Brandenburgische Technische Universität Cottbus
Institut für Informations- und Medientechnik
Lehrstuhl Grafische Systeme

Bachelorarbeit

Stochastischer GPU-Strahlenverfolger für GroIMP

Thomas Huwe
MatrikelNr.: 2400567

Datum der Abgabe: 16.01.2008

1. Betreuer: Prof. Dr. Winfried Kurth
2. Betreuer: Reinhard Hemmerling

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Cottbus, den

_____
- - Unterschrift - -

Danksagung

An dieser Stelle möchte ich einigen Personen, die mich bei dieser Arbeit unterstützt haben, meinen Dank aussprechen.

An erster Stelle wäre das Prof. Dr. Kurth für die Ausschreibung der Arbeit. Ich hätte mir keine schöneres Thema vorstellen können. Sehr dankbar bin ich für die Korrekturvorschläge, die dieser Arbeit den letzen Schliff gaben.

Mein großer Dank geht an Reinhard Hemmerling für seine tatkräftige Unterstützung und Freundschaft. Stets konnte er meine Fragen beantworten und stand mir dank ICQ jederzeit zur Verfügung. Die gemeinsamen Kaffeepausen, bei denen meistens auch Stephan Rogge dabei war, den ich hier auch erwähnen möchte, waren gute Gelegenheiten, Sachverhalte in angenehmer Atmosphäre zu diskutieren.

Weiterhin danke ich Ole Kniemeyer und den anderen GroIMP -Entwicklern,

Ein besonderer Dank geht an meine Freundin Christin Bartel. Sie hat nicht aufgehört, an mich zu glauben und mir Mut zugesprochen, dass ich diese Arbeit erfolgreich abschließen werde. Vielen Dank auch für die aufgebrachte Geduld in den letzen Monaten.

Meinen Eltern möchte ich für ihre Unterstützung in den vergangenen Jahren danken. Ohne sie wäre mein Studium in Cottbus und diese Arbeit nicht möglich gewesen. Meinen Kommilitonen, speziell Dennis Wassenberg, der mir in VPS1 sehr viel Arbeit abgenommen, hat möchte ich für die schöne Zeit danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Diese Arbeit befasst sich mit der Erweiterung der 3D-Modellierungs-Plattform GroIMP um einen Strahlenverfolger, der globale Beleuchtungseffekte durch das Monte-Carlo-Verfahren physikalisch korrekt darstellen kann.

Die Strahlenverfolgung ist ein Algorithmus zum Erzeugen von Computerbildern, der auf der Aussendung und Verfolgung von Strahlen basiert und in seiner einfachsten Form die Lösung des Problems der gegenseitigen Verdeckung dreidimensionaler Objekte im Raum liefert. Dieses grundlegende Verfahren lässt sich einfach um die Bestimmung der weiterführenden Wege der Strahlen nach dem Auftreffen auf Objektoberflächen erweitern. Erweiterungen, die so den Verlauf der Lichtstrahlen durch die Szene simulieren, sind in der Lage, Licht und Schatten zu berechnen, wodurch sehr realistische Bilder synthetisiert werden können.
Eine bekannte Erweiterung, wie die von Whitted aus [Whi80], die sogar die globale Beleuchtung berechnet, generiert aber unnatürliche Bilder, deren Schatten beispielsweise nur hart sind und Reflexionen immer perfekt darstellen.
Die Generierung eines Bildes aus der gegebenen Szene geschieht deshalb mit der Implementierung einer im Jahre 1986 von Kajiya als „Pathtracing“ bezeichneten Fortentwicklung des oben beschriebenen normalen Algorithmus. [Kaj86]
Globale Beleuchtungseffekte wie Reflexion und Refraktion werden durch die von Kajiya entwickelte Rendergleichung naturgetreu simuliert. Die näherungsweise Lösung dieser Gleichung geschieht durch das Monte-Carlo-Verfahren, denn, laut Veach:

> „Currently, only Monte Carlo approaches can handle the wide range of surface geometries, reflection models, and lighting effects that occur in real environments.“ [Vea97]

Das Besondere an der in dieser Arbeit vorgestellten Implementierung ist, dass die Berechnung des Algorithmus auf dem Grafikkartenprozessor heutiger Grafikkarten durchgeführt wird.
Der Grafikkartenprozessor bietet sich für die Berechnung des Algorithmus an, da er ein Vertreter der SIMD (Single Instruction Multiply Data) Architektur ist und damit einen Parallelprozessor darstellt.

> „Furthermore, the fragment processor works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with

each fragment processor engine working on one fragment concurrently)[...] “ [KR05]

Dies lässt sich nutzen, da laut [WSBW01] das Verfahren der Strahlenverfolgung von Natur aus parallel ist und sich in seiner Effizienz steigern lässt, führt man es auf mehreren Recheneinheiten aus.
Durch die stetig wachsende Leistung der Grafikkartenprozessoren und die Möglichkeit der direkten Programmierung dieser, durch „Shader-Programme“, ist es möglich, die Grafikkarten-Hardware auszunutzen, um einen deutlichen Geschwindigkeitsvorteil gegenüber üblichen Berechnungen auf der CPU zu erhalten.

Das Ergebnis dieser Arbeit ist die Implementierung des Strahlverfolgungs-Algorithmus auf dem Grafikkartenprozessor, der ein Bild schneller als CPU-basierte Lösungen generiert.

1.2 Aufbau der Arbeit

Das zweite Kapitel schafft die Grundlagen, die für das Verständnis dieser Arbeit von Bedeutung sind. Dabei wird ein kleiner historischer Überblick über die Entwicklung der Strahlverfolgung skizziert. Zusätzlich wird gezeigt, wie die Arbeitsweise moderner Grafikkarten aussieht und warum es sinnvoll erscheint, das Potenzial der Grafikkarte in unüblicher Weise zu nutzen.

Das dritte Kapitel geht näher auf die Vorüberlegungen ein, die nötig waren, um diese Arbeit zu realisieren. Hier wird näher erläutert, was bei der Programmierung der Grafikkarte zu beachten ist und welche Probleme sich daraus ergeben.
Anschließend wird die direkte Implementierung an Hand von ausgesuchtem Quellcode vorgestellt.

Die entwickelte Lösung wird mittels Testläufen und dem Vergleich mit dem integrierten Strahlverfolger „Twilight“ (welcher auf der CPU rechnet) in Kapitel 5 besprochen.

Die Arbeit schließt mit einer Zusammenfassung und gibt einen Ausblick auf weitere Entwicklungen im Bereich der Grafikkartenprogrammierung.

2 Grundlagen

2.1 Grundlagen der Strahlenverfolgung

2.1.1 Funktionsweise

Die Strahlenverfolgung, im Englischen Raytracing genannt, ist eine Bilderzeugungstechnik, die ein Bild für eine gegebene dreidimensionale Szenenbeschreibung berechnet.
Die Idee, die der Strahlenverfolgung zu Grunde liegt, stammt aus der geometrischen Optik. Diese liefert ein idealisiertes Modell, um das Verhalten von Licht in der makroskopischen Welt zu beschreiben. Licht wird als gedachte Strahlen betrachtet, die sich geradlinig ausbreiten und mit optischen Elementen wechselwirken können. Lichtstrahlen werden reflektiert, gebrochen, oder absorbiert.
Diese Lichtstrahlen gilt es zu verfolgen und auf Schnittpunkte mit in der Szene befindlichen Objekten zu untersuchen.

Arthur Apple war der Erste, der dieses Modell angewandt hat, um damit in [App68] eine Lösung für das Verdeckungsproblem dreidimensionaler Objekte im Raum vorzuschlagen. Er bezeichnet diese minimalistischste Form der Strahlenverfolgung als „Ray Casting". Bei diesem Verfahren werden keine Sekundärstrahlen erzeugt, wodurch Spiegelungen, Reflexionen oder Schatten nicht berechnet werden können. Es lassen sich nur Objekte, die sich gegenseitig verdecken, anhand der Entfernung vom Betrachter zu dem Schnittpunkt sortieren. Somit sind nur die Objekte pro Pixel sichtbar, deren errechneter Schnittpunkt dem Betrachter am Nächsten liegt.

2.1.2 Strahlverfolgung

Bei der Verfolgung der Strahlen unterscheidet man zwei verschiedene Varianten, die „Vorwärts-Strahlverfolgung" und die „Rückwärts-Strahlverfolgung". (siehe Abbildung 2.1)
Beiden Verfahren ist gemein, dass Strahlen mit allen in der Szene enthaltenen Objekten auf Schnittpunkte hin untersucht werden, um festzustellen, ob das Objekt sichtbar ist.
Bei der so genannten „Vorwärts-Strahlverfolgung" ist die Ausbreitungsrichtung des Lichts genau wie in der realen Welt von der Lichtquelle aus in die Szene verlaufend. Die Lichtstrahlen werden zu den Objekten hin verfolgt und anschließend zum Betrachter. Bei diesem Verfahren ist aber die Berechnung einer Mehrzahl an Lichtstrahlen überflüssig, da viele dieser die Objekte verfehlen und noch weniger so reflektiert werden, dass der Beobachter getroffen wird.
Diesen Mehraufwand hat man bei der „Rückwärts-Strahlverfolgung" nicht, da hierbei so genannte Sehstrahlen direkt beim Beobachter erzeugt werden und von dort in die Szene

geschickt werden. Somit werden nur Strahlen verfolgt, die für den Beobachter von Bedeutung sind. Die Folge dessen ist, dass die „Rückwärts-Strahlverfolgung“ abhängig vom Standpunkt des Betrachters ist. Das bedeutet, dass bei einer Änderung der Blickrichtung die gesamte Szene neu berechnet werden muss.

Der bei dieser Arbeit implementierte Stahlverfolger arbeitet nach dem zweiten Prinzip.

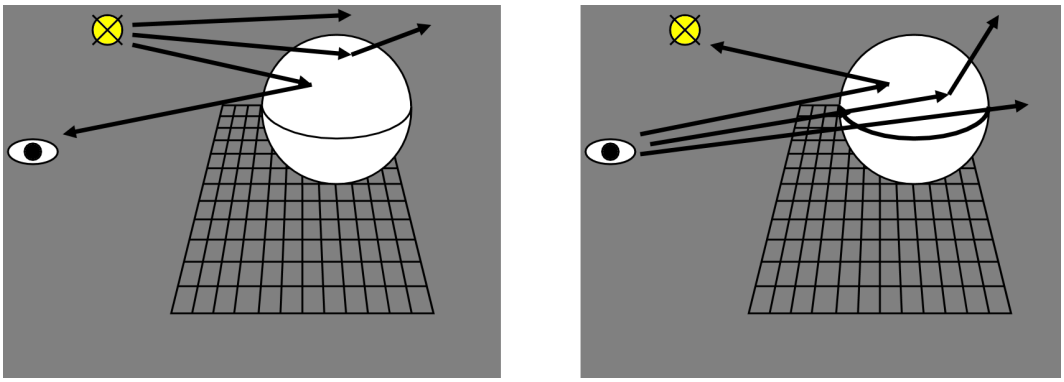


Abbildung 2.1: Vorwärts- und Rückwärts-Strahlverfolgung

2.1.3 Bildebene

In einiger Entfernung zur Position des Betrachters liegt die Bildebene, die dem zu erzeugenden Bild entspricht. Diese Bildebene wird in ein regelmäßiges Gitter unterteilt, das den Pixeln und damit der gewünschten Auflösung entspricht. Durch den Mittelpunkt eines jeden dieser Pixel wird ein Sehstrahl vom Betrachter aus generiert, der weiter in die Szene verfolgt und auf Schnittpunkte mit den Objekten mittels geometrischer Verfahren untersucht wird. (siehe Abbildung 2.2)

2.1.4 Schnittpunktberechnung

Die Kernaufgabe der Strahlverfolgung besteht darin, die in der Szene enthaltenen Primitive auf eventuelle Schnittpunkte mit den Sehstrahlen hin zu untersuchen. Typische Primitive wären zum Beispiel Zylinder, Quader oder Kugel, da diese sich gut als mathematische Gleichungen darstellen lassen, was die Berechnung der Schnittpunkte vereinfacht.
Oft werden aber auch nur Dreiecke als Primitive benutzt, da man mit diesen beliebige Objekte näherungsweise zusammensetzen kann.
Die Schnittpunktberechnung ist aber sehr aufwendig. Bei normalen Szenen mit beispielsweise 100 Objekten und einer Auflösung von 1024 x 1024 Pixeln müssen ca. 100 Millionen Schnittpunktberechnungen durchgeführt werden. Nach [Whi80] werden 75% der Rechenzeit für die Berechnung der Schnittpunkte benötigt und 12% für die Berechnung

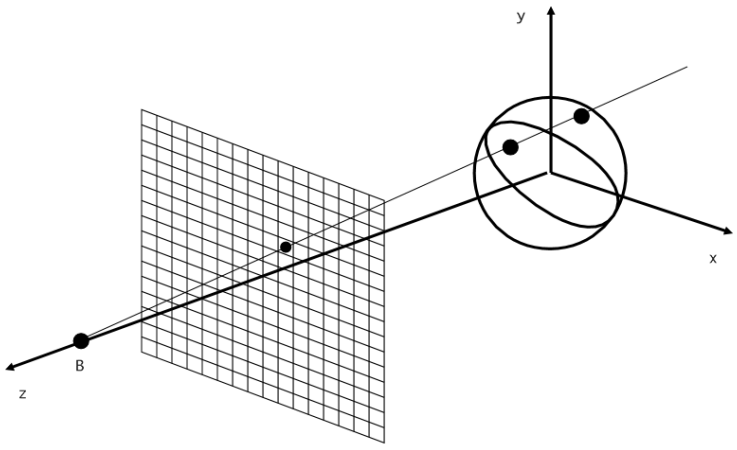


Abbildung 2.2: Die Bildebene

des Beleuchtungsmodells.

Jedes Mal, wenn ein Schnittpunkttest erfolgreich war, wird, wie im nächsten Abschnitt erläutert, der Farbwert der Oberfläche an diesem Punkt mit Hilfe eines lokalen Beleuchtungsmodells berechnet.

2.1.5 Schattenberechnung und Licht

Die Schattenberechnung der Szene lässt sich, wie bereits 1968 von Appel gezeigt, einfach durchführen. An dem gefundenen Schnittpunkt wird pro Lichtquelle ein Sekundärstrahl generiert, den man als Schattenfühler bezeichnet. Diese werden in Richtung jeder Lichtquelle ausgesandt und durchlaufen den Strahlverfolgungs-Algorithmus erneut. Trifft der Schattenstrahl dabei auf ein weiteres Objekt, verdeckt dieses die Lichtquelle und der Schnittpunkt befindet sich im Schatten. Andernfalls wird der Punkt beleuchtet.
Die so erzeugten Schatten sind aber sehr hart abgrenzend, da es nur die Zustände gibt, dass die Lichtquelle getroffen wird oder nicht. Entweder ist der Punkt schwarz oder beleuchtet. In der Natur bestehen Schatten aber aus einem Halb- und einem Kernschatten, deren Übergang fließend ist. Bei der Strahlverfolgung können Halbschatten erzeugt werden, indem mehrere Schattenstrahlen auf der Oberfläche einer flächigen Lichtquelle zufällig verteilt werden. Dadurch entstehen Strahlen, die die Lichtquelle treffen und welche, die die Objekte dazwischen treffen. Aus diesem Verhältnis wird ein entsprechender Farbwert für den Schatten errechnet.

Durch die gute Erweiterbarkeit des Strahlverfolgungs-Algorithmus ist es möglich, das

relativ einfache Verfahren der Verdeckungsberechnung für eine realistische Bildsynthese zu nutzen, indem an den gefundenen Schnittpunkten ein lokales Beleuchtungsmodell angewandt wird. Mögliche lokale Beleuchtungsmodelle wären das Phong-Modell oder die Modelle von Schlick und Blinn. Trifft der Schattenfühler auf kein Objekt, liegt der Punkt nicht im Schatten, und es wird unter Verwendung der für diesen Punkt geltenden Reflexionseigenschaft, dem Normalenvektor und den Vektoren zu den Lichtquellen und dem Beobachter das vorherrschende Lichtverhältnis berechnet.

2.1.6 Rekursive Strahlverfolgung

Mit der Strahlverfolgung ist es möglich, die globale Beleuchtung, also alle möglichen Wege der Lichtstrahlen, in einer dreidimensionalen Szene zu simulieren. Für die Berechnung des Farbwertes eines Punktes wird das direkt einfallende Licht und das Licht, das den Punkt über Reflexion und Transmission erreicht, berücksichtigt. Damit werden die Lichtverhältnisse an anderen Stellen der Szene für die Berechnung der Beleuchtung mit einbezogen.
Dies geschieht durch die Erzeugung und Verfolgung von zwei weiteren Sekundärstrahlen. Es wird ein reflektierter und ein gebrochener (transmittierter) Sehstrahl erzeugt. Diffuse Reflexion kann nicht dargestellt werden, da in diesem Fall der eintreffende Lichtstrahl in unendlich viele Richtungen abgestrahlt werden müsste. Es ist nicht möglich, die Strahlen weiter zu verfolgen.
Die beiden Strahlen werden rekursiv auf ihrem Weg durch die Szene verfolgt, indem sie den Algorithmus zur Schnittpunktberechnung ebenfalls durchlaufen. Die Farbwerte der rekursiven Verfolgung werden zu dem Farbwert des Primärstrahls hinzugerechnet.
Die Abbruchbedingung für die Rekursionsschleife ist entweder das Treffen einer Lichtquelle, eine zu geringe Intensität des Strahls, das Verlassen der Szene oder das Erreichen einer festgelegten Rekursionstiefe.
Die Richtung des reflektierenden Strahls bei spiegelnden Oberflächen ist bei einfachen Strahlverfolgern immer ideal reflektierend. Sie verhält sich gemäß dem Reflexionsgesetzes bei spiegelndem Material beziehungsweise des Brechungsgesetzes bei transparenten Objekten.
Reflexionen an diffusen Oberflächen lassen sich nicht darstellen.

Diffuse Strahlverfolgung

Mit der diffusen Strahlverfolgung können diffuse und spekulare Reflexionen dargestellt werden. Dies ist möglich, da bei diesem Verfahren die Reflexions- oder Transmissionsrichtung zufällig erzeugt wird. Die Erzeugung der Strahlen ist dabei abhängig von einer Verteilung, die der Reflektanzverteilungsfunktion der Oberfläche des getroffenen Objekts entspricht.
Die zufällige Generierung der Strahlen hat zur Folge, dass ein Rauschen im Bild entsteht. Dieses Rauschen lässt sich durch die Verwendung mehrerer Primärstrahlen pro Pixel reduzieren, so dass es nicht mehr stört. Durch die Verwendung mehrerer Primärstrahlen erhält man zusätzlich eine Kantenglättung.

Pathtracing

Bei der diffusen Strahlverfolgung werden nur an spiegelnden Oberflächen Sekundärstrahlen erzeugt. Globales diffuses Licht wird nicht berücksichtigt. Damit eine direkte Beleuchtung durch diffuse Oberflächen in der Umgebung erreicht wird, müsste ein diffuser Strahl erzeugt werden.
Dies erreicht man mit Kajiyas Path-Tracing [Kaj86]. Das Grundprinzip bleibt das Gleiche. Der Unterschied zur normalen Strahlverfolgung ist, dass am Schnittpunkt nicht ein reflektierender und ein transmittierender Strahl weiterverfolgt wird. Es wird nur ein einziger Strahl erzeugt und weiterverfolgt. Die Art des Strahls wird mittels eines Zufallsgenerators festgelegt. Der Zufallsgenerator erzeugt eine gleich verteilte Zufallszahl im Intervall [0,1]. Dieses Intervall wird in drei Bereiche unterteilt. Die Unterteilung des Intervalls entspricht dabei der Reflektanzverteilungsfunktion der getroffenen Oberfläche. Je nachdem, welcher Bereich des Intervalls getroffen wurde, wird ein reflektierter, diffuser oder transmittierter Strahl weiterverfolgt.

