach would be to look up the parent of the active node and to continue from there to find the neighbor cell. This means additional memory costs because the parent of each node has to be stored, but the algorithm does not have to start over at the root node on each step. Thus many intersection tests, which would have to be done multiple times, can be avoided.
- *Stack-Based RSA Approach*
  Using a stack can bring other advantages. Each intersection has to be done only once. If there is more than one child hit, they can be stored sorted and processed later on. Thus sorting has to be done only once, too. And finally the eye point does not have to be moved to assure that a node is not handled multiple times. Hence there are no numerical problems to consider. The disadvantage is that a stack needs much memory space and the size of it has to be precalculated. Equation 4.2 describes how much memory is needed only for the stacks:

$$S = w * h * a * (1 + th * i) \tag{4.2}$$

  where:
  S is the precalculated stack size,
  w is the image width,
  h is the image height,
  a is the byte size of the node address,
  th is the height of the tree and
  i is the maximum number of intersected child nodes.
  Assuming a resolution of 1600 x 1200 (*w* x *h*), this means 1.920.000 pixel and kernel calls. Each kernel has a stack, which at least contains an unsigned integer, the array address of a tree node. This means a is equal to 4 bytes. The worst case, i.e., the maximum number of child nodes, which are intersected by a ray and have to be stored in an octree, is 4 minus 1 - the one, which is used instantly. It can be proven that more than four nodes can not be intersected. Thus i is equal to 3. If now the height of the tree is 5 (th = 5), the result is 122.880.000 bytes.
- *Ropes*
  Another approach to find neighbor nodes uses ropes. Ropes are additional connections inside of the octree. For each cell six pointers are stored - the neighbor cells for each face. The algorithm only looks up which node is the next active one. Therefore no intersection tests between ray and bounding volumes have to be done, but the face has to be identified, where the ray exits the current cell. Ropes need additional texture memory and the build-up process is complex.

The following sub sections will describe the implemented approaches, including a short description, a figure of the traversal and the traversal algorithm. An evaluation can be found in chapter 6. The straightforward and top-down RSA approaches are not useful for fast rendering, so every implemented algorithm is kind of a bottom-up RSA approach.

4.3.2.1 Method 1: Bottom-Up and Stackless RSA Approach

An easy method to traverse an octree is to let the eye point move along the ray. Thus intersections behind the current position of the eye point can be discarded. To find the neighbor of the active cell, the algorithm has to look up the parent to find the next one. Figure 4.14 shows the traversal of a 2D scene using a quadtree. The intersection points between ray and the cell boundaries are colored in red.

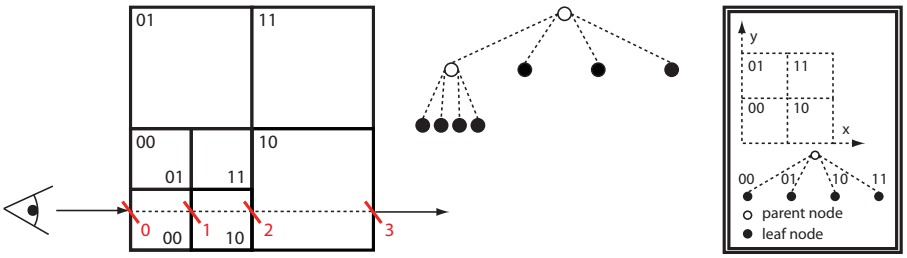


Figure 4.14: RSA Method 1

First the leaf node *00/00* must be found. Then the eye point origin is set on the intersection point *0* - the boundary of the first active node. The next step of the algorithm is to check if some polygons are hit. Afterwards the eye point is moved to the exit point of node *00/00* and the algorithm finds the neighbor node. Therefore the parent of the current node is looked up. If it has another child, which is not handled yet (the ray origin lies on or before its boundaries), this child is set as new active node. If it is not a leaf node, its children have to be handled too.

If the parent node does not have another child, the algorithm simply moves on to the next parent if there is one. It will be set as the new active node. If there is no parent left, the algorithm can be terminated and the ray origin lies at the exit face of the root node (in figure 4.14 it is intersection point 3). Algorithm 3 shows a possible implementation of this approach.

4.3.2.2 Method 2: Bottom-Up and Stack-Based RSA Approach

The next implemented method is based on a stack. This approach has to do each bounding-volume intersection-test only once. Also sorting of the children, if several are hit, does not have to be repeated. Another advantage is that the ray origin does not have to be moved along the ray. It remains at the starting point and thus there are no numeric problems to deal with. By using a stack the algorithm does not have to know which node was already visited. But the memory requirement can be enormous.

Figure 4.15 shows an example of this approach. The bounding volume, which is intersected first, belongs to the root node. Thus the root node is pushed onto the stack. Due to the fact

Algorithm 3 Method 1: Stackless RSA Approach

```
1: ray ⇐ primaryray
2: aN ⇐ root.this {active node}
3: while true do
4:    while !aN.isLeaf() do
5:       aN ⇐ aN.closestChild() {find closest child to eye point}
6:    end while
7:    intersect(ray, aN.polygons)
8:    if checkExitCriteria() then
9:       exit() {terminate algorithm}
10:   end if
11:   ray.origin ⇐ aN.exitPoint
12:   repeat
13:      if aN.hasParent() then
14:         aN ⇐ aN.parent
15:      else
16:         exit() {terminate algorithm}
17:      end if
18:   until aN.hasUnhandledChild()
19: end while
```

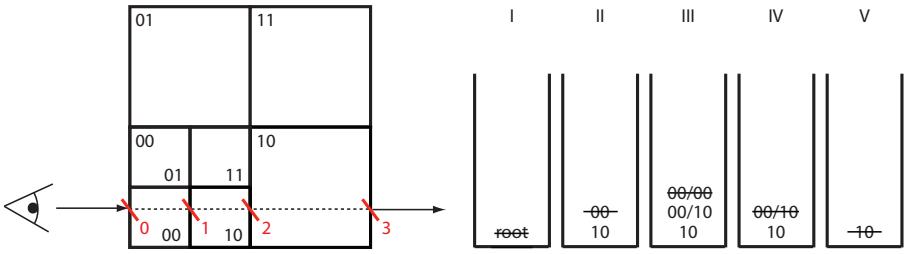


Figure 4.15: RSA Method 2 uses a stack. Left: An eye ray is shot through an octree. Right: The different stack states from *I* to *V*.

that it is used immediately it is popped instantly. This can be seen in stack state *I*, where the immediately popped root node is crossed out. The next step is to intersect the children of the root. As two of them are hit, they have to be sorted. The closest one to the eye point, node *00*, is handled instantly, the other one, node *10*, is pushed onto the stack (stack state *II*). Node *00* is a parent of two intersected children. Again, the closest one (*00/00*) can be handled right away, whereas the second one, node *00/10*, is pushed to the stack (stack state *III*). *00/00* is a leaf node, thus the algorithm checks, if the eye ray intersects the polygons, which are assigned to *00/00*. If the algorithm does not terminate yet it continues with the topmost stack entry (node *00/10*) which is a leaf node. The assigned polygons of node *00/10* have to be intersected and the node is popped from the stack (stack state *IV*). The final stack entry is node *10*. It is popped

and its assigned polygons are checked, because node *10* is a leaf (stack state *V*). The algorithm is terminated, when the stack is empty and every node was handled. Algorithm 4 shows this approach in pseudo code.

Algorithm 4 Method 2: Stack-Based RSA Approach

```
1: aN ⇐ root.this {active node}
2: bV ⇐ root.boundingVolume
3: if !intersect(ray, bV) then
4:    exit() {terminate algorithm}
5: else
6:    stack.push(aN)
7: end if
8: while !stack.isEmpty() do
9:    activeNode ⇐ stack.top() {pops the top element}
10:   intersect(ray, aN.polygons)
11:   if checkExitCriteria() then
12:      exit() {terminate algorithm}
13:   end if
14:   nodeList ⇐ intersect(ray, children)
15:   nodeList.sort() {decreasing distance to eye point}
16:   for i = 0 to nodeList.size do
17:      stack.push(nodeList.elementAt(i)
18:   end for
19: end while
```

4.3.2.3 Method 3: Bottom-Up RSA Approach and Ropes

The traversal of this approach using ropes is not complex. First the algorithm has to find the entrance node by intersection with the hierarchical bounding volumes. When it is found, the number of the intersections between eye ray and bounding volumes of the interior nodes is low, compared to the other approaches in this section. Instead the algorithm has to calculate the exit face of the active cell. After identifying it the neighbor node can be simply looked up in the ropes array. Additional intersection tests are only necessary if the neighbor node is not a leaf node. If there is no neighbor found, the ray exits the volume and the algorithm can be terminated (see algorithm 5).

4.3.3 Tracing Particles

The visualization of particles is realized and implemented in a simple but costly way. They are less important, than the other data sets in this work. Therefore only the distance of each particle to the eye ray is calculated and taken as influence value. Different distributions can be used to

Algorithm 5 Method 3: RSA by using Ropes

```
1: ray ⇐ eyeRay
2: aN ⇐ root.this {active node}
3: bV ⇐ root.boundingVolume
4: if !intersect(ray, bV) then
5:    exit() {terminate algorithm}
6: end if
7: while aNnotNULL do
8:    while !aN.isLeaf() do
9:       aN ⇐ aN.closestChild() {find closest child to eye point}
10:      bV ⇐ aN.boundingVolume
11:   end while
12:   intersect(ray, aN.polygons)
13:   if checkExitCriteria() then
14:      exit() {terminate algorithm}
15:   end if
16:   face ⇐ getExitFace(ray, bV)
17:   aN ⇐ getNodeFromRopes(aN, face) {returns NULL if neighbor does not exist}
18:   bV ⇐ aN.boundingVolume
19: end while
```

blend them together (*Gaussian* or *Cauchy-Lorentz*). Simple monotonically decreasing functions with finite support can be used to reduce the influence of particles lying too far away from the ray. Thus it is possible to render, for example, smooth clouds.

4.3.4 Tracing Voxel Grids

To trace a voxel grid, two methods were implemented, a *front-to-back* and a *back-to-front* approach. The first algorithm needs an interval length, which is equal to the distance between the entrance point and the exit point of the volume, a step size, and an opacity threshold. The *front-to-back* method starts from the eye point and walks from the front of the scene to its back, adding color and opacity until the given threshold is reached. The *back-to-front* algorithm works the other way. It is only used if no opacity threshold is set. The color and opacity can be managed by a simple transfer function.

5. Implementation

This section gives more details about the practical part, the implementation of this work. The framework will be introduced concisely, which is still in development, but which has already been used as basis of the GPU ray tracer. The main topic of this section is the description of the most important classes and kernel functions.

5.1 Framework

In 2008 the development of a new framework called *visdom* was started. The goal of this project was to create a piece of software that would be easy to use and to expand. It should work on many different operating system and also on handhelds. Therefore it was planned as a server-client application. The server has to perform the complex calculations and thus C and C++ seemed to be the right choice. The tracing algorithm is performed on the graphical device of the server. The resulting image is stored as a file (JPEG) and is provided to the client. The client has to be the front end of the system. It manages the input, delegates setting changes to the server and displays the provided image files. For the graphical user interface (GUI) Adobe Flex, i.e., Action Script, is used. It allows to connect nodes of three simple abstract types:

- Producer nodes
  These nodes create data, e.g., reading from files.
- Filter nodes
  Filter nodes change data.
- View nodes
  These nodes finally show the results, for example, of a ray tracer or OpenGL view.

