Leopold-Franzens-Universität Innsbruck
Institut für Informatik
Forschungsgruppe Infmath Imaging

Bakkalaureatsarbeit

Erweiterung des Raytracers

Markus Trenkwalder

28. Oktober 2005

Inhaltsverzeichnis

1 Einleitung

Das Ray tracing ist ein populäres Verfahren der 3D Computergrafik mit dem es möglich ist äußerst realistisch wirkende Darstellungen von dreidimensionalen Szenen zu erstellen. Dabei liegt dem Ray tracing ein relativ simpler Algorithmus zu Grunde, welcher auf physikalischen Gesetzen basiert.

Beim Ray tracing werden die Interaktionen zwischen Lichtstrahlen und Objekten einer Umgebung simuliert um daraus eine grafische Darstellung derselben erstellen zu können. Wie man bereits vermuten kann ist es notwendig eine sehr große Anzahl von solchen Strahlen zu berechnen um gute Resultate zu erzielen. Für jeden Strahl ist es erforderlich die Schnittpunke desselben mit den Objekten der Szene zu berechnen. Das Licht fällt dann auf jenes Objekt, dessen Schnittpunkt den geringsten Abstand vom Ursprung des Strahls hat. Beim Ray tracing sind gerade diese Schnittberechnungen die aufwändigsten Operationen und nehmen somit die größte Rechenleistung (95% ist ein realistischer Wert) in Anspruch.

Beim naiven so genannten *ausschöpfenden (engl. exhaustive)* Ray tracing wird jeder Strahl mit jedem Objekt getestet um einen Schnittpunkt zu bestimmen. Diese Methode mag zwar bei einfachen Szenen mit wenigen Objekten funktionieren wird aber bei komplexeren Szenen völlig unpraktikabel, da diese Herangehensweise eine lineare Zeitkomplexität (O(n)) bezüglich der Anzahl der Objekte besitzt.

Ein einfaches Beispiel kann diesen Umstand gut veranschaulichen: Man nehme an, dass man 1µs benötigt um die Schnittpunktberechnung zwischen einem Strahl und einem Dreieck zu bestimmen. Will man ein einfaches Bild mit einer Auflösung von 640x480 Bildpunkten berechnen so benötigt man mindestens 307200 Strahlen (für jeden Bildpunkt einen) für welche bestimmt werden muss ob es einen Schnittpunkt mit dem Dreieck gibt. Für das gesamte Bild benötigt man somit 0.3072 Sekunden. Möchte man nun ein sehr detailliertes 3D-Modell mit etwa 10 Million Dreiecken rendern müsste man etwa 35 Tage aufwenden. Dass dies völlig inakzeptabel ist versteht sich von selbst.

Für komplexe Szenen ist es notwendig die Leistung zu steigern. Die einfachste Methode ist es einen schnelleren Computer oder gleich einen Cluster zu verwenden. Als nächstes könnte man noch den Quellcode optimieren und bei modernen Prozessoren auf SIMD Extensions wie SSE und 3DNow! zurückgreifen welche für die Vektorrechnung nützlich sein können.

Obwohl durch diese Herangehensweise die Geschwindigkeit um einiges gesteigert werden kann ist es für sich alleine genommen nicht ausreichend um eine akzeptable Geschwindigkeit zu erreichen. Es benötigt somit verschiedene Techniken um den Ray tracing Algorithmus selbst zu optimieren um vom naiven, ausschöpfenden Ray tracing weg zu kommen.

2 Optimierungsmöglichkeiten

Arvo und Kirk [4] identifizieren drei unterschiedliche Strategien um das Ray tracing Verfahren zu beschleunigen:

- Den *durchschnittlichen Aufwand* der Schnittberechnungen verringern.
- Die *Gesamtzahl* der Strahlen vermindern.
- Strahlen durch eine generellere Einheit wie z.B. einen Kegel ersetzen und mit diesem rechnen.

Der durchschnittliche Aufwand von Schnittberechnungen kann verringert werden indem man die Schnittberechnung beschleunigt. Dies kann auf zwei Arten geschehen. Eine Möglichkeit bestehet darin die Strahl-Objekt Schnittberechnungen zu beschleunigen indem man zum Beispiel so genannte *Bounding Volumes* für komplexe Objekte verwendet. Eine weitere Möglichkeit zur Beschleunigung ist es weniger Strahl-Objekt Schnittberechnungen durchzuführen. Dies kann mit Techniken wie *Bounding Volume Hierarchien* oder *räumlichen Datenstrukturen*, welche sich das „Teile und Herrsche“ (engl. Divide-and-Conquer) Prinzip zu nutze machen, erreicht werden. Hier werden die Objekte in einer Baumstruktur organisiert. Diese Baumstruktur ermöglicht es für viele Objekte gleichzeitig zu entscheiden ob es überhaupt einen Schnittpunkt geben kann und erlaubt somit sich den viel größeren Aufwand der Schnittberechnung mit jedem Objekt zu vermeiden. Die Zeitkomplexität kann dadurch typischerweise von $O(n)$ auf $O(\log\ n)$ verbessert werden.

