Interaktives Ray-Tracing und Ray-Casting auf programmierbaren Grafikkarten

Bachelorarbeit, vorgelegt von Christian Lessig

Dezember 2004

Bauhaus-Universität Weimar

Fakultät Medien

Version 1.1

22. Januar 2005

Diese Version entstand durch die Bemerkungen und Hinweise, welche ich im Rahmen der Verteidigung der Arbeit erhielt. Vielen Dank hierfür insbesondere Herrn Professor Bernd Fröhlich.

Versicherung

Hiermit erkläre ich eidesstattlich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weimar, den 24. Dezember 2004

Christian Lessig

Danksagung

Zunächst möchte ich mich bei Professor Fröhlich für seine Unterstützung bedanken, für die Diskussionen in denen die Arbeit immer wieder weiterentwickelt wurde und vor allem auch für die Freiheit die ich bei der Wahl und Ausarbeitung des Themas hatte. Ihm, und Nvidia, ist auch für die Hardware zu danken, welche mir bereits ab Sommer 2004 für die Umsetzung der Arbeit zur Verfügung stand.
Ganz besonders möchte ich mich bei Jan Springer und Hans Pabst bedanken. Ohne ihre Hinweise und die vielen anregenden Diskussionen wäre diese Arbeit niemals zum jetzigen Stand gekommen. Jan Springer gebührt insbesondere auch Dank für das Korrekturlesen der Arbeit.
Meinen Eltern möchte ich für ihre Unterstützung in den vergangenen Jahren danken. Ohne sie wäre mein Studium in Weimar und diese Arbeit nicht möglich gewesen.
Meinen Kommilitonen möchte ich danken, dass sie das Leben im Lab immer wieder zu einer schönen Erfahrung gemacht haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1
Einleitung

1.1. Motivation der Arbeit

Das heute meist verwendete Verfahren zu Bilderzeugung im Computer ist das so genannte *Feed Forward Rendering*, oft auch als *Rastergrafik* bezeichnet. Dieses Verfahren kann, wie nicht zuletzt die aktuelle Grafikhardware zeigt, sehr effizient implementiert werden. Das *Feed Forward Rendering* besitzt jedoch einige Nachteile:

1. Es ist eine Approximation der darzustellenden Geometrie durch Dreiecke notwendig. Insbesondere bei gekrümmten Oberflächen ist dies ein rechenaufwändiger Prozess, der zu einem hohen Speicherplatzbedarf führt. So sind zum Beispiel zur Darstellung einer Kugel – je nach Qualitätsanforderungen – bis zu einigen tausend Dreiecke beziehungsweise einige zehntausend Punkte notwendig. In der algebraischen Darstellung benötigt man dafür vier Koordinaten, den Mittelpunkt im dreidimensionalen Raum und den Radius. Darüber hinaus können selbst bei einer sehr feinen Triangulierung, das heißt der Zerlegung in sehr viele Dreiecke, Fehler in der Darstellung entstehen, so genannte Artefakte. Die Approximation durch Dreiecke wird dann erkennbar.

2. Das *Feed Foward Rendering* erlaubt keine Berechnung von globalen Beleuchtungseffekten, zum Beispiel Reflexion, Brechung und Schatten. Zwar wurden verschiedene Algorithmen wie *Reflection Mapping* [BLI76] oder *Shadow Mapping* [WIL78] vorgeschlagen, um globale Beleuchtungseffekte zu ermöglichen, jedoch sind auch diese Verfahren nur Approximationen.

3. Die Komplexität des *Feed Forward Rendering* ist $O(N)$ für die Anzahl der Objekte, so dass selbst bei der Verwendung von modernster Grafikhardware das Rendern großer bis sehr großer Szenen bei interaktiven Bildwiederholraten nicht möglich ist.

Eine mögliche Lösung für diese Probleme bietet das *Ray-Tracing-Verfahren* [APP68] [MAG68], die *Strahlverfolgung*:

1. Ray-Tracing erlaubt die direkte Darstellung von Objekten ohne Triangulierung. Dies ermöglicht, neben einer Reduzierung des Speicherplatzbedarfs, dass die Geometrie in bestmöglicher Qualität abgebildet werden kann.

2. Ray-Tracing erlaubt die nahezu physikalisch korrekte Berechnung von globalen Beleuchtungseffekten. Dabei kann aber auch, je nach Qualitätsanforderungen und zur Verfügung stehender Rechenzeit, eine ungenauere Approximation verwendet werden.

3. Durch die Verwendung von Beschleunigungsstrukturen kann beim Ray-Tracing eine Komplexität von $\log(N)$ für die Anzahl der Objekte erreicht werden.

Darüber hinaus bietet Ray-Tracing zwei weitere wesentliche Vorteile gegenüber dem *Feed Forward Rendering*:

- Das *Shading* erfolgt beim Ray-Tracing nur für die sichtbaren Oberflächenpunkte [WAL01]

- Das Ray-Tracing-Verfahren ist a inherent parallel. Die Leistung eines Ray-Tracing-Systems kann durch die Verwendung mehrerer Recheneinheiten effizient vergrößert werden [WAL01].

[1] Im Bereich des *Feed Forward* Rendering wird dies auch als *Deferred Shading* bezeichnet.

1.2. Ziele und Methodik der Arbeit

Ziel der Arbeit war die Implementierung eines interaktiven Ray-Tracing-Systems für höhergradige Primitive und parametrische Freiformflächen, zum Beispiel *Quadrics* und *NURBS*-Oberflächen. Um dabei auch für Szenen mit sehr vielen Objekten interaktive Bildwiederholraten zu erreichen, sollten sowohl die CPU als auch die GPU für das zu entwicklende Ray-Tracing-System verwendet werden.
Zwei wesentlich technische Neuerungen des Jahres 2004 stellten für die Zielstellung die Voraussetzungen dar:

1. Die deutlich verbesserte Programmierbarkeit der neusten Generation von Grafikhardware, zum Beispiel der *NV4X*-Architektur von *Nvidia*

2. *PCI express* (*PCIe*) [PCI04], welches den Datentransfer zwischen Grafikhardware und CPU, und dabei insbesondere das Zurücklesen von Daten in den Hauptspeicher beschleunigt beziehungsweise beschleunigen wird.

Allerdings musste im Laufe der Arbeit festgestellt werden, dass die im Moment erhältlichen *PCIe*-Systeme noch nicht die spezifizierten Datentransferraten zwischen GPU und CPU erreichen. Da ein effizienter Datentransfer jedoch eine wesentliche Voraussetzung für das geplante Ray-Tracing-System war, zeigte sich bereits nach Voruntersuchungen, dass die Zielstellung mit der aktuell verfügbaren Hardware nicht erreicht werden kann.
Bei der Implementierung wurde deshalb ein Ray-Casting-System realisiert. Dabei konnte ein bereits für das Ray-Tracing-System entwickelter Beschleunigungsansatz für die Augenstrahlen verwendet werden, welcher insbesondere die Möglichkeiten der Grafikkarte nutzt.

Mit dieser Arbeit und den darin vorgestellten Verfahren wird gezeigt, dass auch mit Ray-Casting und pixelgenauen Berechnungen, selbst für Szenen mit einigen tausend bis zehntausend Primitiven zweiter Ordnung und Dreiecken, interaktive Bildwiederholraten erreicht werden können. Darüber hinaus wurde im Rahmen dieser Arbeit ein Ray-Casting-System für NURBS-Oberflächen auf der GPU implementiert.

1.3. Aufbau der Arbeit

Im zweiten Kapitel werden die zum Verständnis der Arbeit notwendigen Grundlagen erläutert. Zunächst wird dabei auf die aktuelle Grafikhardware und ihre Programmierung eingegangen, bevor im Weiteren das Ray-Tracing vorgestellt wird.
Im dritten Kapitel wird der Ansatz des entwickelten Ray-Tracing-Systems erläutert. Dabei wird zunächst begründet, warum eine Aufteilung des Ray-Tracing-Verfahrens auf CPU und GPU sinnvoll erschien. Anschließend wird die verwendete Beschleunigungsstruktur vorgestellt und näher auf die Probleme bei der Aufteilung auf CPU und GPU eingegangen. Das Kapitel schließt mit der Vorstellung eines neuen, insbesondere für die Grafikkarte geeigneten Beschleunigungsansatzes für die Augenstrahlen.
Auf die Implementierung wird im vierten Kapitel der Arbeit eingegangen. Im ersten Teil wird die Umsetzung des Beschleunigungsansatzes für die Augenstrahlen vorgestellt, bevor anschließend das Ray-Casting-System für Primitive zweiter Ordnung und Dreiecke erläutert wird. Die Vorstellung des Ray-Casting-Verfahrens für NURBS-Oberflächen bildet den Abschluss von Kapitel 4.
Die Arbeit schließt in Kapitel 5 mit einer Zusammenfassung.

Kapitel 2
Grundlagen der Arbeit

2.1. Grafikhardware

2.1.1. Einführung in OpenGL

In den vergangenen Jahren fand im Bereich der Rastergrafik-Hardware ein radikaler Wandel statt. Waren noch vor zehn Jahren teure Spezialsysteme, so genannte Grafik-Supercomputer, wie die der Firmen *SGI* oder *Evans and Sutherland* notwendig, um dreidimensionale Szene in Echtzeit darzustellen, so ist heute dafür eine handelsübliche Grafikkarte von Firmen wie *Nvidia*, *ATI* oder *3DLabs* ausreichend. Die Rechenleistung der Grafikkarten wächst dabei deutlich schneller als die von CPU-Prozessoren[2] (Abb. 4).
Ihre Leistungsfähigkeit erreicht die Grafikhardware dabei vor allem durch die Beschränkung auf einen bestimmten Typ von Primitiven: *„lighted, smooth shaded, Depth Buffered, texture mapped, antialiased triangles“* [AKE93], wie sie zuerst von SGI in der *Reality Engine* eingesetzt wurden. Parallel zur Reality Engine wurde

[2] Während bei CPU-Prozessoren in den vergangenen Jahren zusätzliche Transistoren meist für eine Vergößerung der Cache-Speicher verwendet wurden, ist bei Grafikkarten-Prozessoren mit diesen die Anzahl der ALU's (Arithmetic Logic Unit) vegrößert wurden, zum Beispiel in zusätzlichen Pipelines

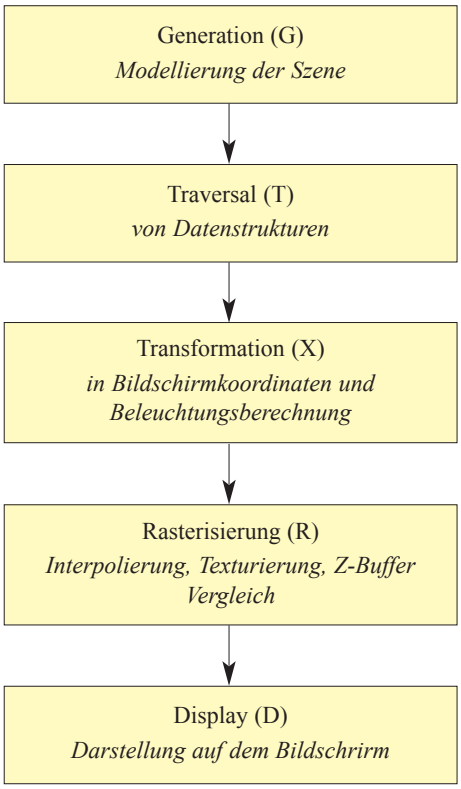


Abbildung 1: GTXRD-Modell

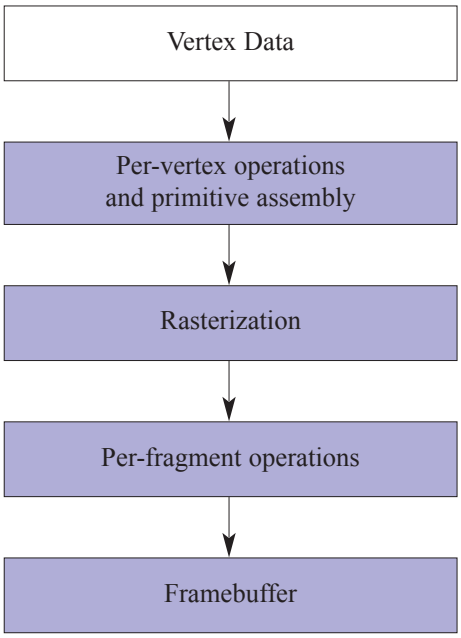


Abbildung 2: OpenGL Pipeline

OpenGL [OGL04] entwickelt, welches heute, neben *DirectX* [MIC04], das am häufigsten verwendete API für diese Art von Rastergrafik ist. Das *Open* im Namen steht für die offene Architektur, welche zwar die Funktionsweise, nicht aber die Implementierung definiert. Dies ermöglicht, dass OpenGL in unterschiedlichen Software Umgebungen und auf unterschiedlicher Hardware verwendet werden kann. Um diese abstrakte Definition zu ermöglichen, wurde das Modell der OpenGL Pipeline entwickelt, welches eine Weiterentwicklung der letzten drei Teile des *GTRXD*-Modells [AKE89] ist (Abb. 1). OpenGL wird dabei häufig auch als Client-Server-Architektur bezeichnet. Die OpenGL Pipeline, wie in Abbildung 2 dargestellt, ist dabei der Server, welche die Befehle der Client-Seite, die API Aufrufe, ausführt beziehungsweise umsetzt. Die OpenGL Standard-Pipeline soll im Folgenden am Weg eines beleuchten, texturierten und korrekt tiefensortierten Dreiecks durch diese erläutert werden (Abb. 3).

Das *Rendern*, das Darstellen des Dreiecks auf dem Bildschirm, beginnt mit den Geometrie-Informationen, welche von der OpenGL Client-Seite an den OpenGL Server übergeben werden. Die Geometrie wird dabei mit Hilfe von Punkten im 3D-Raum, den so genannten *Vertices*, beschrieben. Zusätzlich wird auf Client-Seite der Primitiv-Typ, in diesem Fall „Dreieck“,

definiert, welcher die Relation der Punkte beschreibt. In der Vertex-Einheit erfolgt anschließend die Transformation der Punkte in das Bildschirm-Koordinatensystem. Ebenfalls in der Vertex-Einheit werden die Textur-Koordinaten in das Bildschirm-koordinatensystem transformiert. Der nächste Schritt in der Pipeline ist die *Primitve Assembly*, in welcher die drei Punkte, welche zuvor in der Vertex-Einheit unabhängig voneinander verarbeitet wurden, wieder zu einem Dreieck zusammengefasst werden. Die *Primitve Assembly* ist dabei die Voraussetzung für die *Rasterisierung*. Bei dieser wird für jede Pixelposition überprüft, ob die Projektion des Dreiecks das zugehörige Pixel abdeckt. Ist dies der Fall, so wird für die Pixelposition ein *Fragment* erzeugt. Die Farbe, der Tiefenwerte und die Texturkoordinate für das Fragment werden dabei durch die Interpolation der Werte der Punkte des Dreiecks bestimmt. Nach der Rasterisierung wird anhand der Texturkoordinaten für jedes Fragment die Texturinformation ausgelesen und auf das Fragment angewendet. Die letzte Operation in der Pipeline ist der Tiefentest, auch als *Depth Buffering* oder *Z-Test* bezeichnet. Dabei wird der Tiefenwert des Fragments mit dem im *Depth Buffer* an derselben Pixelposition stehenden Wert verglichen. Ist der Wert im *Depth Buffer* größer als der des Fragments, so liegt das bereits an dieser

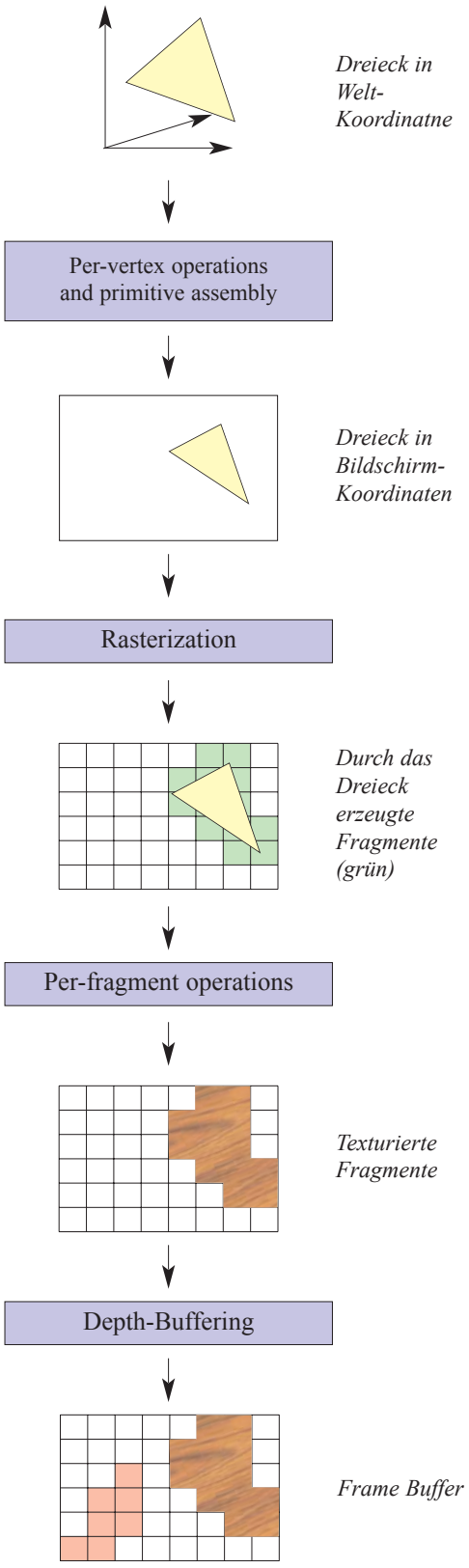


Abbildung 3: Verarbeitung eines Dreiecks in der OpenGL Pipeline

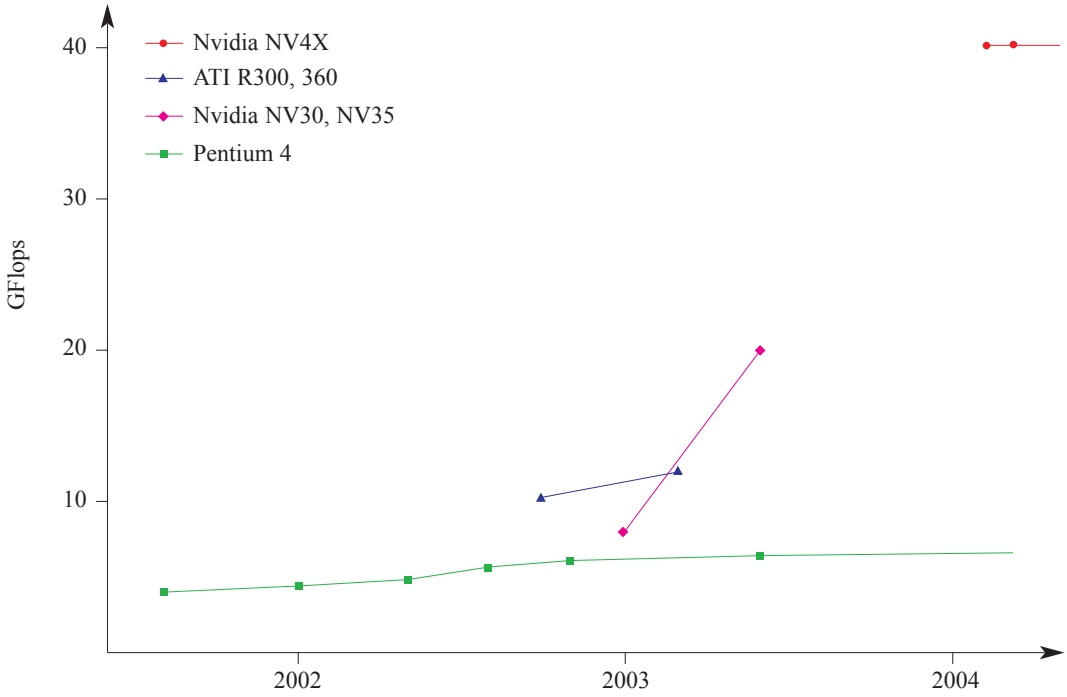


Abbildung 4: Entwicklung der Floating-Point Verarbeitungsleistung von Grafikhardware und CPU-Prozessoren

Pixelkoordinate geschriebene Fragment weiter vom Augenpunkt entfernt als das aktuelle. In diesem Fall werden die Werte des aktuellen Fragments in den *Frame Buffer* oder das *Render Target* geschrieben und es wird damit zu einem *Pixel* [AKE93]. Anderenfalls wird das aktuelle Fragment verworfen, da es durch das bereits in den *Frame Buffer* geschriebene, verdeckt ist.

OpenGL ist dabei als *State Machine* definiert. Jeder Kontext, die Repräsentation einer *State Machine* in einem Programm, besitzt einen Zustand, der die Rendering-Parameter enthält. Die Parameter, zum Beispiel die Position des Augenpunktes oder die Eigenschaften der Lichtquellen, werden dabei solange verwendet, bis eine Änderung des Zustandes erfolgt. Ein Programm kann mehrere *State Machines* beziehungsweise Kontexte besitzen. Allerdings nur eine Kontext pro Prozess aktiv sein kann.

Um trotz der festgelegten Pipeline eine einfache An*pass*ung an spezielle Laufzeitumgebungen und die einfache Integration zusätzlicher Funktionalitäten zu ermöglichen, existiert das Konzept der *Extensions*. Eine *Extension* ist eine Ergänzung oder Veränderung der OpenGL Standard-Pipeline, welche für bestimmte Implementierungen definiert wird. Durch die Verwendung von *Extensions* geht aber ein Teil der Interoperabilität zwischen verschieden Laufzeitumgebungen verloren.

2.1.2. OpenGL und Grafikhardware

Auf modernen Grafikkarten ist die gesamte OpenGL Pipeline in Hardware implementiert. Dies und die Parallelisierung der Berechnungseinheiten für die Per-Vertex und Per-Fragment Operationen ermöglicht eine deutlich höhere Leistung als bei CPU-Implementierungen. Durch die höhere Leistung können dabei mehr Dreiecke und Fragmente pro Takt verarbeitet werden.
Die Qualität der Darstellung, dass heißt die Realitätsnähe, wird aber nicht nur durch die Komplexität der Geometrie bestimmt, sondern auch durch das *Shading*. Unter *Shading* werden dabei alle Operationen zusammengefasst, welche das optische Erscheinungsbild eines Fragments beeinflussen, also zum Beispiel die Beleuchtungsberechnung und Texturierung, aber auch Verfahren wie *Bump Mapping* und *Displacement Mapping* [BLI78]. Um dabei weitere Effekte als das einfache, in OpenGL verwendete, *Gouraud Shading* [GOU71] und mehr als eine Textur zu ermöglichen, wurden ab 2001 von einigen Herstellern auf Grafikkarten programmierbare Einheiten implementiert, die so genannten *Shader*. Zunächst existierten dabei nur *Fragment-Shader*, mit welchen die Eigenschaften eines Fragments verändert werden können. Später folgten auch *Vertex-Shader* für die Manipulation der Vertex Eigenschaften.
Dabei werden jeweils auf allen Vertices beziehungsweise auf allen Fragmenten die identischen, in den Shader-Programmen festgelegten, Operationen ausgeführt. Durch die Verarbeitung in *Pipelines*, den parallelen Berechungseinheiten, ist es dabei nicht möglich, dass die Berechnungen für einen Vertex beziehungsweise ein Fragment von denen eines anderen Vertex oder Fragments desselben *Rendering Passes* abhängen. Ein *Rendering Pass* umfasst alle Operationen, welche auf einem *Render Target* ausgeführt werden, bis dessen Inhalt zum ersten Mal verwendet wird; zum Beispiel bis die Darstellung auf dem Bildschirm oder das Zurücklesen auf die OpenGL Client-Seite erfolgt.

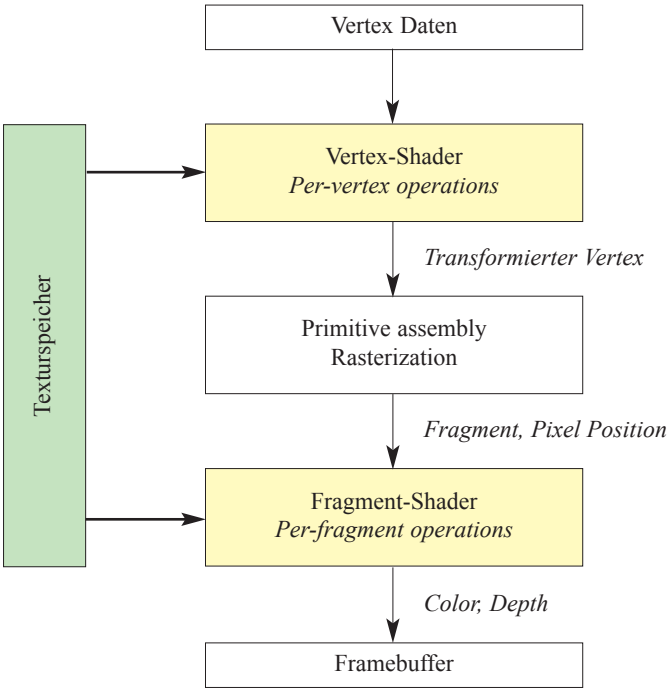


Abbildung 5: OpenGL Pipeline auf programmierbaren Grafikkarten

2.1.3. Grafikkarten-Programmierung und GLSL

Die Einführung von programmierbaren Einheiten auf Grafikkarten führte zu einer Veränderung der OpenGL Standard-Pipeline, wie in Abbildung 5 dargestellt. Die Shader-Programme wurden dabei zunächst in Assembler-Code geschrieben. Dadurch waren die Befehle hardware- und herstellerspezifisch, so dass Programme für jede Generation von Grafikkarten und für jeden Hersteller gegebenenfalls neu geschrieben werden mussten.

Im Jahr 2003 wurde deshalb mit *C for Graphics*, kurz *Cg*, [MAR03] die ersten *High Level Shading Language* für OpenGL vorgestellt, welche die Grafikkarten-Programmierung abstrahiert. Die wesentlichen Konzepte der von Nvidia entwickelten Sprache stammten dabei aus bereits früher vorgestellten *Shading Languages* wie der *RenderMan Shading Language* [HAN90] und *Pixelflow* [OLA98], sowie der Programmiersprache C [PRC99].

Nachdem alle wichtigen Hersteller programmierbare Grafikkarten anboten, erschien die Standardisierung einer *Shading Language* für OpenGL sinnvoll. Im März 2004

erfolgte diese mit der Verabschiedung des *OpenGL Shading Language* Standards [KES04]. Die *OpenGL Shading Language*, meist mit *GLSL* oder *GLSLang* abgekürzt, wurde im Sommer 2004, zusammen mit OpenGL 2.0, offiziell vorgestellt.

Im Folgenden soll eine kurze Übersicht über GLSL gegeben werden, ausführliche Erläuterungen finden sich zum Beispiel im *Orange Book* [ROS04].
GLSL wurde auf Grundlage der Programmiersprache C entwickelt, wobei einige auf der Grafikkarte sinnvolle Typen und Funktionen ergänzt wurden und einige Sprachbestandteile, welche auf aktuellen Grafikkarten nicht implementiert werden können, entfernt wurden. Die wichtigsten Unterschiede sind hier aufgelistet.

Auf der Grafikkarte nicht verfügbar sind:

- Pointer, sowie die damit verbundenen Befehle zur Speicherverwaltung
- Funktionsparameterübergabe per Pointer
- Datentypen für Zeichen und Zeichenketten, sowie die auf diesen definierten Funktionen
- Befehle zum Arbeiten mit Dateien
- `unions` und `enums`
- Bit Operationen

Zum C-Standard hinzugefügt wurden:

- Spezielle Typen für zwei-, drei- und vierdimensionale Vektoren und Matrizen, welche auf verschiedenen Datentypen aufbauen, zum Beispiel `vec3` für einen *Floating Point*-Vektor mit drei Elementen, oder `ivec4` für einen *Integer*-Vektor mit vier Elementen
- Die wichtigsten mathematischen Funktionen welche auf Vektoren und Matrizen definiert sind, zum Beispiel Vektorprodukt, Kreuzprodukt, Vektor-Matrix-Multiplikation und Euklidsche Norm. Die Ausführung der Funktionen ist dabei auf der GPU sehr effizient.
- Typen und Befehle für bildbasierten Texturen, zum Beispiel `sampler2D`

```
varying vec2 my_texcoord;

// vertex shader
void main() {

  // use built-in input attribute variable for vertex
  // position to transform the vertex from world to
  // clipping space
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // pass input color and texture coordinates
  gl_FrontColor = gl_Color;

  // pass texture coordinate using a user-defined
  // varying variable
  my_texcoord = gl_MultiTexCoord0.xy;
}
```

Abbildung 6: Vertex-Shader Programm in GLSL

Zusätzlich wurden drei neuen *type qualifiers* eingeführt, um die Ein- und Ausgangsdaten der Shader-Programme zu spezifizieren:

- *Uniform Variables*, welche auf der OpenGL Client-Seite spezifiziert werden. Diese können dabei als Teil der OpenGL *State Machine* aufgefasst werden und haben in allen Vertex- beziehungsweise Fragment-Shader-Programmen den zuletzt auf OpenGL Client-Seite festgelegten Wert. In der OpenGL Standard-Pipeline sind *Uniform Variables* zum Beispiel die *Model-View-Projection Matrix* oder die Eigenschaften einer Lichtquelle.

- *Attributes* beziehungsweise *Vertex Attributes*, welche nur im Vertex-Shader vorhanden sind . Diese können sich pro Vertex ändern. In der OpenGL Standard-Pipeline sind dies die Vertex-Parameter wie Position und Farbe.

- *Varying Variables*, welche zum Datenaustausch zwischen Vertex- und Fragment-Shader dienen. In der OpenGL Standard-Pipeline sind dies zum Beispiel die Farben und Texturkoordinaten eines Fragments. Dies bedingt, dass *Varying Variables*, entsprechend der OpenGL Pipeline, interpoliert werden.

Die Variablen der OpenGL Standard-Pipeline sind in jedem Shader-Programm automatisch definiert. Darüber hinaus können auch benutzerdefinierte Variablen verwendet werden. Bei diesen sind aber hardwarebedingte Limitierungen zu beachten. Zum

```
varying vec2 my_texcoord;
uniform sampler2D texture;

// fragment shader
void main() {

  // paint the fragment with the weighted sum of the
  // color value and the value read from the texture
  // gl_FragColor, gl_Color and the result from the
  // texture lookup are of type vec4
  // (a four component vector)
  gl_FragColor = (0.3 * gl_Color)
                  + texture2D( texture, my_texcoord);
}
```

Abbildung 7: Fragment-Shader Programm in GLSL

Beispiel ist sowohl die Anzahl der *Vertex Attributes* als auch die Anzahl der *Varying Variables* stark begrenzt.
Die Shader-Programme in Abbildung 6 und 7 sollen die Programmierung in GLSL veranschaulichen. Der Vertex-Shader (Abb. 6) transformiert 3D-Punkte aus dem Welt- in das Augenkoordinatensystem. Neben der Vertex-Position werden die Farbe sowie die Texturkoordinate des Punktes an die Rasterisierungseinheit übergeben.
Im Fragment-Shader (Abb. 7) erfolgt die Berechnung der Farbe des Fragments durch eine gewichtete Summe. Der erste Summand ist die automatisch vorhandene *Varying Variable* `gl_Color`, welche den, von der Rasterisierungseinheit übergebenen, interpolierten Farbwert enthält. Der zweite Summand wird durch das Auslesen aus einer Textur ermittelt. Das Element der Textur welches verwendet wird, das so genannte *Texel*, wird durch die benutzerdefinierte *Varying Variable* `my_texcoord` bestimmt. Dabei müssen `my_texcoord` und `gl_Color` im Fragment-Shader nicht die im Vertex-Shader übergebenen Werten besitzen, da in der Rasterisierungseinheit, in Abhängigkeit vom verwendeten Primitiv-Typ, eine Interpolation erfolgt.

Bei der Einführung von GLSL war eine Erweiterung der OpenGL API auf der Client-Seite notwendig. Neu hinzugekommen sind unter anderem Befehle zum Laden, Kompilieren und Linken der Shader-Programme sowie zur Spezifikation der *Uniform Variables*.
Die für GLSL benötigten Funktionalitäten wie Compiler und Linker sind im OpenGL Treiber implementiert. Da die Treiber herstellerspezifisch sind, ermöglicht dies die Optimierung des GLSL-Programmcodes auf die verwendete Hardware.

2.1.4. NV4X-Architektur

Die Möglichkeiten und Limitierungen bei der Programmierung von Grafikkarten werden stark von der verwendeten Hardware definiert. Aus diesem Grund wird in diesem Abschnitt die *NV4X*-Architektur von *Nvidia* vorgestellt, welche für die Planung und Umsetzung der Arbeit verwendet wurde.

Die NV4X-Architektur ist die sechste Generation von Nvidia Grafikkarten und bringt einige für die Arbeit wesentliche Neuerungen mit sich. Die wichtigste ist die verbesserte Programmierbarkeit, welche bei der NV4X-Architektur erstmals die effiziente Ausführung von bedingten Anweisungen und Schleifen ermöglicht. Darüber hinaus bietet die NV4X-Architektur als erste GPU die Möglichkeit Textur-Daten im Vertex-Shader auszulesen. Nvidia's Grafikkarten der sechsten Generation sind damit, neben der *3DLabs Realizm* [3DL04], die einzigen, welche alle im GLSL-Standard definierten Sprachbestandteile zur Verfügung stellen.
Neben der verbesserten Programmierbarkeit ist bei der NV4X-Architektur auch die Rechenleistung deutlich gestiegen. Dabei werden erstmals 32-Bit Gleitkomma-Operationen ohne Geschwindigkeitsverlust ausgeführt. Die maximal erreichbare *Floating Point* Leistung der NV4X-Architektur ist mit circa 40 GFLOPS[3] in etwa sechs Mal so groß wie die einer Pentium 4 CPU [HAR04a]. Dies wird, wie Abbildung 8 zeigt, durch einen sehr hohen Grad an Parallelisierung erreicht. Die NV4X-Architektur besitzt dabei bis zu sechs Vertex- und 16 Fragment-Pipelines.

Die gemessenen 40 GFLOPS sind eine obere Schranke, welche in der Praxis kaum erreicht wird. Eines der wesentlichen Probleme bei der Ausnutzung der Rechenleistung ist die in der Fragment-Einheit verwendete SIMD-Architektur[4] [FLY72]. In einer Pipeline werden dadurch die Fragmente nicht seriell verarbeitet, sondern es wird jeweils ein Befehl des Shader-Programms für mehrere hundert Fragmente ausgeführt [MED04]. Durch diese Verarbeitungsweise können die auf der Grafikkarte notwendigen Zustandsänderungen verringert werden. Dabei entsteht in Programmen mit bedingten Anweisungen das Problem, dass für alle Fragmente alle auftretenden Verschachtelungen des Programms berechnet werden müssen, dass also

[3] GFLOPS: Floating Point Operations per Second
[4] SIMD: Single Instruction – Multiple Data

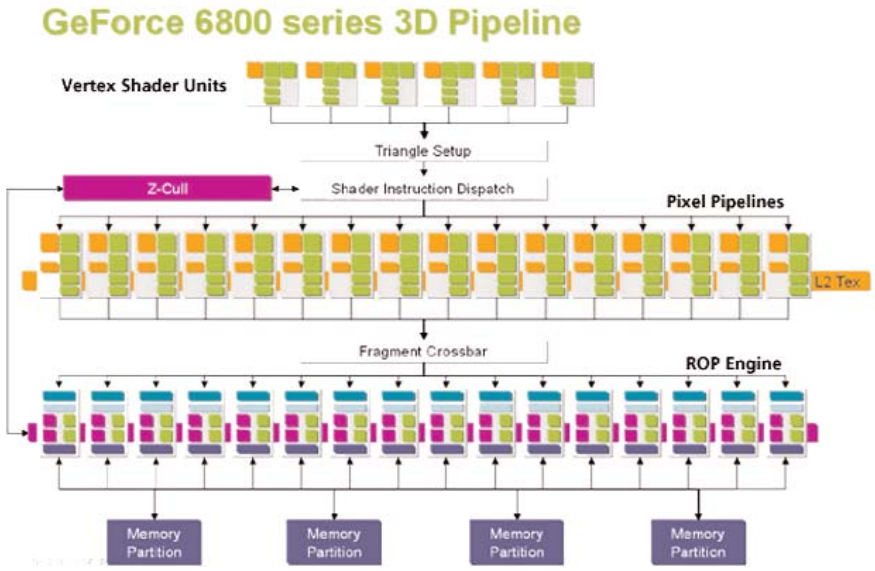


Abbildung 8: Schematischer Aufbau der NV4X-Architektur, aus [THG04]

jede von irgendeinem Fragment durchlaufene Verschachtelung von allen anderen Fragmenten auch durchlaufen werden muss. Im Gegensatz zur Fragment-Einheit besitzen die sechs Pipelines des Vertex-Prozessors eine MIMD-Architektur[5], dass heißt sie arbeiten vollständig unabhängig voneinander. Durch die geringere Anzahl von Pipes sowie durch die geringere Rechenleistung pro Pipeline besitzt der Vertex-Prozessor nicht die *Floating Point*-Leistung der Fragment-Einheit.

2.1.5. General Purpose Computations on GPUs

Durch die schnell steigende Rechenleistung und die wachsende Programmierbarkeit wurden Grafikkarten in den vergangenen Jahren auch für die Berechnung von nicht-graphischen Algorithmen interessant, die so genannten *General Purpose Computations on GPUs*, kurz *GPGPU*. Obwohl die Arbeiten in diesem Bereich sich fast ausschließlich auf den akademischen Bereich beschränken, wurden in den vergangenen Jahren bereits zahlreiche Verfahren auf der GPU implementiert; zum Beispiel *direct solvers for sparse matrices* [KRU03] oder verschiedene Algorithmen aus der Strömungsmechanik [HAR02]. Eine gute Übersicht über die bisherigen Arbeiten gibt [GPG04].

