Raytracing und Szenengraphen

Diplomarbeit von

Björn Schmidt
(bs@iz-media.de)

eingereicht bei

Prof. Dr. Detlef Krömker
Professur Grafische Datenverarbeitung

betreut von

Dr. Tobias Breiner

Fachbereich Informatik und Mathematik
Johann Wolfgang Goethe-Universität Frankfurt am Main

6. November 2006

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Diplomarbeit ohne unzulässige Hilfe und nur unter Verwendung der angegebenen Literatur angefertigt wurde.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Frankfurt am Main, den 6. November 2006

Björn Schmidt

Zusammenfassung

Raytracing ist ein bekanntes Verfahren zur Erzeugung fotorealistischer Bilder. Globale Beleuchtungseffekte einer 3D-Szene werden durch das Raytracing-Verfahren physikalisch korrekt dargestellt. Erst aktuelle Forschungsarbeiten ermöglichen es, das sehr rechenintensive Verfahren bei interaktiven Bildraten in Echtzeit zu berechnen.

Komplexe 3D-Szenen, wie sie beispielsweise in 3D-Spielen oder Simulationen vorkommen, können durch einen *Szenengraphen* modelliert und animiert werden. Damit die Rendering-Ergebnisse eines Szenengraphen näher an einem realen Bild liegen, ist es erforderlich das Raytracing-Verfahren in einen Szenengraphen einzugliedern.

In dieser Arbeit werden die Möglichkeiten zur Integration eines *Echtzeit-Raytracers* in eine Szenengraph-API untersucht. Ziel dieser Diplomarbeit ist die Darstellung dynamischer Szenen bei interaktiven Bildraten unter Verwendung des Raytracing-Verfahrens auf einem herkömmlichen PC. Zunächst müssen bestehende Open Source Szenengraph-APIs und aktuelle Echtzeit-Raytracer auf ihre Eignung zur Integration hin überprüft werden.

Bei der Verarbeitung dynamischer Szenen spielt die verwendete Beschleunigungsdatenstruktur des Raytracers eine entscheidende Rolle. Da eine komplette Neuerstellung der Datenstruktur in jedem Bild zuviel Zeit in Anspruch nimmt, ist eine schnelle und kostengünstige Aktualisierung erforderlich. Die in [LAM01] vorgestellte Lösung, eine *Hüllkörperhierarchie (BVH)* als Beschleunigungsdatenstruktur zu verwenden, fügt sich sehr gut in das Konzept eines Szenengraphen ein. Dadurch wird eine einfache Aktualisierung ermöglicht.

Um das Ziel dieser Arbeit zu erreichen, ist es notwendig, die Parallelisierbarkeit des Raytracing-Verfahrens auszunutzen. Purcell zeigt in [Pur04], dass *Grafikprozessoren (GPUs)* neben ihrer eigentlichen Aufgabe auch für allgemeine, parallele Berechnungen wie das Raytracing verwendet werden können.

Die in bisherigen Arbeiten über GPU-basiertes Raytracing entwickelten Systeme können dynamische Szenen nicht bei interaktiven Bildraten darstellen. Aus diesem Grund wird in dieser Diplomarbeit ein neues System konzipiert und implementiert, das den in [TS05] entwickelten Raytracer erweitert und in die Open Source Szenengraph-API *OGRE 3D* integriert.

Das implementierte System ermöglicht die Darstellung statischer und dynamischer Szenen unter Verwendung einer Consumer-Grafikkarte bei interaktiven Bildraten. Durch seine Erweiterbarkeit bildet das System das Grundgerüst für ein Realtime-High-Quality-Rendering-System.

Abstract

Ray tracing is a well-known photorealistic rendering technique to display global illumination-effects in 3D scenes in a physically accurate way. Only by means of current research works it is possible to compute this expensive technique at interactive framerates in realtime.

Complex scenes as they appear e. g. in 3D games or simulations, can be modeled and animated using a *scenegraph*. To gain more photorealism in scenegraph renderings, it is necessary to integrate ray tracing into a scenegraph-API.

In this thesis possibilities for integrating a *realtime ray tracer* into a scenegraph-API are investigated. The main objective of this diploma-thesis is ray tracing dynamic scenes at interactive framerates on a consumer-PC. First of all existing open source scenegraph-APIs as well as latest realtime ray tracers need to be evaluated with regard to the suitability for this integration.

When rendering dynamic scenes, the acceleration structure of the ray tracer plays a decisive role. Due to the fact that reconstructing the acceleration structure every frame takes too much time a fast and inexpensive method for updating the structure is required. The solution using a *bounding-volume hierarchie (BVH)*, as presented in [LAM01], fits perfectly into the concept of a scenegraph and thus allows a fast and inexpensive method for updating the acceleration structure.

In order to achieve the main objective of this thesis it is essential to take advantage of the possibility of parallelising ray tracing. Purcell shows in [Pur04] that *Graphics Processing Units (GPUs)* can be used for parallel general purpose computations like ray tracing.

The systems developed in previous papers about GPU-based ray tracing are not able to render dynamic scenes at interactive framerates in realtime. Therefor a new system is designed and implemented in this thesis which extends the ray-tracer developed in [TS05] and integrates it into the scenegraph-API *OGRE 3D*.

The developed system is capable of ray tracing static and dynamic scenes using consumer-graphics-hardware at interactive framerates. By reason of its extensibility this system forms the basis of a Realtime-High-Quality-Rendering-System.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Ziel

Aktuelle Echtzeit-Computer-Grafik-Systeme erreichen durch die stetig wachsende Leistung von Prozessoren und Grafikkarten annähernd fotorealistische Ergebnisse bei sehr hohen Bildraten.

Durch die Verwendung von Szenengraphen können auch komplexe Szenen, wie sie beispielsweise in 3D-Spielen, Simulationen oder Anwendungen der virtuellen Realität zu finden sind, komfortabel modelliert, animiert und in Echtzeit dargestellt werden.

Das verwendete Rasterisierungsverfahren ist allerdings nicht in der Lage, auftretende globale Beleuchtungseffekte wie Lichtreflexionen zu berechnen und im physikalischen Sinne korrekt darzustellen. Diese Phänomene können mit Hilfe des Verfahrens bestenfalls nachgeahmt werden. Für eine korrekte Darstellung muss anstelle des Rasterisierungsverfahrens ein globales Beleuchtungsverfahren zum Rendern der Szene verwendet werden.

Eines der bekanntesten globalen Beleuchtungsverfahren ist das Raytracing, das zwar in der Lage ist, physikalisch korrekte, fotorealistische Bilder zu erzeugen, aber sehr rechenintensiv ist. Das Rendern von Szenen mit Hilfe des Raytracing-Verfahrens ist daher lange Zeit nicht in Echtzeit möglich gewesen. Erst die Ergebnisse aktueller Forschungsarbeiten erlauben die Darstellung in Echtzeit bei interaktiven Bildraten.

Diese Echtzeit-Raytracer unterstützen entweder nur das Laden vordefinierter, statischer Szenen aus Programmen wie beispielsweise Maya oder 3D Studio Max oder sie verwenden eine OpenGL-ähnliche Sprache zur Beschreibung statischer und dynamischer Szenen. Die einfache Modellierung und Animation von Szenen, wie sie ein Szenengraph bietet, wird nicht unterstützt.

Wünschenswert wäre es, auch durch einen Szenengraphen modellierte und animierte Szenen mit Hilfe des Raytracing-Verfahrens in Echtzeit darstellen zu können. Zur Zeit existiert jedoch keine Szenengraph-API, die dieses Verfahren unterstützt.

Ziel dieser Arbeit ist daher die Entwicklung eines Systems, das eine Open Source Szenengraph-API um einen Echtzeit-Raytracer erweitert. Dieses System soll dynamische Szenen bei interaktiven Bildraten darstellen können. Es soll erweiterbar sein und dadurch das Grundgerüst für ein Open Source Realtime-High-Quality-Rendering-System bilden.

1.2 Aufbau der Arbeit

In Kapitel 2 wird zunächst die Basis für die Implementierung und Integration eines Echtzeit-Raytracers in einen Szenengraphen geschaffen. Die Erläuterungen beginnen mit den grundlegenden Funktionen eines Szenengraphen, gefolgt von der Vorstellung vorhandener Szenengraph-APIs. In der anschließenden Kurzevaluierung werden die Stärken und Schwächen dieser Systeme herausgearbeitet.

Im Anschluss daran wird auf das Rasterisierungsverfahren eingegangen, das zur Zeit in Echtzeitanwendungen zum Rendern verwendet wird. Da das Rasterisierungsverfahren nicht alle Effekte der globalen Beleuchtung darstellen kann, wurden globale Beleuchtungsverfahren entwickelt, die in diesem Kapitel ebenfalls besprochen werden. Im Zuge dessen werden die Grundlagen des Raytracing-Verfahrens sowie die Datenstrukturen zur Beschleunigung vorgestellt. Den Abschluss des Kapitels bilden die für diese Arbeit notwendigen mathematischen Grundlagen.

Eine State-of-the-Art Analyse der relevanten, derzeit verfügbaren Verfahren für das Echtzeit-Raytracing wird in Kapitel 3 durchgeführt. Dafür werden zunächst die Grundlagen für allgemeine Berechnungen unter Verwendung der Grafikhardware (GPU) geschaffen. Im Anschluss daran findet eine Analyse aktueller Verfahren für CPU- und GPU-basiertes Echtzeit-Raytracing statt.

Die Anforderungsanalyse des zu implementierenden Systems wird in Kapitel 4 durchgeführt. Auf den Vergleich zwischen CPU- und GPU-basierten Raytracern, der zugunsten eines GPU-basierten Raytracers ausgeht, folgt eine ausführliche Begründung dieser Entscheidung. Daraufhin werden die aktuellen Verfahren für das GPU-basierte Raytracing analysiert und die Stärken und Schwächen dieser Systeme herausgearbeitet. Die Motivation zur Entwicklung eines neuen GPU-basierten Raytracers wird im Verlauf dieser Analyse begründet.

Da die verwendete Beschleunigungsdatenstruktur des Raytracers für das Raytracing dynamischer Szenen entscheidend ist, werden in diesem Kapitel verschiedene Beschleunigungsdatenstrukturen auf die Möglichkeit einer einfachen und schnellen Aktualisierung hin untersucht.

Abschließend wird geprüft, inwieweit aktuelle Szenengraph-APIs für die Integration eines GPU-basierten Echtzeit-Raytracers geeignet sind. Die Entscheidung wird zugunsten der Szenengraph-API *OGRE 3D* getroffen.

In Kapitel 5 wird ein neues Konzept für die Implementierung und Integration eines Echtzeit-Raytracers in die Szenengraph-API entwickelt. Hierzu werden zunächst die Grundlagen des zu implementierenden Systems erarbeitet. Nachfolgend wird die Verwaltung der Szenendaten, die Implementierung der Beschleunigungsdatenstruktur

sowie die Implementierung des eigentlichen Raytracing-Kerns innerhalb des entwickelten Systems vorgestellt. Die Beschreibung der implementierten Testapplikationen bildet den Abschluss des Kapitels.

Die im Rahmen dieser Arbeit erstellten Implementierungsarbeiten werden in Kapitel 5 evaluiert. Abschließend wird in Kapitel 6 das Ergebnis dieser Arbeit vorgestellt und ein Ausblick für die künftige Weiterentwicklung des implementierten Systems gegeben.

2 Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen dieser Arbeit geschaffen. Begonnen wird hierbei mit der Vorstellung der grundlegenden Konzepte eines Szenengraphen, gefolgt von einer Kurzevaluierung einer Auswahl der zur Zeit verfügbaren Szenengraphen, in der die Stärken und Schwächen der jeweiligen Systeme herausgearbeitet werden.

2.1 Grundlagen Szenengraphen

Bevor die Grundlagen eines Szenengraphen erarbeitet werden können, ist es notwendig, den Begriff der *3D-Szene* zu definieren.

Eine *3D-Szene* beinhaltet die geometrische Beschreibung der darzustellenden Objekte, Materialdaten der Objekte, wie Farbe und Textur, Beleuchtungsdaten und die Kameradaten des virtuellen Betrachters der 3D-Szene.

Zu Beginn der Echtzeit-Computergrafik waren Entwickler an Spezialhardware gebunden, die nur sehr eingeschränkte Möglichkeiten zur Programmierung boten. Erst die Einführung von *OpenGL*[1] 1992 durch *Silicon Graphics* erlaubte eine komfortable Programmierung der Visualisierungs-Pipeline.

OpenGL ermöglicht es Objekte mit Hilfe der Basisprimitive Punkt, Linie und Polygon zu definieren, deren Materialeigenschaften wie Textur, Farbe und Oberflächenbeschaffenheit zu verändern, sowie deren Transformationsmatrizen zu manipulieren und diese anschließend zu rendern. Das Erzeugen und Verändern von komplexen Szenen ist mit Hilfe dieser Low-Level-Befehle allerdings sehr aufwendig.

Ein *Szenengraph* schließt die Lücke zwischen komplexer Szenenbeschreibung und Rendering durch die Low-Level-Grafik-API wie *OpenGL* oder *Direct3D*. Er erlaubt die Definition, Verwaltung, Manipulation und Animation von komplexen hierarchischen Objekten und ist für das Rendering zuständig.

Aus graphentheoretischer Sicht ist ein Szenengraph ein Baum. Die Wurzel dieses Baums repräsentiert die Gesamtszene. Die Knoten dieser Wurzel können wiederum Knoten als Kinder haben oder ein oder mehrere Objekte der Szene beinhalten. Diese Topologie erlaubt die Spezifikation von Attributen, die sich auf Teilzweige des Graphen auswirken und so die hierarchische Modellierung und Animation komplexer Objekte ermöglichen.

[1] http://www.opengl.org

Wird die Szene gerendert, so wird der Baum von der Wurzel aus rekursiv traversiert. Zunächst wird an jedem Knoten überprüft, ob dieser und somit auch seine Objekte von der aktuellen Kameraposition aus sichtbar sind, respektive ob seine Bounding-Box eine Schnittfläche mit der Sichtpyramide bildet. Ist das der Fall, werden die im Knoten gespeicherten Transformationen auf die Objekte angewendet, und die Geometriedaten des Objekts an die zugrunde liegende Grafik-API übergeben.

Szenengraphen, die in aktuellen Spielen, wie beispielsweise *F.E.A.R.*, *Crisis* oder *Far Cry* oder in Anwendungen der virtuellen Realität eingesetzt werden, beinhalten oftmals auch Funktionen, die über das schiere hierarchische Objektmanagement für das Rendering hinausgehen, so zum Beispiel die Kollisionserkennung, Spielelogik, Physiksimulation, künstliche Intelligenz, etc.

2.2 Kurzevaluierung existierender Szenengraphen

Bei der Kurzevaluierung einer Auswahl aktueller Szenengraphen liegt das Hauptaugenmerk auf deren Erweiterbarkeit im Hinblick auf die Integration eines Raytracers. Die Auswahl erfolgte nach folgenden Kriterien:

- Open Source (GPL oder LGPL)
- Realisierung in C++
- Objektorientierte Implementierung
- „State of the art“
- Verwendung in aktuellen Projekten
- Rege Online-Community und gute Dokumentation

Aufgrund dieser Kriterien scheiden kommerzielle Szenengraphen wie *Gamembryo*, *Gizmo3D*, *Realimation*, *Renderware SG*, *VirTools*, *VisKit*, *OpenGL Performer* und *NVIDIA SG* aus. Nicht in C++ realisierte Szenengraphen wie *Java 3D* oder Szenengraphen, die nicht dem Stand der Technik entsprechen, werden ebenfalls außen vor gelassen.

2.2.1 OpenSceneGraph

Der Szenengraph *OpenSceneGraph*[2] wird aktuell hauptsächlich in wissenschaftlichen Simulationen, Visualisierungen und Anwendungen im Bereich virtuelle Realität (VR)

[2]Projekthomepage: http://www.openscenegraph.org

eingesetzt und bietet sowohl einen OpenGL-Renderer als auch einen Software-Renderer. Der Szenengraph ist auf den Plattformen *Windows*, *Mac OS X*, *Linux*, *IRIX*, *Solaris* und *FreeBSD* verfügbar.

Die Funktionalität des Szenengraphen kann durch sogenannte „Nodekits“ erweitert werden, die Erweiterungen beschränken sich hierbei allerdings auf die in den Knoten des Szenengraphen gekapselten Funktionalitäten wie Partikelsysteme, Terrain-Rendering, etc. Die Basis von *OpenSceneGraph*, wozu auch der eigentliche Renderer zählt, ist monolithisch aufgebaut und kann nicht durch Plugins oder Ähnliches erweitert werden.

Der Szenengraph wurde für die Realisierung des Projekts *Fraktale Erzeugung von gothischen Kathedralen* [LS05] verwendet. Zwar verlief die Realisierung des Projekts mit Hilfe von OpenSceneGraph reibungslos, bei der Portierung des Projektes auf die Plattformen Linux und Mac OS X musste allerdings festgestellt werden, dass die Renderergebnisse der einzelnen Plattformen deutlich voneinander abwichen.

2.2.2 OpenSG

Der von Dirk Reiners [RVB02] entwickelte Szenengraph *OpenSG*[3] verwendet einen OpenGL-Renderer mit Unterstützung der Shader-Sprache *HLSL* und ist auf den Plattformen *Windows*, *Linux*, *Irix* und *Mac OS X* verfügbar. *OpenSG* wurde für den Einsatz in VR-Anwendungen konzipiert. Es basiert auf dem VRML-Konzept und bietet als einziger der hier evaluierten Szenengraphen eine Clustersteuerung für verteiltes Rendern und das Ansteuern von VR-Ausgabegeräten wie *CAVE*[4] oder *HEyeWall*[5].

Erweiterungen können in OpenSG ebenfalls in Form von „Nodekits“ implementiert werden. Die monolithische Basis des Szenengraphen ist nicht erweiterbar.

2.2.3 OGRE 3D

Der Szenengraph *OGRE 3D*[6] ist in der Lage, sowohl *DirectX* als auch *OpenGL* als Rendering-API zu verwenden. Er unterstützt die relevanten Shader-Sprachen *GLSL*, *HLSL* und *Cg* und ist durchgehend modular aufgebaut. Zudem lässt sich seine Funktionalität komfortabel durch Plugins erweitern.

[3] Projekthomepage: http://www.opensg.org
[4] http://www.igd.fhg.de/igd-a4/projects45.html.de
[5] http://www.heyewall.de
[6] Projekthomepage: http://www.ogre3d.org

OGRE 3D zeichnet sich zum einen durch die Funktionsvielfalt, zum anderen durch die Benutzerfreundlichkeit der API aus. Wie in [NSL06] und [Jun06] beschrieben, ermöglicht es diese, schnell und einfach professionelle Ergebnisse zu erzielen. Der Szenengraph wird zur Zeit in zahlreichen öffentlichen Projekten eingesetzt. so wurde beispielsweise das kommerzielle Spiel *Ankh*[7] der deutschen Entwicklerfirma *Deck13* mit Hilfe von *OGRE 3D* realisiert.

OGRE 3D ist auf den Plattformen *Windows*, *Linux* und *Mac OS X* verfügbar.

2.2.4 Irrlicht

Irrlicht[8] bietet ebenfalls die Möglichkeit, sowohl DirectX als auch OpenGL als Rendering-API zu verwenden, unterstützt allerdings nur die Shader-Sprachen *HLSL* und *GLSL*, *Cg* ist nicht verfügbar. Wie die bereits erwähnten Szenengraphen ist auch Irrlicht auf den Plattformen *Windows*, *Linux* und *Mac OS X* verfügbar.

Dem Szenengraphen liegt ein monolithisches Konzept zugrunde, die Funktionalität kann nicht durch Plugins erweitert werden.

2.3 Rasterisierung

In diesem Abschnitt soll auf das Rasterisierungsverfahren näher eingegangen werden, das zur Ausgabe von 3D-Szenen auf einem Rastergerät, wie beispielsweise einem Bildschirm, mit Hilfe der Grafikhardware in aktuellen Echtzeitanwendungen verwendet wird.

Die Objekte einer 3D-Szene werden durch mathematische Primitive wie Polygone, Punkte und Linien beschrieben. Die Eckpunkte dieser Primitive sind die Eingabe für die sogenannte Renderingpipeline, die im Folgenden behandelt wird.