Figure 5.1 shows a small example of the framework GUI. The three nodes *ModelLoader*, *Sample* and *TransferFunction* are producers. They generate data like loading a model file. The *Transformation* node is a filter, which acts on the loaded model data, and on the right side there is a view node, in this case a *Ray-Tracer* node, handling the output. In figure 5.1 this node is called *Ray*). The settings of a node can either be manipulated on the node itself or in an extra panel of the application. Each node has a few pins of different types to connect them to each other. Producers only have outgoing connectors, views have only incoming connectors and filters have

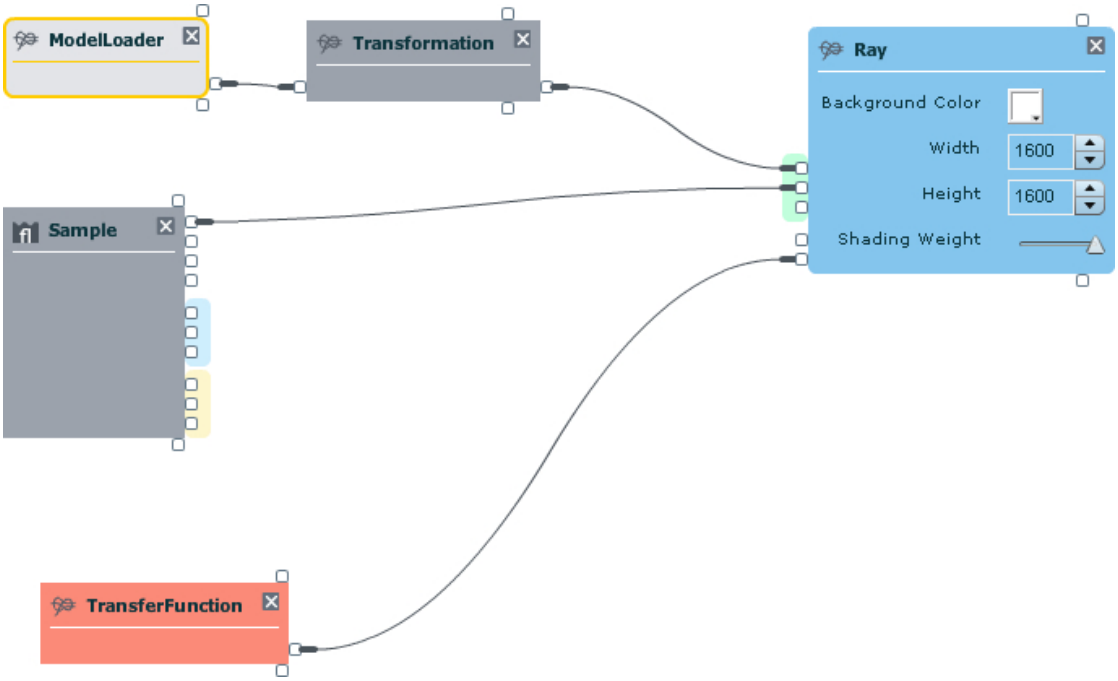


Figure 5.1: The framework GUI: Producer nodes (*ModelLoader*, *Sample* and *TransferFunction*), Filter node (*Transformation*), View node (*Ray*)

both. Each producer creates a *data-flow* object, which is sent along the connectors. In figure 5.1 the *Ray-Tracer* node gets three data-flow objects, one from the sample node (the voxel data object), one from the transfer-function node (the transfer-function data object), and one from the model-loader (the model data object) called *SceneItem*.

5.2 Sourcecode and Classes

The following listed classes and kernel functions give an outline of all implementation work, which has been done for this master thesis. Many helper classes were written as well as an own small math core and lots of geometrical functions, classes and structs.

In figure 5.1 an example of the framework GUI is shown. An outline of the classes, which are needed to implement this GUI setting, is given in figure 5.2. The diagram was designed using *Unified Modeling Language (UML)*. The classes are shown without their attributes and methods. Most of them will be explained closer in the following subsections. The producer nodes, filter nodes and view nodes are specializations of an abstract superclass called *AbstractNode*. Each producer creates a data-flow object which can be manipulated by connected filters. The model

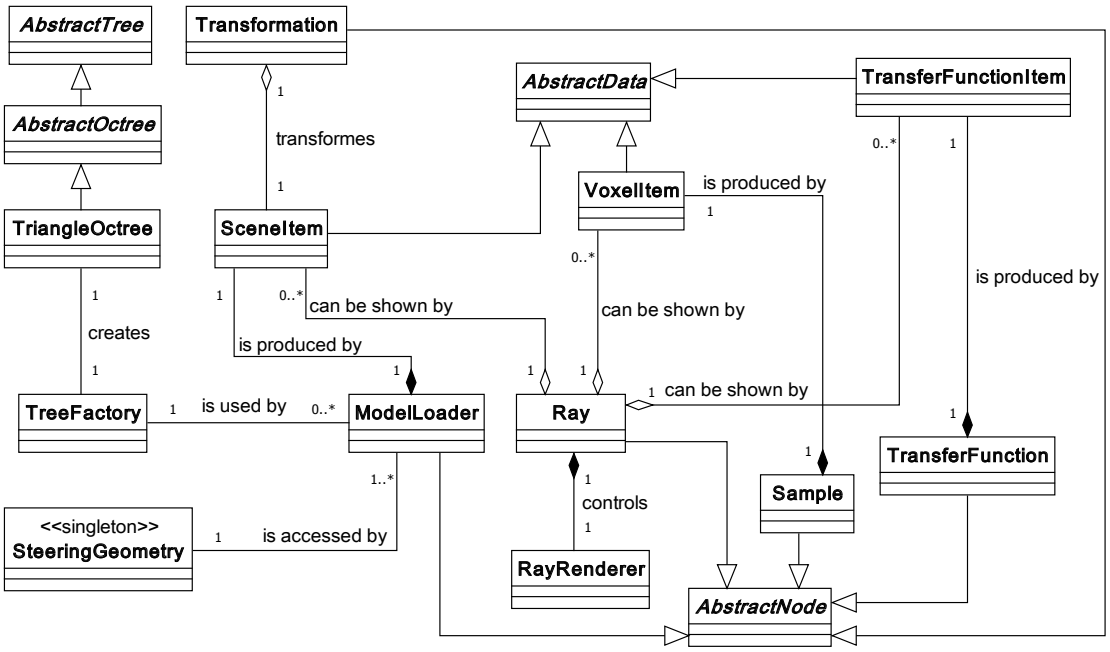


Figure 5.2: A class diagram using Unified Modeling Language

loader can use a class called *TreeFactory* to build up an octree and store it into its data-flow object (*SceneItem*). The singleton class *SteeringGeometry* can be accessed by each *AbstractNode* to add, delete or manipulate the steering triangles. In the example in figure 5.2 it is connected to the model loader.

5.2.1 Scene Item

A scene item is an object generated by the model-loader class. It holds all the properties of the read polygon model like vertices, indices, normals, colors, but also the tree attributes and ropes if they are used. Besides the usual methods, e.g., getters, setters, constructor and destructor, there are some extra functions to handle the memory, where the data is currently stored. The graphical unit is called *device*, the main memory is named *host*. Figure 5.3 shows the class methods which are needed to move the data from host to device and vice versa.

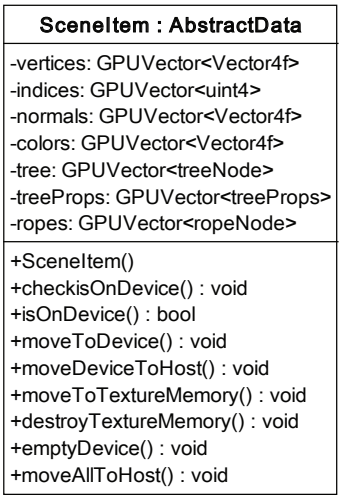


Figure 5.3: SceneItem

5.2.2 Model-Loader Node

Each node is a specialized form of the abstract node class *AbstractNode*. Thus the model-loader node also has its attributes and its methods. But these are less important and therefore not listed in figure 5.4. The model loader opens a given file and loads polygons and attributes into a scene item. There are several different file types the loader can handle. All of them have to contain ASCII data. The class attribute *ModelLoaderSettings* includes all settings sent from the GUI, like building a tree or not. The names of the methods of this class are self explaining. The model loader can use the tree factory directly to push the data into an acceleration structure. Ropes can also be calculated via the tree factory.

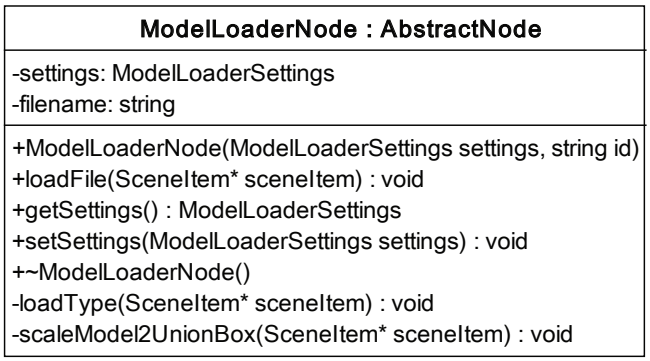


Figure 5.4: ModelLoaderNode

5.2.3 Tree Factory

The tree factory interacts directly with the model loader. It simply needs a data-flow object (a *SceneItem*), which is filled with loaded polygon data. The data-flow object includes the tree properties, e.g., the condition, when the building algorithm has to terminate its process or if it has to create ropes. Figure 5.5 shows not only the method to create a triangle tree, it also includes one for building a particle tree.

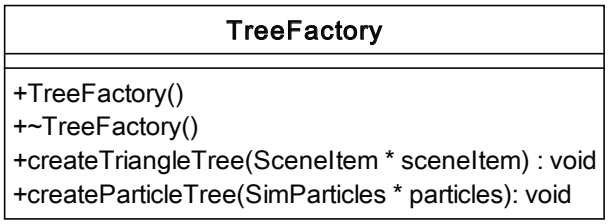


Figure 5.5: TreeFactory

5.2.4 Acceleration Structure - Tree

The three classes shown in figure 5.6 were implemented to manage the model data and to accelerate the handling of it while tracing rays. The acceleration tree is build recursively until it is stopped by certain termination criteria. Each tree node is a specialized *AbstractTree* node and contains a pointer to its children and a pointer to its parent node. The *AbstractTree* provides additional basic attributes, e.g., height of the subtree, the actual depth of the node and some interface methods.

A direct specialization of the *AbstractTree* is the *AbstractOctree* containing attributes like its position in octree, as an example 'left-down-front' coded as '0-0-1', or the number of elements the node is allowed to hold. If there are more items than this given number, the tree has to be split once more. Again there are some interface methods which each specialized class has to implement. Important methods in this class are helpers to find the neighbors of a given node. They are needed to build up ropes. Another specialization of the *AbstractTree* is the *AbstractKdTree* which is not shown in figure 5.6.

The last specialized tree class is called *TriangleOctree*. It contains only one attribute storing the vertices of the triangles. All interface methods have to be implemented. Additional methods check if triangles are overlapping more than one child and therefore have to be assigned to all of them.