[5] MIMD: Multiple Instruction – Multiple Data

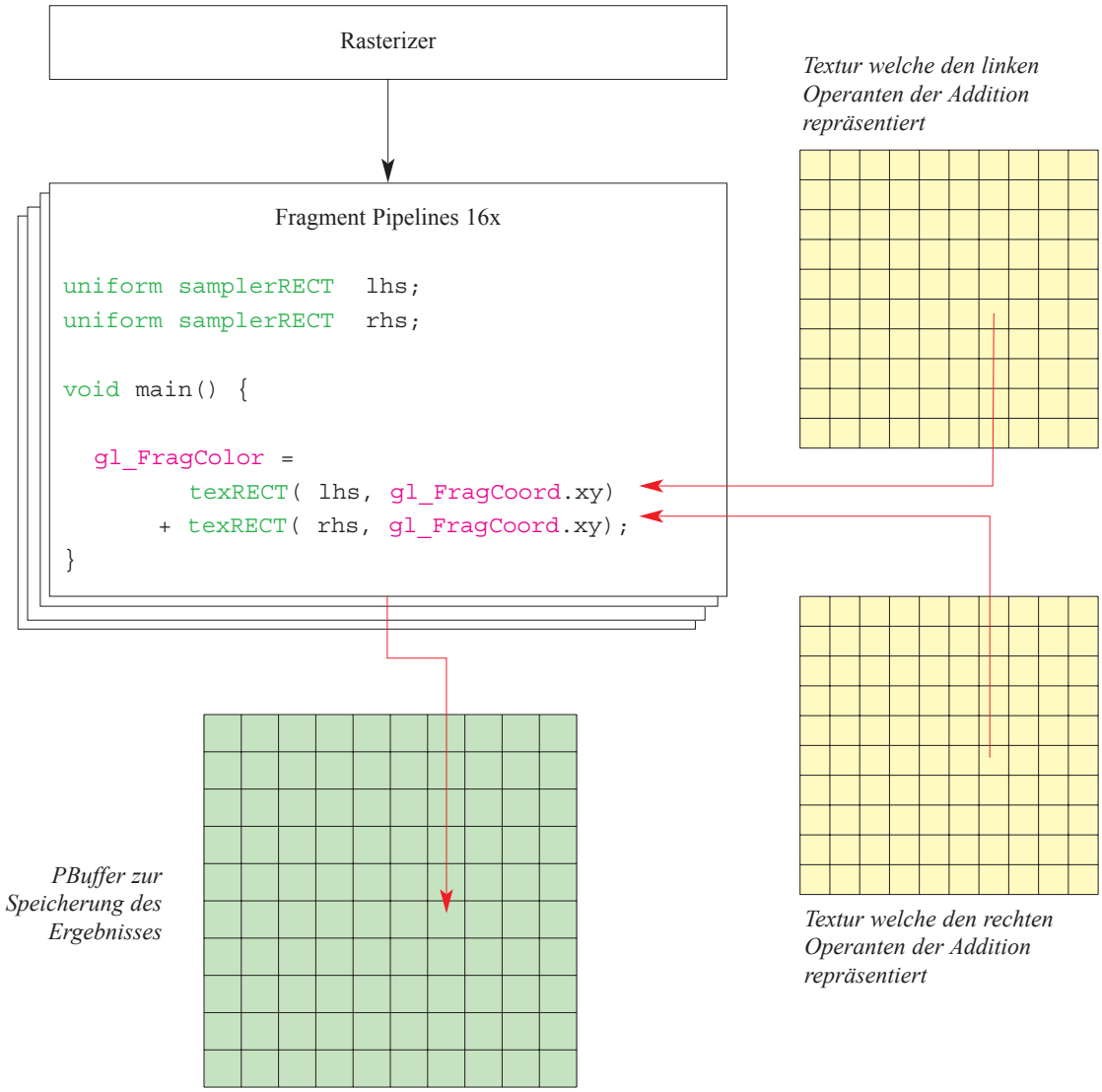


Abbildung 9: Schematische Ablauf einer Vektoraddition auf der Grafikkarte

Für die Abbildung von Algorithmen auf die GPU wird diese meist als *Stream Processor* aufgefasst [PUR04]. Eine *Stream Processor* führt dabei auf einer Menge von Eingangsdaten gleiche Operationen aus, was, wie auf der Grafikkarte, die Parallelisierung der Berechnungen erlaubt. Die Programme, welche die auszuführenden Operationen definieren werden als *Kernel* bezeichnet. Für eine effiziente Implementierung eines Algorithmus auf der GPU ist neben der Parallelisierbarkeit auch die *Arithmetic Intensity* wichtig. Darunter wird das Verhältnis zwischen Rechenoperationen und dafür notwendigen Datentransfer verstanden. Auf der Grafikkarte sollten dabei möglichst viele Berechnungen mit möglichst geringem

Datentransfer durchgeführt werden. Dadurch ist es bei der Programmierung von Stream-Prozessoren oft günstiger Berechnungen mehrfach auszuführen, wenn dadurch der Datentransfer reduziert werden kann.

Die typische Vorgehensweise und einige Probleme bei der Abbildung eines Algorithmus auf die GPU sollen am Beispiel der Vektor-Addition zweier Vektoren $\vec{a}$ und $\vec{b}$ mit jeweils 1024 Elementen erläutert werden (Abb. 9). Die Vektoren sollen dabei im reellen Zahlenraum R^{1024} definiert sein und auf dem Rechner durch 32-Bit *Floating Point* Zahlen repräsentiert werden.
Für die Durchführung der Berechnungen ist zunächst das Speichern der Vektoren auf der GPU notwendig. Allerdings steht dazu auf der Grafikkarte kein Hauptspeicher wie auf der CPU zur Verfügung. Für die Durchführung der Berechnungen ist jedoch nur das Lesen der Daten von $\vec{a}$ und $\vec{b}$ notwendig, so dass der read-only Textur-Speicher der Grafikkarte zum Speichern der Vektoren verwendet werden kann. OpenGL Standard-Texturen können dafür nicht benutzt werden, da diese nur acht Bit pro Farbkanal besitzen. Aus diesem Grund müssen die Daten in so genannten *Floating Point Textures* [NVI03] [ATI02] gespeichert werden, welche eine Bit-Tiefe von 32-Bit pro Farbkanal haben können.
Eine erste Idee, um die Vektor-Addition auf der GPU durchzuführen, wäre das Rendern eines ein Pixel großen Punktes. Durch das Rendern würde ein Fragment erzeugt und das Fragment-Shader-Programm einmal ausgeführt. In dem Programm könnte, in einer Schleife über alle Elemente der Vektoren, die Addition durchgeführt werden. Eine solche Berechnung wäre, zumindest auf der NV4X-Architektur möglich, allerdings könnte der Ergebnisvektor nicht gespeichert werden, da für jedes Fragment nur vier Werte, jeweils einer pro Farbkanal, geschrieben werden kann. Die Verwendung von *Multiple Render Targets (MRTs)* bietet in diesem Fall auf aktuellen Grafikkarten keinen Ausweg, da maximal vier *Render Targets* zur Verfügung stehen, somit also höchstens 16 Werte pro Fragment geschrieben werden können. Durch das *Packing* [NVI04a] von jeweils zwei 16-Bit Ergebniswerten in einen 32-Bit Kanal der *Render Targets* wäre zwar das Speichern von bis zu 32 Werten möglich, allerdings wäre das für die 1024 Elemente des Ergebnisvektors der Addition ebenfalls nicht ausreichend. Darüber hinaus führt *Packing*, durch die geringere Anzahl verwendeter Bits für jeden Wert, zu einem Genauigkeitsverlust.
Um diese Beschränkungen zu vermeiden, wird für GPGPU Berechnungen im

Allgemeinen ein Fragment zur Berechnung eines Wertes verwendet. Eine solche Aufteilung ist auch zur Nutzung der parallelen Einheiten der GPU notwendig. Für das Beispiel der Vektor-Addition wird damit jeweils eine Dimension des Ergebnisvektors in einem Fragment beziehungsweise dem dazugehörigen Fragment-Programm berechnet. Um die Berechnungen durchzuführen, muss das Fragment-Shader-Programm für 1024 unterschiedliche Fragmente genau einmal aufgerufen werden. Dies wird durch das Rendern eines einfachen OpenGL Primitives, meist eines Quadrats, eines so genannten *Quads*[6], erreicht.

Bis jetzt unberücksichtigt geblieben ist die limitierte Bit-Tiefe des *Frame Buffers*, welche bei heutigen Grafikkarten bei acht Bit liegt. Diese Beschränkung kann durch die Verwendung von *Pixel Buffers*, kurz *PBuffers*, vermieden werden [SGI97] [WGL02a], welche anstatt des *Frame Buffers* als *Render Target* definiert werden. Die Bit-Tiefe pro Farbkanal kann bei diesen, da sie nicht auf dem Bildschirm darstellbar sind, unabhängig vom *Frame Buffer* definiert werden. Wie bei *Floating Point Textures* sind maximal 32-Bit pro Farbkanal möglich.

Bei der in Abbildung 9 schematisch dargestellten Implementierung wird bereits eine Optimierung verwendet, da in jedem Fragment die Addition für vier Dimensionen der Vektoren berechnet wird. Dies ermöglicht einen effizienteren Zugriff auf die Daten, da keine Adressierung eines Farbkanals respektive Elementes innerhalb der aus den Texturen ausgelesenen Werte notwendig ist. Darüber hinaus reduziert diese Optimierung den für die Texturen und den *PBuffer* benötigt Speicherplatz.

Das Ergebnis der Addition liegt nach dem *Render Pass*, der Durchführung der Berechnungen, im *PBuffer* vor. Die Daten können anschließend entweder zur OpenGL Client-Seite zurück gelesen, oder für die Verwendung auf der GPU in eine Textur kopiert werden.

In dem einfachen Beispiel der Vektoraddition ist nur ein *Rendering Pass* für die Berechnung des Ergebnisses notwendig. Bei den meisten GPGPU-Berechnungen muss jedoch ein *Multi Pass Rendering* Verfahren verwendet werden; entweder weil die maximal mögliche Programmlänge für die Durchführung der Berechnungen in einem Programm nicht ausreichend ist, oder weil für die Berechnung Zwischener-

[6] Bei der Verwendung eines Quads wird die Anzahl der durch die Projektion entstehenden Fragmente über die Viewport Größe definiert, was den entstehenden Overhead sowohl in Bezug auf den benötigten Speicher als auch in Bezug auf die notwendigen Rechenoperationen mininiert. Das Quad wird mit Parallelprojektion gerendert. Dadurch sind keine Rechenoperationen im Vertex-Shader notwendig

gebnisse aus benachbarten Fragmenten benötigt werden. Der Algorithmus wird dabei in Teilschritte zerlegt und jeder der Schritte in einem *Rendering Pass* berechnet. Nach jedem *Pass* wird das Zwischenergebnis in eine Textur kopiert und im nächsten Schritt des Verfahrens von dort wieder ausgelesen.

Der Vertex-Prozessor wurde bisher, wie bei den meisten GPGPU Berechnungen, nicht in Betracht gezogen. Im Beispiel der Vektor-Addition lag dies an der größeren Rechenleistung der Fragment-Einheit; im allgemeinen Fall kommt noch die im Vertex-Shader geringere maximale Programmlänge hinzu. Zusätzlich besteht bei der Durchführung von Berechnungen im Vertex-Shader das Problem, dass die Ergebniswerte zunächst in die Fragment-Einheit transportiert werden müssen, bevor sie in das *Render Target* geschrieben werden. Ein einfacher Transport der Daten ist dabei, insbesondere auf Grund der Interpolation der *Varying Variables*, nicht immer möglich.

Trotzdem könnte sich dieses Ungleichgewicht zwischen Vertex- und Fragment-Einheit durch die Einführung der NV4X-Architektur verändern. Zum einen, da auf dieser Architektur auch im Vertex-Shader das Auslesen aus Texturen möglich ist, zum anderen durch die in der Vertex-Einheit verwendete MIMD-Architektur. Diese gewährleistet, im Gegensatz zur SIMD-Architektur in der Fragment-Einheit, das Programme auch bei nicht-kohärentem Programmfluss von parallelen Datenelementen effizient verarbeitet werden können.

2.1.6. Zusammenfassung

Grafikkarten bieten heute durch ihre enorme Rechenleistung die mit Abstand schnellsten OpenGL Implementierung. Durch die Einführung von programmierbaren Einheiten wurden dabei nicht nur zusätzliche visuelle Effekte sondern auch die Durchführung von nicht-graphischen Berechnungen auf GPUs möglich.

Es können aber, wie zum Beispiel auch [FAT04] zeigt, nicht alle Berechnungen und Verfahren effizient auf aktuellen Grafikkarten implementiert werden. Insbesondere die Parallelisierbarkeit eines Algorithmus ist für eine effiziente Abbildung auf die GPU wichtig. Darüber hinaus beeinflussen zahlreiche interne Mechanismen der Grafikkarte die Effizienz. Diese sind allerdings nur unvollständig veröffentlicht, so dass die Effizienz eines Verfahrens auf der GPU im Allgemeinen nur nach einer Implementierung bewertet werden kann.

2.2 Ray-Tracing

Im Folgenden soll zum besseren Verständnis der Arbeit, ein Überblick über das Ray-Tracing-Verfahren in der von Whitted [WHI80] vorgeschlagenen Variante, dem so genannten *Whitted Ray-Tracing*, gegeben werden. Eine ausführliche Erläuterung des Ray-Tracing-Verfahrens findet sich in [GLA89].

2.2.1. Whitted Ray-Tracing

Die Farbe eines Pixels wird beim Ray-Tracing durch das Aussenden von Strahlen in die Szene ermittelt. Dabei wird, ausgehend vom Augenpunkt, jeweils ein Stahl durch jedes Pixel in die Szene gesendet und dort weiter verfolgt. Für alle dieser so genannten Augenstrahlen wird anschließend die Strahlanfrage durchgeführt (Abb.11). Dabei wird für jeden Strahl der dem Strahlursprung am nächsten liegende Schnittpunkt mit den Objekten der Szene ermittelt (Abb. 11a). Existiert ein solcher Schnittpunkt so bestimmt dessen Farbwert die Farbe des Pixels, durch welches der Augenstrahl gesendet wurde.
Die Farbe des Schnittpunktes hängt sowohl von der direkten als auch von der indirekten Beleuchtung ab. Die Beleuchtung wird durch Sekundärstrahlen ermittelt. Zur Berechnung der direkten Beleuchtung wird jeweils ein Sekundärstrahl vom Schnittpunkt zu jeder Lichtquelle verfolgt (Abb. 11b). Schneidet einer dieser so

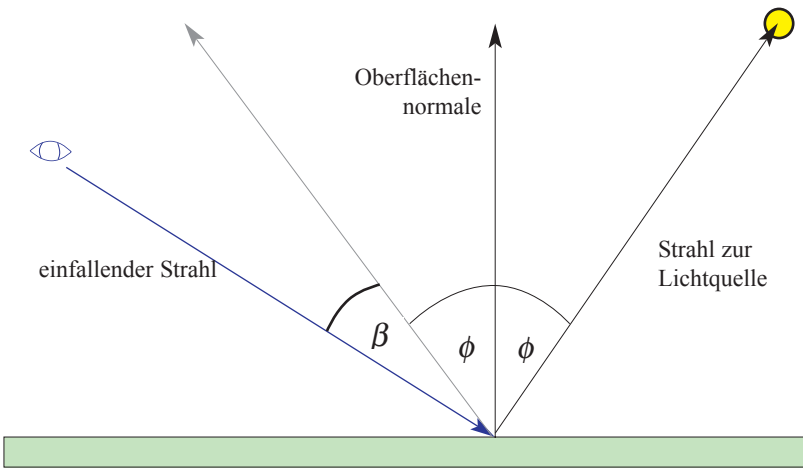
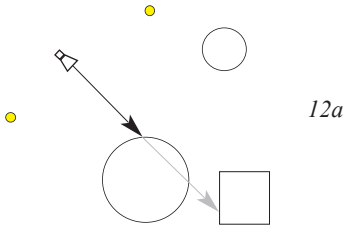


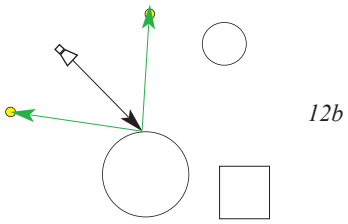
Abbildung 10: Phong-Shading. Für die Berechnung der direkten Beleuchtung benötigte Strahlen.

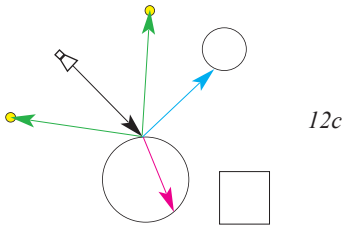
genannten Schattenstrahlen auf seinem Weg zur Lichtquelle ein anderes Objekt der Szene, so ist der Ursprungspunkt des Strahles bezüglich der Lichtquelle abgeschattet. Anderenfalls kann mit Hilfe von Beleuchtungsmodellen, wie dem *Phong Shading* [BUI75], der Farbanteil, der durch diese Lichtquelle zur Farbe des Pixels beiträgt, berechnet werden (Abb. 10). Die Summe aller Farbbeiträge aller Lichtquellen ergibt dabei den Gesamtfarbbeitrag durch die direkte Beleuchtung (Abb. 12).

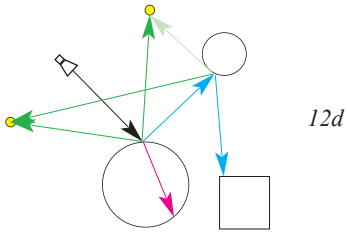
Zur Berechnung der indirekten Beleuchtung werden beim Whitted Ray-Tracing für jeden Schnittpunkt jeweils genau ein Reflexions- und genau ein Refraktionsstrahl erzeugt. Die Richtung der Strahlen kann mit Hilfe der Gesetze der Strahlenoptik berechnet werden. Diese Strahlen werden bei der anschließenden Verfolgung durch die Szene analog zu den Augenstrahlen behandelt; zunächst wird für jeden Strahl die Strahlanfrage durchgeführt, anschließend werden zur Berechnung der Farbwerte der gefundenen Schnittpunkte neue Sekundärstrahlen erzeugt und ebenfalls in die Szene verfolgt (Abb. 11c, 11d).

Der Beitrag eines Reflexions- beziehungsweise Refraktionsstrahl zum Farbwert seines Ursprungs-Schnittpunktes ergibt sich durch den Farbwert des Schnittpunktes des Strahles gewichtet mit dem Reflexions- beziehungsweise Refraktionskoeffizienten. Die Erzeugung von Sekundärstrahlen für einen Schnittpunkt wird meist dann abgebrochen, wenn der Beitrag der Sekundärstrahlen zum Gesamtfarbwert des Pixels unter einem definierten Grenzwert fällt. Dieses von Hall vorgeschlagene Verfahren wird als *adaptive tree depth control* bezeichnet [HAL83].









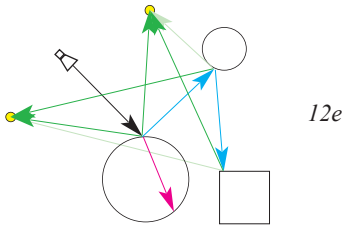


Abbildung 11 a-f: Schematischer Ablauf des Ray-Tracing-Verfahrens

schwarz: Augenstrahl
grün: Strahlen zu den Lichtquellen
cyan: Reflexionsstrahlen
magenta: Refraktionsstrahlen

$$I_d(\lambda) = \sum_{n=1}^{N} I_n(\lambda) * \delta(I_n) * \left[k_d(\lambda) * \cos(\phi_n) + k_s(\lambda) * \cos^m(\beta_n) \right]$$

N : Anzahl der Lichtquellen in der Szene

$I_n(\lambda)$: Von der Lichtquelle emittiertes Licht der Wellenlänge λ

$$\delta(I_n) = \left\{ \begin{array}{l} 0 \text{ , falls der Schnittpunkt bezüglich der n - ten Lichtquelle abgeschattet ist} \\ 1 \text{ , anderenfalls} \end{array} \right\}$$

k_d : diffuser Beleuchtungskoeffizient der Oberfläche des geschnittenen Objektes

k_S : spekulare Beleuchtungskoeffizient der Oberfläche des geschnittenen Objektes

Abbildung 12: Beleuchtungsberechnung beim Ray-Tracing.
Gleichung zur Berechnung der direkten Beleuchtung für alle Lichtquellen einer Szene.

Für eine korrekte Beleuchtung sind die Berechnungen für alle Wellenlängen im Bereich des sichtbaren Lichtes durchzuführen. In der Praxis wird dies jedoch durch eine beschränkte Anzahl von Wellenlängen approximiert, zum Beispiel mit rot (R), grün (G) und blau (B).

2.2.2. Komplexität und Beschleunigung des Ray-Tracing-Verfahrens

Eines der größten Probleme des Ray-Tracing-Verfahrens ist die effiziente Berechnung der Bildinformation. Der intuitive Ansatz, bei dem jeder Strahl mit jedem Objekt in der Szene geschnitten wird, besitzt eine Komplexität von *O(N)* für die Anzahl der Objekte in der Szene. Für Szenen mit einer großen Anzahl von Objekten führt dies, wie beim *Feed Forward Rendering*, zu inakzeptabel langen Berechnungszeiten. Seit der Einführung des Ray-Tracing-Verfahrens spielen deshalb Beschleunigungsansätze eine wesentliche Rolle (Abb. 13).

Der wichtigste Beschleunigungsansatz ist die Reduzierung der Rechenzeit pro Bild, welche für die Schnittberechnung benötigt wird. Neben der Zeit, welche für jeden einzelnen Schnitt-Test notwendig ist, ist die Anzahl der Tests von entscheidender Bedeutung. Um diese zu reduzieren wurden verschiedene *Beschleunigungsstrukturen* vorgeschlagen. Die wichtigsten für beliebige Strahltypen sind dabei Hüllkörper-Hierarchien, Raumteilungsverfahren, wie *Octrees* [GLA84], *BSP*- und *kd-Bäume* [FUC80] [BEN75] und Strahlrichtungs-Techniken [ARV89], wie *Ray Coherence*

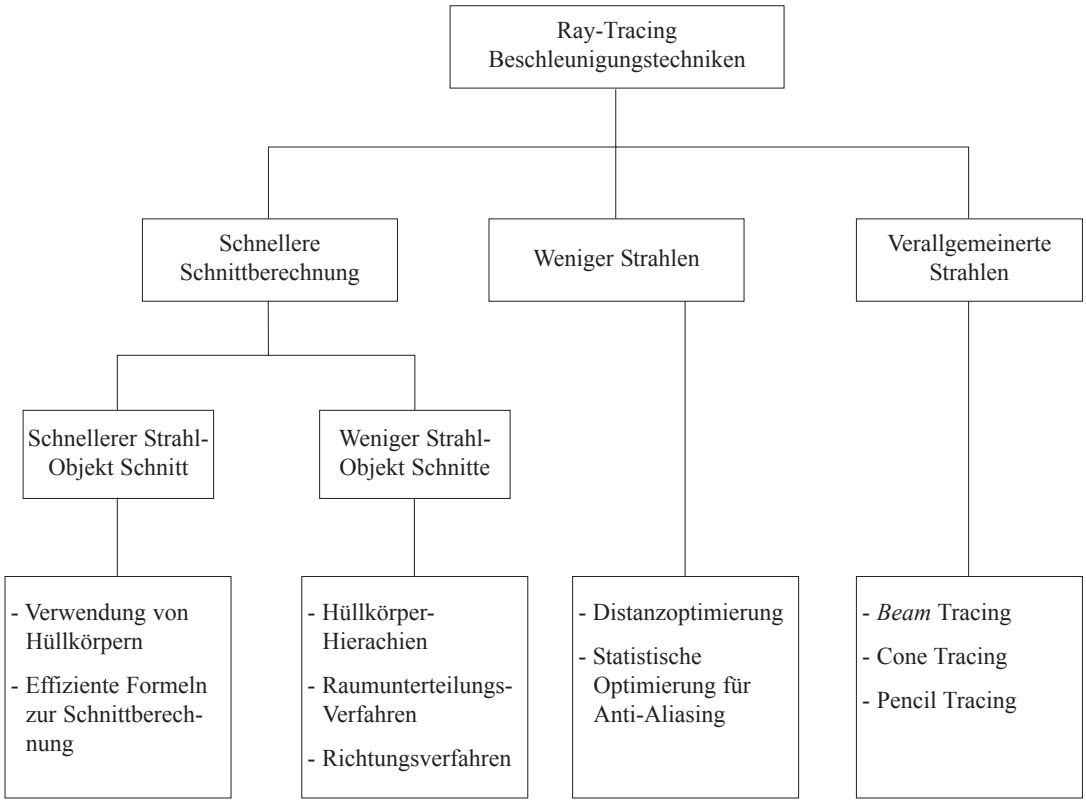


Abbildung 13: Beschleunigungsansätze beim Ray-Tracing (nach [KIR89]*).*

[OHT87] oder *Ray Classification* [ARV87]. Für spezielle Strahltypen wie Schattenstrahlen, existieren weitere Beschleunigungsstrukturen, zum Beispiel der *Light Buffer* [HAI86]. Neben einer einfacheren Implementierung sind diese meist effizienter als die allgemeinen Beschleunigungsstrukturen.

Allen Verfahren ist dabei gemeinsam, dass sie für jeden Strahl R eine Teilmenge T von Objekten der Gesamt-Objektszene S ermitteln, welche für einen Strahl-Objekt-Schnitt in Frage kommt. Die Beschleunigung wird durch die Elemente der Differenzmenge D von S und T erreicht, für welche keine Schnitt-Tests notwendig sind, um die Strahlanfrage von R bezüglich S zu beantworten. Die Beschleunigungsstruktur gewährleistet dabei, dass die Anzahl der Elemente in T, unabhängig von der Größe von S, nahezu konstant bleibt. Durch die hierarchische Organisation der Beschleunigungsstruktur kann die für einen Strahl relevante Teil-Objektszene T in logarithmischer Zeit bestimmt werden. Die Komplexität des Ray-Tracing-Verfahrens kann damit auf $\log(N)$ für die Anzahl der Objekte reduziert werden.

Über die Effizienz der verschiedenen Beschleunigungsstrukturen gab es in den ver-

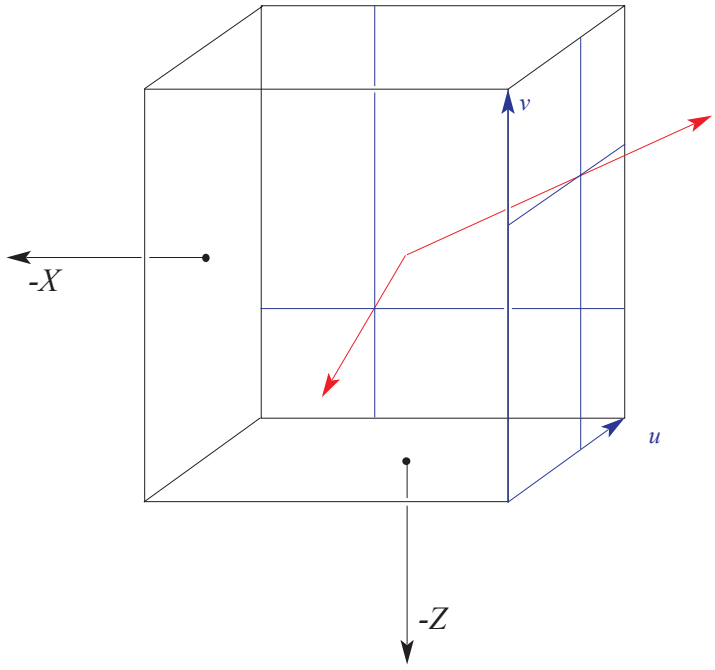


Abbildung 14: Richtungskubus mit zwei Strahlen (rot), sowie deren Repräsentation in uv-Koordinaten auf der durch ihre Strahl-Hauptrichtung festgelegten Kubusseite

gangenen Jahren widersprüchliche Aussagen [HAV01]. Der Hauptgrund dafür war das Fehlen eines unabhängigen und umfassenden Vergleichs der verschiedenen Ansätze. Eine der ersten ausführlichen Untersuchungen findet sich dabei in [HAV01]. Für die dort verwendeten Szenen aus der *Standard Procedural Database* [HAI87] war der kd-Baum die durchschnittlich schnellste Beschleunigungsstruktur. Für eine ausführlichere Analyse der Komplexitätsreduktion und Effizienzsteigerung durch Beschleunigungsstrukturen sei hier auf [SZI96], [SZI97], [SZI98a], [SZI98b], [SZI02] sowie [HAV01] verwiesen.

2.2.3. Strahlrichtungs-Techniken

Die Strahlrichtungs-Techniken sind für das Verständnis der Arbeit von weiterführender Bedeutung, so dass diese hier näher erläutert werden sollen.
Die Idee kann mit Hilfe des *Richtungskubus* (Abb. 14), dessen Zentrum im Ursprung des Weltkoordinatensystems liegt, erläutert werden. Dieser Kubus erlaubt es, jeden Strahl in der Szene, entsprechend seiner Hauptrichtung +X, -X, +Y, -Y, +Z oder -Z,

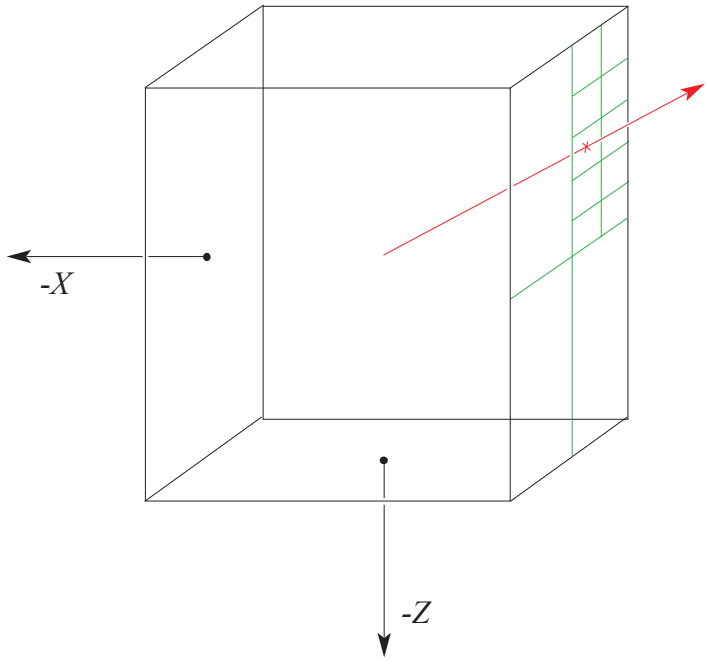


Abbildung 15: Richtungskubus mit regulären Gitter als Beschleunigungsstruktur im Strahlraum

einer der Seiten der Kubus zuzuordnen. Jede beliebige Strahlrichtung kann dabei mit zwei, im Allgemeinen als *u* und *v* bezeichneten Koordinaten, auf der, durch die Strahlhauptrichtung vorgegebenen Seite des Kubus beschrieben werden. Analog zur Hierarchiebildung bei Hüllkörper- und Raumteilungsverfahren kann dies auch bei Strahlrichtungs-Techniken erfolgen (Abb. 15). Die Unterteilung des Raumes sowie die Erzeugung der Hierarchie sind dabei durch die zweidimensionale Darstellung der Strahlrichtungen im *uv*-Raum effizient möglich.

Eine Erweiterung dieses Ansatzes ist das von Arvo und Kirk vorgeschlagene *Ray Classification* Verfahren [ARV87]. Bei diesem werden, neben den zwei Freiheitsgraden, welche der Strahl durch seine Richtung hat, auch die drei Freiheitsgrade berücksichtigt, welche durch die beliebige Lage des Strahlursprungs in der Szene vorhanden sind. Ein Strahl entspricht dann einem Punkt im fünfdimensionalen Raum $R^3 \times S^2$. Der zweidimensionale Raum S^2 wird dabei durch die Polarkoordinaten φ und θ auf der Einheitskugel aufgespannt.

Eine Reduktion der Komplexität des Ray-Tracings kann auch beim *Ray Classification* Verfahren durch eine hierarchische Unterteilung des 5D-Raumes erreicht werden. Die bei der Unterteilung entstehenden Intervalle im fünfdimensionalen Raum werden

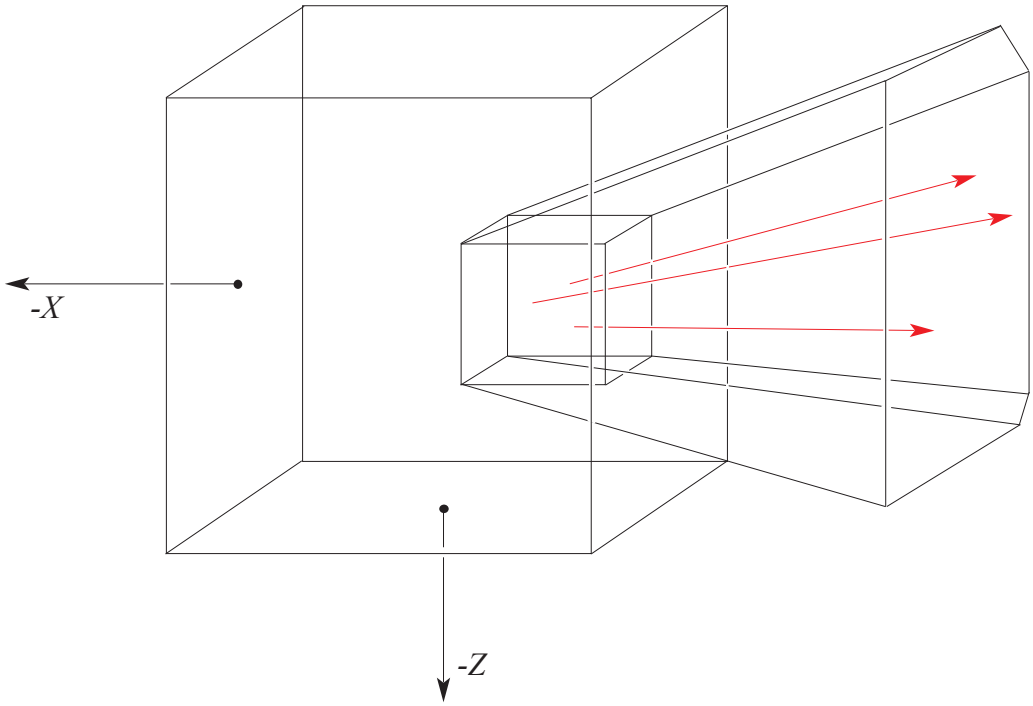


Abbildung 16: Hypercube /Beam mit Strahlen

dabei als *Hypercubes* bezeichnet (Abb. 16). Diese repräsentieren jeweils eine Menge von Strahlen mit ähnlichem Ursprungsgebiet und ähnlicher Richtung. Die Repräsentation eines *Hypercubes* im 3D-Raum ist ein *Beam*.

Zu Beginn des Ray-Tracing-Verfahrens besteht die *Hypercube*-Hierarchie, neben der Wurzel des Baumes, aus sechs Blattknoten. Die Blätter sind dabei die *Beams*, welche durch die sechs Seiten des Richtungskubus und den Richtungskubus selbst aufgespannt werden. Die Größe der *Bounding Box* der Szene bestimmt dabei die minimale Größe des Richtungskubus. Dies ermöglicht, dass alle Strahlen, welche innerhalb der Objektszene ihren Ursprung haben, anhand ihrer Strahlhauptrichtung, in die *Hypercube*-Hierarchie eingefügt werden können. Bei Strahlen deren Ursprung außerhalb der *Bounding Box* der Szene liegt, wird der Strahlursprung auf den ersten Schnittpunkt des Strahles mit der *Bounding Box* der Szene verschoben. Somit können alle Strahlen, welche durch die Objektszene verlaufen und möglicherweise ein Objekt der Szene schneiden, in die *Hypercube*-Hierarchie eingefügt werden. Eine Verschiebung des Strahlursprungs ist dabei möglich, da außerhalb der *Bounding Box* der Szene kein Strahl-Objekt-Schnittpunkt auftreten kann.

Für die Unterteilung des fünfdimensionalen Raumes können adaptierte Raum-

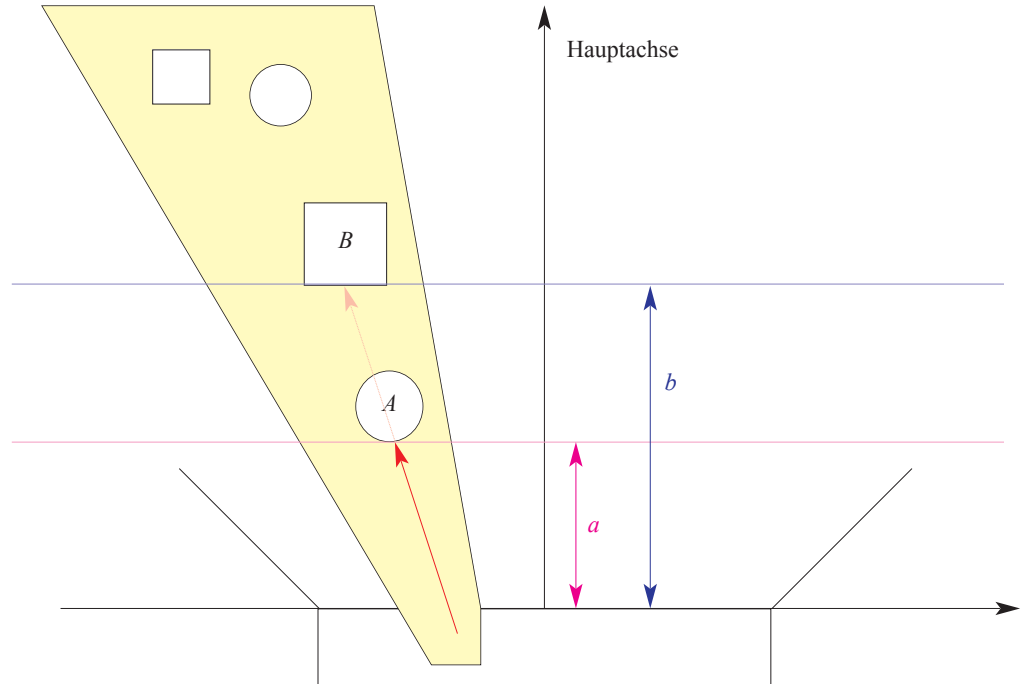


Abbildung 17: Distanzoptimierung bei Strahl-Richtungs-Techniken. Da a kleiner als b ist, muss kein Schnitt-Test *des Strahls (rot) mit B durchgeführt werden.*

teilungsverfahren aus dem 3D-Raum verwendet werden. So schlagen Arvo und Kirk zum Beispiel den *hyper-octree*, das Analogon zum *Octree* im fünfdimensionalen Raum, vor. Da der Speicherbedarf des *Ray Classification* Verfahrens sehr groß werden kann, erfolgt die Unterteilung zur Laufzeit, und nicht, wie bei Raumteilungsverfahren im dreidimensionalen Raum üblich, als Vorberechnungsschritt. Eine Reduktion des Speicherbedarfs wird erreicht, da nur in den Intervallen eine Unterteilung erfolgt, in denen tatsächlich Strahlen vorhanden sind. Diese Vorgehensweise wird als *lazy evaluation* bezeichnet.

Unabhängig von der gewählten Unterteilungsstrategie besitzen die *Hypercubes*, welche Blätter der resultierenden Hierarchie sind, eine *Objekt-* oder *Kandidatenliste*. Diese enthält alle Objekte der Szene, welche möglicherweise von einem der Strahlen des *Hypercube* geschnitten werden, welche also teilweise oder vollständig im *Beam* liegen.

Eine Strahlanfrage erfolgt beim *Ray Classification* Verfahren durch die Auswahl eines Blattknotens der *Hypercube*-Hierarchie und die Durchführung der Schnitt-Tests für alle in diesem Knoten enthaltenen Strahlen.

Neben einer Hierarchie kann beim *Ray Classification* Verfahren die

Distanzoptimierung zur Reduzierung der Anzahl der Strahlanfragen verwendet werden. Unter Distanzoptimierung versteht man dabei, dass keine weiteren Schnitt-Tests notwendig sind, wenn ausgeschlossen werden kann, dass bei einer Fortsetzung der Schnitt-Tests ein neu gefundener Schnittpunkt näher am Augenpunkt liegt, als ein bereits gefundener. Wie Abbildung 17 zeigt, ist dies für einen Strahl in einem *Beam* dann der Fall, wenn die Distanz *a* eines bereits gefundenen Schnittpunktes entlang der Hauptachse vom Strahlursprung geringer ist, als die Distanz *b* der *Bounding Box* eines Objektes entlang der Hauptachse. In diesem Fall ist kein Schnitt-Test mit *B* und allen in der tiefensortierten Liste folgenden Objekten notwendig. Um die Distanzoptimierung beim *Ray Classification* Verfahren verwenden zu können, ist eine Tiefensortierung der Objektlisten bezüglich der Hauptachse des *Hypercubes* notwendig. Diese kann bereits vor Beginn der Unterteilung in der initialen *Hypercube*-Hierarchie erfolgen, da durch die Intervallzerlegung bei der Erzeugung der Hierarchie keine Veränderung der Reihenfolge stattfindet.

2.2.3. Varianten des Ray-Tracings

In der Literatur existieren verschiedene Varianten des Ray-Tracing-Verfahrens. Die bekanntesten sind dabei das bereits erläuterte *Whitted Ray-Tracing* [WHI80], *Monte Carlo Ray-Tracing* [HAL70] und *Ray-Casting*.
Wie bereits erläutert, wird beim Whitted Ray-Tracing an jedem Strahl-Objekt-Schnittpunkt genau ein Reflexions- und Refraktionsstrahl, sowie ein Strahl zu jeder Lichtquelle erzeugt. Dies ermöglicht eine gute Approximation von Reflexion, Refraktion und harten Schatten. Um auch diffuse Beleuchtung und weiche Schatten mit dem Ray-Tracing-Verfahren berechnen zu können, wurde das Monte Carlo Ray-Tracing entwickelt. Dabei werden, ausgehend von jedem Schnittpunkt mehrere Reflexions- und Refraktionsstrahlen, sowie mehrere Strahlen zu jeder Lichtquelle in die Szene verfolgt. Im Vergleich zu Whitted Ray-Tracing ist für das Monte Carlo Ray-Tracing ein höherer Berechnungsaufwand notwendig.
Die minimale Form des Ray-Tracing-Verfahrens ist das Ray-Casting. Dabei werden nach der Ermittlung der Schnittpunkte der Augenstrahlen keine Sekundärstrahlen erzeugt, wodurch keine Berechnung von Spiegelung, Brechung und Schatten, respektive globalen Beleuchtungseffekten möglich ist.

