Studienarbeit

Implementation eines Raytracers innerhalb der 3D-Plattform GroIMP

Michael Tauer

Brandenburgische Technische Universität Cottbus

Oktober 2006

Abstract

Ziel dieser Studienarbeit ist es, für die bestehende 3D-Software-Plattform GroIMP ein Plugin zu entwickeln, welches das Rendern von 3D-Szenen auf Grundlage des Raytracing-Verfahrens erlaubt.
In einem ersten Teil dieser Arbeit werden neben einer Einführung die Funktionen und Einstellungsmöglichkeiten des Raytracers für spätere Benutzer beschrieben.
Der zweite Teil beschreibt den Aufbau und die Funktionsweise der entwickelten Software. Um den Raytracer in seiner Funktionsweise leicht änderbar und erweiterbar zu gestalten, wurde die Software sehr modular entwickelt. Jedes Modul kapselt eine bestimmte Funktionalität und kommuniziert nur über Schnittstellen mit den anderen Modulen. So gibt es zum Beispiel Module für das Antialiasing, für die Schnittpunktberechnungen, für die Licht- und Schattenberechnungen und für den verwendeten Raytracing-Algorithmus.
Da ein Raytracer eine allgemeine Funktionalität zum Rendern von 3D-Szenen bereitstellt, konnte der Großteil der entwickelten Software als unabhängiges Framework konzipiert werden. Dadurch ist es leicht möglich, den Raytracer auch für andere Applikationen zu verwenden.

Selbstständigkeitserklärung

Die vorliegende Studienarbeit wurde von mir selbstständig angefertigt. Die verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis vollständig aufgeführt.

Michael Tauer, Ort Datum

Inhaltsverzeichnis

1. Einleitung

Neben der Benutzung des entwickelten Raytracer-Plugins werden in dieser Arbeit auch der Aufbau und die Funktion der zusammenwirkenden Software-Komponenten beschrieben.
Im ersten Kapitel soll durch eine Motivation ein Einstieg in die Arbeit gegeben werden. Das hier eingefügte Pflichtenheft dient als Funktionsbeschreibung der zu entwickelnden Software. Das zweite Kapitel beschreibt den 3D-Editor der Software GroIMP sowie die Benutzung des erarbeiteten Raytracers. Im dritten Kapitel wird kurz auf die benutzten Raytracing-Verfahren eingegangen. Dieses Kapitel soll außerdem eine Einführung in einige Raytracing-Techniken bieten. Das vierte Kapitel stellt die Struktur und die Funktionsweise der entwickelten Software-Module ausführlich dar, und das fünfte Kapitel beschäftigt sich mit der Einbindung des Raytracers in andere Software-Systeme.

1.1. Motivation

Die Software GroIMP bietet die Möglichkeit, 3D-Szenen zu erstellen und zu bearbeiten, u.a. mithilfe relationaler Wachstumsgrammatiken. Zu Beginn dieser Arbeit gab es für diese Software bereits die Option, 3D-Szenen mit dem externen Raytracer POV-Ray zu rendern. Ein extern benutzter Renderer hat aber den Nachteil, dass die aktuelle Szenenbeschreibung vor jedem Aufruf in ein anderes Format konvertiert werden muss, das auch vom Renderer verstanden wird. Für den externen POV-Ray-Renderer handelt es sich um ein Textformat, welches bei großen Szenen zeitaufwändig zu erzeugen ist. Außerdem gibt es Materialien, die von POV-Ray nicht unterstützt werden und somit nicht korrekt darstellbar sind. Ein weiterer Nachteil für die Verwendung eines externen Werkzeuges ist der Mehraufwand bei der Installation der GroIMP-Software. Ein externes Werkzeug muss zusätzlich zur eigentlichen Software installiert werden.
Um flexibler zu sein, sollte die bestehende Software GroIMP um einen eigenen internen Renderer erweitert werden. Für einen solchen Renderer wäre es möglich, Funktionalitäten anzupassen und zu erweitern.
Ein weiterer Grund für einen eigenen Renderer besteht darin, dass seine Algorithmen auch für ein Lichtmodell im Rahmen der Pflanzenmodellierung eingesetzt werden sollen, hier aber eine physikalisch korrekte Implementierung erforderlich ist. Dieses kann nur bei einer selbst entwickelten Software nachvollzogen und garantiert werden. So sollen auch neue bzw. nicht sehr verbreitete Algorithmen wie z.B. die Monte-Carlo-Integration angewendet werden.
Die für einen Raytracer notwendigen Shader, welche die Materialien und ihre Beleuchtungseigenschaften beschreiben, existieren in GroIMP bereits und können verwendet werden.

1.2. Pflichtenheft

Im Folgenden ist das Pflichtenheft eingefügt:

Thema: Implementation eines Raytracers innerhalb der 3D-Plattform GroIMP

Die am Lehrstuhl Grafische Systeme in Entwicklung befindliche, in Java implementierte 3D-Plattform GroIMP soll um einen integrierten Raytracer erweitert werden. Neben der eigentlichen Aufgabe der hochwertigen Bilderzeugung sollen die dazu nötigen Algorithmen auch für die Implementation eines Lichtmodells verwendet werden, das im Zusammenhang mit der Pflanzenmodellierung von großem Interesse ist.

Der Raytracer muss die folgenden Eigenschaften aufweisen:

- Die Eingabe ist eine komplette GroIMP-Szene (`de.grogra.imp3d.Scene3D`). Der Raytracer berücksichtigt dabei alle relevanten Eigenschaften einer GroIMP-Szene, insbesondere die Möglichkeit der parametrischen Objektinstanzierung.
- Es existieren effiziente Schnittpunktberechnungen für alle im Paket `de.grogra.imp3d.objects` enthaltenen Primitiv-Objekte. Diese bestimmen neben dem Schnittpunkt die weiteren relevanten Parameter (Flächennormale, UV-Koordinaten, Tangentenvektoren usw.). Je nach Aufgabe müssen dabei nicht alle Parameter berechnet werden, für ein Lichtmodell sind etwa die Tangentenvektoren irrelevant. Die Schnittpunktsberechnungen sollten idealerweise als Methoden in den Primitivklassen implementiert werden.
- Zur Effizienzsteigerung des Schnittpunktalgorithmus für Strahlen wird mindestens ein Verfahren aus der Literatur eingesetzt (etwa Hüllkörper, Voxel-Einteilung des Raumes).
- Die im Paket `de.grogra.imp3d.objects` definierten Lichtquellen werden korrekt verarbeitet.
- Die Farbinformationen am Schnittpunkt werden aus dem dem Objekt zugeordneten Shader (`de.grogra.imp3d.shading.Shader`) ausgelesen.
- Als Anti-Aliasing-Techniken werden adaptives Supersampling und stochastisches Raytracing verwendet.
- Die Rekursionstiefe des Raytracings kann gesteuert werden. Der Algorithmus des Path-Tracings soll hierbei eine Option sein.
- Der Quelltext und insbesondere die zentralen Klassen und Methoden werden kommentiert. Die gewählten Bezeichner entsprechen den Java-Konventionen.

Optional können folgende Eigenschaften implementiert werden:

- Schnittpunktberechnung für NURBS-Flächen
- GroIMP besitzt einen vernetzten Betrieb mit mehreren Clients. Raytracing und Lichtmodell könnten parallel auf diesen Clients verwendet werden.
- Radiosity

Das Lichtmodell ist ebenfalls optional. Es sollte, ähnlich wie im Anhang A der Dokumentation „User's Manual for Environmental programs“ zum Softwarepaket cpfg/L-Studio beschrieben, implementiert werden (URL http://www.algorithmicbotany.org/lstudio/enviro.pdf).

2. Benutzerdokumentation

2.1. GroIMP

GroIMP ist eine interaktive 3D-Modellierplattform, in die die regelbasierte Sprache XL integriert ist. Da die Sprache XL keine große Bedeutung für den entwickelten Raytracer hat, soll in diesem Kapitel nur auf Funktion und Benutzung des 3D-Editors näher eingegangen werden. Alle folgenden Beschreibungen sind für die Version 0.9.3.1 geschrieben und können für andere Versionen leicht abweichen.
Um eine Szene mit dem 3D-Editor zu bearbeiten, kann zum Beispiel ein neues Projekt erzeugt werden. Hierfür muss nach dem Starten von GroIMP der Menüpunkt *File→New→Project* angeklickt werden. In Abbildung 1 wird der grundsätzliche Aufbau der Bedienungselemente des Editors aufgezeigt.

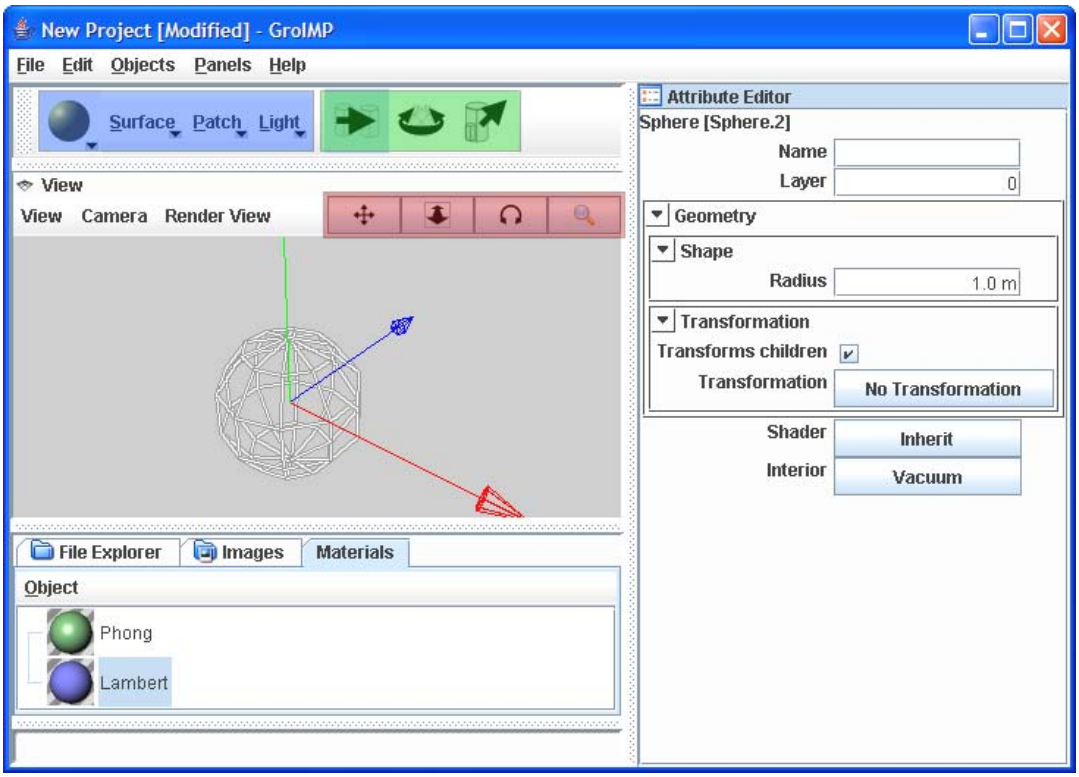


Abbildung 1 - Der 3D-Editor von GroIMP

Unter dem Menü befindet sich eine Iconleiste, die Funktionen zum Erzeugen und Manipulieren von 3D-Objekten zur Verfügung stellt. Durch in der Abbildung blau hervorgehobene Icons lassen sich 3D-Elemente erstellen. Das linke Icon bietet die Möglichkeit, verschiedene Grundobjekte (wie z.B. Kugel, Quader, Kegel, Zylinder, u.a.) zu erzeugen, während mit dem vierten, blau markierten Icon verschiedene Lichter und ein Sky-Objekt erstellt werden. Die anderen beiden Icons erstellen komplexere Objekte, die vom Raytracer noch nicht unterstützt werden. Konkret werden die Primitiv-Objekte Kugel, Quader, Zylinder, Kegel, Kegelstumpf, Parallelogramm, Ebene, die Lichtobjekte Punktlicht, Spotlicht, gerichtetes Licht, ausgedehnte Lichtquellen (diese können für Parallelogramme definiert werden) und das Sky-Objekt vom Raytracer unterstützt. Mit Hilfe der grün hervorgehobenen Icons können ausgewählte 3D-Objekte verschoben, gedreht oder skaliert werden. Die rot markierten Icons im View Fenster bieten die Möglichkeit, die Ansicht zu verändern und in der Szene zu navigieren.
Die Eigenschaften der erstellten Objekte können für ausgewählte Objekte auf der rechten Seite verändert werden. Durch den Button rechts neben der Eigenschaft

Shader kann das Material des jeweiligen Objektes angepasst werden. Grundsätzlich gibt es zwei mögliche Material-Typen. Zum einen ein diffuses Material, welches durch Auswählen von *Materials→Lambert* zugeordnet werden kann, und zum anderen ein spiegelndes Material, welches durch *Materials→Phong* ausgewählt wird.
Um Punktlichter, Spotlichter oder gerichtete Lichtquellen zu erzeugen, wird zuerst ein allgemeines Lichtobjekt erstellt, bei dem durch die Eigenschaft *Light* bestimmt werden kann, um welches konkrete Licht es sich handelt. Obwohl es sich bei flächigen Lichtern auch um Lichter handelt, werden sie nicht durch das *Light*-Icon erstellt. Flächige Lichter sind eine Spezialisierung von Parallelogrammen, und deswegen muss zum Erstellen zuerst das Grundobjekt Parallelogramm erstellt werden. Wird dann die Eigenschaft *Area Light* auf *area* gesetzt, ist das Parallelogramm als flächige Lichtquelle anzusehen, das die Grundfläche des Parallelogramms besitzt.

2.2. Funktionen des Raytracers

Um eine modellierte 3D-Szene zu rendern, d.h. ein möglichst realistisches Bild der Szene zu berechnen, muss der Raytracer durch den Menüpunkt *Render View→Built-In Raytracer* im View Fenster gestartet werden. Die aktuelle Szene wird dann zeilenweise berechnet und dargestellt. Zusätzlich werden im unteren Bereich des View Fensters Status und Fortschritt der Berechnung angezeigt.
Die Eigenschaften des Raytracers können durch ein allgemeines Eigenschaftenfenster eingestellt werden. Dieses ist durch den Menüpunkt *Panels→Preferences* erreichbar. Auf der linken Seite des Fensters können die verschiedenen Komponenten der Software GroIMP ausgewählt werden, die durch Einstellungen konfigurierbar sind. Um den Raytracer konfigurieren zu können, muss in der Kategorie *Renderer* der Unterpunkt *Built-In Raytracer* ausgewählt werden. In Abbildung 2 ist das Aussehen des Fensters nach der Auswahl dargestellt.