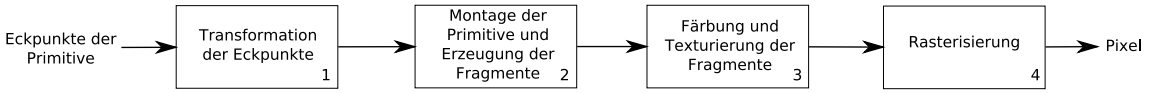

Abbildung 1: Grundlegender Aufbau einer Renderingpipeline

In Abbildung 1 ist der grundlegende Aufbau einer Renderingpipeline skizziert. In der ersten Stufe der Pipeline, Abb. 1 (1), werden die Eckpunkte zunächst in das Kamerakoordinatensystem transformiert. Die Montage der Primitive aus den transformierten

[7] http://www.ankh-game.de
[8] Projekthomepage: http://irrlicht.sourceforge.net

Eckpunkten erfolgt in der zweiten Stufe, Abb. 1 (2). Da nicht alle der so zusammengesetzten Polygone komplett sichtbar sind, müssen diese an der Sichtpyramide der Kamera „geclippt“ werden. Polygone, die völlig außerhalb der Sichtpyramide liegen, werden durch das sogenannte „Culling“ entfernt.

Für Primitive, die „Clipping“ und „Culling“ überstanden haben, muss festgestellt werden, welche Pixel durch das Primitiv überdeckt werden. Dazu wird mit Hilfe des *Scanline-Algorithmus* eine Menge von Fragmenten[9] und Pixel-Positionen generiert.

Die so erzeugten Fragmente der Primitive werden in der dritten Stufe der Pipeline, Abb. 1 (3), gefärbt und texturiert. Die Färbung wird nach dem lokalen Beleuchtungsmodell berechnet, das im späteren Verlauf dieses Kapitels behandelt wird. Vor der Aktualisierung des Bildspeichers wird schließlich in der vierten und letzten Stufe, Abb. 1 (4), für jedes Fragment ein Sichtbarkeitstest durchgeführt.

Zur Lösung dieses Verdeckungsproblems existieren zwei mögliche Verfahren: Der *Painter's Algorithmus* und der *Z-Buffer Algorithmus*.

Das Vorgehen des *Painter's Algorithmus* ist der Malerei entlehnt: Beim Erstellen eines Gemäldes wird mit den am weitesten entfernten Objekten begonnen, und sukzessive mit den näher am imaginären Betrachter liegenden Objekten fortgefahren. Übertragen auf eine 3D-Szene in der Computergrafik bedeutet dies, dass vor dem eigentlichen Rendern der Szene eine Sortierung der Polygone nach der Tiefeninformation erfolgen muss. Da die untere Schranke für das Sortieren von n Objekten $\mathrm{O}(n \log n)$ ist, ist der Painter's Algorithmus ineffizient.

Darüber hinaus hat der *Painter's Algorithmus* einen weiteren entscheidenden Nachteil: Überdecken sich mehrere Polygone auch nur teilweise, kann anhand der Tiefeninformation nicht entschieden werden, welches Polygon welches überdeckt. Man müsste sie unterteilen, um dieses Problem zu lösen.

Die logische Weiterentwicklung des *Painter's Algorithmus* ist der *Z-Buffer Algorithmus*, der ohne eine Sortierung der Polygone nach ihrer Tiefe auskommt. Der Bildspeicher wird im Zuge des Verfahrens um den sogenannten *Z-Buffer* erweitert, der die Tiefeninformation für jeden Pixel speichert. Bevor ein Fragment von der vierten Stufe der Pipeline in den Bildspeicher geschrieben werden kann, muss die Tiefeninformation mit derjenigen an dieser Stelle im Z-Buffer gespeicherten verglichen werden. Ist die Tiefe des Fragments größer, so wird dieses verworfen, da es durch ein anderes Fragment mit geringerer Tiefe überdeckt wird. Hat es hingegen eine geringere Tiefe,

[9] Pixel und Fragmente sind nicht identisch. Ein *Pixel* (= Picture Element) ist der Inhalt des Bildspeichers an einer bestimmten Position, ein *Fragment* hingegen ist ein Zustand, der benötigt wird, um den Inhalt des späteren Pixels zu bestimmen.

wird es in den Bildspeicher geschrieben, und darüber hinaus die Tiefeninformation an dieser Stelle im Z-Buffer mit der des Fragments überschrieben.

Im Gegensatz zum *Painter's Algorithmus* ist der *Z-Buffer Algorithmus* wesentlich effizienter und kann das Problem der Überdeckung mehrerer Polygone lösen. Z-Buffer und Renderingpipeline sind auf aktuellen Grafikchips auf Hardwareebene verfügbar. Diese aktuellen Generationen erreichen durch die parallele Verarbeitung von Eckpunkten und Fragmenten einen Durchsatz von mehreren Millionen Dreiecken in der Sekunde.

2.4 Globale Beleuchtungsverfahren

Das im vorherigen Abschnitt behandelte Rasterisierungsverfahren erzeugt schnell gute Ergebnisse in Echtzeit. Da lediglich ein lokales Beleuchtungsmodell herangezogen wird, das physikalische Phänomene der globalen Beleuchtung, wie Reflexion und Refraktion des Lichts, außer Acht lässt, reicht dieses Verfahren nicht aus, um fotorealistische Bilder zu erzeugen.

Blinn stellt in [Bli76] das „Reflection Mapping“ (auch unter dem Namen „Environment Mapping“ bekannt) vor, das es ermöglicht Spiegelungen darzustellen. Bei diesem Verfahren wird die Umgebung des spiegelnden Objekts auf eine Textur projiziert, die dann wiederum in das Innere eines Würfels oder einer Kugel projiziert wird, die das spiegelnde Objekt umgibt. Beim Rendern der Szenen wird die Umgebungstextur (Environment Map) abgetastet und das Objekt spiegelt somit seine Umgebung.

Das Verfahren hat zwei entscheidende Nachteile: Zum einen kommt es beim Abtasten der Environment Map zu Verzerrungen, die inkorrekte Spiegelungen zur Folge haben. Zum anderen kann nur ein spiegelndes Objekt in der Szene dargestellt werden, für mehrere dieser Art ist das Verfahren nicht anwendbar.

2.4.1 Raytracing

Das Raytracing-Verfahren beruht auf der Verfolgung von Strahlen, die vom Auge des virtuellen Betrachters einer 3D-Szene ausgesendet werden.

In der Computergrafik wurde dieses Verfahren (auch als „Ray Casting“ bezeichnet) zum ersten Mal von Appel in [App68] zur Lösung des Verdeckungsproblems vorgeschlagen. Hierfür wird für jeden Pixel ein Strahl erzeugt, dessen Richtung vom Augpunkt aus zum entsprechenden Pixel der Bildebene weist. Mit Hilfe geometrischer Verfahren wird für jedes Primitiv der Szene der eventuelle Schnittpunkt mit den Strahlen

ermittelt. Es sind genau diejenigen Primitive sichtbar, deren errechnete Schnittpunkte dem Betrachter am nächsten liegen.

Indem an jedem berechneten Schnittpunkt Strahlen zu den Lichtquellen der Szene (*Schattenstrahlen*) ausgesendet werden, ermöglicht es dieses Verfahren, die Schattenberechnung der Szenen durchzuführen. Trifft ein Schattenstrahl ein Objekt, liegt der Schnittpunkt im Schatten. Wird jedoch die Lichtquelle getroffen, so wird der betreffende Punkt beleuchtet.

Turner Whitted beschreibt in [Whi80] als erster einen Algorithmus, der Reflexionen und Refraktionen in einer Szene darstellen kann.

Bei seinem Verfahren werden, wie in Appels Ray Casting Verfahren, Strahlen vom Augpunkt des virtuellen Betrachters (*Primärstrahlen*) in die Szene entsendet und der Schnittpunkt dieser Strahlen mit den Primitiven berechnet. An diesen Schnittpunkten wird der lokale Anteil der Beleuchtung mit Hilfe eines lokalen Beleuchtungsmodells errechnet, das in Kapitel 2.8 ausführlicher behandelt wird.

Zur Berechnung des globalen Anteils der Beleuchtung der Szene werden an diesem Schnittpunkt weitere Strahlen (*Sekundärstrahlen*) erzeugt, deren Richtung vom Material des getroffenen Primitivs abhängt: Ist das Primitiv spiegelnd, so wird ein Strahl gemäß den Reflexionsgesetzen entsendet. Ist es außerdem teilweise transparent, so wird zusätzlich ein Strahl gemäß den Brechungsgesetzen erzeugt. Wird ein diffuses Material getroffen, wird wie es auch in Appels Verfahren der Fall ist, ein Schattenstrahl ausgesendet.

Die Erzeugung der Sekundärstrahlen wird rekursiv fortgesetzt, bis diese entweder eine Lichtquelle treffen oder eine vorher festgelegte maximale Rekursionstiefe erreicht ist.

Der grundlegende Ablauf dieses *Raytracing-Verfahrens* kann in Pseudocode wie folgt notiert werden (angelehnt an [Jen01]):

```
function render()
{
  for(each pixel)
  {
    Ray = generateEyeRay(pixel);
    pixel.color = trace(Ray);
  }
}
```

```
function trace(Ray)
{
  Point = calculateIntersectionPoint(Ray);
  Normal = calculateNormal(Point);
  Color = shade(Point, Normal);
  return Color;
}

function shade(Point, Normal)
{
  Color;
  for(each light source)
  {
    bool hitsLightSource = testShadowRay(Point, light);
    if(hitsLightSource)
    {
      Color = Color + calculateDirectIllumination();
    }

    if(surface is specular)
    {
      Ray = generateReflectedSecondaryRay();
      Color = Color + trace(Ray);
    }

    if(surface is transparent)
    {
      Ray = generateRefractedSecondaryRay();
      Color = Color + trace(Ray);
    }
  }
  return color;
}
```

Abbildung 2 zeigt eine mit Hilfe von *Mental Ray* gerenderte Szene.

Der von Whitted entwickelte Raytracer ist zwar in der Lage, die globalen Beleuchtungseffekte Reflexion und Refraktion darzustellen, jedoch sind die dargestellten Reflexionen immer perfekt, was erheblich von den in der Natur auftretenden Reflexionen abweicht. Des Weiteren sind die erzeugten Schatten scharf abgegrenzt, während sich in der Realität eher weiche Schatten finden lassen. Bei dem Verfahren werden keine Interaktionen zwischen diffusen und transparenten/spiegelnden Oberflächen berück-

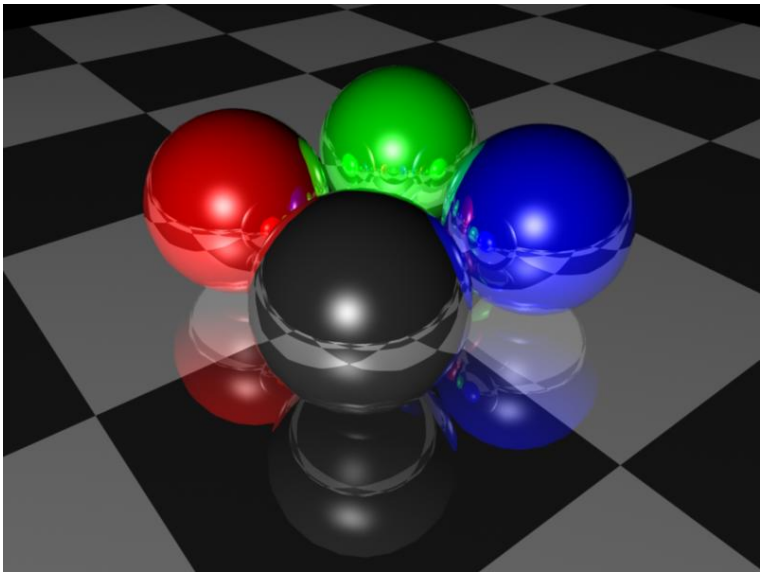

Abbildung 2: Eine mit Hilfe von Raytracing gerenderte Szene

sichtigt, was aber für die Generierung von Kaustiken erforderlich wäre. Zu guter Letzt entstehen durch das Abtasten der Szene mit nur einem Strahl pro Pixel Aliasing-Artefakte.

Für die Beseitigung dieser Probleme und eine entsprechende Erweiterung von Whitteds Raytracer, muss zunächst die theoretische Grundlage für die Berechnung von globalen Beleuchtungseffekten geschaffen werden.

2.4.2 Die Rendergleichung

Um die in einer 3D-Szene auftretenden Effekte der globalen Beleuchtung exakter zu berechnen und damit ein reales Bild zu approximieren, ist die Ermittlung der Strahlungsverteilung des Lichtes notwendig. Im Prinzip wäre dies mit den Maxwell-Gleichungen, der speziellen Relativitätstheorie und der Quantenmechanik möglich; in der Computergrafik erweist sich dieses Verfahren jedoch als unpraktikabel, da die auftretenden Phänomene zu detailliert erklärt werden. Stattdessen bedient man sich der geometrischen Optik.

Kayjiya führt in [Kay86] die sogenannte *Rendergleichung* ein, die die Energieerhaltung bei der Ausbreitung des Lichtes in einer 3D-Szene beschreibt und die Grundlage der folgenden globalen Beleuchtungsverfahren bildet.

Definition 2.4.1 (Rendergleichung nach Kayjiya)

$$L(x, \varphi) = L_e(x, \varphi) + \int_{\Omega_x} b(\varphi', x, \varphi)\ L(x, \varphi')(\varphi' \cdot n)\ d\varphi'$$

Wobei

- $L(x, \varphi)$ *ist die ausgehende Strahlungsdichte einer Oberfläche am Punkt* x *in Richtung* φ
- $L_e(x, \varphi)$ *ist die abgegebene Strahlungsdichte einer Oberfläche am Punkt* x *in Richtung* φ *ohne die reflektierte Strahlungsdichte*
- Ω_x *ist die Einheits-Hemisphäre um* x
- $b(\varphi', x, \varphi)$ *ist der Streuungsterm in Form einer BRDF*[10] *und gibt die aus Richtung* φ' *in Richtung* φ *reflektierte Strahlungsdichte an*
- n *ist die Normale der Oberfläche an Punkt* x

Vereinfacht lautet die Aussage der Rendergleichung wie folgt: Das an einem Punkt x in Richtung φ abgestrahlte Licht entspricht dem durch die Oberfläche abgegebenen Licht zuzüglich der Summe des in Richtung φ reflektierten einkommenden Lichtes.

2.4.3 Path Tracing

In [Kay86] stellt Kayjiya zusammen mit der Rendergleichung das sogenannte *Path Tracing* vor.

Anstatt bei der Erzeugung der Sekundärstrahlen nur *einen* Reflexions- bzw. Refraktionsstrahl auszusenden, werden bei Path Tracing *mehrere* zufällige Sekundärstrahlen erzeugt, die das Integral der Rendergleichung in Definition 2.4.1 nähern. Jeder Primärstrahl sucht sich mit Hilfe dieser Strahlen seinen Weg durch die Szene. Je mehr Primärstrahlen man pro Pixel verwendet, desto besser ist die Annäherung an die Rendergleichung und somit auch an das reale Bild.

Durch das Path Tracing wird der von Whitted entwickelte Raytracer wie folgt erweitert bzw. verbessert:

[10] Bidirectional Reflectance Distribution Function: Liefert für jeden auf ein Material eintreffenden Lichtstrahl den Quotienten aus Bestrahlungsstärke und Strahlungsdichte für jeden austretenden Lichtstrahl.

- Darstellung von weichen Schatten
- Darstellung verschwommener Lichtreflexionen auf spiegelnden Oberflächen
- Anti-Aliasing

In der Praxis ist reines Path Tracing meist zu langsam. Aus diesem Grund kann es mit dem im Folgenden vorgestellten *Photon Mapping* kombiniert werden.

2.4.4 Photon Mapping

Jensen und Christensen stellen in [JC95] das *Photon Mapping* als Erweiterung des klassischen Raytracing-Verfahrens vor. Bei diesem Verfahren werden vor dem eigentlichen Raytracing Photonen von der Lichtquelle aus in die Szene geschossen. Diese werden an den Objekten der Szene reflektiert, gestreut oder gebrochen, abhängig von der Oberflächeneigenschaft des jeweiligen Objekts. Treffen die Photonen auf eine diffuse Oberfläche, werden sie in einer *Photon Map* gespeichert.

Die so gewonnene Photon Map wird im anschließenden Raytracing dazu verwendet, die Dichte und die Energie der einen Strahlenschnittpunkt umgebenden Photonen zu berechnen, um daraus den Beitrag zur indirekten Beleuchtung zu erhalten.

Zur Darstellung von Kaustiken werden die Photonen in einer separaten *Caustic Map* gespeichert. Abbildung 3 zeigt eine Szene, die durch *Mental Ray* mit Hilfe von Photon Mapping gerendert wurde.

Um die Ergebnisse der globalen Beleuchtung und insbesondere den Beitrag der diffusen Flächen weiter verbessern zu können, wurden die in den meisten Renderern gemeinsam verwendeten Verfahren *Final Gather* und *Global Illumination* eingeführt. Auf diese beiden Verfahren wird im Rahmen dieser Arbeit nicht weiter eingegangen, da sie zu rechenaufwendig sind, als dass sie in einen Szenengraph integriert werden und interaktive Bildraten erreichen könnten. Genaueres zu den Verfahren *Final Gather* und *Global Illumination* kann [DBB06] entnommen werden.

2.5 Beschleunigungsdatenstrukturen

Bereits Whitted stellte bei der Implementierung seines Raytracers fest, dass die meiste Rechenzeit dafür aufgewendet wird, festzustellen, mit welchem Primitiv der Szene sich ein Strahl schneidet. Die Laufzeit des Verfahrens wird dadurch maßgeblich bestimmt. Der erste Ansatz, der die Durchführung eines Schnitttests mit jedem Primitiv der

Abbildung 3: Eine mit Hilfe von Photon Mapping gerenderte Szene (übernommen aus [may03])

Szene vorsieht, ist ineffizient: Bei n Primitiven ist die Komplexität der Suche nach dem geeigneten Primitiv $\Theta(n)$.

Whitted verwendet eine *Hüllkörperhierarchie* als Beschleunigungsdatenstruktur, in der die Primitive der Szene organisiert werden. Durch Traversierung der Datenstruktur kann das in Frage kommende Primitiv in $\mathrm{O}(\log n)$ gefunden werden.

2.5.1 Hüllkörperhierarchie

Eine *Hüllkörperhierarchie* oder auch *Bounding Volume Hierarchie (BVH)* ist ein Baum von hierarchisch angeordneten Hüllkörpern wie beispielsweise *achsparallele Quader (AABB)*, *Objekt ausgerichtete Quader (OBB)* oder *Kugeln.*

Die Blätter des Baums definieren einen Hüllkörper, der gerade groß genug ist, ein einzelnes Primitiv der Szene zu umhüllen. Die Knoten des Baums hingegen definieren einen Hüllkörper, der die Hüllkörper all seiner Kinder umschließt. An der Wurzel des Baums befindet sich ein Hüllkörper, der alle Primitive der Szenen einschließt.

Der grundlegende Aufbau einer Hüllkörperhierarchie kann Abbildung 4 entnommen werden: Der linke Teil der Abbildung zeigt die Primitive der Szene und ihre Hüllkörper, der rechte Teil die daraus resultierende Hüllkörperhierarchie.

Der Schnitttest zwischen einem Hüllkörper und einem Strahl kann wesentlich schneller

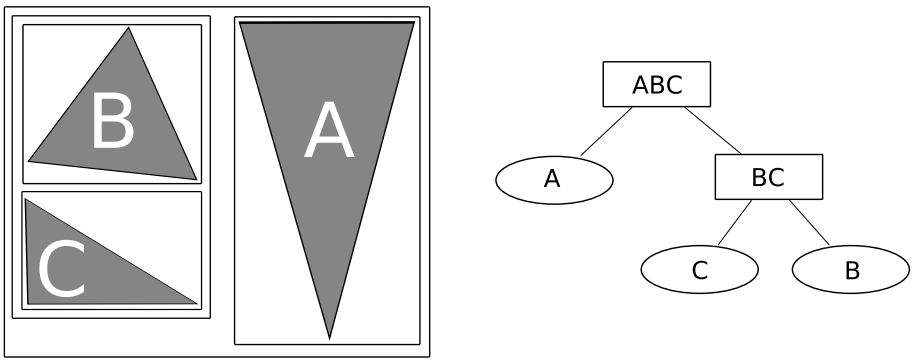

Abbildung 4: Aufbau einer Hüllkörperhierarchie

durchgeführt werden, als die Schnittberechnung zwischen einem Primitiv und einem Strahl. Des Weiteren gilt: Ein Strahl der einen Hüllkörper verfehlt, trifft auch das in ihm eingeschlossene Primitiv nicht, was unnötige Schnittberechnungen zwischen den Strahlen und den Primitiven der Szene vermeidet.

Bei der Suche nach einem Schnittpunkt wird die BVH rekursiv traversiert und an jedem Knoten ein Schnitttest zwischen dem Strahl und dem entsprechenden Hüllkörper durchgeführt. Verfehlt der Strahl den Hüllkörper eines Knotens nicht, wird so lange fortgefahren, bis ein Blatt und somit auch ein Primitiv der Szene erreicht ist. Mit dem so gefundenen Primitiv wird eine Schnittberechnung durchgeführt.