Kapitel 3
Planung des Ray-Tracing-Systems

Die Aufgabenstellung eines interaktiven Ray-Tracing-Systems machte keine Vorgaben

- bezüglich der zu verwendenden Hardware.
- bezüglich der zu benutzenden Variante des Ray-Tracing-Verfahrens.
- bezüglich der zu verwendenden Beschleunigungsstruktur(en).

In der ersten Phase der Arbeit war deshalb eine geeignete Implementierung für die Zielstellung zu entwickeln.

3.1. Auswahl der Hardware

In Bezug auf die für das Ray-Tracing-System einzusetzende Hardware existierten zu Beginn der Arbeit drei Alternativen:

- CPU-basierte Ray-Tracing-Systeme, sowohl auf Einzel-Rechnern als auch auf Clustern
- GPU-basierte Ray-Tracing-Systeme
- Kombinierte CPU- und GPU-basierte Ray-Tracing-Systeme

Es existieren weitere Hardwaresysteme, für welche gezeigt wurde, dass Ray-Tracing auf diesen möglich ist [STR92] [ANA96]. Da diese jedoch nicht für den praktischen Teil der Arbeit zur Verfügung standen, werden sie in den weiteren Betrachtungen nicht berücksichtigt.

3.1.1. CPU-basiertes Ray-Tracing

CPU-basierte Ray-Tracing-Systeme existieren bereits seit mehr als 30 Jahren [MAG68]. Auf Grund der des großen Berechnungsaufwands des Ray-Tracing-Verfahrens, war dabei lange Zeit kein interaktives Rendering möglich. Die Arbeiten von Wald [WAL04a] und anderen zeigen jedoch, dass seit einiger Zeit das Ray-Tracing, auch von großen bis sehr großen Szenen [WAL04b], bei interaktiven Bildwiederholraten auf der CPU möglich ist.
Die Vorteile des CPU-basierten Ray-Tracings liegen unter anderem in einer Vielzahl von vorhandenen Implementierungen und untersuchten Ansätzen, sowie den ausgereiften Programmiersprachen und -modellen. Bei Clustern kommt darüber hinaus die gute Skalierbarkeit hinzu.
Allerdings ist für die Verwendung von Clustern ein hoher administrativer Aufwand notwendig, so dass diese zur Zeit hauptsächlich von Forschungseinrichtungen und große Firmen eingesetzt werden. Ein Problem beim Ray-Tracing auf Einzel-Rechnern ist, dass nahezu die gesamte Rechenleistung der CPU für das Ray-Tracing benötigt wird. Bei den meisten graphischen Anwendungen sind jedoch weitere Berechnungen notwendig, zum Beispiel physikalische Simulationen oder die Verwaltung von Teile-Datenbanken. Dadurch ist CPU-basiertes Ray-Tracing auf Einzel-Rechnern zur Zeit nicht, beziehungsweise nur eingeschränkt für Anwendungen einsetzbar.

3.1.2. GPU-basiertes Ray-Tracing

Ein GPU-basierten Ray-Tracing-System wurde zuerst von Purcell [PUR02] im Jahr 2002 vorgeschlagen. Die erste Implementierung war im Jahr 2003 möglich, nach dem die Grafikkarten die dafür notwendige Programmierbarkeit erreicht hatten.
Dabei konnte gezeigt werden, dass auf der Grafikkarte eine mit CPU-

Implementierungen auf Einzel-Rechnern vergleichbare Leistung erreicht werden kann. Die eingeschränkte Programmierbarkeit, das Hauptproblem bei einer Implementierung auf der GPU, wurde dabei von der deutlich höheren Rechenleistung aufgewogen.
Auf Grund des größeren Wachstums der Rechenleistung (Abb. 4) sowie der sich ständig verbessernden Programmierbarkeit der Grafikkarten, ist für GPU-basierte Ray-Tracing-Systeme eine deutlich schnellere Geschwindigkeitssteigerung zu erwarten, als bei CPU-Implementierungen. Dabei ist aber zu berücksichtigen, dass das von Purcell vorgeschlagene Verfahren noch einige Beschränkungen aufweist, welche verhindern, dass es als graphisches Backend von 3D-Anwendungen verwendet werden kann.

3.1.3. CPU- und GPU basiertes Ray-Tracing

Die Nutzung von CPU und GPU bei der Implementierung des Ray-Tracing-Verfahrens bietet durch die Kombination der Rechenleistung und der unterschiedlichen Programmiermodelle einige Vorteile.
Im Vergleich zu ausschließlich CPU-basierten Ray-Tracing-Systemen ist durch den Einsatz der GPU eine Entlastung der CPU möglich. Auf dieser steht damit mehr Rechenzeit für nicht-graphische Aufgaben zur Verfügung. Darüber hinaus sind auf der GPU, durch die höhere Rechenleistung, mehr Strahl-Objekt Schnitt-Tests pro Sekunde möglich als bei CPU-basierten Ray-Tracing-Systemen.
Im Vergleich zu ausschließlich GPU-basierten Implementierungen entsteht durch die Verwendung der CPU eine größere Flexibilität bezüglich der einsetzbaren Algorithmen. Alle Verfahren welche sich nicht, beziehungsweise nicht effizient auf die GPU abbilden lassen können auf der CPU implementiert werden.
Ein weiterer Vorteil eines nicht nicht nur GPU-basierten Systems ist die Möglichkeit einer Kombination beziehungsweise Integration eines Szenegraphen mit der Beschleunigungsstruktur. Insbesondere bei komplexen, dynamischen Szenen führt dies zu einer Effizienzsteigerung. Es kann dabei ein *lazy evaluation* Verfahren zur Verarbeitung des Szenegraphen verwendet werden. Dieser wird dabei nur für die Teile der Szene evaluiert, in welchen sich tatsächlich Strahlen befinden; dass heißt die Bereiche, welche zum dargestellten Bild beitragen.

	AGP8x[7]	PCI express (16 lanes)
Read	266 MB/s	4 GB/s
Write	2128 MB/s	4 GB/s

Abbildung 18: Datentransferraten von AGP8x [AGP98] und PCI express [PCI04], [TWE04] mit 16 lanes; Read: Readback von Daten in den Hauptspeicher, Write: Schreiben von Daten aus in den GPU-Speicher.

Ein erstes CPU- und GPU-basiertes Ray-Tracing-System wurde von Carr et. al. vorgeschlagen [CAR02]. Als Hauptproblem zeigte sich dort die zu geringe Datentransferrate beim *Readback* aus dem GPU- in den CPU-Speicher. Eine deutliche Verbesserung des Readbacks wurde jedoch für den Sommer 2004 mit der Einführung von *PCI express (PCIe)* [PCI04] angekündigt (Abb. 18).

Auf Grund der bei *PCIe* deutlich verbesserten Datentransferraten, sowie der erläuterten Vorteile eines kombinierten CPU- und GPU-basierten Ray-Tracing-Systems, erschien es sinnvoll, diesen Ansatz weiterzuverfolgen. Weiterhin bot sich dies auch auf Grund eines neuen, GPU-basierten Beschleunigungsansatzes für die Augenstrahlen an, der in ein solches System integriert werden kann.

Das von Carr et. al. vorgeschlagene Verfahren wurde für diese Arbeit nicht aufgegriffen. Die dort verwendeten Grafikkarten unterscheiden sich wesentlich von den NV4X-Architektur, darüber hinaus wurde sich in [CAR02] auf Dreiecke beschränkt.

3.2. Auswahl einer Variante des Ray-Tracing-Verfahrens

Wie unter 2.2. vorgestellt wurde, existieren verschiedene Varianten des Ray-Tracing-Verfahrens. Ray-Casting wurde nicht weiter in Betracht gezogen, da es a priori keine Berechnung von globalen Beleuchtungseffekten ermöglicht.
Neben der realistischen Beleuchtung waren interaktive Bildwiederholraten ein wichtiges Ziel des geplanten Ray-Tracing-Systems. Aus diesem Grund wurde in der Arbeit das von Whitted vorgeschlagene Verfahren verwendet, für welches, im Vergleich zum Monte Carlo Ray-Tracing, ein geringerer Rechenaufwand notwendig ist.

[7] Der *Readback* ist auch bei AGP8x nur mit AGP1x möglich. Durch proprietäre Erweiterungen kann bei der Verwendung von Nvidia Grafikkarten mit AGP8x allerdings ein *Readback* von 800 MB/s erzielt werden.

3.3. Aufteilung des Ray-Tracing-Verfahrens

Für eine effiziente Aufteilung des Ray-Tracing-Verfahrens auf CPU und GPU ist es zunächst notwendig, die Unterschiede zwischen den zwei Prozessoreinheiten zu betrachten.
Auf die Eigenschaften von aktueller Grafikhardware wurden bereits unter 2.1. eingegangen; die wesentlichen Charakteristika der NV4X-Architektur sind die große Rechenleistung für *Floating Point*-Operationen, die nur eingeschränkte Programmierbarkeit sowie die SIMD-Architektur der Fragment-Einheit.
Im Vergleich zur GPU ist auf der CPU eine sehr flexible Programmierung möglich. Dabei stehen hochentwickelte Programmiersprachen und -modelle zu Verfügung, welche zum Beispiel eine variable und effiziente Speicherverwaltung ermöglichen.

Aus diesen Gründen erschien die GPU für die Implementierung eines komplexen Verfahrens wie einer Beschleunigungsstruktur weniger geeignet als die CPU[8].
Die anderen wesentlichen Teile des Ray-Tracing-Verfahrens, die Berechnung der Shading-Informationen und der Strahl-Objekt-Schnittpunkte, schienen für eine Implementierung auf der GPU geeignet:

- Für die Durchführung der Strahlanfrage sind Gleitkommaoperationen, zum Teil für Vektoren, notwendig. Diese können effizient auf Grafikkarten ausgeführt werden.
- Für das *Shading* werden beim Ray-Tracing im Wesentlichen dieselben Verfahren verwendet wie beim *Feed Forward Rendering*. Damit kann die Leistungsfähigkeit moderner GPUs für das *Shading* beim *Feed Forward Rendering* auch für das Ray-Tracing genutzt werden.

[8] Zwar konnte Purcell in [PUR02] zeigen, dass ein reguläres Gitter auf der Grafikkarte implementiert werden kann. Es war jedoch nicht sinnvoll, diesen Ansatz für das geplante Ray-Tracing-System zu übernehmen. Zum einen implementierte Purcell ein ausschließlich GPU-basiertes System; zum anderen beschränkte er sich auf die Verwendung von Dreiecken beziehungsweise triangulierter Geometrie. Dies ermöglicht eine relativ homogene Objektgröße, da jedes Dreieck gegebenenfalls in eine Menge kleinerer Dreiecke zerlegt werden kann. Dies erleichtert die Wahl einer geeigneten Gitterzellengröße, da dann der Fall, dass ein Primitiv in mehreren benachbarten Gitterzellen liegt relativ selten auftritt und das Überspannen mehrerer Gitterzellen durch ein Dreieck vollständig ausgeschlossen werden kann. Dies ist von Bedeutung, da, falls ein Objekt in mehreren Gitterzellen liegt, häufig ein mehrfacher Schnitt-Test zwischen Objekt und Strahl notwendig ist, ein Problem was im übrigen bei allen Raumteilungsverfahren auftritt. Eine Alternative zur mehrfachen Schnittpunkt-Berechnung wurde mit dem Mailbox Verfahren von Arnalldi et. al. vorgeschlagen [ARN87]. Auf Grund der fehlenden Möglichkeiten einer Speicherverwaltung auf der GPU, sowie der beschränkten Anzahl von Ergebniswerten eines Shader-Programms ist eine effiziente Implementierung auf der Grafikkarte im Moment allerdings nur schwer vorstellbar.

Neben der Durchführung der Strahlanfrage und dem Shading wird die Grafikkarte im Ray-Tracing-System auch für den bereits erwähnten Beschleunigungsansatz für die Augenstrahlen verwendet. Dieser wird in Abschnitt 3.5.4. näher erläutert.

3.4. Auswahl einer Beschleunigungsstruktur

Für die Entwicklung des geplanten Ray-Tracing Systems sind die bisherigen, unter 2.2. bereits erwähnten, Untersuchungen der Effizienz der verschiedenen Beschleunigungsstrukturen nur bedingt zu verwenden. Diese wurden für ausschließlich CPU-basierte Ray-Tracing Systeme durchgeführt. Bei einer Verwendung von CPU- und GPU für das Ray-Tracing-Verfahren wird die Effizienz jedoch von zusätzlichen Größen beeinflusst.

Ein wichtiger Faktor ist der Datentransfer zwischen CPU und GPU, welcher auch bei einem leistungsfähigen Bus-Systems wie *PCIe* möglichst gering zu halten ist. Die Datentransferraten sind auch bei *PCIe* beschränkt und jede Datentransfer-Operation führt zu einem nicht unerheblichen Overhead. Auf der Grafikkarte entsteht dieser durch das für eine Readback-Operation notwendige *Flushen* der gesamten OpenGL Pipeline. Dabei müssen vor dem Zurücklesen alle noch nicht abgearbeiteten OpenGL Befehle ausgeführt werden, was zu einem *Pipeline Stall*[9] auf der Grafikkarte führt.

Auf Grund des, im Vergleich zu Strahlrichtungstechniken, großen Datentransfers zwischen CPU und GPU, erscheinen Raumteilungsverfahren und Hüllkörper-Hierarchien für das geplante Ray-Tracing-System nicht geeignet. Ein großer Datentransfer entsteht für diese Verfahren, da eine Strahlanfrage zum Teil für mehrere Blätter der Hierarchie durchgeführt werden muss, bevor die Strahlanfrage bezüglich der Szene beantwortet werden kann. Bei der Verwendung einer dieser Beschleunigungsstrukturen im Rahmen des geplanten Ray-Tracing-Systems wäre nach einer fehlgeschlagenen Strahlanfrage das Zurücklesen der Daten zur CPU notwendig, um das nächste Blatt der Hierarchie entlang der Strahlrichtung bestimmen zu können.
Eine Alternative zu Raumteilungsverfahren und Hüllkörper-Hierarchien ist das *Ray Classification* Verfahren [ARV87]. Diese bietet den Vorteil, dass nach der

[9] Als *Pipeline Stall* wird die Unterbrechung der kontinuierlichen Verarbeitung in einem Prozessor bezeichnet. Dies ist insbesondere auf Stream-Prozessoren mit einem erheblichen Overhead verbunden.

Abarbeitung eines Blattes der Hierarchie die Strahlanfrage eindeutig beantwortet ist, und somit kein mehrfacher Datentransfer zwischen CPU und GPU für einen Strahl notwendig ist.

3.5. Ray-Tracing auf CPU und GPU

Nach der Festlegung der zu verwendenden Hardware und der Verfahren soll an dieser Stelle ein Überblick über den Ablauf des Ray-Tracing-Systems für die Sekundärstrahlen gegeben werden.

Nach der Durchführung der Berechnungen für die Augenstrahlen wird die entstandene ersten Generation von Sekundärstrahlen in die initiale *Hypercube*-Hierarchie eingefügt. Dadurch wird der Prozess der *lazy evaluation* gestartet. Ist dieser Prozess abgeschlossen beziehungsweise existieren die ersten Blätter der Hierarchie, kann mit den Strahlanfragen für die in den Blättern der Hierarchie enthaltenen Strahlen begonnen werden. Dazu müssen zunächst die für die Schnitt-Tests notwendigen Informationen, die Beschreibung der Strahlen sowie die zugehörige Kandidatenliste, zur GPU gesendet werden. Dort werden anschließend die Strahlanfragen durchgeführt und, falls dies notwendig ist, die Sekundärstrahlen sowie das *Shading* berechnet. Der nächste Schritt ist das Zurücklesen der Information aus dem GPU- in den CPU-Speicher. Die erzeugten Strahlen werden anschließend wieder in die *Hypercube*-Hierarchie eingefügt und der oben beschriebene Prozess wiederholt. Das Bild entsteht durch die gewichtete Addition des auf der GPU berechneten *Shading*s im Hauptspeicher.

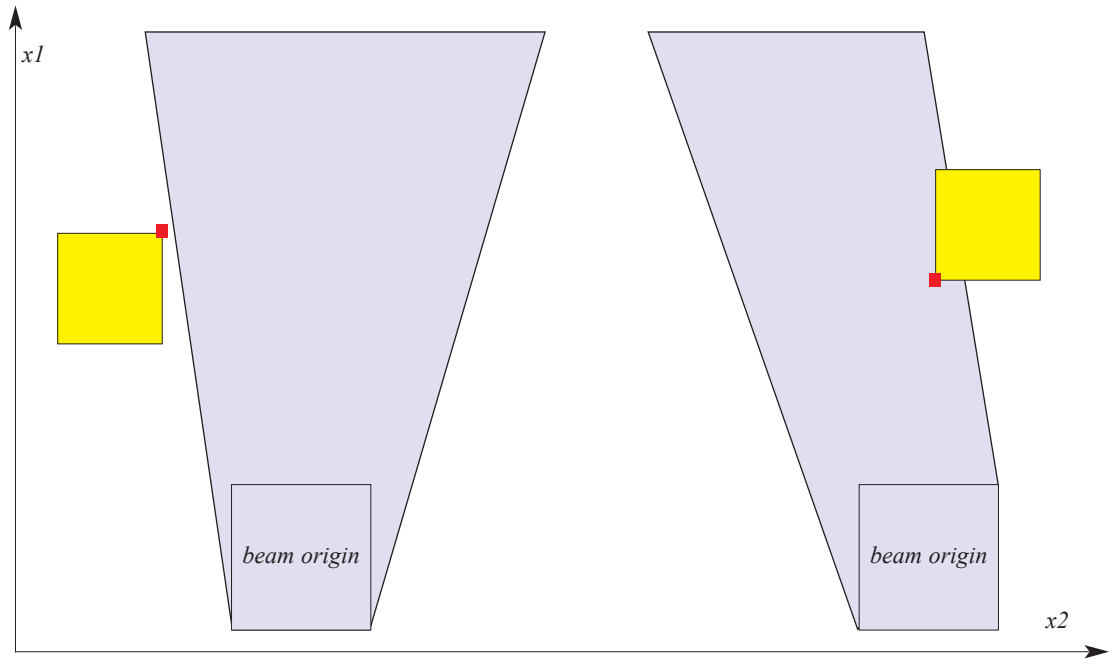


Abbildung 19: In/Out Test einer Bounding Box (gelb) bezüglich eines Beams (blau). Anhand des Anstiegs der Beam Seiten kann entschieden werden, welche Ecke der Bounding Box für den Test verwendet werden muss.

3.5.1. Strahlklassifikations-Verfahren als Beschleunigungsstruktur

Die für das Ray-Tracing-System verwendete Beschleunigungsstruktur basiert auf dem *Ray Classification* Verfahren [ARV87]. Die dazu entwickelte Variante berücksichtigt sowohl einige spätere Verbesserungen des ursprünglichen Verfahrens als auch die speziellen Anforderungen, welche bei der Verwendung von CPU und GPU entstehen.

Eine wichtige Verbesserung des ursprünglichen Verfahrens wurde von Haines vorgeschlagen [HAI94]. Bei dieser werden anstatt von Kugeln und Kegeln wie es Arvo und Kirk vorsahen, *axis-aligned Bounding Boxes* und Polyeder für die Approximation der Objekte beziehungsweise *Beams* verwendet (Abb. 19). Diese Approximationen dienen zur In/Out Klassifikation eines Objektes bezüglich eines *Beams*; es wird anhand von diesen also entschieden, ob das Objekt zur Kandidatenliste des *Beams* gehört (In) oder nicht (Out). Die Vorteile des von Haines vorgeschlagenen Verfahrens liegen zum einen in der Erzeugung einer „besseren“ Kandidatenliste, da weniger Objekte als „In“

klassifiziert werden, welche tatsächlich außerhalb des *Beams* liegen und damit für einen Strahl-Objekt-Schnitt nicht in Frage kommen, zum anderen in der schnelleren Berechnung der Klassifikation [CUO97].
Weiterhin ist für die Effizienz des Strahlklassifikations-Verfahrens die Wahl einer geeigneten Unterteilungsstrategie, sowie eines geeigneten Abbruchkriteriums wichtig. Arvo und Kirk schlagen vor, die Unterteilung der *Hypercubes* dann zu beenden, wenn entweder die Anzahl der Objekte in der Kandidatenliste oder die Größe des *Hypercube* eine untere Schranke unterschreitet. Für das in dieser Arbeit entwickelte Verfahren erschien es jedoch sinnvoll, eine komplexere Kostenfunktion einzuführen, um den unterschiedlichen Aufwand für eine Schnittberechnung bei verschiedenen Arten von Primitiven, insbesondere für parametrische Freiformflächen, berücksichtigen zu können. Darüber hinaus ist bei der Wahl der unteren Schranke für die Unterteilung auch der Berechnungsaufwand für einen Intervallzerlegungsschritt mit einzubeziehen.
Diese Unterteilungskosten entstehen im Wesentlichen durch den Aufwand, welcher für die Erzeugung einer neuen Kandidatenliste notwendig ist. Eine Reduktion dieser Kosten kann erreicht werden, indem nur für die *Hypercube*s in denen sich tatsächlich Strahlen befinden, Kandidatenlisten erzeugt werden. Dieses Vorgehen entspricht dem Prinzip der *lazy evaluation*. Eine weitere Verringerung der Kosten kann durch die Verbindung mehrerer Unterteilungsschritte erreicht werden, wobei neue Kandidatenlisten erst nach dem k-ten Unterteilungsschritt erzeugt werden. Diese Verbesserung wurde schon von Arvo und Kirk vorgeschlagen, da bereits sie feststellten, dass die Kinderknoten eines *Hypercube*s nach dessen Unterteilung in nur einer Dimension sich stark überlappen.
Die Beibehaltung des zweiten von Arvo und Kirk vorgeschlagenen Unterteilungskriteriums, der minimalen *Hypercube*-Größe, erschien durch die Verwendung der Distanzoptimierung sinnvoll. Bei *Hypercube*s unter einer bestimmten Größe ist es unwahrscheinlich, dass, unabhängig von der Anzahl der Elemente in der Kandidatenliste, viele Strahlanfragen gestellt werden müssen. Meist decken in einem solchen Fall bereits die dem Strahlursprungsraum am nächsten liegenden Objekte den gesamten Richtungsraum ab. Für die Bestimmung der minimalen *Hypercube*-Größe sind allerdings die Szene und die Größe der darin enthaltenen Objekte zu berücksichtigen.

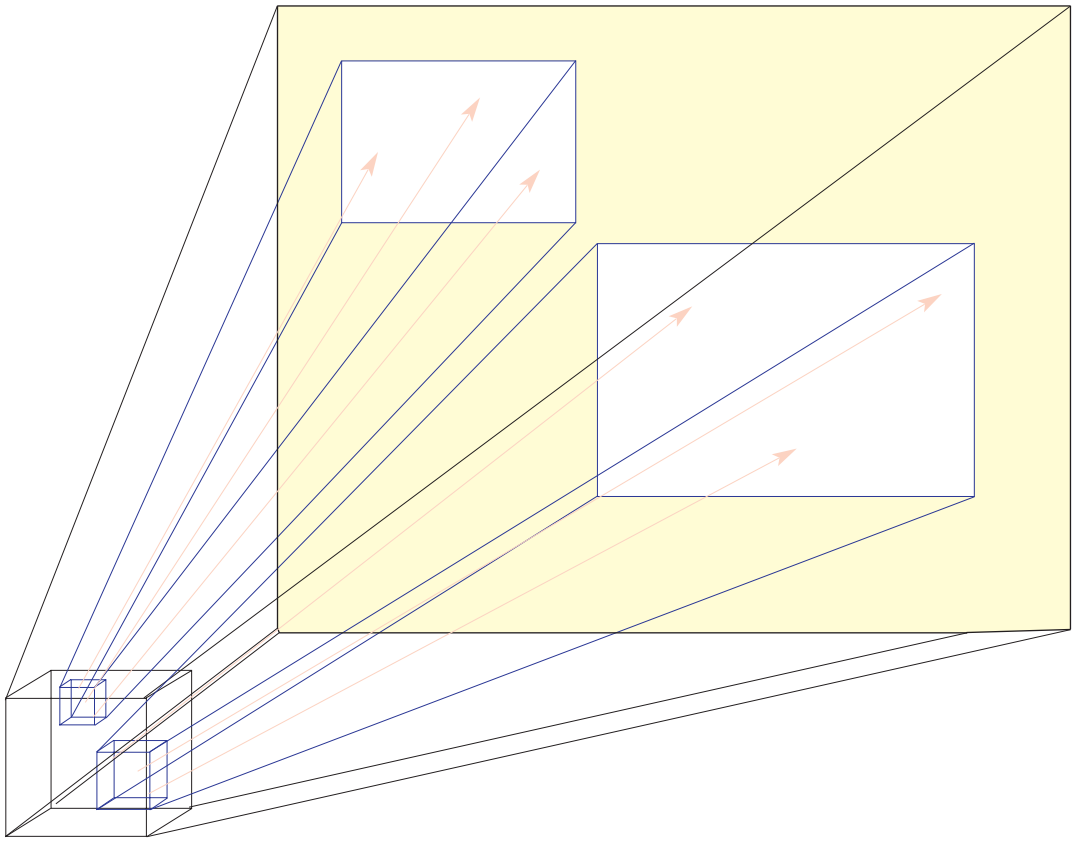


Abbildung 20: Beam (schwarz) mit Kinder-Knoten (blau) nach einem Unterteilungsschritt mit anschließendem Shrinken. Das Remainder-Set im Richtungsraum ist gelb dargestellt, im Ursprungsraum besteht es aus den Teilen des ursprünglichen Kubus, welche zu keinem der beiden Kinder-Kuben gehören.

Eine weitere wichtige Verbesserung des ursprünglichen *Ray Classification* Verfahrens ist das so genannte *Shrinking*. Dabei wird nach dem Erzeugen eines *Hypercube*s *H* mit dem Intervall *I* in R^3 x S^2 durch einen Unterteilungsschritt, das ursprüngliche Intervall *I* auf das Teilintervall *I** im 5D-Raum reduziert, in welchem tatsächlich Strahlen liegen. Durch das *Shrinking* wird erreicht, dass die nach jedem Unterteilungsschritt erzeugte Kandidatenliste eine „bessere“ Qualität hat, dass heißt weniger Objekte enthält, welche bei der Strahlanfrage zu berücksichtigen sind. Das *Shrinking* führt damit zu einer Verringerung der Unterteilungsschritte, welche notwendig sind, bis ein Blattknoten eine zur Durchführung der Strahlanfragen geeignete Größe hat.

Allerdings erlaubt das *Shrinking* zunächst nicht, dass in eine existierende *Hypercube*-Hierarchie zusätzliche Strahlen eingefügt werden. Dies wäre zum Beispiel dann sinn-

voll, wenn durch die Verarbeitung eines Teils der Blätter einer *Hypercube*-Hierarchie bereits Sekundärstrahlen entstanden sind, der Rest der Blätter aber noch auf die Durchführung der Strahlanfrage wartet. Der Vorteil bei einer solchen Vermischung der Strahlgenerationen ist die größere Anzahl von Strahlen in der *Hypercube*-Hierarchie, wodurch die Strahlkohärenz in den Blättern zunimmt und damit „bessere“ Kandidatenlisten entstehen.

Für das geplante System bedeutet dies, dass die von der GPU zurück gelesenen Sekundärstrahlen der Generation *n+1* nicht in die *Hypercube*-Hierarchie mit den Strahlen der Generation *n* eingefügt werden können. Darüber hinaus müssen zunächst alle Strahlen der Generation *n+1* auf der CPU vorliegen, bis eine *Hypercube*-Hierarchie für diese Strahlen erzeugt werden kann.

Alternativ wäre die Erzeugung mehrerer Hierarchien für die Strahlen der Generation *n+1* denkbar, allerdings würde dadurch die Kohärenz in den Blättern der *Hypercube*-Hierarchie verringert und damit die Anzahl der Strahl-Objekt Schnitt-Tests vergrößert.

Die Vermischung von Strahlgenerationen ist dabei nicht möglich, da für einen Knoten V in der Hierarchie mit einem Parameterintervall I_V in $R^3 \times S^2$ durch das *Shrinking* nicht sichergestellt ist, dass seine Kinderknoten I_V abdecken. Dies kann dazu führen, dass ein Strahl zwar bis zu einem bestimmten inneren Knoten in die Hierarchie eingefügt werden kann, jedoch keine Zuordnung zu einem Blattknoten möglich ist. Um weiterhin gewährleisten zu können, dass alle Strahlen der Szene einem Blatt der *Hypercube*-Hierarchie zugeordnet werden können, soll der Begriff des *Remainder Set* eingeführt werden.

Das *Remainder Set* ist ein spezieller Blattknoten der *Hypercube*-Hierarchie, welcher gewährleistet, dass die Vereinigung der Intervalle der Kinderknoten dem Intervall des Vaterknotens entspricht. Das *Remainder Set* ist damit die Differenzmenge zwischen dem Intervall des Vaterknotens und der Vereinigung der Intervalle der Kinderknoten. Es ist, im Gegensatz zu normalen Kinderknoten, für das *Remainder Set* nicht gewährleistet, dass es ein kontinuierlicher Teilraum in $R^3 \times S^2$ ist. Beim Einfügen neuer Strahlen in die Hierarchie wird das *Remainder Set* wie ein normales Blatt der *Hypercube*-Hierarchie behandelt und anhand einer Kostenfunktion entschieden, ob eine Unterteilung sinnvoll ist.

Das *Remainder Set* bietet somit die Möglichkeit, auch bei der Benutzung von *Shrinking*, verschiedene Generationen von Strahlen gleichzeitig in der *Hypercube*-

Hierarchie verarbeiten zu können.

3.5.2. Durchführung auf der GPU

Bei dem kombinierten CPU- und GPU-basierten Ray-Tracing-System sollen auf der GPU die Strahlanfragen und die damit verbundene Berechnungen sowie das *Shading* durchgeführt werden.

3.5.2.1. Berechnung des Shading

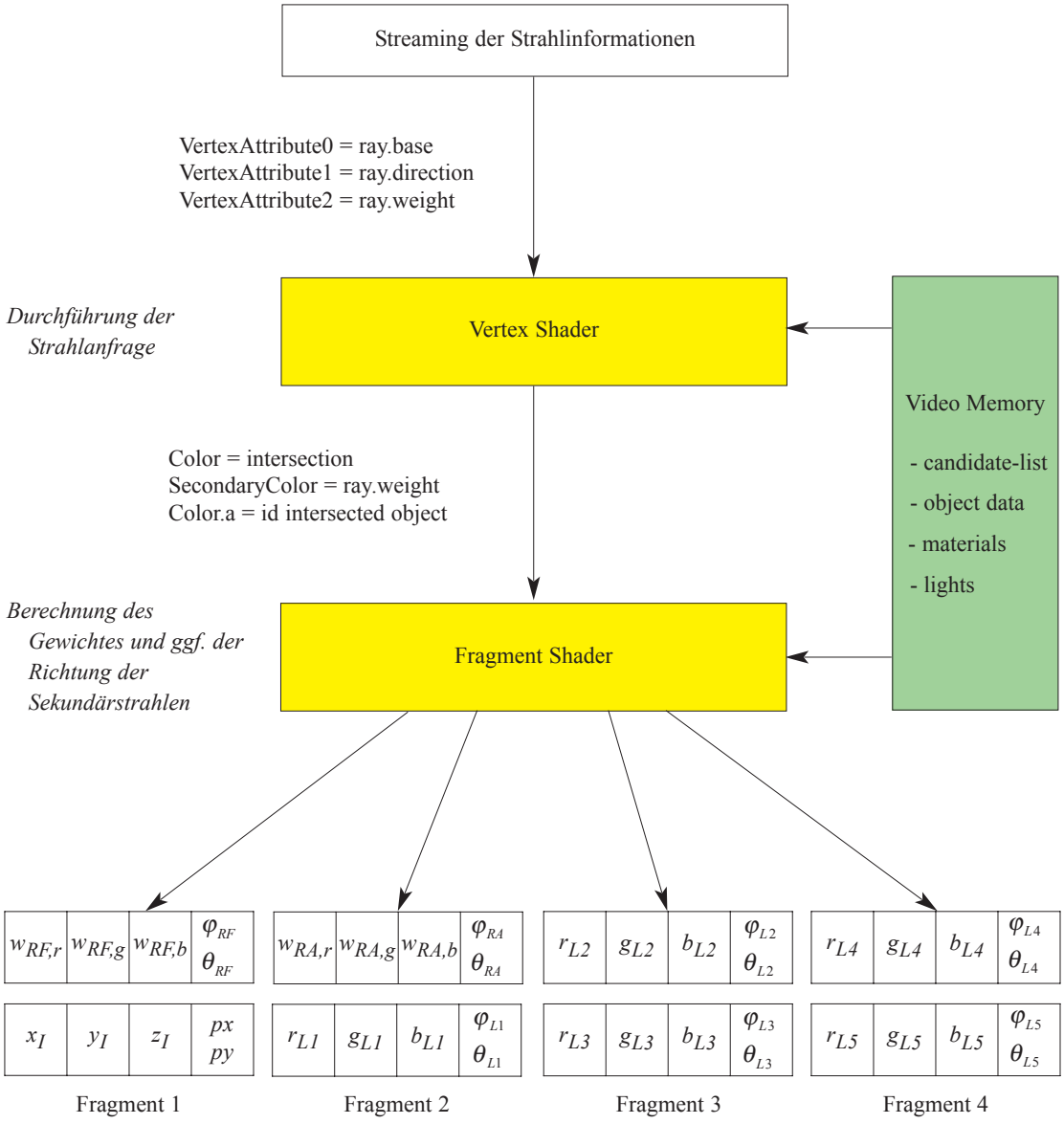
Wie bereits erläutert, kann für das *Shading* die *Shading*-Funktionalität der Grafikkarte genutzt werden. Die Berechnungen erfolgen dabei jeweils vor der Erzeugung der Sekundärstrahlen, welche den Farbbeitrag leisten würden. Dies ermöglicht die Verwendung des *adaptive tree depth control*, so dass bereits auf der Grafikkarte die Erzeugung von Sekundärstrahlen abgebrochen werden kann, welche keinen effektiven, dass heißt sichtbaren Beitrag zum Bild mehr leisten,.

Die Berechnung des *Shading's* vor der Durchführung der Strahlanfrage für die Schattenstrahlen, führt dazu, dass Ergebnis später möglicherweise verworfen werden, Dies erfolgt, wenn der Schnittpunkt bezüglich einer Lichtquelle abgeschattet ist. Das *Shading* ist, selbst bei aufwändigen *Shading* Verfahren, wesentlich weniger rechenintensiv als eine Strahlanfrage, so dass ein solches Vorgehen eine Geschwindigkeitssteigerung verspricht.

3.5.2.2. Durchführung der Strahlanfragen

Neben der Durchführung der Strahlanfrage werden auf der GPU auch die Gewichte der Sekundärstrahlen und, falls dies notwendig ist, die dazugehörigen Richtungsvektoren berechnet. Als Gewicht wird in diesem Zusammenhang der Faktor bezeichnet, mit dem der Strahl zum Gesamtfarbwert des Pixels beiträgt.

Die Berechnung der Gewichte und gegebenenfalls der Richtungsvektoren der Sekundärstrahlen auf der GPU erschien sinnvoll, da die Grafikkarte spezielle Befehle für diese Berechnungen besitzt, zum Beispiel für die Bestimmung des Richtungsvektors des Reflexions- und Refraktionsstrahls, sowie für die Koeffizienten des *Phong Shadings*. Da für die Berechnungen auf der GPU zwei Prozessoreinheiten, Vertex- und Fragment-Shader, zur Verfügung stehen, wurden zwei unterschiedliche



px/py	:	*Pixelposition zu welcher der Strahl beiträgt*
$w_{RF,r}$, $w_{RF,g}$, $w_{RF,b}$	:	*Gewicht des Reflexionsstrtahls*
φ_{RF} ,θ_{RF}	:	*Richtung des Reflexionsstrahls*
$w_{RA,r}$, $w_{RA,g}$, $w_{RA,b}$	:	*Gewicht des Refraktionsstrahls*
φ_{RA},θ_{RA}	:	*Richtung des Reflexionsstrahls*
x_I,y_I,z_I	:	*Schnittpunkt*
r_{Ln}, g_{Ln}, b_{Ln}	:	*Farbbeitrag durch die Beleuchtung der n-ten Lichtquelle*
φ_{Ln},θ_{Ln}	:	*Richtung des Strahls zur n-ten Lichtquelle*

Abbildung 21: Datenfluss bei der Durchführung der Strahlanfrage in der Vertex-Einheit

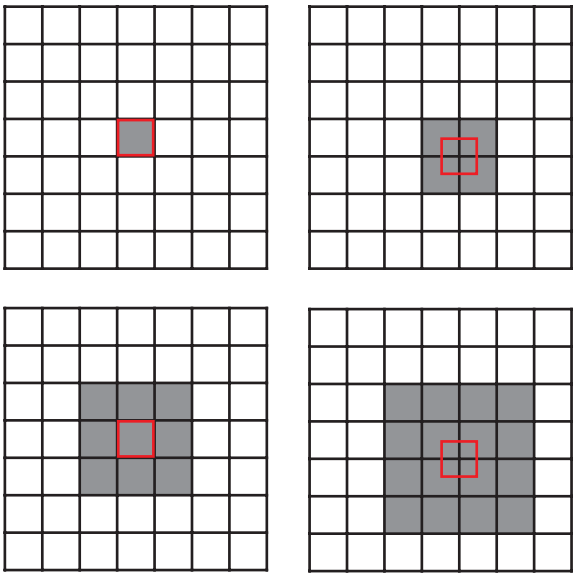


Abbildung 22:Anzahl der aus einem Punkt resultierenden Fragmente bei der Verwendung verschiedener Punktgrößen in OpenGL, links oben, k=1, rechts oben, k=2, links unten k=3, rechts unten k=4. Das rote Quadrat markiert jeweils die Position in welcher der Punkt nach der Projektion liegen muss, um diese Rasterisierung zu erzeugen. Eine ausführliche Beschreibung findet sich in [OGL04]

Ansätze entwickelt, die nachfolgend vorgestellt werden sollen.

Durchführung der Strahlanfrage im Vertex-Shader

Der Datenfluss auf der GPU bei der Verwendung der Vertex-Einheit für die Strahlanfrage ist in Abb. 21 dargestellt. Jeder Vertex repräsentiert dabei einen Strahl, für welchen die Schnitt-Tests durchzuführen sind. Die Schnitt-Berechnungen erfolgen in einer Schleife über alle Objekte der Kandidatenliste, wobei diese, entsprechend der Distanzoptimierung, gegebenenfalls nicht vollständig abgearbeitet werden muss.
Die Vertex-Position dient zur Festlegung der Position der Ergebniswerte im *Render Target*. Um die Strahlinformationen, dass heißt den Ursprung und die Richtung im Vertex-Shader zur Verfügung zu stellen, werden *Vertex Attributes* verwendet.
Die Alternative wäre das Auslesen der Strahlparameter aus einer Textur. Dieses würde aber im Vertex-Shader zu einem, im Vergleich zum Fragment-Shader, großen Overhead führen. Darüber hinaus wäre kein *Streaming* der Daten möglich.
Als *Streaming* bezeichnet man, wenn parallel zur Verarbeitung von Vertices, weitere Daten zur GPU gesendet werden. Dadurch wird in vielen Fällen, insbesondere bei rechenintensiven Vertex-Shader-Programmen, die Gesamtrechenzeit nicht, beziehungsweise nicht wesentlich von der für den Datentransfer benötigten Zeit beeinflußt.