2.1.7 Pseudocode der Strahlverfolgung

Der wesentliche Ablauf des Algorithmus zur Strahlenverfolgung soll hier mit Hilfe von Pseudocode aus [FvDFH90] dargestellt werden. (siehe Listing 2.1)

```
select center of projection and window on view plane;
for (each scan line in image) {
  for (each pixel in scan line) {
    determine ray from center of projection through pixel;
    pixel = RT_trace (ray, 1);
  }
}

RT_color RT_trace (RT_ray ray, int depth) {
  determine closest intersection of ray with an object;

  if (object hit) {
    compute normal at intersection;
    return RT_shade(closest object hit,ray,intersection,normal,depth);
  } else
    return BACKGROUND_VALUE;
}  /* RT_trace */

RT_color RT_shade (
  RT_object object,                    /* Object intersected */
  RT_ray ray,                          /* Incident ray */
  RT_point point,                      /* Point of intersection to shade */
  RT_normal normal,                    /* Normal at point */
  int depth)                           /* Depth in ray tree */
{
  RT_color color;
  RT_ray rRay, tRay, sRay;             /* Reflected, refracted, and shadow rays */
  RT_color rColor, tColor;             /* Reflected and refracted ray colors */

  color = ambient term;
  for (each light) {
    sRay = ray to light from point;
    if (dot product of normal and direction to light is positive) {
      compute how much light is blocked by opaque and transparent surfaces,
        and use to scale diffuse and specular terms before adding them to color;
    }
  }

  if (depth < maxDepth) {
    if (object is reflective) {
      rRay = ray in reflection direction from point;
      rColor = RT_trace (rRay, depth + 1);
      scale rColor by specular coefficient and add to color;
    }

    if (object is transparent) {
      tRay = ray in refraction direction from point;
      if (total internal reflection does not occur) {
        tColor = RT_trace (tRay, depth + 1);
        scale tColor by transmission coefficient and add to color;
      }
    }
  }
  return color;                        /* Return color of ray */
} /* RT_shade */
```

Listing 2.1: Pseudocode der rekursiven Strahlenverfolgung

2.1.8 Beschleunigungsverfahren

Durch spezielle Beschleunigungsverfahren lässt sich die Effizienz des Algorithmus steigern. Da der Großteil der benötigten Rechenzeit für die Berechnung der Schnittpunkte verbraucht wird, konzentrieren sich die meisten Verfahren auf das schnellere Berechnen der Schnittpunkte oder eine Reduzierung der benötigten Berechnungen.

Hüllkörper

Bei der am häufigsten verwendete Methode zur schnelleren Berechnung wird ein Hüllkörper über das zu testende Objekt gelegt. Hüllkörper sind dabei geometrische Primitive, wie eine Kugel oder eine Box, deren Schnittpunktberechnung einfacher auszuführen ist, als der Test mit dem enthaltenen komplexeren Objekt selbst. (siehe Abbilung 2.3).

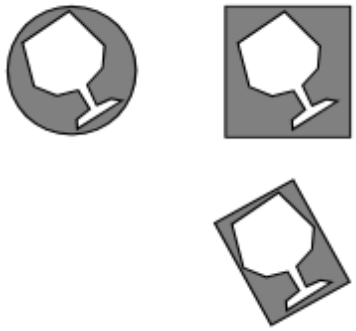


Abbildung 2.3: Hüllkörper

Die Idee dahinter ist, dass der eigentliche Schnittpunkttest mit dem Objekt nur dann erfolgt, wenn es vorher einen Schnittpunkt mit dem Hüllkörper gab.
Objekte, die nah beieinander liegen, lassen sich mit einem größeren Objekt umhüllen. Hier gilt wieder, dass auch die darin enthaltenen Objekte nicht getroffen werden, wenn der Hüllkörper nicht getroffen wird. Dies lässt sich erweitern, indem der Raum hierarchisch unterteilt wird.

Octree

Bei der Verwendung eines Octrees wird der Raum rekursiv in acht gleich große Würfel unterteilt. (siehe Abbildung 2.4). Je nach Objektdichte werden die einzelnen Subräume weiter unterteilt. Bereiche mit einer höheren Anzahl an Objekten werden mehrfach unterteilt. Dadurch passt sich der Octree der Struktur der Szene an. Bei unregelmäßig verteilten Objekten können Leerräume groß werden. Diese leeren Bereiche mit wenigen Zellen können mit wenigen Schritten durchquert werden.
Das Octree-Verfahren ist immer unabhängig von der Szene und ist einfach zu implementieren.

K-dimensionaler-Baum

Der kd-Baum ist eine weitere Möglichkeit, den Raum hierarchisch zu unterteilen. Er wurde von Bentley in [Ben75] vorgestellt.
Die Teilung des Raumes erfolgt rekursiv durch achsenparallele Ebenen in jeweils zwei Teilräume. Dies wird so lange fortgesetzt, bis ein Teilraum keine oder nur wenige Objekte

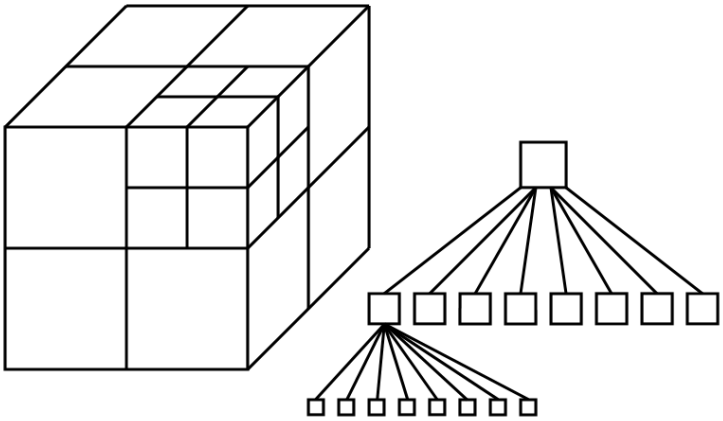


Abbildung 2.4: Octree als 3D-Würfel und als Graph

enthält. Dadurch ist der kd-Baum eine Variante des Binärbaumes. Die Teilräume müssen nicht, wie beim Octree, die gleiche Größe besitzen.

2.1.9 Fazit

Das Verfahren der Strahlverfolgung gilt im Allgemeinen als eine relativ zeitintensive Technik, wobei die Dauer für die Erzeugung eines Bildes von der Komplexität der Szene und der verwendeten Hardware abhängig ist.
Will man eine komplexe Szene mit komplizierten Objekten berechnen und dabei Materialeigenschaften der Flächen wie Farbe oder winkelabhängige Reflexionscharakteristik mit berücksichtigen, existiert keine Alternative zum Raytracing.

> "Leider kann eine vollständige Beleuchtungssimulation nicht sinnvoll auf heutiger Graphik-Hardware implementiert werden. Die Gründe dafür liegen in der Struktur des Rasterisierungsalgorithmus[...]" "Rasterisierung muss für jeden Effekt auf Speziallösungen durch geschickte Approximationen zurückgreifen." [SW01]

Das Einsatzgebiet beschränkt sich daher überwiegend auf Bereiche, bei denen eine hohe Bildqualität im Vordergrund steht.

2.2 Stochastische Strahlverfolgung

2.2.1 Die Rendergleichung

Kayjiya entwickelt in [Kaj86] mit Hilfe der geometrischen Optik die sogenannte Rendering-Gleichung. Sie ist eine Integralgleichung, die die Energieerhaltung bei der Ausbreitung des Lichtes in einer dreidimensionalen Szene beschreibt. Damit bildet diese Gleichung die mathematische Grundlage für eine Mehrzahl der Verfahren zur globalen Beleuchtung.
Die Formel zieht für die Bestimmung der Reflexion für ein Oberflächenstück das komplette einstrahlende Licht in Betracht. Das in Richtung $\vec{w}$ abgestrahlte Licht am Punkt x entpricht dem Licht, das durch die Oberfläche abgegeben wurde, zuzüglich der Summe des in Richtung $\vec{w}$ reflektierten einfallenden Lichtes.

$$L(x, \vec{w}) = L_e(x, \vec{w}) + \int_{\Omega} b(\vec{w'}, x, \vec{w}) L(x, \vec{w'})(\vec{w'} * \vec{n}) d\vec{w'} \tag{2.1}$$

- $L(x, \vec{w})$ beschreibt die Strahlungsdichte einer Oberfläche im Punkt x in Richtung $\vec{w}$
- $L_e(x, \vec{w})$ beschreibt die emittierende Strahlungsdichte einer Oberfläche im Punkt x in Richtung $\vec{w}$, wenn der Punkt x selbst eine Lichtquelle darstellt
- Ω ist die Einheits-Hemisphäre um den Punkt x
- $b(\vec{w'}, x, \vec{w})$ ist der Streuungsterm in Form einer bidirektionalen Reflektanz-Verteilungsfunktion, der die aus Richtung $\vec{w'}$ in Richtung $\vec{w}$ reflektierte Strahlung angibt
- $L(x, \vec{w'})$ beschreibt die Strahlungsdichte, die aus Richtung $\vec{w'}$ den Punkt x erreicht
- $\vec{n}$ ist der Normalenvektor der Oberfläche im Punkt x

2.2.2 Lösung der Gleichung

Das klassische Verfahren zur Strahlverfolgung löst die Rendering-Gleichung nur eingeschränkt. Ein Problem ist, dass das Integral der Rendering-Gleichung keine analytische oder numerische Lösung besitzt und deshalb nur approximiert werden kann.
Für die näherungsweise Lösung des Integrals können Monte-Carlo-Methoden benutzt werden.
Der Grundgedanke für das Lösen von Integralen mit Hilfe von Monte-Carlo-Methoden besteht darin, das Integral der Funktion $\int f(x)dx$ durch das Aufaddieren der Funktionswerte $f(x_i)$ an zufällige Stellen x_i anzunähern.
Dies wird erreicht, indem die Verfolgung der Primärstrahlen mehrfach ausgeführt wird und beim Schnittpunkt ein zufällig gerichteter Sekundärstrahl erzeugt wird. Die damit errechneten Farbwerte werden aufsummiert und über die Anzahl der Primärstrahlen gemittelt.

Für eine möglichst gute Verteilung der Strahlen, was einer gleichmäßigen Abtastung der Funktion entspricht, wird der Definitionsbereich der Zufallszahlen in gleich große Teilintervalle unterteilt. Die Näherung lässt sich verbessern, wenn Informationen über die Oberflächenbeschaffenheit vorliegen. Die Abtastung wird dann auf wichtige Stellen konzentriert. Wenn ein Material mehr reflektiert als transmittiert, wird dieser Bereich im Intervall größer sein, wodurch mehr reflektierende Strahlen erzeugt werden. Dieses Verfahren wird “importance sampling“ genannt.

> „In stochastischen Ray Tracing-basierten Bildgenerierungs- und globalen Beleuchtungstechniken, z.B. in Photon Map-globale Beleuchtungssimulation, muß eine sehr große Anzahl an Strahlen in die Szene geschossen werden, um die globale Beleuchtung und/oder das endgültige Bild zu berechnen. Die Leistung dieser Techniken kann daher wesentlich verbessert werden, indem die Strahlen vorzugsweise in Richtungen geschossen werden, wo sie einen hohen Beitrag liefern. Importance Sampling Techniken versuchen dies zu tun[..]“ [Hey02]

Mit einer steigenden Zahl von Primärstrahlen konvergiert die Näherung gegen das exakte Ergebnis des Integrals der Rendering-Gleichung.
Fehler, die durch die Näherung entstehen, machen sich als ein Bildrauschen bemerkbar. Bei der Verwendung einer relativ kleinen Anzahl von Primärstrahlen werden die berechneten Bilder einen hohen Rauschanteil haben. Durch eine genügend große Anzahl an Primärstrahlen verringert sich das Bildrauschen und wirkt nicht störend.

2.2.3 Fazit

Effekte wie diffuse Reflexion oder Brechung von Lichtstrahlen wirken bei der Verwendung von stochastischen Strahlenverfolgern weitaus natürlicher. Dadurch, dass die Sekundärstrahlen zufällig generiert werden, sind die Reflexionen nicht ideal, was sich mit den in der Natur auftretenden Reflexionen vergleichen lässt. Zusätzlich sind die erzeugten Schatten nicht scharf abgrenzend, sondern ähnlich wie in der Realität eher weich. Diffuse Wechselwirkungen wie das „Ausbluten“ einer Farbe einer Fläche an sie benachbarte Flächen erhöhen den Realismus der Szene weiter.
Da mehrere Primärstrahlen verfolgt werden, wird Kantenglättung erreicht, indem die einzelnen Strahlen stochastisch auf der Fläche der einzelnen Pixel verteilt werden. Die Mittelwertbildung über die aufsummierten Farbwerte verringert Treppeneffekte, auch als Antialiasing bezeichnet.
Das Bildrauschen ist ein weiterer Nebeneffekt, der dem synthetisierten Bild mehr Realismus verleiht, indem die künstliche Note der Computergrafik genommen wird.

Der Rechenaufwand ist im Vergleich zum einfachen Strahlverfolger jedoch erheblich höher.

2.3 Funktionsweise der GPU

2.3.1 Die Grafikpipeline

Eine dreidimensionale Szene besteht aus Objekten, die sich im Raum befinden. Diese Objekte werden durch ihre Eckpunkte definiert, die in einer bestimmten Weise miteinander verbunden werden. Die Informationen über die Lage dieser Eckpunkte im Raum werden mathematisch als Vektoren formuliert (Koordinaten). Damit aus dieser abstrakten Beschreibung ein gerastertes Bild entsteht, das am Monitor dargestellt werden kann, müssen diese 3D-Daten erst durch die Grafikpipeline verarbeitet werden. Die Verarbeitung beinhaltet unter anderem die Umrechnung der Bildschirmkoordinaten in Gerätekoordinaten, das Aufbringen von Texturen, das Klipping an den Bildschirmkanten oder auch die Kantenglättung. Der grundlegende Ablauf der Grafikpipeline wird durch die aus [Krö01a] entnommene Abbildung 2.5 dargestellt.

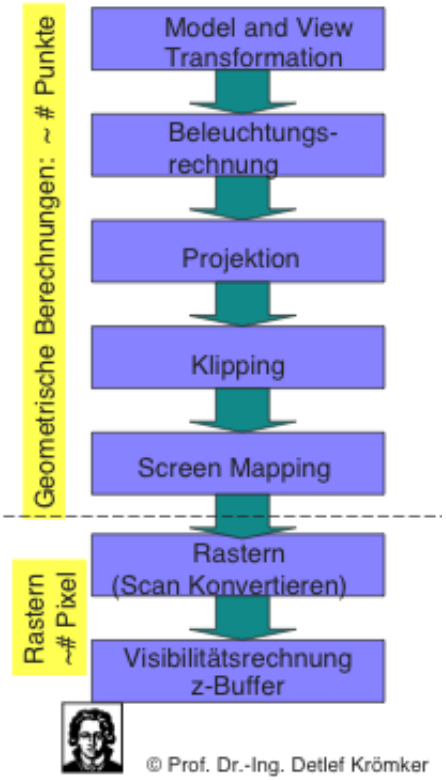


Abbildung 2.5: Die Grafikpipeline

2.3.2 Die Modell- und Sicht-Transformation

Der erste Schritt der Grafikpipeline besteht darin, die Eckpunkte der einzelnen Objekte in ein einheitliches Koordinatensystem zu bringen. Die Objekte befinden sich alle in unterschiedlichen lokalen Koordinatensystemen. Dies wird als Modell-Koordinatensystem bezeichnet. Damit die Objekte durch die so genannten Weltkoordinaten beschrieben werden können, müssen je nach Objekt bestimmte Transformationen durchgeführt werden. Die einfachste Transformation ist dabei die Translation, da es sich hierbei um eine triviale Addition von Vektoren handelt. Weitere Transformationen wären Skalieren, also die Variation der Größe eines Objekts, sowie die Spiegelung und die Rotation.
Diese Modell-Transformationen lassen sich mathematisch in einer Transformationsmatrix zusammenfassen, welche unter anderem auch in OpenGL verwendet wird.

Mit der Sicht-Transformation ist es möglich, den Standpunkt des Betrachters festzulegen, um ihn im Koordinatenursprung zu positionieren und entlang der negativen z-Achse in die Szene schauen zu lassen. Der Name deutet an, dass diese Matrix eine Transformation der Szenerie, sowie des Betrachters nach sich zieht. Als Ergebnis der Sicht-Transformation erhält man so genannte Aug-Koordinaten. Die virtuelle Welt wurde so gedreht und verschoben, dass der Betrachter im Koordinatenursprung steht. (siehe Abbildung 2.6).

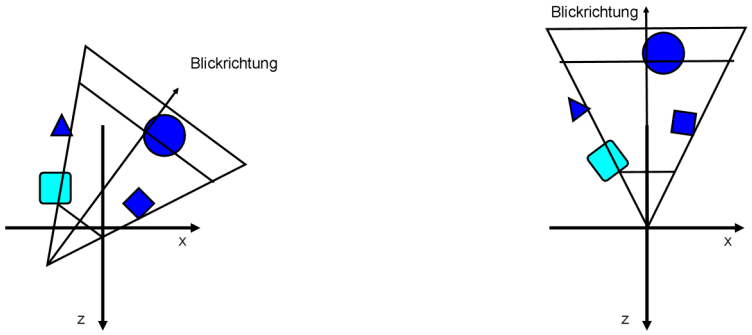


Abbildung 2.6: Die Sicht-Transformation

2.3.3 Beleuchtungsberechnung

Es gibt drei Arten von Lichtquellen: Das Punktlicht, das in alle Richtungen gleichmäßig strahlt, das kegelförmige Licht, das nur in einem gewissen Öffnungswinkel strahlt, und das Umgebungslicht, das keine Richtung hat. Wichtig für die Berechnung der Beleuchtung sind auch die verschiedenen Materialeigenschaften, die angeben, wie die Oberflächen auf einfallendes Licht reagieren. Diese Werte geben an, ob Licht diffus oder glänzend reflektiert wird oder ob es gar absorbiert wird.

Die Berechnung der Beleuchtung kann auf zwei Arten geschehen.

Gouraud-Schattierung

Beim „Gouraud-Shading“ werden die Farbwerte des Polygons an seinen Eckpunkten bestimmt. Es wird ein Normalen-Vektor aus dem Mittelwert der Flächennormalen der anliegenden Flächen erzeugt. Der Winkel zwischen diesem und dem Strahl zur Lichtquelle bestimmt mit dem diffusen Reflexionskoeffizienten des Materials, der Farbe der Fläche und des Lichts den Farbwert für den Eckpunkt.
Danach wird das Polygon zeilenweise abgetastet, wobei die Farbwerte, für die Schnittpunkte der Abtastlinie mit den Kanten des Polygons, aus den interpolierten Farbwerten der Eckpunkte berechnet werden. Die Farbwerte zwischen den Schnittpunkten der Abtastlinie werden aus den interpolierten Farbwerten der Kanten berechnet.

Mit diesem Verfahren ist es aber lediglich möglich, matte Objektoberflächen darzustellen. Solch schattierte Objekte erhalten eine plastikähnliche Erscheinung. Es ist nicht möglich, Transparenz, Schatten oder Spiegelungen darzustellen.
Dafür ist dies eines der schnellsten Verfahren zur Darstellung dreidimensionaler Objekte.

Phong-Schattierung

Beim „Phong-Shading“ werden wie beim „Gouraud-Shading“ ebenfalls die Normalenvektoren an den Eckpunkten des Polygons bestimmt.
Anschließend werden aber die Normalen-Vektoren über die Kanten, sowie über die Abtastlinien interpoliert. Damit ist diese Schattierung ein Per-Pixel-Verfahren, was bedeutet, dass die Beleuchtungsgleichung für jeden Pixel einzeln berechnet wird. Somit können weiche Farbverläufe und Glanzpunkte dargestellt werden, wodurch ein wesentlich realistischerer Eindruck erzeugt wird.
Allerdings ist der Rechenaufwand im Vergleich zum „Gouraud-Shading“ erheblich höher.

2.3.4 Projektion

Bei diesem Verarbeitungsschritt wird die dreidimensionale Szene auf die Bildebene abgebildet. Es handelt sich dabei um eine Projektion in die zweite Dimension. Es gibt zwei Grundformen der Projektion.

Parallelprojektion

Bei der Parallelprojektion befindet sich der Augpunkt in unendlicher Entfernung zur Projektionsebene. Die Projektionsstrahlen verlaufen parallel zur Projektionsrichtung und schneiden die Projektionsebene an den Stellen, wo sich die projizierten Punkte dann im zweidimensionalen Bild befinden. Das Ergebnis ist eine verzerrungsfreie Darstellung von dreidimensionalen Objekten. Allerdings werden Objekte, die unterschiedlich weit von der Projektionsebene entfernt sind, immer gleich groß dargestellt.