Weghorst untersucht in [WHG84] verschiedene Formen von Hüllkörpern auf ihre Performance hin. Dabei stellt er fest, dass die Kosten für die Schnittberechnung zwischen Hüllkörper und Strahl nicht das einzige Kriterium für die Wahl der Form sind. Die Schnittberechnung kann beispielsweise bei Kugeln sehr schnell durchgeführt werden, diese Hüllkörper haben jedoch den Nachteil, dass sie das Objekt in den seltensten Fällen dicht umschließen. Sie verfügen über einen großen Leerraum, der zu unnötigen Schnittberechnungen mit dem Primitiv führt.

Neben der Form der verwendeten Hüllkörpern ist die Struktur der BVH für die Geschwindigkeit der Schnittberechnung maßgeblich. Da bis dato kein Verfahren zur Erzeugung einer optimalen BVH bekannt ist, wurden von Kay und Kayjiya in [KK86] folgende Kriterien für eine annähernd optimale BVH eingeführt:

(i) Jeder Teilbaum der BVH sollte nur Objekte beinhalten, die räumlich nahe beieinander liegen.

(ii) Das Volumen jedes Knotens sollte minimal sein

(iii) Die Summe aller Volumina sollte minimal sein

(iv) Die Konstruktion der BVH sollte sich auf die Knoten nahe der Wurzel konzentrieren, da dies eventuell Teilbäume entfernt und somit zu einer Zeitersparnis beim späteren Rendern führt.

(v) Je mehr Zeit in die Konstruktion einer guten BVH investiert wird, desto mehr Zeit wird beim Rendern der Szene gewonnen.

Zur Erzeugung einer BVH existieren mehrere Verfahren, die aber im Wesentlichen auf die von Kay und Kayjiya in [KK86] sowie von Goldsmith und Salomon in [GS87] entwickelten Verfahren zurückzuführen sind.

Das Verfahren von Kay und Kayjiya verwendet einen rekursiven Top-Down-Ansatz, in dem die Liste aller Primitive der Szene so lange in zwei Hälften unterteilt wird, bis die zwei Teillisten höchstens aus je zwei Elementen bestehen. Die Unterteilung erfolgt dabei in jeder Rekursionsstufe entlang jener Achse, die gemäß obigen Kriterien die günstigste ist. Diese Vorgehensweise erzeugt einen binären Baum. Eine Erweiterung des Verfahrens wird in Kapitel 5.3.3 vorgestellt.

Das Verfahren von Goldsmith und Salomon verwendet einen Bottom-Up-Ansatz. Zunächst wird das erste Primitiv an die Wurzel des Baums gesetzt, für jedes weitere einzufügende Primitiv wird die Position mit Hilfe einer Abstandsfunktion bestimmt und das Primitiv an der günstigsten Stelle in den Baum eingefügt.

Die Qualität der so erzeugten BVH hängt von der ursprünglichen Sortierung der Primitive ab. Der resultierende Baum ist nicht binär.

Hüllkörperhierarchien sind nicht nur als Beschleunigungsdatenstruktur für das Raytracing von Bedeutung, sie werden auch zur Kollisionserkennung eingesetzt. Details können beispielsweise [Ber97] entnommen werden.

2.5.2 Reguläres Gitter

Das *reguläre Gitter* oder auch *Uniform Grid* zählt im Gegensatz zu der Hüllkörperhierarchie als Geometrieunterteilungsverfahren zu den räumlichen Unterteilungsverfahren und ist der einfachste Vertreter dieser Klasse.

Das Verfahren unterteilt den gesamten Raum einer Szene in gleich große Teilräume auch *Voxel*[11] genannt. Die Unterteilung des Raums wird ohne Rücksicht auf die Anordnung der Primitive der Szene durchgeführt, was in ungünstigen Fällen zu sehr großen Leerräumen führen kann. Dadurch wird bei der Traversierung für unnötig viele

[11] Volumemetric Element

Voxel ein Schnitttest durchgeführt. Dieses Problem bezeichnet Eric Haines in [Hai88] als „Teapot in a stadium“-Problem.

Die Traversierung eines regulären Gitters erfolgt mit Hilfe einer dreidimensionalen Abwandlung des *Digital Differential Analyzers (DDA)*, der zum Zeichnen von Linien auf einem Rastergerät verwendet wird. Mit Hilfe dieses Verfahrens berechnet man jene Voxel, die der Strahl durchquert und führt eine Schnittberechnung mit denjenigen Primitiven durch, die innerhalb dieses Voxels liegen. Die Auflösung des Gitters entscheidet somit über die Qualität der Beschleunigungsdatenstruktur. Genaueres hierzu kann [FTI86] entnommen werden.

2.5.3 kd-Baum

Ein *kd-Baum* oder auch *k-dimensionaler Baum* zählt ebenfalls zu den räumlichen Unterteilungsverfahren und wurde von Bentley in [Ben75] eingeführt. Der Raum wird mit Hilfe von achsparallelen Ebenen so lange unterteilt, bis ein Teilraum keine oder nur noch wenige Primitive enthält. Dieses Vorgehen erzeugt einen binären Baum, dessen Knoten die Teilräume repräsentieren. An den Blättern des Baums befinden sich diejenigen Teilräume, die nicht weiter unterteilt werden.

Die Qualität eines kd-Baums hängt von der Positionierung der Unterteilungsebenen ab.

Ein *kd-Baum* ist der Spezialfall des von Fuchs in [Fuc80] vorgestellten *BSP-Baums*[12], bei dem die Positionen der Unterteilungsebenen beliebig gewählt werden können. Die Schnittberechnung zwischen einem Strahl und solch einer beliebig orientierten Unterteilungsebene ist aufwendiger als die Schnittberechnung mit einer achsparallelen Unterteilungsebene. BSP-Bäume finden ihre Anwendung hauptsächlich in 3D-Spielen, wo sie zum effizienten Laden von großen Innenleveln verwendet werden.

Der von Glassner in [Gla86] beschriebene *Octree*, der den Raum rekursiv in acht gleich große Teilräume zerlegt, ist ein Spezialfall des kd-Baums.

Kd-Baum, BSP-Baum und auch der Octree leiden unter dem „Teapot in a stadium“-Problem.

[12] Binary Space Partition

2.6 Weitere Beschleunigungsdatenstrukturen

Es existieren weitere Beschleunigungsdatenstrukturen, unter anderem Mischformen der vorgestellten Datenstrukturen. In dieser Arbeit wurde die Auswahl der Beschleunigungsdatenstrukturen auf die obigen beschränkt, da diese in bisherigen Arbeiten über Echtzeit-Raytracer als Beschleunigungsdatenstrukturen verwendet wurden, wie Kapitel 3.2 zeigen wird.

2.7 Darstellung von globalen Beleuchtungseffekten durch aktuelle Grafikhardware

In [Fer05] entwickeln die Autoren einige interessante Verfahren zur Darstellung von globalen Beleuchtungseffekten auf der aktuellen Generation von Grafikkarten.

King stellt ein Verfahren zur Echtzeitberechnung von Strahlungsdichten-Environment-Maps vor. Diese Technik approximiert die komplexen globalen Beleuchtungseffekte in dynamischen Umgebungen, wie beispielsweise das Radiosity-Verfahren.

Bunnell hat ein interessantes Verfahren entwickelt, das es ermöglicht, diffusen Lichttransfer (Ambient Occlusion) für animierte Szenen zu berechnen.

Toshiyas Artikel „High-Quality Global Illumination Rendering Using Rasterization“ (ebenfalls in [Fer05]) beschreibt ein Verfahren zur Errechnung von globalen Beleuchtungseffekten durch indirekte Beleuchtung mit Hilfe des *Final Gather* Verfahrens. Das Besondere an dieser Vorgehensweise ist, dass die Strahl-Objekt-Schnittberechnungen durch den Rasterisierer der Grafikhardware realisiert werden, anstatt wie beispielsweise von Purcell in [Pur04] beschrieben das komplette Raytracing-Verfahren auf die Grafikkarte abzubilden.

2.8 Mathematische Grundlagen

In diesem Kapitel werden die für diese Arbeit notwendigen mathematischen Grundlagen erarbeitet. Begonnen wird mit der Definition der **baryzentrischen Koordinaten**, die für Schnittberechnung zwischen einem Strahl und einem Dreieck benötigt werden.

Definition 2.8.1 (Baryzentrische Koordinaten)
Seien A, B und C die Eckpunkte eines Dreiecks im Vektorraum V. Jeder weitere Punkt des Dreiecks lässt sich als gewichtetes arithmetisches Mittel der Punkte A, B und C wie folgt beschreiben:

$$P = u \cdot A + v \cdot B + t \cdot C$$

Sind die Bedingungen $u + v + t = 1$ und $u, v, t \geqslant 0$ erfüllt, so liegt Punkt P innerhalb des Dreiecks. Man nennt die Koeffizienten u, v und t von P die ***baryzentrischen Koordinaten*** *des durch A, B und C aufgespannten Dreiecks.*

Ein Strahl $R(p)$ mit Ursprung O und normierter Richtung D kann repräsentiert werden als

$$R(p) = O + pD \tag{1}$$

Möller und Trumbore stellen in [MT97] folgendes effizientes Verfahren zur Schnittberechnung zwischen einem Dreieck und einem Strahl vor:

Definition 2.8.2 (Schnittberechnung zwischen Strahl und Dreieck)
Seien u, v und t die baryzentrischen Koordinaten des Dreiecks mit den Eckpunkten A,B,C und sei R(p) ein Strahl. Nach Def. 2.8.1 kann jeder weitere Punkt T des Dreiecks in Abhängigkeit von u und v geschrieben werden als:

$$T(u, v) = (1 - u - v)A \ + \ uB \ + \ vC \quad (mit\ u, v \geq 0) \tag{2}$$

Der Schnittpunkt zwischen Strahl und Dreieck berechnet sich wie folgt:

$$O + pD = (1 - u - v)A \ + \ uB \ + \ vC \tag{3}$$

Umformen ergibt:

$$[-D, B - A, C - A] \begin{bmatrix} p \\ u \\ v \end{bmatrix} = O - A \tag{4}$$

Definiert man $E = B - A$, $D = C - A$ und $T = O - A$ erhält man durch Anwenden der Cramerschen Regel die Lösung des Systems aus (4):

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E) \cdot F} \begin{bmatrix} (T \times E) \cdot F \\ (D \times F) \cdot T \\ (T \times E) \cdot D \end{bmatrix} \tag{5}$$

und somit den gesuchten Schnittpunkt, wobei u und v die baryzentrischen Koordinaten des Dreiecks sind und p die Distanz zwischen dem Ursprung des Strahls und dem Schnittpunkt ist.

Nachdem das Verfahren zur Bestimmung des Schnittpunkts zwischen Strahl und Dreieck vorgestellt wurde, ist ein weiteres Verfahren notwendig, das später für die Implementierung der Traversierung der BVH benötigt wird. Es wird getestet, ob ein Strahl einen achsparallelen Hüllquader (AABB) schneidet, wobei der Schnittpunkt hier unerheblich ist.

Das Verfahren kann wie folgt realisiert werden:

Definition 2.8.3 (Schnitttest zwischen AABB und Strahl)
Sei $\boldsymbol{B_{min}}$ der minimale Eckpunkt der AABB, $\boldsymbol{B_{max}}$ der maximale Eckpunkt der AABB, $\boldsymbol{O}$ der Ursprungspunkt des Strahls und $\boldsymbol{D}$ der Richtungsvektor des Strahls. Um festzustellen, ob der Strahl die AABB schneidet, berechnet man zunächst die Werte:

$$T_{min} = \frac{(B_{min} - O)}{D} \tag{6}$$

$$T_{max} = \frac{(B_{max} - O)}{D} \tag{7}$$

und ermittelt die maximale Koordinate U_{max} von T_{min} und die minimale U_{min} Koordinate von T_{max}. Gilt nun

$$U_{min} \geq U_{max} \tag{8}$$

so schneidet der Strahl die AABB, ansonsten verfehlt er sie.

Neben den oben behandelten geometrischen Requisiten werden noch Grundlagen der Beleuchtungsrechnung benötigt, auf denen das beim Raytracing verwendete lokale Beleuchtungsmodell nach Phong aufbaut.

Definition 2.8.4 (Diffuse Reflexion nach Lambert)
Ist $\boldsymbol{L}$ der Richtungsvektor von der Oberfläche zur Lichtquelle, $\boldsymbol{N}$ der Normalenvektor der Oberfläche, $\boldsymbol{D}$ die diffuse Farbe der Oberfläche und $\boldsymbol{I_L}$ die Intensität des Lichtes, so berechnet sich die diffuse Intensität der Oberfläche $\boldsymbol{I_D}$ wie folgt:

$$I_D = L \cdot N \cdot D \cdot I_L$$

Definition 2.8.5 (Spiegelnde Reflexion nach Phong)
Ist $\boldsymbol{R}$ der Richtungsvektor des ideal reflektierten Lichtstrahls, $\boldsymbol{V}$ die normierte Blickrichtung des Betrachters, $\boldsymbol{S}$ die Farbe der Oberfläche, $\boldsymbol{k}$ ein konstanter Faktor zur Beschreibung der Rauigkeit der Oberfläche sowie $\boldsymbol{I_L}$ die Intensität des Lichtes, dann berechnet sich die Intensität der spiegelnden Oberfläche $\boldsymbol{I_S}$ wie folgt:

$$I_S = (R \cdot V)^k \cdot S \cdot I_L$$

Definition 2.8.6 (Lokales Beleuchtungsmodell nach Phong)
Ist $\boldsymbol{I_D}$ *der diffuse Anteil der lokalen Beleuchtung aus Def. 2.8.4,* $\boldsymbol{I_S}$ *der spiegelnde Anteil der Beleuchtung aus Def. 2.8.5 und* $\boldsymbol{A}$ *die ambiente Farbe des Objekts, so berechnet sich der lokale Anteil der Beleuchtung wie folgt:*

$$I_{Total} = A \cdot I_L + \sum_{Lichtq.} (I_D + I_S)$$

3 State-of-the-Art Analyse

In diesem Kapitel wird eine State-of-the-Art Analyse durchgeführt, in der die vergangenen und aktuellen Forschungsergebnisse im Bereich Echtzeit-Raytracing vorgestellt und analysiert werden.

3.1 Die GPU als Stream-Prozessor

Wie in Kapitel 2.3 bereits erwähnt wurde, sind aktuelle Generationen von Grafikchips (GPUs) in der Lage, auf mehreren Eingabeströmen von Eckpunkten oder Fragmenten parallel zu arbeiten. GPUs können somit als Vertreter der *SIMD*[13] Architektur nach Flynn [Fly95] angesehen werden, wie von Peercy et al. in [POAU00] beschrieben.

Durch die Einführung der Shader-Sprachen *GLSL* durch das OpenGL ARB[14], *HLSL* durch Microsoft und *Cg* durch NVIDIA können die Vertex- und Fragment-Prozessoren der Renderingpipeline programmiert werden. Aus der fixed-function-Pipeline wird so eine rekonfigurierbare Pipeline [FK03].

3.1.1 Der programmierbare Vertex-Prozessor

In der einfachen Darstellung der Renderingpipeline aus Abbildung 1 in Kapitel 2.3 werden die Eckpunkte (Vertices) in der ersten Stufe transformiert. Aktuelle Grafikpipelines führen diese Stufe durch einen oder mehrere programmierbare Vertex-Prozessoren aus.

Vertex-Prozessoren übernehmen die Funktion der ursprünglichen Pipelinestufe, bieten aber darüber hinaus die Möglichkeit, die Attribute der Vertices, wie Position, Farbe und Texturkoordinaten, mit Hilfe von *Vertex-Shadern* zu verändern.

Details hierzu können [FK03] entnommen werden.

3.1.2 Der programmierbare Fragment-Prozessor

Der programmierbare Fragment-Prozessor tritt an die Stelle der dritten Stufe der Pipeline aus Abbildung 1 in Kapitel 2.3. Neben den ursprünglichen Funktionen dieser Stufe ermöglicht er die Durchführung arithmetischer Operationen auf mehreren

[13] Single Instruction Multiple Data
[14] Architecture Review Board

Texturkoordinaten und Fragmenten durch sogenannte Fragment-Shader[15]. Des Weiteren können Fragment-Shader ihre Ausgabe in mehrere Texturen, sogenannte *Multi-Render-Targets* schreiben, die wiederum als Eingabe für weitere Fragment-Shader dienen können.

Die in einem Shader-Programm verfügbaren Operationen reichen von einfachen mathematischen Funktionen über Funktionen zur Matrixmultiplikation bis hin zu Schleifen, Bedingungen und Funktionen. Details hierzu können [FK03] sowie [Kir02] entnommen werden.

Die Ausführung von Fragment-Shadern ist etwa zehnmal schneller als die Ausführung von Vertex-Shadern [NVI05].

3.1.3 Multi-Render-Targets

Eine GPU verfügt über keinen Speicher mit wahlfreiem Zugriff, sondern lediglich über einen Texturspeicher. Aus diesem kann mit Hilfe von Texturkoordinaten zwar von beliebigen Stellen gelesen (gather), aber nicht an beliebige Positionen geschrieben (scatter) werden. Dagegen kann in einem Fragment-Shader nur der Inhalt der Textur an der betreffenden Position des Fragments verändert werden.

Um die Ausgabe von Fragment-Shadern speichern zu können, ohne sie direkt in den Bildspeicher zu schreiben, unterstützen aktuelle Generationen von GPUs sogenannte *Multi-Render-Targets* (*MRTs*). MRTs bestehen aus einem oder mehreren Pixel-Puffern, die wie Texturen behandelt werden können und als Ein- und Ausgabe für die Fragment-Shader dienen. Weitere Details sind [Cin04] zu entnehmen.

3.1.4 General Purpose Computation on GPU

Die Möglichkeit, die Renderingpipeline mit Hilfe von Vertex- und Fragment-Shadern umzukonfigurieren und den Texturspeicher der Grafikkarte zu manipulieren, legt den Einsatz der GPU für parallele Berechnungen außerhalb der Rasterisierung nahe.

Abbildung 5 illustriert den Unterschied zwischen der Verwendung der GPU für allgemeine Berechnungen (General Purpose Computations) und deren Verwendung für die Rasterisierung.

In beiden Pipelines werden die Eckpunkte der Primitive im ersten Schritt in das Kamerakoordinatensystem transformiert. Während die Eckpunkte bei der Standard-

[15] In der Literatur wird häufig Pixel-Shader synonym für Fragment-Shader verwendet.

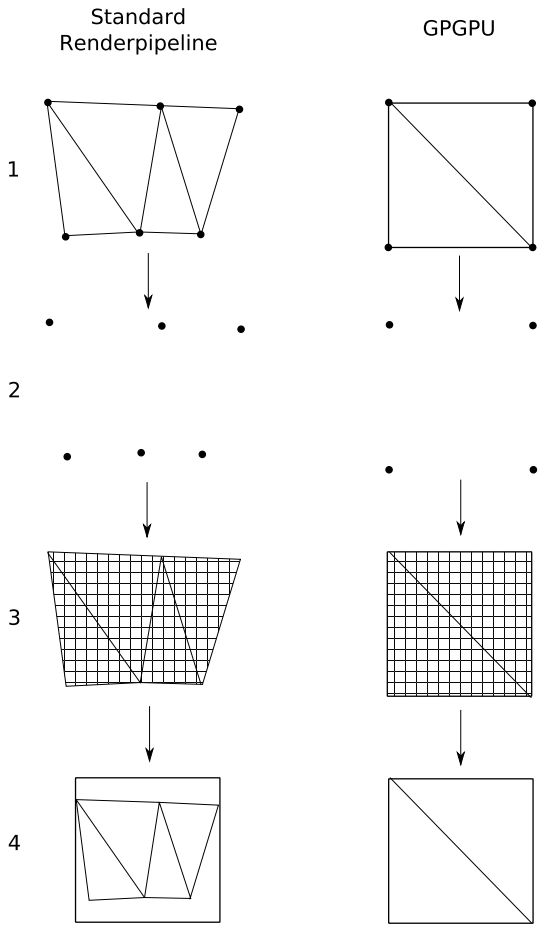

Abbildung 5: Vergleich zwischen der Standard-Renderingpipeline und der GPGPU-Pipeline (übernommen aus [TS05])

Renderingpipeline von den Primitiven der zu rendernden Szene stammen, spannen sie im Fall der GPGPU-Pipeline ein bildschirmfüllendes Rechteck auf. Die Eckpunkte werden im zweiten Schritt an den Vertex-Prozessor übergeben, der eventuell vorhandene Vertex-Shader ausführt und eine Menge von Fragmenten generiert.

Die eigentliche Berechnung erfolgt in der dritten Stufe mit Hilfe einer Menge von Fragment-Shadern, die ihre Zwischenergebnisse in MRTs speichern und das Endergebnis in den Bildspeicher schreiben.