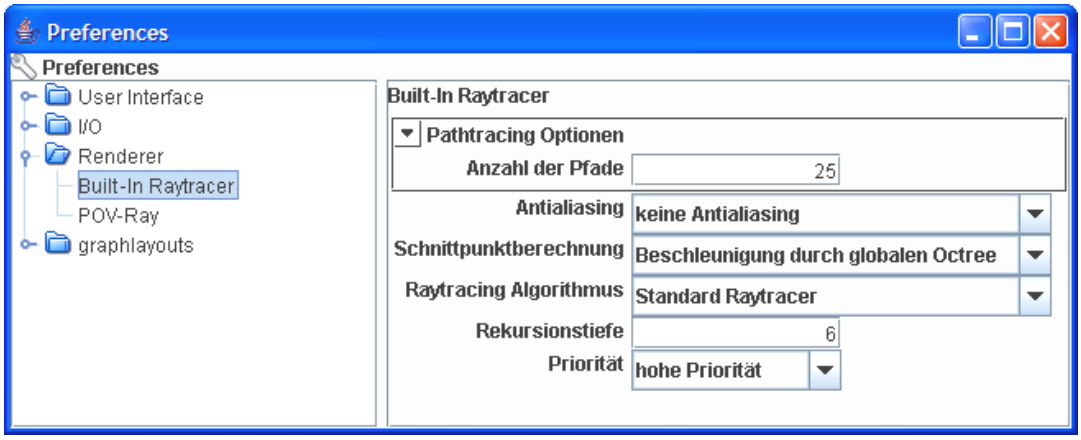


Abbildung 2 - Das Einstellungsfenster

Die erste allgemeine Eigenschaft ist das *Antialiasing*. Hier kann die verwendete Antialiasing-Methode bestimmt werden. Mögliche Werte sind *kein Antialiasing*, *stochastisches Raytracing* oder *adaptives Supersampling*. Das stochastische Raytracing erzeugt qualitativ sehr hochwertige Bilder, ist dafür aber recht langsam. Falls der *Raytracing-Algorithmus Pathtracing MT* ausgewählt ist, sollte diese Antialiasing-Methode verwendet werden, da sie sich sehr gut zur Rauschunterdrückung eignet. Einen guten Kompromiss zwischen Qualität und Rechenzeit stellt die Option *adaptives Supersampling* dar. Falls das Bild kein hochfrequentes Rauschen enthält, ist der Qualitätsunterschied zwischen stochastischem und adaptivem Supersampling kaum erkennbar.

Für die Eigenschaft *Schnittpunktberechnung* sollte die voreingestellte Option *Beschleunigung durch globalen Octree* beibehalten werden, da diese Option in den meisten Fällen die Rechenzeit verkürzt.
Die Eigenschaft *Raytracing Algorithmus* bietet die wahrscheinlich wichtigste Einstellungsmöglichkeit. Hierdurch kann der prinzipielle Berechnungsalgorithmus verändert werden, was sich stark auf das gerenderte Bild auswirkt. Die Option *Standard Raytracer* erzeugt recht unnatürlich wirkende Bilder, da dieser Algorithmus für diffuse Materialien indirekte Beleuchtungen ignoriert und für spiegelnde Materialien stets ideale Spiegelungen berechnet. Dafür ist die Berechnung mit dieser Option recht schnell. Etwas zeitaufwendiger, dafür aber qualitativ besser ist das Rendern mit der Option *Pathtracer MT*. Die hierbei erzeugten Bilder unterstützen diffuse Reflexionen sehr gut, können dafür aber verrauscht sein. In dem obersten Feld *Pathtracing Optionen* befindet sich die Eigenschaft *Anzahl der Pfade*. Diese Eigenschaft wird nur ausgewertet, wenn der *Pathtracer MT* als *Raytracing Algorithmus* ausgewählt wurde, und wirkt sich stark auf das Rauschen im gerenderten Bild aus. Je höher der Wert ist, desto weniger Rauschen enthält das Bild, aber desto länger wird die Berechnung dauern. Der Ausgangswert 25 wird als guter Kompromiss angesehen.
Durch die Eigenschaft Rekursionstiefe wird z.B. bestimmt, wie oft Licht in der Szene während der Berechnung hin und her gespiegelt wird. Durch den Wert 0 können alle Spiegelungen und Brechungen ausgestellt werden. Um realistische Bilder zu erhalten, sollte er auf 6 gesetzt sein.
Mit Hilfe der letzten Eigenschaft Priorität kann der Ressourcenverbrauch während der Berechnung bestimmt werden. Durch die Option *hohe Priorität* werden alle verfügbaren Ressourcen für die Berechnung verwendet. Die Option *mittlere Priorität* bietet die Möglichkeit, das Betriebssystem neben der Berechnung bedingt zu belasten. Durch die Option *niedrige Priorität* wird die Berechnung regelmäßig pausiert, wodurch ein Arbeiten im Betriebssystem parallel zur Berechnung möglich ist.

3. Raytracing

In der Computer-Grafik versteht man unter dem Begriff Raytracing das Erzeugen eines Bildes durch die Verfolgung von Strahlen in einer 3D-Szene. In diesem Kapitel soll kurz auf die verwendeten Raytracing-Verfahren eingegangen werden.
Neben dem Verfolgen von Lichtstrahlen von den Lichtquellen aus gibt es die Möglichkeit, so genannte inverse Lichtstrahlen zu verfolgen. Diese inversen Lichtstrahlen werden von der Kamera ausgesendet und reflektieren bzw. brechen an Szenenobjekten nach denselben Gesetzen, die auch für Lichtstrahlen gelten. Durch inverse Lichtstrahlen können die Farbintensitätswerte der Pixel in der Bildebene durch die Verfolgung von Lichtstrahlen in umgekehrter Richtung ihrer Fortpflanzung von der Kamera durch die Szene zur Lichtquelle bestimmt werden. Eine schematische Darstellung dieses Prinzips ist in Abbildung 3 gegeben. Alle in dieser Arbeit benutzten Verfahren verwenden ausschließlich inverse Lichtstrahlen.

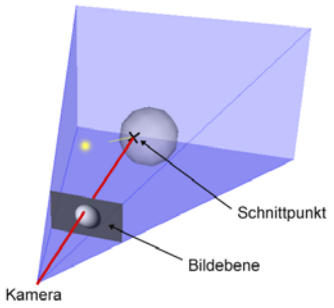


Abbildung 3 - Verfolgung inverser Lichtstrahlen

3.1. Beleuchtung

Trifft ein Lichtstrahl auf ein Objekt in der Szene, kann für den Schnittpunkt die direkte Beleuchtung aller Lichtquellen der Szene bestimmt werden. Um mögliche Schatten von anderen Objekten in dem Schnittpunkt zu ermitteln, wird ein Schattenstrahl vom Schnittpunkt zu jeder Lichtquelle auf Schnittpunkte mit den anderen Szenenobjekten getestet. Gibt es einen Schnittpunkt mit einem anderen Objekt, so schattet dieses Objekt den betrachteten Schnittpunkt ab und die Lichtquelle wird ignoriert.
Wenn in einem Strahl-Objekt-Schnittpunkt reflektierte und gebrochene Strahlen weiter verfolgt werden, können damit auch indirekte Beleuchtungsanteile bestimmt werden.
Der in dieser Arbeit entwickelte einfache Raytracer reflektiert und bricht Strahlen ideal. An diffusen Objekten wird kein reflektierter Strahl erzeugt. Somit werden mit diesem Verfahren nur ideale indirekte Beleuchtungsanteile berücksichtigt. Für diffuse Materialien wird der indirekte reflektierte Beleuchtungsanteil sogar ignoriert. Dies entspricht einer physikalisch nicht korrekten, erhöhten Absorption.

3.2. Pathtracing

Neben einem einfachen Raytracing-Verfahren wurde in dieser Arbeit auch ein Pathtracing-Verfahren implementiert. Beim Pathtracing werden nicht nur ideale Reflexionen oder Brechungen berücksichtigt, sondern auch alle anderen nicht-idealen Möglichkeiten mit einbezogen.
Um das Reflexions- und Brechungsverhalten von Materialien zu beschreiben, wird jedem Material eine BRDF (Bidirectional Reflectance Distribution Function) und eine BTDF (Bidirectional Transfer Distribution Function) zugeordnet. Hierbei handelt es sich um mehrdimensionale Verteilungsfunktionen, die die Reflektion bzw. die Brechung von rauen Materialien beschreiben. Sie liefern für einen gegebenen, auf dem Material auftreffenden, Lichtstrahl den Quotienten aus Bestrahlungsstärke und Strahlungsdichte für jeden

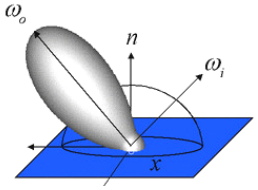


Abbildung 4 - Beispiel einer BRDF aus [9]

reflektierten bzw. gebrochenen Lichtstrahl. In Abbildung 4 ist eine BRDF für den eintretenden Lichtstrahl ω_i visualisiert.

Um eine Abtastung der BRDFs bzw. der BTDFs beim Pathtracing zu ermöglichen, können alle Shader zufällig reflektierte bzw. gebrochene Strahlen, entsprechend der Verteilungsfunktionen, generieren. Beim Pathtracing werden am ersten Schnittpunkt eines Objektes N generierte Strahlen verfolgt. An allen weiteren Schnittpunkten, wird nur noch ein möglicher Strahl verfolgt. Pro Pixel bzw. Subpixel werden somit N verschiedene Pfade verfolgt, die das Licht zur Kamera nehmen könnte. Die Farbwerte des Pixels bzw. Subpixels ergeben sich aus den Farbwerten der Pfade. Je größer N gewählt wird, desto genauer können die Funktionen abtastet werden und desto weniger Rauschen enthält das gerenderte Bild. In Abbildung 5 sind zwei verschiedene Pfade schematisch dargestellt, die für die Bestimmung des Farbwertes eines Pixels verwendet werden.

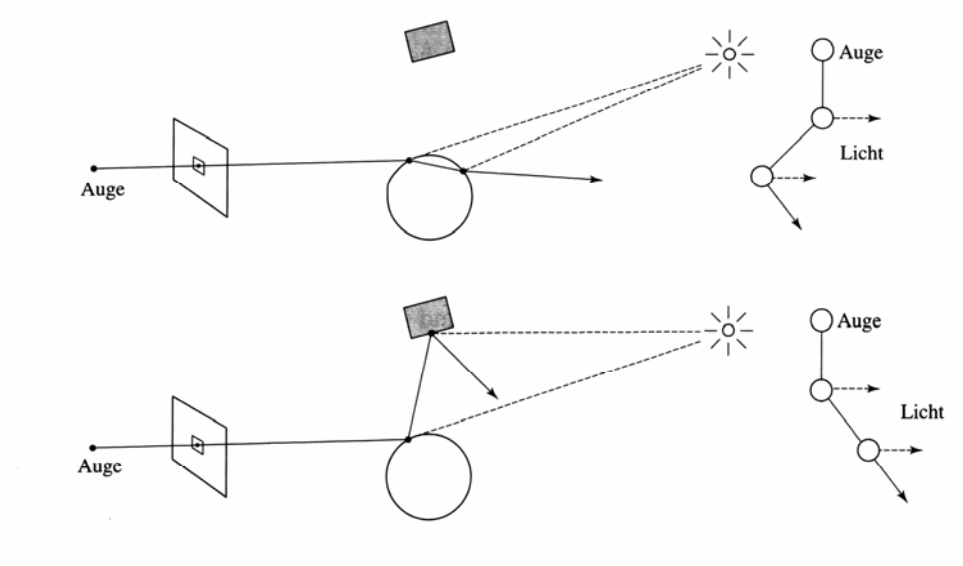


Abbildung 5 - Zwei Pfade beim Pathtracing aus [1]

4. Implementierung

In diesem Kapitel wird näher auf die Entwicklung und den Aufbau der Software eingegangen.

4.1. Architektur

Die grundlegendste Architekturentscheidung war die Implementierung des Raytracers als Framework und Einbindung als Plugin in GroIMP. Dadurch ist eine sehr klare Strukturierung und Trennung von der allgemeinen Funktionalität eines Raytracers und der speziellen Anpassung für GroIMP möglich. Außerdem ist es leicht möglich, den Raytracer auch in andere Projekte einzubinden und zu verwenden. Hierauf soll im fünften Kapitel näher eingegangen werden.

4.1.1. Die Architektur des Frameworks

Grundsätzlich gibt es mehrere funktionale Module, die möglichst unabhängig von einander austauschbar sein sollen. Diese Module sind die verwendete Antialiasing-Methode, die Schnittpunktberechnung, für die es diverse Optimierungsmöglichkeiten gibt, der verwendete Raytracing-Algorithmus sowie die Beleuchtungsstrategie. Jedes dieser Module ist in einem eigenen Package im Raytracer-Plugin gekapselt.

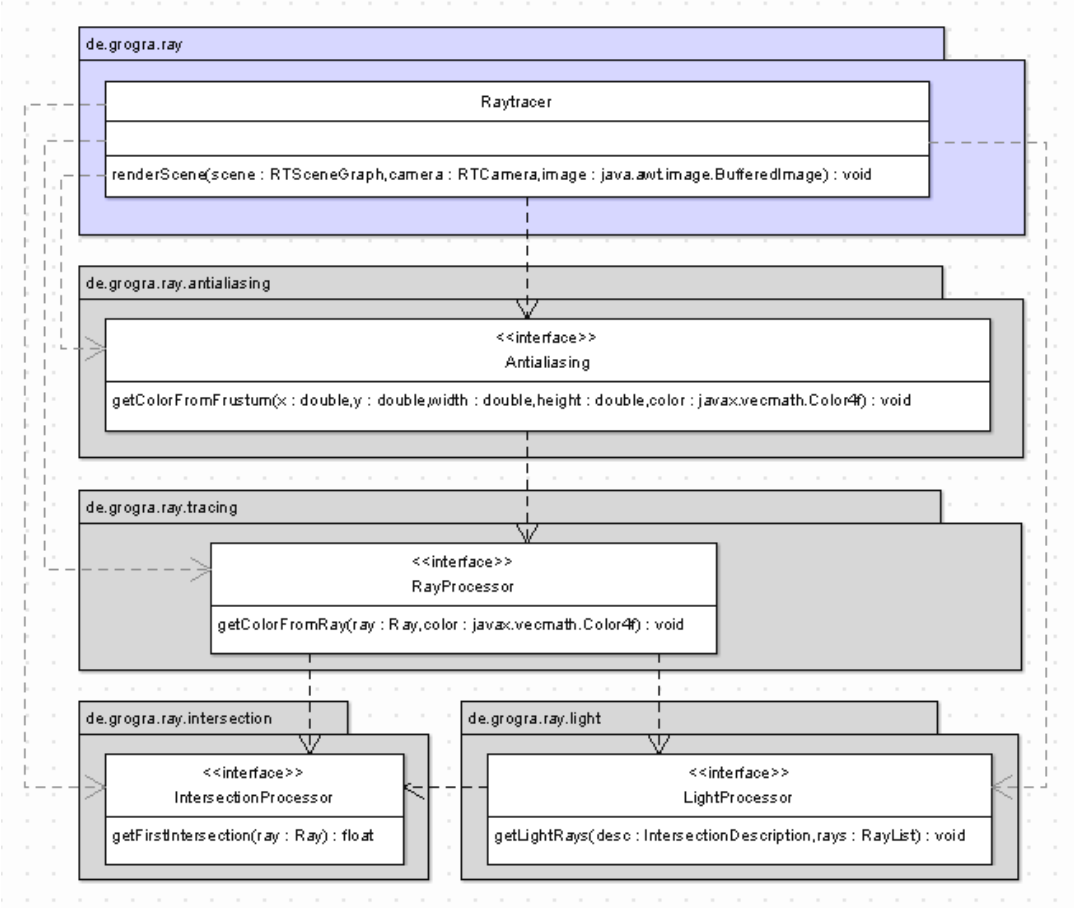


Abbildung 6 - Die vier wichtigsten Module

Zusätzlich zu den Modulen gibt es eine zentrale Komponente, die Klasse `Raytracer`, die den Rendering-Prozess anstößt. Und auch sie benutzt nur die Schnittstellen der Module, um auf sie zuzugreifen.

4.2. Antialiasing

Um eine für alle verwendeten Antialiasing-Methoden einheitliche Schnittstelle zu definieren, wird die zu rendernde Sichtpyramide in Pixel-Sichtpyramiden zerlegt. Für jeden Pixel des späteren gerenderten Bildes existiert eine solche Pyramide, für die der Antialiasing-Algorithmus genau einen Farbwert berechnen muss.