Eine weitere Strategie ist es die Gesamtzahl der zu berechnenden Strahlen zu vermindern. Die kann auf einfache weise erreicht werden, indem man auf Anti-Aliasing Techniken verzichtet oder die Auflösung des zu berechnenden Bildes reduziert. Leider bringt diese Vorgehensweise eine deutlich verminderte Bildqualität mit sich, was nicht immer akzeptabel ist. Eine andere Möglichkeit ist *Adaptive tree-depth control*. Anstatt die Rekursion beim Ray tracing Algorithmus erst beim Erreichen der maximalen Rekursionstiefe zu stoppen wird hier zusätzlich berücksichtigt wie groß der Anteil des, durch die Rekursion berechneten, Farbwerts am Pixel wäre. Wird festgestellt, dass der Anteil am Farbwert des Pixels kleiner als ein zuvor eingestellter Schwellenwert ist kann die Rekursion abgebrochen werden. Auf diese Weise ist es möglich Rechenzeit zu sparen ohne große Qualitätsverluste hinnehmen zu müssen. Eine weitere Möglichkeit die Anzahl der Strahlen zu minimieren sind statistische Optimierungen beim Anti-Aliasing. Durch das Anwenden von statistischen Analysen auf die bisher berechneten Samples ist es möglich zu bestimmen wie gut diese den realen Wert approximieren. Liegt der statistische Fehler innerhalb von vorgegebenen Grenzen geht man davon aus, dass man genug Samples für das aktuelle Pixel berechnet hat.

Als dritte Optimierungsstrategie nennen Arvo und Kirk generalisierte Strahlen. Hierbei wird das einfache Konzept des Strahls durch eine generellere Einheit wie zum Beispiel einen Kegel ersetzt. Der Kern der Technik liegt in der Idee mehrere Strahlen auf einmal zu berechnen. Techniken welche dieses Konzept verwenden sind das Beam tracing, Cone tracing und Pencil tracing.

3 Bounding Volumes und Hierarchien

3.1 Bounding Volumes

Ein *Bounding Volume* (BV) ist ein geometrischer Körper der ein oder mehrere Objekte vollständig umschließt. Dabei sollte ein BV ein einfacher geometrischer Körper sein. Beispiele für Bounding Volumes sind Kugeln, so genannte *Axis-aligned Bounding Boxes* (AABB; Box mit Seitenflächen parallel zu den Koordinatenachsen) und *Oriented Bounding Boxes* (OBB; Beliebig orientierte Box).

Beim Ray tracing erlaubt die Verwendung von Bounding Volumes einen einfacheren Test auf die Existenz eines Schnittpunktes. Nur wenn der Hüllkörper vom Strahl getroffen wird ist es notwendig den Schnittpunkt mit dem komplexeren Objekt zu finden. Obwohl hier zusätzlicher Rechenaufwand für jedes Objekt betrieben wird kann diese Methode die Geschwindigkeit erhöhen, da der weitaus häufiger auftretende Fall, dass ein Strahl das Objekt verfehlt, schneller festgestellt werden kann. Damit ein Bounding Volume effektiv ist sollte es das Objekt so eng wie möglich umschließen.

3.2 Bounding Volume Hierarchien

Bounding Volume Hierarchien (BVH) sind eine Erweiterung zu den Bounding Volumes und sind im Gegensatz zu diesen in der Lage die Anzahl der notwendigen Schnittberechnungen zu verringern und können somit eine logarithmische Zeitkomplexität ($O(\log n)$) erzielen.

Die Objekte einer Szene werden hierbei in einer hierarchischen Baumstruktur organisiert. Die einzelnen Objekte bilden dabei die Blätter des Baumes. Die inneren Knoten können mehrere Unterknoten besitzen und enthalten jeweils ein Bounding Volume, welches alle Unterknoten umschließt. Die Wurzel des Baumes umschließt somit den gesamten Raum einer Szene.

Unter Verwendung einer solchen Bounding Volume Hierarchie ist es nun möglich viele Objekte durch eine einzelne Schnittberechnung von weiteren Untersuchungen auszuschließen. Trifft ein Strahl das Bounding Volume eines Knotens nicht, so weiß man, dass man keinen der Unterknoten mehr betrachten muss.