Buck et al. beschreiben dieses Vorgehen ausführlich in [BLOS04].

Das Projekt GPGPU[16] enthält einen Überblick über Projekte, die parallele Probleme durch Implementierung eines Verfahrens auf der GPU lösen. Die stetig wachsende

[16]http://www.gpgpu.org

Anzahl von Beiträgen zu diesem Thema, lässt einen Trendanstieg in der Verwendung der GPU für allgemeine Berechnungen erkennen.

Beispiele für parallele Probleme, deren Lösungsverfahren sich für die Implementierung auf einer GPU eignen, sind Audio- und Signalverarbeitung [Wha04], oder solche im Bereich der Mathematik, wie beispielsweise das Lösen linearer Gleichungssysteme [GGHM05]. Da die Strahlen der Pixel beim Raytracing unabhängig voneinander verfolgt werden, ist das Verfahren parallelisierbar. Es eignet sich somit sehr gut für die Implementierung auf der GPU.

3.1.5 C for Graphics: Eine Shader-Sprache

Die von Microsoft eingeführte *High Level Shading Language (HLSL)* und die vom OpenGL ARB entwickelte *GL Shading Language (GLSL)* haben beide einen entscheidenden Nachteil: Sie sind nur innerhalb der jeweiligen Grafik-API verfügbar. Der Grafikkartenhersteller *NVIDIA* hat daher eine Sprache entwickelt, die eine Verwendung der Shader-Sprache in beiden Grafik-APIs gestattet [FK03].

Die Syntax der von NVIDIA entwickelten Sprache *C for Graphics (Cg)* ist an HLSL angelehnt und bietet den gleichen Funktionsumfang wie *HLSL* und *GLSL*. Zwar lässt sich ein in *Cg* implementiertes Shader-Programm nur auf Grafikkarten des Herstellers *NVIDIA* kompilieren, die Transformation in HLSL- und GLSL-Programme ist jedoch ohne größeren Aufwand möglich.

3.2 CPU- vs. GPU-basiertes Raytracing

In diesem Abschnitt werden die bisherigen Verfahren zur Implementierung des Raytracing-Verfahrens auf CPU und GPU analysiert und verglichen.

3.2.1 Bisherige Arbeiten über CPU-basierte Raytracer

Im Rahmen dieser Arbeit wird nur auf die historisch wichtigsten und für diese Arbeit relevanten Verfahren für CPU-basiertes Raytracing eingegangen.

Fellner et al. stellen in [FSZ98] ein Verfahren zur verteilten Berechnung von Raytracing und Radiosity in einem heterogenen Netzwerk vor. Die Autoren erweitern das „Minimal Rendering Toolkit“ um die Möglichkeit der Parallelverarbeitung und der Lastverteilung in einem LAN aus Workstations und PCs. Dadurch wird eine lineare

Beschleunigung bei komplexen Szenen erreicht. Bei einfachen Szenen ist keine lineare Beschleunigung möglich, da die Zeiten, in denen die Clients des Netzwerks auf Eingaben warten, zu lang sind.

Das Verfahren ist trotz linearer Beschleunigung nicht in der Lage, Ergebnisse in Echtzeit zu liefern.

In dem von Wald et al. in [WSBW01] vorgestellten Verfahren wird das Raytracing ebenfalls auf einem Rechner-Cluster verteilt. Das implementierte System erlaubt die Berechnungen in Echtzeit und lässt sogar interaktive Bildraten zu. Es wurde stetig weiterentwickelt und um zahlreiche Features wie Kaustiken oder das Raytracing dynamischer und interaktiver Szenen erweitert. Ursprünglich unter einer kommerziellen Lizenz vertrieben, ist das System inzwischen als freie Version für die Verwendung in nicht kommerziellen Projekten unter dem Namen *OpenRT*[17] erhältlich.

In [WIK+06] präsentieren Wald et al. eine Erweiterung des Systems, die das Raytracing dynamischer Szenen bei interaktiven Bildraten unter Verwendung eines dualen Intel Xeon Prozessors ermöglicht.

Schmittler et al. verwenden das von Wald et al. vorgestellte OpenRT in [SPD+04] für die Darstellung globaler Beleuchtungseffekte in 3D-Spielen wie *Quake3: Arena*. In dem von Schmittler et al. entwickelten Spiel *Oasen* wird das Raytracing nicht nur zur Berechnung globaler Beleuchtungseffekte, sondern auch als Physik-Engine und zur Kollisionserkennung verwendet.

Hurley vergleicht in [Hur05] die Rasterisierung und das Raytracing-Verfahren in Echtzeit-Anwendungen und hebt dabei die Skalierbarkeit des Raytracing-Verfahrens als wesentlichen Vorteil des Verfahrens hervor. Das von ihm implementierte System ermöglicht das Raytracing auf einem Pentium 4 Prozessor und ist von SMP[18] bis hin zu Rechner-Clustern linear skalierbar.

In [Nem05] stellt Nemec die Implementierung eines Raytracers auf einem Prozessor der heutigen Generation vor. Dabei wird die Möglichkeit aktueller Prozessoren zur parallelen Verarbeitung mit Hilfe der Befehlssatzerweiterungen SSE[19] und SSE2 genutzt. Das System ist in der Lage, statische Szenen bei interaktiven Bildraten darzustellen.

Durch die Verwendung von „Pixel Selected Ray Tracing“ ist eine Verkürzung der Rechenzeit um ca. das Achtfache möglich. Hierzu werden Primärstrahlen an den Eckpunkten eines vorgegebenen Rasters und nicht mehr wie vorher durch jeden Pixel der Bildebene gesendet. Unterscheiden sich die errechneten Farbwerte dieser Eckpunkte

[17] http://www.openrt.de
[18] Symmetrische Multiprozessorsysteme
[19] Streaming SIMD Extensions

nur um einen vorher definierten Wert, so werden die Farbwerte der Pixel zwischen den Eckpunkten interpoliert. Übersteigt der Unterschied der Farbwerte den definierten Wert, so werden die Farbwerte der Pixel durch Entsendung von Primärstrahlen berechnet. Dieses Verfahren führt zu minimalen Verlusten der Bildqualität.

Eine Erweiterung dieses Vorgehens erlaubt es, adaptives Anti-Aliasing zu implementieren. Dazu wird die Szene zunächst mit einem Strahl pro Pixel abgetastet, und der so erhaltene Farbwert mit denen der benachbarten Pixel verglichen. Überschreitet der Unterschied der Farbwerte einen bestimmten Wert, so werden die betreffenden Pixel mit Hilfe zweier Strahlen erneut abgetastet.

Nemec verwendet einen kd-Baum als Beschleunigungsdatenstruktur. Aufgrund das objektorientierten Konzepts ist eine Integration anderer Beschleunigungsdatenstrukturen möglich.

3.2.2 Bisherige Arbeiten über GPU-basierte Raytracer

Der erste Ansatz zur Verwendung der GPU für die Beschleunigung des Raytracing-Verfahrens wurde von Carr et al. in [CHH02] vorgestellt. Um die Strahl-Dreieck-Schnittberechnung durchführen zu können, verwenden die Autoren einen Fragment-Shader. Die Generierung und Traversierung des verwendeten Octrees, die Generierung der Primär- und Sekundärstrahlen sowie das Shading der Pixel wird von der CPU übernommen. Eine komplette Implementierung des Raytracers auf der GPU ist bis zu diesem Zeitpunkt noch nicht möglich gewesen, da die früheren Generation von Grafikchips und Shader-Sprachen keine MRTs unterstützten.

Dadurch dass die Schnittberechnung ausgelagert wird, müssen die Daten zur Weiterverarbeitung von der GPU über den AGP-Bus[20] zurückgelesen werden. Die beschränkte Bandbreite des Busses ist der Flaschenhals des Systems. Nichtsdestotrotz ist dieses Verfahren schneller als die zuvor von Wald et al. entwickelte CPU-basierte Lösung.

Purcell entwickelt als erster einen vollständig GPU-basierten Raytracer. In [Pur04] unterteilt er das Raytracing-Verfahren in die Teilprobleme Primärstrahlengenerierung, Traversierung der Datenstruktur, Schnittberechnung und Shading. Diese Teilprobleme werden auch als Kernels bezeichnet und von Purcell mit Hilfe von Fragment-Shadern und MRTs implementiert. Als Beschleunigungsdatenstruktur wird ein reguläres Gitter verwendet, dessen Generierung durch die CPU erfolgt. Seine Implementierung erreicht erstmals interaktive Bildraten.

[20] Accelerated Graphics Port

Purcells Implementierung hat aufgrund der Gleitkommazahlen-Präzision der Pixel der Texturen von 24 bit auf der damaligen Generation von GPUs (siehe [NVI05]), einen entscheidenden Nachteil: Bei der Darstellung der Geometrien treten Fehler wie Risse oder Verzerrungen auf.

Da seit 2005 nahezu alle GPUs durchgehend eine Gleitkommazahl-Präzision von 32 bit (siehe [NVI05]) unterstützen, wurde Purcells Ansatz von mehreren Autoren weiterentwickelt.

Karsson und Ljungstedt implementieren in [KL04] einen komplett GPU-basierten Raytracer mit Hilfe der Shader-Sprache Cg und unter Verwendung des regulären Gitters. Aufgrund der Gleitkommazahl-Präzision von 32 bit sind die Rendering-Ergebnisse ihres Verfahrens nicht mehr von denjenigen eines CPU-basierten Renderers zu unterscheiden. Das Verfahren ist jedoch nicht in der Lage interaktive Bildraten zu liefern.

Ein weiterer vollständig GPU-basierter Raytracer wurde von Christen in [Chr05] entwickelt. Er verwendet ebenfalls das reguläre Gitter als Beschleunigungsdatenstruktur, verwendet als Shader-Sprache allerdings HLSL. Die Rendering-Ergebnisse von Christens Raytracer sind ebenfalls von CPU-basierten Render-Ergebnissen nicht zu unterscheiden; auch dieser Raytracer kann keine interaktiven Bildraten erzielen.

Die Verwendung des regulären Gitters als Beschleunigungsdatenstruktur in den vorgestellten Arbeiten ergibt sich aus der Tatsache, dass dieses die einzige Datenstruktur ist, die nicht durch ein rekursives Verfahren traversiert werden muss. Da die GPU keinen Speicher mit wahlfreiem Zugriff besitzt und somit kein Stack verwendet werden kann, ist die Abbildung von rekursiven Verfahren auf der GPU nicht möglich.

Foley und Sugerman entwickeln in [FS05] unter Verwendung einer anderen Beschleunigungsdatenstruktur einen GPU-basierten Raytracer, der in der Lage ist, beinahe interaktive Bildraten zu erzielen. Die zur Traversierung eines kd-Baums vorgestellten Verfahren *kd-Restart* und *kd-Backtrack* benötigen keinen Stack und lassen sich effizient in einer Shader-Sprache implementieren.

Das Problem der Wahl der Beschleunigungsdatenstruktur wird von Thrane und Simonsen in [TS05] weiter analysiert. Die Autoren untersuchen die Traversierung der Datenstrukturen reguläres Gitter, kd-Baum und BVH auf ihre Performance hin und führen ein neues Verfahren zur Traversierung der BVH ein, das ebenfalls ohne einen Stack auskommt. Auf dieses Verfahren wird im späteren Verlauf dieser Arbeit näher eingegangen.

Ein weiterer Ansatz für die Implementierung eines GPU-basierten Raytracers ist der Einsatz von Geometrie-Atlanten, wie ihn Carr et al. in [CHCH06] vorstellen. Dieses Verfahren verspricht sehr gute Ergebnisse bei vergleichbar hohen Bildraten, hat

aber den entscheidenden Nachteil, dass es bei der automatischen Generierung der Geometrie-Atlanten zu erheblichen Artefakten kommt.

3.2.3 Weitere Ansätze

Neben der Implementierung des Raytracing-Verfahrens mit Hilfe von CPU und GPU gibt es weitere Ansätze zur Beschleunigung des Verfahrens.

Woop et al. stellen in [WSS05] eine durch FPGAs[21] realisierte *Ray Processing Unit (RPU)* vor. Diese besitzt mehrere SIMD Prozessoren für die Schnittberechnung und das Shading sowie mehrere Prozessoren zur Traversierung des als Beschleunigungsdatenstruktur verwendeten kd-Baums.

Der auf 66 Mhz getaktete Prototyp erlaubt die Darstellung dynamischer Szenen in einer hohen Auflösung bei interaktiven Bildraten. Die Programmierbarkeit der RPU erlaubt deren Nutzung für das Raytracing und für allgemeine Berechnungen. Ob und wann dieser Spezialprozessor allerdings Marktreife erlangen wird, ist bis jetzt noch unklar.

In [BBDF05] stellen Beck et al. einen hybriden Ansatz zur Implementierung eines Echtzeit-Raytracers vor. Die Autoren verteilen die Rechenlast auf CPU und GPU, was zur Folge hat, dass zu viele Daten über den PCIe-Bus[22] zurückgelesen werden müssen. Die Bandbreite des Busses reicht für eine Erzeugung interaktiver Bildraten nicht aus.

3.3 Zusammenfassung

In diesem Kapitel wurde eine State-of-the-Art Analyse zur Realisierung von Echtzeit-Raytracern durchgeführt. Dabei wurde gezeigt, dass sich die parallele Rechenleistung heutiger Grafikkarten sehr gut für die Berechnungen außerhalb der Rasterisierung nutzen lässt. Es wurde eine Auswahl der bisherigen Arbeiten über Echtzeit-Raytracing vorgestellt und analysiert.

[21] Field Programmable Gate Arrays
[22] Peripheral Component Interconnect Express

4 Anforderungsanalyse

Vor der eigentlichen Implementierung des Raytracers und seiner Integration in die Szenengraph-API muss eine Anforderungsanalyse erstellt werden. Begonnen wird hierbei mit der Spezifikation der Anforderungen an den zu integrierenden Raytracing-Kern, gefolgt von der Spezifikation der Anforderungen an die Szenengraph-API.

4.1 Anforderungen an den Raytracing-Kern

In Kapitel 3.2 wurden die beiden Möglichkeiten für die Realisierung eines Echtzeit-Raytracers vorgestellt: CPU-basierte und GPU-basierte Lösung. Es stellt sich nun die Frage, welcher dieser beiden Ansätze sich am besten für die Integration in einen Szenengraph eignet.

Die in der letzten Zeit auf der CPU entwickelten Echtzeit-Raytracer erzeugen eine enorme Rechenlast auf der CPU. Bei der Integration eines CPU-basierten Raytracers in einen Szenengraphen müssten sich Raytracer und Szenengraph die Rechenleistung des Prozessors teilen, Optimierungen wären nur schwer durchzuführen.

Im Gegensatz dazu stehen die GPU-basierten Echtzeit-Raytracer. Bei der Integration eines GPU-basierten Raytracers wird der größte und aufwendigste Teil des Verfahrens durch die GPU ausgeführt, Szenengraph und Raytracer müssen sich die Rechenleistung der CPU nicht teilen. Des Weiteren sind GPU-basierte Raytracer ein aktuelles Forschungsthema wie beispielsweise in [CHCH06]. Betrachtet man die Leistungssteigerung der Grafikchips in der parallelen Verarbeitung, so erreichte NVIDIA eine Verdreifachung des parallelen Durchsatzes der Pixel-Pipeline innerhalb nur eines Jahres[23].

Zum Rendern dynamischer Szenen muss die Rechenlast gut verteilt werden. Dies kann nur unter Verwendung von GPU-basiertem Raytracing gewährleistet werden.

Nachdem die Entscheidung für GPU-basiertes Raytracing gefallen ist, muss noch geklärt werden, ob es möglich und sinnvoll ist einen bestehenden GPU-basierten Raytracer in einen Szenengraph zu integrieren. Es werden folgende Anforderungen an den Raytracer gestellt:

[23] Vergleich zwischen einer *GeForce 6800GT* mit einem Pixel-Durchsatz von 5.600 MPix/sec aus dem Jahr 2005 und einer *GeForce 7900GTX* mit einem Pixel-Durchsatz von 15.600 MPix/sec aus dem Jahr 2006. Quelle: [NVd06]

- Erreichen möglichst hoher Frame-Raten
- Raytracing dynamischer Szenen
- Refraktion und Reflexion der Strahlen an den Objekten der Szene

Der von Christen in [Chr05] entwickelte Raytracer ist auf statische Szenen beschränkt. Karsson und Ljungstedt entwickelten in [KL04] einen Raytracer, der ebenfalls auf statische Szenen beschränkt ist und implementierungsbedingt vier Frames zum Aufbauen eines Raytracing-Frames benötigt. Der einzige Raytracer, der interaktive Ergebnisse liefern könnte, ist der von Thrane und Simonsen in [TS05] implementierte. Die Autoren haben aber bei ihrer Arbeit den Fokus auf die Evaluierung der Datenstrukturen gelegt und die Erzeugung und Verfolgung von Refraktionsstrahlen außen vorgelassen. Somit erfüllt keine der bisherigen Arbeiten über GPU-basiertes Raytracing alle der oben genannten Anforderungen.

Weiterhin wurden all diese Raytracer auf der damaligen Generation von Grafikchips entwickelt. In dieser Generation wurde erstmals das Shader-Model 3.0 eingeführt, das weitaus komplexere Shader-Programme zuließ als zuvor, jedoch noch erheblichen Einschränkungen unterlag. Bei der aktuellen Generation von Grafikchips konnten diese Einschränkungen weitestgehend aufgehoben werden, so dass die Programmierung von Shadern und vor allem Raytracern auf GPU-Basis wesentlich leichter und effizienter ist.

Daher muss ein eigener GPU-basierter Raytracer entwickelt werden, der zum einen die obigen Anforderungen erfüllt und zum anderen die neusten Shader-Technologien effizient nutzt.

4.2 Anforderungen an die Beschleunigungsdatenstruktur

Eine der Hauptanforderungen an diese Arbeit ist das Raytracing dynamischer Szenen. Die verwendete Beschleunigungsdatenstruktur muss sich daher leicht und möglichst kostengünstig aktualisieren lassen. Da sich weiterhin rekursive Verfahren zur Traversierung der Datenstruktur nicht in einer Shader-Sprache implementieren lassen, muss eine Datenstruktur gewählt werden, deren rekursive Traversierung sich leicht in ein effizientes, iteratives Verfahren umwandeln lässt.

Thrane und Simonsen haben in [TS05] die Datenstrukturen *reguläres Gitter*, *kd-Baum* und *Hüllkörperhierarchie (BVH)* auf ihre Performance bei GPU-basiertem Raytracing untersucht. In ihrer Arbeit stellten sie fest, dass die BVH mit Hilfe eines für die GPU optimierten Traversierungsverfahrens bis zu sieben mal schneller traversiert werden

kann als ihre Konkurrenten. Dieses Verfahren und dessen Erweiterung wird in Kapitel 5.3.4 näher beschrieben.

Bei genauerer Analyse der BVH findet man einen weiteren Vorteil, der bereits von Lext und Akine-Möller in [LAM01] vorgestellt wurde:

Im Gegensatz zu den räumlichen Unterteilungsverfahren wie reguläres Gitter, Kd-Baum oder Octree, ist es bei der BVH möglich, die Datenstruktur für einzelne Objekte getrennt zu behandeln und nur die BVH des Objekts anzupassen, sofern sich an der Transformation des Objekts etwas geändert hat. Dies erspart die explizite Neuerstellung der Datenstruktur der dynamischen Szene in jedem Frame.

Des Weiteren muss die BVH nur bei der Rotation neu erstellt werden, für alle anderen Transformationsoperationen gibt es eine wesentlich günstigere Methode zum Aktualisieren der Datenstruktur. Dieses Thema wird ausführlich in Kapitel 5.3.5 behandelt.

Die Vorteile der BVH liegen auf der Hand, zumal die Datenstruktur alle Anforderungen erfüllt.

4.3 Anforderungen an die Szenengraph-API

Um zu klären, in welche Szenengraph-API sich der Raytracing-Kern am elegantesten integrieren lässt, müssen folgende Anforderungen an die Szenengraph-API definiert werden:

- **Open Source**: Die verwendete API soll als Open-Source-Projekt realisiert sein, um Einblick in den Quellcode des Systems zu erhalten und somit eine Integration zu vereinfachen.
- **Plugin-basiert**: Die in der API enthaltene Funktionalität soll in Plugins gekapselt sein, Erweiterungen des Systems sollen sich wiederum als Plugins realisieren lassen. Dies ermöglicht eine elegante Integration des Raytracers in die Szenengraph-API.
- **Unterstützung der Shader-Sprache Cg**: Die Shader-Sprache Cg soll durch die Szenengraph-API unterstützt werden, dies ermöglicht die spätere Verwendung von *DirectX* und *OpenGL* als Grafik-API.
- **Unterstützung von DirectX 9c und OpenGL 2.0**: Sowohl *DirectX* in der Version 9c als auch *OpenGL* in der Version 2.0 sollen unterstützt werden. Da-

durch wird zum einen die Verwendung von Shader-Model 3.0 möglich, zum anderen die Unabhängigkeit von der verwendeten Grafik-API und somit auch einer gewissen Plattform gewährleistet.