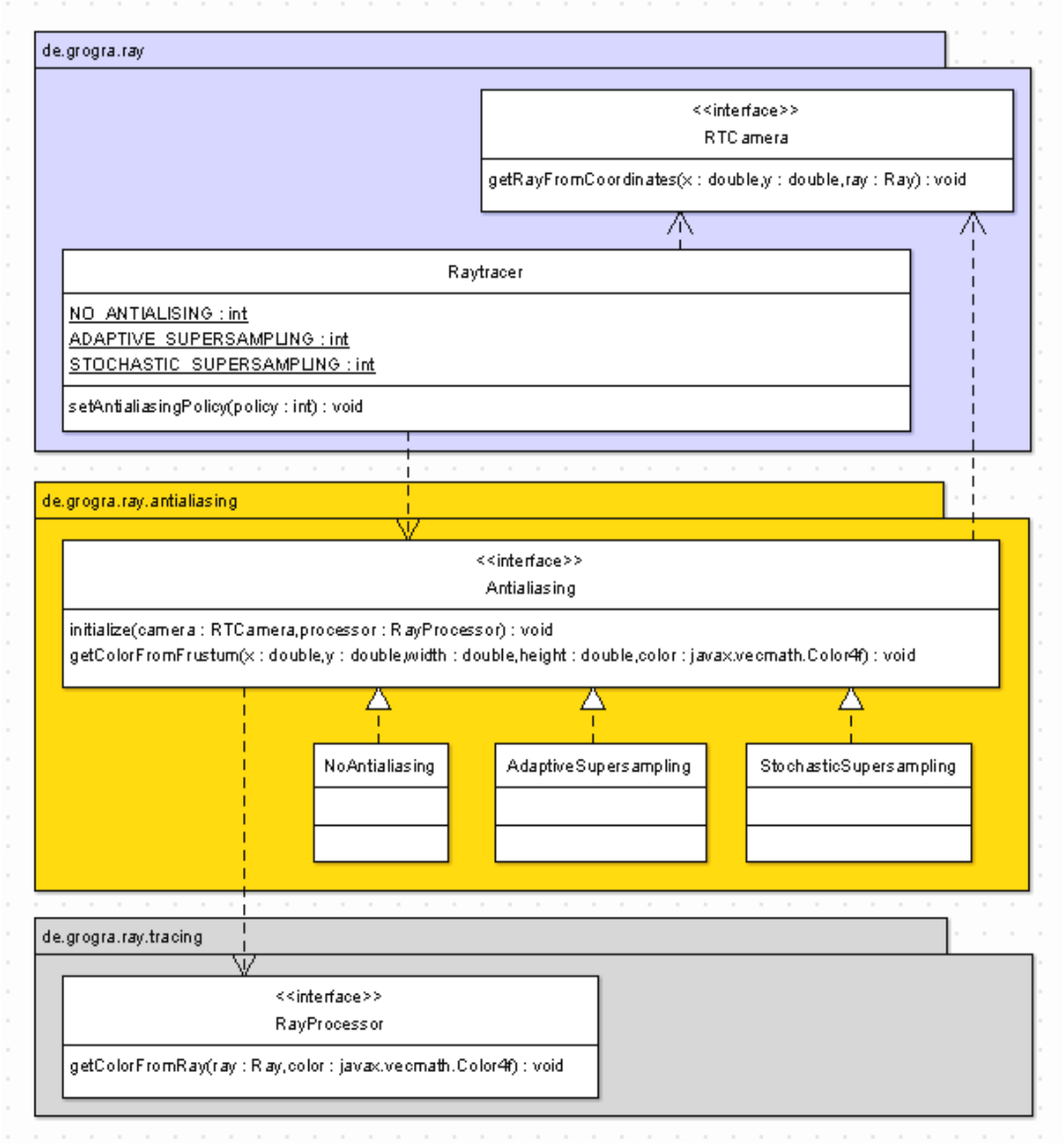


Abbildung 7 – Klassendiagramm Antialiasing

Für den Fall, dass kein Antialiasing berücksichtigt werden soll, gibt die Klasse `NoAntialiasing` einfach den Farbwert zurück, der durch die Verfolgung des Strahls, der mitten durch die Pixel-Pyramide verläuft, ermittelt wird.
Im Folgenden werden noch kurz zwei weitere implementierte Antialiasing-Methoden vorgestellt.

4.2.1. Stochastisches Supersampling

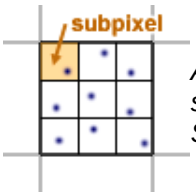


Abbildung 8 - stochastisches Supersampling

Beim Supersampling oder Nachfiltern wird ein virtuelles Bild der Szene mit n-facher Auflösung gerendert, und durch Faltung des virtuellen Bildes entsteht das gewünschte geglättete Bild. Für den Ansatz des stochastischen Supersamplings werden pro Pixel n zufällig gestreute Abtastwerte generiert. Um eine möglichst gleichmäßige Verteilung der Abtastwerte zu garantieren, wird der Originalpixel, wie in Abbildung 8 gezeigt, in n Subpixel unterteilt, und pro Subpixel wird jetzt genau ein zufälliger Abtastwert erzeugt. Dieses Verfahren wird oft als Jittering bezeichnet.
Der Intensitätswert eines Farbkanals eines Pixels wird durch den einfachen Durchschnittswert

$$I = \frac{1}{N}\sum_{n} I_n$$

der Intensitäten der Abtastwerte aller Subpixel eines Pixels berechnet.

4.2.2. Adaptives Supersampling

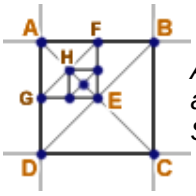


Abbildung 9 - adaptives Supersampling

Das adaptive Supersampling ist ein uneinheitliches Abtastverfahren, bei dem die Supersampling-Rate auf einzelne Bildbereiche (auf einzelne Pixel) angepasst wird. Das Verfahren beruht auf der Idee, dass ein Antialiasing nur für wenige Bildbereiche mit sehr hoher Supersampling-Rate durchgeführt werden muss. Für den größten Teil des Bildes ist nur eine minimale Rate erforderlich. Um die erforderliche Supersampling-Rate abschätzen zu können, wird angenommen, dass Bildbereiche, in denen sich die Intensitätswerte der Farbkanäle nicht stark unterscheiden, homogen sind. Diese Annahme muss nicht stimmen, aber da sie sehr oft zutrifft, kann sie als sinnvoll angesehen werden. Bildbereiche, in denen sich die Intensitätswerte sehr stark unterscheiden, sind inhomogen. Die Supersampling-Rate wird nach der Homogenität bestimmter Bereiche abgeschätzt, wobei für homogene Bereiche eine minimale Rate als ausreichend angenommen wird.

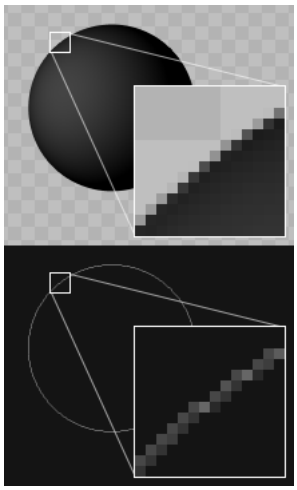


Abbildung 10 - angepasste Supersampling-Rate

Implementiert wurde das Verfahren rekursiv. Initial werden die Intensitätswerte der Farbkanäle für alle 4 Ecken des Pixels bzw. Subpixels sowie des Mittelpunktes berechnet. Der Intensitätswert des Pixels bzw. Subpixels wird als Durchschnittswert

$$I = \frac{1}{4}(I_{AE} + I_{BE} + I_{CE} + I_{DE})$$

der Intensitäten der vier Viertel-Subpixel gebildet. Fällt der Unterschied zweier berechneter Ecken eines Viertel-Subpixels unter eine bestimmte Schranke oder ist die maximale Rekursionstiefe von vier erreicht, so wird die Intensität dieses Viertel-Subpixels als Durchschnittswert berechnet.

$$I_{AE} = \frac{1}{2}(I_A + I_E)$$

Andernfalls wird das Verfahren auf den Viertel-Subpixel angewandt.
Die Intensitäten für die in Abbildung 9 gezeigte Unterteilung bis zur Rekursionstiefe 2 würden berechnet werden durch:

$$I = \frac{1}{4}\{\frac{1}{4}[\frac{1}{2}(I_A + I_H) + \frac{1}{2}(I_F + I_H) + \frac{1}{2}(I_E + I_H) + \frac{1}{2}(I_G + I_H)] + \frac{1}{2}(I_B + I_E) + \frac{1}{2}(I_C + I_E) + \frac{1}{2}(I_D + I_E)\}$$

In Abbildung 10 ist unten die Anzahl der Unterteilungen jedes Pixels für das obere Bild dargestellt. Je heller ein Pixel im unteren Bild dargestellt ist, desto öfter wurde er rekursiv unterteilt. Für schwarze Pixel wurde nur ein einziger Durchschnittswert gebildet. Es ist gut zu erkennen, dass nur für die Kontur der Kugel eine höhere Supersampling-Rate verwendet wurde. Bildbereiche in denen sich die Farbintensitätswerte nur wenig ändern, wurden nicht unterteilt.

Caching

Da bei diesem Verfahren die Farbwerte eines Pixels aus den Farbwerten der vier Eckpunkte bestimmt werden, müssten diese Werte für Eckpunkte, die nicht auf dem Rand des Bildes liegen, mindestens vier Mal berechnet werden. Es ist daher nahe liegend, das Verfahren durch einen Caching-Mechanismus zu optimieren.

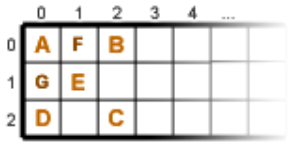


Abbildung 11 - Caching-Matrix

Da alle zu berechnenden Punkte in einem Raster liegen, können berechnete Werte in einer Matrix gecacht werden. Der Index der einzelnen Punkte in dieser Matrix kann durch einfache Funktionen direkt berechnet werden. Die Caching-Matrix wird so dimensioniert, dass sie alle Eckpunkte für Pixel bzw. Subpixel aller Rekursionstiefen aufnehmen kann. Außerdem werden die Mittelpunkte der Pixel bzw. Subpixel bis zur vorletzten Rekursionstiefe aufgenommen, da diese möglicherweise in rekursiv aufgerufenen Funktionen benötigt werden können.
Es kann vorausgesetzt werden, dass das zu rendernde Bild zeilenweise aufgebaut wird. Daher ist eine Matrix für genau eine Bildzeile ausreichend. Wenn das Ende einer Bildzeile erreicht ist, wird die unterste Zeile der Caching-Matrix nach oben kopiert, und alle Matrixzeilen bis auf die oberste werden invalidiert. In Abbildung 11 ist der Aufbau einer solchen Matrix für eine maximale Rekursionstiefe von 2 dargestellt.
Abbildung 12 zeigt das Laufzeitverhalten mit und ohne Caching für verschiedene Szenen. Es ist zu erkennen, dass das Caching besonders für große und komplexere Szenen deutliche Vorteile bringt.

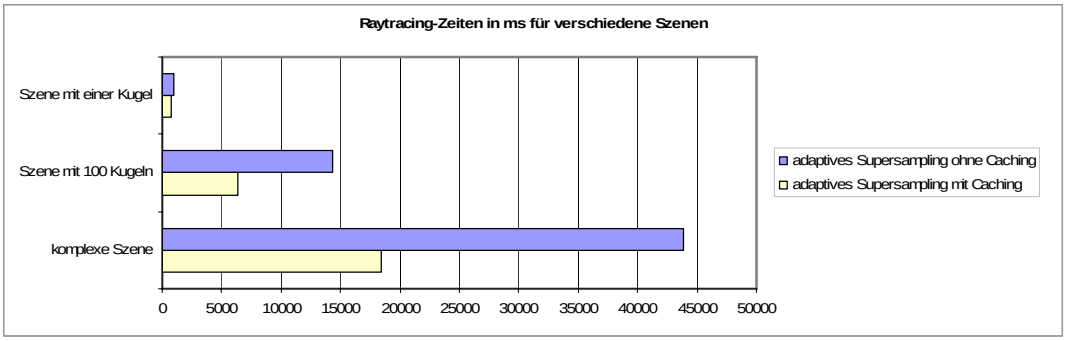


Abbildung 12 - Zeitmessungen für adaptives Supersampling

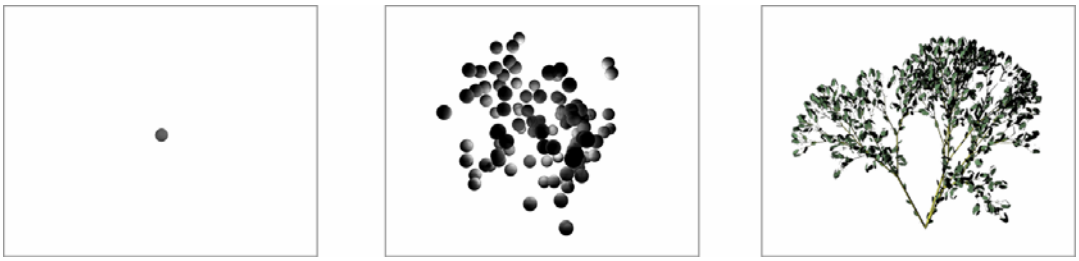


Abbildung 13 - Szene mit einer Kugel, Szene mit 100 Kugeln, komplexe Szene

4.3. Raytracing-Algorithmus

Die durch das Interface `RayProcessor` implementierten Klassen bilden jeweils das Herzstück des Raytracers. Jeder `RayProcessor` wird mit der Szene und einem `IntersectionProcessor` initialisiert und berechnet durch die Methode `getColorFromRay` den Farbwert für einen inversen Lichtstrahl, der vom Kamerazentrum ausgeht. Unter Verwendung der `IntersectionProcessor`-Instanz ist es dem `RayProcessor` möglich, Schnittpunkte mit Szenenobjekten zu bestimmen und Schnittpunktbeschreibungen abzufragen. Diese Beschreibungen enthalten auch einen Verweis auf ein `RTObject`, welches Informationen über das geschnittene Objekt kapselt. Das `RTObject` enthält wiederum einen Verweis auf eine `RTShader`-Instanz, welche zur Beschreibung der physikalischen Materialeigenschaften dient. Mit Hilfe dieses `RTShader` ist es möglich, den Farbwert für eine lokale Beleuchtungssituation abzufragen, wobei vorher durch den `LightProcessor` alle direkten Lichtstrahlen bestimmt werden müssen, die für den jeweiligen Schnittpunkt relevant sind. Auf die Funktionsweise des `LightProcessor` wird im nächsten Abschnitt *4.4 Licht* näher eingegangen. Ein Überblick über die beschriebenen Zusammenhänge der Klassen und Interfaces, die im Zusammenhang mit dem `RayProcessor` von Bedeutung sind, ist in Abbildung 14 dargestellt.