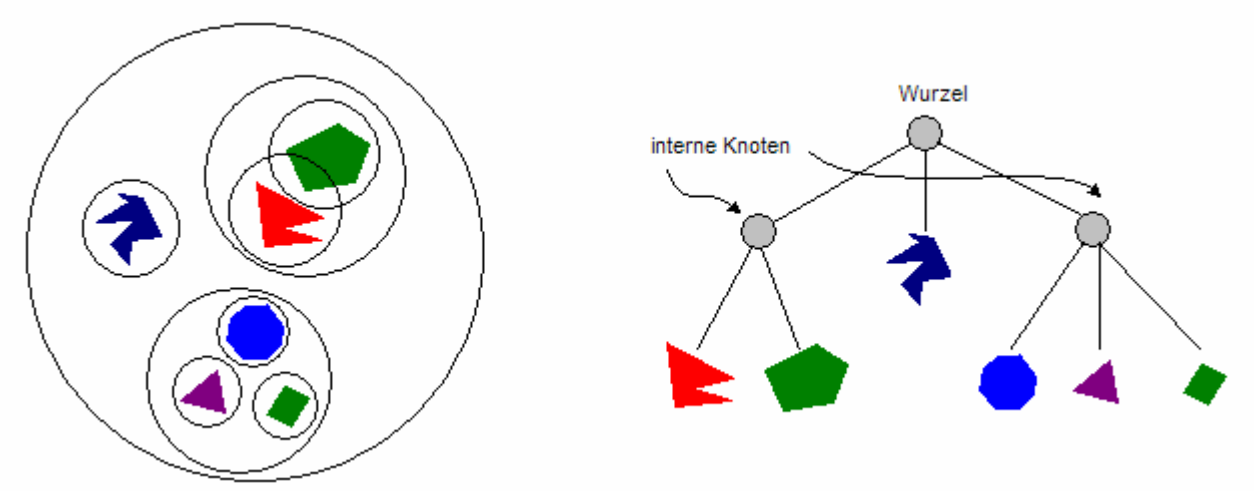


Abbildung 1: Bounding Volume Hierarchie

4 Räumliche Datenstrukturen

Räumliche Datenstrukturen können wie auch die Bounding Volume Hierarchien zum Beschleunigen des Ray tracing Verfahrens verwendet werden. Ähnlich wie bei den BVH machen sich die räumlichen Datenstrukturen einfache geometrische Körper zu nutze um die Objekte auf hierarchische Weise zu organisieren. Dadurch wird es ermöglicht jene Objekte zu bestimmen, welche sich in der näheren Umgebung des vom Strahl zurückgelegten Pfades befinden. Somit wird es möglich die Schnittberechungen auf diese wenigen Objekte zu beschränken.

Im Gegensatz zu den BVH werden bei den räumlichen Datenstrukturen die einzelnen umschließenden Körper nicht anhand der Objekte bestimmt, sondern durch wiederholtes aufteilen des dreidimensionalen Raumes in zwei oder mehrere disjunkte Unterräume. Anstatt eines *bottom-up* Verfahrens, bei dem die Objekte von immer größer werdenden Bounding Volumes umhüllt werden verwendet man hier ein *top-down* Verfahren. Der Raum wird sukzessive in kleinere Unterräume aufgeteilt. Diesen Unterräumen werden dann jene Objekte zugeteilt welche sich vollständig oder teilweise in diesem befinden.

Durch räumliche Datenstrukturen können verschiedenste Abfragen über den Raum und die Objekte beschleunigt werden. Durch die Verwendung einer Hierarchie ist es wie bei den BVH möglich eine Verbesserung von $O(n)$ auf $O(\log n)$ zu erzielen. Das Konzept der räumlichen Datenstrukturen kann auch auf den n-dimensionalen Raum erweitert werden. Im Allgemeinen ist die Konstruktion einer solchen räumlichen Datenstruktur ein zeitaufwendiger Prozess, was es verlangt diese Operation als Vorarbeitsschritt durchzuführen und die Ergebnisse für den späteren Gebrauch zu speichern.

4.1 BSP Trees

Binary Space Partitioning trees sind binäre Bäume, welche eine Ebene verwenden um den Raum in zwei Hälften zu teilen. Der Raum wird dabei in einen so genannten positiven und einen negativen Halbraum geteilt. Der positive Halbraum umfasst alle Punkte **x** für welche die Ebenengleichung $\mathbf{n} \cdot \mathbf{x} + d >= 0$ ist, der negative alle anderen. Der Parameter **n** beschreibt dabei den Normalenvektor der Ebene.