- **Verfügbarkeit auf aktuellen Plattformen**: Die Szenengraph-API soll auf den aktuellen Plattformen wie *Windows*, *Linux* und *Mac OS X* verfügbar sein.

Neben den aufgeführten Anforderungen sollte die API natürlich einen gewissen Abstraktionsmechanismus bieten, um zu vermeiden, dass Befehle der zugrundeliegenden Grafik-API direkt abgesetzt werden müssen.

In Kapitel 2.2 wurde bereits eine Vorauswahl der Szenengraph-APIs getroffen. Vergleicht man nun die Features dieser Szenengraphen mit den obigen Anforderungen, so wird lediglich ein Szenengraph allen Anforderungen gerecht: Das in Kapitel 2.2.3 vorgestellte *OGRE 3D*. Alle anderen Szenengraphen scheiden aufgrund ihrer fehlenden Unterstützung von *DirectX* oder ihres monolithischen Konzepts aus.

Darüber hinaus bietet *OGRE 3D* zwei Konzepte, die sich hervorragend für die Integration des Raytracing-Kerns eignen: Den *SceneManager* und das *Compositor-Framework*. Hierauf wird in Kapitel 5.2 ausführlich eingegangen.

4.4 Zusammenfassung

In diesem Kapitel wurde die Anforderungsanalyse des zu implementierenden Systems durchgeführt. Auf Grund der Parallelverarbeitungsmöglichkeiten aktueller Grafikchips, der Relevanz für die aktuelle Forschung und des vielversprechenden Performance-Zuwachses der GPUs, wurde zugunsten der Implementierung des Raytracing-Kerns auf GPU-Basis entschieden.

Die Wahl der Szenengraph-API fiel aufgrund der umfangreichen Features und der Konzepte des *SceneManagers* und *Compositor-Frameworks* auf *OGRE 3D*.

5 Eigenes Konzept und Implementierung

In diesem Kapitel wird ein neues Konzept für die Integration eines GPU-basierten Raytracers in die Szenengraph-API *OGRE 3D* vorgestellt. Begonnen wird mit dem Grundkonzept des implementierten Systems.

5.1 Grundkonzept des implementierten Systems

Wie in der Anforderungsanalyse in Kapitel 4 dargestellt, ist es sinnvoll, das eigentliche Raytracing auf der GPU durchzuführen, und so die Rechenlast des Szenengraphen und Raytracing-Kerns auf CPU und GPU zu verteilen.

Eines der Kernziele dieser Arbeit ist das Raytracing dynamischer Szenen. Dynamische Szenen werden in einem Szenengraphen durch das Anwenden hierarchischer Transformationen auf die Knoten des Szenengraphen realisiert. Diese Struktur kann bei der Wahl der Beschleunigungsdatenstruktur ausgenutzt werden.

Anstatt eine Datenstruktur für die ganze Szene zu erzeugen und diese in jedem Frame neu zu erstellen, wird für jedes animierbare Objekt der Szene eine Beschleunigungsdatenstruktur erstellt und diese in die Beschleunigungsdatenstruktur der Szene eingefügt. Wird die Transformation eines Objekts geändert, so muss nur die entsprechende Beschleunigungsdatenstruktur aktualisiert werden. Die in Kapitel 2.5.1 vorgestellte Hüllkörperhierarchie (BVH) ist die einzige Beschleunigungsdatenstruktur, die eine einfache Aktualisierung erlaubt.

Der auf der CPU implementierte Teil des Systems übernimmt Erzeugung, Verwaltung und Aktualisierung der BVHs der animierbaren Objekte, sowie das Einfügen dieser BVHs in die BVH der Szene.

Des Weiteren müssen die Daten der Szene für den auf der GPU implementierten Raytracing-Kern von der CPU aufbereitet und übermittelt werden.

5.2 Grundlagen des implementierten Systems

Bevor auf die Implementierung und Integration des Raytracers in die Szenengraph-API näher eingegangen werden kann, ist die Einführung der Grundklassen und Konzepte von *OGRE 3D* notwendig. Eine Übersicht über die Grundklassen von *OGRE 3D* ist Abbildung 6 zu entnehmen.

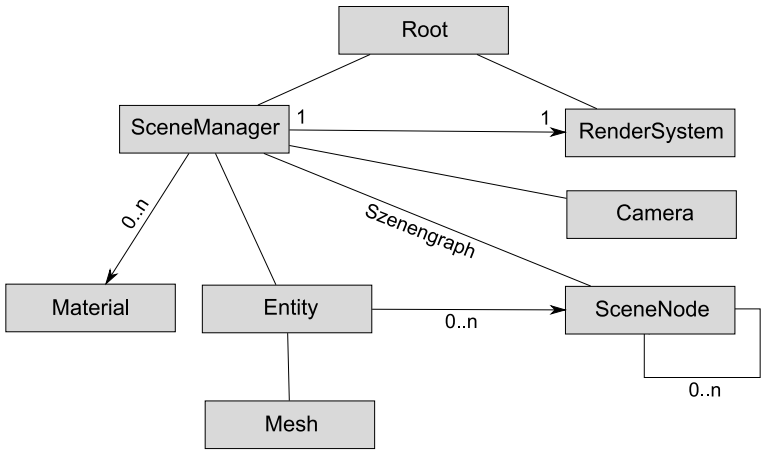

Abbildung 6: Die Grundklassen von OGRE 3D

Der eigentliche Szenengraph wird durch eine Menge von Instanzen der Klasse *SceneNode* repräsentiert, die einen Baum aus Positions- und Orientierungsknoten modellieren. Die Wurzel dieses Baums ist eine Instanz der Klasse *Root*. Zur Abbildung von Bewegungshierarchien werden alle Objekte der Szene an Instanzen der *SceneNode* gebunden.

Instanzen der Klasse *Entity* kapseln die eigentlichen Objekte der Szene. Die Geometriedaten der Objekte, wie Eckpunkt, Normalen und Texturkoordinaten werden durch das *Mesh*-Objekt repräsentiert und an die entsprechende *Entity* gebunden.

Der *SceneManager* verwaltet und aktualisiert den Inhalt der Szene. Er erzeugt und verwaltet sowohl die Instanzen der Klasse *Entity* als auch die von den *Meshes* verwendeten Materialien. In OGRE 3D sind Materialien durch Skripte realisiert, die die Definition von Oberflächenfarben, Reflexionsverhalten, Texturen und Shader-Programmen ermöglichen. Multi-Pass-Materialien werden ebenfalls unterstützt.

Das *Camera*-Objekt modelliert den virtuellen Betrachter der Szene. Der *SceneManager* erhält eine Referenz auf die aktive Kamera und bestimmt mit Hilfe ihrer Sichtpyramide, welche Objekte der Szene durch das *RenderSystem* verarbeitet werden sollen. Das RenderSystem ist für das eigentliche Rendering verantwortlich und übermittelt die Zeichenbefehle an die verwendete Grafik-API *OpenGL* oder *Direct3D*.

Da der *SceneManager* die volle Kontrolle über den Szenengraphen und dessen Objekte hat, liegt es nahe, ihn auch bei der Integration eines Raytracers für die Verwaltung der Szene zu verwenden. Diese umfasst die Erzeugung, Verwaltung und Aktualisierung der Beschleunigungsdatenstruktur mit Hilfe der CPU sowie die sich anschließende Übermittlung der BVH an den auf der GPU implementierten Raytracing-Kern. Die Verwaltung der Szenendaten wird in Abschnitt 5.3 ausführlich behandelt.

Ein erster Ansatz für die Implementierung des Raytracing-Kerns ist die Erstellung einer eigenen Subklasse des *RenderSystems*, in der die gesamte Funktionalität des Raytracers gekapselt wird. Dadurch entstehen mehrere Nachteile. Zum einen verliert man die Unabhängigkeit von den beiden Grafik-APIs, für jede Grafik-API müsste also ein eigenes *RenderSystem* implementiert werden. Zum anderen ist die erfolgreiche Umsetzung dieses Systems nur unter strikter Beibehaltung der Basisklasse *RenderSystem* samt Grafik-APIs gewährleistet.

Neben den Grundklassen enthält OGRE 3D ein weiteres Konzept, das besser zur Implementierung eines Raytracers geeignet ist: Das *Compositor-Framework*.

Das *Compositor-Framework* ermöglicht die Durchführung von Vollbildnachbearbeitungseffekten. Hierbei wird die eigentliche Szene in eine Textur gerendert, die mit Hilfe von Fragment-Shadern entsprechend verändert werden kann. Für die Durchführung komplexer Berechnungen können mehrere dieser *Compositor-Instanzen* hintereinander in einer *Compositorchain* angeordnet werden. Die Compositor-Instanzen können in MRTs schreiben und auch aus ihnen lesen.

OGRE 3D verwendet das Compositor-Framework zur Anwendung von Effekten, wie beispielsweise Weichzeichnern, Bloom oder Ähnlichem auf die gesamte Szene. Die Compositor-Instanzen, die mittels Fragment-Shadern und MRTs realisiert werden, können für die Durchführung allgemeiner Berechnungen verwendet werden, wie sie in Kapitel 3.1.4 vorgestellt wurden. Das Compositor-Framework eignet sich somit auch zur Implementierung eines Raytracers, wie Abschnitt 5.5 zeigen wird.

Neben der Verwaltung der Szenendaten und der Implementierung des eigentlichen Raytracing-Kerns ist es notwendig, die Szenendaten für die GPU entsprechend aufzubereiten. In Abschnitt 5.4 wird hierauf genauer eingegangen.

Die Funktionalität des implementierten Systems wurde in einem Plugin gekapselt, das dynamisch zur Basis von OGRE 3D hinzugeladen werden kann. Das implementierte System trägt den Namen *OgreRT2*. *RT2* ist als RT^2 zu lesen und steht für **R**eal**T**ime **R**ay**T**racing.

5.3 Verwaltung der Szenendaten

Zur Verwaltung der Szenendaten wurde eine Subklasse des *SceneManagers* erstellt.

5.3.1 Der spezielle SceneManager „OgreRT2SceneManager“

Zunächst muss herausgearbeitet werden, welche Methoden des *SceneManagers* durch den *OgreRT2SceneManager* überschrieben werden sollen. Es zeigt sich, dass es ausreicht, die beiden folgenden Funktionen der Basisklasse zu überschreiben:

```
Entity* createEntity(const String &entityName, const
    String &meshName);
void _renderScene(Camera *camera, Viewport *vp, bool
    includeOverlays);
```

Ebenso wie die Methode der Basisklasse erzeugt auch die überschriebene Methode *createEntity* ein Entity-Objekt aus dem übergebenen Mesh. Zusätzlich wird aus den Geometriedaten des Meshes die Beschleunigungsdatenstruktur dieser Entity erzeugt und gespeichert.

Die überschriebene Methode *_renderScene* implementiert das eigentliche Rendern der Szene. Es besteht die Möglichkeit, zwischen dem herkömmlichen OGRE 3D-Renderer und dem Raytracer umzuschalten.

Das Rendern der Szene wird in Kapitel 5.5 ausführlich behandelt.

5.3.2 Die verwendete Beschleunigungsdatenstruktur

Als Beschleunigungsdatenstruktur wird die in Abschnitt 2.5.1 vorgestellte *Hüllkörperhierarchie (BVH)* verwendet, deren Hüllkörper mit Hilfe von achsparallelen Quadern (AABB) realisiert sind.

Da die Struktur eines Szenengraphen bei der Animation von Objekten lediglich hierarchische Transformationen ermöglicht, kann die Grundidee des von Lext und Akenine-Möller in [LAM01] vorgestellten Verfahrens übernommen werden.

Für jedes animierbare Objekt (Entity) wird eine eigene BVH nach dem im nächsten Abschnitt vorgestellten Verfahren erzeugt. Die so erzeugten BVHs der Entities werden in einem Baum angeordnet, der wiederum die BVH der kompletten Szene repräsentiert. Ändert sich etwas an den Transformationen der Objekte, so wird dadurch ermöglicht, nur einzelne Zweige des Baums und somit die BVHs der betroffenen Entities zu aktualisieren, anstatt die Datenstruktur in jedem Frame komplett neu zu erstellen.

5.3.3 Erzeugen der Beschleunigungsdatenstruktur

Die Erzeugung der BVH einer Entity durch den *SceneManager* erfolgt in der Methode *createEntity* des speziellen *OgreRT2SceneManagers*.

Als erstes müssen die Daten des zugrunde liegenden Meshes ausgelesen werden. In OGRE 3D werden Eckpunkte, Normalen und Texturkoordinaten eines Meshes indiziert in Puffern gespeichert. Diese Puffer können eins zu eins auf die auf der GPU vorhandenen Puffer abgebildet werden, was eine effiziente Rasterisierung ermöglicht. Zur Erzeugung der BVH müssen die Indizes, Eckpunkte, Normalen und Texturkoordinaten aus den Puffern gelesen und in temporären Listen (Arrays) gespeichert werden.

Dieses Vorgehen beschränkt das Verfahren auf statische Meshes. In OGRE 3D gibt es die Möglichkeit, durch Vertex-Blending Morph- und Pose-Animationen eines Meshes durchzuführen. Da zur Erzeugung der BVH die Geometriedaten separat gespeichert werden müssen, ist Vertex-Blending zwar nicht verfügbar, es kann aber durch eine Erweiterung implementiert werden.

Die implementierte Klasse *EntityBVHBuilder* unterteilt die Geometrie gleichmäßig und erzeugt rekursiv einen Baum, dessen Knoten und Blätter durch Instanzen der Klasse *EntityBVH* realisiert sind. Das Verfahren zur Erzeugung ist an das von Shirley und Morley [SM03] vorgestellte angelehnt.

In jedem rekursiven Schritt wird zunächst eine Instanz der Klasse *EntityBVH* erstellt, die einen Knoten im BVH-Baum repräsentiert. Dieser Knoten erhält einen Zeiger auf eine Liste von Indizes und einen achsparallelen Hüllquader (AABB), der alle Dreiecke in dieser Indexliste umhüllt. Zur Unterteilung der Indexliste des Knotens wird eine Achse des Mittelpunkts dieser AABB als Pivot gewählt.

Die Achse wird dabei in jedem rekursiven Schritt rotiert. Also gilt: $a_{i+1} = (r_i + 1)\ mod\ 3$, wobei a_{i+1} die Unterteilungsachse für den nächsten rekursiven Schritt und r_i die aktuelle Rekursionsstufe ist. Die Werte $0, 1, 2$ repräsentieren die Achsen x, y, z. Stellt sich heraus, dass diese Achse zur Unterteilung ungünstig ist, wird solange rotiert, bis eine geeignete gefunden wird[24].

Die Elemente der Indexliste werden mit dem Pivotelement verglichen und falls nötig umgeordnet. Anschließend wird der Mittelpunkt der Liste festgestellt. Der linke Teil der Liste enthält die Indizes jener Dreiecke, deren jeweilige Koordinate kleiner ist als das Pivotelement, der rechte Teil die Indizes derjenigen Dreiecke, deren jeweilige Koordinate größer ist als das Pivotelement.

[24] Man stelle sich eine Fläche vor, die plan in der xz-Ebene liegt. Die Unterteilung dieser Fläche entlang der y-Koordinate wäre ungünstig.

Die beiden rekursiven Aufrufe erhalten die linke und rechte Teilliste sowie die nächste Unterteilungsachse.

Dieser Vorgang wird solange fortgesetzt, bis die beiden Teillisten jeweils höchstens noch die Indizes von zwei Dreiecken enthalten. Dann wird eine Instanz der Klasse *EntityBVH* erzeugt, als Blatt markiert und mit den Indizes des jeweiligen Dreiecks versehen.

Dieses Vorgehen erzeugt einen binären Baum. In Pseudocode lässt sich dies wie folgt notieren:

```
function EntityBVH buildTree(Indices[] indices, int axis)
{
  if(nur ein Dreieck in der Liste)
    return EntityBVH(indices[0])
  fi

  if(nur zwei Dreiecke in der Liste)
    return EntityBVH(indices[0], indices[1])
  fi

  //Errechne die Bounding-Box dieses Knotens
  AABB aabb = calculateAABB(indices)
  //Berechne das Pivotelement
  calculatePivot(indices, axis)
  //Unterteile die Indexliste
  splitIndices(indices, axis)

  EntityBVH node

  //Erzeuge rechtes und linkes Kind rekursiv
  node.aabb = aabb
  node.leftChild = buildTree(indices.left,(axis+1) mod 3)
  node.rightChild = buildTree(indices.right,(axis+2) mod
     3)

  return node
}
```

Um bei der späteren Traversierung der BVH die nummerische Stabilität zu gewährleisten, werden die an den Knoten errechneten AABBs in beide Richtungen um $\varepsilon = 3{\cdot}10^{-6}$ ausgedehnt.

Nach der erfolgreichen Erzeugung der BVH der Entity wird diese durch den *EntityBVHBuilder* in ein Array konvertiert. Das Format dieses Arrays entspricht demjenigen, das später auf der GPU verwendet wird.

Der *OgreRT2SceneManager* speichert einen Zeiger auf dieses Array, gleichzeitig speichert er auch die Transformation der Entity unter ihrem Namen.

In Kapitel 2.5.1 wurden die Kriterien für die Erzeugung einer annähernd optimalen BVH von Kay und Kayjiya bereits vorgestellt, von denen das hier vorgestellte Verfahren die Kriterien (i) bis (iii) erfüllt. Die Punkte (iv) und (v) werden nicht berücksichtigt, da eine schnelle Erzeugung der BVH wichtiger ist, wie sich später deutlich zeigen wird.

Das ebenfalls in Kapitel 2.5.1 besprochene Verfahren zur Erzeugung einer BVH nach Kay und Kayjiya unterteilt die Objekte der Szene gleichmäßig, wodurch ein balancierter binärer Baum erzeugt wird. Dies erfordert aber eine Sortierung der kompletten Indexliste nach einer Achse in jedem Schritt. Das hier vorgestellte, an Shirley und Morley angelehnte Verfahren, basiert auf einer gleichmäßigen Unterteilung des Raums und kommt ohne eine Sortierung der Indexliste in jedem Schritt aus. Somit ist es deutlich schneller als das von Kay und Kayjiya entwickelte Verfahren.

5.3.4 Repräsentation der Beschleunigungsdatenstruktur

Die Traversierung der BVH könnte durch ihre objektorientierte Struktur in Form einer rekursiven Tiefensuche auf der CPU implementiert werden. Da sich aber der Raytracing-Kern auf der GPU befindet, muss die Traversierung ebenfalls durch die GPU durchgeführt werden. Eine rekursive Traversierung auf der GPU ist aufgrund des fehlenden Stacks nicht möglich. Des Weiteren kann die objektorientierte Struktur der BVH nicht im Texturspeicher der Grafikkarte abgelegt werden.

Es wird also ein Verfahren zur iterativen Traversierung einer Repräsentation der BVH benötigt, das für die GPU geeignet ist.

In [Ter01] und in [TS05] wird eine Kodierung der BVH vorgestellt, die eine effiziente iterative Traversierung durch die GPU erlaubt.

Die Basis dieser Kodierung bildet die Durchnummerierung der Knoten der BVH entsprechend ihrer Traversierungsreihenfolge, und ihre entsprechende Speicherung in einer Liste. Diese erspart die explizite Speicherung der Zeiger auf das rechte und linke Kind eines Knotens.

Wird die AABB eines Knotens verfehlt, so muss die Traversierung am nächsten rech-

ten Zweig der BVH neu ansetzen. Zu diesem Zweck wird zusammen mit jedem Knoten ein *Escape-Index* gespeichert, der den Index des Knotens enthält, an dem die Traversierung neu ansetzen soll, für den Fall, dass die AABB des Knotens verfehlt wird. Gibt es keinen rechten Zweig, wird der Escape-Index auf n gesetzt, wobei n die Länge der Liste ist.

Die so erhaltene Kodierung der BVH ist in Abbildung 7 dargestellt. Der obere Teil zeigt eine BVH, deren Knoten und Blätter entsprechend der Traversierungsreihenfolge nummeriert sind. Die gestrichelten Pfeile stellen die Escape-Indizes dar. Dem unteren Teil der Abbildung kann die Kodierung dieser BVH in einer Liste entnommen werden.