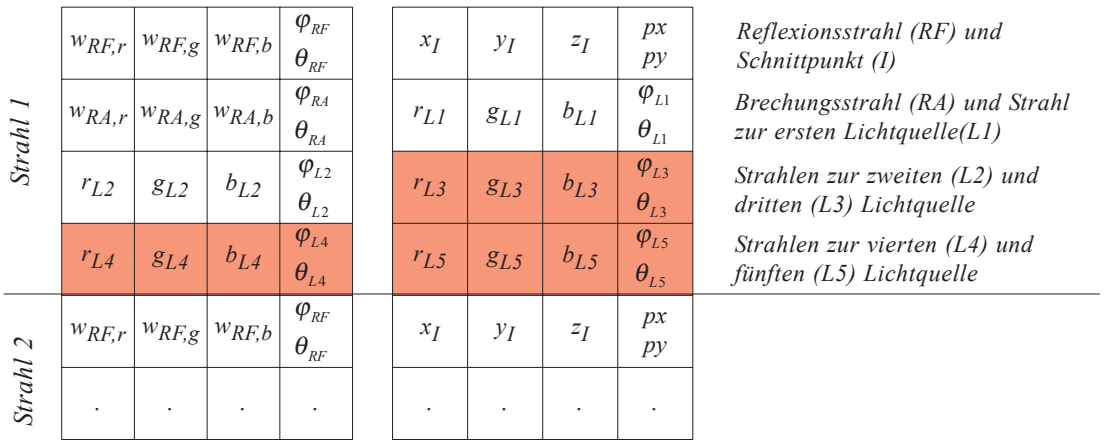


Abbildung 23: Mögliche Datenorganisation für den Readback bei maximal fünf Lichtquellen und der Verwendung von zwei Render Targets. Die rot dargestellten Felder sind die, welche bei Szenen mit zwei Lichtquellen nicht benötigt werden, auf Grund der beschränkten Flexibilität in Bezug auf die aus einem Punkt resultierenden Fragmente aber dennoch mit in den Hauptspeicher zurück gelesen werden muss (Symbole Siehe Abb. 21).

Nach den Berechnungen im Vertex-Shader wird das Ergebnis in den Fragment-Shader transportiert, wo, falls ein Schnittpunkt existiert, die Gewichte der Sekundärstrahlen und gegebenenfalls deren Richtung berechnet werden.

Ein Problem entsteht allerdings durch die Beschränkung auf maximal 16 Ergebniswerte in einem Fragment-Shader-Programm. Dadurch können für Szenen mit mehreren Lichtquellen, selbst bei optimaler Ausnutzung des vorhandenen Speicherplatzes, nicht alle Werte aus einem Fragment-Programm geschrieben werden. Eine Lösung ist die Erzeugung mehrerer Fragmente pro Strahl beziehungsweise pro Vertex. Dabei ist allerdings sowohl die Fragment-Erzeugung als auch der unveränderte Transport von Informationen aus der Vertex- in die Fragment-Einheit schwierig. Für den Datentransport stehen nur *Varying Variables* zur Verfügung, welche, bis auf zwei Ausnahmefälle, in der Rasterisierungseinheit interpoliert werden.

Einer der Ausnahmefälle, bei dem keine Interpolation der *Varying Variables* erfolgt, ist das Rendern von Punkten. Wie in Abbildung 22 dargestellt, kann dabei allerdings nur eingeschränkt die Anzahl der aus einem Vertex resultierenden Fragmente kontrolliert werden. Zwar besteht mit den Befehlen `glPointSize` beziehungsweise `gl_PointSize` sowohl auf OpenGL Client- als auch auf OpenGL Server-Seite die Möglichkeit, die Punktgröße zu verändern, jedoch entstehen dabei immer k^2 Fragmente (mit $k > 0$ und k aus [0,1,2, ...]). Dies führt dazu, dass in vielen Fällen, in Abhängigkeit von der Anzahl der Lichtquellen in der Szene, ein Overhead beim

Reflexionsstrahl				*Brechungsstrahl*				*Schnittpunkt*				*Strahl zur Lichtquelle*			
$w_{RF,r}$	$w_{RF,g}$	$w_{RF,b}$	φ_{RF} θ_{RF}	$w_{RA,r}$	$w_{RA,g}$	$w_{RA,b}$	φ_{RA} θ_{RA}	x_I	y_I	z_I	px py	r_{L1}	g_{L1}	b_{L1}	φ_{L1} θ_{L1}
$w_{RF,r}$	$w_{RF,g}$	$w_{RF,b}$	φ_{RF} θ_{RF}	$w_{RA,r}$	$w_{RA,g}$	$w_{RA,b}$	φ_{RA} θ_{RA}	x_I	y_I	z_I	px py	r_{L1}	g_{L1}	b_{L1}	φ_{L1} θ_{L1}
$w_{RF,r}$	$w_{RF,g}$	$w_{RF,b}$	φ_{RF} θ_{RF}	$w_{RA,r}$	$w_{RA,g}$	$w_{RA,b}$	φ_{RA} θ_{RA}	x_I	y_I	z_I	px py	r_{L1}	g_{L1}	b_{L1}	φ_{L1} θ_{L1}
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Abbildung 24: Mögliche Datenorganisation für den Readback bei einer Lichtquellen und der Verwendung von vier Multiple Render Targets. Jede Zeile entspricht dem Ergebnis eines Strahls und jedes Feld einer Farbkomponente im Buffer; in Feldern in welchen zwei Werte stehen, werden diese gepackt. Analog der Verwendung von vier Ergebisbuffern wäre auch die Erzeugung von vier Fragments pro Strahl und die Verwendung nur eines Render Targets denkbar. (Symbole Siehe Abb. 22)

Readback entsteht. Eine mögliche Datenorganisation für das Zurücklesen bei einer Punktgröße von k=2 zeigt Abbildung 23. Um den vorhandenen Speicherplatz möglichst effizient zu nutzen und so Szenen mit mehr Lichtquellen bei gleicher Punktgröße zu ermöglichen, werden unter anderem die Strahlrichtungen mit Hilfe der Polarkoordinaten φ und θ dargestellt. Neben der Reduktion des benötigten Speicherplatzes kann dadurch zusätzlich Rechenzeit auf der CPU gespart werden, da für das Einfügen der Strahlen in die Hypercube-Hierachie ebenfalls diese Strahlrepräsentation benötigt wird. Eine weitere Verringerung des benötigten Speicherplatzes kann durch das *Packing* von Werten erreicht werden. Es muss jedoch noch näher untersucht werden, ob der auftretende Verlust an Genauigkeit bei den Strahlrichtungsinformation akzeptabel ist, dass heißt, ob dadurch keine Artefakte entstehen.

Die zweite Möglichkeit nicht-interpolierte Daten aus der Vertex- in die Fragment-Einheit zu transportieren, ist die Benutzung von Linien. Im Gegensatz zu Punkten kann bei diesen elementgenau die Anzahl der entstehenden Fragmente kontrolliert werden. Bei Linien ist aber die Anzahl der transportierbaren Datenelemente auf neun beschränkt. Acht nicht-interpolierte Werte können durch die zwei Farbwerte `(gl_Color, gl_SecondaryColor)` in den Fragment-Shader transportiert werden, ein weiterer mit Hilfe der Z-Komponente der Position sofern Parallelprojektion verwendet wird. Die begrenzte Anzahl von transportierbaren Datenwerten führt zu einer Einschränkung der Möglichkeiten eines *Load Balancing* zwischen Vertex- und Fragment-Einheit. Auf Grund der bisherigen Aufteilung ist aber nur eine Verlagerung weiterer Berechnungen in den Vertex-Prozessor denkbar. Dies erscheint allerdings

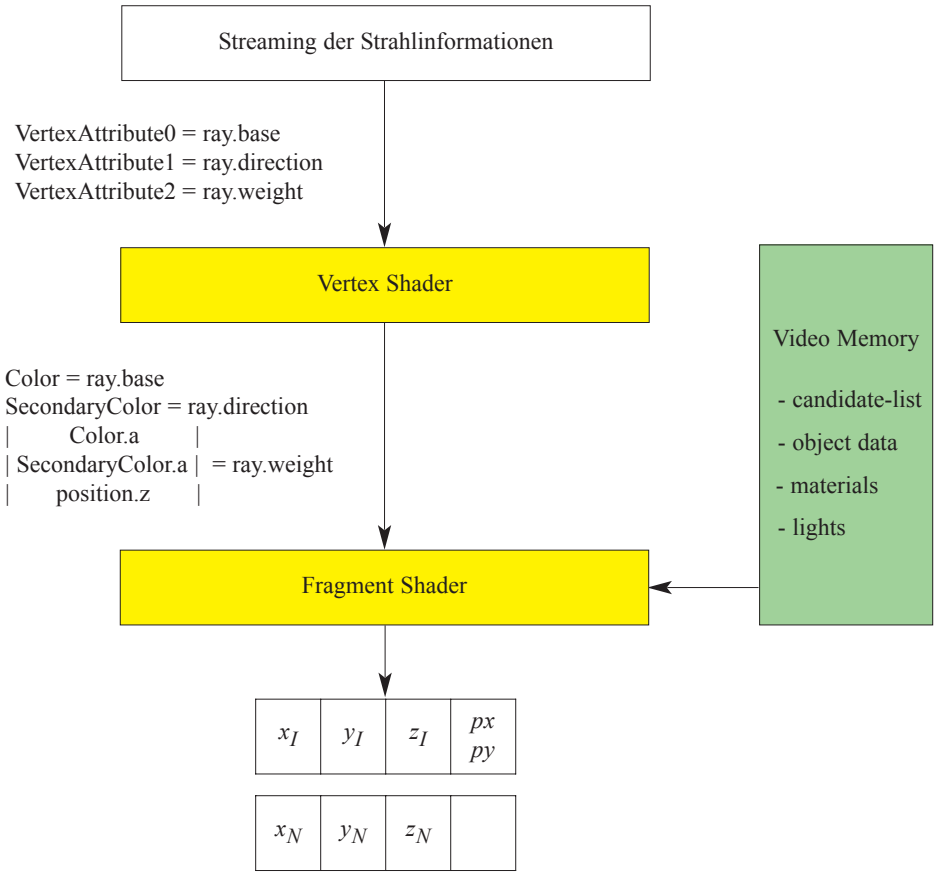


Abbildung 25: Datenfluss bei der Duchführung der Strahlanfrage in der Fragment-Einheit, 1. Rendering Pass

wenig sinnvoll, da die in der Vertex-Einheit durchgeführte Strahlanfrage die aufwändigste Einzelberechnung darstellt. Für Szenen mit nur einer Lichtquelle ist, wie Abbildung 23 zeigt, die Erzeugung eines Fragment pro Strahl ausreichend. Alle benötigten Informationen können in diesem Fall in den 16 möglichen Ergebniswerten gespeichert werden.

Durchführung der Strahlanfrage im Fragment-Shader

Die zweite Möglichkeit zur Durchführung der Strahlanfrage auf der GPU ist die Berechnung der Schnitt-Tests im Fragment-Shader. Für Szenen mit einer Lichtquelle ist dabei ein *Rendering Pass* ausreichend. In diesem Fall kann die, in Abbildung 24 bereits vorgestellte Datenorganisation für das Zurücklesen verwendet werden.
Für Szenen mit mehr als einer Lichtquelle ist bei der Verwendung des Fragement-Shaders für die Strahlanfragen ein *Two Pass Rendering* Verfahren notwendig, um die

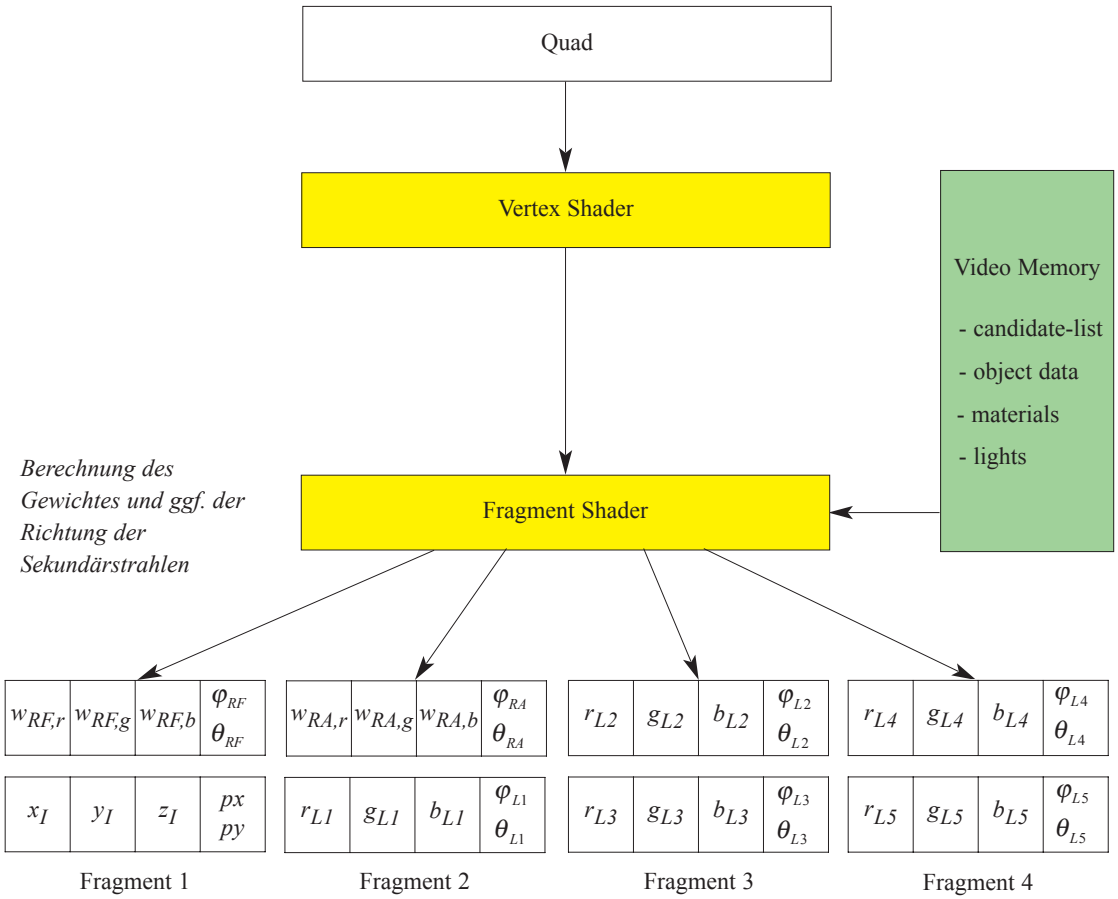


Abbildung 26: Datenfluss bei der Duchführung der Strahlanfrage in der Fragment-Einheit, 2. Rendering Pass

aufwändige Strahlanfrage nicht mehrfach durchzuführen. Der Ablauf der zwei *Rendering Passes* ist in Abbildung 25 und Abbildung 26 dargestellt.

Im ersten *Pass* werden dabei die Strahlanfragen durchgeführt und die Ergebnisse aus den *Render Targets* in Texturen kopiert. Dabei sind die 16 möglichen Ergebniswerte ausreichend, so dass auch eine Verlagerung von Berechnungen in den jeweils anderen *Rendering Pass* möglich ist. So könnten zum Beispiel die lichtquellenunabhängigen Berechnungen für die Beleuchtung im ersten *Pass* durchgeführt werden: Dadurch müssten diese nicht mehrfach im zweiten *Pass,* in den verschiedenen zu einem Strahl gehörenden Fragmenten, berechnet werden. Dabei ist allerdings der Overhead bei der Verwendung von *Multiple Render Targets* zu berücksichtigen. In Abbildung 26 erfolgen deshalb die lichtquellenunabhängigen Berechnungen für die Beleuchtung im zweiten *Rendering Pass* in jedem Fragment. Dabei werden, analog zur Durchführung der Strahlanfrage im Vertex-Shader, mehrere Fragmente pro Strahl erzeugt, um die

Limitierung bezüglich der Anzahl der Ergebniswerte zu überwinden.

3.5.3. Mögliche Optimierungen

3.5.3.1. Strahl-Caching auf der GPU

Unabhängig von der für die Strahlanfrage verwendeten Prozessor-Einheit ist eine Verringerung des Datentransfers durch ein Caching der Strahlen auf der GPU möglich. Dabei werden die Strahlinformationen nicht nur in den Hauptspeicher zurück gelesen, um sie in die Beschleunigungsstruktur einzufügen, sondern es verbleibt auch eine Kopie im Grafikkarten-Speicher. Mit einem geeigneten Adressierungsschema ist dadurch kein Rückschreiben von Strahlinformationen zur GPU notwendig; es würde das Senden einer Strahl-ID ausreichen. Es ist jedoch zu berücksichtigen, dass hierfür zusätzlicher Speicherplatz auf der Grafikkarte benötigt wird, und, sowohl auf OpenGL Client- als auch auf Server-Seite, zusätzlicher Rechenaufwand entsteht.

3.5.3.2. Getrennte Behandlung der Strahlen zu den Lichtquellen

Wie bei ausschließlich CPU-basierten Ray-Tracing-Systemen, zum Beispiel durch die von Haines vorgeschlagenen *Lightbuffer* [HAI86], kann auch bei einem CPU- und GPU-basierten Ray-Tracing-System eine Optimierung durch die getrennte Behandlung der Lichtstrahlen erreicht werden. Wie bereits erläutert, wird der mögliche Farbbeitrag durch die direkte Beleuchtung bereits mit dem Richtungsvektor des Strahls zur Lichtquelle zurück gelesen. Nach der Durchführung der Strahlanfrage für einen Strahl zu einer Lichtquelle auf der GPU ist damit das Zurücklesen der Information, ob ein Schnittpunkt gefunden wurde, ausreichend. Die Verwendung der unter 3.5.2. erläuterten Datenorganisation, bei welcher 512 und mehr Bit pro Strahl zurückgelesen werden, scheint deshalb wenig geeignet für den *Readback* bei Strahlen zu den Lichtquellen.

Eine mögliche Alternative ist die Verwendung eines zusätzlichen *Rendering Passes* für die Lichtstrahlen. Als Ergebnisbuffer kann dabei ein Ein-Byte, Ein-Kanal *PBuffer* verwendet werden, was den erforderlichen Datentransfer wesentlich verringert.

Ein Nachteil einer solchen getrennten Behandlung der Strahlen zu den Lichtquellen auf der GPU ist allerdings, dass dadurch auch innerhalb der *Hypercube*-Hierarchie eine getrennte Verwaltung erfolgen muss. Dies führt zu einer Erhöhung des

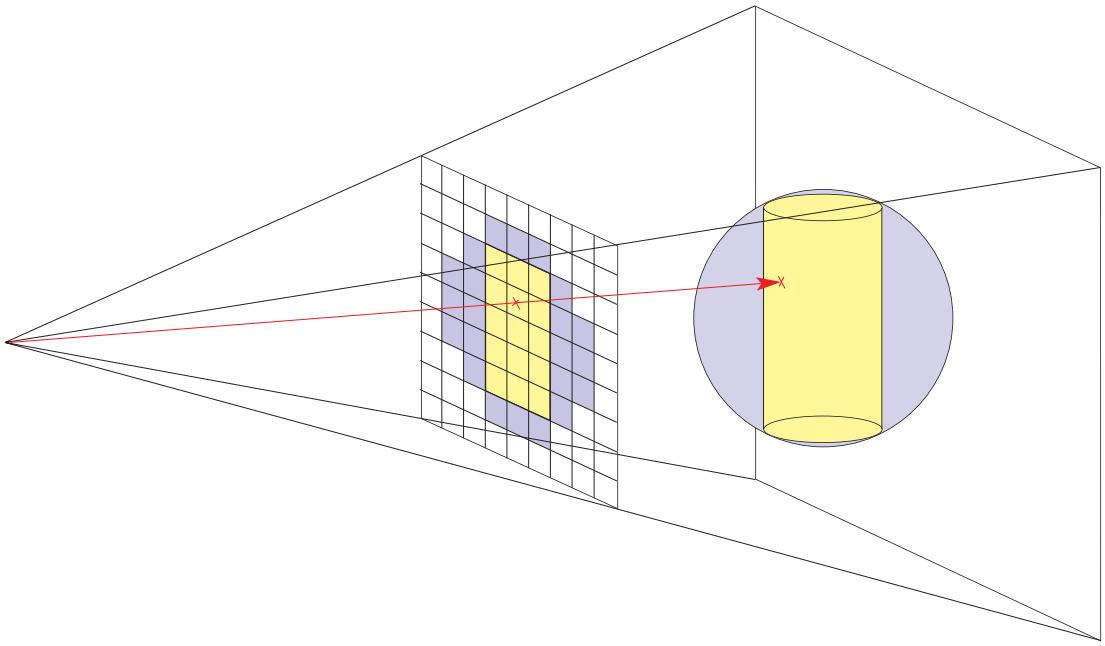


Abbildung 27: Prinzip der Hüllkörperprojektion

Speicherbedarfs und des Rechenaufwandes für die Beschleunigungsstruktur.

3.5.4. Behandlung der Augenstrahlen

Für die Behandlung der Augenstrahlen konnte im Rahmen dieser Arbeit ein effizientes Verfahren zur Durchführung der Strahlanfrage auf der GPU entwickelt werden.

3.6.1 Idee

Die Idee des für die Augenstrahlen verwendeten Verfahrens kann als Umkehrung des Prinzips der Beschleunigungsstrukturen aufgefasst werden. Anstatt der Ermittlung einer Teil-Objektszene, welche für einen Strahl-Objekt-Schnitt in Frage kommt, wird für jedes Objekt eine Teilmenge der Strahlszene ermittelt, welche sich eventuell mit dem Objekt schneidet. Zur Ermittelung der Teil-Strahlszene werden zwei Vorraussetzungen verwendet:

1. Alle Strahlen, welche ein Objekt schneiden, besitzen auch einen Schnittpunkt mit dessen Hüllkörper.

2. Die durch die Projektion eines Objektes auf der Bildebene entstandene Fläche ist äquivalent zu der Menge der Augenstrahlen, welche das Objekt schneiden. Im Fall eines Raster-Bildschirms entspricht jedes Pixel genau dem Augenstrahl welcher durch den Pixelmittelpunkt verläuft.

Die Menge der Strahlen, welche ein Objekt schneiden, kann damit durch die Projektion des Hüllköpers des Objekts ermittelt werden. Durch die Wahl geeigneter, effizient mit der Grafikkarte projizierbarer Hüllkörper ist damit eine hardwarebeschleunigte Ermittelung der Teil-Strahlszene mit Hilfe des *Feed Forward Renderings* möglich.

Die Berechnung des Schnitt-Tests zwischen Objekt und Strahl erfolgt nach der Hüllköper-Projektion im Fragment-Shader. Bei mehr als einer Lichtquelle in der Szene ist nach der Durchführung der Strahlanfrage, entsprechend der unter 3.5 erläuterten Vorgehensweise, ein zweiter *Rendering Pass* erforderlich, um die für die Sekundärstrahlen notwendigen Berechnungen durchzuführen und gegebenenfalls die

[10] Als *durchschnittliche Güte* wird hier die Güte für alle möglichen Blickrichtungen und Augenpunkte bezeichnet.

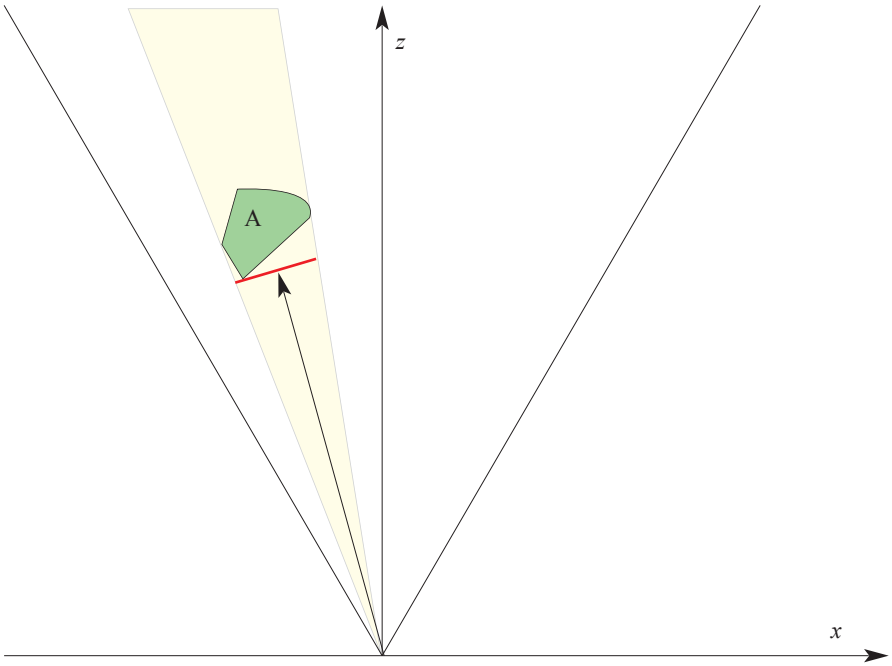


Abbildung 28: Approximation der Projektion eines Objektes mit Hilfe eines Billboards (rot)

Ergebniswerte zu speichern.

3.6.2. Hüllkörper

Für die Qualität eines Hüllkörpers sind zwei Faktoren entscheidend:

1. Die Güte der durch die Projektion entstandenen Teil-Strahlszene, dass heißt, für wie viele durch die Hüllkörperprojektion erzeugte Fragmente beziehungsweise Augenstrahlen tatsächlich ein Schnittpunkt mit dem Objekt existiert. Die Güte bestimmt wesentlich den Berechnungsaufwand für die Schnitt-Tests nach der Hüllkörperprojektion. Die durchschnittliche Güte[10] hängt von der Qualität der Approximation des Objektes durch den Hüllkörper ab.

2. Der für die Projektion des Hüllkörpers notwendige Rechenaufwand.

Bei der Wahl eines geeigneten Hüllkörper muss damit zwischen der Güte der Teil-Strahlszene und dem Projektionsaufwand abgewogen werden.

Eine Alternative zur Verwendung von Hüllkörpern bieten *Billboard*s. Diese erlauben die Approximation der Projektion eines Objektes mit Hilfe eines Rechtecks (Abb.29).

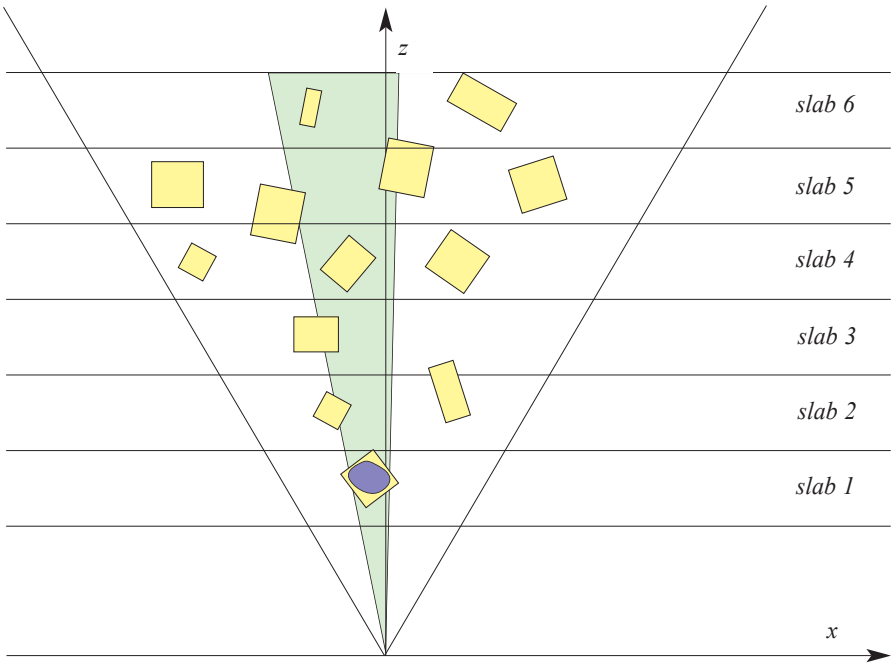


Abbildung 29: Unterteilung der Szene in slabs. Erfolgt das Rendering in Front-To-Back Order, so sind nach dem Schnitt mit dem Objekt in slab 1 keine weiteren Schnitt-Test für Strahlen im grünen Strahlparameterbereich notwendig.

Für die Verwendung im Ray-Tracing-System muss bei der Approximation allerdings sichergestellt werden, dass die erzeugt Teil-Strahlszene alle Strahlen enthält, welche sich tatsächlich mit dem Objekt schneiden.
"*Billboard*" wird im Rahmen dieser Arbeit in Analogie zu der gebräuchlichen Definition des Begriffs verwendet. Es können jedoch, in Abhängigkeit von der Implementierung, Unterschiede existieren, da das *Billboard* im Ray-Tracing-System nur die Erzeugung der Teil-Strahlszene gewährleisten muss.
Da Hüllkörper und *Billboards* die gleiche Funktion erfüllen, bezieht sich der Begriff Hüllkörperprojektion im Folgenden auch auf *Billboards*.

3.6.3. Distanzoptimierung

Ein Problem des Hüllkörperprojektions-Verfahrens besteht darin, dass zur Zeit keine Distanzoptimierung eingesetzt werden kann. Für alle Fragmente, welche auf Grund der Hüllkörperprojektion erzeugt werden, beziehungsweise den entsprechenden Augenstrahlen, muss der Objekt-Strahl Schnitt-Tests durchgeführt werden. Die

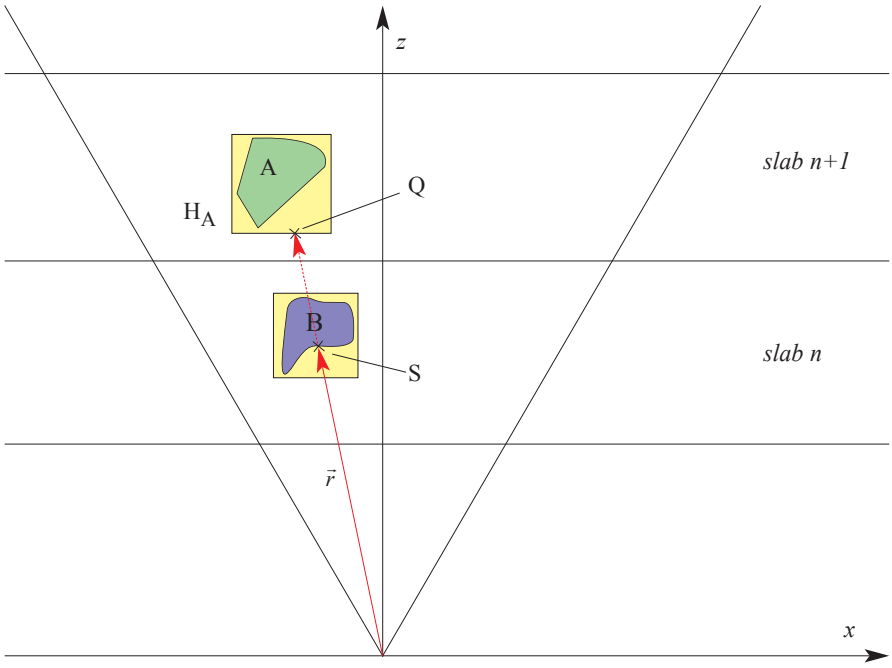


Abbildung 30: Distanzoptimierung bei der Hüllkörperprojektion. Durch das Rendern der slabs in Front-to-Back Order kann im Fragment, welches für den Schnitt-Test von $\vec{r}$ mit A erzeugt wurde, der Schnittpunkt S aus der Depth Buffer Texture ausgelesen werden. Dies verhindert, dass der Schnitt-Tests von $\vec{r}$ mit A durchgeführt werden muss, da dieDistanz des Hüllkörpers von A zum Augenpunkt größer ist, als die Entfernung zum Schnittpunktes von $\vec{r}$ mit B zum Augenpunkt.

Ursache dafür ist die im Moment fehlende Möglichkeit, in der Fragment-Einheit Ergebnisse von im selben *Rendering Pass* bereits verarbeiteten Fragmenten abzufragen.

Eine Möglichkeit um, zumindest eingeschränkt, die Distanzoptimierung nutzen zu können, ist die Verwendung von so genannten *slabs*. Dabei wird die Objektszene, wie in Abbildung 30 dargestellt, in Intervalle entlang der Blickrichtung unterteilt. Das Rendering erfolgt in Front-To-Back Order, dass heißt, jeweils die dem Augenpunkt am nächsten liegende und noch nicht verarbeitete *slab* wird als nächstes gerendert.

Nach dem Rendern jeder *slab* wird der aktuelle *Depth Buffer* in eine Textur kopiert. Durch die in der so genannten *Depth Buffer Texture* gespeicherten Werte kann dann, wie im Folgenden noch näher erläutert wird, bereits anhand der Hüllkörper entschieden werden, ob die Durchführung des Objekt-Strahl Schnitt-Tests notwendig ist.
Es sei, wie in Abbildung 30 dargestellt, A ein beliebiges Objekt der Szene mit dem Hüllkörper H_A in der *slab N+1*; S der Schnittpunkt des Augenstrahles r mit dem

Objekt *B* in *slab N*, und werde bei der Projektion von H_A ebenfalls ein Fragment erzeugt, welches *r* repräsentiert. Ist in einem solchen Fall die minimale Entfernung des Hüllkörpers H_A zum Augenpunkt größer, als die Distanz von *S* zu diesem, so ist sichergestellt, dass jeder mögliche Schnittpunkt von *A* mit *r* eine größere Distanz zum Augenpunkt hat als *S* und damit kein Schnitt-Test notwendig ist. Dies ist gewährleistet, da kein Punkt von *A*, und damit auch kein Schnittpunkt von *r* mit *A*, eine geringere Distanz zum Augenpunkt haben kann, als der dem Augenpunkt am nächsten liegende Punkt des Hüllkörpers. Darüber hinaus kann, falls der erste, bereits erläuterte Test fehlschlägt, der Schnittpunkt *Q* von *r* mit H_A bestimmt werden. Ist die Distanz von *Q* zum Augenpunkt größer als die von *S*, ist ebenfalls kein Schnitt-Test von *r* mit *A* notwendig. Der zweite, zusätzliche Test sollte insbesondere für solche Primitive erfolgen, bei denen der Schnitt-Test mit dem Objekt wesentlich aufwändiger ist, als der mit dem Hüllkörper.

3.6. Zusammenfassung

Es wurde in diesem Kapitel die mögliche Funktionsweise eines sowohl CPU- als auch GPU-basierten Ray-Tracing-Systems vorgestellt. Die Aufteilung des Ray-Tracing-Verfahrens auf beide Prozessor-Einheiten erfolgte dabei mit dem Ziel einer bestmöglichen Geschwindigkeitssteigerung und eines möglichst geringen Datentransfers zwischen CPU und GPU.

Als Beschleunigungsstruktur wurde ein modifiziertes *Ray Classification* Verfahren vorgeschlagen. Dabei sollen zur Verbesserung der Effizienz gegenüber dem ursprünglichen Verfahren *Bounding Boxes* und Polyeder, sowie *Shrinking* verwendet werden. Es wurde dabei das Konzept des *Remainder Set* vorgestellt, welches bei der Verwendung von *Shrinking* das Hinzufügen von Strahlen zu einer existierenden Hypercube-Hierarchie erlaubt.

Für die Durchführung der Berechnungen auf der Grafikkarte wurden zwei Möglichkeiten aufgezeigt. Die Berechnung der Strahlanfrage in der Fragment-Einheit bietet den Vorteil, dass dort die größere Rechenleistung zur Verfügung steht. Diese kann aber, auf Grund der SIMD-Architektur in der Fragment-Einheit, nur bei kohärentem Programmfluss effektiv genutzt werden. Diese Limitierung besteht in der Vertex-Einheit nicht, allerdings steht dort weniger Rechenleistung zur Verfügung. Die Kohärenz des Programmflusses, und damit welche Prozessor-Einheit die größte Leistung bietet, ist jedoch nicht statisch; so hängt sie zum Beispiel von der gerenderten Szene ab. Für ein allgemein gutes System erscheint deshalb eine Kombination beider Ansätze als sinnvoll. Ein solches adaptives System würde dann anhand einer noch zu entwickelnden Heuristik entscheiden, ob für einen Blattknoten der Hypercube-Hierarchie die Strahlanfrage am effizientesten in der Vertex- oder in der Fragment-Einheit der Grafikkarte durchgeführt werden kann.

Für die Augenstrahlen des Ray-Tracing-Verfahrens wurde in diesem Kapitel ein spezieller Beschleunigungsansatz vorgeschlagen, welcher insbesondere durch die Verwendung eines Teils der OpenGL Pipeline auf der Grafikkarte eine effiziente Reduzierung der durchzuführenden Schnitt-Tests verspricht.

Kapitel 4
Implementierung

4.1. Implementierung des Ray-Tracing-Systems

Die zwei wesentlichen, bereits erläuterten, Vorraussetzungen für die Realisierung des vorgeschlagenen interaktiven Ray-Tracing-Systems waren die im *PCIe* Standard angekündigten Datentransferraten, sowie die neuen Möglichkeiten der NV4X-Architektur. Der erste Rechner mit *PCIe* Bus-System und einer NV4X-Grafikkarte stand ab Sommer 2004 zur Verfügung. Es zeigte sich jedoch bereits bei den ersten Tests, dass das *PCIe*-System noch keine höheren Datentransferraten als *AGP8x* erreicht (Abb.32). Die damit zur Verfügung stehende Bandbreite war aber für das vorgeschlagene interaktive Ray-Tracing-System nicht ausreichend.
Um dennoch ein interaktives System realisieren zu können, wurde nur ersten Teil des Ray-Tracing-Verfahrens, das Ray-Casting implementiert, für welches kein Zurücklesen der Daten notwendig ist. Dabei wurde der für die Augenstrahlen entwickelte Beschleunigungsansatz implementiert.
Obwohl das Ray-Casting-Verfahren keine direkte Berechnung von globalen Beleuchtungseffekten ermöglicht, bietet es weiterhin wesentliche Vorteile gegenüber dem *Feed Forward Rendering*:

1. Es ist die direkte Darstellung parametrisch beschriebener Geometrie möglich.

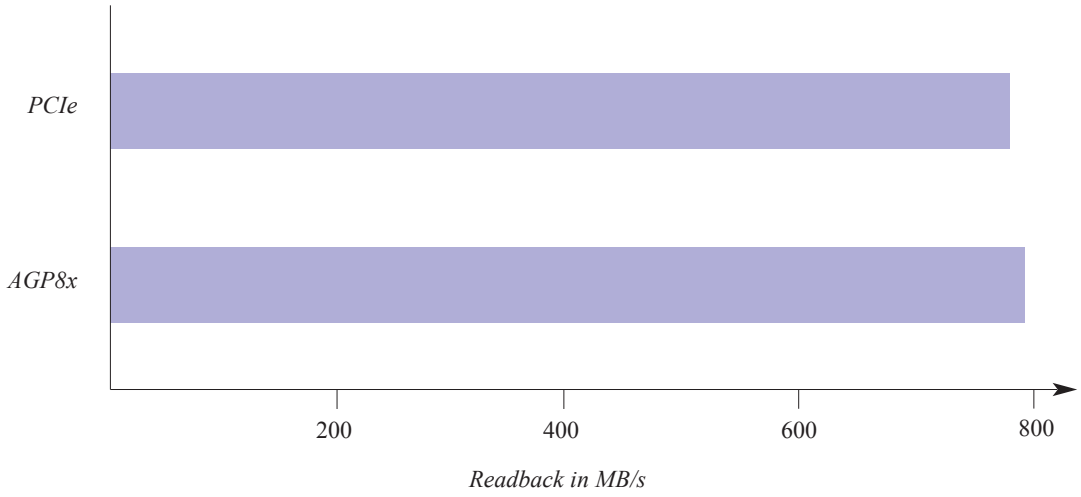


Abbildung 31: Datentransferraten beim Readback mit PCIe und AGP8x. Gemessen mit GPUBench [BUC04]

2. Die Beleuchtungsberechnung erfolgt pro Pixel.

3. Das *Shading* erfolgt nur für sichtbare Fragmente.

Darüber hinaus ist auch bei einem Ray-Casting-System durch den Einsatz von *slab*s eine eingeschränkte Verwendung der Distanzoptimierung möglich.