Ein BSP Tree wird erstellt indem der Raum durch eine Ebene in zwei Hälften geteilt wird und die Geometrie des Raumes in diese einsortiert wird. Der gesamte Prozess läuft nun rekursiv ab, indem die soeben erzeugten Hälften erneut geteilt werden und man die dort enthaltenen Objekte in die neuen Unterhälften einsortiert. Dies wird solange wiederholt bis ein gewisses Abbruchkriterium erreicht wird. Ein oft verwendetes Kriterium ist hierbei eine vom Benutzer eingestellte maximale Rekursionstiefe oder wenn die Anzahl der Polygone in einer Hälfte einen eingestellten Schwellenwert unterschreitet.

Es gibt viele verschiedene Varianten von BSP Trees, welche sich in der Wahl der Partitionsebene unterscheiden. Man kann grob zwischen den so genannten *Polygon-aligned* BSP Trees und den *Axis-aligned* BSP Trees unterscheiden. Allen BSP Trees liegt jedoch eine Eigenschaft zugrunde, welche es ermöglicht die enthaltene Geometrie ausgehend von einem beliebigen Betrachtungspunkt zu sortieren (ungefähr bei den Axis-aligned BSPs und exakt bei den Polygon-aligned BSPs), sofern man den Baum in einer bestimmten Weise durchläuft.

Polygon-Aligned BSP Tree

Ein Polygon-Aligned BSP Tree ist eine spezielle Art von BSP Tree bei welchem jeweils die Ebene eines Polygons aus der Szene als Partitionsebene verwendet wird. Die Ebene des Polygons wird verwendet um den Rest der Szene in zwei Mengen zu unterteilen. Polygone, welche die Partitionsebene schneiden, werden gespalten und die Hälften den entsprechenden Mengen zugeordnet. Der gesamte Prozess läuft jetzt rekursiv für die beiden Hälften ab. Für jeden Halbraum wird wieder ein Polygon ausgewählt, welches diesmal aber nur die Polygone im eigenen Halbraum teilt.

Teilt man eine Szene mittels eines Polygon-Aligned BSP Trees auf ergibt sich die interessante Eigenschaft, dass es nun möglich ist alle Polygone ausgehend von einem beliebigen Betrachtungspunkt exakt zu sortieren. Dies erlaubt es beispielsweise eine Szene ohne die Verwendung eines Z-Buffers zu zeichnen.

Einen effizienten Polygon-Aligned BSP Tree zu erstellen ist ein zeitintensiver Prozess. Die Effizienz hängt dabei stark von der Wahl des Polygons ab, welches zum Aufteilen der Szene verwendet wird. Ein gutes Polygon zu finden ist dabei selbst aufwändig und nicht trivial.

Axis-Aligned BSP Tree

Ein Axis-Aligned BSP Tree ist ebenfalls ein binärer Baum, welcher zum Unterteilen von Geometrie verwendet wird. Im Unterschied zu den Polygon-Aligned BSPs wird hier die Partitionsebene jedoch stets parallel zu den Koordinatenachsen gewählt.

Ein Axis-Aligned BSP Tree wird erstellt indem die gesamte Szene zuerst durch eine Axis-Aligned Bounding Box (AABB) umschlossen wird. Diese Box wird dann rekursiv in immer kleiner werdende Unterboxen unterteilt. Beim Unterteilen wird dabei jeweils eine Ebene gewählt, welche zu einer der drei Koordinatenachsen parallel verläuft.

Für die Wahl der Koordinatenachsen und der Ebene gibt es wieder viele verschiede Möglichkeiten. Eine Möglichkeit wäre beispielsweise die längste Koordinatenachse zu wählen und den Raum dann exakt in der Mitte zu teilen. Eine weitere Variante wählt die x-Achse für den Wurzelknoten aus und dann für die Kindknoten alternierend eine der drei Koordinatenachsen. BSP Trees, welche diese Variante wählen werden meist KD-Trees genannt.

Wie auch beim Polygon-Aligned BSP Tree ist auch hier die Wahl der Ebene für die Effizienz der Datenstruktur ausschlaggebend.

4.2 Octree

Ein Octree ist den Axis-Aligned BSP Trees ähnlich, da der Raum ausgehend von einer minimalen Axis-Aligned Bounding Box immer entlang der Koordinatenachsen geteilt wird. Im Gegensatz zu den BSP Trees wird hier der Raum aber nicht nur entlang einer einzigen Achse, sondern gleich entlang aller drei Achsen geteilt. Der Schnitt geht dabei stets durch das Zentrum der Axis-Aligned Bounding Box eines jeden Knotens, was eine reguläre Struktur zur Folge hat.