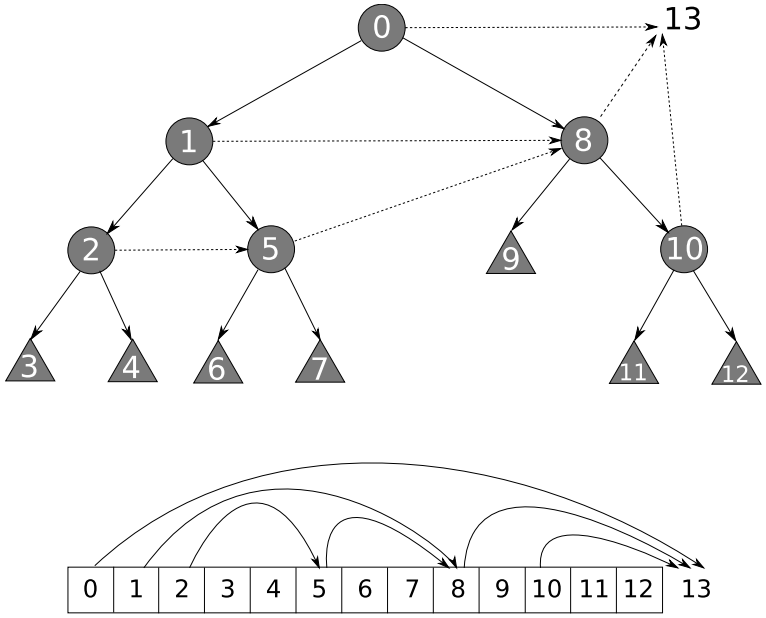

Abbildung 7: Kodierung der BVH

Diese Repräsentation der BVH ist effizient iterativ traversierbar. In Pseudocode kann die Traversierung der BVH wie folgt notiert werden (angelehnt an [TS05]).

```
BVH[] bvh
Ray r
int i=0
EntityBVH node
IntersectionPoint p

while i < bvh.size() do
  node = bvh[i]
  //Wir sind an einem Knoten
  if(node.type == AABB)
```

```
      if(intersects(r, node))
        i++
      else
        i = bvh[i].escape
      fi
    //Wir sind an einem Blatt
    else
      p = calculateIntersection(r,node)
      if(p != NULL)
        saveIntersectionPoint(p)
      else
        i++
      fi
    fi
  od
```

Mit Hilfe der Kodierung kann die BVH im Texturspeicher der GPU abgelegt werden. Das Verfahren kann mit Hilfe eines Fragment-Shaders umgesetzt werden, da kein Stack benötigt wird.

5.3.5 Verwaltung und Aktualisierung der Beschleunigungsdatenstruktur

Wird die Methode *_renderScene* des *OgreRT2SceneManagers* aufgerufen, werden zunächst eventuell ausstehende Animationen (Translation, Rotation und Skalierung) auf die SceneNodes des Szenengraphen übertragen. Danach beginnt die Traversierung des Szenengraphen an dessen Wurzel.

Jede SceneNode wird nach Entities abgesucht. Wird eine Entity gefunden, so wird die unter dem Namen der Entity gespeicherte Transformation mit der anzuwendenden Transformation der SceneNode verglichen. Sind die Transformationen identisch, kann die betreffende BVH der Entity direkt in die BVH der Szene eingefügt werden.

Sind sie hingegen verschieden, müssen folgende Fälle unterschieden werden:

Translation

Die Position einer Entity hat sich verändert, was auch eine Verschiebung der BVH der Entity zur Folge hat. Auf die Struktur der BVH hat dies aber keine Auswirkung, da die achsparallele Ausrichtung der AABB weiterhin gewährleistet ist. Dementsprechend

müssen die Elemente der EntityBVH (Eckpunkte der Dreiecke und Eckpunkte der AABBs) beim Einfügen in die BVH der Szene mit der neuen Translationsmatrix multipliziert werden.

Skalierung

Die Entity wurde skaliert. Da die achsparallele Ausrichtung der AABBs, unverändert bleibt müssen die Elemente der BVH beim Einfügen in die BVH der Szene entsprechend dem neuen Wert skaliert werden. Zusätzlich ist es erforderlich, die Normalen des zugrunde liegenden Meshes der Entity ebenfalls zu skalieren.

Rotation

In diesem Sonderfall ändert sich die Orientierung der Entity, so dass die achsparallele Ausrichtung der AABBs der BVH ist nicht mehr gewährleistet. Die BVH muss gemäß der aktuellen Orientierung der Entity neu erstellt werden. Dazu wird die EntityBVH dieser Entity gelöscht und durch den EntityBVHBuilder neu aufgebaut. Der SceneManager erhält den Zeiger auf das Array der EntityBVH und fügt sie in die BVH der Szene ein.

Es kann vorkommen, dass sich die AABBs der Objekte in der Szene überlappen. Trifft ein Strahl beispielsweise die AABB von Objekt A, jedoch keines der in ihr eingeschlossenen Primitive, würde das Verfolgen des Escape-Pointers zum Abbruch der Traversierung führen, obwohl die Schnittpunktsuche eigentlich an der BVH von Objekt B hätte fortfahren müssen, dessen AABB die des Objekts A überlappt.

Sofern sich die AABBs der Objekte überlappen, müssen beim Einfügen der BVH einer Entity in die BVH der Szene die Escape-Indizes von Objekt A auf den Index der Wurzel der BVH von Objekt B geändert werden.

Auf die Aktualisierung und das Einfügen der BVH in die BVH der Szene folgt die Speicherung der neuen Transformation der Entity durch den *OgreRT2SceneManager*.

5.4 Aufbereitung der Szene für die GPU

Wie Abschnitt 5.3.4 zeigt, ist es notwendig, die Daten der Szene so aufzubereiten, dass sie im Texturspeicher der Grafikkarte abgelegt werden können.

Aktuelle Generationen von GPUs und Fragment-Shadern unterstützen eine Vielzahl von Texturformaten. Dazu zählt auch das 32 bit RGBA-Format, das pro Pixel vier Gleitkommazahlen mit einer Präzision von je 32 bit speichern kann. Obwohl RGBA-Farbwerte eigentlich aus $[0, 1]^4$ stammen, erlaubt dieses Format das Speichern von beliebigen Werten aus $\mathbb{R}^4$ mit einer Präzision von 32 bit. Es eignet sich somit für die Speicherung allgemeiner Daten im Texturspeicher.

Für das Raytracing auf der GPU müssen folgende Daten in Texturen gespeichert sein:

- Geometriedaten für Traversierung und Schnittberechnung
- Materialdaten für Shading und Sekundärstrahlenerzeugung
- Strahlursprünge und Richtungen
- Beste Treffer für Schnittberechnung und Shading
- Positionen der Lichtquellen der Szenen

Jeder Fragment-Shader hat die Möglichkeit, zweidimensionale Texturen wie ein eindimensionales Array zu behandeln und anhand eines Index daraus zu lesen.

Auf den Aufbau dieser Texturen wird in den nun folgenden Abschnitten eingegangen.

5.4.1 Geometriedaten

Die Textur der Geometriedaten speichert die BVH der Szene gemäß der in Abschnitt 5.3.4 vorgestellten Repräsentation.

Dabei muss zwischen einem Knoten und einem Blatt unterschieden werden. Ein Knoten speichert den minimalen und maximalen Eckpunkt der zugehörigen AABB und den Escape-Zeiger. Um seinen Datenblock in der Textur später von demjenigen eines Blattes unterscheiden zu können, muss ein Knoten als solcher gekennzeichnet werden.

Es werden also zwei RGBA-Werte für das Speichern eines Knotens benötigt.

Ein Blatt speichert die drei Eckpunkte eines Dreiecks und den Index des verwendeten Materials. Die Speicherung eines Dreiecks erfolgt in drei RGBA-Werten. Der Index

gibt die Position des Materials in der Materialtextur an, die im nächsten Abschnitt vorgestellt wird. Auch hier wird der Typ des Datenblocks gespeichert.

Der Aufbau dieser Geometrietextur ist in Abbildung 8 dargestellt.

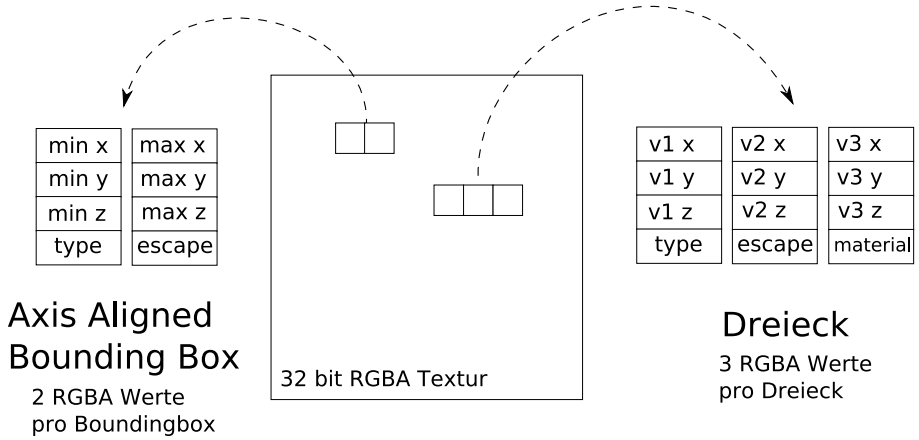

Abbildung 8: Aufbau der Geometrietextur (angelehnt an [TS05])

OGRE 3D ermöglicht den direkten Zugriff auf einen Pixelpuffer der GPU. Beim Rendern der Szene durch die Methode *_renderScene* wird nach dem Erzeugen der BVH der Szene deren Inhalt nach obigem Aufbau über einen Pixel-Puffer in eine Textur geschrieben. Das Schreiben in diesen Puffer ist für das Aktualisieren in jedem Frame optimiert und läuft in hoher Geschwindigkeit ab.

5.4.2 Materialdaten

Die Materialien werden ebenfalls in einer Textur gespeichert, die folgende Werte des Materials enthalten muss:

- Die Normalen n_1, n_2, n_3 der Eckpunkte
- Die Texturkoordinaten t_1, t_2, t_3 der Eckpunkte
- Die diffuse Farbe (RGB)
- Die spiegelnde Farbe (RGB)
- Die Umgebungsfarbe (RGB)
- Den Grad der Verspiegelung

- Ein Markierungszeichen, das deutlich macht, ob es sich um eine durchsichtige Oberfläche handelt

Diese Daten können in sieben RGBA-Werten gespeichert werden. Der Aufbau eines Blocks der Materialtextur kann Abbildung 9 entnommen werden.

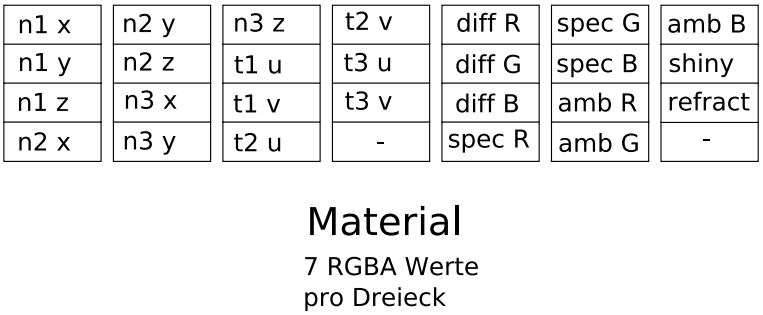

Abbildung 9: Aufbau eines Blocks der Materialtextur

Die Materialtextur wird nach dem Erzeugen der BVH der Szene ebenfalls in den Texturspeicher der Grafikkarte geladen.

5.4.3 Primär- und Sekundärstrahlen

Für die eigentliche Verfolgung von Primär- und Sekundärstrahlen müssen auch Strahlrichtungen und Ursprünge in Texturen abgelegt werden. Da die Abtastung der Szene mit einem Strahl pro Pixel erfolgt, werden die Ursprünge und Richtungen der Strahlen in je einer Textur in der Ausgabeauflösung gespeichert.

Das Speichern der x-, y- und z-Koordinaten der Strahlrichtung erfolgt im RGB-Wert des Pixels der Strahlenrichtungstextur. Der Alpha-Wert des Pixels gibt Auskunft über den Typ des Strahls. Der Wert 0 entspricht einem Reflexionsstrahl, 1 einem Schattenstrahl. Die Refraktion erfordert den Wert 2 für einen Strahl, der außerhalb einer transparenten Geometrie in das Innere führt. Der Wert 3 wird für einen Strahl, der aus dem Inneren nach außen zeigt, verwendet.

Im Gegensatz zur Geometrie- und Materialtextur werden die Strahlen nicht durch den *OgreRT2SceneManager* erzeugt, sondern durch einen Fragment-Shader des Raytracing-Kerns. Hierauf wird in Kapitel 5.5 eingegangen.

5.4.4 Beste Treffer

Die besten Treffer der Schnittberechnungen werden ebenfalls in einer Textur der Größe der Ausgabeauflösung abgelegt. Die R- und G-Werte eines Pixels enthalten zwei der baryzentrischen Koordinaten eines Schnittpunktes. Im B-Wert des Pixels wird die Distanz zwischen Strahlursprung und Schnittpunkt gespeichert.

Der Alpha-Wert des Pixels speichert den Index des Materials des getroffenen Dreiecks oder den Wert -1, falls kein Dreieck getroffen wurde.

Die Textur der besten Treffer wird durch einen Fragment-Shader des Raytracing-Kerns befüllt.

5.4.5 Lichtquellen

Das Speichern der Positionen der Lichtquellen einer Szene erfolgt ebenfalls in einer Textur. Die RGB-Werte der Pixel speichern die x-, y- und z-Koordinaten der Lichtquellen. Die Textur wird durch den *OgreRT2SceneManager* während der Erzeugung der Textur der BVH der Szene befüllt.

5.5 Der Raytracing-Kern

In diesem Kapitel wird der GPU-basierte Raytracing-Kern des implementierten Systems vorgestellt.

In Abschnitt 3.2.2 wurde Purcells Ansatz zur Implementierung eines GPU-basierten Raytracers durch Unterteilung des Verfahrens in die Kernel *Primärstrahlenerzeugung*, *Traversierung der Datenstruktur*, *Schnittberechnung* und *Shading* vorgestellt.

In der von Purcell verwendeten Generation von GPUs und Shader-Sprachen sind While-Schleifen auf 256 Iterationen beschränkt. darüber hinaus darf die Länge des kompilierten Programms 65535 Zeichen nicht übersteigen. Diese Beschränkung hat zu Folge, dass beispielsweise der Kernel zur Traversierung der Datenstruktur mehrmals ausgeführt werden muss, um die Datenstruktur komplett zu traversieren.

Da aktuelle Generationen von GPUs und Shader-Sprachen Fragment-Shader-Programme von beliebiger Länge und While-Schleifen beliebiger Größe erlauben, kann die von Purcell vorgeschlagene Unterteilung weiter vereinfacht werden. Die Kernel für die Traversierung der Datenstruktur und die Schnittberechnung können in einem Kernel vereint werden. Das Gleiche gilt für die Schnittberechnung und das Shading.

Der Aufbau der Kernels des implementierten Raytracing-Kerns kann Abbildung 10 entnommen werden.

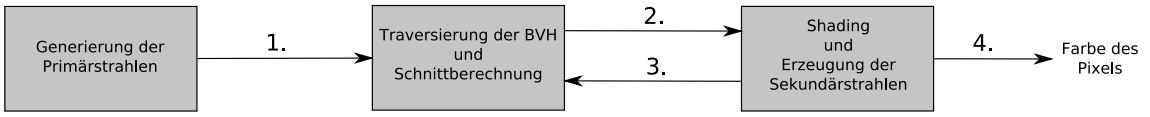

Abbildung 10: Die Kernels des Raytracers

Abhängig von der gewählten Anzahl an Iterationen findet die eigentliche Iteration des Verfahrens in Schritt 2 und 3 statt.

Zur Vereinfachung der Implementierung von allgemeinen Berechnungen auf der GPU existieren spezielle Bibliotheken wie die in [BFH+04] vorgestellte *„Brook für GPUs“-Bibliothek* oder die in [MFM04] vorgestellte *Sh-Bibliothek*. Diese wurden aus zwei Gründen nicht für die Implementierung des Raytracing-Kerns verwendet. Zum einen ist es erforderlich, auf die hardwarenahen Befehle der Implementierung direkten Zugriff zu haben, die sonst durch die Verwendung diese Bibliotheken abstrahiert werden würden. Zum anderen würde die Integration der Bibliotheken in OGRE 3D zu ungewollten Abhängigkeiten führen.

In Anbetracht dessen erfolgt die Implementierung allgemeiner Berechnungen mit Hilfe des Compositor-Frameworks. Die Kernel in Abbildung 10 wurden als Instanz eines Compositors realisiert. Die zugehörigen Fragment-Programme wurden mit Hilfe der Shader-Sprache *Cg* implementiert.

Die Initialisierung der MRTs, die von den Compositor-Instanzen als Ein- und Ausgabe verwendet werden, erfolgt durch den *OgreRT2SceneManager*. Die Erzeugung und Registrierung der Compositor-Instanzen selbst wird ebenfalls durch den *OgreRT2SceneManager* vorgenommen. Die Compositor-Instanzen sind in einer Compositor-Chain angeordnet, die aktiviert und deaktiviert werden kann. Ist sie deaktiviert, übernimmt der Standardrenderer von OGRE 3D das Rendern der Szene.

Die Compositor-Chain wird so vorkonfiguriert, dass sechs Iterationen des Verfahrens durchgeführt werden können. Standardmäßig sind drei der sechs Iterationen aktiv. Die Anzahl der Iterationen lässt sich zur Laufzeit zwischen eins und sechs über das Interface des *OgreRT2SceneManager* verändern.

Neben den MRTs zur Ein- und Ausgabe benötigen die Fragment-Shader der Compositor-Instanzen Parameter. Die sogenannten *Uniform Parameter* werden durch den *OgreRT2SceneManager* in jedem Frame an den jeweiligen Compositor übergeben. Die einzelnen Parameter und die Implementierung der Kernels wird in den folgenden Abschnitten erläutert.

5.5.1 Primärstrahlenerzeugung

Dieser Kernel ist der erste in der Compositor-Chain, und wird jedes Frame einmal aufgerufen. Er erzeugt für jeden Pixel der Ausgabeauflösung einen Primärstrahl und schreibt die Richtungen und Ursprünge der Strahlen in jeweils eine Textur. Der Fragment-Shader dieses Kernels efordert folgende Parameter, die durch den *OgreRT2-SceneManager* übermittelt werden:

- Den Richtungsvektor der Kamera, der nach rechts zeigt
- Den Richtungsvektor der Kamera, der nach oben zeigt
- Den Vektor zur oberen, linken Ecke der Bildebene der Sichtpyramide der Kamera
- Die Position der Kamera
- Die Breite eines Pixels der Ausgabeauflösung

Die Position der erzeugten Primärstrahlen entspricht der aktuellen Kameraposition. Die Richtung der Strahlen kann aus den obigen Parametern mit Hilfe der Texturkoordinaten der Fragmente interpoliert werden.

5.5.2 Traversierung der Datenstruktur und Schnittberechnung

Die Traversierung der BVH der Szene und die Schnittberechnung wird durch diesen Kernel vorgenommen. Für die Traversierung der BVH wird das in Abschnitt 5.3.4 erarbeitete Verfahren angewendet.

Der Kernel erhält die Textur der BVH der Szene, die Texturen der Strahlrichtungen sowie die Strahlursprünge als Eingabe. Ob ein Strahl die AABB eines Knotens schneidet, wird anhand des in Definition 2.8.3 eingeführte Verfahren ermittelt. Gelangt die Traversierung der BVH an ein Blatt und somit an ein Dreieck, so wird der Schnittpunkt gemäß Definition 2.8.2 berechnet.

Für jeden Pixel wird so ein bester Treffer ermittelt, der die Koordinaten des Schnittpunktes enthält; vorausgesetzt, für den Strahl dieses Pixels wird ein Dreieck getroffen. Andernfalls wird an dieser Stelle der Textur mit den besten Treffern festgehalten, dass es für diesen Pixel keinen Treffer gibt. Dieser Pixel wird von der weiteren Bearbeitung komplett ausgeschlossen.

Eine Übersicht des Kernels ist Abbildung 11 zu entnehmen.

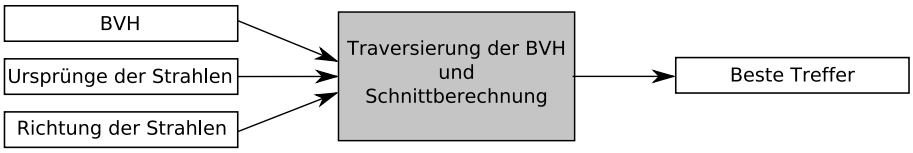

Abbildung 11: Kernel zur Traversierung der Datenstruktur und Schnittberechnung

5.5.3 Shading und Sekundärstrahlenerzeugung

Dieser Kernel ist für das Shading und die Erzeugung der Sekundärstrahlen zuständig. Als Eingabe dienen die mittels des vorangegangenen Kernels berechneten besten Treffer der Pixel, die Richtungen und Ursprünge der Strahlen, die Materialien der Szene, die in der Szene verwendeten Lichter sowie der Farbpuffer der vorherigen Iteration.