4.2. Implementierung des Ray-Casting-Systems

Die Implementierung des Ray-Casting-Verfahrens, welche sowohl unter Linux als auch unter Windows erfolgte, basiert auf dem unter 3.6. vorgestellten Beschleunigungsansatz für die Augenstrahlen.

Im Folgenden soll zunächst die Umsetzung des Beschleunigungsansatzes vorgestellte werden, bevor der Programmablauf des Ray-Casting-Systems für Primitive zweiter Ordnung und Dreiecke erläutert wird. Anschließend wird näher auf das Ray-Casting-System für NURBS-Oberflächen eingegangen.

4.2.1. Hüllkörper

Im Rahmen der Arbeit wurden als Hüllkörper *Bounding Spheres* (Kugeln), *Bounding Boxes* (Quader) sowie *Billboards* implementiert.

$$\cos(\alpha) = \frac{r}{|\vec{m}|}$$

$$s = \cos(\alpha) * r$$
$$t = \sin(\alpha) * r$$

$$\vec{m}' = \vec{m} - s * \left(\frac{\vec{m}}{|\vec{m}|} \right)$$

$$\vec{w} = \vec{m}' + t * (\vec{m} \times \vec{u})$$

M : Mittelpunkt der Bounding Sphere
$\vec{m}$: Ortsvektor zu M
r : Radius der Bounding Sphere
$\vec{m}'$: Ortsvektor zu M'
$\vec{w}$: Ortsvektor zu W
u : Up - Vektor der Kamera

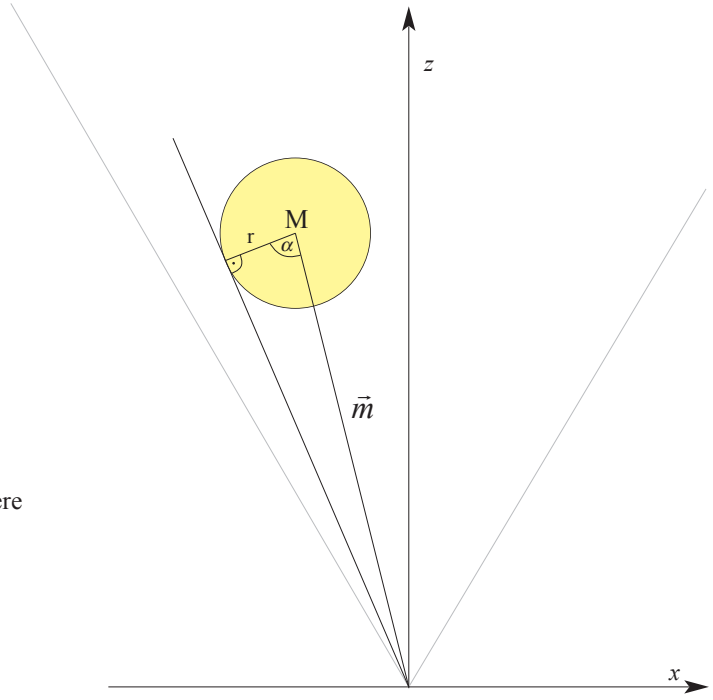


Abbildung 32: Berechnung des Größe der Projektion der Bounding Sphere im Augenkoordinatensystem (Siehe auch Abbildung 34)

4.2.1.1. Bounding Spheres

Bounding Spheres sind für das Hüllkörperprojektions-Verfahren auf Grund ihrer einfachen und schnellen Projektion geeignet. Für die Erzeugung dieser ist lediglich das Rendern eines Punktes mit variabler Größe notwendig. Dies ist insbesondere dann effizient, wenn die Größe der Projektion der *Bounding Sphere*, und damit die des Punktes, in der Vertex-Einheit berechnet wird. In diesem Fall ist zum Beispiel kein Datentransfer zwischen CPU und GPU notwendig. Für die Bestimmung der Punktgröße werden neben dem Augenpunkt der Mittelpunkt und der Radius der *Bounding Sphere*, sowie die automatisch zur Verfügung stehende *Model-View-Projection Matrix* benötigt.

Die Berechnungen werden, wie in Abbildung 32 dargestellt, im Augenkoordinatensystem durchgeführt, wodurch eine Vereinfachung erreicht wird. Im Ergebnis erhält man die Position der Punkte *M'* und *W* im Augenkoordinatensystem. *W* ist dabei der Punkt der *Bounding Sphere*, dessen Projektion die größte Distanz zu *M'* hat. Um dies sicherzustellen ist, wie in Abbildung 35 veranschaulicht, die Lage der *Bounding Sphere* bezüglich der Z-Achse zu berücksichtigen. Diese kann mit Hilfe der

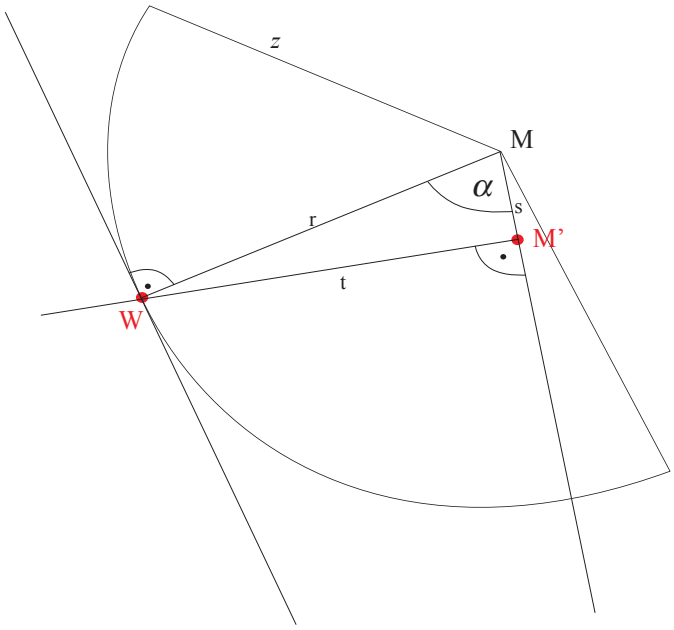


Abbildung 33: Berechnung des Projektion der Bounding Sphere im Augenkoordinatensystem. (Detailansicht)

Projektion des Augen-Koordinatensystems in die XY-Ebene bestimmt werden. Die Position von *W* bezüglich *M'* mit *M'=(x,y)* wird dabei durch die Richtung *k* mit $k:\max(|x|,|y|)$ und $k \in \{+X, -X, +Y, -Y\}$ definiert. Die Distanz zwischen *M* und *W* in Pixelkoordinaten ist die Größe des gerenderten Punktes.

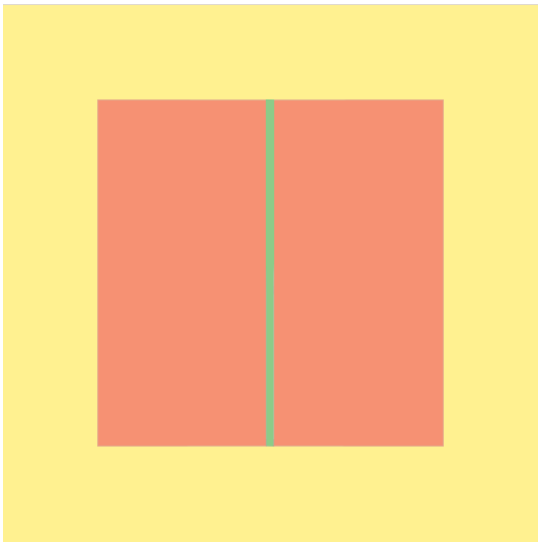


Abbildung 34: Unzureichende Approximation durch die Bounding Sphere. Der Zylinder (grün) wird schlecht durch die Projektion der Bounding Sphere (rot) approximiert; die entstehende Strahlszene hat eine geringe Güte.

Diese relativ aufwändigen Berechnungen sind sinnvoll, da dadurch die für eine *Bounding Sphere* entstehende Strahlszene minimiert werden kann. Eine optimale Strahlszene wird bei der Verwendung von Punkten allerdings dadurch verhindert, dass diese in OpenGL eine quadratische und keine runde Projektion haben.

Der wesentliche Nachteil von *Bounding Spheres* ist die für allgemeine Objekte schlechte Approximation der Form; zum Beispiel für Objekte mit einem asymmetrischen Seitenverhältnis, wie in Abbildung 34 dargestellt.

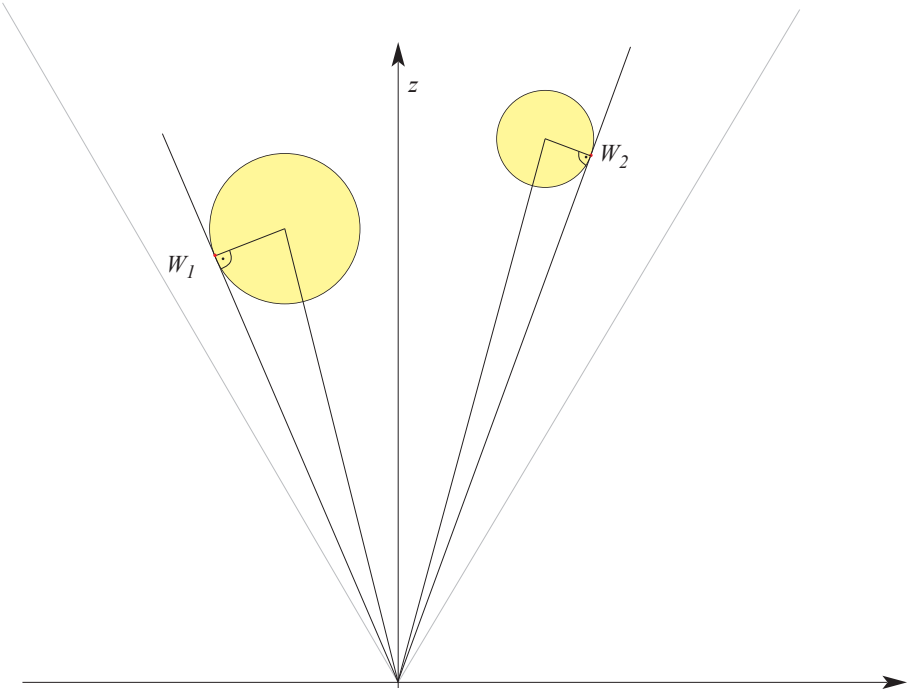


Abbildung 35: Abhängigkeit der Position des Punkts W von der Lage der Bounding Sphere bezüglich der Z-Achse

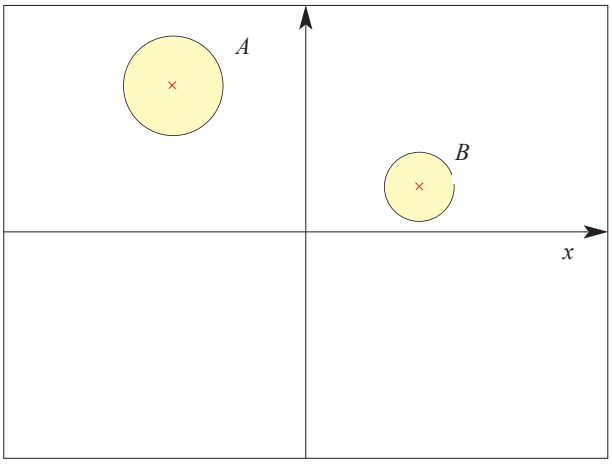


Abbildung 36: Projektion des Augen-Koordinatensystems in die XY-Ebene
Die Richtung k der Bounding Sphere A ist +Y; k von B ist +X

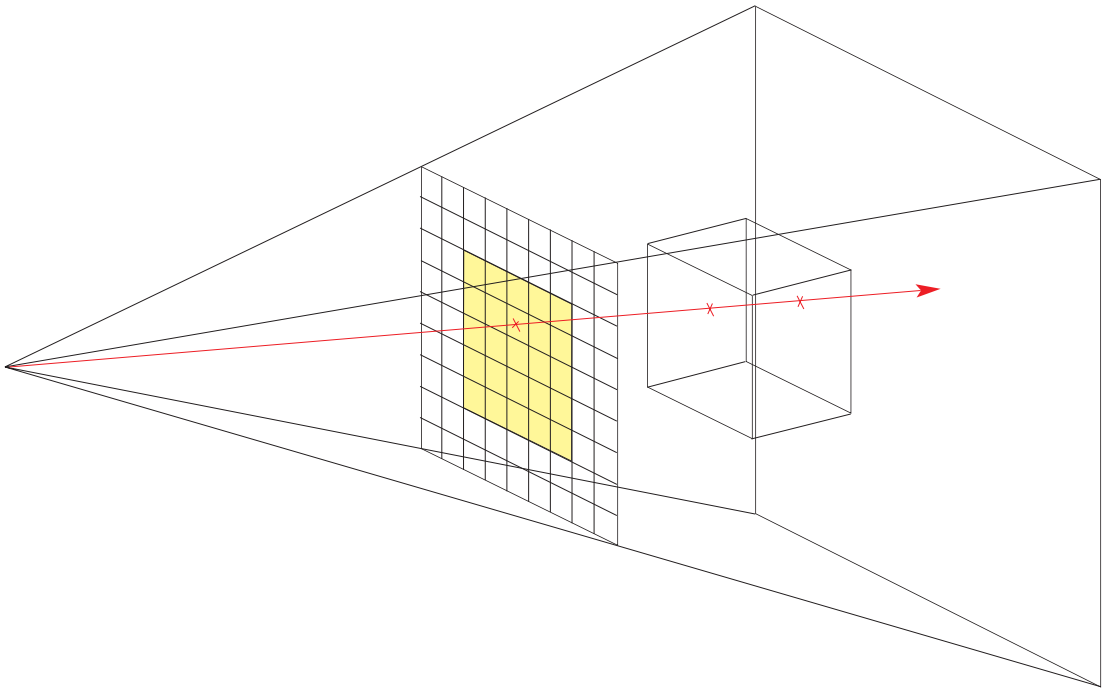


Abbildung 37: Backface Culling bei Bounding Boxes. Erfolgt kein Backface Culling so werden Vor- und Rückseite der Bounding Box (vom Augenpunkt aus gesehen) rasterisiert. Dadurch erfolgt eine doppelte Schnittberechnung mit dem von der Bounding Sphere eingeschlossenen Objekt.

4.2.1.2. Bounding Boxes

Der zweite implementierte Hüllkörper-Typ sind *Bounding Boxes*. Diese ermöglichen für beliebige Objekte eine gute Approximation der Form, bringen jedoch den Nachteil mit sich, dass pro *Bounding Box* das Rendern von 18 Vertices notwendig ist. Ein spezielles Vertex-Shader-Programms ist beim Rendern von *Bounding Boxes* nicht notwendig.

Wichtig für die Effizienz von *Bounding Boxes* ist die Verwendung von *Backface Culling*. Anderenfalls werden, wie in Abbildung 37 veranschaulicht, jeweils zwei Fragmente pro Strahl erzeugt, was zu einer Verdoppelung der durchzuführenden Schnitt-Tests führt, ohne dass das Ergebnis dadurch verändert wird.

4.2.1.3. Billboards

Neben *Bounding Spheres* und *Bounding Boxes* wurden *Billboards* zur Erzeugung der Teil-Strahlszene, welche für einen Objekt-Strahl-Schnitt in Frage kommen, implementiert.

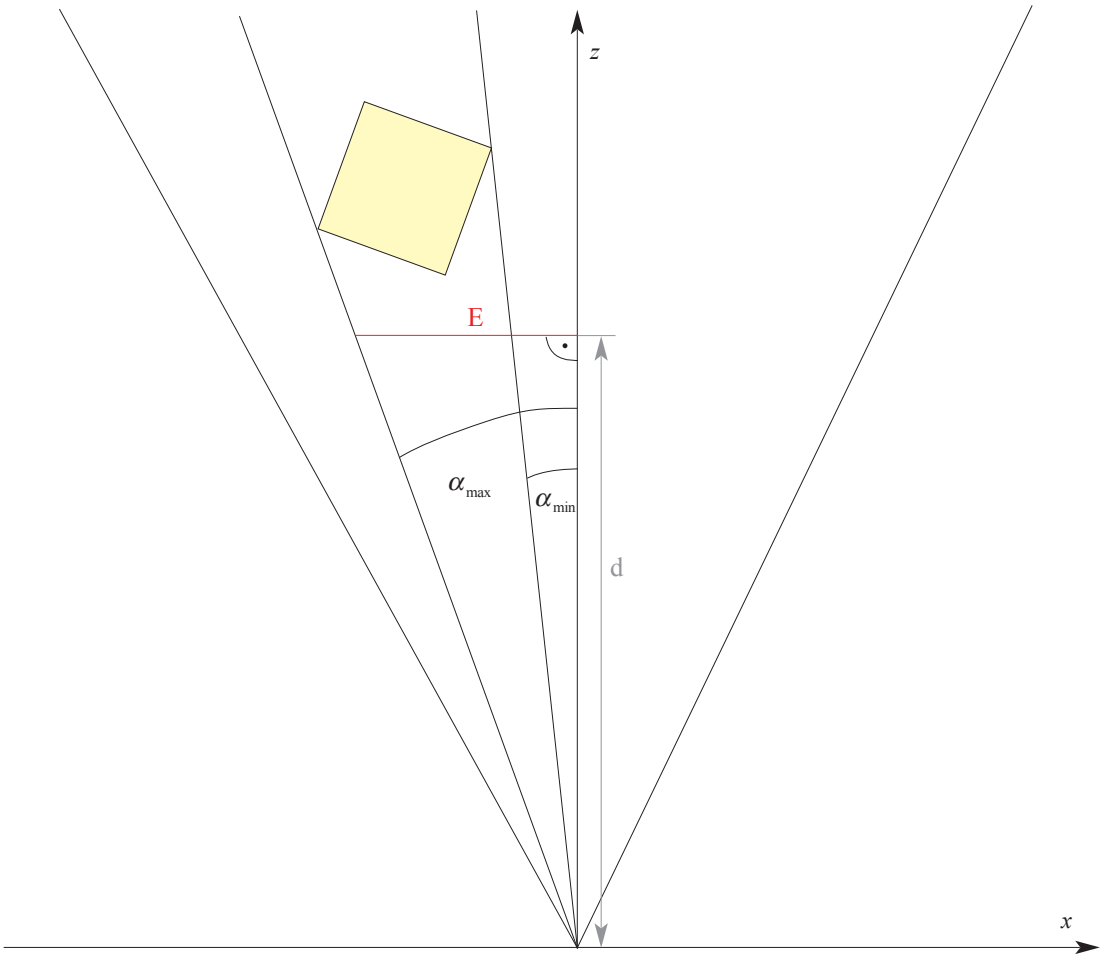


Abbildung 38: Berechnung der Größe des Billboards

Die blickpunkt- beziehungsweise blickrichtungsabhängige Orientierung des *Billboards* wird im Vertex-Shader anhand der *Bounding Box* des Objektes berechnet. Um den benötigten Speicherplatz und damit auch den notwendigen Datentransfer gering zu halten, wurde eine Repräsentation der *Bounding Box* durch drei Punkte und die Höhe gewählt, so dass alle für die Erzeugung des *Billboards* notwendigen Informationen in drei vierdimensionalen Vektoren zur Verfügung gestellt werden können.

Die Berechnungen zur Erzeugung des *Billboards* erfolgen im Augenkoordinatensystem (Abb. 38). Zunächst wird dabei für jeden Eckpunkt der *Bounding Box* der Winkel α zwischen Z-Achse und der Projektion des Punktes auf die XZ-Ebene, sowie der Winkel β zwischen Z-Achse und der Projektion des Punktes auf die YZ-Ebene berechnet. Anschließend werden die Minima und Maxima

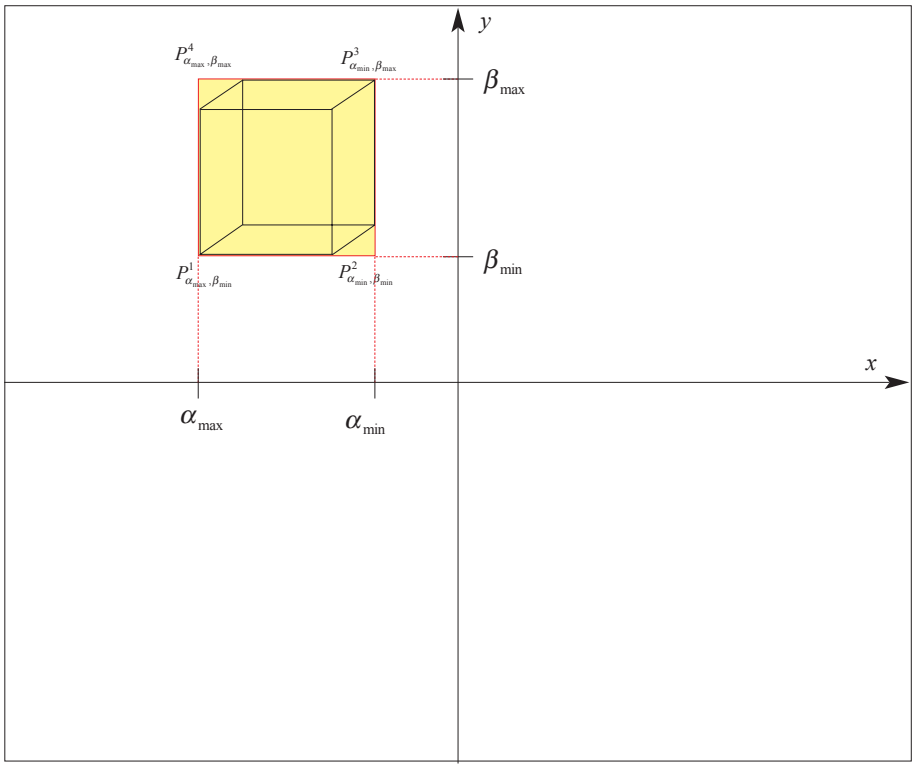


Abbildung 39: Bestimmung der Eckpunkte des Billboards mit den berechneten Winkeln und der Ebene E

dieser Winkel, α_{min} und α_{max} beziehungsweise β_{min} und β_{max}, ermittelt. Die Kombination dieser definiert, wie in (Abb. 39) dargestellt, die Eckpunkte des *Billboard*s. Zur Berechnung der Punkte aus den Winkeln wird eine Hilfsebene *E* im Abstand *d* zum Ursprung eingefügt. Der Wert von *d* kann dabei beliebig gewählt werden, solange gewährleistet ist, das *E* zwischen Near- und Far-Plane liegt Die Ebene *E* ist parallel zu XY-Ebene, so dass mit Hilfe des Kosinus die Punkte des *Billboards* bestimmt werden können.
Die Berechnungen für die vier Punkte unterscheiden sich lediglich in den verwendeten Winkeln. Auf Grund der Unabhängigkeit der Pipelines auf der Grafikkarte müssen diese dennoch viermal, für jeden Eckpunkt einzeln, durchgeführt werden.

Wie hier deutlich wird, kann das für die *Bounding Spheres* verwendete Verfahren auch als Erzeugung des, dabei stets quadratischen, *Billboards* der *Bounding Spheres* aufgefasst werden.

4.2.2. Raycaster für Primitive zweiter Ordnung und Dreiecke

In diesem Abschnitt wird das Ray-Casting-System für Primitive zweiter Ordnung und Dreiecke vorgestellt. Zur Darstellung von Flächen wurde das Rendern von Dreiecken in das System integriert. Dadurch kann auch triangulierte Geometrie verwendet werden, was die Flexibilität des Ray-Casting-Systems erhöht.
Zunächst wird der Programmablauf erläutert, anschließend werden ausgewählte Ergebnisse präsentiert und diese dann diskutiert.

4.2.2.1. Programmablauf

Der Programmablauf kann in zwei wesentliche Phasen unterteilt werden:

1. Initialisierungs-Phase
2. Rendering-Phase

1. Initialisierungsphase

Die Szene wird auf der OpenGL Client-Seite mit Hilfe eines Szenegraphen repräsentiert. Dies erlaubt ein einfaches Erstellen von Test-Szenen und ermöglicht die Evaluierung des Erweiterungspotentials in Bezug auf eine spätere Integration in ein Szenegraph-basiertes Rendering-System.
Jeweils zu Programmstart erfolgt die einmalige Traversierung des Szenegraphen, bei welcher die für das Rendering auf der GPU notwendigen Daten erzeugt werden. Für jeden besuchten Knoten, der ein Objekt repräsentiert, werden dabei zunächst die Daten gespeichert, welche für die Schnittberechnung und das *Shading* benötigt werden. Anschließend erfolgt das Speichern der Daten für die Hüllkörperprojektion: die Parameter des Hüllkörpers sowie die Texel-ID. Sowohl Objekt- als auch Hüllkörper-Daten werden dabei auf OpenGL Client-Seite zwischengespeichert, bevor sie während der Initialisierung der *GPUStreams*[11] zur Grafikkarte gesendet werden.

[11] Eine nähere Erläuterung zum Konzept der GPUStreams befindet sich im Anhang A

Für das Zwischenspeichern wird für jeden Objekt- beziehungsweise Hüllkörper-Typ eine eigene Datenstruktur verwendet. Dadurch liegen die Daten bereits auf OpenGL Client-Seite in der gleichen Datenorganisation vor, wie später auf der GPU. Dies ermöglicht einen schnelleren Datentransfer beim Senden der Informationen zur Grafikkarte.

Nach der vollständigen Traversierung des Szenegraphen erfolgt die Initialisierung der *GPUStreams*. Ein *GPUStream* abstrahiert dabei einen *Rendering Pass*. Wie Versuche zu Beginn der Arbeit gezeigt haben, ist die Verwendung je eines OpenGL Kontextes pro *Rendering Pass*, zumindest unter Linux, am effizientesten. Da OpenGL Kontexte unabhängig voneinander sind[12], kann der, für den *Rendering Pass* benötigte, Zustand der *State Machine* bereits in der Initialisierungsphase definiert werden. Damit sind während des Renderings nahezu keine Zustandsänderungen notwendig. Die einzige Ausnahme ist die Projektionsmatrix, welche bei einer nicht-statischen Kamera für jedes Bild aktualisiert werden muss. Neben dem Setzen des Zustandes der OpenGL *State Machine* erfolgt in der Initialisierungsphase auch das Laden, Kompilieren und Linken der Shader-Programme, sowie das bereits erwähnte Senden der Daten zur GPU. Für die Vertex-Daten der Hüllkörper werden *Vertex Buffer Objects* verwendet, da diese das Speichern beliebiger *Vertex Attributes* auf der GPU ermöglichen. Dadurch ist in der Rendering-Phase nahezu kein Datentransfer zwischen OpenGL-Client und -Server notwendig. Bei der Verwendung von *Texture Mapping* für das *Shading* erfolgt in dieser Phase das Speichern der Texturen auf der Grafikkarte.

2. Rendering Phase

Die Rendering-Phase beginnt mit der Bestimmung des sichtbaren Objektes für jedes Fragment. Dies erfolgt durch die Schnitt-Tests mit den Objekten der Szene.

Durch die Verwendung verschiedener Hüllkörper sowie der Integration von Dreiecke ergibt sich der in Abbildung 40 auf der rechten Seite dargestellte Ablauf. Ein Multi-Pass Verfahren mit einem Rendering-Pass pro Hüllkörper-Typ minimiert hier den benötigten Speicherplatz[13]. Die Reihenfolge der verschiedenen *Rendering Passes* der

[12] Eine Ausnahme sind Texturen und *Vertex Buffer Objects*, welche aus Gründen der Speichereffizienz von mehreren OpenGL Kontexten verwendet werden.

[13] Würde eine solche Trennung nicht erfolgen, so müssten auf Grund der Hardwarearchitektur der Grafikkarte für alle Vertices die identische Anzahl von Vertex Attributen gespeichert werden. Da für jeden Vertex eines Billboards drei Attribute verwendet werden, würden bei *Bounding Spheres* und *Bounding Boxes* zwei nicht verwendete Vertex Attribute gespeichert.

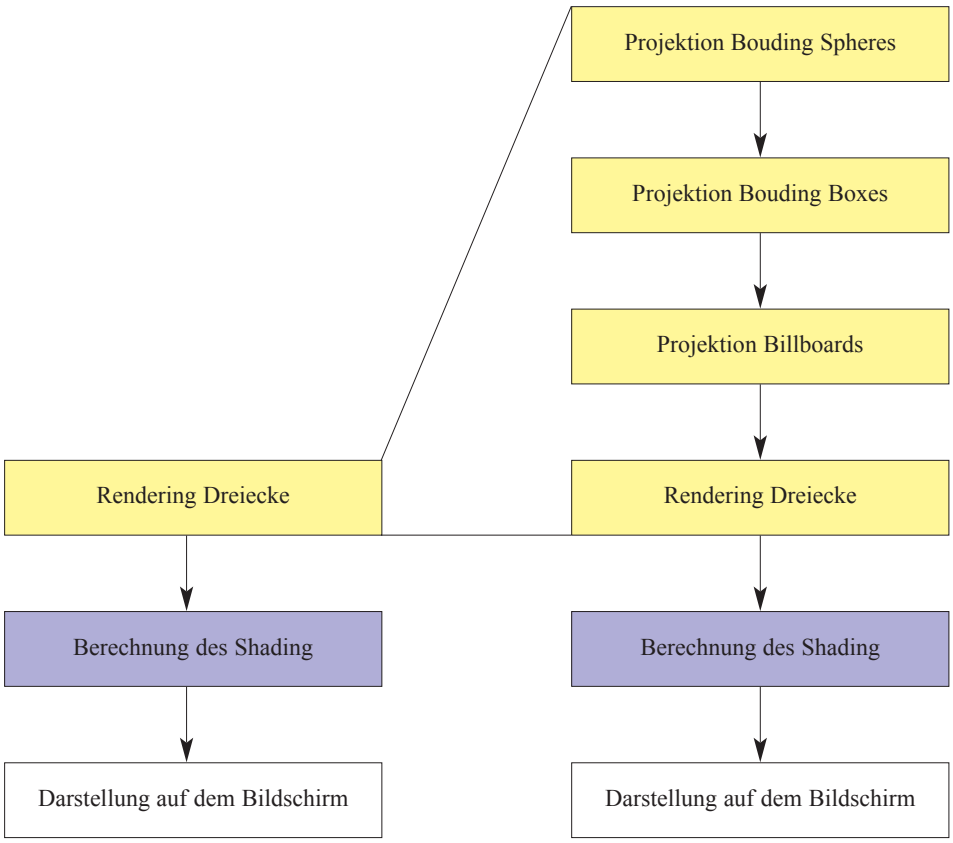


Abbildung 40: Ablaufdiagramm des Ray-Casters für Primitve zweiter Ordnung und Dreiecke

ersten Phase ist dabei beliebig. Der Programmablauf soll nun für die Berechnung eines Bildes näher erläutert werden.

Im ersten *Rendering Pass* erfolgt die Berechnung der Schnitt-Tests für die Objekte, deren Hüllkörper eine *Bounding Sphere* ist. Für jedes Objekt wird ein Punkt gerendert und, entsprechend dem unter 4.2.1.1. beschriebenen Verfahren, die für einen Objekt-Strahl-Schnitt in Frage kommende Teil-Strahlszene erzeugt. Mit Hilfe von *Varying Variables* wird durch die Projektion ebenfalls die Texel-ID des zu schneidenden Objektes im Fragment-Shader zur Verfügung gestellt. Durch diese kann, mit dem in Abbildung 42 dargestellten ID-Intervallschema, der Primitiv-Typ, und damit die zu verwendende Schnittfunktion, für das Objekt bestimmt werden.

Die für den Schnitt-Test notwendigen Objekt-Parameter werden aus dem Texturspeicher ausgelesen. Im Ray-Casting-System wird für jeden Objekt-Typ eine Textur, die so genannte Objekt-Textur, verwendet. Die Adresse beziehungsweise das

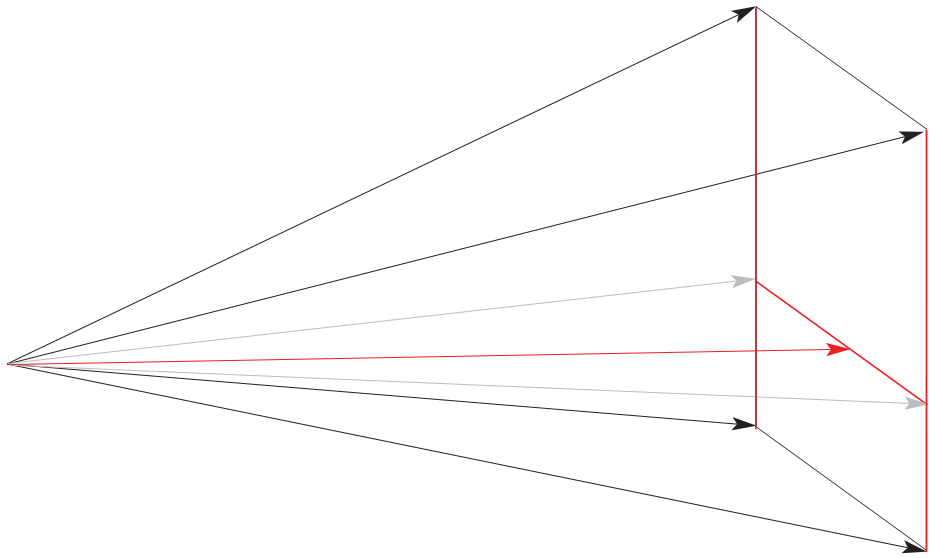


Abbildung 41: Interpolation der Strahlrichtung (rot) anhand der vier äußersten Strahlen (schwarz). Ebenfalls dargestellt sind die Interpolationsachsen (rot), sowie die Zwischenergebnisse (grau) nach der Interpolation entlang der Y-Achse in Bildschirmkoordinaten.

Texel für die Objekt-Parameter innerhalb der Objekt-Textur wird durch die Texel-ID angegeben.

Eine Ausnahme bilden Kugeln. Da diese auch als Hüllkörper verwendet werden, müssen die Objekt-Parameter bereits im Vertex-Shader zur Verfügung stehen. Wie Versuche gezeigt haben, ist es in diesem Fall günstiger, die Parameter direkt in die Fragment-Einheit zu transportieren, anstatt sie aus einer Textur auszulesen. Zusätzlich können im Vertex-Shader bereits einige strahlunabhängige Teile der Schnittgleichung gelöst werden. Dadurch wird vermieden, dass in parallelen Einheiten identischen Berechnungen ausgeführt werden.

Für den Schnitt-Test sind neben den Objekt- auch die Strahlinformationen notwendig. Der Strahlursprung, sowie die Richtungsvektoren der vier äußersten Strahlen, welche zur Berechnung der Strahlrichtung notwendig sind werden durch *Uniform Variables* zur Verfügung gestellt. Die Strahlrichtung kann damit durch eine gewichtete lineare Interpolation anhand der Pixelposition ermittelt werden (Abb. 41). Eine Alternative zu dem vorgestellten Verfahren ist die Berechnung der Strahlrichtungen für alle Fragmente beziehungsweise Augenstrahlen in einem zusätzlichen *Rendering Pass* vor der Hüllkörperprojektion. Für die Schnittberechnung ist bei einer solchen

Kugeln	[2, 4000)
Dreiecke	[4000, 20000)
Quadrics	[20000, 100000)

Abbildung 42: ID-Intervallschema zur Bestimmung des Objekttyps

Implementierung das Auslesen der Strahlrichtung aus einer Textur notwendig. Versuche zu Beginn der Arbeit haben jedoch gezeigt, dass die mehrfache Berechnung der Strahlrichtung die effizientere Möglichkeit ist.
Die Berechnung der Schnitt-Tests erfolgt mit den in [HAI89] erläuterten Verfahren. Durch die Verwendung der NV4X-Architektur, sowie den in GLSL automatisch zur Verfügung stehenden Vektordatentypen, war eine direkte und effiziente Implementierung möglich. Nach der Durchführung des Schnitt-Tests wird, falls dieser erfolgreich war, der Tiefenwert berechnet. Im Folgenden wird hierauf noch näher eingegangen. Anschließend werden der Schnittpunkt und die Texel-ID des geschnittenen Objektes sowie der Tiefenwert in den als *Render Target* verwendeten 32-Bit *floating point PBuffer* geschrieben. Zum Speichern der drei Koordinaten des Schnittpunktes wird der RGB-Kanal verwendet, für die Texel-ID der Alpha-Kanal. War der Schnitt-Test nicht erfolgreich, so wird das Fragment verworfen. Dies kann mit dem Befehl `discard` oder mit dem Schreiben eines Tiefenwerts von 1.0 erreicht werden.
Der Ablauf bei der Projektion einer *Bounding Box* beziehungsweise eines *Billboards* ist, bis auf die unterschiedlichen Vertex-Shader-Programme, identisch.
Das Rendern der Dreiecke erfolgt im letzten *Pass* vor dem *Shading*. Die Operationen im Vertex-Shader entsprechen dabei denen des *Feed Forward Renderings*. Zusätzlich wird gewährleistet, dass die Texel-ID im Fragment-Prozessor zur Verfügung steht. Dafür werden auch hier *Varying Variables* verwendet. In den durch die Projektion entstandenen Fragmenten erfolgt, entsprechend dem Hüllkörperprojektions-Verfahren, die Schnittberechnung zwischen den Augenstrahlen und dem Objekt. Die Schnittberechnung ist dabei, wie im folgenden Abschnitt noch näher erläutert wird, für eine konsistente Tiefensortierung der Szene notwendig. Für Dreiecke ist damit die Berechnung der Distanz des Schnittpunkts vom Strahlursprung ausreichend, wodurch einige Berechnungsschritte, welche zur vollständigen Bestimmung des Schnittpunkts notwendig wären, nicht durchgeführt werden müssen. Ein *Point in Polygon*-Test ist

ebenfalls nicht notwendig. Dieser kann entfallen, da bei Dreiecken das Objekt selbst und kein Hüllkörper projiziert wird. Dadurch ist gewährleistet, dass jedes entstandene Fragment beziehungsweise jeder erzeugte Augenstrahl tatsächlich das Objekt schneidet.

Ein wichtiger Teil der Rendering-Phase ist die Ermittlung des dem Augenpunkt am nächsten liegenden Schnittpunktes, das so genannte *Depth Buffering*. Im Ray-Casting-System wird dafür die Tiefensortierung der OpenGL Standard-Pipeline verwendet. Allerdings kann, wie Abbildung 42 veranschaulicht, nicht der durch die Projektion des Hüllkörpers automatisch generierte Tiefenwert benutzt werden. Für eine korrekte Tiefensortierung ist die Verwendung der Entfernung des Schnittpunkts vom Strahlursprung notwendig.
Im implementierten Ray-Casting-System wird die Distanz in Weltkoordinaten verwendet, da für die Primitive bereits die Berechnung des Objekt-Strahl-Schnittpunktes in diesem Koordinatensystem erfolgt. Auf Grund des beschränkten Wertebereichs des Tiefen-Buffers von [0,1] ist eine Skalierung der Tiefenwerte notwendig. Der Skalierungsfaktor ist dabei in Abhängigkeit von der Szene zu wählen. Zum einen muss er größer sein, als die maximale Distanz eines Schnittpunktes vom Strahlursprung, zum anderen nimmt jedoch die Genauigkeit der Tiefensortierung mit der Größe des Faktors ab.
Die Verwendung der Distanz des Schnittpunktes vom Strahlursprung in Weltkoordinaten für die Tiefensortierung macht die teilweise Schnittberechnung für Dreiecke notwendig. Der durch die Projektion eines Dreiecks durch die OpenGL Standard-Pipeline erzeugte Tiefenwert ist zwar die Entfernung des Objekt-Strahl-Schnittpunktes vom Augenpunkt, allerdings in normalisierten Device-Koordinaten. Eine Rücktransformation in Weltkoordinaten ist, wie Tests gezeigt haben, aufwändiger als die für Dreiecke beziehungsweise Flächen durchzuführenden Operationen zur Ermittlung der Distanz des Schnittpunktes vom Strahlursprung.
Um ein konsistentes Ergebnis für alle Objekte zu erhalten, ist die Verwendung eines *Depth Buffers* in allen *Rendering Passes*, welche in die Tiefensortierung einfließen sollen, notwendig. Dies impliziert die Verwendung eines *PBuffer*, da der *Depth Buffer* ein Bestandteil des *Render Targets* ist.