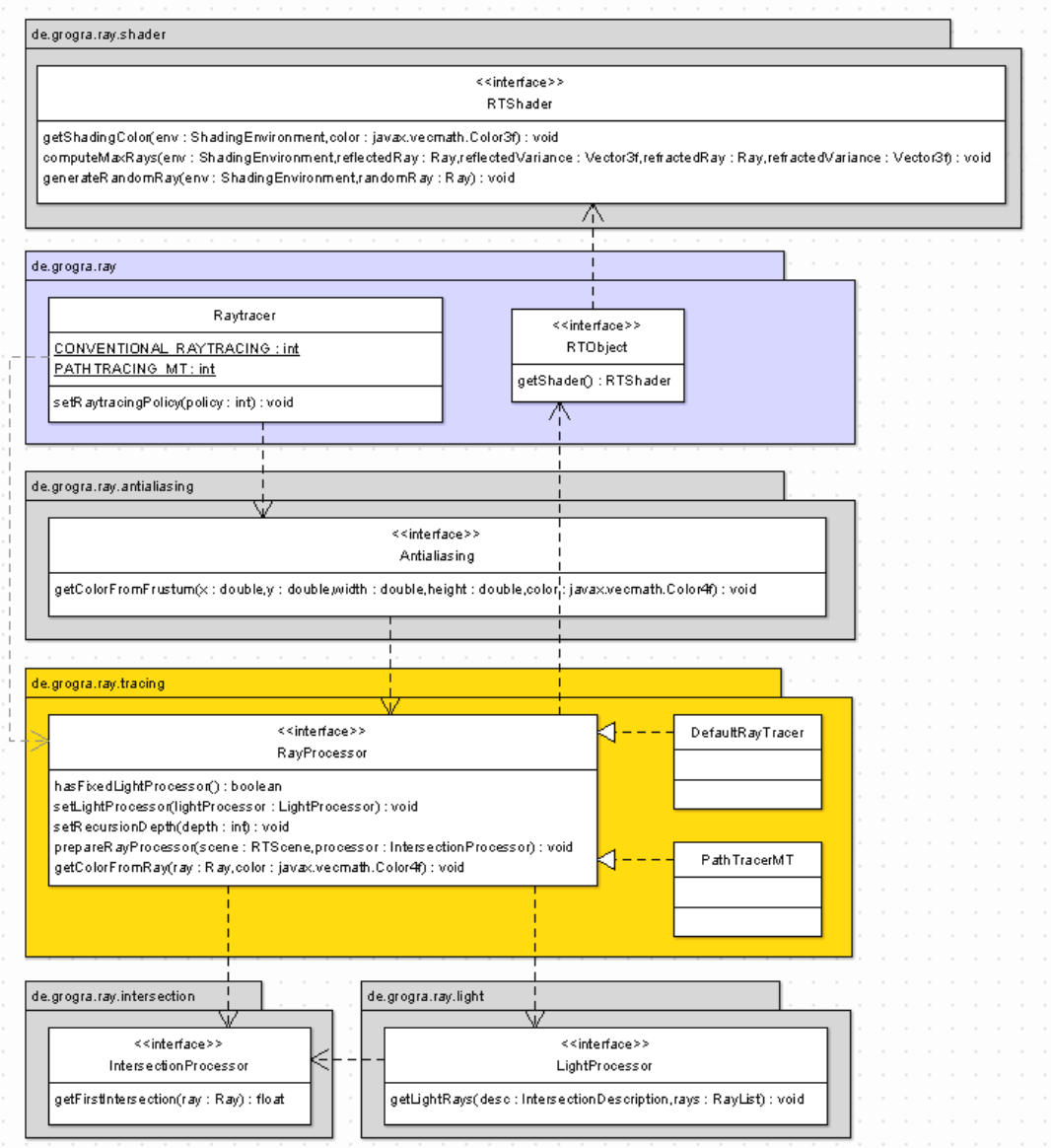


Abbildung 14 - Klassendiagramm Raytracing

Im Weiteren soll etwas spezieller auf zwei verschiedene Implementierungen des Interfaces `RayProcessor` eingegangen werden. Die Klasse `DefaultRayTracer` stellt eine Implementierung dar, die den wahrscheinlich ältesten und geläufigsten Raytracing-Algorithmus realisiert. Hierbei werden die Reflexionseigenschaften aller Materialien als stark vereinfacht und idealisiert vorausgesetzt. Diffuse Materialien, wie der in GroIMP zu Verfügung stehende Lambert-Shader, reflektieren kein Licht. Dagegen reflektieren andere Materialien, wie zum Beispiel der Phong-Shader in GroIMP, jeden Lichtstrahl ideal. Die Brechung der Lichtstrahlen ist in beiden Fällen als ideal anzusehen. Um die ideal reflektierten und gebrochenen Lichtstrahlen zu ermitteln, stellt jede `RTShader`-Klasse die Methode `computeMaxRays` zur Verfügung. Auf die Implementierungen der verschiedenen `RTShader`-Klassen wird hier nicht näher eingegangen, da sie zu Beginn der Arbeit bereits entwickelt waren und nicht Teil der Arbeit selbst sind.
Eine zweite Implementierung stellt die Klasse `PathTracerMT` dar. Sie implementiert einen Pathtracing-Algorithmus, bei dem nicht nur die Wege von ideal reflektierten Strahlen verfolgt werden, sondern durch gegebene Wahrscheinlichkeitsverteilungen, die durch BRDFs (Bidirectional Reflectance Distribution Functions) bzw. (Bidirectional Transfer Distribution Functions) in den Shadern definiert sind, auch andere Pfade für die Strahlen zufällig ausgewählt werden. Die Methode `generateRandomRay` der Klasse `RTShader` erzeugt, entsprechend der definierten BRDF bzw. BTDF, genau einen zufällig reflektierten oder gebrochenen Strahl. Durch die Angabe einer Pfadanzahl wird festgelegt, wie viele Pfade für einen von der Kamera ausgehenden Strahl verfolgt und schließlich gewichtet zusammengefasst werden sollen. Die Gewichtung eines Pfades entspricht der Gesamtwahrscheinlichkeit dieses generierten Pfades.

4.4. Licht

Obwohl für die Lichtberechnung nur eine Variante implementiert wurde, ist sie trotzdem modular gestaltet worden, um spätere Erweiterungen durch Austauschen der konkreten Implementierung des Interfaces `LightProcessor` zu ermöglichen.
Die Klasse `DefaultLightProcessor` implementiert einen `LightProcessor`, der speziell für Raytracing-Verfahren benötigt wird, die ausschließlich inverse Lichtstrahlen verfolgen. Die Methode `getLightRays` ermittelt zu allen Lichtquellen alle Schattenstrahlen, die für einen bestimmten Schnittpunkt von Bedeutung sind. Ob ein möglicher Schattenstrahl einen Beitrag zur direkten Beleuchtung im betrachteten Schnittpunkt hat, wird vom übergebenen `ShadowProcessor` entschieden.
Für das Interface `ShadowProcessor` wurden zwei Implementierungen entwickelt. Die Klasse `NoShadows` ignoriert für solide Objekte Schattenstrahlen (Strahlen vom Objektschnittpunkt zur Lichtquelle), die in entgegengesetzter Richtung der Flächennormalen verlaufen. Für nicht-solide Objekte werden Schattenstrahlen ignoriert, die sich auf der entgegengesetzten Seite der betrachteten Fläche befinden. Somit werden in der Schattenberechnung nur die Schattierungen der Objekte selbst mit einbezogen, nicht aber die Schatten anderer Objekte.
Eine zweite Implementierung des Interfaces `ShadowProcessor` stellt die Klasse `Shadows` dar. Diese Klasse bezieht in die Schattenberechnung nicht nur das Objekt selbst mit ein, sondern auch alle anderen Objekte der Szene. Um Schnittpunkte der Schattenstrahlen mit Szenenobjekten berechnen zu können, wird ihr initial ein `IntersectionProcessor` übergeben. Mit Hilfe der Methode `getFirstIntersectionT` kann bestimmt werden, ob sich zwischen Lichtquelle und Schnittpunkt weitere Objekte befinden, die den Schnittpunkt abschatten würden. Somit nutzen die Raytracing-Algorithmen wie auch die Schattenberechnungen dieselbe Schnittpunktberechnung.
In Abbildung 15 ist eine Übersicht der wichtigsten Interfaces und Klassen zur Licht- und Schattenberechnung dargestellt.

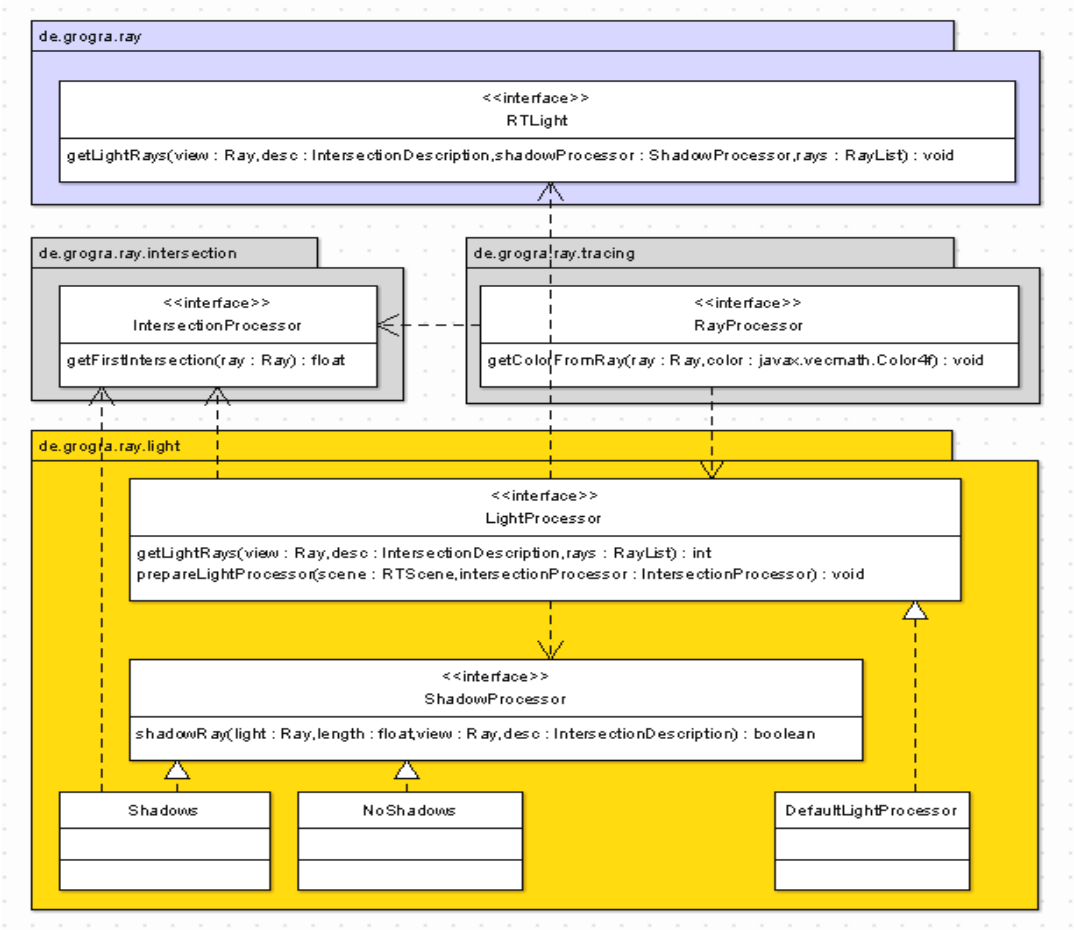


Abbildung 15 - Klassendiagramm Licht- und Schattenberechnung

4.5. Schnittpunktberechnung

Die Funktionen der Schnittpunktberechnung werden sowohl vom Raytracing-Algorithmus als auch von der Licht- und Schattenberechnung benötigt. Daher werden alle Implementierungen der Interfaces `RayProcessor` und `LightModel` mit einer Instanz des Interfaces `IntersectionProcessor` initialisiert.
Das Interface `IntersectionProcessor` kapselt sehr allgemein die Funktionalitäten der Schnittpunktberechnung und stellt Methoden zur Verfügung, die für einen gegebenen Strahl den nächsten Schnittpunkt mit einem Objekt der Szene berechnen. Eine solche Funktion wird in zwei Varianten angeboten. Zum einen kann durch die Methode `getFirstIntersectionT` der Abstand bis zum ersten Schnittpunkt abgefragt werden. Zum anderen erzeugt die Methode `getFirstIntersectionDescription` eine detailierte Beschreibung des ersten Schnittpunktes. Um die zweite Methode benutzen zu können, muss zuerst `getFirstIntersectionT` mit denselben Hilfsparametern aufgerufen werden.
Um Berechnungsfehler zu vermeiden, die aus Ungenauigkeiten von Gleitkommaoperationen resultieren, werden den Methoden `getFirstIntersectionT` und `getFirstIntersectionDescription` neben dem Strahl (dieser wird durch die Klasse `Ray` definiert) zusätzliche Kontextinformationen (diese werden in der Klasse `RayContext` definiert) mit übergeben. Diese Kontextinformationen dienen der Optimierung von Schnittpunktberechnungen mit soliden konvexen Objekten. So kann zum Beispiel für einen nach außen reflektierten Strahl ein Schnittpunkt mit dem reflektierenden Objekt selbst ausgeschlossen werden. Außerdem kann mit Hilfe dieser Informationen bestimmt werden, ob der erste oder zweite Schnittpunkt des Strahls mit dem zu testenden Objekt der gesuchte ist. Für

Strahlen innerhalb eines transparenten Objektes wird immer der zweite Schnittpunkt mit diesem Objekt der gesuchte sein, und für Strahlen außerhalb des Objektes ist es immer der erste.
Im Rahmen dieser Arbeit wurden zwei verschiedene Implementierungen der Schnittstelle `IntersectionProcessor` entwickelt. Die Klasse `DefaultIntersectionProcessor` implementiert einen naiven und einfachen Ansatz. Um zum Beispiel den ersten Schnittpunkt eines Strahls mit Objekten der Szene zu bestimmen, werden in einer Schleife alle in der Szene befindlichen Objekte getestet, und der Schnittpunkt mit dem kürzesten Abstand zum Strahlursprung entspricht dem gesuchten. Eine durch Hüllobjekte optimierte Implementierung stellt die Klasse `OctreeIntersectionProcessor` dar. Auf sie wird im Abschnitt *4.5.2 Optimierung durch einen Octree* näher eingegangen.

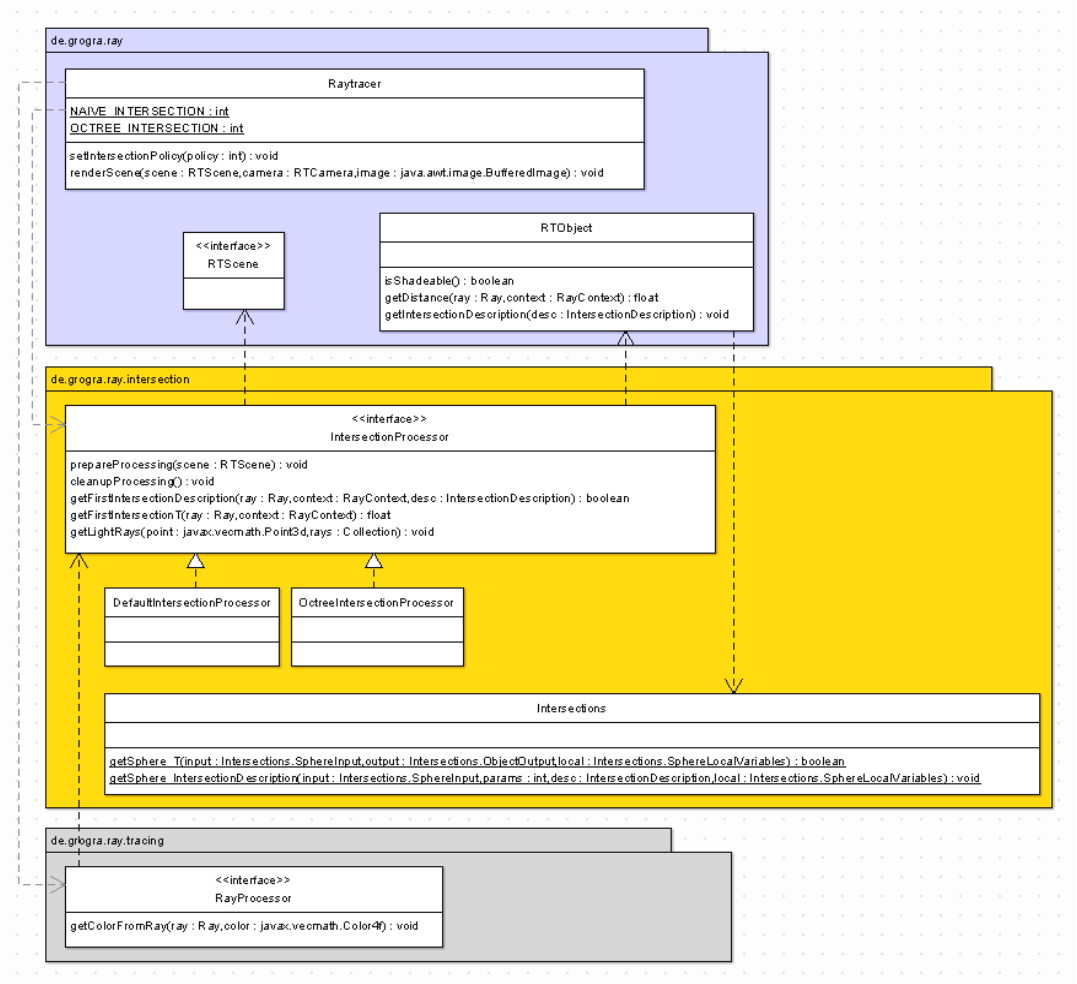


Abbildung 16 - Klassendiagramm Schnittpunktberechnung

Anders als in der Aufgabenstellung dieser Arbeit vorgeschlagen, wurden die Schnittpunktberechnungen der Primitiv-Objekte nicht in den Primitivklassen implementiert, sondern als öffentliche statische Methoden in der Klasse `Intersections` verwirklicht. Diese Entscheidung kann durch den Framework-Charakter der Arbeit begründet werden, da die Schnittpunktberechnungen auch Teil des Raytracers sind und nicht spezifisch für die benutzende Anwendung GroIMP implementiert werden müssen. Die Primitivklassen selbst implementieren das Interface `RTObject` und können die benötigten statischen Methoden der Klasse `Intersections` nutzen. In Abbildung 16 wird eine Übersicht über die wichtigsten Klassen und Schnittstellen gegeben, die in Bezug auf die Schnittpunktberechnungen von Bedeutung sind.