Zentralprojektion

Die Zentralprojektion funktioniert wie eine virtuelle Lochkamera. Die Projektionsstrahlen müssen sich im Augpunkt schneiden, wodurch eine Sichtpyramide entsteht. Die Schnittpunkte mit der Projektionsebene sind wiederum die Punkte in der zweidimensionalen Darstellung.
Die Zentralprojektion ist eine realitätsnähere Darstellung, da die perspektivische Verkürzung von Objekten mit zunehmender Entfernung berücksichtig wird.

2.3.5 Klippen

Das Klippen bezeichnet ein Verfahren, bei dem Objekte beziehungsweise Teile von Objekten, die außerhalb des darzustellenden Bereichs liegen, entfernt werden. Der darzustellende Bereich wird als Sichtvolumen bezeichnet und ist bei der Parallelprojektion ein Quader, der in Projektionsrichtung unendlich ausgedehnt ist. Bei der Zentralprojektion ist das Sichtvolumen eine Pyramide. Das Sichtvolumen wird durch eine vordere und hintere Ebene, die parallel zur Projektionsebene liegen, in der Tiefe beschränkt.
Die Objekte können beim Klippen auf drei Arten behandelt werden. Liegt das Objekt im darzustellenden Bereich, wird es gezeichnet. Liegt es außerhalb, wird es entfernt. Wenn

es nur teilweise im darzustellenden Bereich liegt, müssen neue Eckpunkte am Rand der Sichtpyramide berechnet werden.

2.3.6 Screen Mapping

Im letzten Schritt der geometrischen Berechnung werden die Weltkoordinaten auf die Bildschirmkoordinaten abgebildet.

2.3.7 Rasterisierung

Die Rasterisierung ist die Umwandlung graphischer Primitive wie Linie, Dreieck oder Polygon in Pixel. Beispielsweise wird zum Umwandeln von Linien in Rasterpunkte der Bresenham-Algorithmus benutzt, da hierbei nur Integer-Arithmetik und schnelle Addition nötig ist. Beliebige Polygone lassen sich mit dem Scan-Linien-Algorithmus in ein Rasterbild umwandeln.

2.3.8 Sichtbarkeitsberechnung

Die einzelnen Rasterpunkte werden auf ihre Sichtbarkeit hin untersucht. Dafür hat jedes Pixel neben seinem Farbwert einen zusätzlichen Wert, der seine Position in der Tiefe speichert. Dies ist der so genannte z-Wert. Dieser z-Wert wird im z-Speicher gespeichert. Jedes Mal, wenn ein Pixel gezeichnet werden soll, wird sein z-Wert mit dem Wert verglichen, der an der korrespondierenden Position im z-Speicher abgelegt ist. Ist der neue Wert kleiner als der vorhandene, befindet sich das Pixel näher am Betrachter und wird gezeichnet. Der Wert im z-Speicher wird mit dem neuen Wert aktualisiert.

2.4 Programmierung der GPU

Die heutige Generation von Grafikkarten erlaubt den Austausch von Teilen der Grafikpipeline durch eigene Programme. Diese selbst geschriebenen Programme laufen auf den frei programmierbaren Grafikprozessoren. Dabei hat man die Wahl zwischen dem Vertex-Prozessor und dem Fragment-Prozessor.

Der Vertex-Prozessor

Mit einem auf dem Vertex-Prozessor laufenden Programm ist es möglich, die Attribute der „Vertices“ zu verändern. Das Programm ersetzt den Transformations- und Beleuchtungsteil der fixen Grafikpipeline und übernimmt dabei folgende Aufgaben:

- Koordinatentransformation
- Normalentransformation und Normalisierung
- Erstellung und Transformation von Texturkoordinaten
- Beleuchtung
- Materialfarben

Der Fragment-Prozessor

Mit einem auf dem Fragment-Prozessor laufenden Programm können Fragmente manipuliert werden. Ein Fragment bezeichnet dabei die Vorstufe eines Pixels. Für solch ein Fragment liegen Informationen über Texturkoordinaten, Farbe oder Tiefe vor. Diese Informationen werden aus den entsprechenden Werten der Eckpunkte interpoliert. Es ist für den Fragment-Prozessor nicht möglich, die x,y-Position der einzelnen Fragmente zu verändern. Er hat nur Zugriff auf den z-Wert. Außerdem wird jedes Fragment einzeln behandelt, was bedeutet, dass keinerlei Informationen über benachbarte Pixel verfügbar sind. Der Fragment-Prozessor kann lesend auf gebundene Texturen zugreifen. Die Ausgaben des Fragment-Programms können in mehr als eine Textur umgeleitet werden. Dieses Verfahren nennt sich „Multiple-Render-Targets“.
Die Aufgaben sind:

1. Texturenzugriff
2. Texturenanwendung
3. Nebel
4. Berechnung der Farbsumme

2.4.1 GPGPU

Durch die freie Programmierbarkeit der Grafikkarte hat sich ein Bereich entwickelt, der Berechnungen von nicht graphischen Algorithmen auf der Grafikkarte durchführt. Dies nennt sich General Purpose Computations on GPUs, kurz GPGPU. Die Leistungsfähigkeit von Grafikkarten wird in Bereichen wie Numerische Simulationen, Verfahren aus der Bildanalyse oder aber auch für komplexe Datenbankoperationen ausgenutzt.

Damit Berechnungen auf der Grafikkarte durchgeführt werden können, wird eine Rahmenanwendung benötigt, die entsprechende Funktionsaufrufe der Grafikkarte tätigt. Das bedeutet, dass von der Rahmenanwendung Fragments erzeugt werden müssen, damit der geschriebene Fragment-Shader überhaupt ausgeführt werden kann. Dies geschieht durch das Rendern einen bildschirmfüllenden Rechtecks. Die Rahmenanwendung versorgt die Grafikkarte zusätzlich mit den nötigen Daten, dem Shader selbst, und kümmert sich um die Verarbeitung des Ergebnisses.

Die GPU als Datenstrom-Prozessor

Die GPU wird nach [Pur04] wie ein Datenstrom-Prozessor behandelt. Solch ein Stream-Prozessor führt immer die gleichen Anweisungen auf einer Menge von Eingangsdaten aus. Diese Anweisungen sind dabei das Shader-Programm und werden als Kernel bezeichnet. Ein Beispiel aus der Bildbearbeitung wäre die Anwendung von Filtern auf ein Bild. Die Pixel des Bildes beschreiben den Strom der Eingangsdaten, die die Kernel (Filter) nacheinander passieren. (siehe Abbildung 2.7)

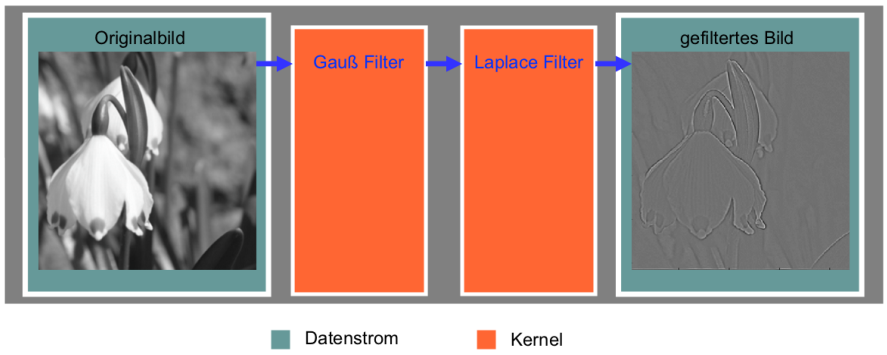


Abbildung 2.7: Beispiel für einen Stream-Prozessor

Für die Grafikkarte ergibt sich daraus, dass die Texturen den Eingabestrom darstellen und das Fragment-Programm den Kernel.
Anhand der Abbildung 2.8, aus [LBOS04], wird das Verfahren verdeutlicht.

- Es wird ein bildschirmfüllendes Rechteck gezeichnet
- Der Kernel wird über jedes Fragment ausgeführt
- Der Eingabestrom wird aus den gebundenen Texturen gelesen
- Das Ergebnis wird in den Bildschirmspeicher geschrieben

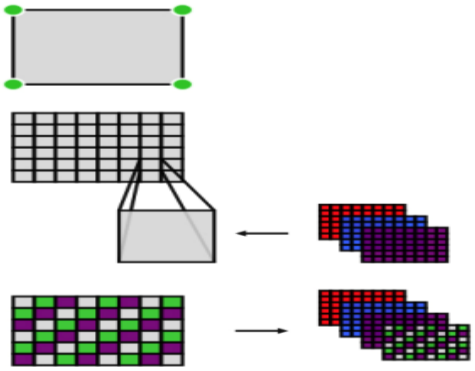


Abbildung 2.8: Der Streamprozessor

Das Rechteck ist der Datenstromgenerator. Da in jedem Texel ein Strahl codiert ist, wird eine 1:1-Abbildung von Pixel auf Texel benötigt. Deshalb wird dieses Rechteck bildschirmfüllend gezeichnet.

2.4.2 Der Eingabestrom

Die Texturen, auf die der Fragment-Shader frei lesenden Zugriff hat, stellen den Eingabestrom dar. In diese Texturen müssen die benötigten Daten codiert werden. Texturen verhalten sich analog zu zweidimensionalen Feldern. Auf die einzelnen Feldeinträge kann mit zwei Koordinaten zugegriffen werden. Andere Datenstrukturen als zweidimensionale Felder müssen auf diese abgebildet werden.

Eine Bibliothek, die andere Datenstrukturen auf solch zweidimensionale Felder abbildet, ist Glift [LSK+05].
Glift wurde unter anderem von Aaron E. Lefohn entwickelt und bietet Iteratoren an, die entsprechende Adressübersetzungen durchführen.

> „A data structure abstraction for graphics processing units (GPUs) can simplify the description of new and existing data structures, stimulate development of complex GPU algorithms, and perform equivalently to hand-coded implementations."

Diese Bibliothek wird aber in dieser Arbeit nicht verwendet.

2.4.3 Die Programmierschnittstelle

Für die Programmierung der Grafikkarte wird eine API (Application Programming Interface) benötigt. Eine API ist eine vereinheitlichte Programmierschnittstelle zwischen unterschiedlichsten Grafikkarten und der Applikation und bietet damit eine Abstraktion der Hardware an. Im Wesentlichen stehen zwei APIs zur Verfügung, OpenGL (Open Graphics Library) und Direct3D.

OpenGL

OpenGL bietet circa 250 Befehle und ist plattformübergreifend, wodurch es im professionellen Bereich nach wie vor führend ist. Entstanden ist OpenGL aus IrisGL von Silicon Graphics Inc. Die Spezifikationen der API wurden 1992 fertiggestellt und 1993 implementiert. Zusammen mit weiteren Herstellern von Grafikkarten wollte Silicon Graphics OpenGL als offenen Standard etablieren. Die Weiterentwicklung von OpenGL wird durch ein Gremium, dem ARB (OpenGL Architecture Review Board), überwacht und vorangetrieben. OpenGL wurde auf einer Vielzahl von Systemen implementiert, so dass es unter anderem auf dem Macintosh, PCs und Unix-basierten Systemen zur Verfügung steht.

Direct3D

Die andere API nennt sich Direct3D und ist eine proporietäre Schnittstelle von Microsoft, die im Bereich von Computerspielen sehr beliebt ist. Direct3D ist nur auf Betriebssystemen aus dem Hause Microsoft lauffähig.

GroIMP ist „Open Source", in Java geschrieben und auf jedem Betriebssystem lauffähig, für das es eine JavaVM gibt. Für die Live-Ansicht der Szene wird OpenGL bereits verwendet, wodurch die Wahl der API für die Programmierung des Strahlverfolgers auf OpenGL fiel.

2.4.4 Java und OpenGL

Die Verwendung von OpenGL-Befehlen mittels Java wird mit der externen OpenGL-Programmierbibliothek „ jogl“ möglich. Jogl stellt Java-Wrapperklassen bereit, die Schnittstellen zu den nativen OpenGL-Funktionen liefern. Mit „ jogl“ besteht die Möglichkeit zur plattformübergreifenden Grafikprogrammierung ohne spezielle Anpassungen an die verschiedenen Plattformen. Dabei gliedert sich „ jogl“ in bestehende „Swing“- und „AWT“-Komponenten ein, wodurch Java-Kenntnisse weiter verwendet werden können. Jogl stellt das Grundgerüst für die Programmierung der Grafikprozessoren bereit. Mit den „ jogl“-Funktionen werden die Texturen erzeugt, Bildschirmpuffer angelegt und die auf den Grafikprozessoren laufenden Programme erstellt und gestartet.

2.4.5 Shader-Sprache

Damit die feste Grafikpipeline durch „kleine“ selbst geschriebene Programme ersetzt werden kann, wird eine Programmiersprache benötigt.

> „The OpenGL code that is intended for execution on one of the OpenGL programmable processors is called a SHADER.“ [Ros05]

Gemäß dieser Definition wird im Weiteren von einem Shader gesprochen.

In den Anfängen der Shader-Programmierung wurde der Programmcode in Assembler geschrieben. Im Jahre 2004 wurde die OpenGL 2.0 Spezifikation veröffentlicht, mit der es möglich wurde, Hochsprachen für die Programmierung zu verwenden. Die Hochsprache, mit der Shader für OpenGL programmiert werden, nennt sich OpenGL Shading Language (GLSL oder glSlang). Entstanden ist GLSL aus der Programmiersprache C. Zu den üblichen Datentypen gibt es spezielle, für 3D Operationen typische Datentypen wie Vektoren und Matrizen. Die Sprache bietet die Möglichkeit zur Verwendung von Schleifen, Funktionen und bedingten Verzweigungen.
Unterschiede zu C wären das Fehlen von Datentypen wie Pointer, Strings oder Charakter, sowie Bedingungsblöcke. Begründet wird dies mit der vorrangigen Verarbeitung von numerischen Daten. Es gibt keinen Bedarf für solche Bestandteile, die die Sprache unnötig komplizieren würden.

2.4.6 Ausführen der Shader

Damit ein Shader ausgeführt werden kann, muss er vorher kompiliert und gelinkt werden. Dies geschieht während der Laufzeit der OpenGL-Anwendung. Der Quelltext eines Shaders ist ein Feld, bestehend aus einzelnen Strings. Dieser kann als separate Datei vorhanden sein oder aber direkt im Code der OpenGL-Anwendung erstellt werden. Das Erzeugen eines individuellen GLSL-Codes ist somit möglich und wird bei der Implementierung des Strahlverfolgers genutzt.

Wenn ein eigener Fragmentshader benutzt wird, ist es nicht zwingend nötig, auch einen eigenen Vertexshader zu nutzen. Die feste Grafikpipeline übernimmt dann wieder die in 2.4 beschriebenen Aufgaben.

2.4.7 Beschränkungen

Die Shader-Programme sind in der Anzahl an ausführbaren Instruktionen beschränkt. Dabei wird in statische und dynamische Instruktionen unterschieden.
Die Anzahl der statischen Instruktionen gibt an, wie viele Befehle im Programm stehen, wenn es kompiliert wird. Die dynamische Anzahl spiegelt die Zahl der tatsächlich ausgeführten Befehle wider. Diese Zahl kann durch die Verwendung von Schleifen und Funktionsaufrufen weitaus höher sein.
Die Tabelle 2.1 aus der NVIDIA General FAQ[1] gibt einen Überblick über die Beschränkungen der anfangs benutzten Geforce Fx 5200 und der aktuell verwendeten Grafikkarte der Geforce6 Serie.

GPU	Vertexshader	Fragmentshader
GeForce FX	256 statische Instruktionen 65,535 dynamische Instruktionen	512 statische Instruktionen 512 dynamische Instruktionen
GeForce6	512 statische Instruktionen 65,535 dynamische Instruktionen	2048 statische Instruktionen 65,535 dynamische Instruktionen

Tabelle 2.1: Beschränkungen der Grafikkarten

2.4.8 Fazit

Pathtracing erlaubt diffuse Beleuchtung an Objekten, deren Oberflächen nicht spiegelnd sind. Dies wird dadurch erreicht, dass immer nur ein Strahl rekursiv weiter verfolgt wird, nachdem ein Objekt getroffen wurde. Dies lässt sich durch die Verwendung eines Fragment-Shaders exakt umsetzen, da jedes Fragment einzeln berechnet wird und in jedem Fragment ein Strahl codiert ist. Nach dem Treffen eines Objekts wird der Sekundärstrahl erzeugt und für einen weiteren Durchgang gespeichert.

Die OpenGL-API ist unabhängig vom verwendeten Fragment-Prozessor. Sie skaliert bei unterschiedlicher Hardware automatisch. Das bedeutet, dass bei der Verwendung einer moderneren Grafikkarte, die mehrere oder schnellere Fragment-Prozessoren besitzt, die Berechnung schneller ausgeführt wird.

[1] http://developer.nvidia.com/object/General_FAQ.html, Stand Januar 2008

3 Entwicklung der eigenen Lösung

3.1 Aufbau des Strahlverfolgers

In diesem Kapitel werden die Vorüberlegungen präsentiert, die nötig waren, um den Strahlverfolger auf der Grafikkarte zu implementieren.
Als erstes muss überlegt werden, welche Eingabedaten vorliegen beziehungsweise aus GroIMP extrahiert werden müssen, um sie in einen Datenstrom umzuwandeln. Weiterhin werden hier die Kernel beschrieben, die für den Strahlverfolgungs-Algorithmus benötigt werden.
Die Eingabedaten wären die Position der virtuellen Kamera und deren Parameter, die die Bildebene aufspannen, die einzelnen Strahlen, die durch die Pixel der Bildebene gehen, und eine Beschreibung der Szene. Darin enthalten sind Informationen über die Geometrie und die Oberflächen der darzustellenden Objekte.
Parameter wie die Anzahl der Strahlen pro Pixel, die Rekursionstiefe oder die Verwendung des Pathtracing-Verfahrens werden in einem Menüeintrag in GroIMP gesetzt.

3.1.1 Texturen als Speicher

Wie in 2.4.2 bereits erwähnt, stellen die Texturen den Eingabestrom für den Datenstrom-Prozessor dar. Normalerweise enthalten Texturen Bilder, die mittels „Texture-mapping“ auf die Objekte gelegt werden, um dieses detaillierter darzustellen. Mit Texturen ist es unnötig, feine Details eines Objekts zu modellieren. Die verwendeten Texturen für den Strahlverfolger enthalten aber Datenstrukturen, die den zu verfolgenden Strahlen und den Szenenobjekten entsprechen.
Auf die einzelnen Feldeinträge der Texturen wird durch „Texture-Look-ups“ direkt zugegriffen. Jede Position einer Textur speichert einen RGBA-Wert. Dabei gibt es unterschiedliche Arten von Texturen, mit unterschiedlicher Genauigkeit. Bei dieser Arbeit wurden Texturen verwendet, die pro Pixel vier 32Bit-Fließkommawerte speichern.
Somit ist es nötig, die vorhandenen Datenstrukturen auf die Texturen abzubilden, damit dem Shader-Programm diese dann zur Verfügung stehen.

Aufteilung der Daten in Texturen

Für jedes Fragment, das bearbeitet wird, müssen folgende Daten vorhanden sein: der zu testende Strahl, seine Intensität und der akkumulierte Farbwert des Pixels. Diese Daten werden benutzt, verändert und müssen für den nächsten Renderdurchgang wieder zur Verfügung stehen. Geometrisch gesehen ist der Strahl eine Halbgerade. Er hat

einen Ursprung und einen parametrisierten Richtungsvektor. Dies sind zwei dreidimensionale Vektoren, also sechs Werte. Der Farbwert und die Intensität des Strahls sind RGBA-Werte, was acht Werten entspricht. Insgesamt werden also 14 Werte pro Fragment bearbeitet.
Das Problem ist, dass bei aktuellen Grafikkarten nur maximal vier Puffer als Ziel angegeben werden können. Das heißt, es werden höchstens 16 Werte pro Fragment geschrieben. Dadurch müssen die 14 Werte auf diese vier Texturen aufgeteilt werden. Eine weitere Einschränkung ist, dass es nicht möglich ist, gleichzeitig lesend und schreibend auf eine Textur zuzugreifen. Deshalb werden acht Texturen benötigt. Zusätzlich sind noch zwei Texturen für die Szenendaten notwendig.
Insgesamt werden also zehn Texturen verwendet.