Eine Alternative zur Verwendung der Distanz in skalierten Weltkoordinaten ist das *Depth Buffering* mit normalisierten Device-Koordinaten. Dadurch ist bei Dreiecken

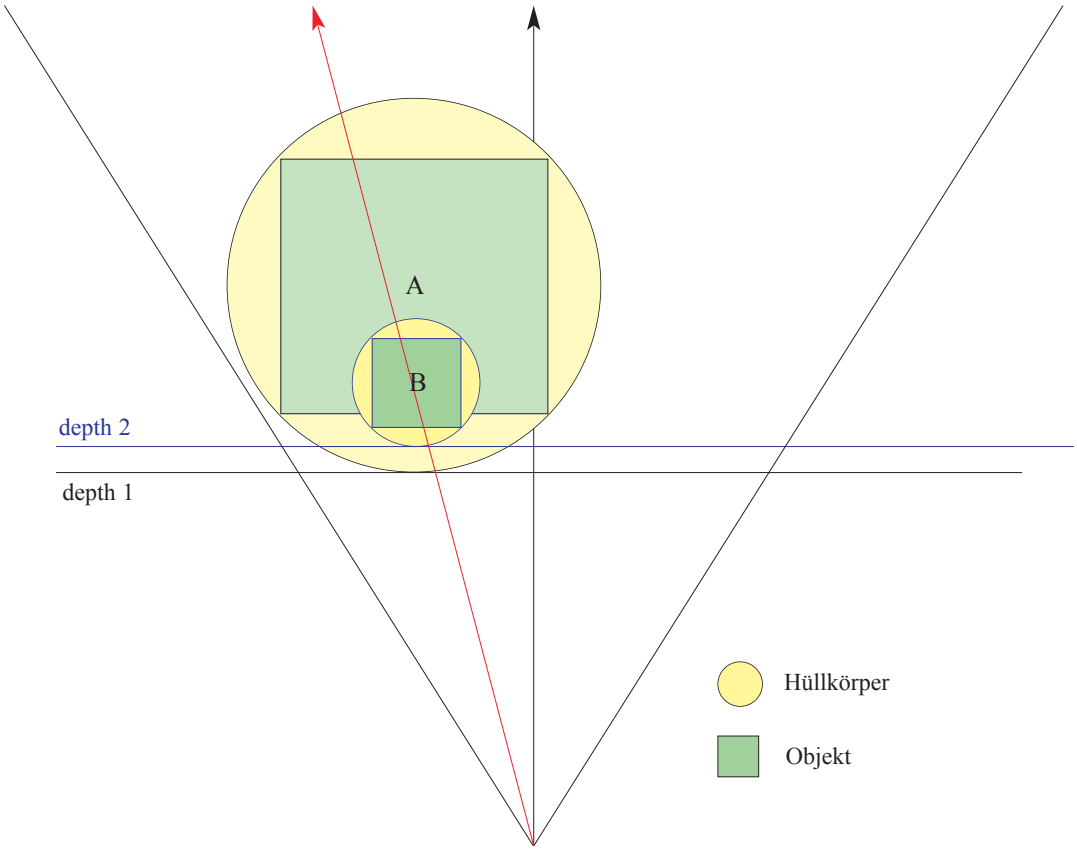


Abbildung 43: Depth Buffering. Werden die Tiefen-Werte der Hüllkörper verwendet, so wird der Strahl-Objekt Schnittpunkt mit A in den Tiefen-Buffer geschrieben, obwohl der Schnittpunkt des Strahls mit B näher am Augenpunkt liegt und damit B das sichtbare Objekt in diesem Pixel ist.

keine Berechnung der Entfernung des Schnittpunktes vom Augenpunkt notwendig. Es müssen jedoch die im Fragment-Shader ermittelten Schnittpunkte in Welt-Koordinaten, zum Beispiel für Kugeln und Quadrics, in normalisierte Device-Koordinaten überführt werden. Für Szenen, in denen Dreiecke beziehungsweise Flächenprimitive die deutliche Mehrzahl der Objekte darstellen, ist mit diesem Ansatz der Verwendung des Tiefenwerts in normalisierten Device-Koordinaten ein Geschwindigkeitsvorteil gegenüber der aktuellen Implementierung der Tiefensortierung zu erwarten.

Das Ergebnis des ersten Teils der Rendering-Phase ist ein *PBuffer* in welchem für jedes Fragment die Texel-ID des dem Augenpunkt am nächsten liegende Objektes im Alpha-Kanal gespeichert ist. Darüber hinaus steht in Abhängigkeit vom Objekt-Typ der Schnittpunkt im RGB-Kanal zur Verfügung.

Das *Shading* erfolgt im letzten *Rendering Pass* vor der Darstellung auf dem Bildschirm. Dabei wird zunächst der Inhalt des *PBuffer,* welcher in den vorangegangenen *Rendering Passes* als *Render Target* verwendet wurde, in eine Textur kopiert. Damit können die ermittelten Schnittpunkte, sowie die Texel-ID des geschnittenen Objektes ausgelesen werden. Die Texel-ID ermöglicht anschließend auch in diesem *Rendering Pass* das Lesen der Objekt-Parameter aus der Objekt-Textur. Zunächst wird dabei die ID des Objekt-Materials ausgelesen, da dieses die folgenden Schritte festlegt. Entspricht die Material-ID einer Textur, so werden die Texturkoordinaten berechnet und anschließend die Farbinformationen aus der durch die ID definierten Textur ausgelesen. Wird keine Textur verwendet, so ist lediglich das Auslesen der Material-Eigenschaften aus einer Material-Textur notwendig. Anschließend erfolgt das *Phong Shading* für die Berechnung der Beleuchtung. Die dafür benötigte Objektnormale kann, in Abhängigkeit vom Objekttyp, mit dem Schnittpunkt und den, durch die Objekt-Textur verfügbaren, Objekt-Parametern, ermittelt werden.

Das *Shading* in einem zusätzlichen *Rendering Pass* führt dazu, dass Objekt-Parameter zum Teil zweimal aus der Objekt-Textur ausgelesen werden. Wie sich jedoch gezeigt hat, ist selbst bei einfachem *Phong Shading* das mehrfache Auslesen der Objekt-Informationen günstiger, als die mehrfache Berechnung des *Shading*s.

4.2.2.2. Ergebnisse

Im Folgenden werden einige Testergebnisse des Ray-Casting-Systems vorgestellt. Als Testumgebung wurde das in Anhang B vorgestellte Hardware-System verwendet.

Die Ergebnisse A bis G zeigen die Geschwindigkeit des Ray-Casting-Systems bei verschiedenen Parameterwerten. Während in B bis E Belastungssituationen des Systems simuliert wurden, erfolgte in A und G die Ermittelung der maximalen Verarbeitungsleistung der Hüllkörperprojektion beziehungsweise der Schnittberechnungen.
Die visuelle Qualität der Ray-Casting-Systems wird in H demonstriert.

A.) Verarbeitungsleistung für *Billboard*s, *Bounding Boxes* und *Bounding Spheres* in Abhängigkeit von der Anzahl der Hüllkörper

B.) *Bounding Boxes* und *Billboards* in Abhängigkeit von der Anzahl der Objekte

C.) Kugeln bei der Verwendung einer *Bounding Sphere* als Hüllkörper

D.) *Bounding Box* und *Billboard* in Abhängigkeit vom Rotationswinkel

E.) Vergleich der Leistung des Ray-Casting-Systems mit und ohne *Shading*

F.) Einfluss der Tiefenkomplexität auf die Effizienz des Ray-Casting-Systems

G.) Maximal erreichbare Anzahl von Schnitt-Tests

H.) Visuelle Qualität des Ray-Casting-Systems

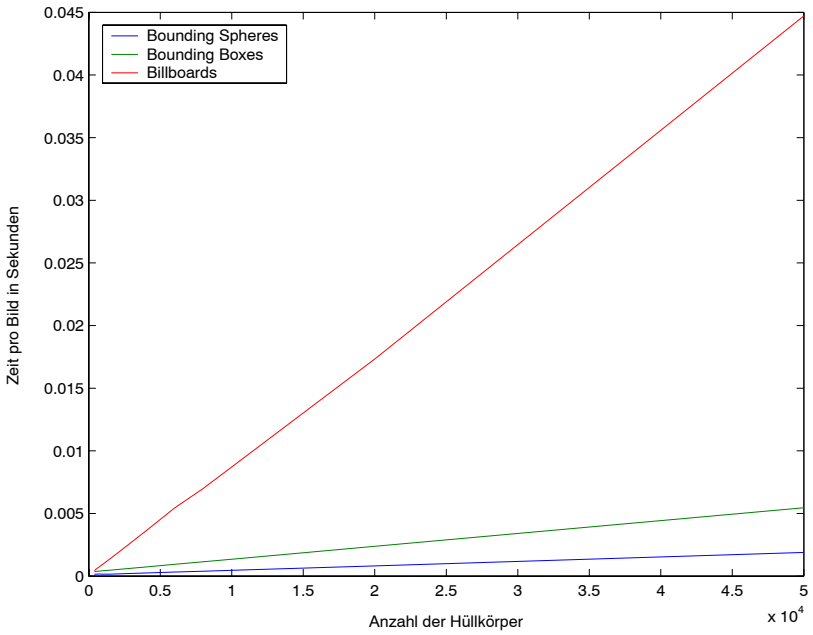


Abbildung 44: Verarbeitungsleistung für Billboards, Bounding Boxes und Bounding Spheres in Abhängigkeit von der Anzahl der Hüllkörper.

A.) Verarbeitungsleistung für Billboards, Bounding Boxes und Bounding Spheres in Abhängigkeit von der Anzahl der Hüllkörper.

Bei diesem Versuch, dessen Ergebnisse in Abbildung 44 und 45 dargestellt sind, wurde die maximal mögliche Verarbeitungsleistung der Hüllkörperprojektion untersucht. Um einen Einfluss der Rasterisierungseinheit oder des Fragment-Prozessors auf das Ergebnis zu vermeiden, wurden die Vertices in der letzten Operation des Vertex-Shader Programms außerhalb des *Clipping Space* positioniert.

Die *Bounding Sphere* ist in Bezug auf die Projektion der effizienteste Hüllkörper. Durch den hohen Aufwand zur Berechnung der vier Eckpunkte ist das *Billboard*, trotz der geringeren Anzahl von Vertices, nur bei der Verwendung von weniger als 300 Hüllkörper effizienter als die *Bounding Box* (Abb. 45). Bei mehr als 300 Hüllkörpern ist die Leistung der Vertex-Einheit nicht mehr ausreichend, um alle Vertices sofort zu verarbeiten (Abb. 44).
Der Anstieg der Verarbeitungszeit bei *Bounding Boxes* ist relativ gering. Dies kann auf die Verwendung der *Vertex Buffer Objects* sowie die hohen Datentransferraten des internen Bussystems der Grafikkarte zurückgeführt werden.

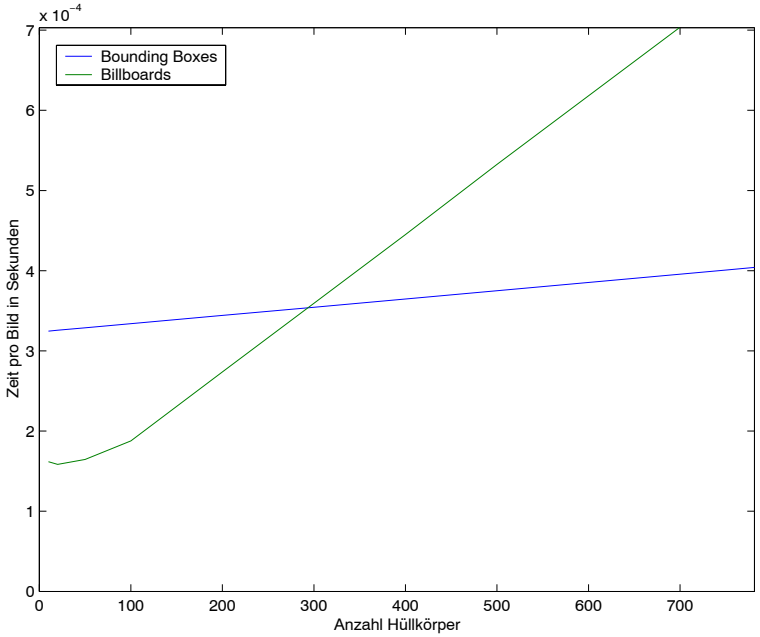


Abbildung 45: Verarbeitungsleistung für Billboards, Bounding Boxes und Bounding Spheres in Abhängigkeit von der Anzahl der Hüllkörper.(Detailansicht zu Abbildung 43)

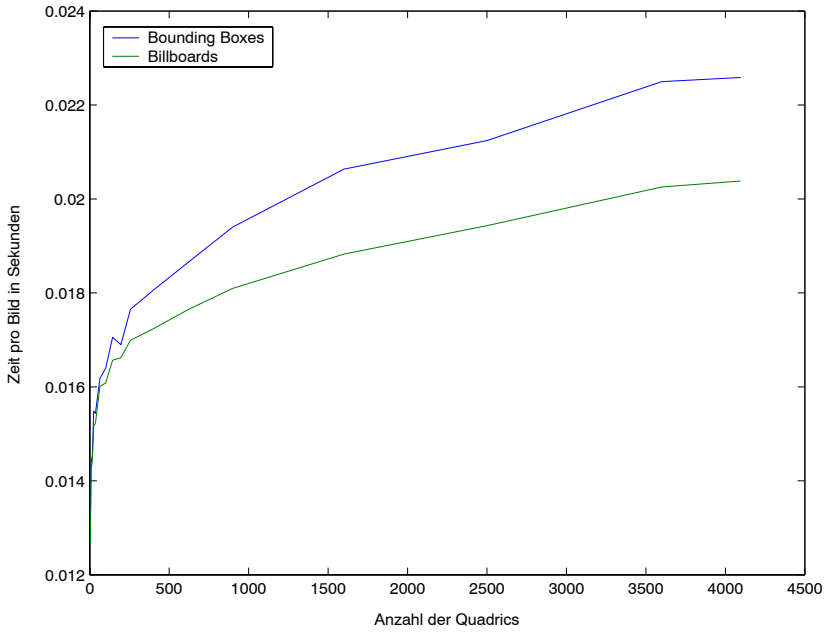


Abbildung 46: Bounding Boxes und Billboards in Abhängigkeit von der Anzahl der Objekte

B.) Bounding Boxes und Billboards in Abhängigkeit von der Anzahl der Objekte

Bei diesem Versuch, dessen Ergebnisse in Abbildung 46 dargestellt sind, wurde die Verarbeitungsleistung des Ray-Casting-Systems bei der Verwendung von *Bounding Boxes* und *Billboards* untersucht. Es wurden dabei alle *Rendering Passes* des Ray-Casting-System ausgeführt. In der Fragment-Einheit wurden, unabhängig vom Hüllkörper-Typ und der Anzahl der Objekte, immer 512 x 512 Schnitt-Tests mit einem Quadric durchgeführt. Die Projektion erfolgt so, dass die *Bounding Boxes* und das *Billboard* die gleiche Anzahl von Fragmenten erzeugen.

Bei der verwendeten Projektion ist das *Billboard*, unabhängig von der Anzahl der Objekte, effizienter als die *Bounding Box*. Dieses Ergebnis ist insbesondere im Vergleich mit Abbildung 44 interessant, welche zeigt, dass die Projektion der *Billboards* aufwändiger ist als die der *Bounding Boxes*. Der Unterschied resultiert möglicherweise aus der unterschiedlichen Rasterisierung von *Billboard* und den Flächen der *Bounding Box*. Die geringere Größe der einzelnen Flächen der *Bounding Box* könnte zu einem *Pipeline Stall* in der Fragment-Einheit führen; das heißt, dass die entstehenden Fragmente der einzelnen Flächen nicht so verarbeitet werden, dass eine kontinuierliche Auslastung des Fragment-Prozessors gewährleistet ist.

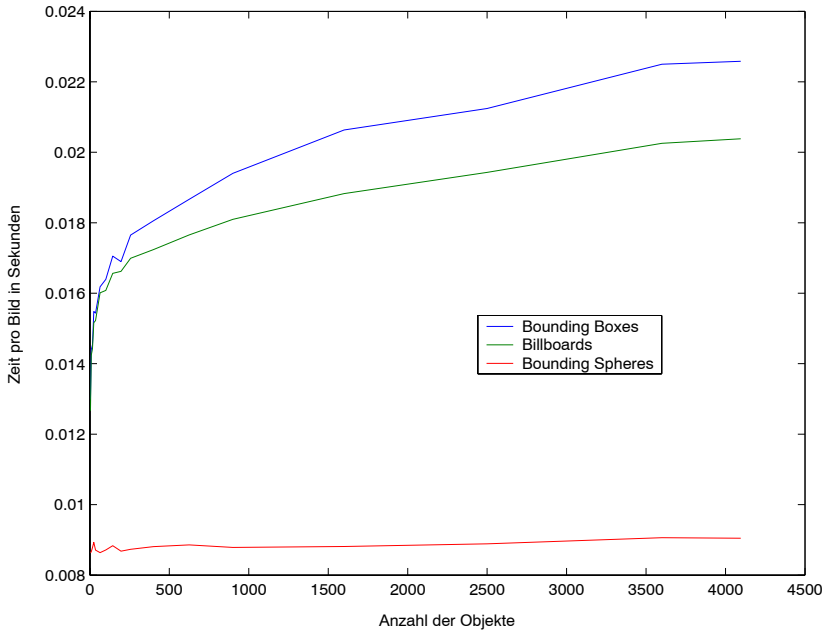


Abbildung 47: Kugeln bei der Verwendung einer Bounding Sphere als Hüllkörper

C.) Kugeln bei der Verwendung einer Bounding Sphere als Hüllkörper

Bei diesem Versuch, dessen Ergebniss in Abbildung 47 dargestellt ist, wurde die Effizienz des Schnitt-Tests mit Kugeln bei der Verwendung einer *Bounding Sphere* als Hüllkörper untersucht. Es wurden alle *Rendering Passes* des Ray-Casting-System ausgeführt. Zum Vergleich ist die Verarbeitungsleistung des Systems bei der Schnittberechnung mit einem Quadric und der Verwendung von *Bounding Boxes* beziehungsweise *Billboards* als Hüllkörper im Diagramm abgetragen (aus Abb. 46).

Wie Abbildung 47 zeigt, wächst die für ein Bild benötigte Zeit nur sehr leicht mit der Anzahl der Kugeln. Dies zeigt, dass sowohl Vertex- als auch Fragment-Prozessor nicht überlastet sind.
Der deutliche Unterschied zwischen Bounding Sphere und Bounding Box / Billboard resultiert zum einen aus der schnelleren Projektion des Hüllkörpers pro Objekt bei der Verwendung der Bounding Sphere, zum anderen auch aus der für Kugeln wesentlich weniger aufwendigen Schnittberechnung.

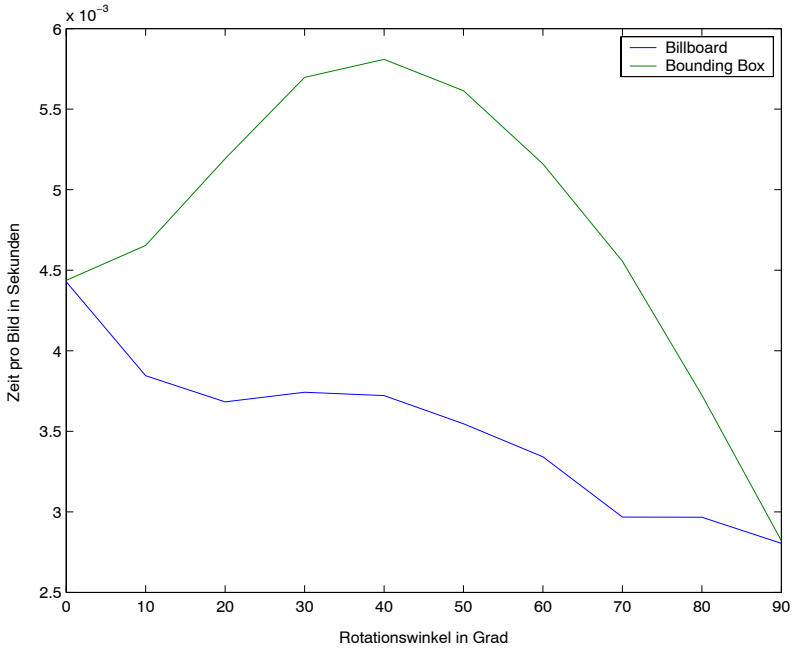


Abbildung 48: Bounding Box und Billboard in Abhängigkeit vom Rotationswinkel (0° entspricht einer senkrechten Orientierung des Zylinders)

D.) Bounding Box und Billboard in Abhängigkeit vom Rotationswinkel

Bei diesem Versuch, dessen Ergebnisse in Abbildung 48 dargestellt sind, wurden die Abhängigkeit der Effizienz von *Bounding Boxes* und *Billboards* vom Rotationswinkel des Objekts untersucht. Es wurde jeweils der Zylinder in Abbildung 49 gerendert und alle *Rendering Passes* des Ray-Casting-Systems ausgeführt.

Für eine vertikale (0°) beziehungsweise horizontale (90°) Orientierung des Zylinders zeigen *Bounding Box* und *Billboard* eine vergleichbare Leistung. Bei einem Winkel von 45° ist die *Bounding Box* jedoch wesentlich effizienter. Wie Abbildung 49 zeigt, ist in diesem Fall die Projektion des *Billboards* wesentlich größer als die der *Bounding Box*, so dass bei der Verwendung des *Billboards* mehr Schnitt-Tests in der Fragment-Einheit ausgeführt werden müssen.

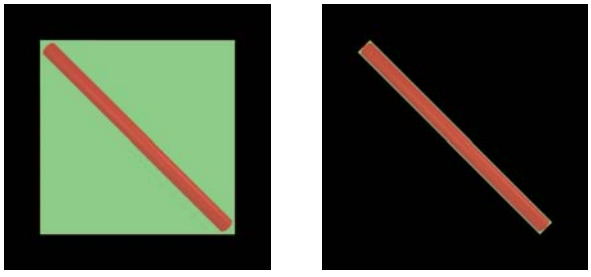


Abbildung 49:
Visueller Vergleich der Hüllkörperprojektion (grün) für einen Zylinder (rot) bei Billboard (links) und Bounding Box (rechts).
Die Bounding Box ist im rechten Bild nur als Silhouette erkennbar.

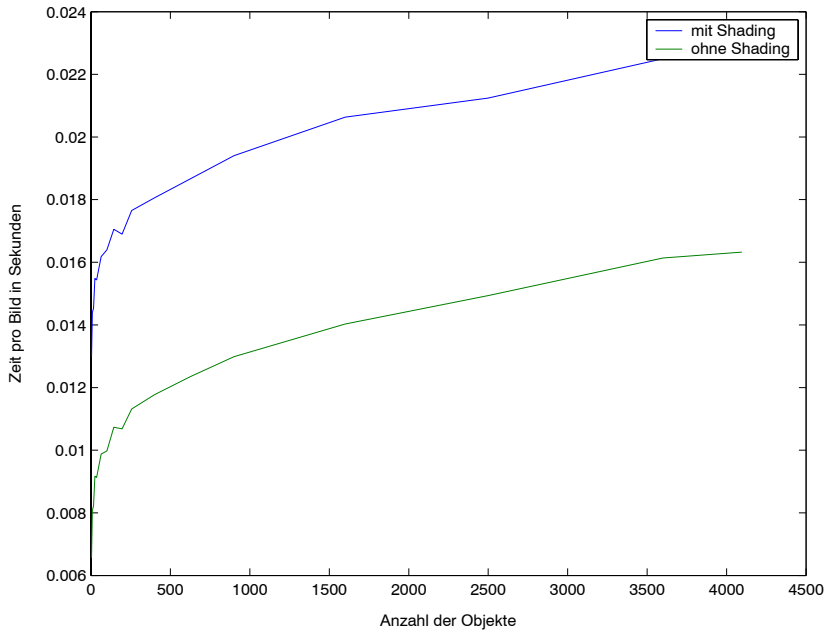


Abbildung 50: Vergleich der Leistung des Ray-Casting-Systems mit und ohne Shading

E.) Vergleich der Leistung des Ray-Casting-Systems mit und ohne Shading

Bei diesem Versuch, dessen Ergebnisse in Abbildung 50 dargestellt sind, wurde der Einfluss des *Shadings* auf die Gesamtleistung des Systems untersucht. Als Objekte wurde Quadrics verwendet, als Hüllkörper *Bounding Boxes*. Es wurden dabei alle *Rendering Passes* des Ray-Casting-Systems ausgeführt.

Das *Shading* erfolgt beim Ray-Casting-System pro Pixel, so dass eine konstante Zeitersparnis beim Rendern ohne *Shading* erzielt wird. Bei einem *Shading* pro Fragment würden die Kurven mit steigender Objektanzahl divergieren.

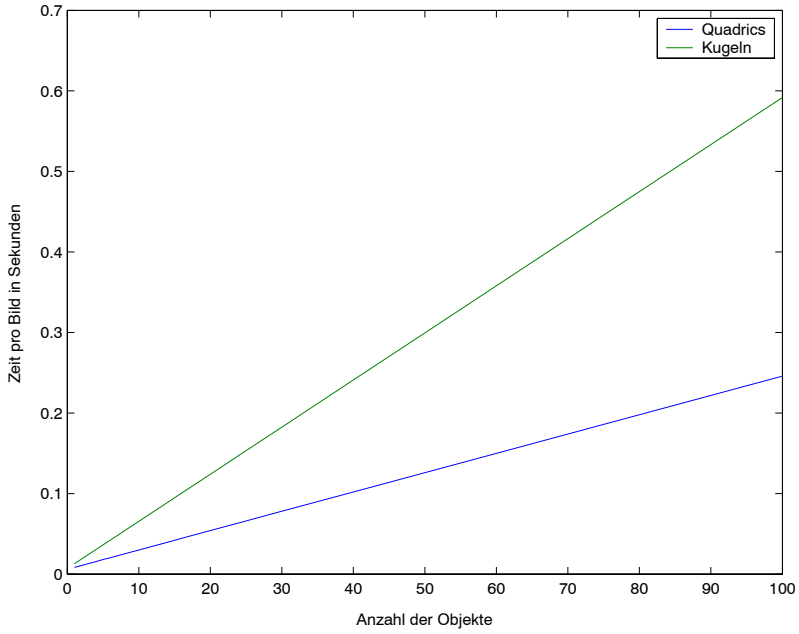
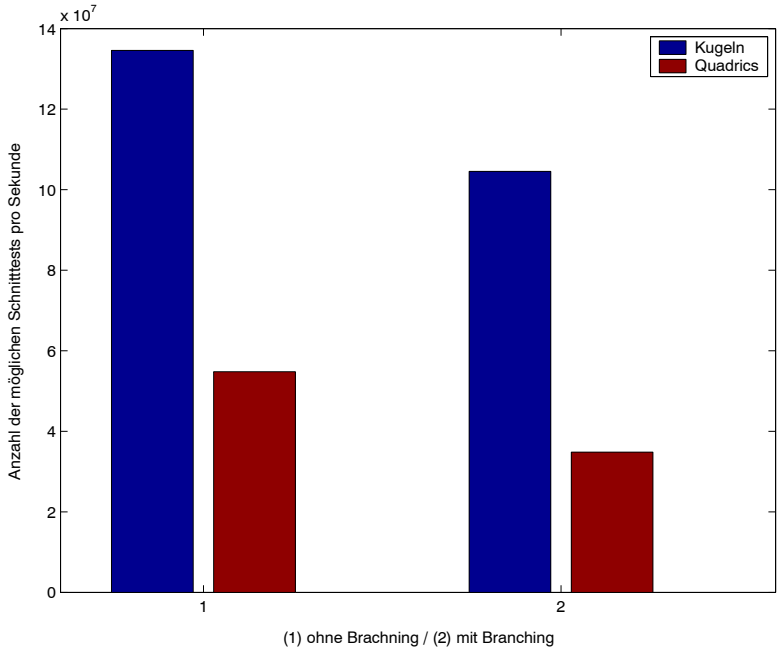


Abbildung 51: Einfluss der Tiefenkomplexität auf die Effizienz des Ray-Casting-Systems

F.) Einfluss der Tiefenkomplexität auf die Effizienz des Ray-Casting-Systems

Bei diesem Versuch, dessen Ergebnisse in Abbildung 51 dargestellt sind, wurde der Einfluss der Tiefenkomplexität auf die Geschwindigkeit des Ray-Casting-Systems untersucht. Es wurden jeweils bildschirmfüllende Primitive gerendert. Die Position dieser war identisch, lediglich die Anzahl wurde verändert. Für Quadrics wurden *Bounding Boxes* als Hüllkörper verwendet, für die Kugeln *Bounding Spheres*. Es wurden dabei alle *Rendering Passes* des Ray-Casting-Systems ausgeführt.

Sowohl für *Bounding Spheres* als auch *Bounding Boxes* als Hüllkörper besitzt das Ray-Casting-System eine lineare Abhängigkeit von der Tiefenkomplexität der Szene. Der Anstieg der Geraden ist für Kugeln auf Grund der weniger aufwändigen Schnittberechnung flacher.
Bei der Verwendung von *slabs* ist ein deutliches Abflachen des Kurvenverlaufs zu erwarten. Allerdings besitzt das Ray-Casting-System auch bei der Verwendung von slabs eine lineare Tiefenkomplexität da weiterhin alle Hüllkörper projiziert werden müssen und für alle erzeugten Fragmente mindestens ein Auslesen des Tiefenwerts aus dem *Depth Buffer* der letzten *slab* notwendig ist.



	Ohne Branching	Mit Branching
Kugeln	134.598.098	104.548.270
Quadrics	54.801.727	34.836.506
Dreiecke	282.596.474[13]	-

Abbildung 52: Maximal erreichbare Anzahl von Schnitt-Tests

G.) Maximal erreichbare Anzahl von Schnitt-Tests

Bei diesem Versuch, dessen Ergebnisse in Abbildung 52 dargestellt sind, wurde die maximal erreichbare Anzahl von Schnitt-Tests in Abhängigkeit vom Objekttyp untersucht. Für Kugeln standen die Objekt-Parameter, durch die Verwendung von *Bounding Spheres* als Hüllkörper, direkt im Fragment-Shader zur Verfügung. Für Quadrics wurden diese aus der Objekt-Textur ausgelesen wobei lediglich ein gutes Caching-Verhalten sichergestellt wurde. Branching gibt in Abbildung 52 an, dass eine bedingte Anweisung im Shader-Programm enthalten ist, wobei in diesem Versuch keine Vermischung von Objekttypen stattfand.

Die Anzahl der möglichen Schnittberechnungen ist auf der GPU deutlich größer als auf der CPU [WAL01]. Der Overhead durch die bedingte Anweisung ist auch bei kohärentem Programmfluss erheblich.

[14] Anzahl der maximal ausführbaren Distanzberechnungen zwischen Schnittpunkt und Augenpunkt

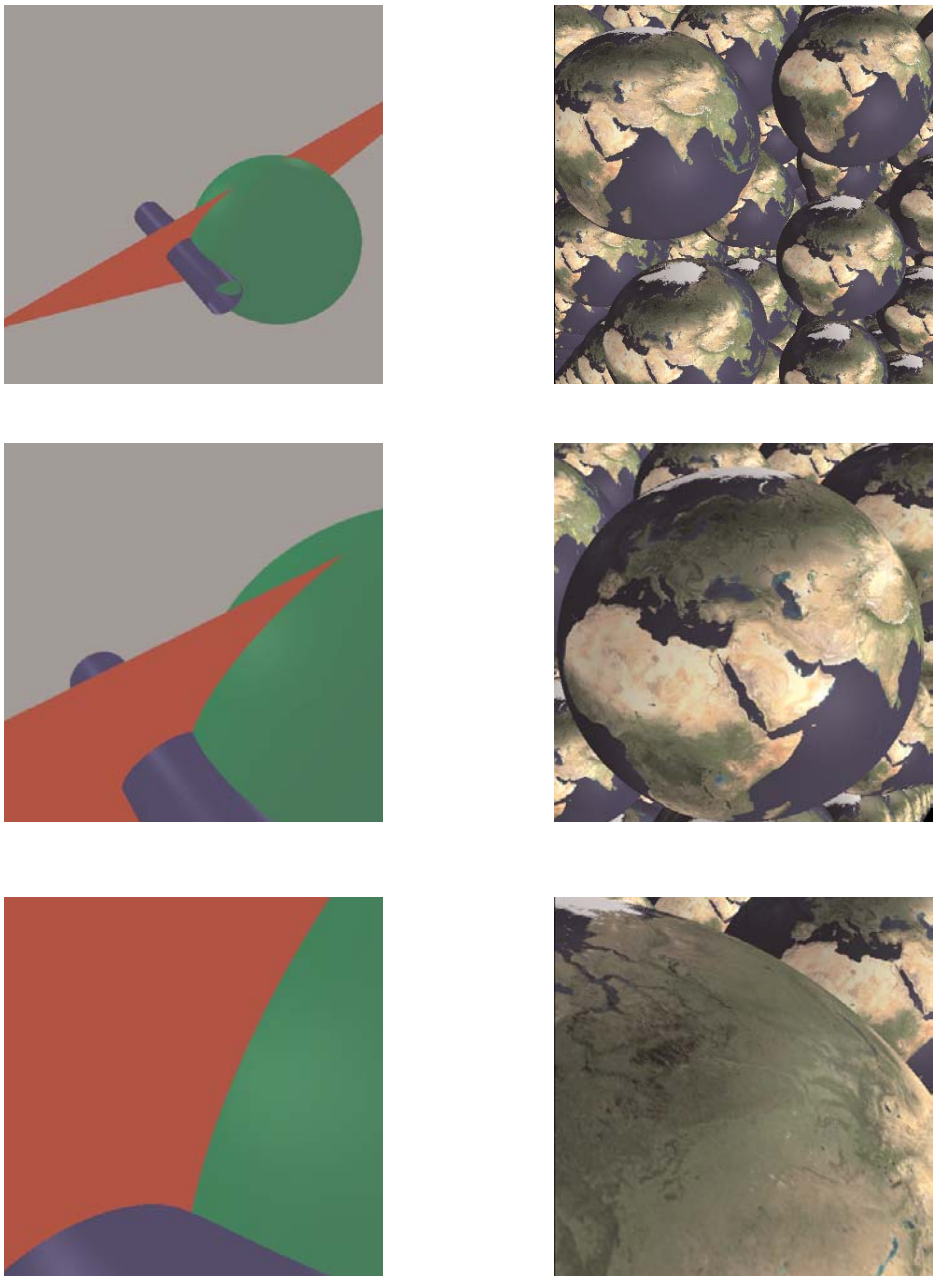


Abbildung 53: Visuelle Qualiät des Ray-Casting-Systems bei der Berechnung der Geometrie pro Pixel

H.) Visuelle Qualität des Ray-Casting-Systems

Wie Abbildung 53 zeigt, ist unabhängig von der Entfernung durch die pixelgenaue Berechnung immer eine sehr hohe visuelle Qualität gewährleistet. Die abgebildeten Szenen werden, blickpunktunabhängig immer mit interaktiven Framraten gerendert. Abbildung 54 und 55 zeigen zwei Beispielszenen welche mit 85 beziehungsweise 170 Bildern pro Sekunde gerendert werden.

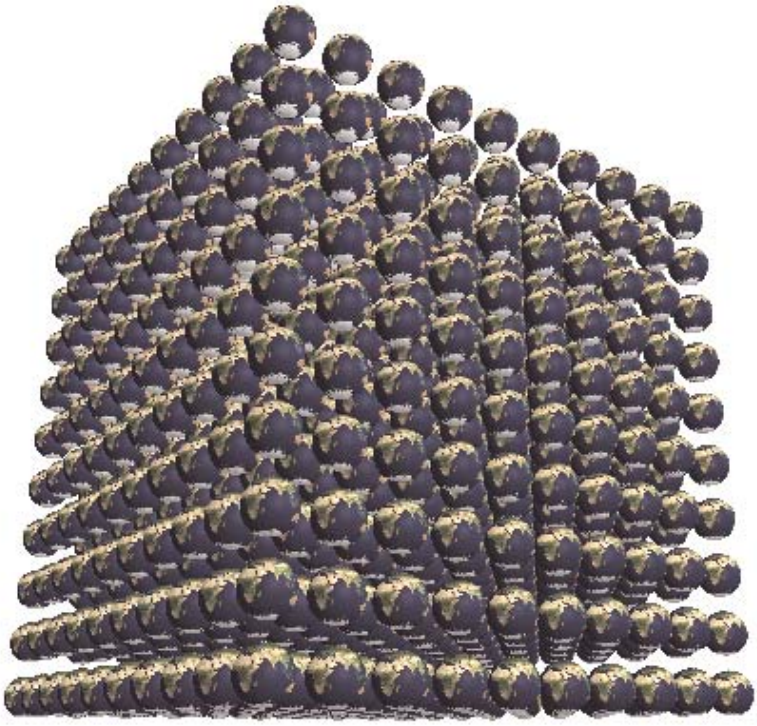


Abbildung 54: 1000 texturierte Kugeln, welche mit 85 Bildern pro Sekunde gerendert werden.

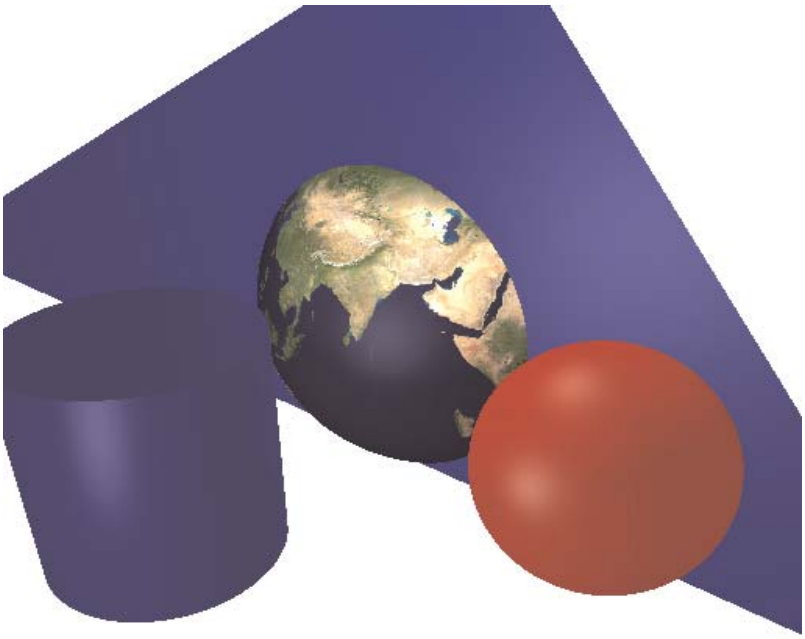


Abbildung 55: Szene mit Dreieck, Zylinder und Kugeln, welche mit 170 Bildern pro Sekunde gerendert wird.

4.2.2.3. Diskussion

Bei der Implementierung der Hüllkörperprojektion hat sich gezeigt, dass für allgemeine Objekte und beliebige Blickpunkte die gute Approximation eines Objektes durch einen Hüllkörper einer schnellen Hüllkörperprojektion vorzuziehen ist (Abb.48). Der Grund ist die Limitierung des Ray-Casting-Systems durch die Verarbeitungsleistung der Fragment-Einheit. Die *Bounding Box* ist im implementierten System deshalb am besten als Hüllkörper für beliebige Objekte geeignet. Für Spezialfälle wie symmetrische Objekte oder einen beschränkten Freiheitsgrad der Kamera, können jedoch durch die Verwendung von *Bounding Spheres* oder *Billboards* Geschwindigkeitsvorteile erzielt werden (Abb. 46 und Abb. 47).
Auf die Auswahl optimierter Schnittfunktionen wurde in dieser Arbeit kein Schwerpunkt gelegt, da zunächst grundsätzliche Fragen bezüglich der Implementierbarkeit zu untersuchen waren. Die erreichte Schnittleistung auf der GPU liegt dabei aber bereits, wie Abbildung 52 zeigt, um einen Faktor 10 über der von CPU-Implementierung [WAL01].
Bei der Arbeit mit dem Ray-Casting-System hat sich gezeigt, dass mit einem Szenengraph auch bei einem GPU-basierten Raycaster eine einfache und effiziente Verwaltung der Szenedaten möglich ist.