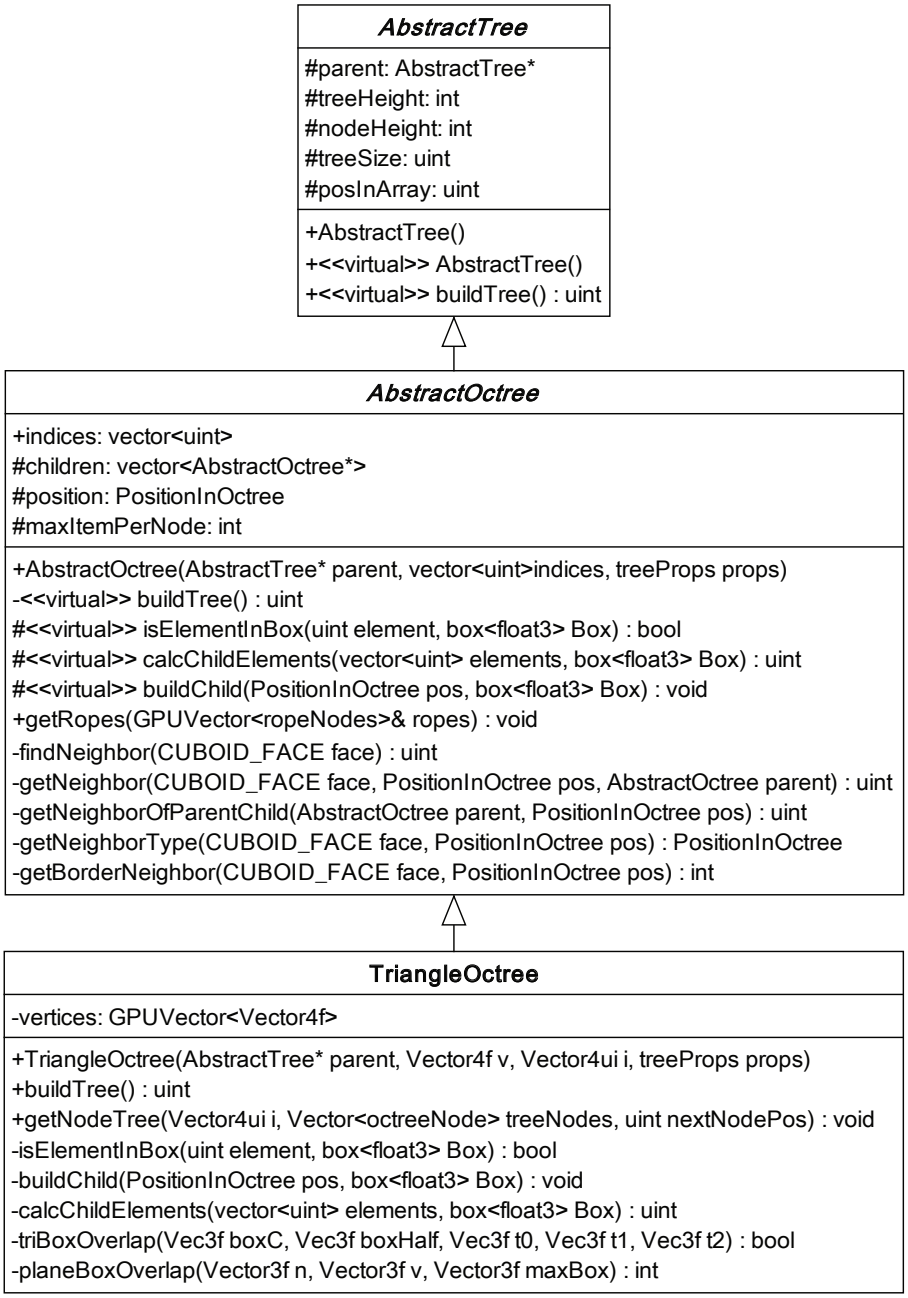


Figure 5.6: Tree classes

5.2.5 Steering Geometry

The steering-geometry class interacts directly with the different producer nodes, therefore it is a singleton class (see figure 5.7). Each producer can add elements to a list of polygons. These polygons are needed to handle the ray tracing. Thus it is defined, when which type of data set has to be shown. The steering-geometry class builds the core of the system. To move triangles and steering data to the graphical memory, a method called *moveToDevice* was implemented. An important difference to the other data needed on the graphical unit (like vertices, tree, aso.) is, that the steering data is not stored as linear texture but as a *cudaArray*. The access to a *cudaArray* is less time consuming, but the amount of space is limited.

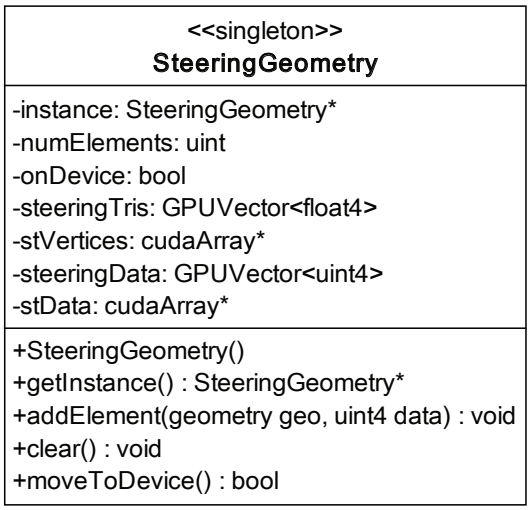


Figure 5.7: SteeringGeometry

5.2.6 Ray Renderer

The *RayRenderer* is a view node which provides the possibility to change settings of the GPU ray tracer. The methods of this class can call CUDA functions to start the rendering of the scene, to bind the datasets to the memory of the graphical hardware and to unbind them. These functions are listed in figure 5.8.

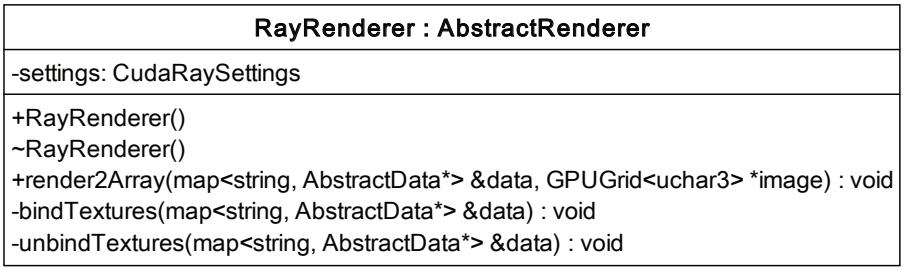


Figure 5.8: RayRenderer

5.2.7 Cuda Ray Node

Figure 5.9 shows the *Cuda Ray Node* class. It provides functions to bind and unbind the different sets of data to the textures on the graphical device.

Figure 5.9: CudaRayNode

5.2.8 GPU - CUDA Kernels

The *RayRenderer_kernel.cuh* file contains the main function of the ray tracer (see figure 5.10). The method, which is used to render the image (*stack less*, *stack based* or *using ropes*), can be set by arguments. It can be rendered interlaced, too.

RayRenderer_kernel.cuh
+rayTracer_main_kernel(RAY_TRACER_TYPE type, uchar3 *image_d, bool interlaced = false) : void +test_kernel(ray<float3> eyeRay, colorRGBA & color) : void

Figure 5.10: RayRenderer Kernel

RayRenderer_functions.cuh
+setupConstants(CudaRaySettings settings) : void +getChildBox(box<float3> Box, int child, box<float3> childBox) : uint +getParentBox(box<float3> Box, int child, box<float3> parentBox) : uint +euclDistance(float3 p, float3 q) : float +intersectRayTriangle(ray<float3> Ray, float3 t0, float3 t1, float3 t2, float &t, float &u, float &v) : bool +intersectRaySphere(ray<float3> Ray, Sphere sphere) : float +intersectRayPlane(ray<float3> Ray, float3 v0, float3 v1, float3 v2, float3 v3, float3 & intersectionPt) : bool +intersectSteeringTris(ray<float3> Ray, float & minT, uint steeringStartpos, uint steeringData, uint lastObject) : int +getSteeringPair(ray<float3> Ray, steeringArea & sa0, steeringArea & sa1) : int +getNextSteeringTri(ray<float3> Ray, steeringArea & sa) : int +drawLineOfTriangles(ray<float3> Ray, float width, float3 from, float3 to) : int +drawBoxOfTriangles(ray<float3> Ray, box<float3> Box, float width) : int +drawLineOfSpheres(ray<float3> Ray, float3 from, float3 to, float width, float4 color, uint num, float4 & color) : void +drawBoxOfSpheres(ray<float3> Ray, box<float3> Box, float width, float4 color, uint num, float4 & color) : void +drawAxes(ray<float3> Ray, float4 & color) : bool +drawSphere(ray<float3> Ray, Sphere sphere, float4 color, float4 & color) : void +calcRefractionRay(ray<float3> eyeRay, ray<float3> normal, ray<float3> refractionRay, float eta_i, float eta_r) : void +calcRefractionRayThinObjects(ray<float3> eyeRay, ray<float3> normal, ray<float3> refractionRay, float eta_i, float eta_r) : void

Figure 5.11: RayRenderer Functions

Figure 5.11 shows another CUDA header file which provides the ray tracer with some important

functions. *setupConstants* can be called to set variables which can be accessed from each thread. They contain global settings like the viewing matrix and the position of the eye. Octree specific functions are *getChildBox* and *getParentBox*. They can be used to calculate a child or parent bounding box of a given box. Essential intersection functions and draw methods, which are used for debugging, are included as well as the calculation of certain effects (refraction and reflection). And finally the most important methods are contained, i.e., the steering functions to get the next steering triangle or the next pair of steering triangles.

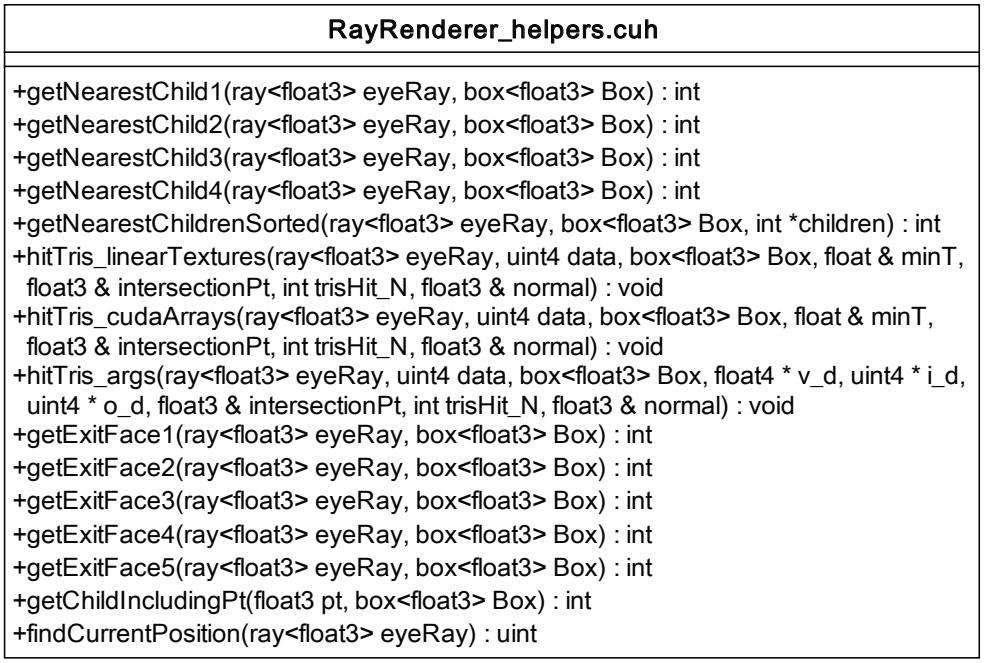


Figure 5.12: RayRenderer Helpers

Like the *RayRenderer_function.cuh* file the *RayRenderer_helpers.cuh* (see figure 5.12) contains important functions, too. There are some different approaches to return the child node which is the closest one to the eye (*getNearestChildN*). To intersect the triangles, which are assigned to a node, three methods were written (*hitTrisX*). Each of the three functions can handle one special kind of memory. The different *getExitFaceN* methods are needed for the traversal using ropes. Five approaches were tested to find the fastest way to determine the exit face of a bounding box.

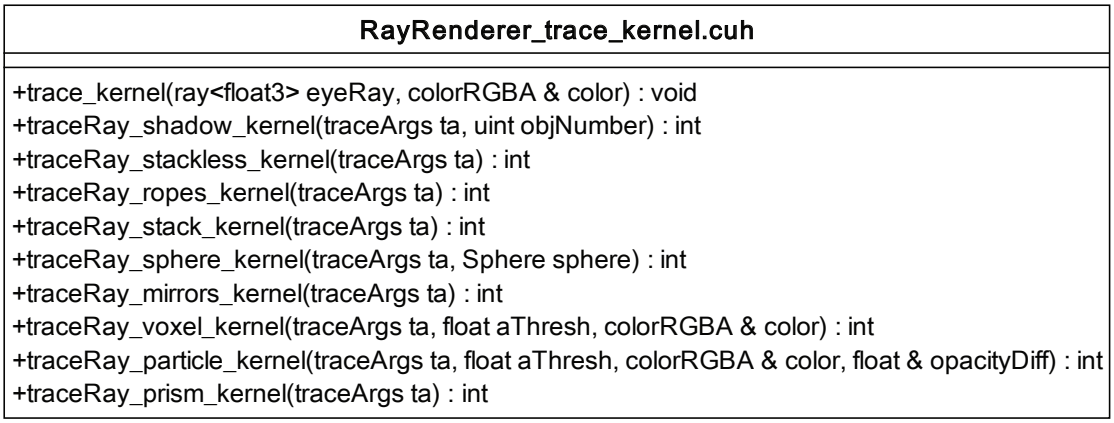


Figure 5.13: RayRenderer Trace Kernel

Figure 5.13 shows the content of the *RayRenderer_trace_kernel.cuh* file. It provides the different tracing methods, including the three presented approaches (see 4.3.2) to handle polygons, voxels and particles. The other functions are used to visualize various effects.