Vier Texturen (A_0 - A_3) werden als Eingabe-Texturen bezeichnet und weitere vier Texturen (B_0 - B_3) als Ausgabe-Texturen.
Damit der Kernel Zugriff auf die Eingabe-Texturen hat, werden diese mittels „Texturemapping“ auf das bildschirmfüllende Rechteck gelegt.
Das Ergebnis der Berechnung durch die Kernel wird in die vier Ausgabe-Texturen umgeleitet.

Die wichtigste Textur ist A_0. In dieser werden die errechneten Farbwerte der Pixel akkumuliert. Diese Textur enthält nach Beendigung des ganzen Verfahrens das synthetisierte Bild.

In Textur A_1 und A_2 werden die generierten Strahlen abgelegt.
Der Ursprung wird als homogener Vektor in A_1 und der Richtungsvektor in A_2 abgespeichert.

Die letzte Textur A_3 speichert den Betrag, den der Lichtstrahl zur Beleuchtung der Szene beisteuert. Dieser Betrag nimmt bei jedem Rekursionsschritt in Abhängigkeit vom getroffenen Material ab.
Die von GroIMP bereitgestellten Szenendaten werden in einer separaten Textur gespeichert. Diese ist sehr viel größer als die Ein-/ Ausgabe-Texturen. Die Verwendung der zehnten Textur wird näher in 3.3.2 erläutert.
In Abbildung 3.1 wird verdeutlicht, wie die Texturen organisiert sind.

3.1.2 Framebuffer-Objekt

Ein normaler Render-Durchgang schreibt sein Ergebnis in den Bildschirmspeicher. Dieser kann dann als Bild auf dem Monitor ausgegeben werden. Damit die Daten wieder zur Verfügung stehen, müsste das Ergebnis aus dem Bildschirmspeicher zurück in eine Textur kopiert werden. Dies ist aber ein sehr langwieriges Verfahren.
Es ist aber auch möglich, die Ausgabe in eine Textur umzuleiten. Dieses Verfahren nennt sich „render to texture“, und geschieht durch die Verwendung eines „Framebuffer-Objektes“, kurz FBO. An solch ein FBO können aktuelle Grafikkarten bis zu vier Spei-

cher binden, in die dann gleichzeitig gezeichnet wird. Die bei dieser Arbeit verwendete GeForce6 ist ebenfalls in der Lage, vier Texturen an das „Framebuffer-Objekt“ zu binden.

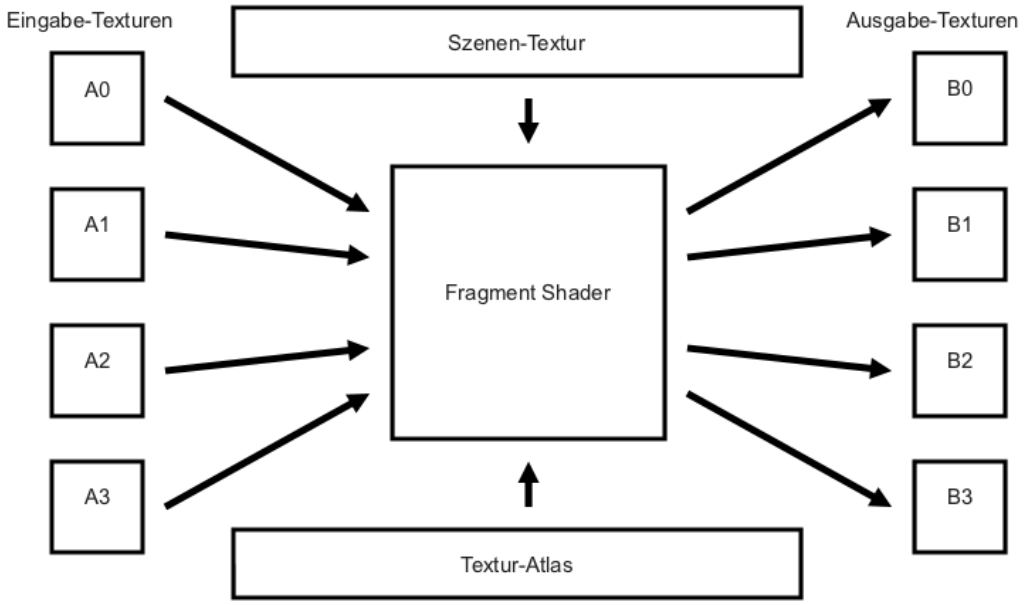


Abbildung 3.1: Organisation der Texturen

3.1.3 Ping-Pong-Technik

Dominik Göddeke beschreibt in seinem „Basic Math Tutorial[1]“ die Ping-Pong-Technik.

> „Ping pong is a technique to alternately use the output of a given rendering pass as input in the next one.“

Diese Technik wird in dieser Arbeit für die rekursive Strahlverfolgung benutzt. Die rekursive Strahlverfolgung geschieht dadurch, dass auf den vorherigen Farbwert der neue Farbwert des weiterverfolgten Sekundärstrahls dazu addiert wird. Das geschieht solange, bis eine eingestellte Rekursionstiefe erreicht worden ist.

[1] http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html; Stand Januar 2008

Das Ergebnis eines Render-Durchgangs befindet sich in den vier Ausgabe-Texturen. Die Werte in den Eingabe-Texturen werden nicht mehr gebraucht. Deshalb kann die Rolle der Texturen vertauscht werden. Somit sind die Ergebnisse des ersten Render-Durchgangs die Eingangsdaten des nächsten Render-Durchgangs.
Dieses Vertauschen kann auf drei unterschiedliche Arten erfolgen:

1. Es werden zwei FBOs benutzt. Die Ausgabe-Texturen der FBOs sind die Eingabe-Texturen des jeweils anderen FBOs. Bei jedem Render-Durchgang wird der FBO getauscht.

2. Es wird nur ein FBO benutzt, an das bei jedem Render-Durchgang die Ausgabe-Texturen neu gebunden werden.

3. Es wird ein FBO benutzt, an das mehrere Texturen gebunden sind, zwischen denen mit dem Befehl „glDrawBuffer()“ gewechselt wird.

Bei dieser Arbeit wurde die erste Variante gewählt.

3.1.4 Die verwendeten Kernel

Der Strahlverfolger ist als ein Fragment-Shader-Programm implementiert, da der Vertexshader nicht in mehrere Ausgabetexturen schreiben kann und ein Geschwindigkeitsvorteil besteht. Laut NVIDIA ist die Ausführung der Fragment Shader um den Faktor 10 schneller. [NVI06a]

Damit exakt kontrolliert werden kann, welche Daten berechnet oder aus der Textur genommen werden, wird eine spezielle Projektion benutzt, die die dreidimensionale Welt auf den zweidimensionalen Bildschirm abbildet. Zusätzlich wird eine 1:1-Abbildung zwischen den Pixeln, die gezeichnet werden sollen, und den Texeln, aus denen die Daten genommen werden, benötigt. Dies geschieht durch eine orthogonale Projektion. Die geometrischen Koordinaten liegen somit an exakt der gleichen Position wie die Texturkoordinaten.
Dementsprechend wird mittels OpenGL ein bildschirmfüllendes Rechteck gezeichnet, auf das die vier Eingabe-Texturen gelegt werden. Beim Zeichnen wird der Fragment-Shader für jedes Pixel dieses Rechtecks aufgerufen und führt das selbstgeschriebene Fragment-Programm aus.
Hier nun ein Überblick über die geschriebenen Kernel und deren Funktion:

Initial Kernel

Der Initial Kernel ist dafür zuständig, dass alle RGBA-Werte der vier Ausgabe-Texturen auf null gesetzt werden. In den Alpha-Kanal der A_0-Textur wird der Startwert für den Zufallsgenerator gesetzt. Dies wird näher in 4.1.2 erläutert.

Erzeugung der Strahlen

Ein Kernel ist für die Erzeugung der Primärstrahlen zuständig. Über eine globale Variable, die vom Rahmenprogramm an den Shader übergeben wird, werden die benötigten Kameraparameter diesem zur Verfügung gestellt. Solch eine globale Variable nennt sich „uniform“ Variable, sie kann im Vertex- und Fragment-Shader gelesen, aber nicht geschrieben werden.
Die „up“ und „right“ Vektoren der Kamera bestimmen die Maße der Bildebene. Mit der Höhe und Breite des zu erzeugenden Bildes wird die Bildebene in entsprechend große Pixel unterteilt, durch die jeweils ein Strahl generiert wird. Dieser Kernel benötigt von den Eingabe-Texturen nur A_0. Es findet lediglich ein Kopiervorgang der in dieser Textur enthaltenen RGBA-Werte in die Ausgabe-Textur statt. In B_1 wird der Ursprung des Strahls gespeichert und in B_2 die Richtung. Jedes Mal, wenn dieser Kernel ausgeführt wird, setzt er B_3 wieder zurück auf den Wert 1.0, was 100% Strahlintensität bedeutet.

Verfolgung der Strahlen

Nach der Ausführung des Kernels für die Strahlgenerierung findet der eigentliche Strahlverfolgungsalgorithmus statt. In den Eingabe-Texturen stehen dem Kernel die Farbe des vorherigen Render-Durchgangs zur Verfügung, sowie Ursprung und Richtung der zu verfolgenden Strahlen. Wenn ein Schnittpunkt zwischen Objekt und Primärstrahl existiert, wird der lokale Anteil des einfallenden Lichts berechnet. Der berechnete Farbwert wird zu dem in A_0 gespeicherten Farbwert addiert und in B_0 gespeichert. Für die Berechnung der Reflexion und Refraktion wird an dem gefundenen Schnittpunkt ein Sekundärstrahl erzeugt. Dieser Sekundärstrahl wird in die Ausgabe-Texturen B_1 und B_2 gespeichert. Der Betrag des Lichtstrahls nimmt je nach Materialeigenschaft des getroffenen Objekts ab. Der neue Wert wird nach B_3 gespeichert.

3.1.5 Beschreibung eines Render-Durchgangs

Die Reihenfolge der Kernel-Ausführungen ist in Abbildung 3.2 dargestellt.
Als erstes ist es nötig, die Texturen in einen initialen Zustand zu versetzen. Dies geschieht mit der Initialisierung der Ausgabe-Texturen mit Null. Dieser Kernel wird nur ein einziges Mal ausgeführt.
Nach jeder Ausführung eines Kernels werden die Ein-/ Ausgabe-Texturen miteinander vertauscht. Dem Primärstrahl-Kernel stehen also die initialisierten Texturen als Eingabe-Texturen zur Verfügung. In einer äußeren Schleife wird der Primärstrahl-Kernel ausgeführt und erzeugt für jeden Pixel ein Bündel an Strahlen. Bei jeder Ausführung dieses Kernels wird der erzeugte Strahl ein wenig verändert. Die Richtung des Strahls wird stochastisch auf der Fläche des Pixels verteilt.
Innerhalb dieser Schleife läuft eine weitere Schleife, deren Zähler die Tiefe der Rekursion des Strahlverfolgungsalgorithmus angibt. In dieser Schleife wird der Strahlverfolgungsalgorithmus mehrfach ausgeführt und die Sekundärstrahlen rekursiv durch die Szene weiterverfolgt.

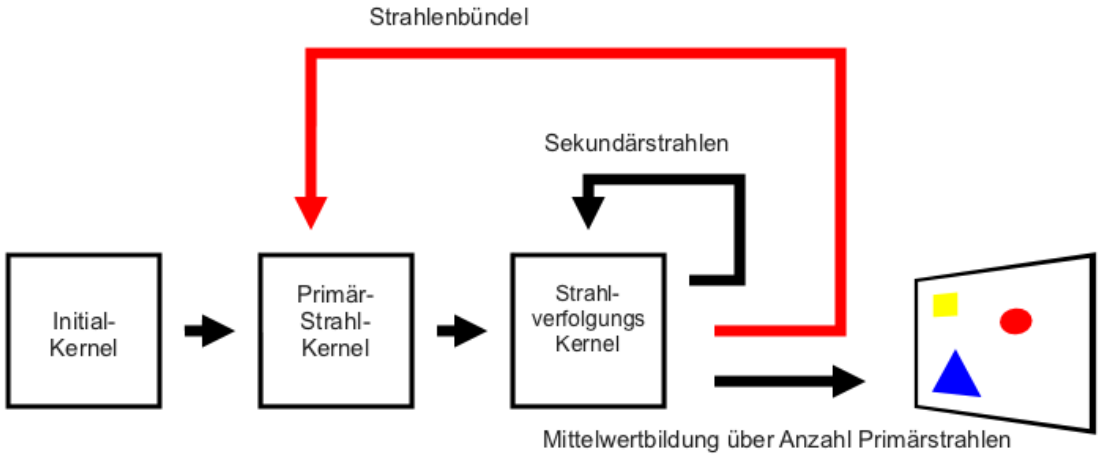


Abbildung 3.2: Ausführungsreihenfolge der Kernel

Im ersten Render-Durchgang befinden sich in A_1 und A_2 Ursprung und Richtung der Primärstrahlen. Die Pixel in A_0 sind alle noch schwarz, da sich die Textur im Initialzustand befindet. Die Intensität des Strahls beträgt 100%. Der Strahlverfolgungs-Kernel wird für jedes Pixel ausgeführt und nimmt den gespeicherten Strahl aus den Texturen, um ihn mit den Objekten der Szene zu testen. Der Schnittpunkt mit einem Objekt stellt den Ursprung des Sekundärstrahls dar und wird in B_1 gespeichert. Je nach Materialeigenschaft wird eine Richtung des Sekundärstrahls generiert und in B_2 gespeichert. Der Restbetrag des Strahls wird in B_3 gespeichert. In B_0 wird zu dem Initialwert von Null der berechnete Farbwert am Schnittpunkt dazu addiert. Nach der Ausführung des Kernels werden die Texturen vertauscht.
Der Schleifenzähler der inneren Schleife zählt um eins hoch und führt den Strahlverfolgungsalgorithmus erneut aus. Diesmal werden aber nicht die Primärstrahlen auf Schnittpunkte getestet, sondern die im vorherigen Schritt erzeugten Sekundärstrahlen. Diese werden verfolgt und liefern möglicherweise einen Beitrag zum vorher berechneten Farbwert.
Nach dem Erreichen einer festgelegten Rekursionstiefe zählt die äußere Schleife einen Wert weiter und führt den Primärstrahl-Kernel erneut aus. Dieser erzeugt neue Primärstrahlen, setzt B_3 wieder auf 100% und kopiert das erzeugte Bild von A_0 nach B_0. Nach Beendigung der äußeren Schleife muss der Mittelwert über die aufsummierten Bilder berechnet werden.

3.2 Die Szenentextur

Der Strahl benötigt, für den Test auf Schnittpunkte, Informationen über die in der Szene enthaltenen Objekte. Diese Informationen sollen in einer Textur gespeichert werden, auf die das Shader-Programm wahlfreien Zugriff hat.

3.2.1 Organisation der Objekte

Eine Textur ist als ein zweidimensionales Feld vorstellbar. Mit zwei Koordinaten ist es möglich, jeden beliebigen Eintrag dieses Feldes auszuwählen und vier Werte in Form eines Pixels aus der Textur zu lesen. Dies nennt sich „Texture-Lookup“. Diese beiden Koordinaten bilden somit eine Adresse innerhalb der Textur, analog zu einem Zeiger, der den Arbeitsspeicher adressiert.
Die Idee ist, dass geometrische Primitive wie Kugel, Box oder Zylinder durch mathematische Formeln beschrieben werden können. Eine Kugel um den Mittelpunkt (a, b, c) und den Radius r kann durch die folgende Gleichung dargestellt werden.

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$$

Da in diese Formel der zu testende Strahl eingesetzt wird, benötigt der Algorithmus die Werte für die Parameter a, b, c und r. Für die anderen Primitive verhält es sich analog. Die benötigten Parameter werden aus GroIMP ausgelesen und an entsprechender Stelle in die Textur eingefügt. Zusätzlich besitzen die Objekte noch eine Transformationsmatrix und einen Material-Shader, die ebenfalls in die Textur gespeichert werden.

3.2.2 Beliebig orientierte Objekte

Die Objekte werden mathematisch definiert, um sie auf Schnittpunkte hin untersuchen zu können. Dabei besteht die Möglichkeit, die Objekte entweder an einer aktuellen Position zu definieren oder aber im Koordinatenursprung.
Dementsprechend gibt es zwei Möglichkeiten, beliebig orientierte Objekte auf Schnittpunkte mit den Strahlen hin zu untersuchen. Entweder werden spezielle Schnittpunkt-Algorithmen für Schnittpunktberechnungen im Weltkoordinatensystem benutzt (siehe 3.2.1) oder der Strahl wird in das Objektkoordinatensystem transformiert.
Die zweite Möglichkeit erlaubt eine vereinfachte Berechnung der Schnittpunkte, da eine Kugel als Einheitskugel mit Radius 1 betrachtet werden kann. Deshalb wurde dieses Verfahren bei dieser Arbeit implementiert.
Die obige Formel vereinfacht sich dann auf die Einheitskugel:

$$x^2 + y^2 + z^2 = 1$$

Die Vorgehensweise ist folgende:
Es wird eine Transformation bestehend aus Skalierung, Rotation und Translation definiert, die das Objekt durchlaufen muss, um aus der Standardposition in die gewünschte Position zu kommen. Diese Transformation stellt GroIMP als eine Transformationsmatrix bereit. Für den Schnittpunkttest wird diese Transformationsmatrix invertiert und

mit den Strahlen multipliziert. Die so transformierten Strahlen werden dann mit dem Objekt in seiner Standardposition getestet. Die eventuell resultierenden Schnittpunkte werden durch Multiplikation mit der nicht invertierten Transformationsmatrix in ihre korrekten Positionen zurücktransformiert.

Der Vorteil ist, dass die Schnittpunktberechnung mit diesen Einheits-Objekten einfacher ist. Außerdem wird so Platz in der Textur gespart, da die Objekte innerhalb des Shader-Programms definiert werden. Beispielsweise muss der Ursprung der Kugel nicht mit in die Textur geschrieben werden. Einstellbare Parameter wie Länge, Höhe oder Radius, die die Objekte in ihrer Form von den Einheitsobjekten abweichen lassen, werden in die Transformationsmatrix integriert.

Die Transformationsmatrix

Die Transformationsmatrix ist die erste Information über ein Objekt, das in die Textur an entsprechender Stelle geschrieben wird. Direkt dahinter kommt die inverse Transformationsmatrix. Es wäre möglich, die inverse Matrix durch das Shader-Programm berechnen zu lassen. Dies soll aber nicht geschehen, da auf der Grafikkarte nur die Berechnungen für die Strahlverfolgung durchgeführt werden sollen.

Die Transformationsmatrix ist eine 4x4-Matrix, besitzt also 16 Einträge. Von dieser Matrix ist die unterste Zeile unwichtig, da sich diese Werte niemals ändern. Es ist daher sinnvoll, diesen Platz in der Textur zu sparen und im Shader-Programm durch die entsprechenden Werte zu ergänzen.

Es werden also zwei 4x3-Matrizen gespeichert, die sechs Pixel in der Textur belegen.

Material-Shader

In der einfachsten Form sind die Informationen über das Material ein einfacher RGBA-Wert, der einen Pixel Speicher benötigt. Weitere Materialeigenschaften wie Angaben über glatte oder stumpfe Oberflächen werden nicht berücksichtigt.

Die Grundgröße eines jeden Objekts beträgt also sieben Pixel.

Die Kugel

Eine Kugel (siehe Formel 3.2.1) wird beschrieben durch ihren Ursprung und einen Radius. Die Einheitskugel befindet sich im Koordinatenursprung und hat einen Radius von eins. Die Werte für den Ursprung wären null und brauchen deshalb nicht in der Textur gespeichert zu werden. Es ist in GroIMP möglich, den Radius manuell einzugeben. Der entsprechende Wert wird in eine Skalierungsmatrix integriert und mit der normalen Transformationsmatrix multipliziert. Die Informationen bestehen so nur aus den zwei Matrizen und einer Materialfarbe.
Somit ist die Kugel sieben Pixel groß.

Die Box

Eine achsenparallele Box wird durch die Koordinaten zweier Eckpunkte definiert. Diese zwei Punkte geben die untere linke Ecke der Vorderseite und die obere rechte Ecke der Rückseite an. Die Seitenlänge der Einheitsbox beträgt immer 1, wodurch sich folgende Werte ergeben.

$$P_{min} = (-0.5, -0.5, -0.5) \text{ und } P_{max} = (0.5, 0.5, 0.5)$$

Diese Werte werden erst im Shader-Programm hinzugefügt, womit die Größe einer Box ebenfalls sieben Pixel beträgt.