Der Kernel speichert die Richtungen und Ursprünge der generierten Sekundärstrahlen in jeweils einer Textur. Des Weiteren werden die, durch das Shading errechneten, Farbwerte eines Pixels in einer Textur abgelegt.

Abbildung 12 zeigt eine Übersicht des Kernels.

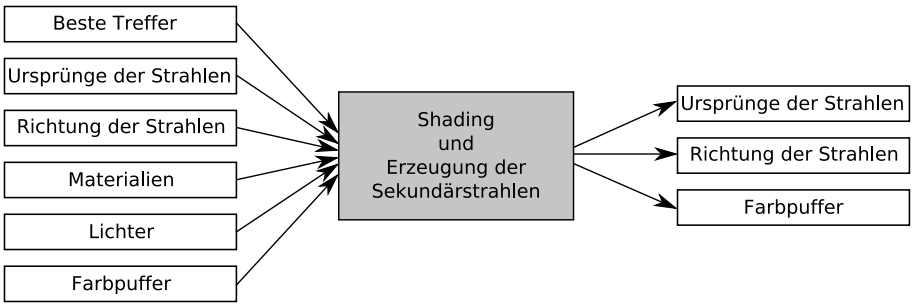

Abbildung 12: Kernel für Shading und Sekundärstrahlenerzeugung

Der Fragment-Shader dieses Kernels ermittelt zunächst das Material des für einen Pixel getroffenen Dreiecks, indem er die Materialeigenschaften aus der Materialtextur liest. Anschließend wird für jede Lichtquelle der Anteil der lokalen Beleuchtung gemäß des in Definition 2.8.6 vorgestellten Verfahrens errechnet. Wird kein Dreieck getroffen, oder handelt es sich bei dem eingehenden Strahl um einen Schattenstrahl, der die Lichtquelle nicht getroffen hat, so wird der Pixel schwarz gefärbt.

Die so ermittelte Farbe des Pixels wird mit dem Inhalt des Farbpuffers der vorherigen Iteration an der vom Typ des getroffenen Materials abhängigen Stelle kombiniert und in die Ausgabetextur geschrieben.

Im nächsten Schritt werden Sekundärstrahlen erzeugt. Deren Ursprung entspricht dem Schnittpunkt zwischen dem Strahl der vorherigen Iteration und dem Dreieck.

Die Richtung der Sekundärstrahlen wird abhängig vom Typ des Materials des getroffenen Dreiecks berechnet. Handelt es sich um ein spiegelndes Material, wird die Richtung des Strahls gemäß den Reflexionsgesetzen erzeugt. Ist das Material transparent, wird die Richtung des Strahls gemäß den Brechungsgesetzen erzeugt. Ferner wird festgehalten, ob der Strahl von außen in ein Objekts eindringt oder vom Inneren des Objekts nach außen tritt. Dies ist für die Berechnung des Brechungsindex erforderlich. Des Weiteren müssen die Richtungen der Normalen des Dreiecks beim Berechnen der Farbe umgekehrt werden, sollte der Strahl von Innen nach außen führt.

Wird ein diffuses Material getroffen, so zeigt die Richtung des Strahls auf die erste Lichtquelle der Szene.

5.6 Testapplikationen

Um das implementierte System testen zu können, wurden mehrere Testapplikationen erstellt. Diese gliedern sich in Tests für das Raytracing statischer, dynamischer und interaktiver Szenen.

In statischen Szenen bleibt die Transformation der Geometrien der Szene gleich, nur die Kamera ist frei beweglich. In dynamischen Szenen dagegen verändert sich die Transformation mindestens eines Objekts über die Zeit. Die Kamera kann hier ebenfalls frei bewegt werden. Interaktive Szenen bieten dem Benutzer die Möglichkeit, neben der Kamera auch die Objekte der Szene frei zu bewegen.

5.6.1 Raytracing statischer Szenen

Zum Testen statischer Szenen wurden zwei Testapplikationen implementiert. Die in der ersten Applikation verwendete Szene besteht aus vier spiegelnden Kugeln auf einem spiegelnden Schachbrett. Die Szene der zweiten Applikation beinhaltet eine reduzierte Version des *Stanford Bunnys* auf einem spiegelnden Schachbrett.

Screenshots der beiden Szenen können Abbildung 13 und 14 entnommen werden.

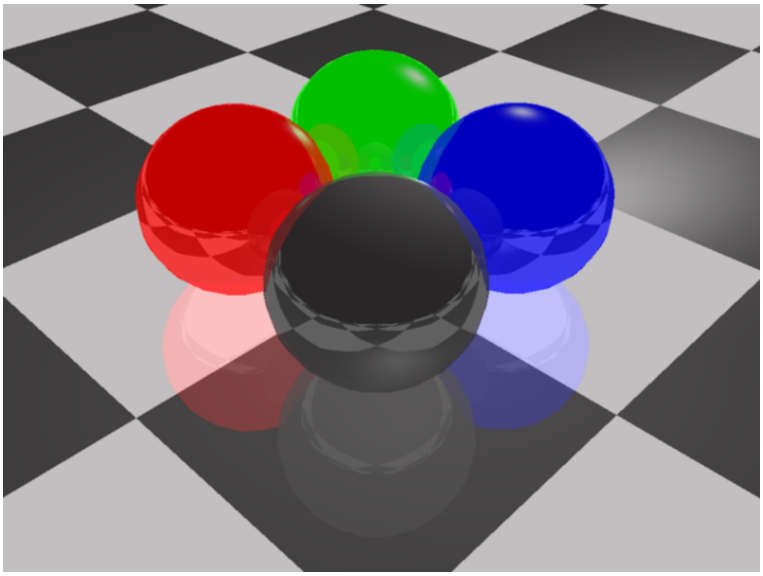

Abbildung 13: Testszene Nr. 1 (statisch)

5.6.2 Raytracing dynamischer Szenen

Für das Testen dynamischer Szenen wurden ebenfalls zwei Testapplikationen entwickelt. Die Szene der ersten Applikation besteht aus einem spiegelnden *Utah Teapot*, der über einem statischen, spiegelnden Schachbrett kreist.

Ein Screenshot dieser Szene ist Abbildung 15 zu entnehmen.

Abbildung 16 zeigt einen Screenshot der zweiten Applikation. Die gläserne Kugel bewegt sich in x-Richtung hin und her. Der gelbe, spiegelnde Würfel wird in y-Richtung skaliert.

5.6.3 Raytracing interaktiver Szenen

Um auch Szenen mit nicht fest vorgegebener Animation testen zu können, wurde diese Testapplikation interaktiver Szenen implementiert. Der Aufbau der Szene entspricht der in Abschnitt 5.6.1 vorgestellten ersten Applikation. Der Benutzer hat hier allerdings die Möglichkeit, die Kugeln der Szene mit Hilfe der Pfeiltasten frei zu bewegen.

Abbildung 17 zeigt einen Screenshot der Szene, nachdem die Positionen aller vier Kugeln verändert wurden.

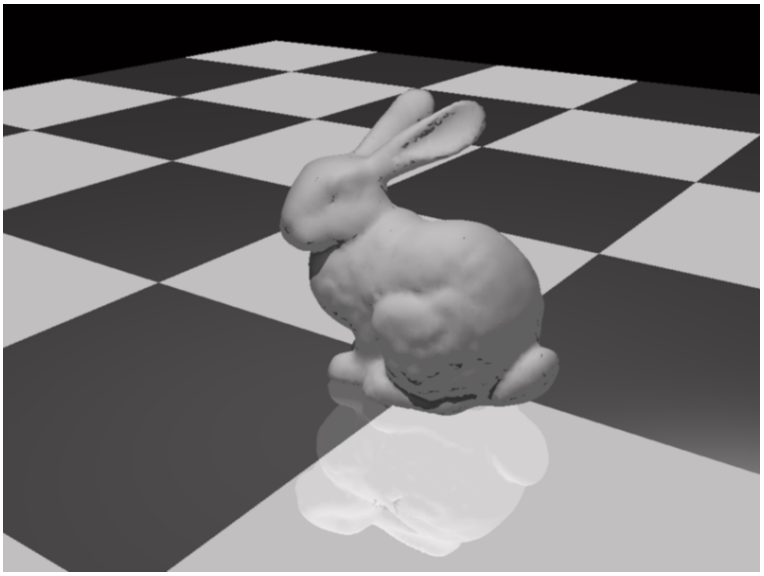

Abbildung 14: Testszene Nr. 2 (statisch)

5.7 Zusammenfassung

In diesem Kapitel wurde das neue Verfahren für die Integration eines Echtzeit-Raytracers in die Szenengraph-API OGRE 3D vorgestellt. Die Verwendung der BVH ermöglicht die Aktualisierung der Datenstruktur bei Skalierung und Translation. Einzig bei der Rotation der Objekte müssen die betreffenden BVHs neu erstellt werden.

Der implementierte GPU-basierte Raytracer verwendet eine Erweiterung des von Thrane und Simonsen entwickelten Verfahrens zur Traversierung der BVH. Durch die Neuerungen in den aktuellen Shader-Sprachen konnte die Implementierung des Raytracers effizienter gestaltet werden als in vorangegangenen Arbeiten.

Die Kapselung des implementierten Systems in einem Plugin erlaubt die leichte Einarbeitung zukünftiger Erweiterungen. Des Weiteren sind keine Änderungen am eigentlichen Kern von OGRE 3D erforderlich.

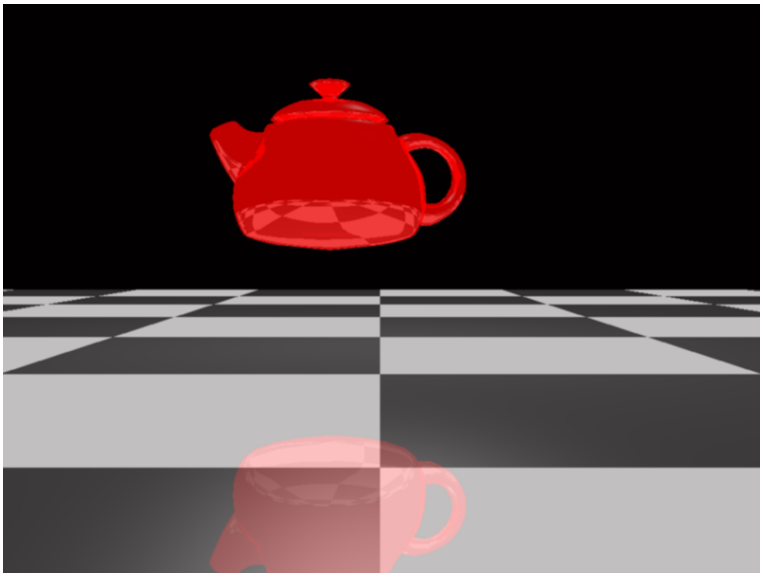

Abbildung 15: Testszene Nr. 3 (dynamisch)

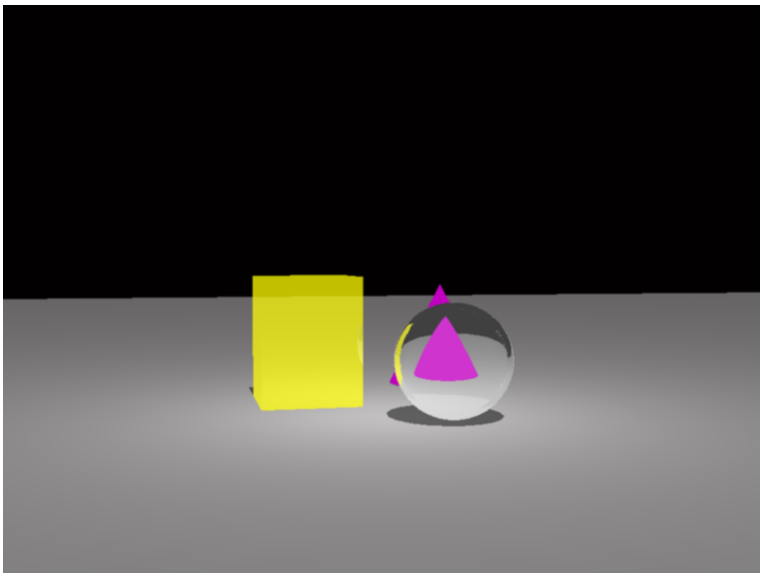

Abbildung 16: Testszene Nr. 4 (dynamisch)

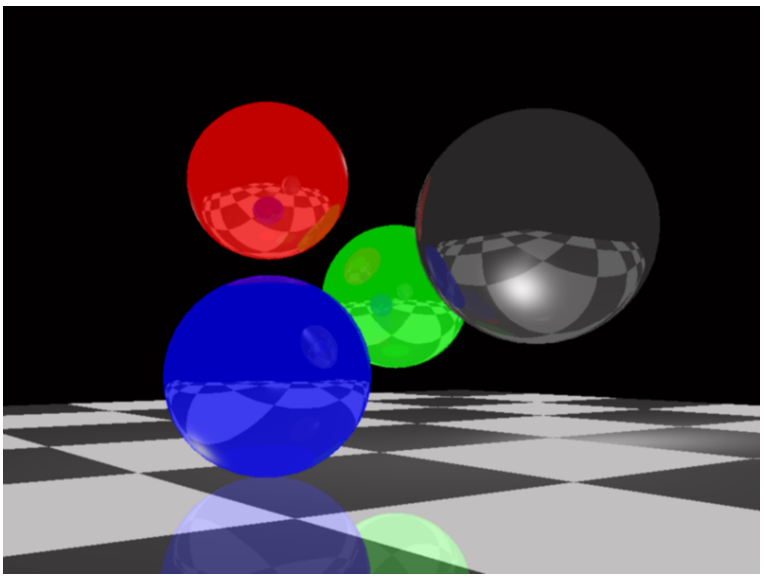

Abbildung 17: Testszene Nr. 5 (interaktiv)

6 Evaluation

In diesem Kapitel wird eine Evaluation des implementierten Systems durchgeführt.

Um die Ergebnisse von Echtzeit-Raytracern beurteilen und vergleichen zu können, wurde in [LAM00] das *BART-Toolkit* (**B**enchmarks for **A**nimated **R**ay **T**racing) vorgestellt. Dieses erlaubt es, mittels fest vorgegebener, animierter Szenen Benchmarks eines Raytracers durchzuführen. Der Nachteil des BART-Toolkits liegt darin, dass die Szenen und Animationen in einem eigenen Format gespeichert und durch einen in C implementierten Loader geladen werden müssen. Dieser Loader müsste mit erheblichem Aufwand in OGRE 3D integriert werden, um die Testszenen laden zu können.

Stattdessen wurden die Benchmarks des implementierten Systems anhand der in Kapitel 5.6 vorgestellten Testapplikationen durchgeführt.

Alle Messungen wurden auf dem System „Adrienne“ vorgenommen. Dieses System verfügt über einen *AMD Athlon 64 X2 Dual Core 4200+* Prozessor, 1 GB DDR2 RAM sowie eine *GeForce 7900GTX* GPU mit 512 MB Texturspeicher. Als Betriebssystem dient *Windows XP SP 2* mit einem *ForceWare* Treiber für die Grafikkarte in der Version 91.47.

Tabelle 1 sind die bei einer Auflösung von 512×512 und einer permanenten Bewegung der Kamera erzielten Messergebnisse zu entnehmen.

Testszene	Typ	Polygone	fps
1	statisch	6.540	5,7
2	statisch	33.066	2,8
3	dynamisch	2224	2,6
4	dynamisch	1008	10,2
5	interaktiv	6.540	4,7

Tabelle 1: Messergebnisse

Die Messergebnisse zeigen, dass statische und dynamische Szenen, in denen keine Rotation durchgeführt wird, bei interaktiven Bildraten verarbeitet werden können. Bei dynamischen Szenen, in denen eine Rotation durchgeführt wird, sinkt die Bildrate hingegen erheblich.

Es ist daher anzunehmen, dass die Neuerstellung der BVH bei einer Rotation den Flaschenhals des Verfahrens darstellt, und dass die GPU somit auf die Daten von der CPU warten muss, bevor das nächste Bild berechnet werden kann.

Das NVPerfKit[25] erlaubt es, diverse Performance-Daten der GPU auszulesen, darunter auch die Zeit, die die GPU mit dem Warten auf CPU-Daten verbringt (Idle-Time).

Bedauerlicherweise kann das NVPerfKit nur unter DirectX verwendet werden. Das implementierte System benötigt unter DirectX das Compiler-Profil ps3.0, das erst im NVIDIA Cg-Compiler 1.5 verfügbar ist. Diese Version befindet sich zur Zeit noch im Beta-Stadium. Der Versuch, den in OGRE 3D beinhalteten Cg-Compiler 1.42 durch die Beta-Version auszutauschen, führte zu Abstürzen, deren Ursache auf einen Fehler im Cg-Compiler selbst zurückzuführen ist.

Somit kann das implementierte System zur Zeit nur unter Verwendung von OpenGL genutzt werden. Mit der zu erwartenden Finalisierung des Cg-Compilers wird auch die Verwendung von DirectX als auch die Analyse mit Hilfe des NVPerfKit möglich sein.

Neben den Messergebnissen ist es interessant zu erfahren, wie viele Dreiecke durch das System verarbeitet werden können.

Die verwendete GPU hat einen Texturspeicher von 512 MB. Ein Pixel des zur Speicherung der benötigten Daten verwendeten 32 bit RGBA-Formats benötigt 128 bit. Somit können $3,2 \cdot 10^7$ Pixel dieses Formats im Speicher der Grafikkarte abgelegt werden.

Sei $B(n)$ der Speicherbedarf der BVH und $M(n)$ der Speicherbedarf der Materialien eines Modells mit n Dreiecken. Für den Gesamtspeicherbedarf $G(n)$ gilt:

$$G(n) = B(n) + M(n) \tag{9}$$

Die BVH eines Modells mit n Dreiecken benötigt drei Pixel für einen Knoten und zwei Pixel für jedes Dreieck. Man erhält für den Speicherbedarf der BVH:

$$B(n) = 2(2n - 1 - n) + 3n = 5n - 2 \tag{10}$$

Für jedes Dreieck werden sieben Pixel zur Speicherung der Materialdaten benötigt, für den Speicherbedarf der Materialien gilt also:

$$M(n) = 7n \tag{11}$$

Der Gesamtspeicherbedarf errechnet sich aus:

$$G(n) = B(n) + M(n) = 12n - 2 \tag{12}$$

Die maximale Anzahl an Dreiecken, deren Daten in 512 MB Texturspeicher abgelegt werden können, beläuft sich somit auf $3,2 \cdot 10^7$. Da das Verfahren zur Durchführung

[25] http://developer.nvidia.com/object/nvperfkit_home.html

aber sieben MRTs in der Ausgabeauflösung benötigt, stehen nicht die vollen 512 MB zur Speicherung der Daten der Dreiecke zur Verfügung.

Rechnet man mit einer Ausgabeauflösung von 512×512, so bleiben zum Speichern der Dreiecke nur noch $3,06 \cdot 10^7$ Pixel. Daraus ergibt sich eine maximale Anzahl von $2,55 \cdot 10^6$ Dreiecken, die durch das System bei einer Ausgabeauflösung von 512×512 verarbeitet werden können.

Bei einer Ausgabeauflösung von 1024×1024 beläuft sich die maximale Anzahl an Dreiecken auf $2,05 \cdot 10^6$.

Der Raytracing-Kern des implementierten Systems unterstützt zur Zeit keine Texturen. Deren Verwendung kann aber durch eine Erweiterung implementiert werden. Dazu müssen alle Texturen in einer einzelnen oder mehreren großen Texturen (Texturatlanten) gespeichert werden. Die Materialdaten eines Dreiecks sind um die Position der verwendeten Textur im Texturatlas zu ergänzen. Der Kernel für Shading und Sekundärstrahlenerzeugung muss so erweitert werden, dass die Texturen aus dem Texturatlas gelesen und entsprechend den Texturkoordinaten des Dreiecks abgetastet werden.

Würde man Texturen verwenden, müssten diese zusammen mit den für das Raytracing benötigten Daten im Texturspeicher der GPU abgelegt werden. Die Zahl der maximal verarbeitbaren Dreiecke würde somit abhängig von der Anzahl und Größe der verwendeten Texturen sinken. Die kürzlich erschienene GeForce 7950 GX2 besitzt einen Texturspeicher von 1 GB, und Probleme dieser Art sind daher ausgeschlossen. Mit einer weiteren Vergrößerung des Texturspeichers in kommenden Grafikkartenmodellen ist zu rechnen.

7 Ergebnisse und Ausblick

In dieser Arbeit wurde ein GPU-basierter Echtzeit-Raytracer implementiert und in die Szenengraph-API OGRE 3D integriert.