6. Evaluation

This section presents the results of all testing work. First of all the issue of how to measure the time in relation to the scene will be addressed, including the position of objects and camera to each other. Afterwards the results of speed measurements of some basic algorithms like intersection tests and memory access to the graphical hardware are included. The measurements of the different tracing methods, the implemented steering technology, and the results of complex scenarios are added.

6.1 An Expressive Way of Testing Speed

An easy way to quantify a scene is to measure the elapsed time from the start of the rendering kernel to its end. Without further enhancement this works only for one specified object under the same viewing settings. By using this simple method it is not possible to compare different scenes with different settings, especially including other objects and different positions and directions of the camera.

The model can be far away from the eye point and therefore most of the ray kernels would break up after the first bounding box intersection. Thus using only the resolution, which means the multiplication of the screen width with the screen height, is not expressive enough. Instead, the number of primary and secondary rays, which are shot in a scene, can be used. In addition the number of visited tree nodes can be interesting.

Another problem of comparing is different hardware - the used test configuration is not up-to-date. Thus speed tests of basic operations are included. Their measurements in addition to the number of their calls should allow to estimate the frames per second a scene would need to be rendered on another hardware configuration. For this only the time of the basic operations, the other machine needs, should be determined.

6.2 Basic Speed Tests

In this section time measurements of the most important operations are shown which have to be performed while tracing rays. They can be categorized in *intersection* and *memory access*. First measurements of simple, but essential, tasks are given to show how fast the GPU calls are. Each

kernel has to calculate the origin and the direction of its eye ray (*ray calculation*). A second task for each kernel is writing into the output array (*writing output*).

#kernel calls	dimension/ resolution	ray calculation (μs)	writing output (μs)
1	1x1	21.9	23.4
100	10x10	21.9	23.5
10000	100x100	21.9	70.3
16384	128x128	23.5	100.0
65536	256x256	28.1	325.1
76800	320x240	39.9	378.2
262144	512x512	46.9	1440.9
307200	640x480	51.6	1562.5
480000	800x600	68.7	2237.1
786432	1024x768	98.4	4273.5
921600	1280x720	109.3	4608.3
1000000	1000x1000	118.8	4651.2
1048576	1024x1024	125.0	6250.0
1920000	1600x1200	207.8	8928.6
2073600	1920x1080	221.9	9615.4
2560000	1600x1600	268.8	11904.8

Table 6.1: Basic measurements

Table 6.1 shows the results of the two tests, the *ray calculation*, and *writing output*. The tests are performed on 16 different screen resolutions which lead to 16 different numbers of rays to be calculated, i.e., its origins and directions. Table 6.1 shows the time which is consumed by these simple tasks. The more rays are shot the more time is consumed. Especially writing into the output array is an expensive process.

The first column of table 6.1 shows the number of rays, which are shot into the scene. This number is equal to the number of GPU kernel calls. In the third column the time is given in microseconds which is needed to handle the kernels and their simple ray calculation only. Writing the output back to the CPU is very costly. The last column shows the time in microseconds needed for writing and moving the output.

The ray calculation and writing output are essential tasks that have to be performed. Thus they can only be accelerated by using better hardware. The more kernel calls are performed, the more time is needed. This means the kernels are not executed simultaneously.

6.2.1 Ray Intersections

It is important to optimize the ray-intersection tests, because they have to be done often and usually are temporally very complex. There are two intersection types which are most often used in this work: ray-triangle and ray-AABB. A third one is needed for debugging only, i.e., the ray-sphere intersection. A problem of time-measurement systems can be that they are not precise enough to handle one single kernel call. For this reason the results of the tests, which are shown in 6.2, are averaged values. This means the kernel calls are performed 10.000 times.

#kernel calls	ray - triangle (sec)	ray - AABB (sec)	ray - spheres (sec)
1	0.0430	0.0220	0.0201
100	0.0750	0.0230	0.0229
10000	0.6510	0.4262	0.3061
16384	0.9925	0.6325	0.4455
65536	3.9569	2.4559	1.7059
76800	4.1329	2.8669	1.9609
262144	14.656	9.7181	6.6881
307200	17.637	11.386	7.8234
480000	25.541	17.775	12.212
786432	45.074	29.090	19.996
921600	48.672	34.093	23.406
1000000	52.412	36.975	25.397
1048576	55.953	38.781	26.625
1920000	99.621	70.979	48.730
2073600	107.18	76.653	52.607
2560000	132.79	94.622	64.950

Table 6.2: Ray Intersections - Triangle, AABB and Sphere. Each intersection is performed 100.000 times per kernel.

The results of the basic intersection tests are given in table 6.2. The intersections are performed 100.000 times in each kernel. The ray-sphere intersection is the fastest one, followed by the intersection ray-AABB, which is even faster than the ray-triangle intersection. Table 6.2 shows that intersection calculations are time-consuming tasks, thus the number of them has to be minimized.

6.2.2 Memory Access Measurements

Another basic operation is accessing the memory. Due to the fact that there exist different possibilities to store data on CUDA, they have to be examined. The first type of memory to be mentioned is called *Linear Texture Memory*. If it is used as a one-dimensional array, the maximum width is 134.217.728 units. Another kind of memory, *cudaArrays*, has a maximum width of 8.192 units. The third type is called *Constant Memory*, it can only store 64KB of data. All of these memories are read only. The unit numbers can be found in the NVIDIA Programming Guide [NVI09] on page 102f. The document deals only with GPUs with identical unit numbers for the mentioned memory types.

#kernel calls	LT (sec)	CA (sec)	C (sec)
1	0.0251	0.0378	0.0014
100	0.0411	0.0387	0.020
10000	0.2431	0.2281	0.0561
16384	0.3355	0.3365	0.0705
65536	1.2379	1.2379	0.2999
76800	1.4289	1.4299	0.3509
262144	4.8751	4.8751	1.5311
307200	5.6984	5.6824	1.3864
480000	8.8683	8.8683	2.1653
786432	14.527	14.543	4.2606
921600	17.016	17.016	4.1567
1000000	18.474	18.475	4.5062
1048576	19.360	19.360	6.1090
1920000	35.432	35.401	8.6522
2073600	38.247	38.246	9.3411
2560000	47.215	47.200	11.543

Table 6.3: Memory Access - Linear Texture Memory (LT), cudaArrays (CA) and Constant Memory (C). Each memory access is performed 10.000 times per kernel.

Table 6.3 shows the results of the texture-access measurements. To test the needed time 16 bytes were read in 10.000 times. The *Constant Memory* is the fastest to access, but its capacity is too low to store more than a few global settings. *CudaArrays* and *Linear Texture Memory* are very similar in terms of access time. Again, the capacity is crucial. *CudaArrays* can not store enough data to satisfy the requirements for this project. Thus all data like vertices, octree nodes and ropes are stored by using *Linear Texture Memory*.

6.3 Octree Settings

As explained in chapter 4 the octree, which is used to accelerate the traversal of polygonal objects, has a few settings. The two most important ones are the tree height and a given threshold of polygons. If the number of polygons a node contains falls below this threshold, the tree-building algorithm terminates at this branch. If the height of the octree is increased, the number of nodes, which have to be visited and intersected until a leaf is found, increases, too. Therefore the number of triangle intersections decreases. A good threshold should prevent the system of building up too large trees which are much more costly to traverse than more triangle intersections would be. Deeper trees require more memory space, especially for storing more tree nodes, a bigger number of polygons (if they overlap nodes), and eventually ropes.

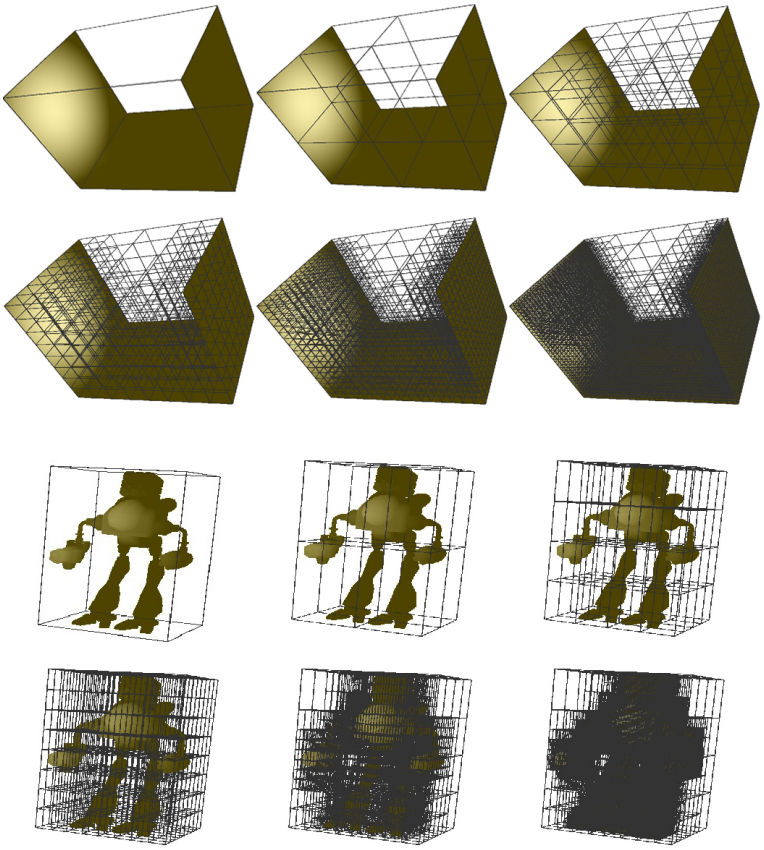


Figure 6.1: Octree Settings: Increasing number of nodes per level. Top: A simple shaped object called *3D'U'*. Bottom: A complex object called *Robot*.

6.3.1 Tree Height

The following tests are performed on two different objects: a very simple one and a more complex model of a robot. The first one is shaped like a three dimensional 'U'. It is called *3D'U'* and consists of six triangles and eight vertices. The second object consists of 2.573 vertices and 4.699 triangles and is called *Robot*. The method used to run the tests for both of the objects is a stackless ray-casting algorithm as described in chapter 4. Only the simple and fast surface shading (by Phong) is calculated in addition (see figure 6.1).

The *3D'U'* scene seems to be too simple, but it demonstrates an important issue in an expressive way. Increasing the height of the octree leads to additional traversal costs, because more nodes have to be visited which means more ray-AABB intersections have to be calculated. Furthermore additional data must be stored and moved to the graphical device (e.g., triangles, octree nodes, ropes, aso.).

Considering this issue there has to be an optimal tree height h for each octree which leads to the fastest rendering speed measured in frames per second (*FPS*). Further increasing of h would reduce the number of *FPS*, because the additional traversal would be temporally more complex. Thus it would annihilate the benefit of a higher tree, which is to reduce the number of ray-polygon intersections.

h	th	# nodes	# visited interiors	#leaves	# visited leaves	#tris	#ray-tri intersections	FPS
0	0	1	0	1	126224	6	757344	49.1
1	0	9	259127	8	230446	24	657396	42.2
2	0	73	817865	64	430713	96	575068	28.7
3	0	393	1423757	344	646016	384	517958	20.3
4	0	1801	2014775	1576	854739	1536	484936	15.9
5	0	7698	2586616	6728	1057067	6144	467472	12.9
6	0	31753	3145951	27784	1255129	24576	458516	10.5
0	0	1	0	1	62769	4699	294951531	0.10
1	0	9	149474	8	104869	20200	271977284	0.84
2	0	73	404701	64	185416	30309	79177153	2.21
3	0	577	904208	505	349959	44318	21723675	4.67
4	0	2689	1220001	2353	453148	65914	6113960	6.81
5	0	13353	1542849	11684	558928	115781	2044779	8.10
6	0	64025	1793543	56022	641045	253182	905493	7.53

Table 6.4: Octree Setting (Height): Top: Measurements of object *3D'U'*. Bottom: Measurements of object *Robot*.