4.5.1. Schnittpunktberechnungen der Grundobjekte

Die elementaren Schnittpunktberechnungen für die Grundobjekte werden in der Klasse `Intersections` zusammengefasst. Diese Klasse bietet statische Methoden für Schnittpunktberechnungen eines Strahls mit einer Kugel, einer Ebene, einem Parallelogramm, einem Quader, einem Zylinder, einem Kegel und einem Kegelstumpf an.
Für jede Schnittpunktberechnung stehen zwei Stufen der Berechnung zur Verfügung. Die erste Stufe berechnet den Abstand eines Objekt-Schnittpunktes vom Ursprung des schneidenden Strahls. Für einen Strahl, der in der Parameterform $\vec{x} = \vec{g} + t \cdot \vec{d}$ gegeben ist, wird also der Parameter t für den Schnittpunkt berechnet. In einer zweiten Stufe können zu einem zuvor berechneten Schnittpunkt weitere Parameter wie z.B. Normalenvektor, Tangentenvektoren und Texturkoordinaten berechnet werden.
Da die Schnittpunktberechnung in einem Raytracer sehr zeitkritisch ist, gibt es in keiner Methode lokale Variablen. Jeder Methode werden vielmehr Datenstrukturen übergeben, die den Speicherplatz aller benötigten lokalen Variablen bereits enthalten. Dadurch muss der Speicher für die lokalen Variablen nicht bei jeden Aufruf alloziert und wieder freigegeben werden. Auf dieses Verfahren wird im Abschnitt *4.7 Optimierung* näher eingegangen.
Im Folgenden soll kurz eine Herleitung für die Schnittpunktberechnung einiger Grundobjekte gegeben werden. Um die Berechnung der Parameter stark zu vereinfachen, werden die Schnittpunktberechnungen nicht an den transformierten Grundobjekten ausgeführt, sondern es wird ein invers-transformierter Strahl an untransformierten Standard-Objekten getestet.

Die Kugel

Das einfachste zu berechnende solide Grundobjekt ist die Kugel. Die Gleichung für die Ermittlung der Schnittpunkte einer Geraden mit einer Kugel ist kurz und lässt sich schnell berechnen, weshalb Kugeln oft auch als Bounding-Körper verwendet werden.

Es sei eine Gerade in Parameterform mit $\vec{x} = \vec{g} + t \cdot \vec{d}$ gegeben. Weiterhin sei eine Kugel im Ursprung mit gegebenem Radius durch $x^2 + y^2 + z^2 = radius^2$ definiert. Durch Einsetzen der einzelnen Komponenten der Geradengleichung in die Kugelgleichung erhält man

$$(g_x + t \cdot d_x)^2 + (g_y + t \cdot d_y)^2 + (g_z + t \cdot d_z)^2 = radius^2 .$$

Nach dem Ausmultiplizieren ergibt sich eine quadratische Gleichung mit nur einer Unbekannten

$$t^2 \cdot (d_x{}^2 + d_y{}^2 + d_z{}^2) + t \cdot (2g_x d_x + 2g_y d_y + 2g_z d_z) + (g_x{}^2 + g_y{}^2 + g_z{}^2 - radius^2) = 0 .$$

Daraus ergeben sich unmittelbar die Parameter der allgemeinen quadratischen Gleichung $at^2 + bt + c = 0$

$$a = \vec{d} \circ \vec{d}$$

$$b = 2 \cdot (\vec{g} \circ \vec{d})$$

$$c = (\vec{g} \circ \vec{g}) - radius^2 .$$

Diese hat die Lösungen

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$

Mit Hilfe der Diskriminante $D = b^2 - 4ac$ kann die Anzahl der Lösungen bestimmt werden. Ist D negativ, gibt es keinen Schnittpunkt. Für positive Werte gibt es zwei Schnittpunkte, und falls D gleich null ist, gibt es genau einen Schnittpunkt.

Um die Schnittpunktberechnungen auf Strahlen anstatt Geraden zu spezialisieren, muss am Ende geprüft werden, ob t positiv ist.

Der Zylinder

Um die Schnittpunkte einer Geraden mit einem Zylinder zu bestimmen, müssen die drei verschiedenen Mantelflächen einzeln geprüft werden.
Für den Körper werden die Schnittpunkte einer Röhre mit einer Geraden ermittelt. Es seien eine Gerade in Parameterform und eine Röhre im Ursprung mit definiertem Radius und definierter Höhe gegeben. Die Rotationsachse für die Röhre sei die Z-Achse. Für eine Röhre beliebiger Längenausdehnung gilt: $x^2 + y^2 = radius^2$. Durch Einsetzen der X- und Y-Komponente der Geradengleichung ergibt sich

$$(g_x + t \cdot d_x)^2 + (g_y + t \cdot d_y)^2 = radius^2 .$$

Durch Umformen erhält man die quadratische Gleichung

$$t^2 \cdot (d_x{}^2 + d_y{}^2) + t \cdot (2 g_x d_x + 2 g_y d_y) + g_x{}^2 + g_y{}^2 - radius^2 = 0$$

mit den Parametern

$$a = d_x{}^2 + d_y{}^2$$

$$b = 2 g_x d_x + 2 g_y d_y$$

$$c = g_x{}^2 + g_y{}^2 - radius^2$$

und den Lösungen

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$

Für jedes positive t kann ein Schnittpunkt $\vec{p}$ für die Einheitsröhre berechnet werden. Falls $0 \le p_z \le höhe$ gilt, handelt es sich um einen Schnittpunkt mit dem Einheitszylinder auf der Körperfläche.
Für die Basisfläche wird der Schnittpunkt einer Geraden mit einer Kreisfläche berechnet. Gegeben sei die Gerade in Parameterform. Die gegebene Kreisfläche soll in der X,Y-Ebene liegen mit dem Mittelpunkt im Ursprung. Mit der Bedingung $z = 0$ kann der Schnittpunkt mit der X,Y-Ebene ermittelt werden. Durch einfaches Einsetzen der Z-Komponente aus der Geradengleichung ergibt sich: $g_x + t \cdot d_x = 0$. Somit gibt es die Lösung $t = -\frac{g_x}{d_x}$, falls $d_x \neq 0$ gilt und t positiv ist. Für ein mögliches t kann der Schnittpunkt $\vec{p}$ berechnet werden. Wenn gilt $p_x{}^2 + p_y{}^2 \le radius^2$, schneidet der Strahl die Basis-Kreisfläche.
Für die Schnittpunktberechnung der Geraden mit der Deckel-Kreisfläche kann analog vorgegangen werden. Der Unterschied ist nur, dass sich die Kreisfläche in der Ebene $z = höhe$ befindet. Daraus ergibt sich die mögliche Lösung $t = höhe - \frac{g_x}{d_x}$.

Der Kegel

Die Schnittpunktberechnung einer Geraden mit einem Kegel ist ähnlich der einer Geraden mit einem Zylinder. Allerdings entfällt zum einen die Berechnung mit der Deckel-Mantelfläche, und zum anderen ist die Berechnung für die Röhre abgewandelt. In der für den Zylinderkörper verwendeten Bedingung $x^2 + y^2 = radius^2$ muss der Radius als veränderliche Variable ersetzt werden. Für einen Einheitskegel mit der Höhe 1 und einem Radius von 1 ergibt sich die Bedingung $radius = 1 - z$. Durch Ersetzen des festen Radius ergibt sich:

$x^2 + y^2 = 1 - 2z + z^2$. Durch Einsetzen der Komponenten der Geradengleichung erhält man:

$$(g_x + t \cdot d_x)^2 + (g_y + t \cdot d_y)^2 = 1 - 2(g_z + t \cdot d_z) + (g_z + t \cdot d_z)^2 .$$

Durch Umformung ergibt sich die quadratische Gleichung

$$t^2 \cdot (d_x^{\,2} + d_y^{\,2} - d_z^{\,2}) + t \cdot (2g_x d_x + 2g_y d_y - 2g_z d_z + 2d_z) + g_x^{\,2} + g_y^{\,2} - g_z^{\,2} + 2g_z - 1 = 0$$

mit den Parametern

$$a = d_x^{\,2} + d_y^{\,2} - d_z^{\,2}$$

$$b = 2g_x d_x + 2g_y d_y + 2d_z(1 - g_z)$$

$$c = g_x^{\,2} + g_y^{\,2} - (1 - g_z)^2$$

und den Lösungen

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$

Für jedes positive t kann ein Schnittpunkt $\vec{p}$ für die Kegelkörperfläche berechnet werden. Falls $0 \le p_z \le 1$ gilt, handelt es sich um einen Schnittpunkt mit dem Einheitskegel auf der Körperfläche.
Die Schnittpunktberechnung für die Basiskreisfläche erfolgt analog zum Zylinder.

Der Quader

Um die Schnittpunkte einer Geraden mit einem Quader zu bestimmen, müssen alle sechs Mantelflächen geprüft werden. Exemplarisch soll das für eine Fläche gezeigt werden.
Es seien eine Gerade in Parameterform und ein achsenparalleler Quader mit Mittelpunkt im Ursprung gegeben. Eine dieser Begrenzungsflächen muss in der Ebene $z = \frac{1}{2} ausdehnung_z$ liegen. Durch Einsetzen der Z-Komponente und Umstellen der Gleichung ergibt sich:

$$t = \frac{1}{2} ausdehnung_z - \frac{g_z}{d_z} \text{ mit } d_z \neq 0 .$$

Für ein positives t kann der Schnittpunkt $\vec{p}$ ermittelt werden. Falls gilt:

$$-\frac{1}{2} ausdehnung_x \le p_x \le \frac{1}{2} ausdehnung_x \text{ und}$$

$$-\frac{1}{2} ausdehnung_y \le p_y \le \frac{1}{2} ausdehnung_y ,$$

handelt es sich um einen Schnittpunkt mit dem Quader.
Wenn gegeben ist, ob der eindringende oder austretende Schnittpunkt gesucht wird, kann die Berechnung optimiert werden. Ausgehend von der Richtung der schneidenden Geraden ist es ausreichend, nur für drei Flächen die Schnittpunkte zu berechnen.

Die Ebene

Ein besonderes Grundobjekt ist die Ebene. Sie ist nicht solide und hat daher höchstens eine Lösung.
Es sei wieder eine Gerade in Parameterform gegeben. Die Ebene, welche die X,Y-Ebene sein soll, sei durch $z = 0$ gegeben. Durch einfaches Einsetzen der Z-Komponente aus der Geradengleichung ergibt sich: $g_x + t \cdot d_x = 0$. Somit gibt es die Lösung $t = -\frac{g_x}{d_x}$, falls $d_x \neq 0$ gilt und t positiv ist.

4.5.2. Optimierung durch einen Octree

Um die Laufzeit der Schnittpunktberechnungen zu verringern, gibt es grundsätzlich zwei Möglichkeiten. Zum einen können die einzelnen Strahl-Objekt-Schnittpunktberechnungen beschleunigt werden und zum anderen kann die Zahl der Strahl-Objekt-Berechnungen selbst verringert werden. Das Ziel der zweiten Beschleunigungsvariante besteht also darin, nur eine beschränkte Zahl von Objekten auf einen Schnittpunkt testen zu müssen, unabhängig von der tatsächlichen Anzahl der Objekte in der Szene. Um dieses Ziel zu erreichen, wird die Szene meist in irgendeiner Form aufgeteilt, und alle Objekte werden den speziellen Unterteilungen zugeordnet. Ein Strahl, der sich durch die Szene bewegt, wird nicht gegen die Szenenobjekte, sondern zunächst gegen die Unterteilungen getestet. Kreuzt der Strahl eine Unterteilung, werden erst an dieser Stelle Schnittpunkte mit den zugewiesenen Objekten berechnet. Dadurch kann die Anzahl der zu testenden Szenenobjekte stark verringert werden.
Eine Möglichkeit, die Szene zu unterteilen, bietet der Octree. Er ist das dreidimensionale Pendant zum zweidimensionalen Quadtree und unterteilt die Szene hierarchisch und uneinheitlich.
Formal ist ein Octree ein geordneter Baum der Ordnung 8. Er wird rekursiv erzeugt, indem jeder entstehende Quader – anfangs der einhüllende Szenenquader – nach einer bestimmten Bedingung achsenparallel in acht gleichgroße Kindquader unterteilt wird. Die Reihenfolge der entstehenden Kindquader legt hierbei ihre räumliche Anordnung eindeutig fest. Hat der Baum eine bestimmte Tiefe erreicht, wird die Unterteilung abgebrochen.
Für den Raytracer enthält jeder Quader eine Liste mit Szenenobjekten, die sich teilweise oder ganz in ihm befinden. Als Bedingung, ob ein Quader unterteilt wird, kann eine maximale Anzahl von Objekten definiert werden, die sich in einem Quader befinden dürfen. Falls sich mehr Objekte in einem Quader befinden, wird dieser unterteilt. In Abbildung 17 ist für eine Beispielszene der erzeugte Octree dargestellt. Quader, deren Objektliste leer ist, wurden in der Visualisierung ausgeblendet.