Durch die Weiterentwicklung der in [Pur04] und in [TS05] vorgestellten Forschungsergebnisse und die Nutzung der neusten GPU- und Fragment-Shader-Generation konnte die Parallelisierbarkeit des Raytracing-Verfahrens optimal ausgenutzt und die Last des Systems zwischen GPU und CPU verteilt werden.

Obwohl der Raytracing-Kern unter Verwendung einer NVIDIA Grafikkarte und der Shader-Sprache Cg implementiert wurde, ist das System nicht an einen Grafikkartenhersteller gebunden. Die Fragment-Shader können mit wenig Aufwand in GLSL- und HLSL-Programme übersetzt werden und lassen sich so auf einer beliebigen, das Shader-Modell 3.0 unterstützenden GPU ausführen.

Die Wahl der Hüllkörperhierarchien als Beschleunigungsdatenstruktur ermöglicht die vollständige Nutzung der Struktur des Szenengraphen. Animierte Szenen können so bei interaktiven Bildraten mit Hilfe des Raytracing-Verfahrens gerendert werden, wie die Kapitel 5.3 und 6 gezeigt haben.

Der Raytracing-Kern konnte durch die Erweiterung bestehender Konzepte der Szenengraph-API OGRE 3D nahtlos in den Szenengraphen integriert werden. Zukünftige Weiterentwicklungen des Systems lassen sich somit komfortabel realisieren.

Diese Arbeit bildet den Grundstock für eine Reihe weiterer wissenschaftlicher Arbeiten in diesem Umfeld, die sich auf die Erweiterungen des vorgestellten Systems konzentrieren. Denkbar ist hier die Verwendung von Texturen und Maßnahmen zum Anti-Aliasing sowie die Implementierung der in Kapitel 2.4.3 und 2.4.4 vorgestellten Erweiterungen des Raytracing-Verfahrens Path Tracing und Photon Mapping.

Des Weiteren wäre der Ausbau des Material-Frameworks von OGRE 3D möglich, der die Definition und Verwendung von spezifischen Raytracing-Materialien erlaubt. Dadurch können die Renderergebnisse weiter verbessert und die Oberflächeneigenschaften der Objekte detaillierter beschrieben werden.

Darüber hinaus ist die Performance des implementierten Systems auf den sich immer mehr durchsetzenden Multi-GPU-Systemen interessant.

Die Verbindung mehrerer GPUs über *SLI*[26] oder *Crossfire*[27] ermöglicht die Bündelung der GPU-Leistungen. SLI erlaubt verschiedene Betriebsmodi, unter ihnen auch der „Split-Frame-Mode“, der jedes zu berechnende Frame in mehrere Teile zerlegt und auf die Grafikprozessoren verteilt.

Einen neuen Ansatz für ein dediziertes „Visual Computing System“ stellt NVIDIA in Form der *QuadroPlex 1000*[28] auf der ACM SIGGRAPH 2006 vor. Das skalierbare System wurde speziell für grafische Berechnungen entwickelt und bündelt die Leistung mehrerer GPUs.

[26] Scalable Link Interface: Durch den Grafikkartenhersteller NVIDIA eingeführte Multi-GPU-Technik, die die Zusammenschaltung zweier oder mehrerer Grafikchips ermöglicht.

[27] Durch den Grafikkartenhersteller ATI eingeführte Multi-GPU-Technik zum gleichzeitugen Betreiben zweier Grafikkarten in einem PCI-Express-System

[28] http://www.nvidia.com/page/quadroplex.html

A Schnittstellen

In diesem Kapitel werden die wichtigsten Schnittstellen der Klassen des implementierten Systems vorgestellt. Der komplette Quellcode einschließlich des Quellcodes der Fragment-Shader des Raytracing-Kerns befindet sich auf der beiliegenden CD.

A.1 Die Klasse OgreRT2SceneManager

Diese Klasse repräsentiert den speziellen SceneManager für das Raytracing von Szenen. Sie ist verantwortlich für das Erzeugen und Verwalten der BVH der Entities und der Szene sowie für die Erzeugung und Steuerung der Compositor-Instanzen. Ebenso obliegt ihr der Download der Daten auf die GPU und die Registrierung der Compositor-Instanzen.

```
public:

  virtual Entity* createEntity( const String &entityName,
                                const String &meshName);

  virtual void _renderScene(  Camera *camera,
                              Viewport *vp,
                              bool includeOverlays);
  void createSceneBVH();

  void raytracingOn();
  void raytracingOff();

  void createCompositor(Viewport* vp);
  void registerCompositor(Viewport* vp);

  virtual void notifyMaterialSetup( uint32 pass_id,
                                    MaterialPtr &mat);

  virtual void notifyMaterialRender(uint32 pass_id,
                                    MaterialPtr &mat);

protected:

  void _render();
  void _traverseScene(Node* node);
```

```
void _commitEntityBVH(   String eName,
                         size_t currpos_BVH,
                         size_t   currpos_Materials,
                         bool doPositionUpdate,
                         bool doScaleUpdate,
                         bool doRescaleNormals);

void _commitSceneBVHToGPU();
```

Die Methode *createEntity* erzeugt eine Entity aus einem Mesh sowie die BVH der Entity. Ein Zeiger auf das Array der BVH wird in der dafür vorgesehenen Hashmap gespeichert.

_renderScene wird in der Basisklasse zum Rendern der von der aktuellen Kameraposition aus sichtbaren Objekte verwendet. Beim Raytracing müssen allerdings alle Objekte der Szene berücksichtigt werden, so dass es notwendig ist, diese Methode zu überschreiben.

_render() wird durch *_renderScene* aufgerufen und implementiert das Rendern mit Hilfe des Raytracing-Verfahrens.

_traverseScene wird durch *_render()* aufgerufen und traversiert den Szenengraphen rekursiv, sucht ihn nach Entity-Objekten ab und fügt diese in eine Render-Warteschlange ein. Im Anschluss daran werden die BVHs dieser Entities, sofern notwendig, durch die Methode *_commitEntityBVH* aktualisiert bzw. neu erstellt und in die BVH der Szene eingefügt.

Nachdem alle Entities in die BVH der Szene eingefügt wurden, wird diese durch *_commitSceneBVHToGPU* über einen Pixel-Puffer an die GPU übermittelt.

createCompositor erzeugt die für den Raytracing-Kern notwendigen Compositor-Instanzen. Diese werden im Anschluss durch *registerCompositor* bei dem aktiven Viewport der Kamera registriert.

Die Methode *notifyMaterialSetup* wird verwendet, um die erzeugten Compositor-Instanzen zu initialisieren. Dies umfasst die Konfiguration der Ein- und Ausgabe-Multi-Render-Targets (MRTs) sowie die Initialisierung der Parameter der Fragment-Shader.

Durch *notifyMaterialRender* werden notwendige Parameter vor der Berechnung jedes Bildes an die Fragment-Shader übergeben.

raytracingOn und *raytracingOff* schalten das Raytracing an bzw. aus. Ist es ausgeschaltet, so wird der Standard-Renderer von OGRE 3D verwendet.

A.2 Die Klasse EntityBVHBuilder

Diese Klasse erzeugt die BVH einer Entity. Knoten und Blätter der BVH werden durch Instanzen der Klasse *EntityBVH* repräsentiert.

```
public:

  EntityBVH* load(   Entity *ent,
                   const Quaternion &orient);
  void getMeshInformation(  const Entity * entity,
                            size_t &vertex_count,
                            Ogre::Vector3* &vertices,
                            size_t &index_count,
                            unsigned long* &indices,
                            VertexElementSemantic sem,
                            const Vector3 &position,
                            const Quaternion &orient,
                            const Vector3 &scale);

  EntityBVH* buildBranch( unsigned long *indices,
                          size_t num_indices,
                          int axis = 0);

  void createEntityBVHArray(EntityBVH* bvh,
                            Vector4* &targetEntityArray,
                            Vector4* &targetMaterialArray,
                            size_t &bvharraysize,
                            size_t &matarraysize);

private:

  EntityBVH* _build(const Quaternion &orient);
  int _split( unsigned long *indices,
              size_t size,
              float pivot_val,
              int axis);
```

Die Methode *load* lädt die Daten des Meshes einer Entity mit Hilfe der Methode *getMeshInformation* aus den Puffern und speichert sie in den übergebenen Arrays. Danach wird die BVH mit Hilfe der Methode *_build* nach einem an [SM03] angelehnten Verfahren entsprechend der Orientierung der Entity rekursiv erzeugt.

_split wird innerhalb der *_build*-Methode verwendet, um die Liste der Indizes entlang einer Achse zu unterteilen. *buildBranch* implementiert den rekursiven Aufruf und erzeugt einen Teilzweig des Baums aus der übergebenen Liste der Indizes.

Nach dem Erzeugen der BVH wird die objektorientierte Struktur mit Hilfe von *createEntityBVHArray* in ein Array konvertiert, das dem auf der GPU verwendeten Format entspricht.

B Ressourcen

B.1 Inhalt der beiliegenden CD-ROM

Der Druckversion dieser Arbeit liegt eine CD-ROM mit folgendem Inhalt bei.

B.1.1 Installer

Dieses Verzeichnis enthält einen Installer für Windows XP SP2. Nach der Installation können die Testapplikationen aus Kapitel 5.6 über das Startmenü gestartet werden. Die Lauffähigkeit der Testapplikationen kann nur unter Verwendung einer GeForce 7900GTX sowie installierter ForceWare in einer neueren Version als 85.96 gewährleistet werden.

B.1.2 Sourcecode

Der Sourcecode des implementierten Systems beinhaltet den C++ Code des OgreRT2-SceneManager-Plugins sowie den Cg Code des Raytracing-Kerns. Es sind Projektdateien für Microsoft Visual Studio 2005 enthalten. Für die Kompilierung des Sourcecodes muss die OGRE 3D Visual Studio 2005 SDK installiert sein.

B.1.3 Video

Um die Renderingergebnisse des implementierten Systems auch auf Systemen ohne die entsprechende GPU betrachten zu können, wurde ein Demovideo erstellt. Durch die Aufnahme vom Bildschirm kommt es allerdings zu niedrigeren Bildraten, als bei den entsprechenden Demos auf einer GeForce 7900 GTX. Des Weiteren führt die Komprimierung des Videos zu Qualitätsverlust.

B.2 Ressourcen im WWW

Ressourcen zu dieser Arbeit können im WWW unter der Adresse http://www.iz-media.de/diplom_bjoern/ abgerufen werden. Informationen und SDKs zu OGRE 3D stehen unter http://www.ogre3d.org zur Verfügung.

Literatur

[App68] Appel, Arthur: Some Techniques for Shading Mashine Renderings of Solids. In: *Spring Joint Computer Conference* (1968) 22

[BBDF05] Beck, Stephan ; Bernstein, Andreas-C. ; Danach, Daniel ; Fröhlich, Bernd: CPU-GPU Hybrid Realtime Ray Tracing Framework. (2005) 44

[Ben75] Bentley, Jon L.: Multidimensional binary search trees used for associative searching. In: *Communications of the ACM* 19(10) (1975), S. 509–517 31

[Ber97] van den Bergen, Gino: Efficient collision detection of complex deformable models using AABB trees. In: *J. Graph. Tools* 2 (1997), Nr. 4, S. 1–13. – ISSN 1086–7651 30

[BFH+04] Buck, Ian ; Foley, Tim ; Horn, Daniel ; Sugerman, Jeremy ; Houston, Mike ; Hanrahan, Pat: Brook for GPUs: Stream Computing on Graphics Hardware. In: *ACM Transactions on Graphics* (2004) 63

[Bli76] Blinn, Jim F.: Texture and Reflection in Computer Generated Images. In: *Communications of the ACM* 18(9) (1976), S. 542–547 22

[BLOS04] Buck, Ian ; Lefohn, Aaron ; Owens, John ; Strzodka, Robert: GPGPU: General Purpose Computation on Graphics Processors. In: *IEEE Visualization 2004 Tutorial* (2004) 39

[CHCH06] Carr, Nathan A. ; Hoberock, Jared ; Crane, Keenan ; Hart, John C.: Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. In: *To appear in the Proceedings of Graphics Interface 2006* (2006) 43, 45

[CHH02] Carr, Nathan A. ; Hall, Jesse D. ; Hart, John C.: The Ray Engine. In: *Graphics Hardware* (2002), S. 1–10 42

[Chr05] Christen, Martin: *Ray Tracing on GPU*, University of Applied Sciences Basel (FHBB), Diplomarbeit, 2005 43, 46

[Cin04] *CineFX 3.0 The Next Wave of Stunning Visual Effects.* NVIDIA Corporation, 2004 38

[DBB06] Dutre, Philip ; Bala, Kavita ; Bekaert, Philippe: *Advanced Global Illumination.* AK Peters, 2006 27

[Fer05] Fernando, Randima ; Pharr, Matt (Hrsg.): *GPU Gems 2.* Addison-Wesley, 2005 32

[FK03] Fernando, Randima ; Kilgard, Mark J.: *The Cg Tutorial*. Addison-Wesley, 2003. – ISBN 0321194969 37, 38, 40

[Fly95] Flynn, Michael J.: *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, Inc., 1995. – ISBN 0867202041 37

[FS05] Foley, Tim ; Sugerman, Jeremy: KD-Tree Acceleration Structures for a GPU Raytracer. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2005), S. 15–22 43

[FSZ98] Fellner, D. ; Schäfer, S. ; Zens, M.: Photorealistic Rendering in Heterogeneous Networks. In: D'Hollander, E. H. (Hrsg.) ; Joubert, G. R. (Hrsg.) ; Peters, F. J. (Hrsg.) ; Trottenberg, U. (Hrsg.): *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany* Bd. 12. Amsterdam : Elsevier, North-Holland, 1998, S. 113–120 40

[FTI86] Fujimoto, Akira ; Tanaka, Takayu ; Iwata, Kansei: ARTS: Accelerated Ray-Tracing System. In: *IEEE Computer Graphics and Applications* 6(4) (1986), S. 127–133 31

[Fuc80] Fuchs, Henry: On visible surface generation by a priori structures. In: *Computer Graphics, Proceedings of the ACM SIGGRAPH* 14 (1980), S. 127–133 31

[GGHM05] Galoppo, Nico ; Govindaraju, Naga K. ; Henson, Michael ; Manocha, Dinesh: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In: *Proceedings of the ACM/IEEE SC|05 Conference* (2005) 40

[Gla86] Glassner, Andrew S.: Space Subdivision for Fast Ray Tracing. In: *IEEE Computer Graphics & Applications* 6(4) (1986), S. 16–26 31

[GS87] Goldsmith, Jeffrey ; Saloman, John: Automatic Creation of Object Hierarchies for Ray Tracing. In: *IEEE Computer Graphics & Applications* 7 (1987), S. 14–20 30

[Hai88] Haines, Eric: Spline Surface Rendering, and What's Wrong with Octrees. In: *Ray Tracing News* 1(2) (1988) 31

[Hur05] Hurley, Jim: Ray Tracing Goes Mainstream. In: *Intel Technology Journal, Compute-Intensive, Highly Parallel Applications and Uses* 9/2 (2005) 41

[JC95] Jensen, Henrik W. ; Christensen, Niels J.: Efficiently Rendering Shadows using the Photon Map. In: *Proceedings of Compugraphics* (1995), S. 285–291 27

[Jen01] Jensen, Henrik W.: *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001. – ISBN 1568811470 23

[Jun06] Junker, Gregory: *Pro OGRE 3D Programming*. APress, 2006. – ISBN 1590597109 20

[Kay86] Kayjiya, James: The rendering equation. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986) 25, 26

[Kir02] Kirk, David: *Cg Toolkit User's Manual - A Developer's Guide to Programmable Graphics*. NVIDIA Corporation, 2002 38

[KK86] Kay, Timothy ; Kayjiya, James: Ray Tracing Complex Scenes. In: *Computer Graphics* 10 (1986), S. 269–278 29, 30

[KL04] Karsson, Filip ; Ljungstedt, Carl J.: *Ray Tracing fully implemented on programmable graphics hardware*, Chalmers University of Technology, Diplomarbeit, 2004 43, 46

[LAM00] Lext, Jonas ; Assarsson, Ulf ; Möller, Tomas: BART: A Benchmark for Animated Ray Tracing / Department of Computer Engineering Chalmers University of Technology. 2000. – Forschungsbericht 71

[LAM01] Lext, Jonas ; Akenine-Möller, Tomas: Towards Rapid Reconstruction for animated Ray Tracing. In: *EUROGRAPHICS* (2001) V, VII, 47, 52

[LS05] Lorion, Yann ; Schmidt, Björn. *Fraktale Erzeugung von gothischen Kathedralen*. Realisierung im Rahmen des Grafikpraktikums an der J.W. Goethe-Universität. 2005 19

[may03] *Learning Maya 5 | Rendering*. Alias Wavefront Education / Sybex, 2003. – ISBN 1894893433 XII, 28

[MFM04] Moreno-Fortuny, Gabriel ; McCool, Michael: Unified stream processing raytracer. In: *Poster at GP2: The ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004 poster* (2004) 63

[MT97] Moller, Tomas ; Trumbore, Ben: Fast, Minimum Storage Ray-Triangle Intersection. In: *JGTOOLS: Journal of Graphics Tools* 2 (1997) 33

[Nem05] Nemec, Toni: *Entwicklung eines Echtzeit-Raytracers*, J.W. Goethe-Universität, Diplomarbeit, 2005 41

[NSL06] Nitschke, Benjamin ; Schürmann, Tim ; Lange, Thorsten: Bau dir dein eigenes Spiel. In: */GameStar/dev* 02/2006 (2006), S. 11–25 20

[NVd06] *NVIDIA Ressourcen für Entwickler http://developer.nvidia.com.* NVIDIA Corporation, 2006 45

[NVI05] *NVIDIA GPU Programming Guide Version 2.4.0.* NVIDIA, 2005 38, 43

[POAU00] Peercy, Mark S. ; Olano, Marc ; Airey, John ; Ungar, Jeffrey: *Interactive multi-pass programmable shading.* ACM Press/Addison-Wesley Publishing Co., 2000. – 425–432 S. – ISBN 1–58113–208–5 37

[Pur04] Purcell, Timothy J.: *Ray Tracing On A Stream Processor*, Stanford University, Diss., 2004 V, VII, 32, 42, 75

[RVB02] Reiners, Dirk ; Voss, Gerrit ; Behr, Johannes: OpenSG: Basic Concepts / opensg.org. 2002. – Forschungsbericht 19

[SM03] Shirley, Peter ; Morley, R. K.: *Realistic Ray Tracing.* A K Peters, 2003. – ISBN 1568811985 53, 80

[SPD+04] Schmittler, Jörg ; Pohl, Daniel ; Dahmen, Tim ; Vogelgesang, Christian ; Slusallek, Philipp: Realtime Ray Tracing for Current and Future Games. In: *GI Jahrestagung (1)*, 2004, S. 149–153 41

[Ter01] Terdiman, Pierre: Memory-optimized bounding-volume hierarchies. In: *http://www.codercorner.com/Opcode.htm* (2001) 55

[TS05] Thrane, Niels ; Simonsen, Lars O.: *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*, University of Aarhus, Diplomarbeit, 2005 V, VII, XII, 39, 43, 46, 55, 56, 60, 75

[Wha04] Whalen, S.: Audio and the Graphics Processing Unit. In: *Audio and the Graphics Processing Unit, IEEE Vis 2004 GPGPU Tutorial* (2004) 40

[WHG84] Weghorst, H. ; Hooper, G. ; Greenberg, D.P.: Improved Computational Methods for Ray Tracing. In: *ACM Transactions on Graphics* 3 (1984), S. 52–69 29

[Whi80] Whitted, Turner: An improved illumination model for shaded display. In: *Communications ACM* 23 (1980), Nr. 6, S. 343–349. – ISSN 0001–0782 23

[WIK+06] Wald, Ingo ; Ize, Thiago ; Kensler, Andrew ; Knoll, Aaron ; Parker, Steven G.: Ray Tracing Animated Scenes Using Coherent Grid Traversal. In: *ACM Transactions on Graphics* 25 (2006), Nr. 3, S. 485–493 41

[WSBW01] Wald, Ingo ; Slusallek, Philipp ; Benthin, Carsten ; Wagner, Markus: Interactive Rendering with Coherent Ray Tracing. In: Chalmers, A. (Hrsg.) ; Rhyne, T.-M. (Hrsg.): *EG 2001 Proceedings* Bd. 20(3). Blackwell Publishing, 2001, S. 153–164 41

[WSS05] Woop, Sven ; Schmittler, Jörg ; Slusallek, Philipp: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In: *Proceedings of ACM SIGGRAPH* (2005) 44

[WW92] Watt, Alan ; Watt, Mark: *Advanced Animation and Rendering Techniques*. Addison-Wesley, 1992. – ISBN 0201544121