The following list explains the captions of table 6.4 and table 6.5:

h:	the height of the octree
th:	the octree build up stops, if the number of triangles falls under this threshold
#nodes:	number of nodes in octree, including *interior nodes* and *leaves*
#visited interiors:	number of interior nodes which are intersected by the ray
#leaves:	number of leaf nodes in octree
#visited leaves:	number of leaf nodes which are intersected by the ray
# tris:	sum of triangles of each leaf
# ray-tri intersections:	sum of all ray-triangle intersection which have to be done
FPS:	frames per second

Table 6.4 shows the effect of changing the height of the tree on the two objects. The image resolution of both scenes is 1000x1000 which leads to 1.000.000 rays shot. The number of primary rays, i.e., rays which intersect at least one leaf node, is 126.224 in the simple object scene and 62.769 in the other. Figure 6.1 shows the increasing number of nodes on each additional level of the octree.

The measurements of *3D'U'* (table 6.4 line 1 to 7) show that the optimal octree height for traversing this simple object is 0, i.e., no octree should be used. With increasing the tree height the FPS decreases. The more complex *Robot* shows the use of an octree. The time to render it decreases with increasing the octree height, until its optimal height is reached at the value of 5.

6.3.2 Polygon Threshold

Using a threshold to let the octree-building algorithm terminate, if a node contains a number of elements below this value, should accelerate the performance of the ray tracer. In table 6.5 some test results are given. Different combinations of octree height and threshold were examined and the frames per second measured. All tests are performed on the *Robot* as described in section 6.3.1. The explanation of the column headings of table 6.5 can be found there, too. The *3D'U'* scene is too simple and thus not suitable for further measurements.

Contrary to the assumption the performance should be accelerated by using a threshold, no acceleration can be measured, at least not by using the *Robot*. But another advantage was found during the test runs - the threshold gives the possibility to build up higher trees. Thus more complex objects can be loaded and the problem of running out of local memory on the CPU can be solved. Many node overlapping triangles do not have to be assigned multiple times again and again on each new level. Which settings fit best, depends considerably on the scene and its included objects. Good settings are essential for a fast performance.

h	th	# nodes	# visited interiors	#leaves	# visited leaves	#tris	#ray-tri intersections	FPS
2	100	73	404701	64	185416	30309	79177153	2.21
4	5	2721	1272814	2381	470621	65993	6174802	6.81
4	10	2681	1291543	2346	476970	66149	6337065	6.67
4	100	1681	1030484	1471	389243	67158	12388130	5.08
5	1	13425	1610159	11747	581078	115836	2056737	8.10
5	2	13385	1627154	11712	586835	115890	2071371	7.90
5	5	12825	1636883	11222	590568	116169	2205503	7.72
5	10	11681	1602826	10221	579229	115558	2555430	7.44
5	100	2641	1052177	2311	395867	83122	11334210	5.04
6	2	62841	1950293	54986	693029	253143	964043	7.27
6	5	54249	1907246	47468	679007	246543	1194867	7.11
6	10	38825	1762422	33972	630858	223906	1767882	6.95
6	100	3209	1053668	2808	396319	91857	11290138	4.89
12	50	8409	1287814	7358	473262	128667	6155211	5.71

Table 6.5: Octree Setting (Threshold): Measurements of the object *Robot*.

6.4 Traversal Methods

The three different methods of traversing an octree discussed in chapter 4 are tested concerning performance in this section. To get significant numbers, three scenes, each including different objects, were used and all of them were viewed under two different camera settings. In addition each object is rendered by using different opacity thresholds. Thus in the first case only primary rays are sent. Figure 6.2 shows the three objects. The octrees of the objects are all built using the same settings - height 5, threshold 0. The *Robot* has 115.781, the *Ufos* scene 225.560 and the still life (*Bottle*) has 454.608 triangles. Table 6.6 shows the first set of measurements. Table 6.7 shows the second one in which the camera is closer to the scene and thus more primary rays are shot.

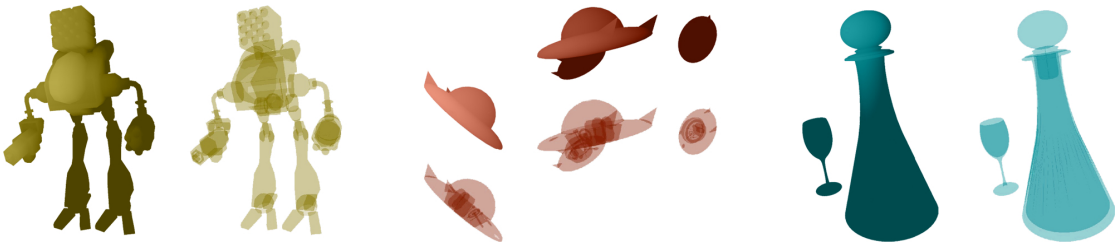


Figure 6.2: Traversal Methods. Left: *Robot*. Middle: *Ufos*. Right: *Bottle*.

The evaluation of the tables shows that the fastest method to trace rays is the *ropes* approach. Which of the other two methods is coming next depends on the scene. But another important issue is the quality of the rendered pictures. The stackless approach often has some pixel errors which result from numerical problems as described in section 4.3.2. The ropes and the stack-based approach produce error free images. In return they require much more memory.

#kernel calls	object	#primary rays	#secondary rays	method	FPS
65536	Robot	1069	0	stack based	45.66
65536	Robot	1069	0	stackless	45.66
65536	Robot	1069	0	ropes	64.10
65536	Ufos	627	0	stack based	45.66
65536	Ufos	627	0	stackless	53.19
65536	Ufos	627	0	ropes	58.48
65536	Bottle	1161	0	stack based	16.86
65536	Bottle	1161	0	stackless	16.42
65536	Bottle	1161	0	ropes	22.08
1000000	Robot	16267	0	stack based	7.710
1000000	Robot	16267	0	stackless	8.651
1000000	Robot	16267	0	ropes	12.08
1000000	Ufos	9649	0	stack based	8.000
1000000	Ufos	9649	0	stackless	9.406
1000000	Ufos	9649	0	ropes	11.22
1000000	Bottle	17759	0	stack based	3.786
1000000	Bottle	17759	0	stackless	3.831
1000000	Bottle	17759	0	ropes	5.565
65536	Robot	1069	1069	stack based	31.95
65536	Robot	1069	1069	stackless	33.67
65536	Robot	1069	1069	ropes	49.26
65536	Ufos	627	627	stack based	23.70
65536	Ufos	627	627	stackless	24.63
65536	Ufos	627	627	ropes	35.46
65536	Bottle	1161	1161	stack based	10.16
65536	Bottle	1161	1161	stackless	10.32
65536	Bottle	1161	1161	ropes	14.88
1000000	Robot	16267	16267	stack based	5.872
1000000	Robot	16267	16267	stackless	6.468
1000000	Robot	16267	16267	ropes	9.843
1000000	Ufos	9649	9649	stack based	5.615
1000000	Ufos	9649	9649	stackless	6.154
1000000	Ufos	9649	9649	ropes	8.204
1000000	Bottle	17759	17759	stack based	2.644
1000000	Bottle	17759	17759	stackless	2.452
1000000	Bottle	17759	17759	ropes	3.902

Table 6.6: Three methods of traversal - Camera Position 1

#kernel calls	object	#primary rays	#secondary rays	method	FPS
65536	Robot	10826	0	stack based	14.88
65536	Robot	10826	0	stackless	8.418
65536	Robot	10826	0	ropes	22.08
65536	Ufos	4652	0	stack based	12.80
65536	Ufos	4652	0	stackless	8.651
65536	Ufos	4652	0	ropes	21.28
65536	Bottle	11404	0	stack based	5.120
65536	Bottle	11404	0	stackless	3.722
65536	Bottle	11404	0	ropes	8.000
1000000	Robot	165223	0	stack based	1.894
1000000	Robot	165223	0	stackless	1.298
1000000	Robot	165223	0	ropes	3.049
1000000	Ufos	71020	0	stack based	2.099
1000000	Ufos	71020	0	stackless	1.151
1000000	Ufos	71020	0	ropes	1.828
1000000	Bottle	174120	0	stack based	0.989
1000000	Bottle	174120	0	stackless	0.780
1000000	Bottle	174120	0	ropes	1.350
65536	Robot	10826	36064	stack based	5.079
65536	Robot	10826	36064	stackless	3.555
65536	Robot	10826	36064	ropes	8.097
65536	Ufos	4652	15984	stack based	2.540
65536	Ufos	4652	15984	stackless	1.702
65536	Ufos	4652	15984	ropes	3.460
65536	Bottle	11404	40687	stack based	1.855
65536	Bottle	11404	40687	stackless	1.385
65536	Bottle	11404	40687	ropes	3.190
1000000	Robot	165223	551706	stack based	0.831
1000000	Robot	165223	551706	stackless	0.578
1000000	Robot	165223	551706	ropes	1.325
1000000	Ufos	71020	243247	stack based	0.621
1000000	Ufos	71020	243247	stackless	0.292
1000000	Ufos	71020	243247	ropes	0.821
1000000	Bottle	174120	620694	stack based	0.361
1000000	Bottle	174120	620694	stackless	0.281
1000000	Bottle	174120	620694	ropes	2.807

Table 6.7: Three methods of traversal - Camera Position 2

#kernel calls	dimension/ resolution	#primary rays	#secondary rays	#shadow rays	FPS
10000	100x100	4047	92598	0	12.82
65536	256x256	26511	606818	0	5.348
1000000	1000x1000	404532	9259657	0	0.703
2560000	1600x1600	1035588	23704594	0	0.322
10000	100x100	7503	98511	7503	21.28
65536	256x256	49396	656214	49396	4.255
1000000	1000x1000	755117	10014774	755117	0.350
2560000	1600x1600	1933566	25638160	1933566	0.141
10000	100x100	9496	99100	0	6.410
65536	256x256	62085	644628	0	2.062
1000000	1000x1000	948093	9806363	0	0.464
2560000	1600x1600	2427370	25094795	0	0.235

Table 6.8: Steering Scenarios - *Head in Glass*, *Robots in Desert* and *Bottle and Glass*

6.5 Steering Scenarios

Table 6.8 shows the results of the *steering-geometry approach*. Three different and complex scenarios are used to measure its performance. Pictures of them are presented in chapter 7. The *Head in Glass* scene contains a set of voxel data inside of a polygonal object which consists of 236.745 triangles. The voxel volume overlaps the bowl of the glass and so the steering technology is needed to cut off the overlapping parts. The first four lines of table 6.8 contain the results of rendering this scene. About 40 percent of all rays shot hit an object, most of them the glass. Due to its translucency these rays do not stop, but are refracted and are sent further into the scene. This leads to the shown number of secondary rays.

The next scenario could be taken from a video game. Some robots are standing in a desert under a cloudy sky. This scene contains particles, which are used to visualize the clouds, the terrain and robots consisting of 168.234 polygons. A light source is added which let the objects cast shadows. The shadow rays are already included in the secondary rays of table 6.8.

The final scene is called *Bottle and Glass* and pictures of them can also be found in chapter 7. The polygonal objects (a bottle and a glass) are standing on a mirror and in front of another one. The number of polygons of this objects is 454.608, the highest of all three scenes. The position of the camera is chosen to let about 95 percent of all rays hit the object. The two mirrors and the translucent objects are leading to a high number of secondary rays. Table 6.8 shows that the implemented ray tracer is not able to render complex scenes by using the *steering-geometry approach* in real-time.