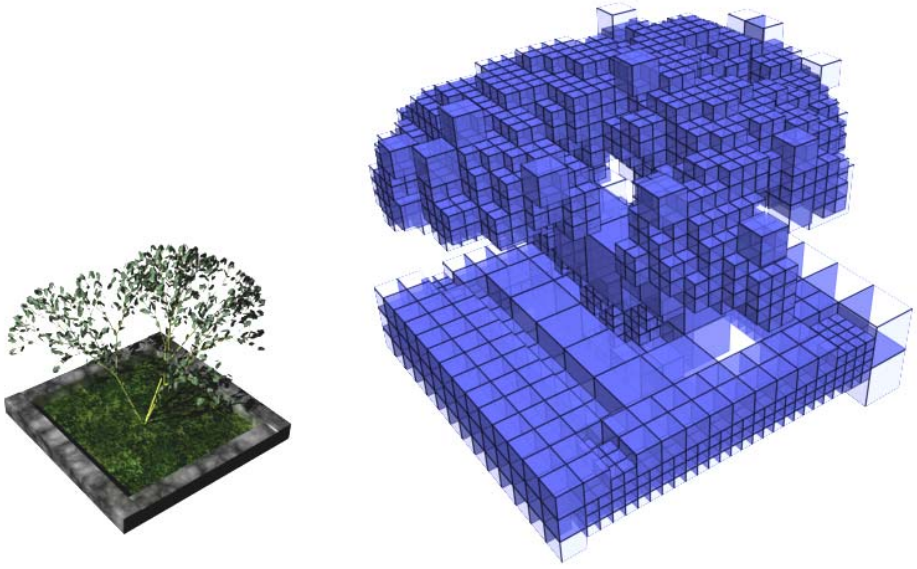


Abbildung 17 - Beispielszene mit zugehörigem Octree der Tiefe fünf

Bei der Verfolgung eines Strahls durch die Szene wird ein Algorithmus benötigt, der eine Liste aufeinander folgender Blattzellen des Octrees liefert, welche der Strahl durchdringt. Dieser Algorithmus wird in [3] als Strahlgenerator bezeichnet. Die Blattzellen der generierten Liste umschließen den Stahl vollständig und werden als Strahlzellen bezeichnet. In Abbildung 18 sind in einen Quadtree alle für einen Strahl generierten Strahlzellen markiert.

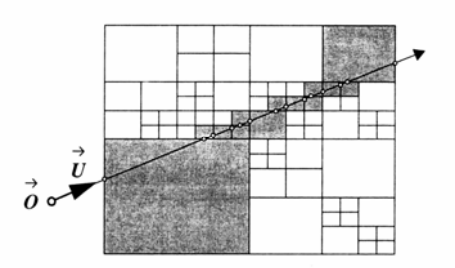


Abbildung 18 - generierte Strahlzellen, aus [3]

Im Raytracer werden Strahlgeneratoren durch das Interface `CellGenerator` beschrieben. Die Klasse `DefaultCellGenerator` implementiert einen Strahlgenerator, der von Glassner 1984 vorgestellt wurde. Für einen Strahl, der den Octree durchdringt und sich in einer bestimmten Strahlzelle befindet, kann ein Austrittspunkt berechnet werden. Mit Hilfe des Austrittspunktes kann ein Punkt konstruiert werden, der sich auf jeden Fall innerhalb der nächsten Strahlzelle befindet. Dieser wird durch Addition bzw. Subtraktion eines Wertes erzeugt, der auf jeden Fall kleiner als die Kantenlänge der kleinsten Zelle des Octrees ist, erzeugt. Ausgehend von der Wurzel wird im Octree die Blattzelle gesucht, die den Punkt enthält. In Abbildung 19 ist das Vorgehen noch einmal schematisch dargestellt.

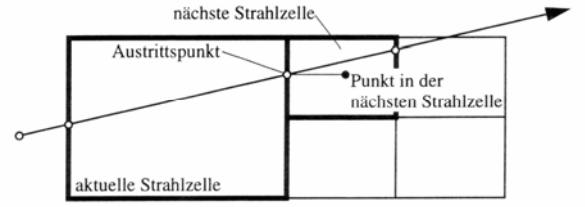


Abbildung 19 – Bestimmung der nächsten Strahlzelle, aus [3]

In der Arbeit [3] wird durch Laufzeitmessungen gezeigt, dass ein abgeänderter Strahlgenerator weitere Optimierungen bringen kann. Durch Verweise auf alle Nachbarblattzellen in jeder Blattzelle kann das Bestimmen der nächsten Strahlzelle deutlich beschleunigt werden. Jede Blattzelle erhält für jede ihrer sechs Seiten eine Quadtree-ähnliche Datenstruktur, die in ihren Blättern direkte Verweise auf Nachbarzellen trägt. In Abbildung 20 ist der Aufbau eines erweiterten Quadtrees stellvertretend für die erweiterte Octree-Datenstruktur dargestellt.

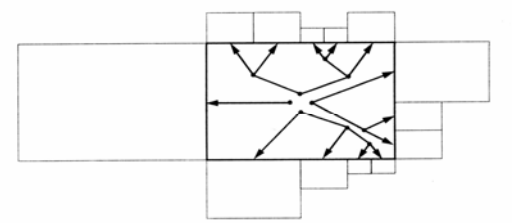


Abbildung 20 - Zelle eines Quadtrees mit Nachbar-Binärbäumen, aus [3]

Ein Strahlgenerator, der die erweiterte Octree-Datenstruktur verwendet, ist in der Klasse `EndlCellGenerator` implementiert. Eine Zusammenfassung der wichtigsten Interfaces und Klassen für die Octree-Optimierung ist in Abbildung 21 gegeben.

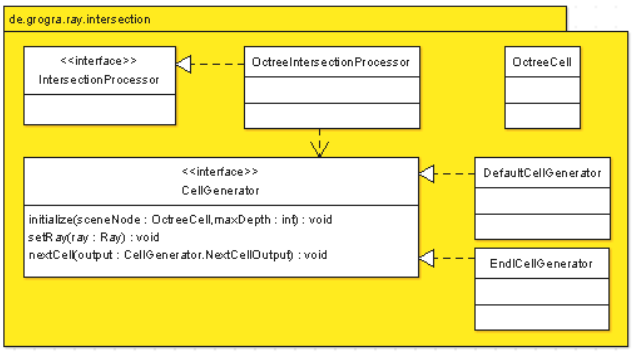


Abbildung 21 - Klassendiagramm Octree-Optimierung

Laufzeitverhalten des Raytracers mit Octree-Optimierung

Um die Optimierungen der Schnittpunktberechnung durch Verwendung eines Octrees bewerten zu können, wird im Folgenden das Laufzeitverhalten des Raytracers mit und ohne Optimierung verglichen.

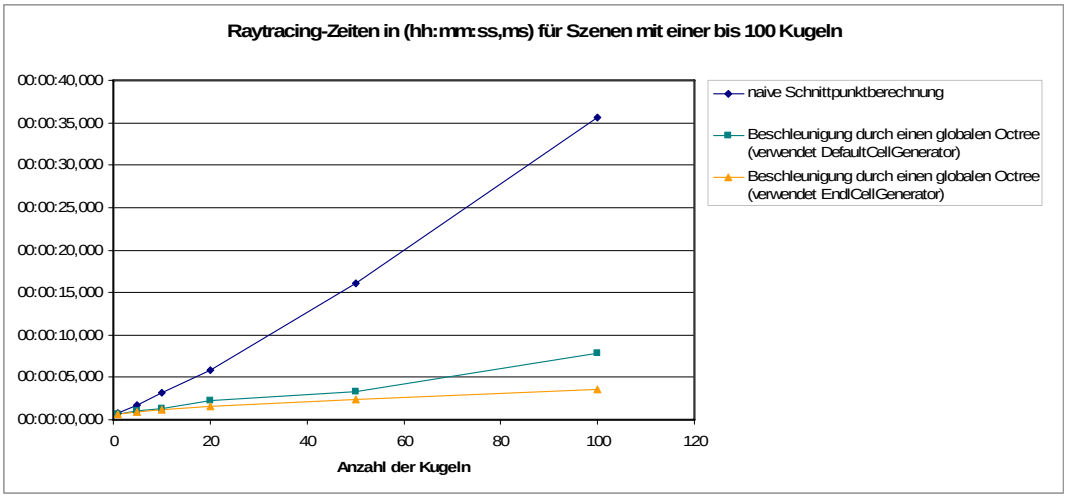


Abbildung 22 - Laufzeitverhalten für Szenen mit bis zu 100 Kugeln

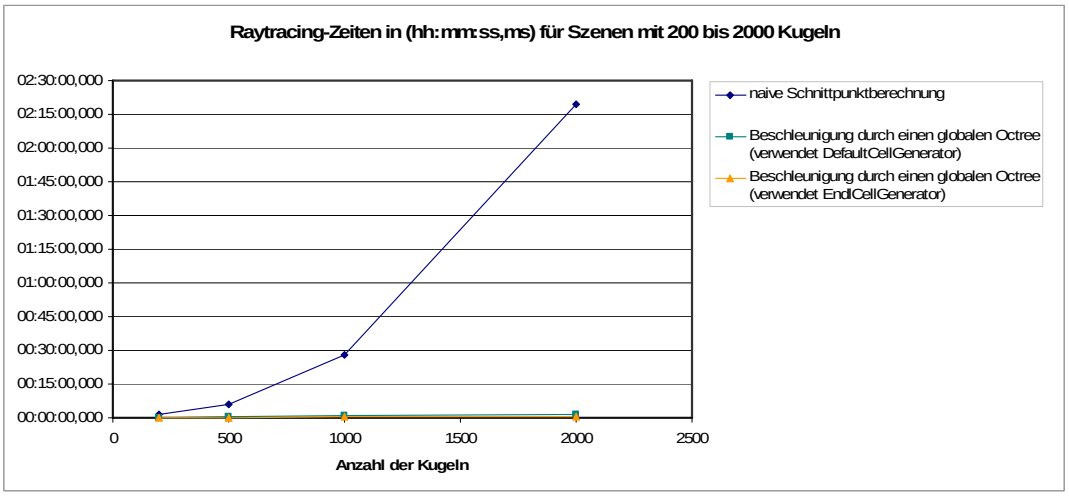


Abbildung 23 - Laufzeitverhalten für Szenen mit 200 bis 2000 Kugeln

In einer ersten Testreihe wurde in den Testszenen eine variable Anzahl von Kugeln und jeweils fünf Lichtquellen innerhalb der Sichtpyramide der Kamera erzeugt. Damit soll die Abhängigkeit der Laufzeit von der Anzahl der Objekte in

einer Szene untersucht werden. In der Testreihe wurden Szenen mit 1, 5, 10, 20, 50, 100, 200, 500, 1000 und 2000 Kugeln mit einem Radius von 0.3 erzeugt. Da die Laufzeiten der optimierten und nicht-optimierten Variante stark divergieren, wurde die Darstellung der Zeitmessungen auf zwei Diagramme aufgeteilt.

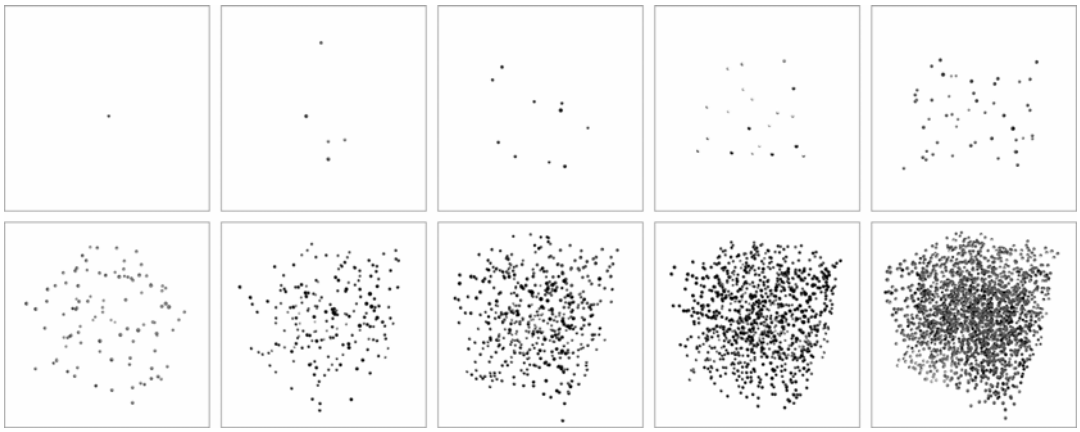


Abbildung 24 - Szenen mit einer bis 2000 Kugeln

Abbildung 22 zeigt die Laufzeiten für die Szene mit einer bis 100 Kugeln. Es ist zu erkennen, dass die optimierte Variante schon für eine Szene mit nur 10 Kugeln deutlich kürzere Raytracing-Zeiten aufweist als die naive Schnittpunktberechnung. Für die nicht-optimierte Variante steigen die Zeiten linear stark mit der Anzahl der Kugeln an.
Für die in Abbildung 23 dargestellte Laufzeitmessung für Szenen mit 200 bis 2000 Kugeln zeigt sich sogar ein noch schlechteres Laufzeitverhalten für die naive Schnittpunktberechnung. Außerdem lässt sich vermuten, dass die Optimierung durch den Octree besonders für Szenen mit vielen Objekten eine große Zeitersparnis für die Berechnung der Bilder bringt. Für eine Szene mit 10000 Kugeln mit einem Radius von 0.02 zeigt sich dies noch deutlicher. Hierfür braucht der Raytracer mit naiver Schnittpunktberechnung 1:18 Stunden, während er mit der Octree-Optimierung nur 2,5 Sekunden benötigt.
Da der Octree einen gewissen Overhead darstellt, können auch Szenen konstruiert werden, in denen die optimierte Variante Nachteile gegenüber der naiven Schnittpunktberechnung bringt. In einer zweiten Testreihe soll das Laufzeitverhalten der beiden Varianten für eine Szene mit nur einer Kugel, deren Radius variiert wird, untersucht werden. Es werden Szenen mit einer Kugel mit dem Radius 0.2, 1, 2, 5, 10 und 20 erzeugt. benötigen

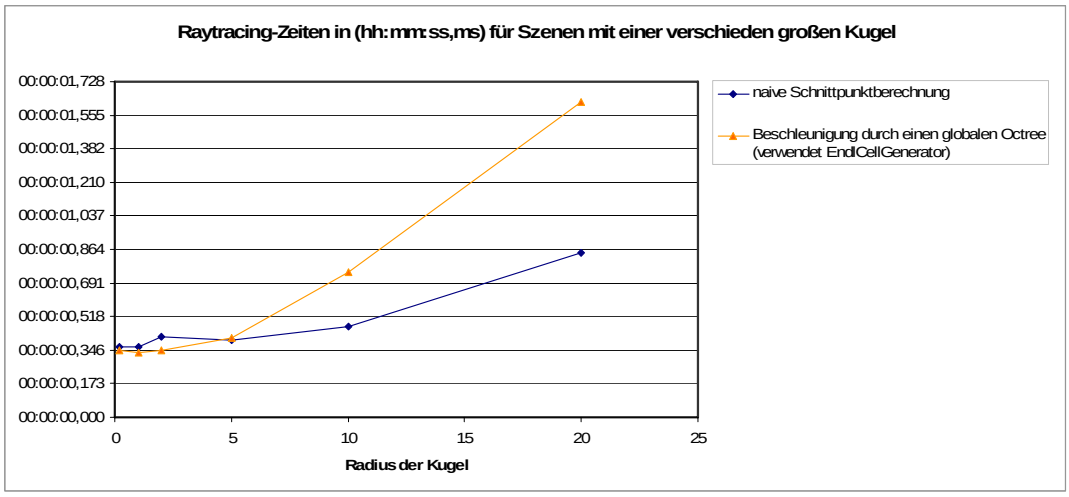


Abbildung 25 - Laufzeitverhalten für Szenen mit einer verschieden großen Kugel

In Abbildung 25 ist das Laufzeitverhalten der Testreihe für die optimierte und die nicht-optimierte Variante dargestellt. Es ist zu erkennen, dass die optimierte Variante für größere Kugeln immer ungünstiger wird. Da die Kugel mit dem Radius 20 schon die gesamte Bildfläche bedeckt, stellt sie für die Testreihe den worst case dar. Für diesen schlechtesten Fall ist die optimierte Variante, begründet durch einen höheren Overhead, 750 ms langsamer. Verglichen mit der großen Zeitersparnis in komplexeren Szenen ist dieser Nachteil akzeptabel.