Das größte Problem des Ray-Casting-Systems ist im Moment die lineare Abhängigkeit der Rendering Geschwindigkeit von der Tiefenkomplexität (Abb. 51). Neben der unter 3.6.3. erläuterten *slab*s wurde während der Implementierung eine weitere Möglichkeit zur Reduzierung der Tiefenkomplexität gefunden.
Anstelle des Kopierens des *Depth Buffers* nach dem Rendern jeder *slab* wird dieser direkt als Textur verwendet. Dies ist durch die `NV_render_depth_texture` OpenGL Extension [WGL02b] möglich, welche allerdings zur Zeit nur unter Windows zur Verfügung steht. Entgegen den Spezifikationen, wird damit in einem *Rendering Pass* sowohl in den *Depth Buffer geschrieben* als auch aus diesem *gelesen*. Dies ermöglicht eine wesentliche Reduzierung der notwendigen Schnittoperationen, da die Distanzoptimierung mit dem ausgelesenen Tiefenwert möglich ist.
Im Fragment-Shader wird dazu, vor der Schnittberechnung, an der Pixelposition des Fragments der Tiefenwert aus dem als Textur gebundenen *Depth Buffer* ausgelesen. Analog zu dem Vorgehen bei *slabs* kann durch einen Vergleich des ausgelesenen

Tiefenwerts mit der Position der *Bounding Box* entschieden werden, ob ein Schnitt-Test notwendig ist.
Allerdings erfolgt auch beim *Depth Buffer*, wie bei jeder anderen Textur, ein Caching der Daten, so dass nicht in jedem Fall gewährleistet ist, dass der Wert, den man beim Auslesen erhält, tatsächlich der zuletzt für dieses Fragment geschriebene Tiefenwert ist. Dies führt jedoch zu keinen Fehlern, sondern lediglich zu der Berechnung einiger nicht notwendiger Schnitt-Tests.
Durch die *Depth Buffer*-Textur kann damit erreicht werden, dass die Anzahl der im Fragment-Shader durchgeführten Schnittoperationen nahezu unabhängig von der Anzahl der Objekte in der Szene ist. Allerdings ist auch bei der Verwendung der *Depth Buffer*-Textur für jedes Primitiv die Projektion des Hüllkörpers notwendig, sowie für jedes entstandene Fragment der Vergleich mit dem ausgelesenen Tiefenwert. Dadurch besitzt das Hüllkörperprojektionsverfahren weiterhin eine lineare Komplexität für die Anzahl der Objekte. Der Faktor des linearen Wachstums ist dabei, durch die Verwendung der Grafikkarte für die Hüllkörperprojektion, jedoch deutlich geringer als bei der Durchführung der Schnitt-Tests. Die Integration dieses Verfahrens zur Reduzierung der Tiefenkomplexität in das Ray-Casting-System muss zukünftigen Arbeiten überlassen werden.

4.2.3. NURBS Ray-Casting auf der GPU

Im Rahmen der Arbeit wurde ein Verfahren für das Ray-Casting von *Nonuniform Rational B-Spline (NURBS)*-Oberflächen auf der Grafikkarte implementiert. Dieses basiert auf dem von Martin et. al. in [MAR00] vorgeschlagenen Ansatz und wurde zur Integration in das bestehende Ray-Casting-System erweitert.
Dieses Verfahren erschien geeignet, da bei diesem zunächst der Schnitttest zwischen einem Strahl und einem Hüllkörper erfolgt, bevor gegebenenfalls der Schnitttest mit dem Objekt selbst berechnet wird. Dies entspricht einer alternativen Interpretation des Hüllkörperprojektions-Verfahrens, bei welchem durch die Projektion von Hüllkörpern die Augenstrahlen ermittelt werden, welche sich mit den Hüllkörpern schneiden, und nur für diese im Fragment-Shader ein Strahl-Objekt Schnitttest berechnet wird.

4.2.3.1. Überblick über das Verfahren

Für das Ray-Casting wird zunächst in einem Vorberechnungsschritte eine Menge von *Bounding Boxes* erzeugt, welche zusammen eine konvexe Hülle der NURBS-Oberfläche bilden. Durch die Projektion der *Bounding Boxes* (Abb. 56) wird, neben der Erzeugung der Teil-Strahlszene, welche für einen Objekt-Strahl-Schnitt in Frage kommt, der Aufsatzpunkt für eine *Newton Iteration* zur Verfügung gestellt. Mit der Newton Iteration wird anschließend in der Fragment-Einheit der Objekt-Strahl Schnitttest durchgeführt.

Abbildung 56: NURBS-Oberfläche mit der Visualisierung von zwei Bounding Boxes

4.2.3.2. Grundlagen des Verfahrens

Das Problem der Schnittberechnung eines Strahles mit einer NURBS-Oberfläche kann wie folgt beschrieben werden:

Es sei Q eine NURBS-Oberfläche

$$Q : S(u,v) = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} C_{i,j} B_{j,k_u}(u) B_{i,k_v}(v) \tag{1}$$

mit dem Kontrollpunktgitter P, welches $N \times M$ Elemente enthält, und den Basisfunktionen B_{j,k_u} und B_{j,k_v}, welche auf den Knoten Vektoren τ_u und τ_v definiert sind mit

$$\tau_u = \{u_j\}_{j=0}^{N-1+k_u} \tag{2}$$

$$\tau_v = \{v_i\}_{i=0}^{M-1+k_v} \tag{3}$$

wobei k_u beziehungsweise k_v die Ordnung der Basisfunktionen sind. Die Ordnung der Basisfunktionen ist dabei um eins größer als der Grad der Basisfunktionen.

Ein Strahl

$$\vec{r} = \vec{b} + t * \vec{d} = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} + t * \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} \tag{4}$$

wird, entsprechend dem Ansatz von Kajiya [KAJ82], durch den Schnitt zweier Ebenen repräsentiert

$$E_1 = \left\{ \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \mid \begin{pmatrix} n_{1x} \\ n_{1y} \\ n_{1z} \\ d_1 \end{pmatrix} \circ \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = 0 \right\} = \left\{ p \mid e_1 \circ (p,1)^T = 0 \right\} e_1 = \begin{pmatrix} n_{1x} \\ n_{1y} \\ n_{1z} \\ d_1 \end{pmatrix} \tag{5}$$

$$E_2 = \left\{ \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \mid \begin{pmatrix} n_{2x} \\ n_{2x} \\ n_{2x} \\ d_2 \end{pmatrix} \circ \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = 0 \right\} = \left\{ p \mid e_2 \circ (p,1)^T = 0 \right\} e_2 = \begin{pmatrix} n_{2x} \\ n_{2y} \\ n_{2z} \\ d_2 \end{pmatrix} \tag{6}$$

beziehungsweise in der Hess'schen Normalenform

$$E_1 : p_x \cdot n_{1x} + p_y \cdot n_{1y} + p_z \cdot n_{1z} - d_1 = 0$$

$$E_2 : p_x \cdot n_{2x} + p_y \cdot n_{2y} + p_z \cdot n_{2z} - d_2 = 0$$

Die Normalen der Ebenen werden dabei wie folgt definiert

$$N_1 = \begin{pmatrix} n_{1x} \\ n_{1y} \\ n_{1z} \end{pmatrix} = \left\{ \begin{array}{l} (d_y \quad -d_x \quad 0)^T, \text{falls} \left|d_x\right| > \left|d_y\right| \text{und} \left|d_x\right| > \left|d_z\right| \\ (0 \quad d_z \quad -d_y)^T, \text{ansonsten} \end{array} \right\} \tag{7}$$

$$N_2 = \begin{pmatrix} n_{2x} \\ n_{2y} \\ n_{2z} \end{pmatrix} = N_1 \times \vec{d} \tag{8}$$

wodurch gewährleistet ist, das N_1 und N_2 orthogonal zur Strahlrichtung sind. Die Ebenenparameter d_1 und d_2 können durch das Einsetzen des Strahlursprungs in die Ebenengleichung ermittelt werden.

Ein Schnittpunkt zwischen einem Strahl und der NURBS-Oberfläche muss dann die Bedingung erfüllen

$$\begin{pmatrix} n_{1x} \\ n_{1y} \\ n_{1z} \\ d_1 \end{pmatrix} \circ (S(u,v),1) = 0 \qquad \begin{pmatrix} n_{2x} \\ n_{2y} \\ n_{2z} \\ d_2 \end{pmatrix} \circ (S(u,v),1) = 0 \qquad (9)$$

Der Schnittpunkt kann durch numerische Nullstellenfindungsverfahren bestimmt werden. Entsprechend dem Vorschlag von Martin et. al. wurde dabei das *Newton'sche Iterationsverfahren*, kurz *Newton Iteration*, verwendet. Bei diesem wird, insbesondere für eine schnelle Konvergenz, ein guter Startpunkt für das Verfahren benötigt.
Der Startpunkt für die Newton Iteration $(u_0, v_0)^T$ wird mit Hilfe von *Bounding Boxes* bestimmt, welche in einem Vorberechnungsschritt, dem *Flattening*, erzeugt werden.

4.2.3.3. Flattening

Das Flattening dient der Erzeugung der *Bounding Boxes*, so dass durch den Startpunkt, gewährleistet ist, dass die Newton Iteration sicher und schnell konvergiert.
Beim Flattening erfolgt eine Verfeinerung des Kontrollpunktgitters. Diese ist notwendig, da, wie im Folgenden noch näher erläutert wird, anhand des Kontrollpunktgitters die *Bounding Boxes* erstellt werden. Der notwendige Grad der Verfeinerung wird anhand eines *flatness criteria* bestimmt.

Zur Vereinfachung soll zunächst nur eine NURBS-Kurve betrachtet werden. Die Ergebnisse werden anschließend auf NURBS-Oberflächen übertragen.

Es sei *C(t)* eine NURBS-Kurve mit dem Knotenvektor τ_t mit

$$\tau_t = \{t_j\}_{j=0}^{N-1+k_t} \qquad (10)$$

wobei k_t die Ordnung der NURBS-Kurve ist. Für die Verfeinerung eines Intervalls $[t_i, t_{i+1})$ muss sowohl die Krümmung des zugehörigen Kurvensegments als auch dessen Bogenlänge berücksichtige werden.

- Die Krümmung des Segments gewährleistet, dass die Newton Iteration sicher konvergiert, dass heißt, dass nicht mehrere Nullstellen in der Umgebung eines Startwerts $(u_0, v_0)^T$ liegen.

- Die Bogenlänge stellt sicher, dass die Newton Iteration schnell konvergiert; der Startwert $(u_0, v_0)^T$ für die Iteration also nahe genug an der Nullstelle liegt.

Damit ergibt sich für die Verfeinerung des Knotenvektors, aus welcher die zusätzlichen Kontrollpunkte bestimmt werden können

$$n = C * \max_{[ti,ti+1)} \{curvature(c(t))\} * arclen(c(t))_{[t_i,t_{i+1})} \qquad (11)$$

wobei *n* die Anzahl der Knoten ist, welche in den Knotenvektor einzufügen sind. Der Parameter *C* kann durch den Nutzer festgelegt werden.
Allerdings sind weder die Krümmung noch die Bogenlänge direkt ermittelbar. Martin et. al. verwenden deshalb eine Approximation für *n*, welche für diese Arbeit übernommen wurde. Für eine detaillierte Erläuterung sei an dieser Stelle auf [MAR00] verwiesen.

Im dreidimensionalen Fall, für eine NURBS-Oberfläche, wird für jedes Intervall des *u*-Knotenvektors in jeder Spalte des Gitters die Anzahl der einzufügenden Knoten ermittelt. Das Maximum über alle Spalten in jedem Intervall $[u_i, u_{i+1})$ ist die Anzahl der einzufügenden Knoten in $[u_i, u_{i+1})$. Beim Einfügen werden die neuen Knoten gleichmäßig über das Intervall $[u_i, u_{i+1})$ verteilt. Das analoge Verfahren wird für die Verfeinerung des *v*-Knotenvektors durchgeführt.
Durch die Verfeinerung erhält man zunächst die verfeinerte Knotenvektoren k_u* und k_v*. Diese werden anschließend so verändert, dass jedes nicht leere Intervall $[u_i, u_{i+1}) \times [v_i, v_{i+1})$ aus $\tau_u \times \tau_v$ einer Bézier-Oberfläche entspricht. Anhand der verfeinerten und modifizierten Knotenvektoren kann das neue Kontrollpunktgitter C_I* berechnet werden[15].
Die Umwandlung der NURBS-Oberfläche in ein Bézier-Patch kann durch das Duplizieren der Element von τ_u beziehungsweise τ_v erreicht werden. Jeder innere Knoten muss anschließend *(k_u -1)*- beziehungsweise *(k_v -1)*-mal im Knotenvektor enthalten sein. Die Anzahl der äußeren Knoten von τ_u und τ_v muss der Ordnung in der

[15] Ein Verfahren dazu findet sich zum Beispiel in [PIE97] , eine Implementation in [NUR04]

Basisfunktion in der jeweiligen Parameterdimension entsprechen.
Durch die Erzeugung der Bézier-Oberflächen $W_{a,b}$ für jedes nicht leere Intervall $I_{a,b} = [u_a, u_{a+1}) \times [v_b, v_{b+1})$ ist gewährleistet, dass die Kontrollpunkte $C_{I_{a,b}}$ eines Intervalls die konvexe Hülle der Bézier-Oberfläche $W_{a,b}$ bilden. Als Bounding Box von $I_{a,b}$ kann die Bounding Box von $C_{I_{a,b}}$ verwendet werden. Der durch eine Bounding Box $B_{a,b}$ gegebene Startpunkt für die Newton Iteration ist damit

$$\begin{pmatrix} u_0 \\ v_0 \end{pmatrix} = \left(\frac{u_{a+1} - u_a}{2}, \frac{v_{b+1} - v_b}{2} \right)^T \tag{12}$$

Der Parameter *C* des *flatness criteria* aus Formel 1 ermöglicht somit, wie die Formel für die Anzahl der in den Knotenvektor einzufügenden Knoten *n* zeigt, die Anzahl der entstehenden *Bounding Boxes*, und damit die Güte des Startwertes der Newton Iteration zu bestimmen.

4.2.3.4. Ray-Casting

Der Ray-Casting-Prozess beginnt mit der Projektion der *Bounding Boxes*. Dadurch werden die für einen Objekt-Strahl-Schnitt in Frage kommenden Fragmente beziehungsweise Augenstrahlen erzeugt und für jeden der Strahlen der Startpunkt für die Newton Iteration bestimmt.
Die wichtigsten Schritte der Newton Iteration zur Bestimmung des Schnittpunktes sollen im Folgenden erläutert werden.

Das Ziel der Newton Iteration ist die Bestimmung der Nullstelle der Zielgleichung

$$F(u,v) = \begin{pmatrix} N_1 \circ S(u,v) + d_1 \\ N_2 \circ S(u,v) + d_2 \end{pmatrix} \tag{13}$$

Ein Iterationsschritt hat dabei die Form

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - J^{-1}(u_n, v_n) * F(u_n, v_n) \tag{14}$$

wobei $J^{-1}(u_n, v_n)$ die Inverse der Jacobi-Matrix $J = (\vec{f}_{u_n}, \vec{f}_{v_n})$ ist mit

$$\vec{f}_{u_n} = \begin{pmatrix} N_1 \circ S_{u_n}(u_n, v_n) \\ N_2 \circ S_{u_n}(u_n, v_n) \end{pmatrix} \text{mit } S_{u_n} = \frac{\partial S(u_n, v_n)}{\partial u} \tag{15}$$

$$\vec{f}_{v_n} = \begin{pmatrix} N_1 \circ S_{v_n}(u_n, v_n) \\ N_2 \circ S_{v_n}(u_n, v_n) \end{pmatrix} \text{mit } S_{v_n} = \frac{\partial S(u_n, v_n)}{\partial v} \tag{16}$$

Die Inverse der Jacobi-Matrix erhält man aus

$$J^{-1} = \frac{1}{\det(J)} * adj(J) \tag{17}$$

wobei $adj(J)$ die Adjunkte der Jacobi-Matrix ist.

Die Inverse der Jacobi-Matrix kann allerdings nur bestimmt werden, wenn die Jacobi-Matrix selbst nicht singulär ist. Dies kann durch

$$\det(J) < \varepsilon_J \tag{18}$$

überprüft werden. Ist die Jacobi-Matrix singulär, so erfolgt anstatt eines Iterationsschrittes eine Verschiebung des Punktes im *uv*-Raum, um den Problembereich zu verlassen. Die geschieht durch

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - a * \begin{pmatrix} rand * (u_0 - u_n) \\ rand * (v_0 - v_n) \end{pmatrix} \tag{19}$$

wobei *rand* ein zufällig erzeugter Wert aus [0,1] ist.

Eine wichtiger Teil der Newton Iteration sind die Abbruchkriterien.

Eine Nullstelle wird dabei als gefunden betrachtet, dass heißt die Iteration erfolgreich abgebrochen, wenn die Distanz zur Nullstelle unter einen definierten Schwellwert fällt

$$\left|F(u_n, v_n)\right| = \left\| \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} \right| < \varepsilon_A \qquad (20)$$

Ist das Ergebnis nach einem Iterationsschritt $n+1$ jedoch weiter von der Nullstelle entfernt als nach dem Schritt n, tritt also keine Konvergenz auf, so wird die Newton Iteration erfolglos abgebrochen. Die Konvergenz kann mit

$$\left|F(u_{n+1}, v_{n+1})\right| > \left|F(u_n, v_n)\right| \qquad (21)$$

überprüft werden.

Für die Anzahl der Iterationsschritte des Newton Verfahrens wird eine obere Grenze definiert. Wird diese erreicht, ohne dass die Nullstelle gefunden wurde, so wird angenommen, dass kein Strahl-Objekt Schnittpunkt existiert und die Newton Iteration ebenfalls erfolglos abgebrochen. Die Wahl einer geeigneten Anzahl von Iterationsschritten hängt dabei von der NURBS-Oberfläche und der Blick- beziehungsweise Strahlrichtung ab.

4.2.4.4. Evaluierung der NURBS-Oberfläche

Der rechenintensivste Teil der Newton Iteration zur Bestimmung des Strahl-Schnittpunktes ist die in jedem Schritt notwendige Evaluierung der NURBS-Oberfläche für $(u_n, v_n)^T$, dass heißt die Bestimmung des Oberflächen-Punktes für diesen Parameterwert. Neben $S(u_n, v_n)$ müssen dabei auch die partiellen Ableitungen $S_u(u_n, v_n)$ und $S_v(u_n, v_n)$ bestimmt werden.

Die Idee des verwendeten Verfahrens soll zunächst wieder für NURBS-Kurven erläutert werden (Abb. 56)

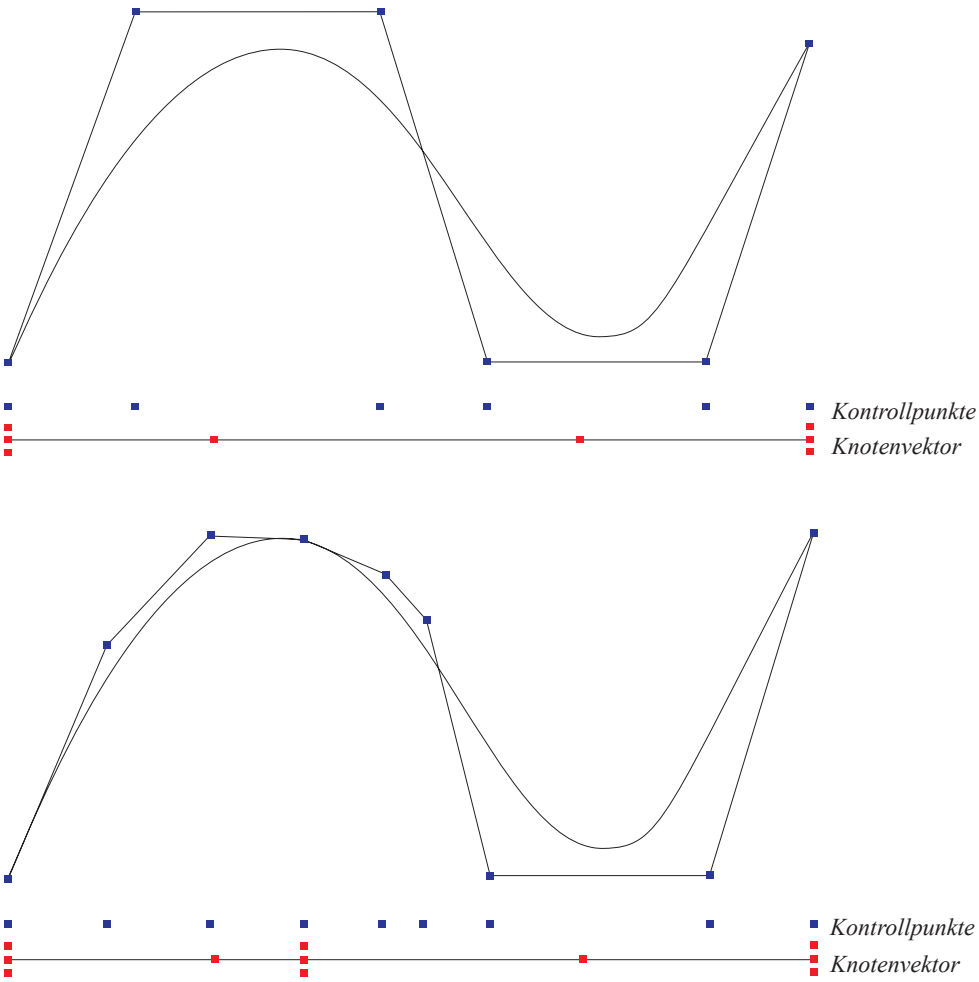


Abbildung 56: Evaluierung eines Kurvenpunktes durch Verfeinerung des Knotenvektors

Es sei Q eine NURBS-Kurve der Ordnung k mit dem Grad *(k-1)*, C das Kontrollpunkt-Polygon der Kurve mit N_C Punkten und τ_t der Knotenvektor. Die Anzahl der Knoten ist dabei durch den Grad der Basisfunktion und die Anzahl der Kontrollpunkte N_k mit

$$N_{k_u} = (N_C - 1) + (k - 1) \qquad (22)$$

festgelegt.

Zur Evaluation von Q an der Stelle u' mit u' aus $[u_i, u_{i+1})$ wird eine neuer, verfeinerter Knotenvektor k_u' und ein neues Kontrollpunkt-Polygon C' erzeugt. Der Knotenvektor k_u' wird aus k_u bestimmt, indem zu den in k_u vorhandenen Knoten, noch so viele mit dem Wert u′ hinzugefügt werden, dass mindestens *(k-1)* Knoten mit dem

Wert u' in k_u' vorhanden sind. Nach der Bestimmung von C' ist der Wert der NURBS-Kurve in u′ direkt durch den Kontrollpunkt $c'_{\mu-k+1}$ aus C' gegeben. Analog dazu können auch die partiellen Ableitungen $S_u(u_n, v_n)$ und $S_v(u_n, v_n)$anhand von k_u' mit Hilfe des Differenzenquotienten bestimmt werden.

Für die Evaluierung einer NURBS-Oberfläche in

$$(u', v') \in [u_{\mu_u}, u_{\mu_u+1}] \times [v_{\mu_v}, v_{\mu_v+1}] \tag{23}$$

ergibt sich damit

$$S(u', v') = C'_{\mu_v-k_v+1,\mu_u-k_u+1} \tag{24}$$

$$S_u(u', v') = \frac{(k_u-1)\cdot\omega_{\mu_v-k_v+1,\mu_u-k_u+2}}{(u_{\mu_u}+1-u')\,\omega_{\mu_v-k_v+1,\mu_u-k_u+1}} \cdot \left[C'_{\mu_v-k_v+1,\mu_u-k_u+2} - C'_{\mu_v-k_v+1,\mu_u-k_u+1}\right] \tag{25}$$

$$S_v(u', v') = \frac{(k_v-1)\cdot\omega_{\mu_v-k_v+2,\mu_u-k_u+1}}{(v_{\mu_v}+1-v')\,\omega_{\mu_v-k_v+1,\mu_u-k_u+1}} \cdot \left[C'_{\mu_v-k_v+2,\mu_u-k_u+1} - C'_{\mu_v-k_v+1,\mu_u-k_u+1}\right] \tag{26}$$

wobei ω das Gitter ist, welches durch die beiden Knotenvektoren aufgespannt wird.

Zur Bestimmung von $S(u_n, v_n)$, $S_u(u_n, v_n)$ und $S_v(u_n, v_n)$ ist somit die Erzeugung des verfeinerten Kontrollpunktgitters C' notwendig. Dieses muss aber nicht vollständig geschehen, da lediglich $C'_{\mu_v-k_v+1,\mu_u-k_u+2}$, $C'_{\mu_v-k_v+2,\mu_u-k_u+1}$ und $C'_{\mu_v-k_v+1,\mu_u-k_u+1}$ aus C' benötigt werden. Eine solche teilweise beziehungsweise lokale Verfeinerung des ursprünglichen Kontrollpunktgitters C, das *Partial Refinement*, ist dabei wesentlich effizienter zu berechnen als die vollständige Erzeugung von C'. Eine mögliche Vorgehensweise für ein Partial Refinement findet sich in [MAR00] oder auch in [PIE97].

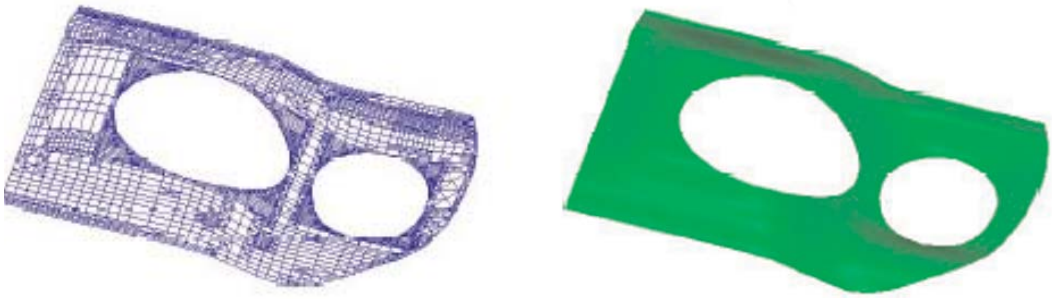


Abbildung 58: Trimmed NURBS-Oberfläche, aus [MAY04]

4.2.3.5. Trimmed NURBS

Als *Trimming* von NURBS-Oberflächen wird die Begrenzung des *uv*-Parameterraums bezeichnet, dessen Punkte auf der NURBS-Oberfläche dargestellt werden. Dadurch wird, wie Abbildung 58 zeigt, eine größere Flexibilität bei der Definition von Flächen im Raum erreicht.
Für die Begrenzung werden im *uv*-Parameterraum der NURBS-Oberfläche definiert Kurven verwendet. Das Trimming erfolgt nach dem Objekt-Strahl Schnitttest. Für jeden gefunden Schnittpunkt wird die Lage bezüglich der Kurve ermittelt, wobei die Orientierung einer Kurve bestimmt, ob ein Schnittpunkt verworfen wird oder nicht.

4.2.3.6. Implementierung

Vor der Implementierung auf der Grafikkarte wurde das Ray-Casting-Verfahrens für NURBS-Oberflächen auf der CPU implementiert. Dies erschien sinnvoll, um zunächst die effiziente Umsetzbarkeit auf der GPU zu untersuchen. Dabei zeigte sich, dass eine Abbildung auf die GPU sowie eine direkte Integration in das unter 4.2.2 vorgestellt Ray-Casting-System möglich war.

Für die Hüllkörperprojektion konnten die bereits implementierten *Bounding Boxes* verwendet werden. Lediglich die Möglichkeit den Startpunkt für die Newton Iteration pro Bounding Box zur Verfügung zu stellen musste ergänzt werden. Die Objekt-ID der NURBS-Oberfläche sowie der Startpunkt des Iterationsverfahrens werden bei der Projektion mit Hilfe von *Varying Variables* in den Fragment-Prozessor transportiert.
Die für die Durchführung des Schnitttests im Fragment-Shader notwendigen

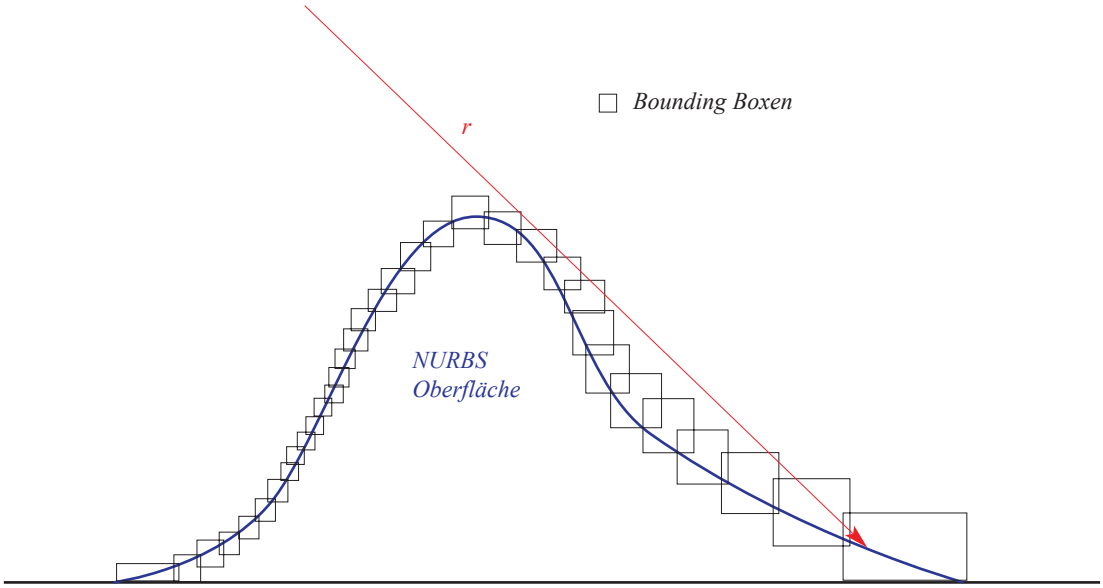


Abbildung 59: Ursache für fehlerhafte Ergebnisse bei der Durchführung der Newton Iteration in mehreren Rendering Passes

Informationen, das Kontrollpunktgitter und der Knotenvektor, werden anhand der Objekt-ID aus einer Textur ausgelesen. Die Abbildung des Algorithmus zur Evaluierung eines Punktes auf der NURBS-Oberfläche konnte, auf Grund der C-ähnlichen Syntax von GLSL, in starker Anlehnung an die CPU-Implementierung erfolgen.

Als schwieriger erwies sich die Umsetzung der Newton Iteration auf der GPU, da das Kompilieren des Shader-Programms zu Programmabstürzen durch den OpenGL Treiber führte. Versuche zeigten jedoch, dass die Ausnahmefehler bei einer Reduzierung der Komplexität des Programms, der Beschränkung der Newton Iteration auf den ersten Schritt, nicht auftraten. Dies eröffnete die Perspektive, die Newton Iteration bei einer Aufteilung auf mehrere *Rendering Passes* vollständig ausführen zu können.

Eine solche Aufteilung impliziert jedoch teilweise inkorrekten Ergebnissen (Abb. 60). Ein Fehler entsteht, da bereits nach dem initialen Iterationsschritt entschieden werden muss, in welcher durch den Strahl geschnittenen Bounding Box der dem Strahlursprung am nächsten liegende Schnittpunkt gefunden wird (Abb 59). Nach dem initialen Schritt der Newton Iteration steht für diese Entscheidung nur der

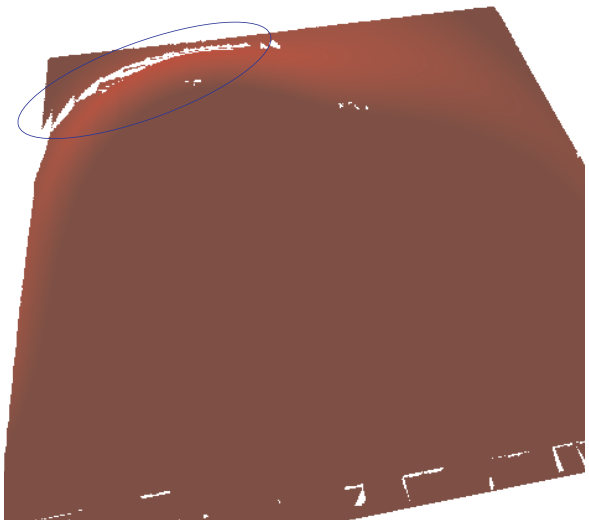


Abbildung 60: Fehler (blau eingekreist) in der NURBS Oberfläche durch die Aufteilung auf mehrere Rendering Passes

Oberflächenpunkt zur Verfügung, welcher mit dem Startwert $(u_0, v_0)^T$ ermittelt wurde. Eine Entscheidung zu diesem erlaubt somit keine Aussage über die Konvergenz des Iterationsverfahrens.

Trotz dieses, der Implementierung inhärenten Problems, wurde der Ansatz der Durchführung der Newton Iteration in mehreren *Rendering Pass*es weiterfolgt, da dies zu diesem Zeitpunkt, im übrigen bis heute, die einzige Möglichkeit war, das Verfahren näher zu untersuchen.
Dies erschien sinnvoll, da eventuell zu einem späteren Zeitpunkt die Implementierung des ursprünglichen Verfahrens, mit der Durchführung der vollständigen Newton Iteration in einem *Rendering Pass*, möglich sein wird. Auf Grund der nicht vorhandenen Fehlermeldungen kann die Ursache der Fehler im Moment nicht abschließend geklärt werden.

Der Ablauf der Newton Iteration in mehreren *Rendering Pass*es ist in Abbildung 61 schematisch dargestellt. In jedem *Rendering Pass* wird ein Iterationsschritt durchgeführt und die berechneten Ergebnisse werden in einen *PBuffer* geschrieben, so dass sie nach dem Kopieren in eine Textur im nächsten Pass wieder zur Verfügung stehen. Durch das Beenden eines Iterationsschrittes direkt nach der Berechnung von

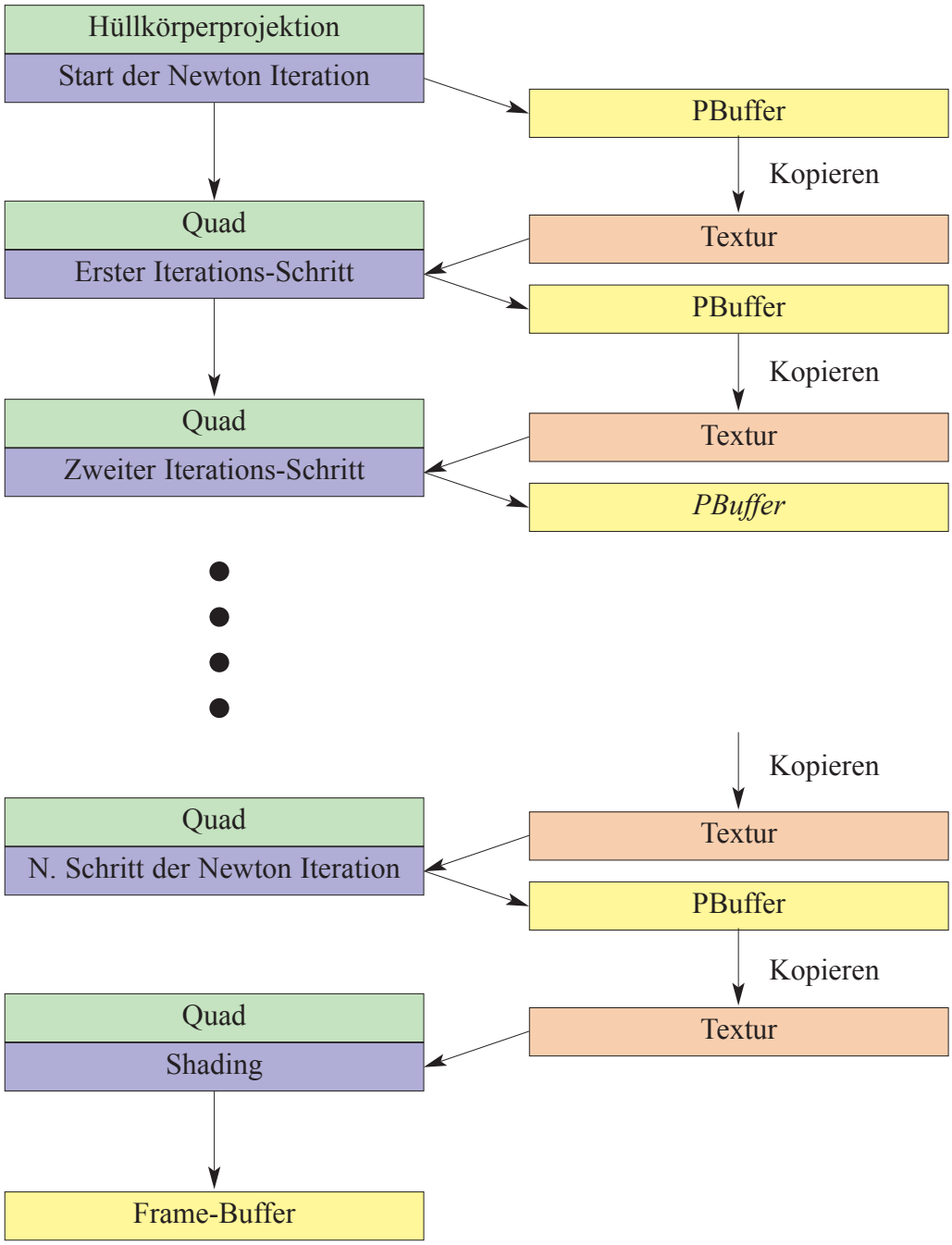


Abbildung 61: Ablauf des Multi-Pass-Renderings beim NURBS-Raycaster

$(u_{n+1}, v_{n+1})^T$ ist ein *PBuffer* zum Speichern aller Informationen zwischen den verschiedenen *Rendering Pass*es ausreichend.

Die Berechnung eines Schritts der Newton Iteration erfolgt dabei nur für die Fragmente beziehungsweise die Augenstrahlen, für welche die Newton Iteration noch nicht abgebrochen wurde. Dies kann anhand der aus der Textur ausgelesenen Informationen entschieden werden. Diese Informationen werden auch zur Ermittlung der Fragmente genutzt, für welche gar keine Augenstrahlen durch die Hüllkörperprojektion erzeugt wurde. Ein solcher Test ist notwendig, da nach der Hüllkörperprojektion in jedem weiteren *Rendering Pass* ein Quad projiziert wird.

Eine Geschwindigkeitssteigerung beim *Multi-Pass-Rendering* kann durch die Verwendung eines *PBuffer*s für alle Passes erreicht werden. Dies ermöglicht, dass nach dem Abbruch der Newton Iteration keine weiteren Daten für das entsprechende Fragment geschrieben werden müssen.

Im letzten *Rendering Pass* erfolgt kein weiterer Schritt der Newton Iteration, sondern die Berechnung der Oberflächennormale sowie das Shading für die Darstellung.