Ein Octree wird analog zu den BSP Trees erstellt. Eine Axis-Aligned Bounding Box wird rekursiv in jeweils acht kleinere Boxen aufgeteilt bis ein Abbruchkriterium den Prozess unterbricht. Gängige Abbruchkriterien sich auch hier die Anzahl von Objekten in den Blattknoten oder das Erreichen einer maximalen Rekursionstiefe.

Hat man nun den Raum mithilfe eines Octrees organisiert müssen beim Ray tracing nur mehr jene Objekte auf einen Schnittpunkt mit einem Strahl überprüft werden, welche sich in den Blattknoten befinden, welche vom Strahl getroffen werden. Dieser Umstand ist in den folgenden Bildern in 2D dargestellt wo diese Datenstruktur *Quadtree* genannt wird.

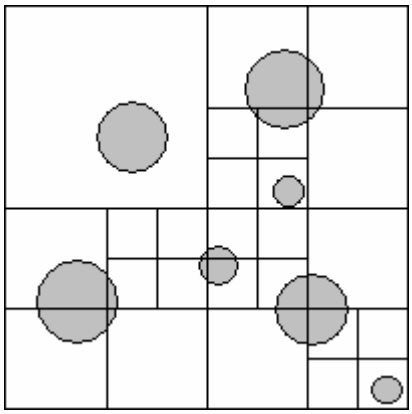


Abbildung 2: Quadtree mit maximal drei Ebenen Tiefe.
Aufgeteilt immer dann, wenn mehr als ein Objekt in einer Zone ist

4.3 Reguläres Gitter

Die Verwendung eines regulären dreidimensionalen Gitters stellt eine weitere Möglichkeit der Raumunterteilung dar. Die Idee ist dieselbe wie bei den bereits besprochenen Datenstrukturen, jedoch findet hier keinerlei Rekursion statt. Es wird ein fest vorgegebenes Gitter einheitlicher Größe verwendet um den Raum aufzuteilen. Beim Ray tracing werden dann die Voxel und die dazugehörigen Objekte in der Reihenfolge bestimmt und berechnet, in welcher sie vom Strahl durchlaufen werden.

Durch die Regelmäßigkeit der Datenstruktur ist es effizient die von einem Strahl durchlaufenen Voxel zu bestimmen. Ein weiterer Vorteil der Datenstruktur der aus der Regelmäßigkeit hervorgeht ist die schnelle Konstruktion. Durch die feste Vorgabe des Gitters ist diese Datenstruktur jedoch komplett unabhängig von der räumlichen Annordnung der Objekte und kann sich somit nicht an diese anpassen. In leeren Regionen werden gleich viele Voxel verwendet wie in Regionen mit vielen Objekten. Dadurch wird das Überprüfen von leerem Raum aufwändiger als bei BSP Trees und in Regionen mit sehr vielen Objekten ergibt sich auch kein Vorteil, da die Unterteilung hier womöglich zu grob ist. Für Regionen mit vielen Objekten ließe sich die Effizienz durch eine feinere Aufteilung natürlich steigern ist aber mit erheblichen Speicherkosten verbunden und verlangsamt zudem noch die Berechnungen in den leeren Knoten. Aufgrund der genannten Gründe ist diese Datenstruktur im Allgemeinen also weniger effizient als BSP Trees oder Octrees.

4.4 „Stolpersteine“

Will man eine der besprochenen räumlichen Datenstrukturen in einem Programm umsetzen, so gibt es zwei „Stolpersteine“, die einem Probleme bereiten können, wenn man nicht an sie denkt.

Ein Problem kommt auf, wenn man sich entscheiden muss wie man jene Objekte behandelt, die die Schnittebenen, welche den Raum aufteilen, schneiden. Hier gibt es mehrere verschiedene Möglichkeiten. Eine Möglichkeit ist es jedes Objekt selbst aufzuspalten und die beiden Hälften dann in die entsprechenden Raumhälften einzuordnen. Dies mag zwar bei 3D-Modellen bestehend aus Polygonen funktionieren, ist aber bei anderen Objekten, welche nicht in polygonaler Form vorliegen unpraktikabel. Eine weitere Möglichkeit besteht darin die Objekte nicht nur in den Blattknoten abzuspeichern. Die Objekte, welche die Schnittebenen kreuzen könnten auch in den inneren Knoten abgespeichert werden. Eine dritte Möglichkeit ist es Referenzen auf die Objekte in den Blattknoten zu speichern. Bei Objekten, welche die Schnittebene kreuzen wird eine Referenz in beiden Raumhälften abgespeichert. Verwendet man diese Möglichkeit muss man bedenken, dass es nun möglich ist, dass im Laufe der Schnittberechnung von einem Strahl mit der Szene ein und das selbe Objekt öfters getestet werden kann, da es ja in mehreren Voxeln vorkommen kann. Dies kann zu ungewollten Performanceeinbußen führen. Um doppeltes Testen zu vermeiden kann man so genannte *Mailboxes* (Briefkästen) verwenden. Jedem Strahl wird hier eine eindeutige Identifikationsnummer zugewiesen. Für jedes Objekt merkt man sich die Nummer des letzten getesteten Strahls zusammen mit den Ergebnissen. Will man nun den Schnittpunkt zwischen einem Strahl und dem Objekt berechnen, so vergleicht man zuerst die Nummer des Strahls mit der gespeicherten. Wenn beide Nummern gleich sind kann man sich ein erneutes Berechnen ersparen.