4.6. Raytracer

Die Klasse `Raytracer` ist die zentrale Klasse des entwickelten Raytracers. Über sie lassen sich Einstellungen setzen, und der eigentliche Rendering-Prozess kann durch Methoden dieser Klasse angestoßen werden.
Eine Einstellungsmöglichkeit, auf die noch nicht näher eingegangen wurde, ist das Definieren der Priorität des Rendering-Prozesses. Es gibt drei mögliche Werte, die die Priorität annehmen kann. Wird sie als hoch definiert, dann kann der Thread, in dem das Rendering ausgeführt wird, am Anfang jeder Bildzeile vom Benutzer unterbrochen werden. Die restliche Zeit nimmt er alle ihm zur Verfügung stehenden Ressourcen in Anspruch. Bei einer mittleren Priorität gibt der Thread zusätzlich vor jeder Berechnung eines Bildpunktes dem System die Chance, ausstehende Aufgaben abzuarbeiten. Ist die Priorität auf niedrig gesetzt, schläft der Thread am Anfang jeder Bildzeile 100 Millisekunden, wodurch die CPU relativ gleichmäßig entlastet wird.

4.7. Optimierung der Performance

Für die in dieser Arbeit implementierten Raytracing-Methoden machen die Schnittpunktberechnungen einen Großteil der Gesamtrechenzeit eines Bildes aus. In Abbildung 26 ist eine Klassifikation aus [3] gegeben, in der hierarchisch alle Möglichkeiten einer Beschleunigung für Raytracing-Verfahren aufgezeigt werden. Wie in der Abbildung zu sehen ist, gibt es zwei grundsätzliche Ansätze, um die Rechenzeit der Schnittpunktberechnungen zu verkürzen. Zum einen kann man versuchen, die Strahl-Objekt-Berechnungen selbst immer weiter zu optimieren und zu beschleunigen. Zum anderen kann man die Anzahl einzelner Strahl-Objekt-Berechnungen reduzieren.

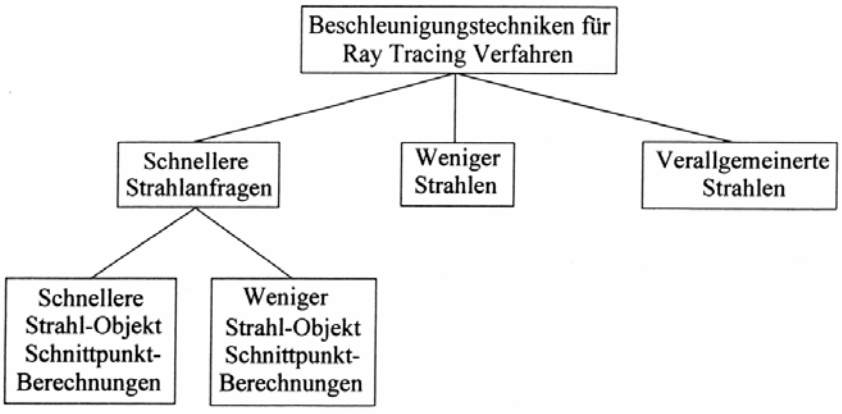


Abbildung 26 - Klassifikation von Beschleunigungstechniken, aus [3]

Im Folgenden soll nun darauf eingegangen werden, welche Möglichkeiten es gibt, die Berechnungen selbst zu optimieren. Es soll also nur auf die erste der beiden Varianten näher eingegangen werden. Eine Methode zur Reduzierung der Berechnungen wurde bereits im Abschnitt *4.5.2 Optimierung durch einen Octree* näher vorgestellt.

4.7.1. Speicherverwaltung

Neben den verwendeten Algorithmen und Gleichungen für die Strahl-Objekt-Berechnungen, auf die im Abschnitt *4.5.1 Schnittpunktberechnungen der Grundobjekte* bereits näher eingegangen wurde, ist eine weitere Optimierung im Zusammenhang mit der Speicherverwaltung sinnvoll.
Allgemein sind Methoden, die pro Pixel mehrmals aufgerufen werden, extrem zeitkritisch. Selbst Methoden mit nur einem Aufruf pro Pixel, werden für ein 640 mal 480 Pixel großes Bild schon 307.200-mal aufgerufen, was auch kleinste Optimierungen als sinnvoll erscheinen lässt. Die im Folgenden beschriebenen allgemeinen Optimierungsmöglichkeiten der Speicherverwaltung wurden für alle Methoden umgesetzt, die im Raytracer besonders oft aufgerufen werden, und sind nicht nur auf die Strahl-Objekt-Berechnungen beschränkt. Es ist vorstellbar, sie für beliebige zeitkritische Applikationen anzuwenden.
Um unnötige Speicherallozierungen und –freigaben zu vermeiden, kann man innerhalb von zeitkritischen Methoden, die besonders oft ausgeführt werden müssen, die lokalen Variablen entfernen. Grundsätzlich gibt es drei Möglichkeiten, lokale Variablen durch andere programmiertechnische Konzepte zu ersetzen. Diese sollen im Weiteren vorgestellt werden.

Private Attribute statt lokaler Variablen

Die einfachste Möglichkeit, lokale Variablen in Methoden zu vermeiden, besteht darin, alle in einer Methode benötigten lokalen Variablen außerhalb der Methode als private Klassenattribute zu definieren. Der Speicherplatz der Attribute muss nur einmal beim Instanziieren der Klasse erzeugt und beim Zerstören dieser Instanz wieder freigegeben werden. Dagegen wird der Speicherplatz für lokale Variablen bei jedem Methodenaufruf erzeugt und wieder freigegeben.
Diese Optimierung ist aber nur möglich, falls die Methode nicht rekursiv oder nebenläufig verwendet wird. Sollte es sich um eine statische Methode handeln, muss das private Attribut auch als statisch definiert werden.

Parameter statt lokaler Variablen

Eine andere Möglichkeit, lokale Variablen zu vermeiden, bieten Parameter. Alle in der Methode benötigten lokalen Variablen werden in einer Hilfklasse gekapselt. Diese Hilfsklasse wird außerhalb der Methode vor der Ausführung erstellt und als Parameter übergeben. Im entwickelten Raytracer wurden derartige Hilfsklassen als öffentliche innere Klassen der jeweiligen Methodenklasse implementiert.
Auch diese zweite Variante ist für rekursive Methoden nicht geeignet, aber sie kann dazu verwendet werden, lokale Berechnungen einer Methode einer anderen Methode zur Verfügung zu stellen. Im folgenden Beispiel wird ein Aufruf der statischen Methode `getSphere_T` der Klasse `Intersections` gezeigt, wobei für diese Methode auch die Eingabe- und Rückgabeparameter in einem Hilfsobjekt gekapselt werden:

```
// allocate input
Intersections.SphereInput sphere_input = new Intersections.SphereInput();
// set input data
Sphere_input.ray.getOrigin().set(10.0f,10.0f,10.0f);
Sphere_input.ray.getDirection().set(0.0f,0.0f,1.0f);
Sphere_input.radius = 5.0f;
// allocate output
Intersections.ObjectOutput sphere_output = new Intersections.ObjectOutput();
// allocate locals
Intersections.SphereLocalVariables method_locals =
    new Intersections.SphereLocalVariables();
// call method
Intersections.getSphere_T(sphere_input,sphere_output,method_locals);
```

Obwohl es einen Mehraufwand bedeutet, Methoden in dieser Weise aufzurufen, und der Code auch nicht mehr so gut lesbar ist, hat es sich gezeigt, dass ein solches Verfahren große Geschwindigkeitsvorteile bringen kann.

Eigene Speicherverwaltungen für lokale Variablen

Um auch für rekursive Methoden eine Optimierung der Speicherverwaltung zu erhalten, kann man eine eigene Speicherverwaltung implementieren, die auf Grund von speziellen Voraussetzungen bei der Speicherallozierung und –freigabe einfach zu entwickeln ist und Geschwindigkeitsvorteile gegenüber der allgemeinen Speicherverwaltung von Java bringen kann.
Man vermeidet bei dieser Variante nicht die lokalen Variablen, sondern optimiert ihre Verwaltung, wodurch neue Instanzierungen von Klassen verhindert werden. Um die automatische Garbage Collection von Java zu umgehen, müssen alle lokalen Instanzen speziell instanziiert und auch wieder freigegeben werden. Wenn vorausgesetzt wird, dass eine zuletzt erzeugte Instanz auch zuerst wieder freigegeben wird (und diese Voraussetzung ist für rekursive Methoden leicht zu erfüllen), kann man eine Speicherverwaltung verwenden, welche eine sehr einfache Freispeicherverwaltung besitzt. Initial wird ein Array mit n Instanzen einer bestimmten Klasse erzeugt. Das Allozieren und Freigeben funktioniert ähnlich einem Stack. Wird eine neue Instanz der Klasse benötigt, kann das Element über der Spitze zurückgegeben werden und die Spitze wird inkrementiert. Nach der Freigabe eine Instanz wird die Spitze dekrementiert. Bei einem Stack-Overflow wird seine Kapazität einfach verdoppelt. Eine solche Speicherverwaltung ist in der Klasse `MemoryPool` speziell für Instanzen der Klasse `Ray`, `Vector3f`, `Color3f` und `Color4f` implementiert.
Mit Hilfe dieser einfachen Speicherverwaltung lassen sich unnötige Instanzierungen und Freigaben innerhalb von rekursiven Methoden vermeiden, und für größere Instanzen kann ein solches Verfahren auch deutliche Geschwindigkeitsvorteile bringen.

4.7.2. Caching

Da letzten Endes keine allgemeine Caching-Strategie für den implementierten Raytracer verwendet wurde, soll im folgenden Abschnitt genau diese Entscheidung erläutert und begründet werden.
Da es eine Reihe von Methoden gibt, die mit gleichen Eingangsparametern öfter aufgerufen werden und auch selbe Ergebnisse liefern, wurde schon sehr früh ein allgemeiner Ansatz zum Cachen dieser Ergebnisse entwickelt. Dieser Ansatz sah transparent gestaltete cachende Hüllen für die benötigten Klassen vor. Die Benutzung dieser Hüllenklassen sollte ähnlich der in Java eingehüllten Sammlungen sein. In Java würde z. B. eine synchronisierende Hülle für eine Liste durch folgenden Code erstellt werden:

```
List my_synchronized_list = Collections.synchronizedList(my_list);
```

Der Code für eine cachende Hülle für ein Kameraobjekt sähe dementsprechend so aus:

```
RTCamera my_cached_camera = Caching.cachedCamera(my_camera);
```

Die Implementierung einer cachenden Hülle wurde über Hash-Tabellen gelöst, welche zu einem gegebenen Parametervektor ein Ergebnis speichern konnten.
Obwohl ein allgemeiner Ansatz eine aus softwaretechnischer Sicht sehr saubere Lösung darstellt, war er für den implementierten Raytracer nicht zu verwenden, da der Zugriff auf die gehashten Ergebnisse zu lange dauert. Es brachte also nur sehr wenige bis gar keine Geschwindigkeitsvorteile, da die Berechnung der

Ergebnisse im Verhältnis zum Auslesen der gecachten Werte nicht lange genug dauert und der Overhead für eine allgemeine Caching-Strategie zu hoch ist.
In der endgültigen Version des Raytracers wurde aus den oben genannten Gründen auf ein allgemeines Caching verzichtet, obwohl dieses zwischenzeitlich implementiert wurde. Da ein Caching in einigen Fällen trotzdem sinnvoll ist, wurden für diese Fälle eigene Caching-Strategien entwickelt. Ein Beispiel für einen speziellen Caching-Ansatz wurde bereits im Abschnitt *4.2.2 Adaptives Supersampling* beschrieben.

5. Verwendung des Raytracers als Frameworks

Da der entwickelte Raytracer als Framework konzipiert wurde, soll in diesem Kapitel nun auf die Schnittstellen und das Einbinden des Frameworks näher eingegangen werden.
Es soll an dieser Stelle darauf hingewiesen werden, dass es zurzeit keine Standard-Shader-Implementierungen innerhalb des Frameworks gibt. Da es in der Software GroIMP bereits speziell implementierte Shader gab, war es im Rahmen dieser Arbeit nicht notwendig, die Shader zu reimplementieren. Für eine spätere allgemeine Verwendung des Frameworks könnte dies allerdings sinnvoll sein.

5.1. Schnittstellen

Die zur Einbindung des Frameworks benötigten Schnittstellen befinden sich hauptsächlich im Package `de.grogra.ray`. Lediglich die für die Shader benötigten Schnittstellen befinden sich im Package `de.grogra.ray.shader`.
Um den Raytracer auf die in der konkreten Applikation verwendeten Implementierungen der Szene mit den zugehörigen 3D-Objekten und Lichtern anzupassen, müssen die Interfaces `RTScene`, `RTObject` und `RTLight` für alle 3D-Komponenten implementiert werden. Objekte, die keine wirklichen 3D-Komponenten darstellen, sondern eher theoretischer Natur sind, wie zum Beispiel ein Sky-Objekt, werden durch das Interface `RTFakeObject` implementiert. Außerdem muss jede Kamera das Interface `RTCamera` implementieren. Die Klasse `Intersections` bietet für die Schnittpunktberechnungen in den Implementierungen der einzelnen 3D-Objekte statische Methoden an, die zur Vereinfachung genutzt werden können.
Zur Beschreibung der Materialeigenschaften müssen die Interfaces RTShader und RTMedium implementiert werden. Die `RTShader`-Implementierungen beschreiben hierbei die Eigenschaften der Objektoberfläche, wogegen die Implementierungen des Interfaces `RTMedium` das Innere von soliden 3D-Objekten definieren. Besonders bei den Shadern sind die Schnittstellen stark an Methoden bereits existierender Klassen von GroIMP angelehnt, um den Overhead der Adapter-Interfaces für die Einbindung in GroIMP möglichst gering zu halten.
Um den Fortschritt der Berechnung des Raytracers in der GUI der benutzenden Applikation anzuzeigen, kann das Listener-Interface `RTProgressListener` implementiert werden und wird dann der Klasse `Raytracer` mit der Methode `addProgressListener` übergeben.

5.2. Einbindung in GroIMP

Das Einbinden des Raytracers in GroIMP kann als Beispiel für eine mögliche Einbindung des Frameworks gesehen werden. Obwohl das Framework speziell für diese Software entwickelt wurde, ist es denkbar, den Raytracer durch eine ähnliche Einbindung auch anderen Applikationen zur Verfügung zu stellen.
Alle für die Einbindung des Raytracers in GroIMP benötigten Schnittstellen-Implementierungen und Klassen befinden sich im GroIMP-Projekt *IMP-3D* im Package `de.grogra.imp3d.ray`.
Durch den Import des Raytracer-Plugins wird in GroIMP mittels des Menüpunktes *Render View→Build-In Raytracer* die Methode `run` der Klasse `de.grogra.im3d.ray.Renderer` gestartet. Diese Methode generiert initial einen Graphen mit Objekten, die durch GroIMP-Objekte erzeugt werden und das Interface `Raytraceable` implementieren. Diese GroIMP-Objekte generieren für jede 3D-Komponente, die vom Raytracer unterstützt wird, eine spezielle `RTObject`- oder `RTLight`-Implementierung. Alle aktuellen Parameter und

Eigenschaften der 3D-Objekte werden an dieser Stelle übergeben. Der erzeugte Graph enthält neben den `RTObject`-/`RTLight`-Instanzen als Blätter auch noch innere Transformationsknoten. In einem zweiten Schritt werden für jedes 3D-Objekt die Transformationsknoten, die im Graphen auf dem Weg von der Wurzel zum Objekt liegen, zu einer Transformationsmatrix zusammengefasst und dem jeweiligen Objekt übergeben. Für die Klasse `GroIMPSceneGraph`, die das Interface `RTScene` implementiert, ist nun das Aufzählen aller `RTObject`- und `RTLight`-Objekte durch Traversieren des erzeugten Graphen leicht möglich.
Nach der Erzeugung eines speziellen Szenegraphen wird die Klasse `GroIMPRaytracer` in einem neuen Thread gestartet. In dieser Klasse wird ein `Raytracer`-Objekt erzeugt und mit den aktuellen Einstellungen konfiguriert. Außerdem wird eine Listener-Instanz erzeugt (die Klasse `GroIMPRTProgressListener`, die das Interface `RTProgressListener` implementiert), die der `Raytracer`-Instanz übergeben wird. Diese Klasse `GroIMPRTProgressListener` kann mit Hilfe einer übergebenen `Workbench`-Instanz den Statustext, einen Fortschrittsbalken sowie das Bild im View Fenster aktualisieren.
Nach dem Konfigurieren der `Raytracer`-Instanz wird durch die Methode `renderScene` der Rendering-Prozess angestoßen.