7. Results

This section presents the results of the implementation of this work. After demonstrating the features of the ray tracer, some scenarios with more complex settings will be given. All the pictures are rendered by using the implemented framework. Due to the server-sided image compression, the server-client design leads to minor pixel errors, so called *artefacts*. As an introduction an object and its octree is shown (see figure 7.1). The implemented ray tracer can render the boundaries of the octree cells.

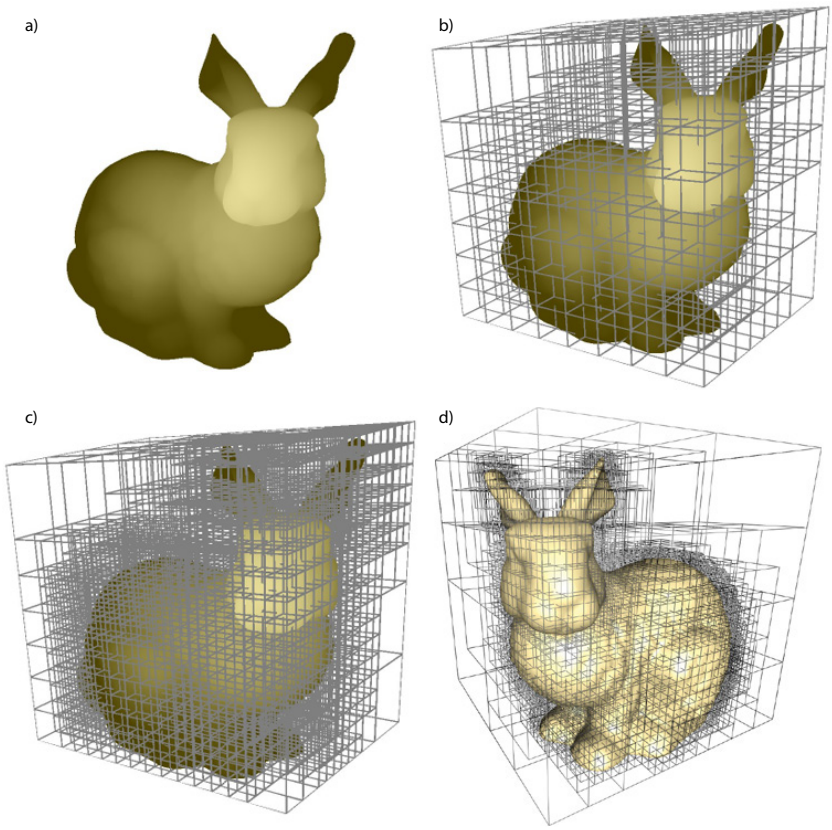


Figure 7.1: a) An object rendered by using the implemented ray tracer b) The octree of the object is rendered using a tree height of 3 c) The octree height is increased to 4 d) This octree picture is taken from GPU Gems 2 ([PF05])

7.1 Features

On top of figure 7.2 a simple shaded model of a robot on a diffuse ground plane is shown. The used surface shading (by Phong) is described in Hearn and Baker [HB04]. Underneath the steering polygons are visualized additionally.

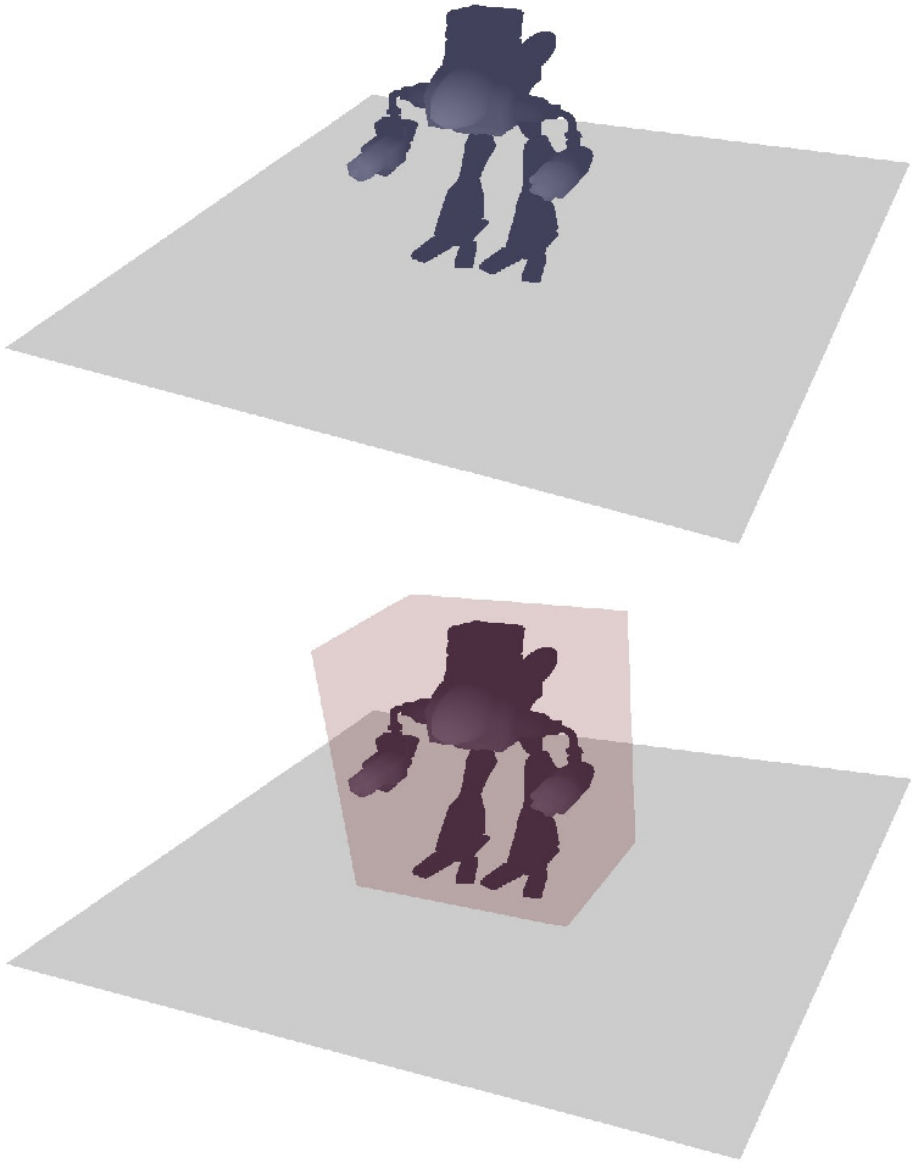


Figure 7.2: Features: Top: A simple shaded polygonal object. Bottom: The steering polygons are visualized.

On top of figure 7.3 the shading of the object is done by tracing an additional ray from the point of intersection to the light source. This ray is called *shadow ray*. If the shadow ray intersects an object on its way to the light source, the intersection point between object and eye ray lies in the shadow. Shadow rays can be switched on and off for every object separately. At the bottom of figure 7.3 the self-shadowing of the polygonal object is shown. Whereas the position of the light is the same as in the figure 7.3 on the top, the simple shading is deactivated.

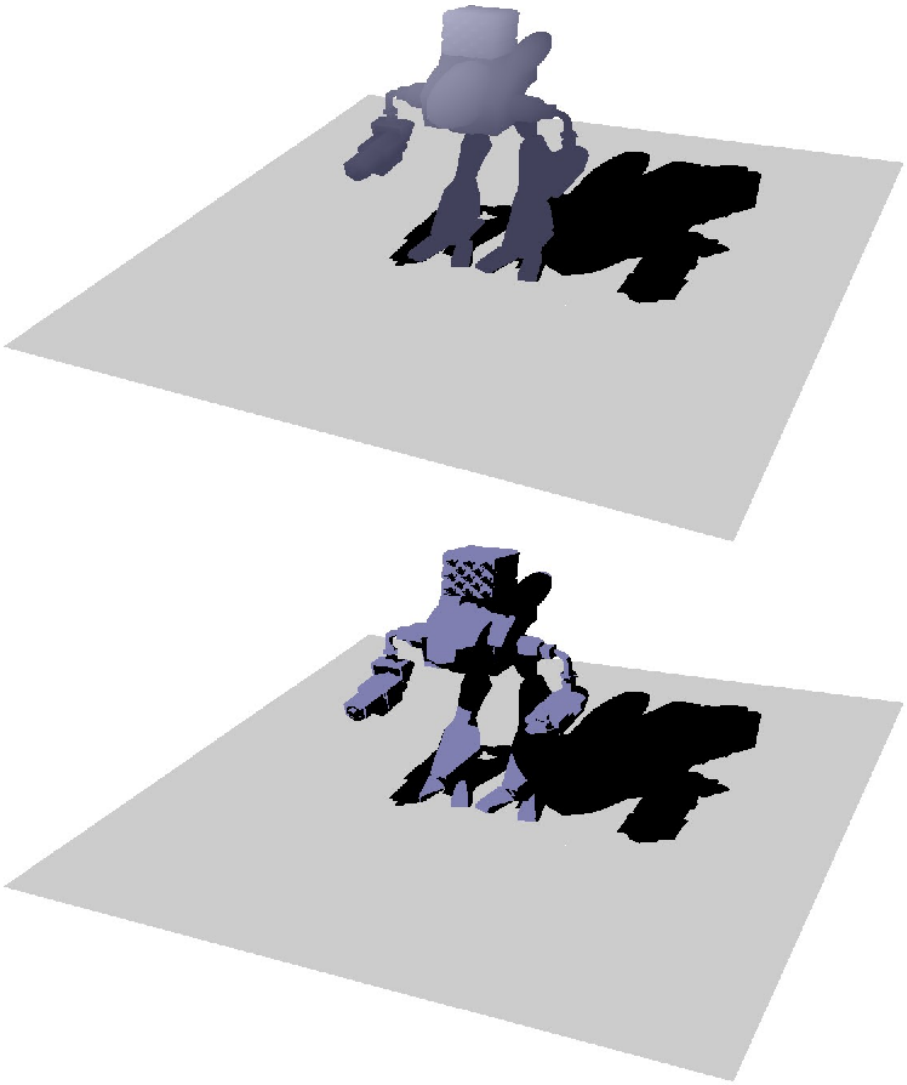


Figure 7.3: Features: Top: Shadow rays and simple shading. Bottom: Shadow rays and self-shadowing.

The next feature is reflection. The object stands on a mirror plane (see figure 7.4 top)). All rays, which hit this plane, are reflected as described in section 1.1. At the bottom of figure 7.4 a second vertical mirror is included.

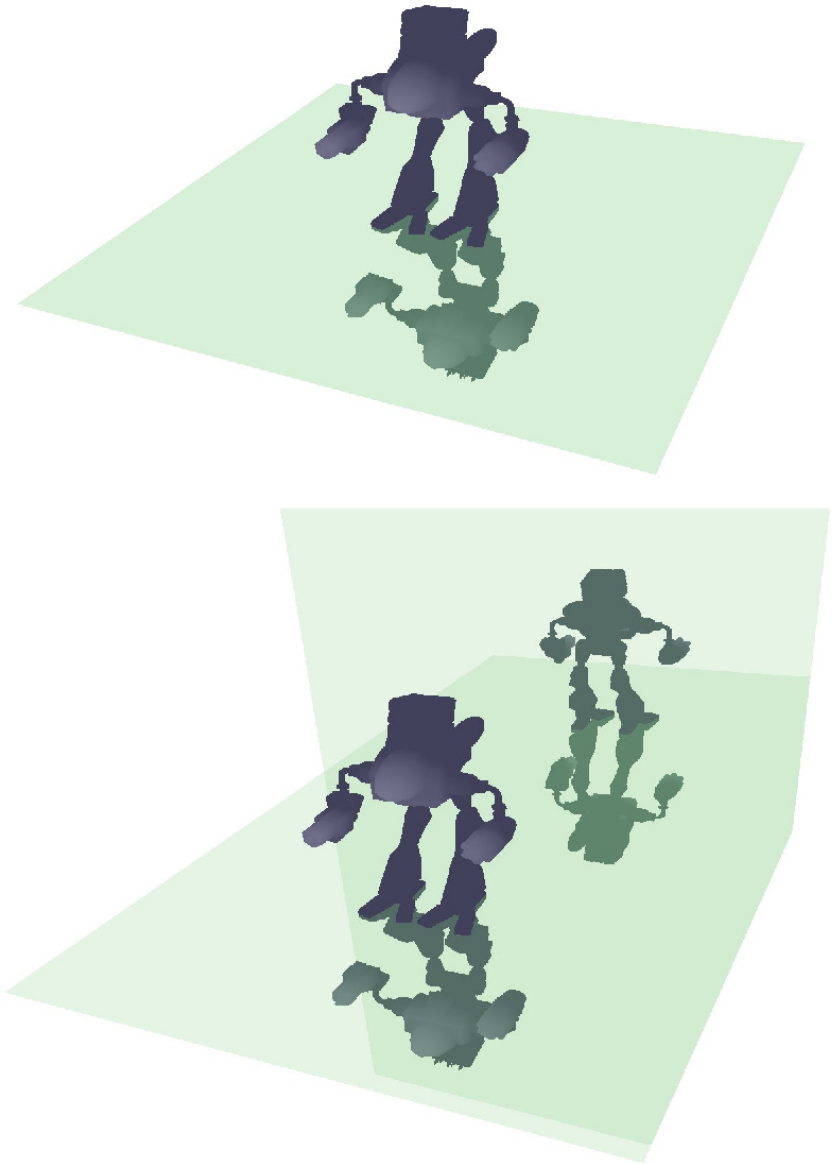


Figure 7.4: Features: Top: A polygonal object on reflecting plane. Bottom: A polygonal object and two mirror planes.