Eine weitere problematische Situation wird in Abbildung 3 dargestellt. In dieser Situation werden in den Voxeln Referenzen auf die Objekte gespeichert, die sich ganz oder teilweise in ihnen befinden. Beim Ray tracing werden die einzelnen Voxel dann in der Reihenfolge durchlaufen, in welcher der Strahl sie trifft. Kommt der Strahl in das Voxel mit der Nummer 3, so wird hier eine Referenz auf das Objekt B vorgefunden, mit welchem es auch einen Schnittpunkt gibt. Hört man jetzt auf durch die einzelnen Voxel zu wandern vergisst man das Objekt A, welches im dargestellten Fall auch vom Strahl getroffen wird und dessen Schnittpunkt vor dem des Objektes B liegt. Um Fehler dieser Art zu vermeiden darf man das Durchlaufen der Voxel erst dann stoppen, wenn der berechnete Schnittpunkt auch in dem Voxel liegt, in dem man sich befindet.

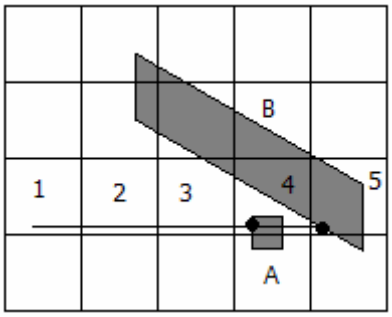


Abbildung 3

5 Implementierung

Im Rahmen der vorliegenden Bakkalaureatsarbeit habe ich den in meiner ersten Arbeit erstellten Ray tracer erweitert. Mein Ziel war es eine der besprochenen räumlichen Datenstrukturen zu implementieren um hoch detaillierte polygonale Modelle in akzeptabler Zeit rendern zu können.

Als Datenstruktur habe ich den Kd-tree gewählt, da ich mir von diesem eine gute Leistung bei vergleichsweise geringer Implementierungskomplexität versprach. Große polygonale 3D Modelle mit denen ich die Performance testen konnte habe ich aus dem *Stanford 3D Scanning Repository* [8] bezogen. Hier werden verschiedene 3D Modelle, welche mehrere Millionen Polygone besitzen und mittels eines 3D Scanners erstellt wurden, für wissenschaftliche Zwecke zum kostenlosen download angeboten.

Die folgenden Abschnitte dokumentieren die verschiedenen von mir ausgeführten Arbeitsschritte und Entscheidungen die ich getroffen habe um eine erfolgreiche Implementierung zu erzielen.

5.1 Das PLY Dateiformat

Die 3D Modelle aus dem 3D Scanning Repository liegen im so genannten PLY (kurz für Polygon; siehe [9]) Dateiformat vor. Dieser Umstand machte es somit notwendig sich mit diesem Dateiformat zu beschäftigen um die Daten laden zu können und somit die Grundlage für weitere Arbeitsschritte zu schaffen.

Das PLY Dateiformat wurde an der Stanford Universität entwickelt und wird zur einfachen Beschreibung von polygonalen Modellen verwendet. Eine Datei im PLY Format besteht dabei aus einem Header, welche die genaue Struktur und Menge der Daten spezifiziert. Nach dem Header folgen die so genannten *Vertex* Daten, welche die Koordinaten der Eckpunkte der einzelnen Polygone spezifizieren. Nach den Vertex Daten werden dann noch die Polygondaten selbst gespeichert. Für ein Polygon werden dabei die Anzahl der Eckpunkte und entsprechend viele Indizes in die Vertex Daten gespeichert. Eine interessante Eigenschaft dieses Dateiformats ist die sehr große Flexibilität. Es ist möglich beliebige Zusatzdaten für jeden Vertex zu speichern, und dies im Header zu spezifizieren. Es können auch noch weitere Daten wie beispielsweise Materialien gespeichert werden.