Der Zylinder

Die Objekte Zylinder, Kegel und Kegelstumpf leiten sich von einer Oberklasse CFC ab, was für die Anfangsbuchstaben der englischen Bezeichner steht: „cone, frustum, cylinder“.
Die mathematische Formel für einen Zylinder definiert einen unendlich langen Zylinder. Erst durch die Angabe eines Minimum- und Maximum-Wert wird er in der Höhe beschränkt. Die Werte für den Einheitszylinder sind „min“ = 0 und „max“ = 1 und bräuchten nicht mit in die Textur geschrieben zu werden.

Kegel

Die mathematische Formel für einen Kegel definiert einen unendlich langen Doppelkegel, wie in Abbildung 3.3 zu sehen ist.

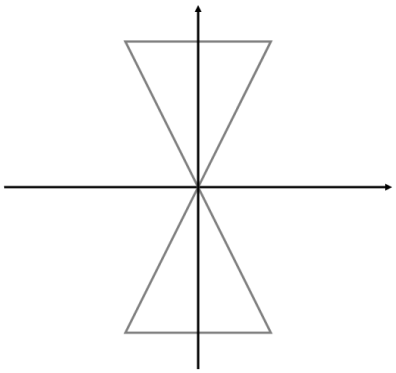


Abbildung 3.3: Der Doppelkegel

Die Werte für den sich nach oben verjüngenden Kegel sind „min“ = -1 und „max“ = 0 und müssen ebenfalls nicht mit in die Textur geschrieben werden.

Kegelstumpf

Die mathematische Formel für den Kegel beschreibt auch den Kegelstumpf. Es gibt aber eine Besonderheit. Der Einheitskegelstumpf hat eine Höhe von 1. Der untere Radius beträgt 1 und der obere Radius $0,5$.
Das Problem besteht darin, einen Kegel zu finden, der diesen Kegelstumpf enthält. Beim Einheitskegelstumpf ist es der Einheitskegel mit doppelten Höhe. (siehe Abbildung 3.4).

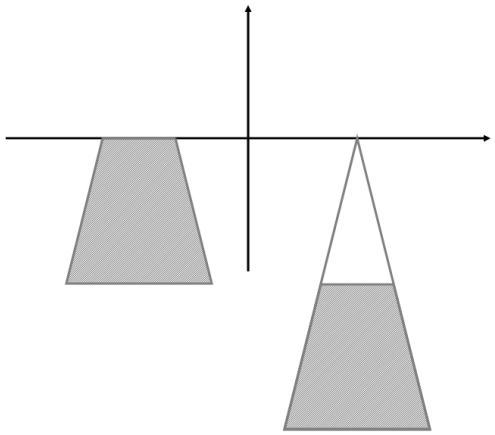


Abbildung 3.4: Der Kegelstumpf

Die Transformationsmatrix wird so angepasst, dass in Abhängigkeit vom oberen Radius ein Kegel mit entsprechender Höhe erzeugt wird. Das Minimum des Kegels beträgt weiterhin -1. Das Maximum ist für unterschiedliche obere Radien immer anders. Dieses Maximum wird durch den Strahlensatz berechnet.
Zylinder, Kegel und Kegelstumpf werden als ein Objekt angesehen und müssen im Shader unterscheidbar sein. Dies geschieht durch einen Typbezeichner, der ein einfacher Integer-Wert ist. Weiterhin können diese Objekte an ihren Enden geschlossen oder geöffnet sein. Diese drei Informationen werden in einem zusätzlichen Pixel codiert.

Damit ist das CFC-Objekt acht Pixel groß.

3.3 Die Material-Shader

Damit ein errechnetes Bild möglichst realistisch erscheint, werden Materialeigenschaften der Objekte benötigt. Ein poliertes Metallobjekt soll die Umgebung spiegeln, wohingegen ein stumpfes Objekt wie ein Ast aus Holz überhaupt nicht reflektiert.
Solche Materialeigenschaften werden in GroIMP durch einen Phong-Shader definiert. Es lässt sich ein beliebiger Shader-Baum zusammen stellen, der Einfluss auf die Beleuchtungsberechnung nimmt. Diese Shader werden in GLSL-Code reimplementiert, damit sie auf der Grafikkarte ausgeführt werden können.

Der Shader-Baum wird traversiert und in entsprechenden GLSL-Code konvertiert. Folgende Parameter stehen dabei zur Auswahl:

- diffuse Farbe
- spekulare Farbe
- Reflexionsfaktor (Materialkonstante)
- Transparenz
- diffuse Transparenz
- ambiente Farbe
- emittierende Farbe

Das einfachste Verfahren ist, den einzelnen Parametern normale RGBA-Farbwerte zuzuordnen. Es ist aber auch möglich, weitaus komplexere Materialeigenschaften zu erzeugen. Statt der Farben können prozedual erzeugte Texturen benutzt werden, wie ein Schachbrettmuster. Aus Dateien geladene Bilder lassen sich ebenfalls als Texturen verwenden. Vorgefertigte Materialien wie Granit oder Holz sind bereits vorhanden und können einfach ausgewählt werden. Jeder Knoten des Shader-Baumes besitzt einen „input“-Kanal, über den sich der ausgewählte Shader noch weiter manipulieren lässt. Beispielsweise kann auf das Schachbrettmuster eine Turbulenz einwirken. Dabei werden die „uv“-Koordinaten verändert, was zur Folge hat, dass das Schachbrettmuster aussieht wie „Kuhflecken“.

Diese Shader müssen im Kernel reimplementiert werden, was entsprechend aufwendig ist. Für diese Arbeit wurden deshalb nur das prozeduale Schachbrettmuster, der Textur-Shader und der UV-Transformations-Shader implementiert.

3.3.1 Anordnung der Objekte

Das Ablegen der Objekte in der Textur erfolgt geordnet. Die Szene wird in GroIMP als ein Graph repräsentiert, in dem die in der Szene enthaltenen Objekte als Knoten codiert sind. Dieser Graph wird zweimal traversiert.
Im ersten Durchlauf wird die Anzahl der in der Szene enthaltenen Objekte gezählt, um damit eine entsprechend große Textur zu erzeugen, die diese Objekte aufnehmen kann. Weiterhin werden die Material-Shader der Objekte gesammelt. Dabei wird darauf geachtet, dass identische Material-Shader nicht mehrfach gespeichert werden. Dasselbe trifft auf die Texturen der Objekte zu. Diese werden auch nur einmal im Textur-Atlas (siehe 3.3.2) gespeichert.
Im zweiten Durchlauf wird der entsprechende Code für die Material-Shader generiert und den Objekten zugewiesen. Objekte mit gleichen Material-Shadern erhalten einen Verweis auf dieselbe reimplementierte Shader-Funktion. Anschließend werden die Objekte in die Szenentextur geschrieben.

Die Organisation des Speicherns übernimmt eine eigene Klasse, die Kenntnis über die einzelnen Bereiche der Textur besitzt. Die Objekte werden entsprechend ihrer Typen hintereinander abgelegt. Die Reihenfolge ist Kugel, Box, Zylinder, Kegel, Kegelstumpf, Ebene und Parallelogramm. Dahinter kommen nur noch die Lichtquellen.

3.3.2 Textur-Atlas

Texturen, die auf die Objekte gelegt werden können, werden in einem Textur-Atlas organisiert. Solch ein Atlas bezeichnet eine große Textur, in der viele kleine Texturen abgelegt sind. (siehe Abbildung (3.5)[2]. Diese große Textur wird zusätzlich erzeugt und in den Speicher der Grafikkarte geladen. Die Bilder, die als Texturen dienen sollen, werden auf ein einheitliches Format skaliert. Die Größe der einzelnen Texturen beträgt 512x512 Pixel.
Die Textur für den Texturatlas wird nicht mit der gleichen hohen Genauigkeit der Ein-/ Ausgabe-Texturen erzeugt. Es werden nur vier Byte pro Pixel verwendet, wodurch sich eine höhere Anzahl an unterschiedlichen Texturen ablegen lässt. Eine kleine Textur mit 512x512 Pixeln benötigt 1MB Speicher auf der Grafikkarte. Die verwendete Grafikkarte ist in der Lage, Texturen mit der maximalen Größe von 4096x4096 Pixeln zu erzeugen. Würde diese Größe für den Textur-Atlas verwendet, könnten darin 8x8 kleine Texturen abgelegt werden. Der benötigte Speicher würde dabei 64MB betragen.

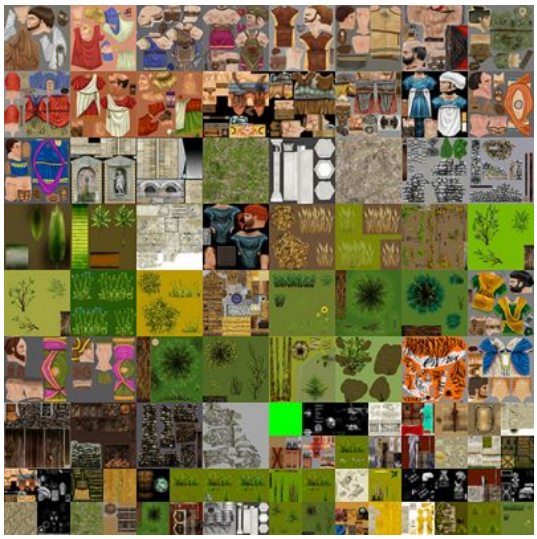


Abbildung 3.5: Beispiel für einen Textur-Atlas

[2]http://www.gamasutra.com/features/20060126/ivanov_01.shtml; Stand Januar 2008

4 Implementierung der eigenen Lösung

4.1 Generierung der Primärstrahlen

4.1.1 Kameraparameter

Von GroIMP werden die Kameraparameter bereitgestellt, die für die Erzeugung der Primärstrahlen benötigt werden. Als Erstes benötigt man natürlich die Position der Kamera. Von dort aus soll in einer gewissen Entfernung die Bildebene aufgespannt werden. Die Größe der Bildebene wird durch die „up“ und „right“ Vektoren der Kamera angegeben. Die Entfernung zur Bildebene gibt der Richtungsvektor der Kamera durch seine Länge an. Abbildung 4.1 verdeutlicht dies.

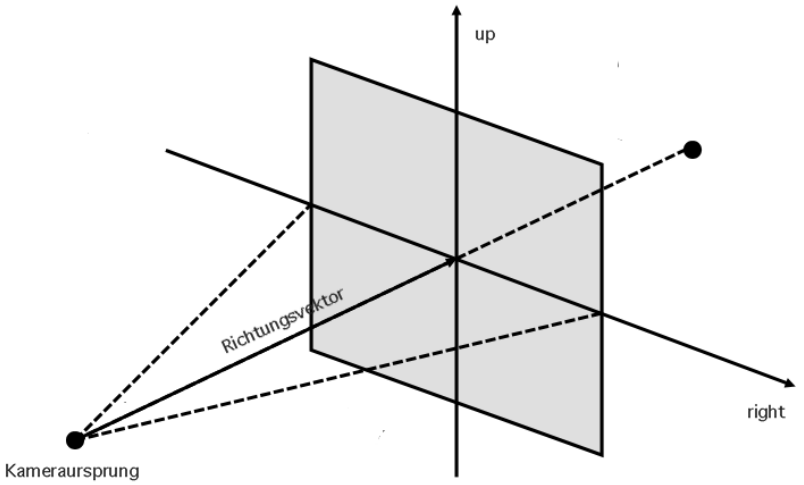


Abbildung 4.1: Kamera spannt Bildebene auf

Der „viewport“ wird auf die Größe des zu rendernden Bildes gesetzt und soll ein bildschirmfüllendes Rechteck darstellen. Der Fragmentshader wird dann für jedes Pixel dieses zu zeichnenden Rechtecks ausgeführt. Da die Strahlen geometrisch gesehen Halbgeraden

mit einem Ursprung und einem parametrisierten Richtungsvektor sind, kann man die Kameravektoren benutzen, um diese zu erzeugen.
Von einem Anfangsstrahl aus wird der Strahl um die Breite und Höhe eines Pixels weitergesetzt. Dieser Anfangsstrahl wird auf die untere linke Ecke der Bildebene gesetzt. Dies geschieht mittels linearer Algebra durch die Subtraktion des „up“ und „right“ Vektors vom Richtungsvektor.
Mit Hilfe der Funktion „gl_FragCoord“ kann man sich die x, y -Koordinaten des gerade aktiven Pixels angeben lassen. Diese Werte werden als Faktoren für die Positionierung des Strahls verwendet, indem sie mit der Breite und Höhe des Pixels multipliziert werden und auf den Richtungsvektor des Anfangsstrahls addiert werden.

Die so errechneten Strahlen werden in die entsprechenden Texturen geschrieben.

4.1.2 Strahlenbündel

Durch die Verwendung eines Strahlenbündels pro Pixel wird eine Glättung der Kanten erreicht, um Aliasing-Effekte zu verringern.
Die Generierung und Verfolgung der Primärstrahlen wird in einer Schleife mehrfach ausgeführt. Mit jedem Set Primärstrahlen wird ein neues Bild erzeugt, das zum vorherigen Bild addiert wird. Zum Schluss wird der Mittelwert über die Anzahl der Strahlen pro Pixel gebildet.
Würden immer die gleichen Strahlen verfolgt werden, wäre das Ergebnis der Mittelwertbildung identisch zu einer Verfolgung mit nur einem Strahl pro Pixel.
Deshalb werden die Primärstrahlen stochastisch auf der Fläche des Pixels verteilt. Dafür wird ein Zufallszahl-Generator benötigt. OpenGL stellt zwar „noise“-Funktionen bereit, die diesen Zweck erfüllen könnten, das Problem ist aber, dass von den Grafikkartenherstellern „3DLabs“ als einziger diese Funktionen ausprogrammiert hat. Die Treiber von NVIDIA liefern als Rückgabewert immer null. [NVI06b]
Das Problem wird durch die Implementierung eines Pseudozufallszahlgenerators gelöst. Es wird ein linearer Kongruenzgenerator benutzt, der zufällig aussehende Folgen von Zahlen erzeugt, die deterministisch berechenbar sind.

Die neu erzeugte Zufallszahl wird in Abhängigkeit von der vorherigen Zahl berechnet. Die Formel dafür lautet:

$$X_{n+1} = (aX_n + c) \; mod \; m$$

X_n ist die Reihe der zufälligen Zahlen. Folgende Bedingungen müssen erfüllt sein:

- $0 < m$
- $0 \leq a < m$
- $0 \leq c < m$
- $0 \leq X_0 < m$
- a und m haben keine gemeinsamen Teiler

Die Wahl der Parameter ist von entscheidender Bedeutung für die Qualität der Zufallszahlen. Es werden die Parameter des Minimalen-Standard-Kongruenz-Generators von Lewis, Goodman und Miller benutzt. [LGM69]
Dies sind: $a = 7^5, c = 0, m = 2^{31} - 1$. Der Startwert für die Berechnung ist ein entscheidender Faktor. Das Programmierparadigma des Fragment-Shaders hat zur Folge, dass der gleiche Code für jeden Pixel des Bildes ausgeführt wird. Der Startwert für den Minimalen-Standard-Kongruenz-Generator ist null. Würde dieser Startwert genommen werden, wäre die erzeugte „Zufallszahl“ für jeden Pixel die gleiche. Dies ist ein Problem, da so lineare Muster auftreten. In 5.4.2 wird verdeutlicht, welche Auswirkung dies hat. Es hat sich gezeigt, dass das Produkt der x- und y-Koordinaten des gerade aktiven Pixels als Startwert eine gleichmäßige Verteilung zur Folge hat. (siehe Abbildung 5.6)

Jede erzeugte Zufallszahl muss für den nächsten Berechnungsschritt gespeichert werden. Dafür wird der Alpha-Kanal der A_0 -Textur genutzt. Dieser Wert wird im Strahlverfolgungskernel einfach in die Ausgabe-Textur kopiert, damit er beim nächsten Schleifendurchlauf wieder zur Verfügung steht.

Die erzeugte Zufallszahl wird auf das Intervall $[0, 1]$ beschränkt, indem sie durch m geteilt wird. Die Strahlen sollen auf der Fläche des Pixels verteilt werden. Dafür wird der Anfangsstrahl auf die untere linke Ecke des Pixels gesetzt. Zu diesem Strahl werden Strahlen addiert, die der Breite und Höhe des Pixels entsprechen. Diese Strahlen werden mit den normierten Zufallszahlen multipliziert, wodurch die gewünschte Verteilung erfolgt.
Die Verteilung der Strahlen auf dem Pixel ist sehr wichtig. Bei „schlechten“ Zufallszahlen können Artefakte auftreten, wie in Abbildung 4.2 dargestellt ist.

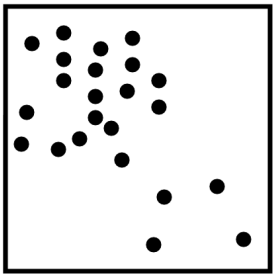


Abbildung 4.2: Schlechte Verteilung

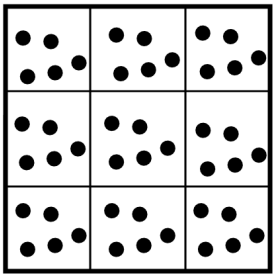


Abbildung 4.3: Gute Verteilung

Solche Artefakte werden vermieden, indem die Fläche des Pixels unterteilt wird. Meist wird ein 3x3 Gitter verwendet. Höhere Unterteilungen sind auch möglich. In jedem Quadranten wird eine entsprechende Anzahl an Strahlen verteilt. Dadurch ergibt sich für die gesamte Fläche des Pixels eine gute Verteilung, auch wenn die Verteilung der Strahlen

in den einzelnen Quadranten schlecht ist.

Die Ausführungszeit der Bildberechnung erhöht sich proportional zur Anzahl der Strahlen pro Pixel. Eine Darstellung des Qualitätsgewinns wird in 5.2 präsentiert.

4.2 Unterteilung der Bildebene in Kacheln

4.2.1 Problem

Je nach verwendetem Grafikkartenmodell ist nur ein begrenzter Speicher verfügbar. Die während der Ausarbeitung dieser Arbeit verwendete „GeForce6“ von NVIDIA besitzt einen Speicher von 128MB. Dieser Wert beeinflusst maßgeblich das zu erzeugende Bild. Der verfügbare Speicher beschränkt die Komplexität der Szene und die Größe des zu generierenden Bildes. Es ist wichtig, diesen Speicher effizient zu nutzen und so aufzuteilen, dass alle benötigten Texturen hineinpassen.

4.2.2 Texturen als Speicher

Die verwendete „GeForce6“ ist laut Spezifikation in der Lage, Texturen mit der maximalen Größe von 4096x4096 Pixel zu verarbeiten. Für eine hohe Rechengenauigkeit erfolgt die Berechnung mit vier Byte großen Fließkomma-Werten. Die hohe Rechengenauigkeit ist für die Farbwerte, die in den Texturen A_0, B_0 und A_3, B_3 gespeichert werden, nicht zwingend nötig. Ein Speichergewinn wäre möglich, wenn diese Texturen nur 16 Bit Farbwerte speichern würden. Dies ist allerdings nicht möglich, da an das „Framebuffer-Objekt“ nur Texturen im gleichen Format gebunden werden können.

> All images attached to the attachment points COLOR_ATTACHMENT0_EXT through COLOR_ATTACHMENTn_EXT must have the same internal format. [Ope]

Pro Pixel der 4096x4096 Pixel großen Textur werden drei Werte für die einzelnen Farbkanäle sowie ein Wert für den Alphakanal gespeichert. Damit diese Textur benutzt werden könnte, bräuchte die Grafikkarte 256MB RAM. Allerdings wäre der verfügbare Speicher komplett belegt. Die Ein-/ Ausgabe-Texturen haben bei dieser Konfiguration keinen Platz.
Es ist also nicht möglich, das verfügbare Maximum zu nutzen. Der vorhandene Speicher muss anders organisiert werden.