On top of figure 7.5 a turquoise prism in front of the object is shown. The rays, which intersect this prism, are refracted, as also described in section 1.1. If the polygons of the object are not fully opaque, the rays will not stop at the object surface, but enter it. The figure at the bottom shows the result if the entering rays are neither reflected nor refracted, but pierce the object in a straight line.

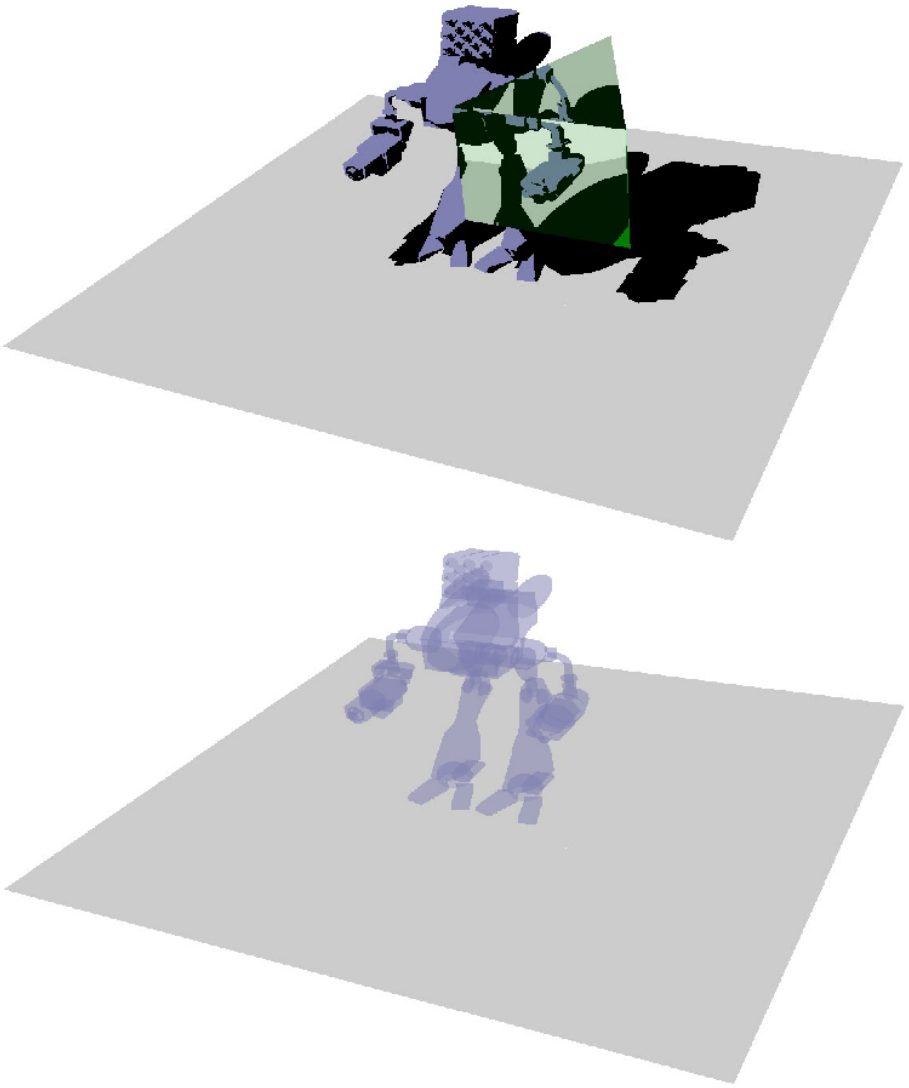


Figure 7.5: Features: Top: Refraction at a prism. Bottom: Opacity.

7.2 Steering Scenarios

The following pictures demonstrate the possibilities of the implemented ray tracer using the *steering-geometry approach*. The scenarios shown are already mentioned in chapter 3.

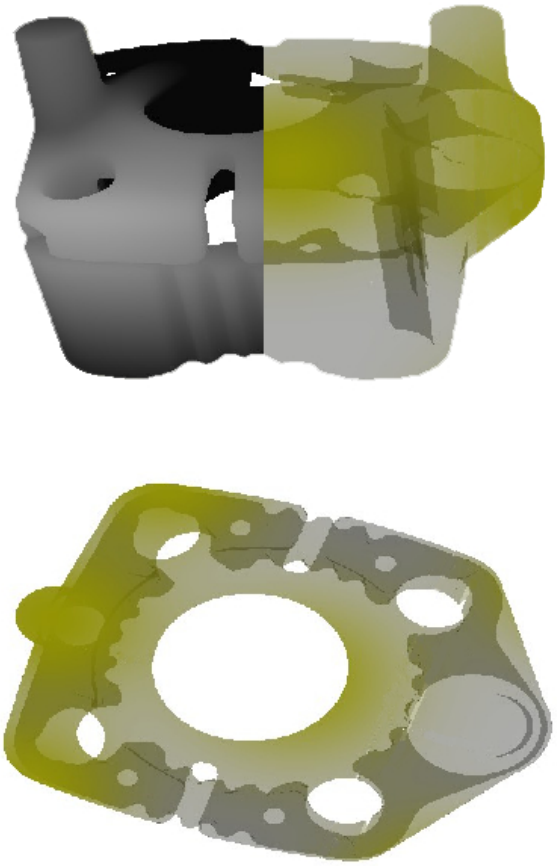


Figure 7.6: Industry: *Cooling Jacket*. A polygonal object filled with particles. Top: Split into two pieces: A simple shaded surface on the left side and a transparent surface on the right side. Bottom: The complete cooling jacket is shown from above.

In figure 7.6 a cooling jacket is shown. The model of the workpiece consists of triangles. A particle cloud is located inside of it to represent, e.g., the pressure or temperature of a cooling fluid. The upper figure is split into two parts. On the left side the simple shaded surface of the workpiece is shown. On the right side and in the figure at the bottom its surface is transparent and the particle cloud can be seen.

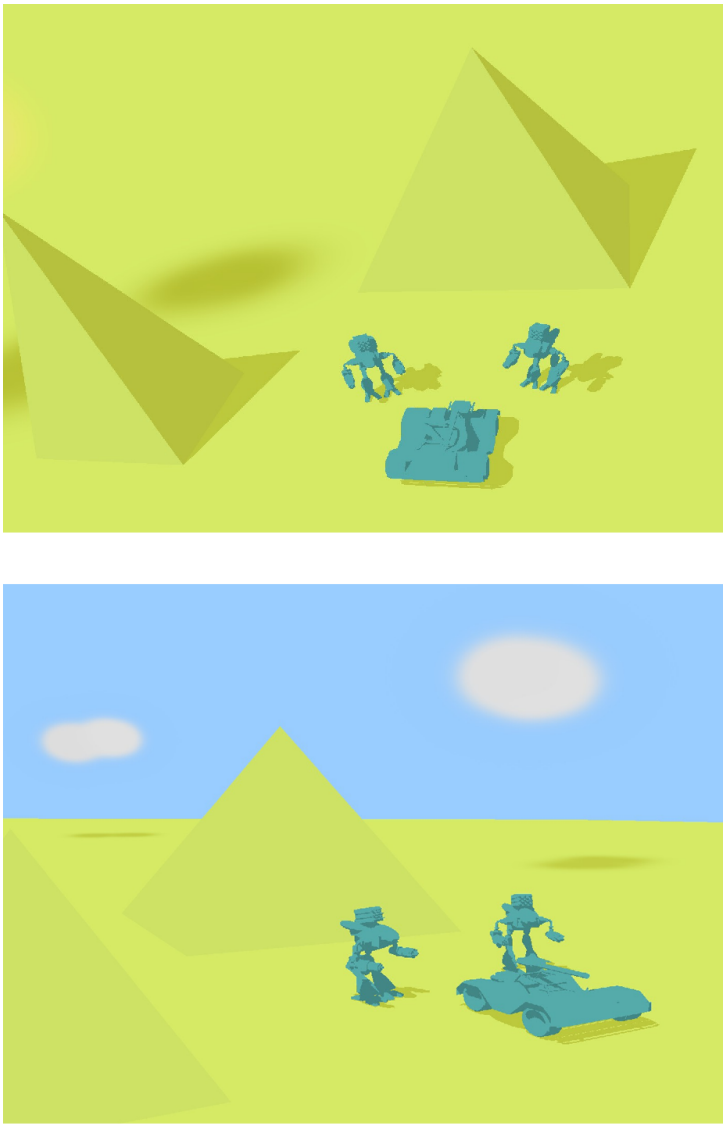


Figure 7.7: Video Games: *Robots in Desert*. Two robots and a vehicle are standing in a desert. Clouds are in the sky.

Figure 7.7 shows a scenario which could be captured from a video game. Two robots and a vehicle are standing in a desert surrounded by pyramids. Two white clouds are in the sky. Whereas the robots, the vehicle, the pyramids and the desert consist of polygons, the clouds are made of particles. All objects cast shadows onto other objects and on themselves (self shadowing).

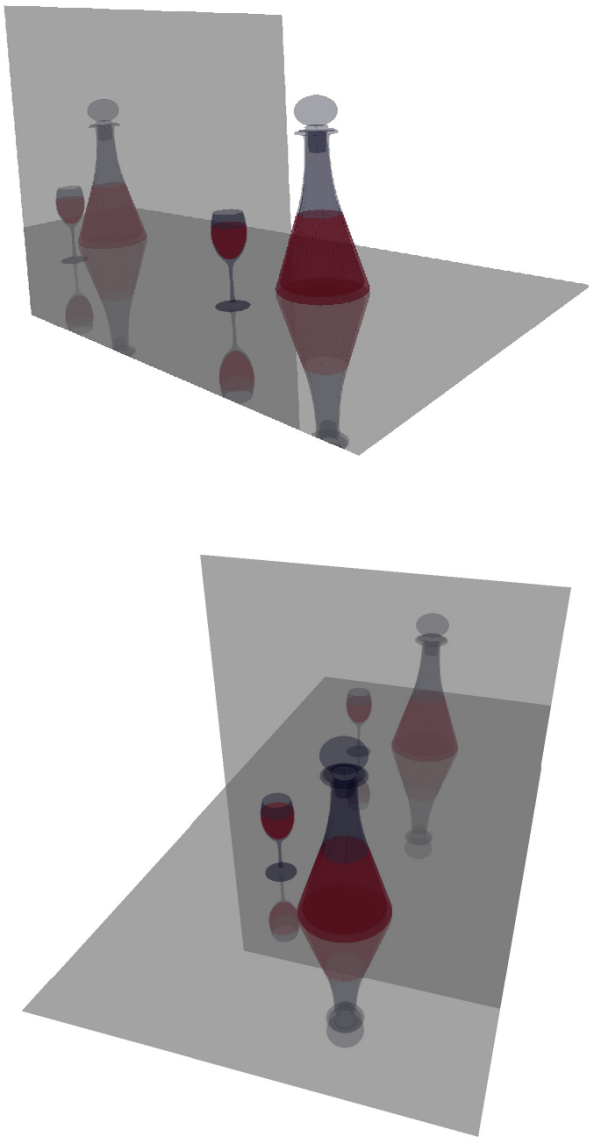


Figure 7.8: Still life: *Bottle and Glass*. A bottle and a glass, both are filled with a red liquid.