Der folgende Abschnitt gibt ein Beispiel zum Aufbau des Dateiformats. Hier werden die Daten eines Würfels abgespeichert, der eine Seitenlänge von zwei Einheiten hat und dessen Mittelpunkt im Koordinatenursprung liegt.

```
ply
format ascii 1.0
comment created by platoply
element vertex 8
property float32 x
property float32 y
property float32 z
element face 6
property list uint8 int32 vertex_indices
end_header
-1 -1 -1
1 -1 -1
1 1 -1
-1 1 -1
-1 -1 1
1 -1 1
1 1 1
-1 1 1
4 0 1 2 3
4 5 4 7 6
4 6 2 1 5
4 3 7 4 0
4 7 3 2 6
4 5 1 0 4
```

Ein Vorteil des PLY Datenformats ist, dass die Daten in einer vom Menschen lesbaren Art abgespeichert werden und durch die Abspeicherung als Text auch Plattformunabhängig sind. Textdateien haben allerdings den Nachteil, dass sie recht groß werden und das Auslesen der Daten langsamer ist. Genau aus diesem Grund gibt es auch eine Binärversion.

Um mit PLY Dateien einfacher arbeiten zu können habe ich den vom Georgia Institute of Technology bereitgestellten C Code verwendet. Dieser stellt Funktionen zum Öffnen und Auslesen der in einer Datei vorhandenen Informationen bereit und erlaubt es transparent Dateien im Text- und Binärformat zu verwenden.

Auf ein Problem bin ich bei Binärdaten allerdings gestoßen. Binärdaten können im Big Endian oder Little Endian Format abgespeichert werden. Beim Einlesen einer im Big Endian Format vorliegenden Datei auf einem PC waren die Daten nicht korrekt, da auf einem PC ein x86 basierter Prozessor zum Einsatz kommt, welcher eine Little Endian Architektur verwendet. Dieser Umstand wurde von den Bibliotheksroutinen nicht automatisch berücksichtigt, wodurch es notwendig war die Konvertierung manuell durchzuführen.

5.2 Aufbau des Kd-Tree

Den Aufbau des Kd-Trees zu implementieren hat mehr Aufwand erfordert als ich zuvor angenommen hatte. Es gab drei unterschiedliche Möglichkeiten mit Polygonen umzugehen, welche die Partitionierungsebenen des Kd-Tree kreuzten:

- Das Polygon aufteilen und die jeweiligen Hälften in die richtigen Halbebenen einteilen.
- Die Polygone nicht aufteilen, sondern im Knoten selbst speichern.
- Eine Referenz auf das Polygon in beiden Halbebenen speichern.

Zuerst hatte ich mich für die erste Möglichkeit entschieden, da ich hier beim Rendern den Spezialfall, dass Polygone teilweise außerhalb der Grenzen eines Halbraumes sein können nicht berücksichtigen gemusst hätte. Wie sich herausstellte war dies aber keine gute Wahl. Ein aus etwa 740000 Polygonen bestehendes 3D-Modell benötigte ungefähr 15 MB Speicher für die Vertex und Index Daten der Polygone. Durch das Aufteilen der Polygone welche die Schnittebene kreuzten war es notwendig neue Vertexe und zusätzliche Polygone zu erstellen, welche zusätzlichen Speicherplatz benötigten. Die Anzahl der Polygone hat sich dadurch je nach Tiefe des erstellten Kd-Tree verdoppelt bis verdreifacht, was zu einem dementsprechend höheren Speicheraufwand führte. Der Speicherbedarf war mir bereits viel zu groß und der Speicheraufwand für den Kd-Tree selbst war dabei noch nicht mitgerechnet.

Die erste Möglichkeit hatte ich damit verworfen. Ich ging dann zur zweiten Möglichkeit über, bei der keine zusätzlichen Vertex- und Polygondaten einzuführen waren und somit als zusätzlicher Speicheraufwand lediglich der Kd-Tree anfallen würde. Auch diese Möglichkeit stelle sich als nicht gut heraus. Es war mir zwar möglich Bilder zu rendern, doch ein Zeitaufwand von mehr als 25 Sekunden auf einem AMD Athlon 3500+ war mir viel zu groß. Zudem kam noch dazu, dass das gerenderte 3D-Modell „Löcher“ aufwies. Dieses Problem zeigt Abbildung 4.

Abbildung 4: "Löcher" als Fehler im Rendering

Ich war nicht in der Lage die Quelle des Fehlers zu finden. Ich vermute jedoch dass irgendwo in einer Berechnung numerische Fehler aufgetaucht sind, wodurch gewisse Polygone nicht richtig klassifiziert wurden, da weniger „Löcher“ im Bild auftauchten, wenn die Tiefe des Kd-Trees geringer war.