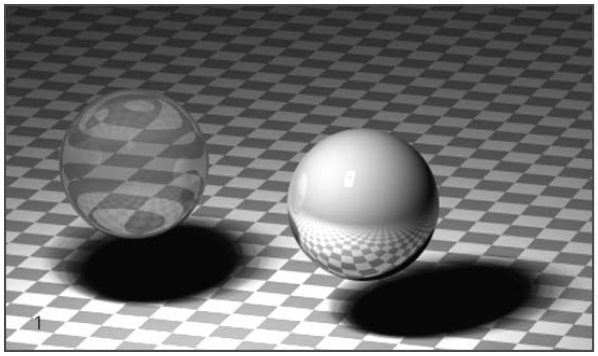
6. Zusammenfassung

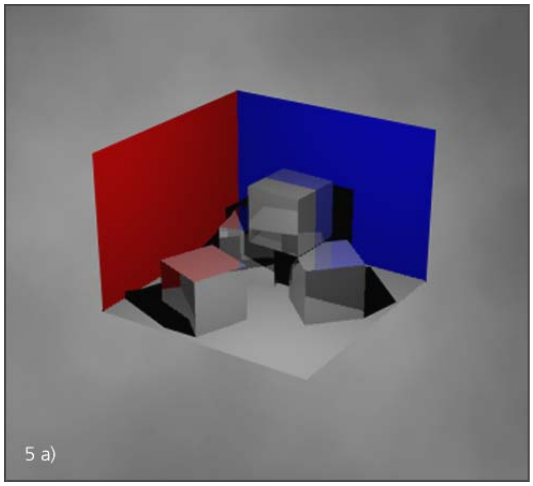
Dieses Kapitel soll eine Zusammenfassung der vorliegenden Studienarbeit bieten, in der ein Raytracer für die Software GroIMP entwickelt wurde. Folgende GroIMP-3D-Objekte werden vom Raytracer unterstützt: Kugel, Quader, Zylinder, Kegel, Kegelstumpf, Parallelogramm und Ebene. Alle Objekte können sowohl opak als auch transparent gerendert werden, wobei für transparente Objekte Lichtbrechungen unterstützt werden. Im Anhang sind in der ersten Abbildung zwei Kugeln dargestellt, wobei die linke Kugel transparent und die rechte Kugel opak gerendert wurde. Die zweite Abbildung zeigt zwei Parallelogramme, die mit einer teilweise transparenten Textur überzogen wurden und entsprechende Schatten werfen, womit aufgezeigt werden soll, dass sich halbtransparente Texturen auch auf die Form von Objekten auswirken können. Neben den 3D-Objekten unterstützt der Raytracer alle in GroIMP vorhandenen Lichtquellen (Punktlicht, Spotlicht, gerichtetes Licht und ausgedehnte Lichtquelle) sowie das Sky-Objekt. Die dritte Abbildung im Anhang zeigt einen Baum, der von einer ausgedehnten Lichtquelle bestrahlt wird und daher diffuse Schatten wirft.
Neben einem sehr einfachen Raytracing-Algorithmus, in dem inverse Lichtstrahlen vom Kamerazentrum aus durch die Szene verfolgt und an Objekten ideal reflektiert und gebrochen werden, wurde ein physikalisch korrekterer Algorithmus implementiert, welcher auch nicht-ideale Spiegelungen und Brechungen mit einbezieht. In der vierten Abbildung wird an fünf Kugeln gezeigt, welchen Einfluss die Materialeigenschaft Shininess des Phong-Materials auf die Spiegelung des Sky-Objektes bei der Verwendung des zweiten Algorithmus, dem Pathtracer, hat. Je kleiner der Wert für die Shininess ist, desto diffuser wird der Himmel in den Kugeln gespiegelt. Die Abbildungen fünf bis neun sollen in einem direkten Vergleich die Ausgaben der beiden Raytracing-Algorithmen aufzeigen. In der jeweils linken Abbildung wurde die Szene mit dem Standard-Raytracer, in der rechten mit dem Pathtracer gerendert. Die fünfte Abbildung zeigt, dass beim Standard-Raytracer das diffuse Material der Wände das auftreffende Licht des Himmels absorbiert, weshalb das gesamte Bild dunkler ist. Außerdem ist für den Pathtracer gut zu erkennen, dass die rote und blaue Wand den weißen Boden indirekt beleuchten. Abbildung sechs zeigt die Ergebnisse beim Verwenden einer emissiven Textur zum Erstellen selbstleuchtender Objekte. Es wurde keine weitere Lichtquelle in dieser Szene erzeugt. Die siebente Bildfolge stellt eine Szene dar, in der eine Fläche mit einem Spotlicht bestrahlt wird und die gesamte Szene indirekt beleuchtet. In der rechten Abbildung ist zu sehen, dass der Pathtracer für indirektes Licht sogar Kaustik-Effekte erzeugen kann. Bei der Verwendung des Standard-Raytracers wird fast das gesamte indirekte Licht ignoriert. Die achte und neunte Abbildung zeigen abschließend zwei Beispielszenen der GroIMP-Anwendung.
Da die Erzeugung der Bilder rechenaufwendig und damit auch zeitaufwendig ist, wurde viel Aufwand in die Optimierung des Raytracers gesteckt, und es wurden verschiedene Ansätze dazu verwendet. In einem Ansatz, der davon ausgeht möglichst wenige Schnittpunkte von Lichtstrahlen mit Objekten berechnen zu müssen, werden durch einen Octree Hilfsobjekte in die Szene eingefügt, durch die sich die Anzahl der Berechnungen minimieren lässt. Ein anderer Ansatz geht davon aus, die Berechnungen selbst zu beschleunigen. Hierfür wurde für zeitkritische Funktionen eine eigene Speicherverwaltung eingeführt. Außerdem wurden für einige Algorithmen spezielle Caching-Mechanismen entwickelt, um diese zu beschleunigen.
Um die Qualität der Ausgabebilder zu verbessern, unterstützt der Raytracer zwei Antialiasing-Methoden, stochastisches und adaptives Antialiasing.

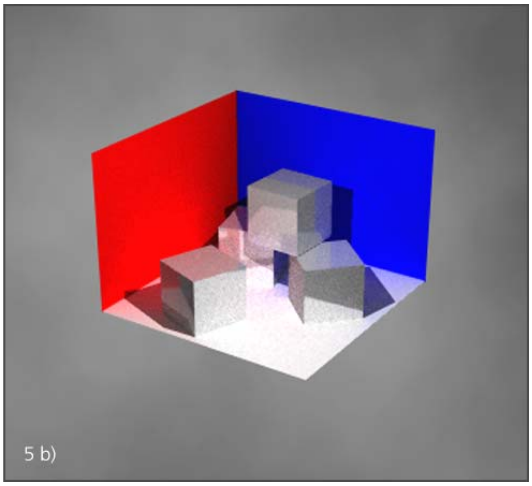
7. Ausblicke

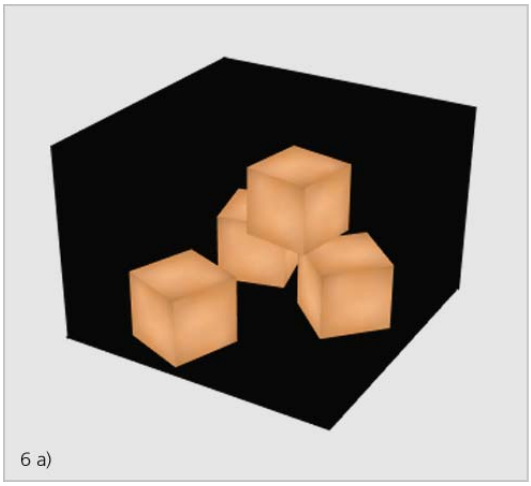
Im letzten Kapitel soll nun näher auf weiterführende Arbeiten eingegangen werden. Eine mögliche Erweiterung und Vervollständigung des Raytracers wäre die Unterstützung weitere 3D-Objekte, wie zum Beispiel NURBS, Patches und Polygone. Hierfür müsste die Klasse `Intersections` um Methoden zur Schnittpunkt- und Parameterberechnung erweitert werden. Zusätzlich müssten alle entsprechenden GroIMP-Klassen der 3D-Objekte das Interface `Raytraceable` realisieren und zugehörige Implementierungen des Interfaces `RTObject` erstellt werden.
Eine weitere Richtung zukünftiger Arbeiten könnte darin bestehen, andere Modelle für die globale Beleuchtung zu nutzen. Das bidirektionale Pathtracing würde sich anbieten, um Kaustik-Effekte auch für direkte Beleuchtungen zu unterstützen. Weitere mögliche Algorithmen wären zum Beispiel Photon Mapping und Radiosity. Diese Richtung wird bereits in laufenden Arbeiten verfolgt.
Die Entwicklung eines realistischen Kameramodells wäre sinnvoll, um Tiefenunschärfe zu ermöglichen. Hierfür müsste eine Implementierung des Interfaces `Antialiasing` erstellt werden, wobei der Name des Interfaces dann nicht mehr sinnvoll wäre und ersetzt werden sollte.

8. Anhang

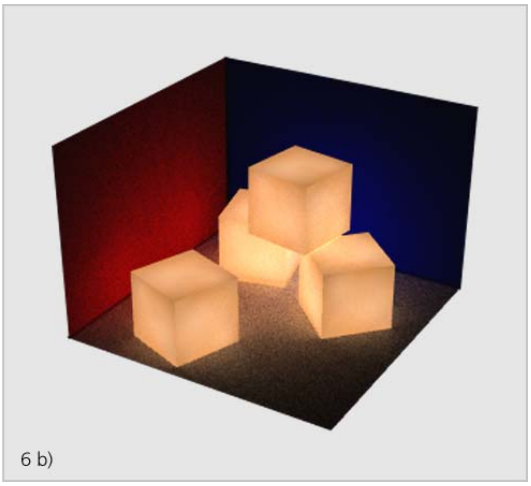




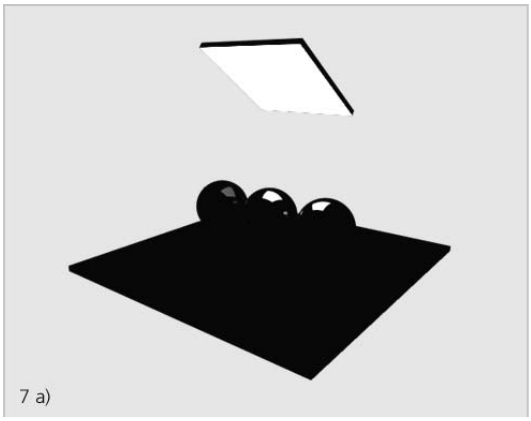




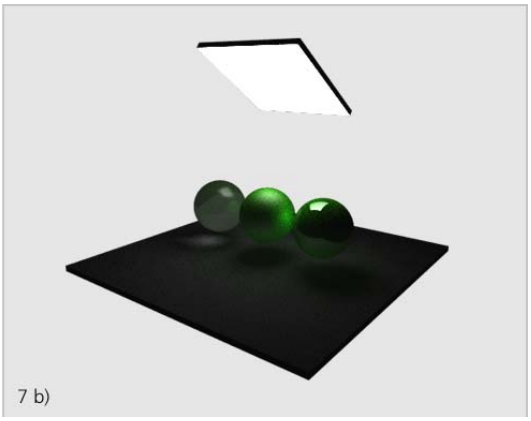
6 a)



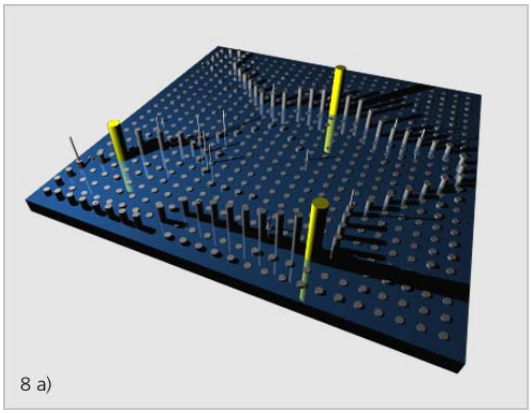
6 b)



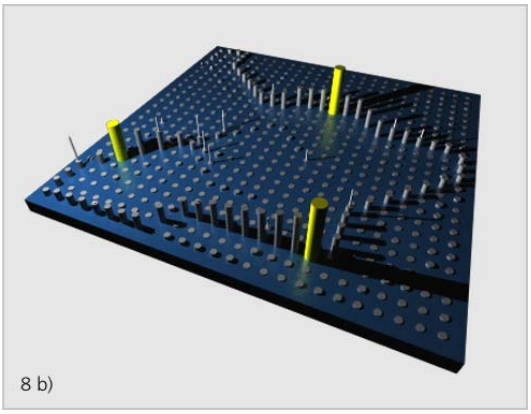
7 a)



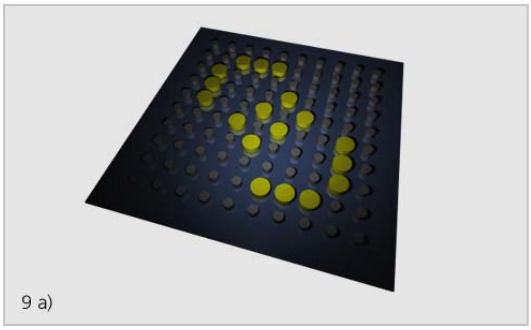
7 b)



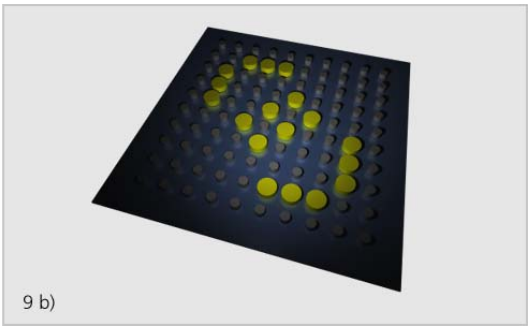
8 a)



8 b)



9 a)



9 b)

9. Abbildungsverzeichnis

10. Literatur

[1] Alan Watt (2002). *3D-Computergrafik*. Pearson Studium, München.

[2] Andrew S. Glassner (1989). *An Introduction to Ray Tracing*. Acad. Press, London.

[3] Robert Endl (1995). *Octree-Strahlengeneratoren*. Würfel-Verlag, Biebertal.

[4] Christopher D. Watkins, Stephen B. Coy, Mark Finlay (1993). *Fotorealismus und Ray Tracing in C*. Heise, Hannover.

[5] Ken Arnold, James Gosling, David Holmes (2001). *Die Programmiersprache Java*. Addison-Wesley, Bonn.

[6] http://de.wikipedia.org/wiki/Raytracing (27.09.2006).

[7] http://de.wikipedia.org/wiki/BRDF (27.09.2006).

[8] http://de.wikipedia.org/wiki/BTDF (27.09.2006).

[9] http://www.dlib.org/dlib/february02/goesele/02goesele.html (27.09.2006).