In figure 7.8 a still life is shown. A bottle and a glass are standing above and in front of two mirrors. They are filled with a red liquid (particles). All objects consist of polygons. If a ray hits a mirror, it is reflected. At the glass surfaces they are refracted.

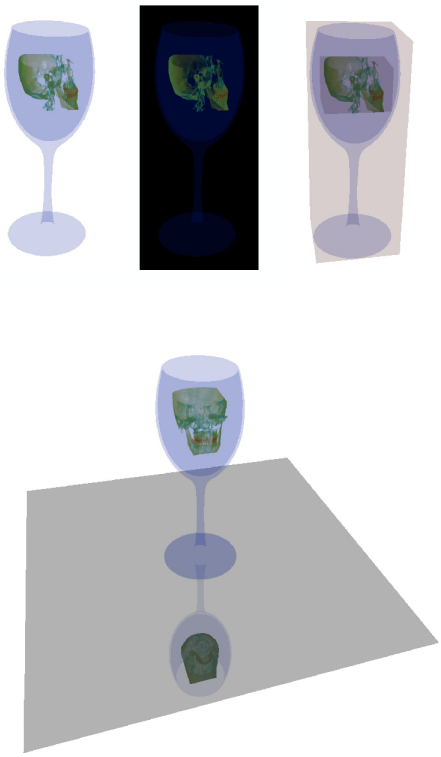


Figure 7.9: Still life: *Head in Glass*. Top (from left to right): white background, black background, with steering geometry. Bottom: Objects on a reflective plane.

Figure 7.9 shows a human head in a glass. The glass consists of polygons, the head is stored as a set of voxels. To each voxel a density value is assigned which can be manipulated by a transfer-function. On top of figure 7.9 three different pictures of the rendered scene are shown. The two on the left side and in the middle have different background colors. The third one visualizes the scene including the steering polygons which are colored reddish and which enclose the set of voxels and the glass. At the bottom of figure 7.9 and on top of figure 7.10 the glass and the head are shown on a reflective plane using different transfer-functions. At the bottom of figure 7.10 the steering polygons can be seen.

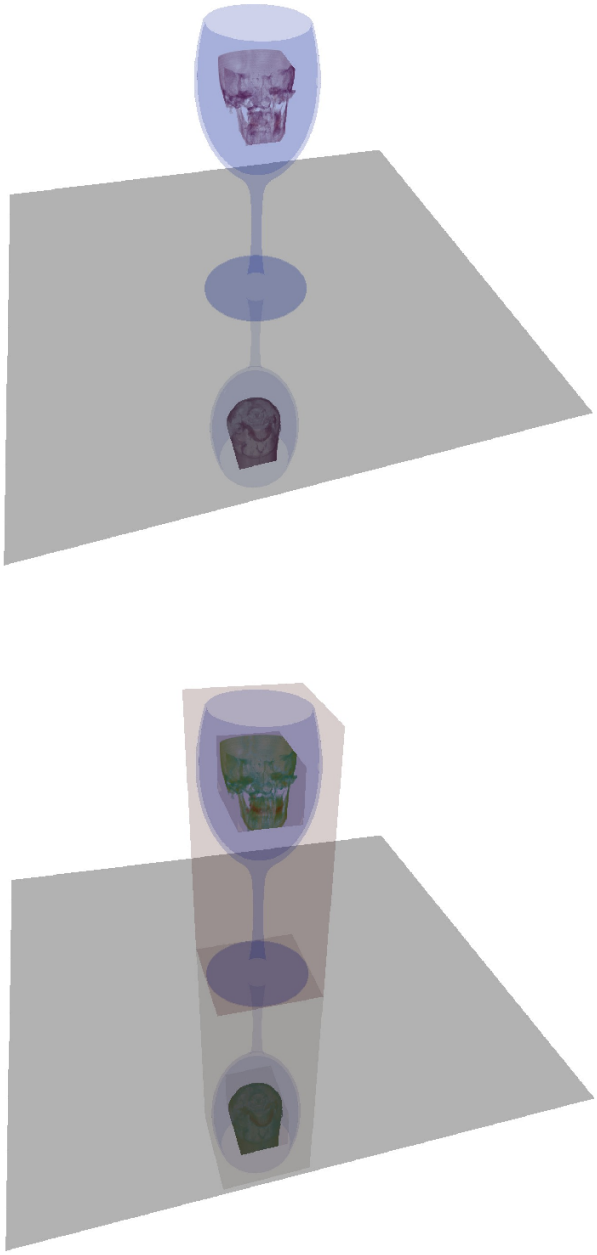


Figure 7.10: Still life: *Head in Glass*. Top: Objects on a reflective plane. Bottom: The steering polygons are visualized.

8. Conclusion

This work introduces three methods to handle different sets of data while tracing rays on the GPU. One of them was implemented and examined. It also gives an insight of planning and developing a GPU ray tracer and an expandable framework for it. The ray tracer was evaluated and the speed tested. This was done not only for the whole tracer, but also for the implemented intersection functions and texture accesses to find the most time consuming operations.

The used acceleration structure does not fit for every case. Splitting cells strictly into eight even-sized subcells can lead to an exploding number of extra polygons which have to be stored and moved to the graphical unit. A polygon, which overlaps a cell, has to be assigned to each cell it intersects. While ray tracing, the calculation of the closest subcell, which is intersected, is a time-consuming and complex task, too.

The steering method works as designed and planned, but the polygons have to be set manually yet. Finally, the goal to implement a GPU ray tracer, which can handle different sets of not necessarily spatially separated data, was reached, even if it does not run in real-time as planned - at least not under the hardware configuration it was tested with. There are plenty of extensions and speed-ups, which can improve the system and much research could be done to optimize it.

As future work there are several points to mention. The most important and interesting issue would be to implement the two other introduced methods of handling different types of data, separated objects and one global tree. Therefore a Kd-tree would be a natural choice. There are also interesting extensions for the octree. For example, arbitrary splitting planes could speed up the system.

Because of the simplicity of the test scenarios, which were shown in this work, only a few steering polygons are necessary. With increasing complexity their number will strongly increase, and an own acceleration structure would be essential.

By now only one graphical unit is in use while tracing rays, but the framework is already designed to handle multiple GPUs. Thus it would be interesting and worthwhile to parallelize more steps.

The steering geometry has to be set manually or at least semi-automatically. It would be helpful to write a graphical user interface, which gives the possibility to manipulate, i.e., insert, delete and edit the geometry.

The conclusion of this long time project is:

- Planning and developing an expandable framework is a time-consuming task. Most of the time was used on team meetings and in performing the software developing loop. Often changes in the core of the software lead to changes on many other components and incurred severe delays.
- The development of an expandable ray tracer is more difficult than it looks like.
- Algorithms should be kept as simple as possible if the method of debugging is based on colored output images and wrong GPU memory accesses can result in a freezing system.

Bibliography

[AK10] Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proceedings of High-Performance Graphics 2010*, 2010.

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics*, pages 145–149, 2009.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Bou05] Solomon Boulos. Notes on efficient ray tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 10, 2005.

[CW88] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4:65–83, 1988.

[EGMM07] Martin Eisemann, Thorsten Grosch, Stefan Müller, and Marcus Magnor. Fast ray/axis-aligned bounding box overlap tests using ray slopes. *Journal of Graphics Tools*, 12:35–46, 2007.

[ES94] Robert Endl and Manfred Sommer. Classification of ray-generators in uniform subdivisions and octrees for ray tracing. *Computer Graphic Forum*, 13:3–19, 1994.

[FGD+06] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. Exploring the use of ray tracing for future games. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 41–50, 2006.

[FH07] Raphael Fuchs and Helwig Hauser. Visualization of multi-variate scientific data. In *EuroGraphics 2007 State of the Art Reports (STARs)*, pages 117–134, 2007.

[FP02] Sarah F. Frisken and Ronald N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7:1–11, 2002.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[FTI86] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Arts: Accelerated ray tracing system. In *IEEE Computer Graphics and Applications*, pages 16–26, 1986.

[Gig90] Michael Gigante. Accelerated ray tracing using non-uniform grids. In *Proceedings of Ausgraph '90*, pages 157–163, 1990.

[GL10] Kirill Garanzha and Charles T. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, 29:289–298, 2010.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. In *IEEE Computer Graphics and Applications*, pages 15–22, 1984.

[Gla89] Andrew S. Glassner. *An introduction to ray tracing*. Academic Press Ltd., 1989.

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 113–118, 2007.

[GRHS08] Iliyan Georgiev, Dmitri Rubinstein, Hilko Hoffmann, and Philipp Slusallek. Real time ray tracing on many-core-hardware. http://graphics.cs.uni-saarland.de/fileadmin/cguds/papers/2008/georgiev_08_rtrt/RTRTIntuition.pdf (last checked: 2011/01/09), 2008.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. In *IEEE Computer Graphics and Applications*, pages 14–20, 1987.

[Hav99] Vlastimil Havran. A summary of octree ray traversal algorithms. *Ray Tracing News*, 12:11–23, 1999.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000.

[HB04] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Pearson Prentice Hall, 2004.

[HBv98] Vlastimil Havran, Jiri Bittner, and Jiri Žára. Ray tracing with rope trees. In *Proceedings of SCCG'98 (Spring Conference on Computer Graphics)*, pages 130–139, 1998.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, 2007.

[HT92] Ping-Kang Hsiung and Robert H. Thibadeau. Accelerating arts. *The Visual Computer*, 8:181–190, 1992.

[Kap85] Michael R. Kaplan. Space-tracing: A constant time ray-tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, pages 149–158, 1985.

[Kap87] Michael R. Kaplan. The use of spatial coherence in ray tracing. In *Techniques for Computer Graphics*, pages 173–193, 1987.

[Kay86] James T. Kayija. The rendering equation. In *ACM SIGGRAPH Computer Graphics*, pages 143–150, 1986.

[KB00] Jaroslav Krivanek and Vojtech Bubnik. Ray tracing with BSP and rope trees. http://www.cg.tuwien.ac.at/hostings/cescg/CESCG-2000/JKrivanek/ (last checked: 2011/01/09), 2000.

[Kno08] Aaron Knoll. A survey of octree volume rendering methods. http://www.cs.utah.edu/∼knolla/octsurvey.pdf (last checked: 2011/01/09), 2008.

[Lef93] Wilfrid Lefer. An efficient parallel ray tracing scheme for distributed memory parallel computers. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pages 77–80, 1993.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9:245–261, 1990.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

[NVI09] NVIDIA. NVIDIA CUDA Programming Guide Version 2.31. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf (last checked: 2011/01/09), 2009.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21:703–712, 2002.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems)*. Addison-Wesley Professional, 2005.

[PGSS06] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, 2006.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26:415–424, 2007.

[RUL00] Jorge Revelles, Carlos Ureña, and Miguel Lastra. An efficient parametric algorithm for octree traversal. In *Journal of WSCG*, pages 212–219, 2000.

[Sam89] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics*, 13:445–460, 1989.

[Sun91] Kevin Sung. A DDA octree traversal algorithm for ray tracing. In *Proceedings of Eurographics '91*, pages 73–85, 1991.

[TS05] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, 2005.

[WBMS05] Amy Williams, Steve Barrus, Russell Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9, 2005.

[Whi79] Turner Whitted. An improved illumination model for shaded display. In *Communications of the ACM*, pages 343–349, 1979.

[WSC+95] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-R: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1:343–349, 1995.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27:1–11, 2008.