4.2.3 Größe der Ein-/ Ausgabetexturen

GroIMP stellt mit Hilfe von OpenGL eine „Live-Ansicht“ der zu rendernden Szene bereit. In diesem Anzeigefenster wird nach dem Rendern das erzeugte Bild dargestellt. Somit stellt dieses Anzeigefenster mit seiner definierten Höhe und Breite in erster Linie das Ausgabeformat des zu generierenden Bildes dar. Weiterhin ist es aber auch möglich, das gewünschte Ausgabeformat des Bildes manuell einzugeben und direkt in eine Datei

speichern zu lassen. Beiden Möglichkeiten ist gemein, dass die virtuelle Bildebene mit den gegebenen Maßen erzeugt werden muss und pro Pixel ein Strahl generiert wird. Für die Speicherung dieser Strahlen wird eine Textur benötigt, die die gleiche Größe wie die Bildebene besitzt. Wie in 3.1.1 erklärt, werden acht gleich große Texturen benötigt.
Die mögliche Größe der Szenen-Textur wäre also immer abhängig von der Größe des Anzeigefensters beziehungsweise von der eingestellten Auflösung der zu erzeugenden Bilddatei und der damit verbundenen Größe für die Ein-/ Ausgabetexturen. Dies ist aber nicht nur unpraktisch, sondern kann auch dazu führen, dass auf Grund von Speichermangel das „Framebuffer-Oobjekt“ nicht initialisiert werden kann.
Bei einer typischen Bildschirmauflösung von 1280x1024 Pixeln und acht Ein-/ Ausgabetexturen werden 160MB Speicher benötigt. Diese Konfiguration benötigt mehr Speicher als vorhanden ist und würde eine Fehlermeldung erzeugen. Eine etwas geringere Auflösung von 1024x768 Pixeln würde nur 96MB belegen. Der restliche Speicherplatz könnte aber nur eine kleine Textur mit einer geringeren Anzahl an Objekten aufnehmen.

Das Ziel war es, unabhängig von dem verfügbaren Speicher auf der Grafikkarte Bildgrößen generieren zu können, die normalerweise den verfügbaren Speicher der Grafikkarte übersteigen würden. Außerdem soll die Textur, in der die Szene gespeichert wird, maximal groß werden, damit komplexe Szenen mit sehr vielen Objekten gerendert werden können.

4.2.4 Lösung

Die Parallelisierbarkeit des Strahlverfolgungs-Algorithmus wird ausgenutzt, indem die Bildebene in 256x256 Pixel große Teile unterteilt wird. Diese Kacheln werden separat gerendert. Der Strahlverfolgungs-Kernel wird für jede Kachel ausgeführt und das endgültige Bild aus den einzelnen Kacheln zusammengesetzt. In Abbildung 4.4 wird das Verfahren verdeutlicht.
Die acht Texturen pro Kachel belegen nur noch 8MB Speicher, wodurch die Szenen-Textur 120MB groß werden kann, was einer ungefähren Auflösung von 2804x2804 Pixeln entspricht.
Ein Bild mit den Maßen 1280x1024 Pixeln würde in 5x4, also 20 Kacheln zerlegt werden. Die Ausführung der Kernel geschieht innerhalb von zwei Schleifen. Die äußere Schleife zählt die horizontalen Kacheln und die innere Schleife die vertikalen Kacheln.
Die erste Kachel beginnt in der unteren linken Ecke der Bildebene. Der “viewport“ wird diesmal auf die Größe der Kachel gesetzt. Der Anfangsstrahl beginnt nun in der unteren linken Ecke der gerade aktiven Kachel. Dafür müssen dem Shader-Programm nur die Zähler der Schleifen übergeben werden. Diese dienen als Faktoren, die mit der Größe der Kachel multipliziert werden. Somit fängt die erste Kachel bei (0; 0) und die nächste 256 Pixel weiter rechts bei (256; 0) an.

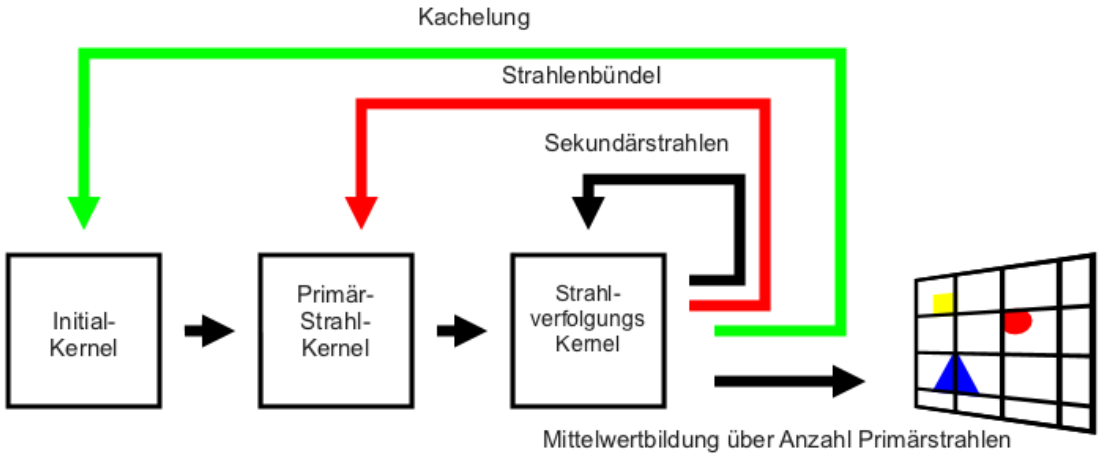


Abbildung 4.4: Renderdurchgang mit Kachelung

4.2.5 Zusammensetzen des Bildes

Bei dem gerade besprochenen Beispiel werden 20 Render-Durchgänge durchlaufen. Nach jedem Durchgang befindet sich ein Teil des gerenderten Bildes in einer Textur. Diese Teilbilder müssen nun zu dem endgültigen Bild zusammengesetzt werden.

Dafür wird das Rendern in das „Framebuffer-Objekt“ ausgeschaltet und normal in den Bildschirmspeicher, beziehungsweise in diesem Fall in einen Pixel-Buffer gerendert. Es wird mit Hilfe von OpenGL ein 256x256 Pixel großes Rechteck erzeugt, entsprechend der gerade aktiven Kachel an der richtigen Stelle positioniert und mit der erzeugten Textur beklebt.
Nachdem alle 20 Texturen erzeugt, auf entsprechende Rechtecke gelegt und in den Pixel-Buffer gerendert wurden, befindet sich in diesem das fertige Bild. Als letzter Schritt muss dieser Buffer in eine Datenstruktur ausgelesen werden, die man als eine Bilddatei auf der Festplatte speichern beziehungsweise in der „Live-Ansicht“ von GroIMP darstellen kann. Das Auslesen des Pixel-Buffers erfolgt mit der Funktion „gl_ReadPixel“ die einen Bytebuffer verlangt, in dem die Farbwerte der Pixel geschrieben werden. Dieser Bytebuffer lässt sich mit Java leicht in eine Bilddatei konvertieren.

4.3 Die Szenentextur

Die Textur, in der die Objekte gespeichert werden, wird in der Java-Anwendung mit der Funktion „glTexSubImage2D“ gefüllt. Diese Funktion erwartet dabei einen Bytebuffer. Dieser Bytebuffer soll Fließkommazahlen speichern, was bei der Initialisierung des Puffers

berücksichtigt werden muss.
Jeder Pixel besteht aus vier Komponenten, die alle vier Byte groß sind. Dadurch muss die Größe des Bytebuffers 16 mal so groß sein wie die Dimension der Textur.

Bei jedem „Texture-Lookup“ liest das Shader-Programm vier Werte aus der Textur.
Beispiel: Es soll die Transformationsmatrix für ein Objekt in die Textur geschrieben werden und vom Shader-Programm ausgelesen werden.

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Es wird die Transformationsmatrix und die invertierte Transformationsmatrix in den Puffer geschrieben.

```
Bytebuffer b = new Bytebuffer(size);
Matrix4f m   = new Matrix4f( 1, 0, 0, 2
                             0, 1, 0, 0
                             0, 0, 1, 4
                             0, 0, 0, 1
                             );

for(int i = 0; i < 2; i++)
{
        b.putFloat(m.m00);
        b.putFloat(m.m01);
        b.putFloat(m.m02);
        b.putFloat(m.m03);
        .
        .
        .
        m.invert();
}
```

Listing 4.1: Speicherung einer Matrix in der Szenentextur

Der entsprechende „Texture-Lookup“ sieht wie folgt aus:

```
        uniform sampler2DRect scene;

        mat4 m;
        m[0] = Texture2DRect(scene, vec2(0.0, 0.0);
        m[1] = Texture2DRect(scene, vec2(1.0, 0.0);
        m[2] = Texture2DRect(scene, vec2(2.0, 0.0);
        m[3] = Texture2DRect(scene, vec2(3.0, 0.0);
```

Listing 4.2: Texture-Lookup

4.4 Materialeigenschaften

Für die Beleuchtungsberechnung werden die Materialeigenschaften der Objekte benötigt. Diese sind in GroIMP als ein Shader-Baum definiert, der traversiert wird und in entsprechenden GLSL-Code überführt werden muss. Für diesen Zweck wurden Klassen geschrieben, die GLSL-Code erzeugen und bei Bedarf geladen werden. Damit einem dreidimensionalen Punkt Materialeigenschaften zugewiesen werden können, muss dieser Punkt in so genannte „uv“ -Koordinaten umgerechnet werden. Am Beispiel der Kugel soll dies kurz erklärt werden.

4.4.1 UV-Koordinaten

Die „uv“ -Koordinaten repräsentieren zweidimensionale Punkte. Diese Punkte müssen auf die Oberfläche eines dreidimensionalen Objektes abgebildet werden. Das Problem beim Rendering ist aber die Umkehr dieser Funktion und wird als „Inverse Mapping-Problem“ bezeichnet.

> Beim Rendering muss jedoch das inverse Mapping-Problem gelöst werden, d.h. den bekannten (x,y,z)-Koordinaten des Flächenpunktes P müssen (u,v)-Koordinaten zugeordnet werden. [Krö01b]

$$(u, v) = F_{inv\,map}(P) = F_{inv\,map}(x, y, z)$$

Die „uv“ -Koordinaten der implementierten Objektprimitive lassen sich relativ leicht berechnen. Es wird die Idee der umhüllenden Flächen benutzt, da diese den implementierten Objekten entsprechen. Die „uv“ -Koordinaten des Zylinders, Kegels und Kegelstumpfes werden mit dem Zylinder-Mapping berechnet. Bei der Box, der Ebene und dem Parallelogramm wird planares Mapping verwendet. Die Kugel wird sphärisch gemappt.

Kugel-Mapping

Punkte auf der Oberfläche der Kugel lassen sich durch Kugelkoordinaten ϕ und θ parametrisieren (siehe Abbildung 4.5), entnommen aus einem PDF vom Lehrstuhl Computergrafik der BTU Cottbus[1].
Die Formel für die Berechnung von u und v lautet wie folgt:

$$u = \frac{\pi + \arctan(y, x)}{2\pi} \tag{4.1}$$

$$v = \frac{\arccos(z)}{\pi} \tag{4.2}$$

[1] http://www-gs.informatik.tu-cottbus.de/∼wwwgs/cg2_v13a.pdf; Stand Januar 2008

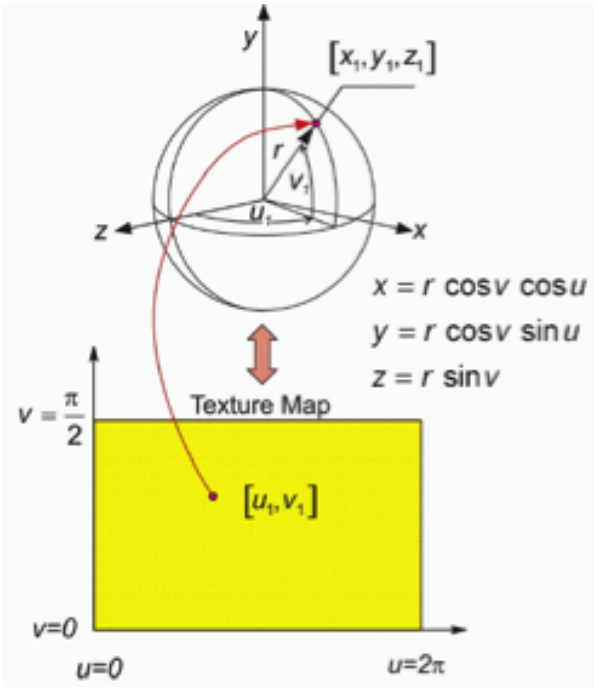


Abbildung 4.5: Sphärisches Mapping

4.4.2 Reimplementierung der Shader

Die Klassen zur Erzeugung von GLSL-Code implementieren das Interface `SunshineShader`, wodurch die Funktion „getCode()“ ausprogrammiert werden muss. Diese Funktion liefert einen `String` zurück, der den Programmcode enthält und an entsprechender Stelle beim Erzeugen des Strahlverfolgungs-Kernels eingetragen werden muss.
Es gibt zwei Arten von GLSL-Code erzeugenden Klassen. Eine Klasse erstellt einen `Struct`, der die in 3.3 vorgestellten Parameterwerte enthält. Die andere Art erzeugt Funktionen, die die Funktionalität der GroIMP-Shader implementiert.
Je nachdem, wie die einzelnen Parameter in GroIMP definiert wurden, werden zum Füllen des `Structs` diese Shader-Funktionen aufgerufen oder aber RGBA-Farbwerte als „immediate“ eingetragen.
Wenn ein Objekt einen Phong-Shader besitzt, muss das innerhalb des Kernels bekannt sein. Dem Objekt wird ein Bezeichner mitgegeben, an dem sich feststellen lässt, welches zuvor erstellte `Phong-Struct` zu dem Objekt gehört. Dieser Bezeichner ist ein einfacher Integer-Wert, der durch ein Bedingungskonstrukt überprüft wird.

Der Schachbrettmuster-Shader

Hier ein Bespiel für die Reimplementierung der Funktion des Schachbrettmuster-Shaders.

```
vec4 getCheckerColor(UV uv, vec4 cl1, vec4 cl2)
{
        return
        mod(floor(2.0*uv.u), 2.0) == mod(floor(2.0*uv.v), 2.0)
        ? cl1 :  cl2;
}
```

Listing 4.3: Schachbrettmuster-Shader

Diese Funktion erwartet als Parameter die „uv“ -Koordinaten und zwei RGBA-Farben. Anhand der übergebenen „uv“ -Koordinaten wird entschieden, welche der zwei Farben zurückgegeben wird.
Diese Funktion wird von allen Objekten aufgerufen, die in ihrem Shader-Baum ein Schachbrettmuster haben. Damit nicht jedes Mal wieder die gleichen Funktionen erzeugt und in den Code des Kernels geschrieben werden, wird mitgezählt, welche Funktionen bereits integriert wurden.

Der Textur-Shader

Die Objekte können mit Bildern texturiert werden. Diese Bilder müssen im Speicher der Grafikkarte vorhanden sein. Wie in 3.3.2 beschrieben, wird ein Textur-Atlas verwendet, in dem alle Texturen untergebracht sind. Die „uv“ -Koordinaten müssen entsprechend der Position der Textur im Atlas angepasst werden. Damit Bilder in den Speicher der Grafikkarte geladen werden können, müssen sie in ein Format gebracht werden, das der dafür benutzte OpenGL-Aufruf erwartet. Dies ist ein Puffer, der Integerwerte speichert.

Damit haben die Objekte innerhalb des Strahlverfolgungs-Kernels Zugriff auf ihre Materialeigenschaften und können entsprechend beleuchtet werden. Für die Beleuchtung gibt es zwei unterschiedliche Implementierungen.

4.5 Lokale Beleuchtung

Die berechneten Farbwerte der Objekte durch die lokale Beleuchtung sind abhängig von deren Materialeigenschaften. Die Objekte können als Materialeigenschaften einfache RGBA-Farbwerte besitzen oder aber komplexe Beschreibungen durch einen Shader-Baum. Je nachdem, welche Materialeigenschaften definiert wurden, wird eine andere Berechnung für die Beleuchtung benutzt.

RGBA-Shader

Beim einfachen RGBA-Shader wird die Beleuchtung des Objekts in Abhängigkeit vom einfallenden Winkel des Lichts berechnet. Bei einem parallel zum Normalenvektor verlaufenden Lichtquellenvektor wird der Punkt komplett beleuchtet. Die Idee ist, dass in diesem Fall der Punkt von der maximalen Anzahl an Photonen getroffen wird. (siehe Abbildung 4.6).

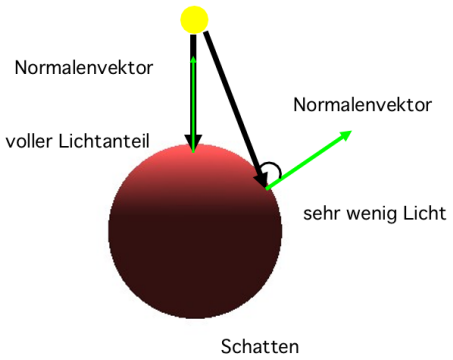


Abbildung 4.6: Direktes Licht

Der Winkel zwischen Normalenvektor und Lichtquellenvektor wird immer steiler, je weiter außen auf der Kugel sich der zu beleuchtende Punkt befindet. Die Anzahl der Photonen nimmt bei steilerem Winkel ab. Dieses Prinzip lässt sich als Formel ausdrücken, indem man die Grundfarbe des Objekts mit dem Kosinus des Winkels multipliziert. Die Farbe der Lichtquelle wird als weiterer Faktor in der Rechnung berücksichtigt. Der Kosinus des Winkels zwischen Normalenvektor $\vec{n}$ und Lichtquellenvektor $\vec{PL}$ ist das Skalarprodukt dieser beiden normalisierten Vektoren.
Es muss darauf geachtet werden, dass der Kosinus eines Winkels, der größer als 90° ist, negativ wird. Dies wird durch eine Maximumfunktion ausgeglichen.

$$max(0.0, \vec{n} \odot \vec{PL})$$

Shader-Baum

Bei der Verwendung eines Phong-Shader-Baumes wird die lokale Beleuchtung mit der gleichen Formel berechnet, die auch in OpenGL benutzt wird. Die Parameter der Formel werden aber entsprechend der Werte des Shader-Baumes verändert. Die Formel wurde

aus [SA04] entnommen und lautet:

$$\begin{aligned} c_{pri} &= e_{cm} + a_{cm} * a_{cs} \\ +\sum_{i=0}^{n-1} (att_i)(spot_i)[a_{cm} * a_{cli} \\ &+(n \odot \vec{VP}_{pli}) d_{cm} * d_{cli} \\ &+(f_i)(n \odot h_i)^{s_{rm}} S_{cm} * S_{cli}] \end{aligned} \tag{4.3}$$

Die verwendeten Operatoren $*$, $\odot$ und die Funktion f_i werden wie folgt definiert:

$$c_1 * c_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$$
$$d_1 \odot d_2 = max(d_1 \cdot d_2, 0.0)$$

$$f_i = \begin{cases} 1, \; n \odot \vec{VP}_{pli} \neq 0 \\ 0, \; sonst \end{cases}$$

Die benötigten Parameter lauten wie folgt:

- c_{pri}: Primärfarbe
- a_{cs}: ambiente Farbe der Szene
- a_{cm}: ambiente Farbe des Materials
- d_{cm}: diffuse Farbe des Materials
- s_{cm}: spekulare Farbe des Materials
- e_{cm}: emittierende Farbe des Materials
- s_{rm}: Materialkoeffizient
- a_{cli}: ambiente Intensität der Lichtquelle
- d_{cli}: diffuse Intensität der Lichtquelle
- s_{cli}: spekulare Intensität der Lichtquelle
- n: der Normalenvektor am Punkt
- $\vec{VPL}_{pli}$: der Vektor vom Punkt zur Lichtquelle
- h: der Halbvektor zwischen Licht- und Betrachterrichtung (Blinn-Phong-Modell)

5 Bewertung der eigenen Lösung

5.1 Einleitung

In diesem Kapitel werden die Ergebnisse des implementierten GPU Strahlverfolgers „Sunshine“ besprochen. Anhand von Beispielbildern soll die erreichte Qualität und Geschwindigkeit gezeigt werden. Die entwickelte Lösung wird dabei mit dem bestehenden CPU Strahlverfolger „Twilight“ verglichen.
Damit ein gerechter Vergleich durchgeführt wird, berechnet „Twilight“ die Szenen ebenfalls ohne Octree. Dieser konnte nicht ganz abgeschaltet werden, sondern nur auf die Wurzel reduziert werden, wodurch weiterhin ein gewisser Vorteil besteht.

5.1.1 Beschreibung des Testsystems

Der Strahlverfolger wurde auf einem Desktop PC unter WindowsXP Professional programmiert und getestet und enthält folgende Bauteile.