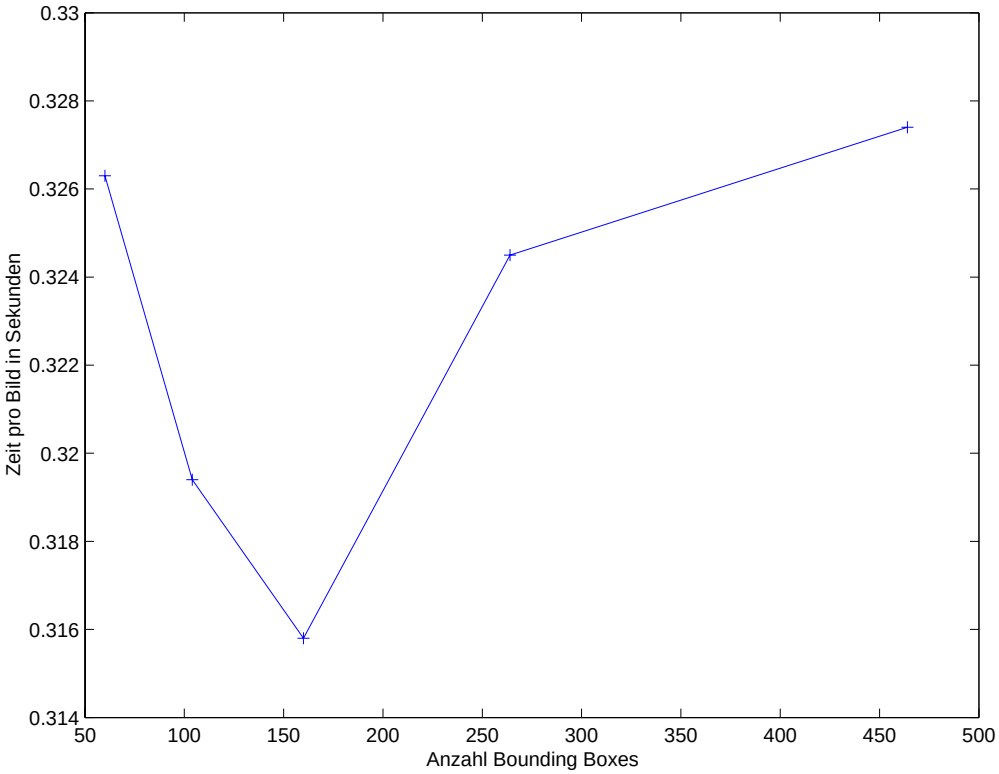


Abbildung 62: Rendering der NURBS-Oberfläche in Abhängigkeit von der Anzahl der verwendeten Bounding Boxes (Detailansicht zu Abbildung 63)

4.2.3.7. Ergebnisse[16]

Geschwindigkeit des NURBS-Raycasters in Abhängigkeit von der Anzahl der verwendeten Bounding Boxes

Bei diesem Versuch, dessen Ergebnisse in Abbildung 62 und Abbildung 63 dargestellt sind, wurde der Einfluss des Verfeinerungsfaktors *C* beziehungsweise der Anzahl der Bounding Boxes auf die Geschwindigkeit beim Rendern der NURBS-Oberfläche untersucht. Der Blickpunkt wurde für bei dem Versuch nicht verändert. Es wurden 4+1 Rendering Passes ausgeführt, dass heißt es erfolgten vier Iterationsschritte der Newton Iteration.

[16] Die verwendete Testumgebung ist in Anhang B spezifiziert

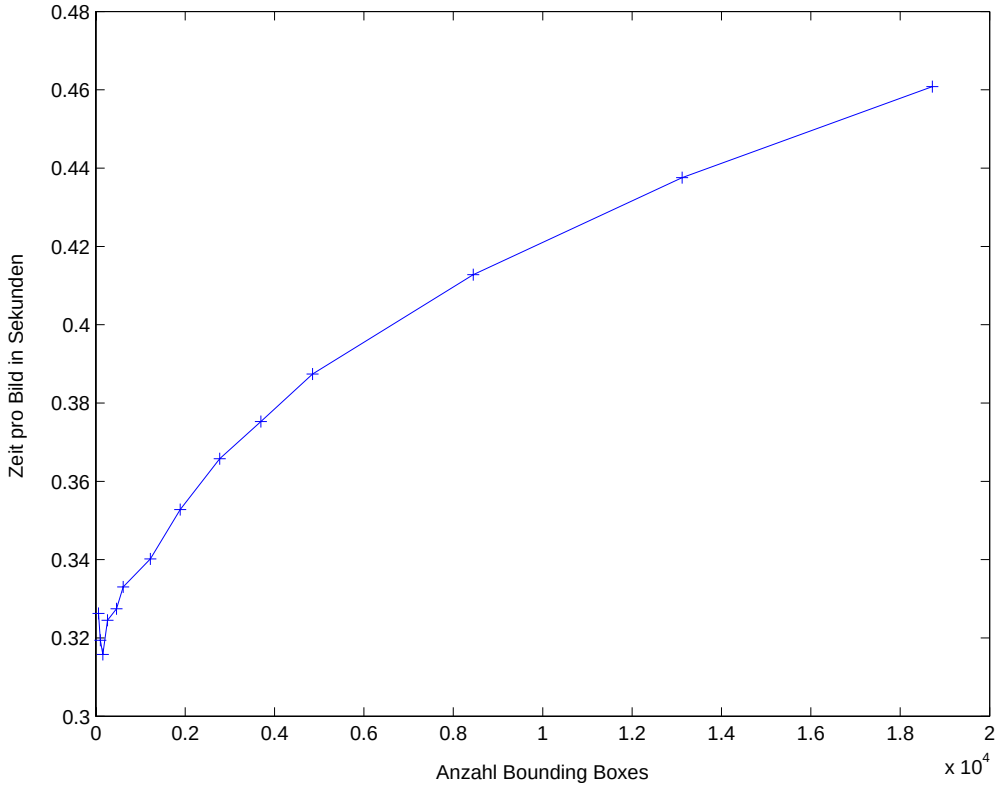


Abbildung 63: Rendering der NURBS-Oberfläche in Abhängigkeit von der Anzahl der verwendeten Bounding Boxes

Wie Abbildung 62 zeigt, existiert für die Anzahl der Bounding Boxes ein Optimum. Dieses liegt für die gerenderte NURBS-Oberfläche und für den gewählten Blickpunkt bei 160 *Bounding Boxes*. Bei einer geringeren Anzahl ist der Startwert der Newton Iteration nicht nahe genug an der Nullstelle und es müssen mehr Iterationsschritte ausgeführt werden. Darüber hinaus entstehen in diesem Fall Artefakte. Werden mehr als circa 50 *Bounding Boxes* gerendert[17], so nehmen die Überlappungen dieser zu und der erste Schritt der Newton Iteration muss für mehr Fragmente durchgeführt werden. Dies führt zu der in Abbildung 63 erkennbaren Zunahme der für das Rendern benötigte Zeit mit der Anzahl der *Bounding Boxes*.

[17] Der konkrete Wert hängt von der verwendeten NURBS-Obefläche ab.

4.2.4.8. Diskussion

Auf Grund des im Moment nicht ausgereiften GLSL-Compilers war keine korrekte Implementierung des Ray-Casting-Verfahrens für NURBS-Oberflächen möglich. Die deshalb entwickelte Alternative der Durchführung des Ray-Casting in einem *Multi Pass Rendering* Verfahren wurde auf der Grafikkarte implementiert. Dabei traten, neben den durch die Aufteilung auf mehrere *Rendering Pass*es zu erwartenden Bildfehlern, weiter Artefakte auf (Abb. 47). Bei näherer Analyse zeigte sich, dass diese durch eine fehlerhafte Übersetzung der GLSL-Programme in Assembler-Code entstehen. Dabei wird der Wert von Registern überschrieben, obwohl dieser noch zu einem späteren Zeitpunkt im Programmablauf benötigt wird. Die Ursache dafür ist wahrscheinlich eine zu aggressive Optimierung durch den GLSL-Compiler. Die Artefakte konnten durch das Einfügen zusätzlicher Operationen reduziert werden. Dabei werden die zunächst überschriebenen Werte direkt, beziehungsweise als Teil einer einfachen arithmetischen Operation, einem Ergebniswert zugewiesen, welcher in das *Render Target* geschriebenen wird.

Trotz dieser Probleme ist eine erste Bewertung der GPU-Implementierung möglich. Dabei ist insbesondere der auf der GPU erzielte Geschwindigkeitsvorteil gegenüber der CPU-Referenzimplementierung mit einem Faktor von mindestens 100 sehr viel versprechend und lässt eine weitere Arbeit an dem Verfahren sinnvoll erscheinen. Bei der Abbildung auf die GPU zeigte sich, dass die mit der CPU-Implementierung ermittelten Parameter für die Verfeinerung sowie die Iterationsschritte auch auf der GPU verwendet werden können (Abb. 65). Die Anzahl der *Bounding Boxes* hat auf der GPU einen geringeren Einfluss auf die Geschwindigkeit, da deren Projektion durch die Grafikkarte beschleunigt wird.

Um die aktuellen Beschränkungen durch den GLSL-Compiler zu überwinden, kann alternativ die Verwendung von Cg überprüft werden. Cg bietet den Vorteil, dass anstatt von High-Level- auch vorkompilierter Assembler-Code verwendet werden kann. Dies ermöglicht eine Optimierung beziehungsweise Korrektur des durch den Compiler erzeugten Codes, wodurch zum Beispiel sichergestellt werden kann, dass kein benötigter Wert vorzeitig überschrieben wird.

Abbildung 64: Artefakte beim NURBS-Ray-Casting

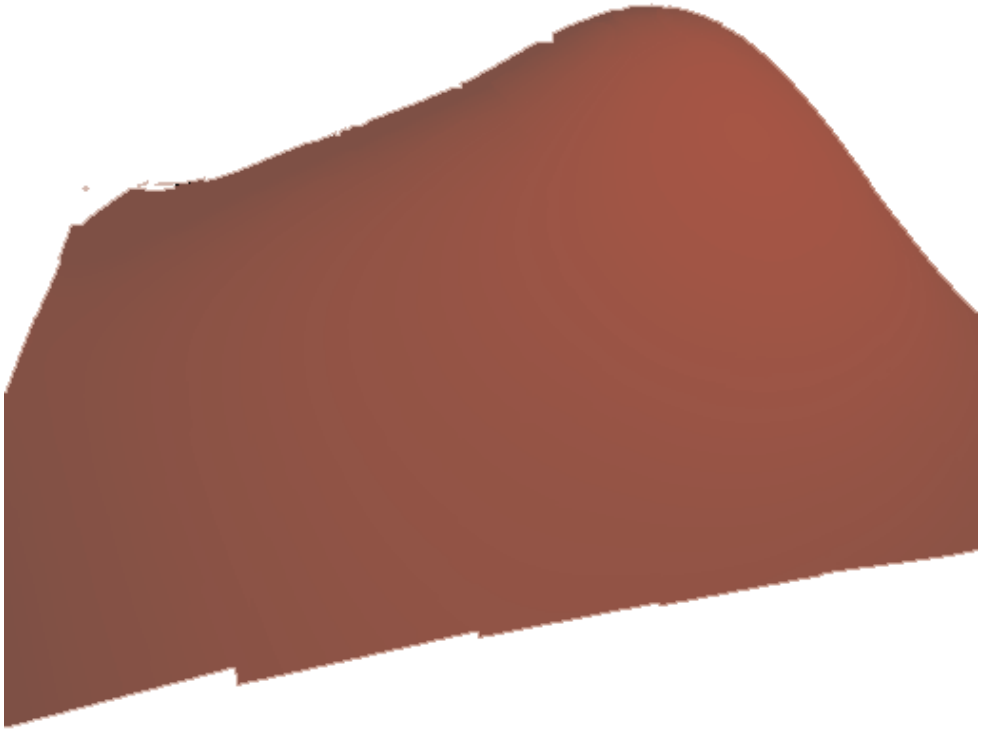


Abbildung 65: NURBS-Ray-Casting. Die dargestellte NURBS-Oberfläche kann mit 2,8 Bildern pro Sekunde gerendert werden

Kapitel 5
Zusammenfassung und Ausblick

Das implementierte, sowohl unter Linux als auch unter Windows lauffähige Ray-Casting-System demonstriert, dass interaktives Rendering auch bei der Verwendung von nicht-triangulierten Objekten und pixelgenauer Berechnung der Geometrie- und Shading-Informationen möglich ist. Durch die speichereffiziente Repräsentation der Primitive in parametrischer Form können dabei wesentlich größere Szenen direkt auf der Grafikkarte gespeichert werden, was zu einem schnelleren Rendering beiträgt. Zum Beispiel sind 150 MB Grafikkartenspeicher für das Speichern von 4,3 Millionen Kugeln ausreichend. Wesentlich für interaktive Bildwiederholraten ist jedoch der gewählte Beschleunigungsansatz, die Projektion der Hüllkörper, welcher eine hardwarebeschleunigte und dadurch effiziente Reduzierung der Schnittoperationen ermöglicht. Die Durchführung der Strahlanfrage auf der Grafikkarte lässt eine deutliche Leistungssteigerung des Ray-Casting-Systems in den kommenden Jahren erwarten. Dies gilt insbesondere für Primitive für die eine analytische Bestimmung des Schnittpunktes möglich ist. Eine Geschwindigkeitssteigerung bereits heute ist durch die Verwendung mehrerer Fragment-Einheiten möglich, zum Beispiel mit der *SLI* Technologie von Nvidia [NVI04b] oder der *VPU/VSU* Technologie von 3DLabs [3DL04]. Auf Grund der Möglichkeit große Szenen, insbesondere mit Primitiven,

direkt darzustellen, sowie der pixelgenauen Berechnung der Bildschirminformationen bieten sich zum Beispiel CAD- und Design-Applikationen für eine Integration des Ray-Casting-System an. Dabei kann diese Arbeit aber nur als erster Schritt verstanden werden. Eine Weiterentwicklung des Systems beinhaltet:

- die Untersuchung des Ray-Casting-Systems bei realistischen Szenen und die Entwicklung von Heuristiken für eine effiziente Parametrisierung des Systems
- die Erweiterung des Ray-Casting-Systems für das Rendern dynamischer Szenen
- die Untersuchung weiterer Hüllkörper wie *Discrete Orientated Polytopes* (k'dops) oder *slabs* [KAY86] untersucht werden.
- die Optimierung der implementierten Schnittgleichungen. Eine Möglichkeit ist die Verwendung weiterer, spezialisierter Schnittgleichungen für Quadrics. Neben der zur Zeit verwendeten Gleichung für Kugeln existieren solche für Zylinder, Konus, Paraboloid, Hyperbolid und Torus. Allerdings sind dabei die Limitierungen in der Fragment-Einheit, insbesondere die SIMD-Architektur zu berücksichtigen, wodurch bei Szenen mit zahlreichen unterschiedlichen Objekten möglicherweise die Verwendung der allgemeinen Schnittgleichung effizienter ist.
- die Integration weiterer Primitive, wie zum Beispiel *Superquadrics* [HAN89]
- die Integration eines Raycasters für Volumendaten. Dieser kann auf der NV4X-Architektur in einem zusätzlichen *Rendering Pass* implementiert werden.
- die Reduzierung der linearen Komplexität für die Anzahl der Objekte sein. Dies kann durch die Verwendung von Hüllkörper-Hierarchien erreicht werden. Die sichtbaren Teile der Szene beziehungsweise die Knoten der Hierarchie können dabei mit Hilfe von *Occlusion Queries* bestimmt werden.

Eine Weiterentwicklung des Ray-Casting-System für NURBS-Oberflächen erscheint insbesondere auf Grund der deutlich höheren Darstellungsgeschwindigkeit auf der GPU, im Vergleich zur CPU, sinnvoll. Vor einer Fortsetzung der Arbeit an diesem System müssen aber zunächst die erläuterten Probleme mit dem GLSL-Compiler gelöst werden. Diese ließen es zur Zeit nicht sinnvoll erscheinen, das *Trimming* von

NURBS-Oberflächen auf der GPU zu implementieren. In zukünftigen Systemen ist dies in einem zusätzlichen *Rendering Pass* vor dem *Shading* möglich. Ebenfalls zukünftigen Arbeiten muss eine Optimierung der *Bounding Boxes* vorbehalten bleiben. Anstatt eines Startwertes pro *Bounding Box* kann für jede Ecke der *Bounding Box* der zugehörige *uv*-Wert gespeichert werden. Durch die Verwendung von *Varying Variables* erfolgt eine Interpolation der *uv*-Werte beim Transport aus der Vertex- in die Fragment-Einheit. Der Startwert für die Newton Iteration ist dadurch für jedes Fragment beziehungsweise jeden Augenstrahl günstiger, als der bisher verwendete. Dies erhöht die Konvergenz der Newton Iteration, so dass weniger Schritte für das Rendern einer NURBS-Oberfläche notwendig sind.

Das gesetzte Ziel der Arbeit, ein interaktives Ray-Tracing-System, konnte jedoch (noch) nicht erreicht werden. Auf Grund der zu geringen Datentransferraten war die Umsetzung der in der Arbeit vorgeschlagenen Implementierung nicht möglich.
Für das Ray-Casting-System stehen aber Alternativen zur Berechnung von globalen Beleuchtungseffekten zur Verfügung. Verfahren wie *Shadow Mapping* [WIL78] und *Reflection Mapping* [BLI76] können auch innerhalb des Ray-Casting-Systems eingesetzt werden. Die Berechnungen können dabei auf der Grafikkarte effizient durchgeführt werden, da diese Techniken auch beim *Feed Forward Rendering* verwendet werden [NVI04c]. Eine weitere Möglichkeit für die Berechnung von globalen Beleuchtungseffekten ist die Kombination des GPU-basierten Ray-Casting-Systems mit einem CPU-basierten Raytracer. Bei einem solchen System werden nach der Durchführung der Strahlanfragen für die Augenstrahlen auf der GPU alle weiteren Strahlgenerationen vollständig vom CPU-basierten Raytracer behandelt. Der notwendige Daten-Readback für jedes Bild ist dadurch geringer als bei dem in Kapitel 3 der Arbeit vorgeschlagenen Ray-Tracing System. Die Leistung des Gesamtsystems hängt damit weniger von den Datentransferraten ab. Als Beschleunigungsansatz auf der CPU kann die in Kapitel 3 vorgeschlagene Beschleunigungsstruktur verwendet und untersucht werden.

Anhang A

GPUStreams

Für die Abstraktion der Programmierung auf OpenGL Client-Seite wurde das Konzept der *GPUStreams* verwendet und weiterentwickelt. Dieses basiert auf dem von Alexandrescu vorgeschlagenem *Policy-based Class Design* [ALE01] in C++ und wurde von Harris erstmals für die Grafikkarten-Programmierung verwendet [HAR04b].

A.1. Policy-based Class Design

Policy-based Class Design in C++ ist eine konsequente Weiterentwicklung der bereits in der *Standard Template Library* [STL04] begonnenen funktionalen Parametrisierung von Templates. Mit einem Template-Argument wird nicht mehr ein Typ substituiert, sondern die Implementierung einer Funktionalität. Dafür werden Klassen, so genannte *Policies*, als Template-Argumente verwendet. In Erweiterung des in der STL benutzen Konzepts der *type traits* schlägt Alexandrescu vor, die durch Templates parametrisierte Klasse, die *Host Class* von den *Policies* abzuleiten. Dies ermöglicht, dass auch die Struktur der *host class* sowie deren Member-Funktionen durch die Template-Argumente parametrisiert werden.

Die Vorteile des *Policy-based Class Design* sind dabei unter anderem

- ein hoher Optimierungsgrad des Programmcodes durch den Compiler.
- ein hoher Abstraktionsgrad bei der Programmierung durch die Aufteilung der Implementierung auf unabhängige funktionale Einheiten.
- eine hohe Flexibilität der Funktionalität der *host class*, da durch diese lediglich das minimale Interface der *Policies* definiert wird.
- eine größere Sicherheit bei der Programmierung.
- ein hohe Wiederverwendbarkeit des Programmcodes.

Policies können allerdings nur verwendet werden, wenn ihre Funktionalitäten orthogonal zueinander sind, dass heißt ihre Implementierung nicht voneinander abhängt. Insbesondere sollten dabei keine, beziehungsweise nur möglichst geringe Datenabhängigkeiten zwischen den *Policies* bestehen.

Das *Policy-based Class Design* soll am Beispiel der Implementierung einer Klasse für Vektoren näher erläutert werden.

```
template< class T,
          unsigned int Size = 3 >
class Vector {

private:

  T data[ Size ];
};
```

In der Klasse `Vector` sind bereits der Typ der Daten sowie die Größe des Vektors über Template-Argumente parametrisiert. Um die Vektor-Klasse verwenden zu können, werden noch Member-Funktionen benötigt, zum Beispiel eine Funktion `at()` zum Zugriff auf die Elemente.

```
template< class T,
           unsigned int Size = 3 >
class Vector {

public:

  /* … */

  const T&
  at( unsigned int index) const {

    return data[index];
  }

private:

  T data[ Size ];

};
```

Die Funktion `at()` kann in dieser Implementierung jedoch zu einem nicht definierten Programmverhalten führen, wenn `index` größer als `Size` ist.

Eine Lösung ist die Abfrage des Werts von `index` vor dem Zugriff.

```
  const T&
  at( unsigned int index) const {

    if ( index > Size) {

      throw std::range_error("Range error.");
    }

    return data[index];
  }
```

Diese zusätzliche Sicherheit erzeugt einen Overhead. In der Praxis möchte man deshalb möglichst einfach zwischen einer sicheren und einer schnellen Programmausführung wählen beziehungsweise wechseln können.

Dies kann durch die Implementierung der *range check*-Funktionalität[18] in einer *Policy* erreicht werden. Vector wird dann zu einer *host class*.

```
template< class T,
          unsigned int Size,
          template< unsigned int > class Range_Check_Policy >
class Vector : public Range_Check_Policy< Size > {

  typedef Range_Check_Policy< Size >  Range_Check;

public:

  const T&
  at( unsigned int index) const {

    Range_Check::check( index);

    return data[index];
  }

private:

  T data[ Size ];

};
```

[18] Als *range check* wird die Überprüfung eines Zugriffsindex bei Feldern o.ä. Typen bezeichnet, so dass sichergestellt ist, dass keine ungültigen Zugriffsoperationen ausgeführt werden.

Die Implementierungen der *Policies* wären

```
// policy for debug version
template< unsigned int Size >
class Perform_Check {

public:

  static void
  check( unsigned int index) {

    if ( index > Size) {

      throw std::range_error("Range error.");
    }
  }

};
```

sowie

```
// policy for release version
template< unsigned int Size >
class No_Check {

public:

  static void
  check( unsigned int index) throw() { }

};
```

Durch den Compiler ist dabei gewährleistet, dass die *host class* mit der `No_Check Policy` bei der Ausführung die gleiche Effizienz besitzt, wie die zunächst vorgestellte Vektor-Klasse mit der minimalen Version der `at()` Funktion, welche nur die `return` Anweisung enthält.

Je nach benötigter Programm-Version muss nur damit nur template-Argument ausgetauscht werden, um das Verhalten der Klasse zu verändern.

```
  // release version
  Vector<float, 2 , No_Check>       myvec1;
  // debug version
  Vector<float, 2, Perform_Check>   myvec2;
```

Für mehrere Klassen, zum Beispiel in einem Framework, kann dies über ein `typedef` geschehen.

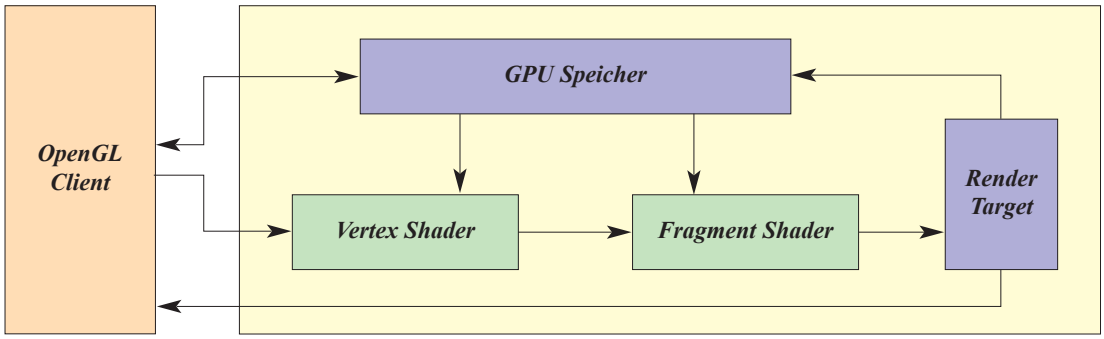


Abbildung 66: Grafikkartenabstraktion für die GPUStreams

A.2. Policy-based Class Design für GPU-Programmierung

Das *Policy-based Class Design* ist insbesondere auf Grund der OpenGL *State Machine* für die Abstraktion der Programmierung auf OpenGL Client-Seite geeignet. Die OpenGL *State Machine* gewährleistet, dass die einzelnen Teile des Rendering Prozesses unabhängig voneinander definiert werden können und das nahezu keine Datenabhängigkeiten zwischen diesen bestehen. So können zum Beispiel Shader-Programme geladen und gelinkt werden, ohne dass dabei ein *handle* auf eine verwendete Textur übergeben werden muss.
Für GPGPU Anwendungen, wozu auch das Ray-Casting-System zu zählen ist, bietet sich die Abstraktion eines *Rendering Pass*es in einer *host class* an. Die OpenGL Pipeline kann dafür in drei *Policies* abgebildet werden (Abb. 66):

- Die *Data Policy* (blau) definiert die Daten welche für die Berechnung verwendet werden, wohin die Ergebnis-Daten geschrieben werden und wie diese für die weitere Verarbeitung zur Verfügung gestellt werden

- Die *Shading Policy* (grün) definiert welche Berechnungen ausgeführt werden, zum Beispiel über die verwendeten Shader-Programme.

- Die *Compute Policy* (orange) kontrolliert die Durchführung der Berechnungen, zum Beispiel durch die Definition des Viewports und die gerenderte Geometrie.

Die *Shading* und die *Data Policy* sind dabei selbst *host classes*, wie zum Teil auch ihre Policies. Dies ermöglicht einen hohe Wiederverwendbarkeit des Programm-Codes, da jede OpenGL Funktionalität, wie zum Beispiel *PBuffer* oder *Vertex Buffer*

Objects, in einer eigenen Klasse implementiert sind. Für einen bestimmten *GPUStream* beziehungsweise *Rendering Pass* werden die Policies welche eine bestimmte OpenGL Funktionalität implementieren nur noch durch eine *Parameter Policy* oder eine *Data Policy* parametrisiert. Darüber hinaus konnte durch die Verwendung einer *Policy* für jede Funktionalität die Komplexität bei der Implementierung reduziert werden.

Anhang B

Testumgebung

CPU:	Intel Pentium4, 2.8 GHz (hyperthreaded)
Hauptspeicher:	1024 MB
Chipsatz:	Intel 875
Bussystem:	AGP8x
Grafikkarte:	Nvidia GeForce 6800 GT mit 256 MB RAM und 350 MHz Taktfrequenz
Betriebssystem:	Linux Fedora Core 2 (Kernel 2.6.9.1)
Grafikkarten Treiber:	Nvidia 1.0-6629

Literaturverzeichnis

[3DL04] Datasheet for Wildcat Realizm
http://content.3dlabs.com/Datasheets/realizm800.pdf

[AGP98] Accelerated Graphics Port Interface Specification, Revision 2.0, Intel Corporation, May 4, 1998
http://www.motherboards.org/files/techspecs/agp20.pdf

[AKE89] K. Akeley, *The Silicon Graphics 4D/240GTX Superworkstation*, IEEE Computer Graphics and Applications, v.9 n.4, p.71-83, 1989

[AKE93] K. Akeley, *Reality Engine graphics*, In Proceedings of ACM Siggraph 1993, pp.109-116, September 1993

[ALE01] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001

[ANA96] C. S. Ananian, G. Humphreys, *Tigershark: A hardware accelerated ray-tracing engine*. Technical report, Princeton University, 1996

[APP68] A. Apple, *Some Techniques for Shading Machine Renderings of Solids*, SJCC, pp. 37-45, 1968

[ARN87] B. Arnalldi, T. Priol und K. Bouatouch, *A new space subdivision method for raytracing CSG modeled scenes*, 1987, The Visual Computing, Springer Verlag, Vol. 3, pp. 98-108, 1987

[ARV87] J. Arvo, D. Kirk, *Fast Ray Tracing by Ray Classification*, In Proceedings of ACM Siggraph 1987, pp. 55-64, 1987

[ARV89] J. Arvo, D. Kirk, *A Survey of Ray Tracing Acceleration Techniques*, In A. Glassner (editor), *An Introduction to Ray Tracing*, Academic Press New York, 1989

[ATI02] GL_ATI_texture_float OpenGL extension, Revision 4, Date 04.12.2002
http://oss.sgi.com/projects/ogl-sample/registry/ATI/texture_float.txt

[BEN75] J. Bentley, *Multidimensional binary search trees used for associative searching*, Communication of the ACM, 18:509-517, 1975.

[BEN04] C. Benthin, I. Wald, P. Slusallek, *Interactive Ray Tracing of Free-Form Surfaces*, In Proceedings of Afrigraph 2004, 2004

[BLI76] J. F. Blinn, M. E. Newell, *Texture and reflection in computer generated images*, In Communications of the ACM, Vol. 19, No. 10, pp. 542-547, 1976

[BLI78] J. F. Blinn, *Simulation of wrinkled surfaces*, In Proceedings of ACM Siggraph 1978, pp. 286-292, 1978

[BUC04] I. Buck, K. Fatahalian, P. Hanrahan, *GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications*, In Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, 2004 http://graphics.stanford.edu/projects/gpubench/

[BUI75] B. T. Phong, *Illumination for computer generated pictures*, Communications of the ACM, Vol.18, No.6, pp. 311-317, 1975

[CAR02] N. A. Carr, J. D. Hall, J. C. Hart, *The ray engine*, In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware, 2002

[CUO97] N. D. Cuong, *An exploration of coherence-based acceleration methodes using the ray tracing kernel G/GX*, TU-Dresden http://www.stud.tu-ilmenau.de/~juhu/Papers/IWK/node21.html

[FAT04] K. Fatahalian, J. Sugerman, P. Hanrahan, *Understanding the efficiency of gpu algorithms for matrixmatrix multiplication*, In Proceedings of Graphics hardware, Eurographics Association, 2004

[FLY72] M. J. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, Vol. C-21, No. 9, pp. 948, 1972

[FUC80] H. Fuchs, Z. M. Kedem, B. F. Naylor. *On visible surface generation by a priori tree structures*, In Proceedings of ACM Siggraph 1980, pp. 124-133, 1980.

[GLA84] A. Glassner*, Space subdivision for fast ray tracing*, IEEE Computer Graphics and Applications, 4(10):15-22, 1984

[GLA89] A. Glassner et. al, *An Introduction to Ray Tracing*, Academic Press New York, 1989

[GOU71] H. Gouraud, *Continuous Shading of Curved Surfaces*, IEEE Transactions on Computers, Vol. C-20, No. 6, pp. 623-628, 1971

[GPG04] Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, Los Angeles, 2004

[HAI86] E. A. Haines, D. P. Greenberg, *The Light Buffer: A Ray Tracer Shadow Testing Accelerator*, IEEE Computer Graphics and Applications, vol. 6 no. 9 , pp. 6-15, 1986

[HAI87] E. A. Haines, *A proposal for standard graphics environments*, IEEE Computer Graphics and Applications, 7(11):3-5, 1987 http://www.acm.org/pubs/tog/resources/SPD/overview.html

[HAI89] E. A. Haines, *Essential Ray Tracing Algorithms*, In A. Glassner (editor), *An Introduction to Ray Tracing*, Academic Press New York, 1989

[HAI94] E. A. Haines, J. R. Wallace, *Shaft Culling for Efficient Ray-Traced Radiosity*, Photorealistic Rendering in Computer Graphics In Proceedings of theSecond Eurographics Workshop on Rendering, Springer-Verlag, New York, 1994, p.122-138, 1994

[HAL70] J. H. Halton, *A Retrospective and Prospective Survey of the Monte Carlo Method*, Society for Industrial and Applied Mathematics Review, 12, pp. 1-63, 1970

[HAL83] R. A. Hall, D. P. Greenberg, *A Testbed for Realistic Image Synthesis*, IEEE Computer Graphics and Applications, Vol. 3, No. 8, November, 1983, pp. 10-19, 1983

[HAN89] P. Hanrahan, *A Survey of Ray-Surface Intersection Algorithms*, A. Glassner (editor), *An Introduction to Ray Tracing*, Academic Press New York, 1989

[HAN90] P. Hanrahan, J. Lawson. *A language for shading and lighting calculations*. In Proceedings of ACM Siggraph 1990, pp. 289-298, 1990

[HAR02] M. J. Harris , G. Coombe , T. Scheuermann , A. Lastra, *Physically-based visual simulation on graphics hardware*, Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware, 2002

[HAR04a] M. Harris (Nvidia), *GPGPU: General-Purpose Computation on GPUs*, Presentation at Eurographics 2004

[HAR04b] M.Harris, *GPGPU Fluid*, Part of Nvidia SDK 8.0, 2004 http://download.developer.nvidia.com/developer/SDK/ Individual_Samples/samples.html

[HAV01] V. Havran. *Heuristic Ray Shooting Algorithm*s, PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000

[KAJ82] J. T. Kajiya, *Ray tracing parametric patches*, In Proceedings of the ACM Siggraph 1982, pp. 245-254, July 26-30, 1982

[KAY86] T. Kay, J. Kajiya, *Ray Tracing Complex Scenes*, In Proceedings of ACM Siggraph 1986, pp. 269-278, 1986.

[KAP85] M. Kaplan, *Space tracing a constant time ray tracer,* In Proceedings of the ACM Siggraph 1985, State of the Art in Image Synthesis, Course Notes, Vol. 11, 1985

[KES04] J. Kessenich, D. Baldwin, R. Rost, *The OpenGL Shading Language*, Language Version 1.10, Revision 59

[KRU03] J. Krüger, R. Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms,* In Proceedings of ACM Siggraph 2003, 2003

[MAG68] Mathematical Application Group, Inc., *3D Simulated Graphics Offered by Service Bureau*, Datamination, 13(1) February 1968

[MAY04] M. Y. Liang, T. Hewitt, *Adaptive Tessellation for Trimmed NURBS Surface*, Manchester Visualization Centre (unpublished) http://www.psy.gla.ac.uk/~ylma/Tessellation.pdf

[MAR00] W. Martin , E. Cohen , R. Fish , P. Shirley, *Practical ray tracing of trimmed NURBS surfaces*, Journal of Graphics Tools, v.5 n.1, pp. 27-52, 2000

[MAR03] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard. *Cg: A system for programming graphics hardware in a c-like language*, In Proceedings of ACM Siggraph 2003, pp. 896-907, 2003

[MAY04] Ma Ying Liang, Terry Hewitt, *Adaptive Tessellation for Trimmed NURBS Surface*, Manchester Visualization Centre (unpublished) http://www.psy.gla.ac.uk/~ylma/Tessellation.pdf

[MED04] A. Medvedev, K. Budankov, *NVIDIA GeForce 6800 Ultra (NV40)*, 16.12.2004, 20:30 http://www.digit-life.com/articles2/gffx/nv40-part1-b.html,

[MIC04] Microsoft. DirectX Homepage. http://www.msdn.com/directx.

[NUR04] *The NURBS library, The NURBS package*, Project at sourceforge http://libnurbs.sourceforge.net/

[NVI03] GL_NV_float_buffer OpenGL Extension, Revision 16, Date 16.06.2003
http://oss.sgi.com/projects/ogl-sample/registry/NV/float_buffer.txt

[NVI04a] NV_fragment_program2 specifications, Revision 72, 17.05.2004
http://www.nvidia.com/dev_content/nvopenglspecs/
GL_NV_fragment_program2.txt

[NVI04b] Nvidia SLI Homepage
http://www.nvidia.com/page/sli.html

[NVI04c] Nvidia Ultra Shadow II, Technical Brief
http://developer.nvidia.com/object/
geforce_6_series_tech_briefs.html

[OGL04] The OpenGL Graphics System: A Specification, Version 2.0, 22.10.2004,
http://www.opengl.org/documentation/specs/version2.0/
glspec20.pdf

[OHT87] M. Ohta, M. Maekawa, *Ray coherence theorem and constant time ray tracing algorithm*, Computer Graphics 1987, Proc. CG International '87, pp. 303-314, 1987

[OLA98] M. Olano , A. Lastra, *A shading language on graphics hardware: the pixelflow shading system*, In Proceedings of ACM Siggraph 1998, pp.159-168, 1998

[PCI04] PCI express Specifications 1.0
http://www.pcisig.com/specifications/pciexpress/

[PIE97] L. Piegl, W. Tiller, *The NURBS Book*, Springer 1997

[PRC99] INCITS/ISO/IEC 9899-1999, *C Programming Language*

[PUR02] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan. *Ray Tracing on Programmable Graphics Hardware*, In Proceedings of ACM Siggraph 2002, pp. 703-712, 2002

[PUR04] T. J. Purcell. *Ray Tracing on A Stream Processor*, PhD thesis, Depatement of Computer Science, Stanford University, 2004

[ROK91] J. Rokne. *The area of a simple polygon*; In Graphics Gems II., Editor: J. R. Arvo, Academic Press, 1991

[ROS04] R. J. Rost, OpenGL shading language *The Orange Book*, Addison Wesley 2004

[RUB80] S. Rubin, T. Whitted, *A Three-Dimensional Representation for Fast Rendering of Complex Scenes*, Computer Graphics, Volume 21, No. 4, pp. 110-116, 1980

[SGI97] GLX_SGIX_*pbuffer* OpenGL Extension, Revision 1.28, Date 20.03.1997
http://oss.sgi.com/projects/ogl-sample/registry/SGIX/pbuffer.txt

[STL04] Standard Template Library Programmer's Guide
http://www.sgi.com/tech/stl/

[STR92] D. Stredney , R. Yagel , S. F. May , M. Torello, *Supercomputer assisted brain visualization with an extended ray tracer*, In Proceedings of the 1992 workshop on Volume visualization, pp.33-38, 1992

[SZI96] L. Szirmay-Kalos, G. Márton. *On the complexity of ray shooting*, In Dagstuhl Seminar on Rendering, 1996, 1996

[SZI97] L. Szirmay-Kalos, G. Márton. *On the limitations of worst-case optimal ray shooting algorithms*, In Proceedings of Winter School of Computer Graphics 97, pp. 562-571, 1997

[SZI98a] L. Szirmay-Kalos, G. Márton. *Analysis and construction of worst-case optimal ray shooting algorithms*, Computers and Graphics, 22(2-3):167-174, 1998

[SZI98b] L. Szirmay-Kalos, G. Márton, *Worst-case versus average case complexity of ray-shooting.Computing*, 61(2):103-131, 1998

[SZI02] L. Szirmay-Kalos , V. Havran , B. Balázs , L. Szécsi, *On the efficiency of ray-shooting acceleration schemes*, Proceedings of the 18th spring conference on Computer graphics, 2002

[THG04] Tom's Hardware Guide, *Performance Leap: NVIDIA GeForce 6800 Ultra,* 05.12.2004, 10:30 http://graphics.tomshardware.com/graphic/20040414/geforce_6800-09.html

[TWE04] *PCI Express: Kurz erklärt*, 16.12.2004, 19:30, http://www.tweakpc.de/hardware/infos/mainboard/pci_express/s01.php

[WAL01] I. Wald, C. Benthin, M. Wagner, P. Slusallek. *Interactive Rendering with Coherent Ray Tracing*, Computer Graphics Forum, 20(3), 2001

[WAL04a] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*, PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, 2004

[WAL04b] I. Wald, A. Dietrich, P. Slusallek, *An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models*, Eurographics Symposium on Rendering, 2004

[WGL01] WGL_ARB_render_texture OpenGL Extension, Date 16.07.2001 http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_render_texture.txt

[WGL02a] WGL_ARB_*pbuffer* OpenGL Extension, Revision 1.1, Date 12.03.2002, http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt

[WGL02b] NV_render_depth_texture OpenGL Extension, Revision 7, Date 08.01.2003
http://oss.sgi.com/projects/ogl-sample/registry/NV/render_depth_texture.txt

[WHI80] T. Whitted, *An improved illumination model for shaded displays*, Communications of the ACM, Vol. 23, No. 6, pp. 343-349, 1980

[WIL78] L. Williams, *Casting curved shadows on curved surfaces*, In Proceedings of ACM Siggraph 1978, pp. 270-274, 1978