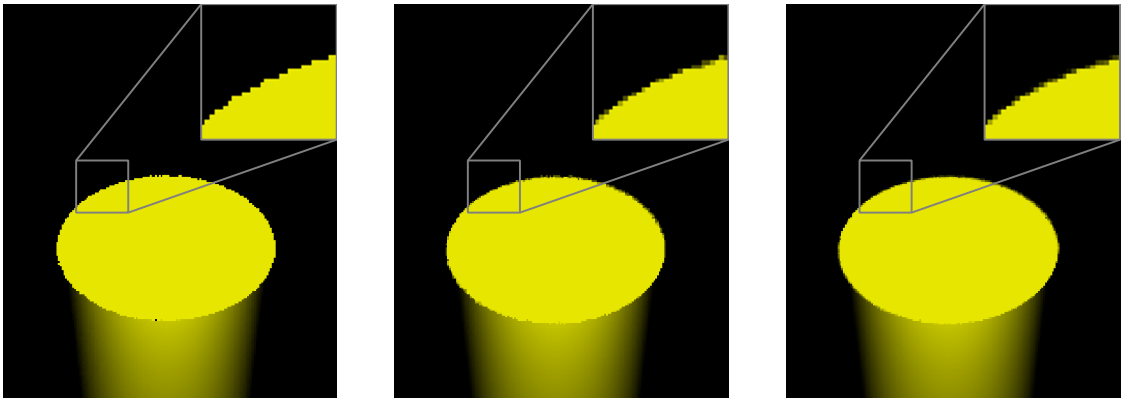
Bauteil	Beschreibung
Grafikkarte	Geforce6600 128MB RAM AGP
CPU	AMD 1800+
Arbeitsspeicher	1024MB
Festplatte	120GB

Tabelle 5.1: Das Testsystem

Diese Konfiguration entspricht nicht dem aktuellen Standard heute verfügbarer Computer und ist daher nicht repräsentativ. Besonders der Umstand, dass heutzutage Mehrkernprozessoren standardmäßig verbaut werden und das Testsystem mit nur einem Prozessorkern rechnet, lässt keine genaue Aussage über einen möglichen Geschwindigkeitsvorteil zu.

5.2 Kantenglättung

Durch die Verwendung von mehreren, über die Fläche des Pixels verteilten Primärstrahlen wird eine Kantenglättung erreicht. Die erreichte Qualität und die dafür benötigte Zeit wird anhand von Abbildung 5.1a bis 5.1c verdeutlicht. Gerendert wurde ein einzelner Zylinder bei einer Unterteilung der Pixel in ein 3x3-Gitter.
Wie man sieht, steigt die Qualität mit der Erhöhung der Anzahl an Strahlen kontinuierlich. Die benötigte Zeit für einen Render-Durchlauf erhöht sich wie erwartet ebenfalls.



(a) Ein Strahl: 5,27 sec (b) Vier Strahlen: 6,64 sec (c) 16 Strahlen: 9,02 sec

Abbildung 5.1: Kantenglättung

„Twilight“ benötigt für eine 16-fache Kantenglättung 32, 47 sec. Die erreichte Qualität ist dabei genau so gut.

5.3 Schattenberechnung

Die Schattenberechnung trägt einen Großteil zu einer realistischen Darstellung der Szene bei. In diesem Abschnitt wird gezeigt, welche Arten von Schatten von „Sunshine“ erzeugt werden können.

5.3.1 Halbschatten

Bei der Verwendung von Punktlichtquellen erzeugt die Schattenberechnung hart abgrenzende Schatten, wie in Abbildung 5.2 zu sehen ist. Dies liegt daran, dass ein Punkt entweder beleuchtet wird oder nicht. Durch die Verwendung von flächigen Lichtquellen wie dem Parallelogramm können weiche Halbschatten dargestellt werden.
Dafür werden mehrere Schattenstrahlen auf der Oberfläche der Lichtquelle zufällig verteilt. Ein Objekt kann die Lichtquelle teilweise verdecken, woraufhin die Schattenfühler mal die Lichtquelle treffen oder aber ein Objekt dazwischen.
Der Quotient der Anzahl Strahlen, die kein Objekt treffen, und der Anzahl der getesteten Strahlen wird mit dem Farbwert der direkten Beleuchtung multipliziert. Beispielsweise werden vier Schattenfühler verfolgt, von denen zwei ein Objekt und zwei die Lichtquelle treffen. Der Punkt wird dann mit der halben Intensität beleuchtet.

$$I_{neu} = 0 + 0 + 1/4 * I + 1/4 * I$$

Abbildung 5.3 zeigt den Realitätsgewinn bei der Verwendung einer flächigen Lichtquelle. Die Lichtquelle wurde mit 191 Schattenstrahlen abgetastet. Die Szene hat eine Auflösung von 715x458 Pixel und wurde ohne Kantenglättung und mit Rekursionstiefe eins

berechnet. Es wurden 327.470 Primärstrahlen verfolgt. Die Dauer für die Berechnung betrug 8, 36 sec.

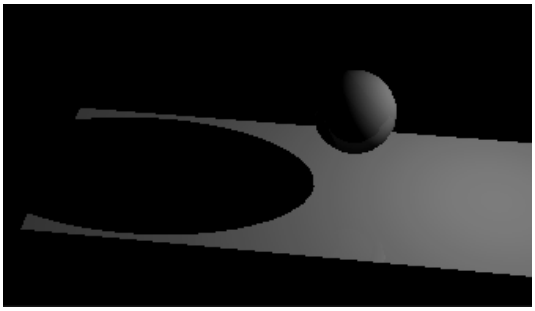


Abbildung 5.2: Schatten durch Punktlicht

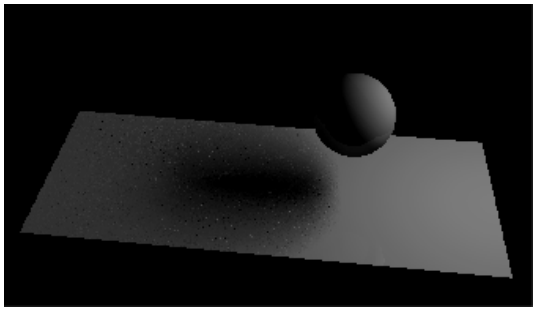


Abbildung 5.3: Halb- und Kernschatten

Der Übergang zwischen Halb- und Kernschatten wird fließender, wenn mehr Schattenfühler verwendet werden. Im Gegenzug steigt aber auch die Rechenzeit für die vermehrten Schnittpunktberechnungen.
Dieselbe Szene von „Twilight“ berechnet, erzeugte Abbildung 5.4. Dabei wurde links der Standard-Strahlverfolger verwendet, der 20, 5 sec gerechnet hat, und rechts der Pathtracer mit einer Rechendauer von 39, 3 sec.

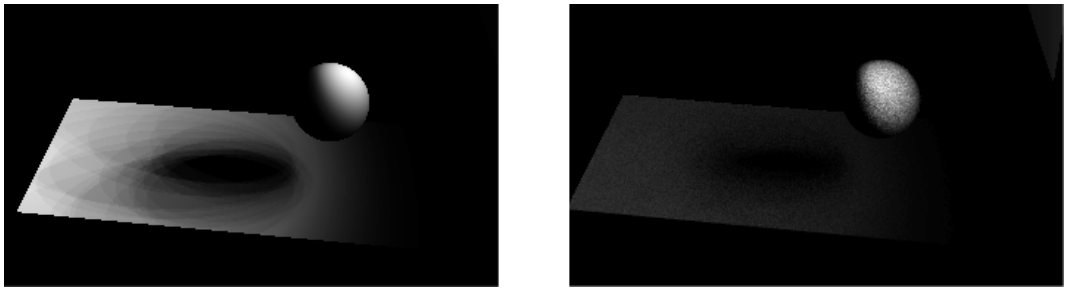


Abbildung 5.4: Halbschatten mit „Twilight“ berechnet

Die Berechnung des Bildes auf der Grafikkarte benötigte nur die Hälfte der Rechenzeit die die CPU brauchte und sieht, im Vergleich zum Standard-Strahlverfolger, subjektiv gesehen schöner aus.

5.3.2 Objektschatten

Die Verwendung von halbtransparenten Texturen wirkt sich auf die Form, der auf sie aufgebrachten Objekte aus. Dementsprechend muss der Schattenwurf dieser Objekte der Form der Textur entsprechen. Für die Blätter der Bäume werden Parallelogramme verwendet. Ein eckiger Schattenwurf würde sehr unrealistisch aussehen. Abbildung 5.5 zeigt ein Blatt mit korrektem Schattenwurf.

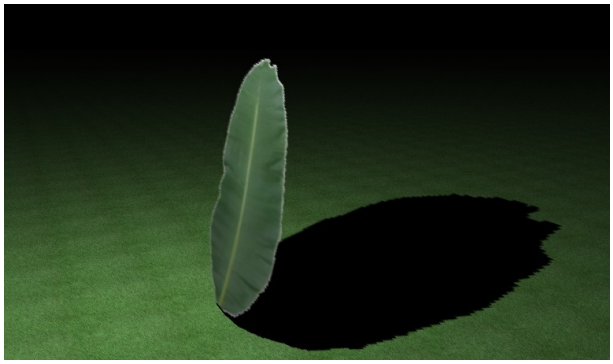


Abbildung 5.5: Schattenwurf eines einzelnen Blattes

5.4 Pathtracing

In diesem Abschnitt wird der implementierte Pathtracer vorgestellt. Es wird die diffuse Wechselwirkung gezeigt, die mit dem Pathtracing möglich ist, und die Probleme, die durch den verwendeten Zufallsgenerator entstanden.

5.4.1 Diffuse Wechselwirkung

Mit Hilfe des Pathtracing ist es möglich, diffuse Wechselwirkungen zu simulieren. Anhand der Cornell-Box (Abbildung 5.6) wird gezeigt, dass die rote und die grüne Wand etwas Farbe in Richtung der grauen Wand im Hintergrund und in Richtung des Boden diffus reflektieren. Deutlich sieht man das Rauschen im Bild, das den Fehlern der genäherten Lösung der Rendering-Gleichung entspricht.

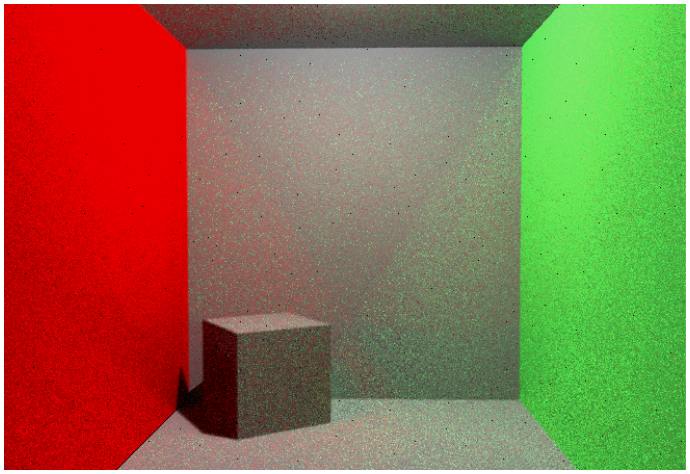


Abbildung 5.6: Ausbluten von Farbe an Hand der Cornell-Box

Es wurden 16 Strahlen pro Pixel generiert, womit 5.261.504 Primärstrahlen verfolgt wurden. Die Rekursionstiefe beträgt drei Schritte. Die benötigte Zeit für die Berechnung beträgt $34,56$ sec. Relativ deutlich sind die diagonal verlaufenden Farbschimmer der grünen und roten Wand auf der Wand im Hintergrund zu erkennen. Dies ist eine Auswirkung des linearen Kongruenz-Generators, der die Zufallszahlen generiert.

5.4.2 Probleme

Die lineare Abhängigkeit der Zufallszahlen hat zur Folge, dass wie in Abbildung 5.6 zu sehen ist, Streifen im Bild entstehen. Besonders deutlich wird dieses Problem in Abbildung 5.7 dargestellt.

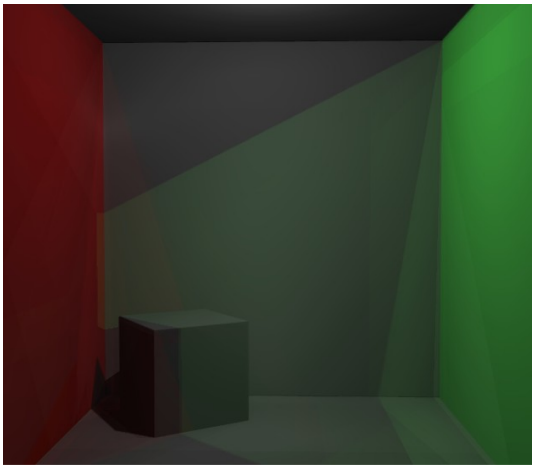


Abbildung 5.7: Streifenbildung

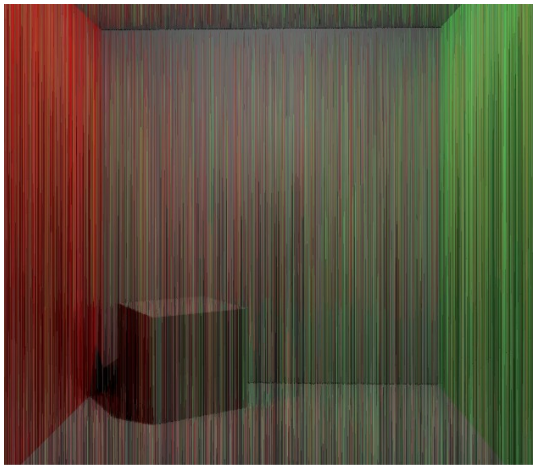


Abbildung 5.8: Streifen in y-Richtung

Diese deutlichen Streifen sind entstanden, da für jedes Pixel des Bildes dieselbe Zahl als Startwert des Zufallsgenerators verwendet wurde. Die nachfolgenden Zahlen waren dann immer gleich, wodurch die Sekundärstrahlen alle in die gleiche Richtung gezeigt haben. Bei Abbildung 5.8 wurde die x-Koordinate des aktiven Pixels als Startwert benutzt, was auch kein zufriedenstellendes Ergebnis lieferte. Eine annehmbare Lösung (Abbildung 5.6) stellt das Produkt der x,y-Koordinaten des aktiven Pixels als Startwert dar.
Der verwendete Zufallszahl-Generator ist eine sehr einfache Methode, zufällige Zahlen zu generieren. Es sollte in Betracht gezogen werden, einen anderen Zufallszahlen-Generator zu implementieren.

5.5 Geschwindigkeitstest

In diesem Abschnitt wird gezeigt, ob ein Geschwindigkeitsvorteil bei der Berechnung unterschiedlicher Szenen im Vergleich zum vorhandenen Strahlverfolger „Twilight“ besteht. Die Schnittpunktberechnung steht dabei im Vordergrund. Zu diesem Zweck werden Szenen mit steigender Anzahl von Objekten berechnet.

5.5.1 Szene ohne Kantenglättung

Anzahl Objekte	Twilight	Sunshine	Twilight mit Octree
7	10,74 sec	15,89 sec	10,89 sec
25	12,16 sec	17,61 sec	11,56 sec
85	20,27 sec	20,26 sec	12,14 sec
271	44,37 sec	27,86 sec	14,66 sec
841	2 min 26,9 sec	39,20 sec	16,33 sec

Tabelle 5.2: Geschwindigkeitstest

Als Szene wurde ein Busch ausgewählt, der pro Wachstumsschritt n um 3^n Äste mit je einem Blatt wächst.

(a) 25 Objekte (b) 85 Objekte (c) 271 Objekte

Abbildung 5.9: Testszene Busch

An Hand der Kurven in Abbildung 5.10 ist zu erkennen, dass der CPU-Strahlverfolger bei wenigen Objekten schneller rechnet als die Grafikkarte. Dies wird auf eine optimierte, objektorientierte Programmierung und den verbleibenen Hüllkörper zurückgeführt. Außerdem ist die Initialisierungsphase beim GPU Strahlverfolger länger, da das Shader-Programm erstellt werden muss und die Szenendaten auf die GPU geladen werden. Bei 85 Objekten wird diese Diskrepanz ausgeglichen. Ab diesem Punkt zeigt sich die Leistungsfähigkeit der Grafikkarte. Die benötigte Rechenzeit erhöht sich sehr viel langsamer als es bei „Twilight“ der Fall ist und befindet sich selbst ohne Beschleunigungsstruktur in einem akzeptablen Bereich.
Als Vergleich wurden dieselben Szenen unter Verwendung eines Octree zur Beschleunigung noch einmal berechnet. Die erreichten Zeiten unterscheiden sich deutlich von den beiden vorherigen Berechnungen. Es ist daher anzunehmen, dass eine Implementierung des Octree-Beschleunigungsverfahrens auf der Grafikkarte einen ähnlichen Leistungszuwachs bringen wird.

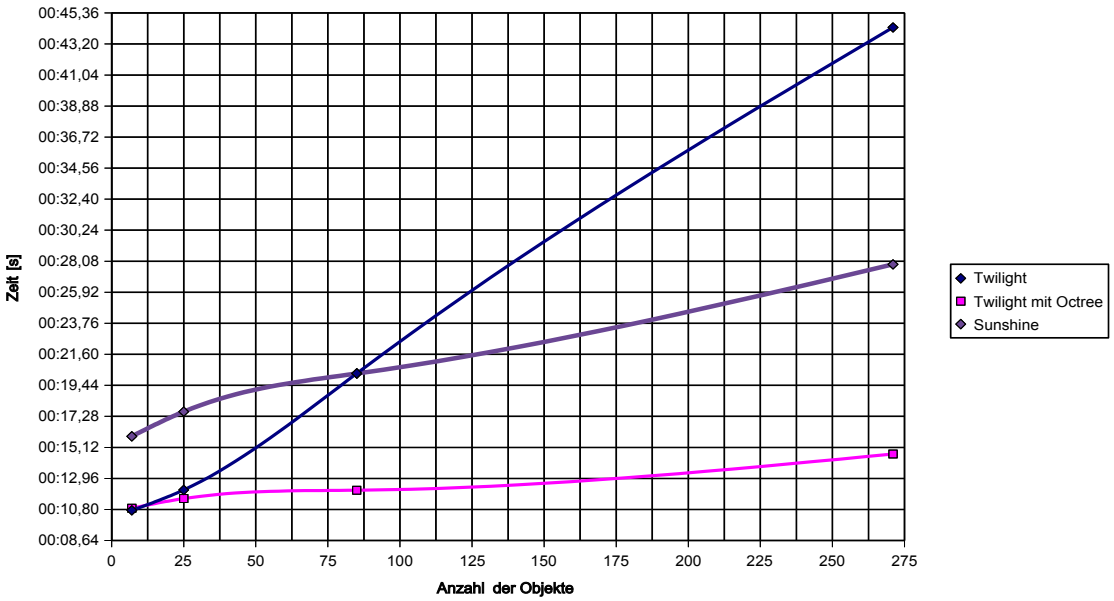


Abbildung 5.10: Zeiterfassung für unterschiedlich viele Objekte

Die erreichten Zeiten von „Twilight“ werden auf aktuellen Zweikernprozessoren um mehr als das Doppelte schneller sein. Da aber auch die Grafikkarten mit jeder Generation schneller werden, ist anzunehmen, dass es ebenfalls einen Punkt geben wird, an dem die Schnittpunktberechnung auf der Grafikkarte schneller erfolgt als auf den CPU-Kernen.

5.5.2 Szene mit steigender Kantenglättung

In diesem Abschnitt wird die Anzahl der Strahlen pro Pixel kontinuierlich erhöht. Die Anzahl der Objekte in der Szene bleibt dabei gleich. Es wurde die Busch-Szene mit 25 Objekten und mit Rekursionstiefe 1 verwendet.
Wie in Abbildung 5.11 zu sehen ist, gibt es bei vier Strahlen pro Pixel eine Übereinstimmung der Rechenzeit. Nach diesem Punkt zeigt sich ein klarer Geschwindigkeitsvorteil für die Grafikkarte.
Es wird nur einmal die Szene erfasst und in den Speicher der Grafikkarte geladen. Das Rahmenprogramm startet die Berechnung auf der GPU und vertauscht dann nur noch die Ein-/ Ausgabe-Texturen. Dadurch verbleiben die Daten im Speicher der Grafikkarte. Das Verhältnis zum Schreiben der Ergebnisse und der wiederholten Verfolgung der Primärstrahlen verbessert sich mit jedem zusätzlichen Strahl pro Pixel. Durch diesen Test zeigt sich der entscheidende Vorteil der Berechnung durch die Grafikkarte.

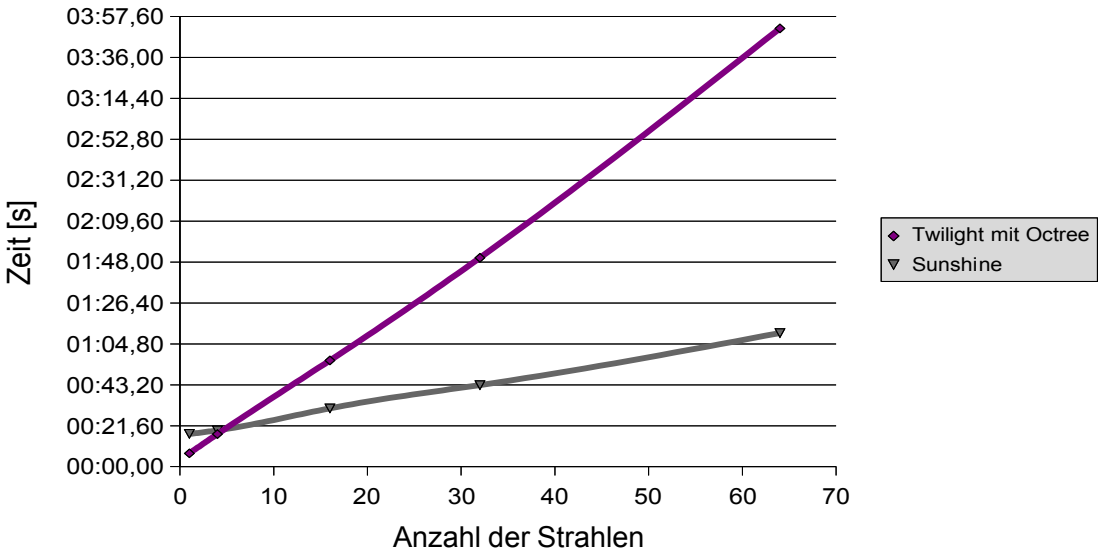


Abbildung 5.11: Zeiterfassung für unterschiedlich viele Strahlen pro Pixel bei 25 Objekten

5.5.3 Probleme

Beim Testen größerer Szenen trat ein Fehler in Form eines schwarzen Bildes auf. Die Ursache dafür lässt sich nicht genau feststellen. Das „Debuggen“ von OpenGL und speziell der Shader-Programme ist keine triviale Angelegenheit. Es gibt Programme wie das „PerfKit“ von NVIDIA oder dem „GDEBugger“ von Gremedy, aber die Verwendung dieser Werkzeuge mit „ jogl“ ließ sich nicht bewerkstelligen. Das OpenGL Error-Log zeigte keine Fehler. Dennoch wird angenommen, dass das Limit der erlaubten Anzahl an Shader-Anweisungen erreicht wurde.

5.5.4 Fazit

Die Strahlverfolgung ist ein sehr rechenintensives Verfahren zur Erzeugung von Bildern. Die benötigte Rechenzeit hängt dabei von der Komplexität der Szene und der verwendeten Hardware ab. Für das angewendete Testsystem wurde gezeigt, dass die implementierte Lösung die Schnittpunktberechnung auf der GPU schneller ausführt als auf der CPU. Würde das Testsystem mit einem Zwei- oder gar Vierkern-Prozessor betrieben, würde der Test höchstwahrscheinlich zu Gunsten der CPU entschieden werden. Deshalb lässt sich nicht pauschal sagen, dass das eine Verfahren schneller ist als das andere. Wenn allerdings mit Kantenglättung gerechnet wird, zeigt sich ein klarer Geschwindigkeitsvorteil für die Grafikkarte.
Im Anhang sind einige Arbeitsproben angefügt, um einen Eindruck zu vermitteln, was der implementierte Strahlverfolger zu leisten im Stande ist.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In diesem Kapitel wird das Ergebnis der vorliegenden Bachelorarbeit zusammengefasst. Die Aufgabe dieser Arbeit war es, für GroIMP einen Strahlverfolger zu implementieren, der auf der Grafikkarte ausgeführt wird. Dieses Ziel wurde erreicht. Die implementierte Lösung ist für den produktiven Einsatz aber noch nicht bereit. Zum einen ist das Verfahren noch zu langsam, da ein Beschleunigungsverfahren wie der Octree fehlt, zum anderen gibt es ungeklärte Probleme bei der Berechnung größerer Szenen. Sollte sich herausstellen, dass die erlaubte Anzahl an Shader-Anweisungen überschritten wurde, wäre der Octree ebenfalls eine Lösung dafür, da dieser die Anzahl der Schnittpunktberechnungen erheblich verringert. Für die Implementierung des Octrees existieren bereits Pläne, so dass in einer folgenden Version dieser vorhanden sein wird.

Der Strahlverfolgungs-Algorithmus wurde in zwei Varianten implementiert, einen naiven Strahlverfolger, bei dem an den Objektoberflächen nur ideale Sekundärstrahlen erzeugt werden und einem stochastischen Pathtracer, der diffuse Reflexion und damit indirekte Beleuchtung ermöglicht. Für qualitativ hochwertige Bilder kann ein Faktor für die Kantenglättung angegeben werden.
Es wurden dabei alle in GroIMP vorhandenen Objektprimitive implementiert. Als Lichtquellen können Punktlichter, Scheinwerferlichter und flächige Lichter in Form des Parallelogramms verwendet werden. Dafür wurde eine Datenstruktur entworfen, die eine Abbildung dieser Objekte auf die Grafikkarten-Texturen ermöglicht.
Von den Material-Shadern wurde der Textur-Shader, der UV-Transformations-Shader und der Schachbrett-Shader reimplementiert. Die restlichen in GroIMP verfügbaren Material-Shader werden in den folgenden Versionen ergänzt.
Die Texturen von texturierten Objekten werden durch einen Textur-Atlas der Grafikkarte zur Verfügung gestellt. Dabei wird bei halbtransparenten Texturen der geworfene Schatten korrekt berechnet.
Die implementierte Lösung ist zur Zeit nur auf Grafikkarten von NVIDIA lauffähig. Eine Anpassung an ATI/AMD-Karten steht noch aus.

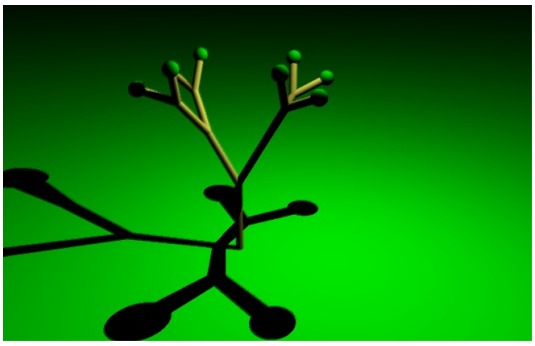
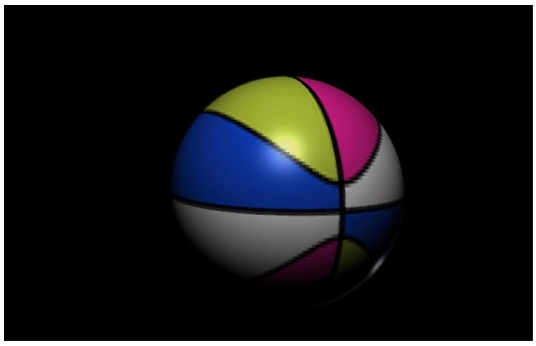
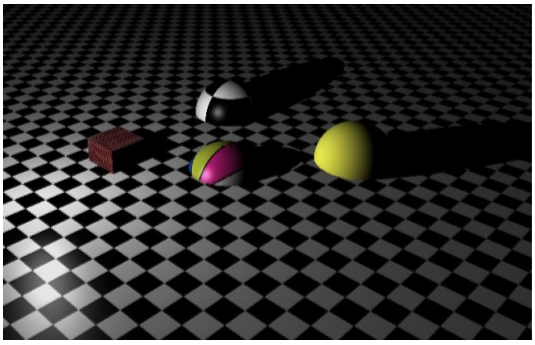
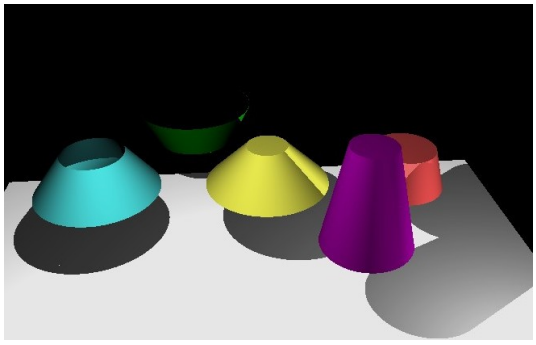
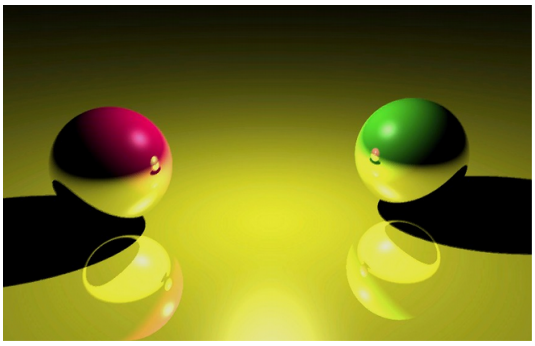
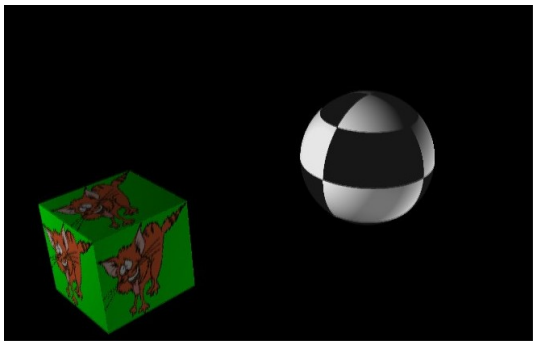
6.2 Ausblick

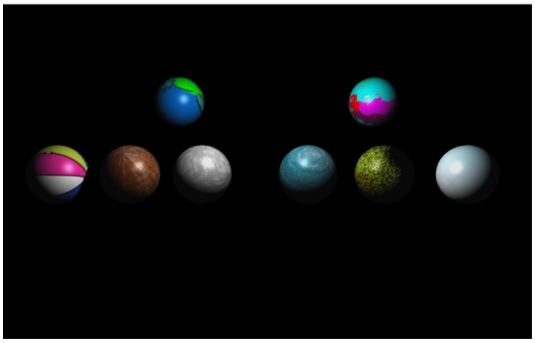
Dieser letze Abschnitt zeigt mögliche Richtungen, die in zukünftigen Arbeiten weiterverfolgt werden können.
Heutzutage lassen sich die Taktraten der Prozessoren nur noch unter sehr hohem Aufwand erhöhen. Daher ist man dazu übergegangen, mehrere Kerne pro Prozessor zu verwenden.
Ein vergleichbarer Ansatz wird bei Grafikkarten auch betrieben. NVIDIA nennt den gleichzeitigen Einsatz von bis zu vier Grafikkarten SLI-Betrieb, bei ATI/AMD wird das Verfahren „CrossFire“ genannt. Auf der diesjährigen „CES“ wurde sogar schon ein „Motherboard“ mit acht PCI-Express-Schnittstellen vorgestellt, mit dem es theoretisch möglich wäre acht Grafikkarten parallel zu betreiben, sofern Treiber existieren, die das unterstützen.
Zukünftige Arbeiten könnten dies nutzen und die Berechnung eines Bildes auf mehrere Grafikkarten aufteilen. In diesem Zusammenhang könnte auch eine Grid-Plattform mit entsprechenden Grafikkarten ausgestattet werden, um die Berechnung zu verteilen.

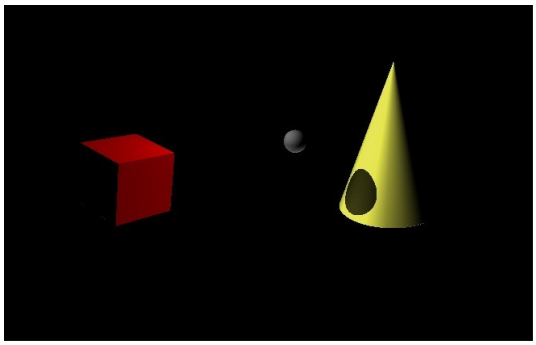
Weiterhin wäre die Umsetzung weiterer Beleuchtungsverfahren auf der Grafikkarte möglich. Bidirektionales Pathtracing oder Radiosity wären denkbare Algorithmen. Interessant wäre auch die Implementierung zusätzlicher Objekte wie NURBS oder Polygone.

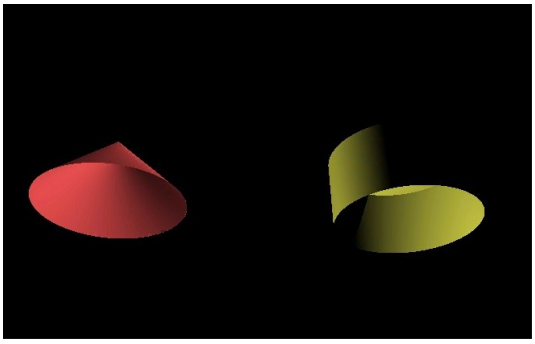
Zukünftige Generationen von Grafikkarten werden sich mit heute schon vorhandenen Entwickler-Werkzeugen wie „CUDA“ von NVIDIA bequemer programmieren lassen. Denkbare wäre zum Beispiel eine Aufhebung der Limitierung der Shader-Instruktionen. Die implementierten Shader-Programme sollten noch an ATI/AMD Grafikkarten angepasst werden.

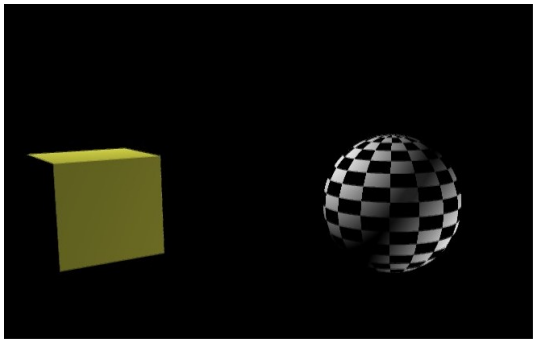
7 Anhang

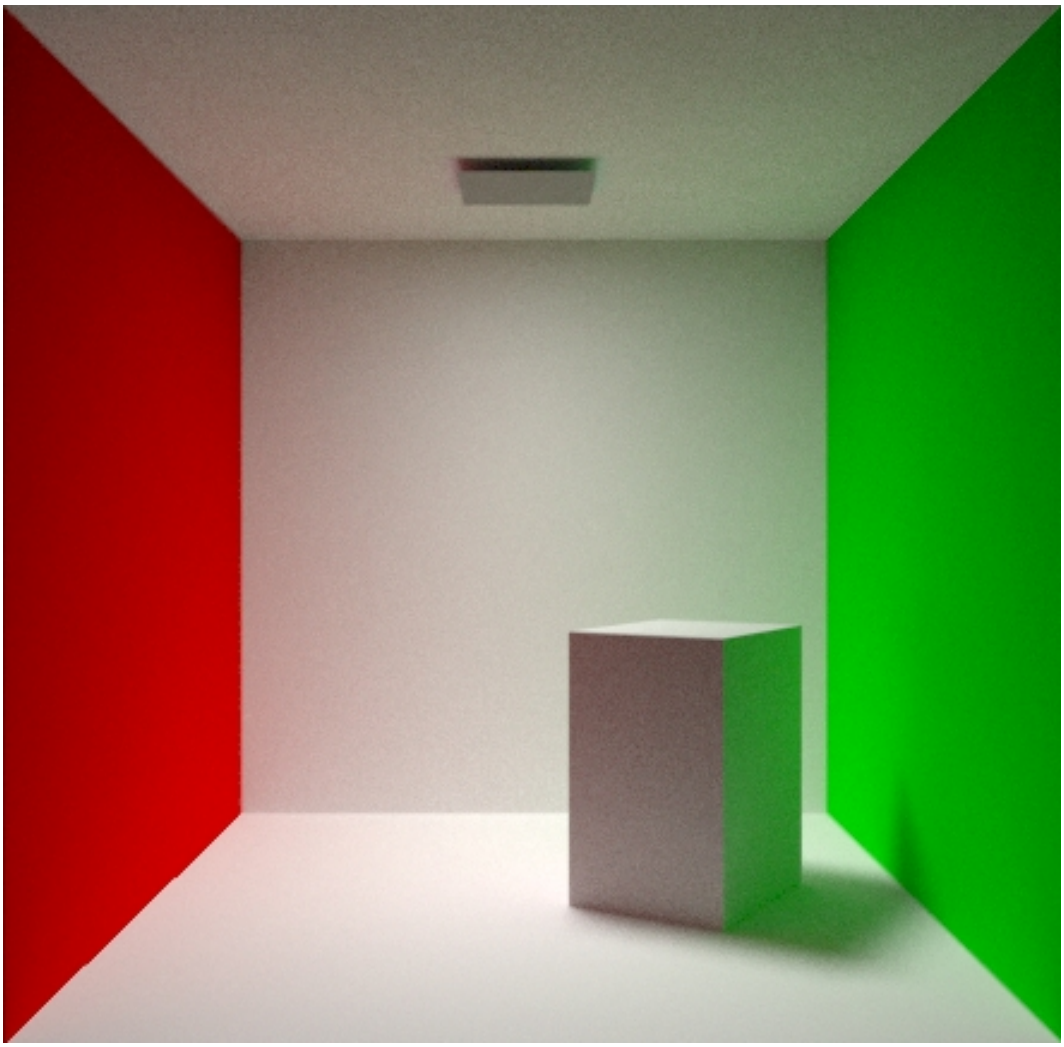

















Literaturverzeichnis

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference Proceedings*, volume 32, pages 37–45, 1968.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles And Practice (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Hey02] Heinrich Hey. *Photorealistic and Hardware Accelerated Rendering of Complex Scenes*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2002.

[Kaj86] James T. Kajiya. *The rendering equation*. ACM Press, New York, NY, USA, 1986.

[KR05] Emmett Kilgariff and Fernando Randima. The Geforce 6 Series Gpu Architecture. Technical report, NVIDIA Corporation, 2005.

[Krö01a] Detlef Krömker. Graphische Datenverarbeitung - Geometrische Transformation. Skript, 2001. http://www.gdv.informatik.uni-frankfurt.de/lehre/ss2001/GDV.html.

[Krö01b] Detlef Krömker. Graphische Datenverarbeitung - Texturen. Skript, 2001. http://www.gdv.informatik.uni-frankfurt.de/lehre/ss2001/GDV.html.

[LBOS04] Aaron Lefohn, Ian Buck, John D. Owens, and Robert Strzodka. GPGPU: General-Purpose Computation on Graphics Processors. In *Tutorial 3 at IEEE Visualization*, October 2004.

[LGM69] P.A. Lewis, A.S. Goodman, and J.M. Miller. A pseudo-random number generator for the system/360. *IBM Systems Journal*, 8:136–146, 1969.

[LSK+05] Aaron Lefohn, Shubhabrata Sengupta, Joe M. Kniss, Robert Strzodka, and John D. Owens. Dynamic adaptive shadow maps on graphics hardware. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*, August 2005.

[NVI06a] NVIDIA. NVIDIA GPU Programming Guide 2.5.0. Technical report, NVIDIA, 2006.

[NVI06b] NVIDIA. Release Notes for NVIDIA OpenGL Shading Language Support. Technical report, NVIDIA, 2006.

[Ope] OpenGL. The opengl framebuffer object extension. http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt; Stand Januar 2008.

[Pur04] Timothy J. Purcell. *Ray Tracing On A Stream Processor*. PhD thesis, Stanford University, 2004.

[Ros05] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[SA04] Mark Segal and Kurt Akeley. The OpenGL Graphics System. A Specification. Technical report, Silicon Graphics, Inc., 2004.

[SW01] Philipp Slusallek and Ingo Wald. Interaktive Beleuchtungssimulation und Bildsynthese mit Ray-Tracing. *Magazin 'Forschung' der Universität des Saarlandes*, 2001.

[Vea97] Eric Veach. *Robust Monte Carlo Methods For Light Transport Simulation*. PhD thesis, Stanford University, December 1997.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 153–164. Blackwell Publishing, 2001.