Im dritten Anlauf implementierte ich nun die letzte Möglichkeit, bei der Referenzen auf die Polygone, welche die Partitionsebene schneiden in beiden Baumknoten gespeichert werden. Diese Wahl stellte sich als gute Wahl heraus. Die Effizienz ist hervorragend. Ein 3D-Modell mit etwa 740.000 Polygonen kann jetzt in weniger als einer Sekunde gerendert werden.

Die folgenden Abschnitte gehen kurz auf die notwendigen Kenntnisse und Details zur Implementierung ein.

5.2.1 Berechnung von AABB und Partitionierung von Polygonen

Die Berechnung einer Axis-Aligned Bounding Box für eine Menge von Polygonen ist sehr einfach. Um eine AABB zu beschreiben sind lediglich zwei Vektoren notwendig, welche die minimalen und maximalen Koordinaten der Box speichern. Diese Koordinaten zu finden ist ein linearer Prozess. Man läuft einfach in einer Schleife durch alle Punkte der Polygone für die man die AABB berechnen möchte und bestimmt für jede Koordinatenachse unabhängig den minimalen und maximalen Wert.

Um Polygone partitionieren zu können bedarf es etwas mehr Mathematik. Zum bestimmen auf welcher Seite einer Ebene sich ein Polygon befindet, oder ob es die Ebene schneidet benötigt man die Koordinaten der einzelnen Eckpunkte und die Gleichung der Ebene. Die Ebenengleichung $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ wurde bereits in der ersten Bakkalaureatsarbeit besprochen. Den Abstand eines Punktes **x** zur Ebene kann berechnet werden, indem dieser in die Gleichung der Ebene eingesetzt wird. Befindet sich ein Punkt im positiven Halbraum ist $\mathbf{n} \cdot \mathbf{x} + d \geq 0$. Ist der Punkt im negativen Halbraum ergibt sich dementsprechend ein negativer Wert. Ein Polygon liegt im positiven Halbraum, wenn alle Eckpunkte des Polygons dort liegen. Ein Polygon wird von der Partitionierungsebene geschnitten, wenn nicht alle Punkte im selben Raum liegen.

5.2.2 Wahl der Schnittebene

Beim Kd-Tree wird abwechselnd die x, y und z-Achse als Schnittachse verwendet. Dabei kann sich die Schnittebene aber an einer beliebigen Position zwischen den Grenzen der AABB des jeweiligen Knotens des Baums befinden. Wie bereits erwähnt wurde hängt die Effizienz der Datenstruktur beim Rendern stark von der Wahl der Schnittebene ab.

Bestimmt man die Schnittebene für jeden Baumknoten so, dass auf beiden Seiten in etwa gleich viele Polygone liegen, so führt das zu einem ausgeglichenen Baum. Eine

Schnittebene zu finden, die dieses Kriterium genau erfüllt ist aufwendig und deshalb wird meist der Median der Koordinaten der Eckpunkte verwendet, welcher schneller zu bestimmen ist.

Man könnte meinen, dass ein ausgeglichener Baum die optimale Performance liefere, doch das ist zumindest beim Ray tracing gar nicht der Fall. Bei meinen Tests hat ein Kd-Tree bei dem immer der Median der Eckpunktkoordinaten der entsprechenden Achse verwendet wurde signifikant schlechtere Ergebnisse geliefert als die einfachere Methode, bei welcher die Schnittebene die AABB des Baumknotens exakt in der Mitte teilt.

Um eine möglichst ideale Schnittebene zu finden habe ich die in [5] besprochene Kostenfunktion auf mehrere denkbare Schnittebenen angewendet und jene gewählt, für welche die Kosten am geringsten waren. Mögliche Schnittebenen, welche ich betrachtet habe waren dabei das wegschneiden von leerem Raum auf der rechten und linken Seite, die Schnittebene in der Mitte des AABB, die Schnittebene durch den Median Vertex und das Terminieren der Rekursion. Die Kostenfunktion hat dabei folgende Form:

$$C_V = \frac{1}{SA(V)}\left[SA\left(leftChild(V)\right)\left(N_L + N_{SP}\right) + SA\left(rightChild(V)\left(N_R + N_{SP}\right)\right)\right]$$

C_V beschreibt hier die Kosten für das Volumens V, $SA(V)$ ist die Mantelfläche des Volumens V und damit proportional zur Wahrscheinlichkeit, dass dieses Volumen von einem Strahl getroffen wird. N_L, N_R und N_{SP} sind die Anzahl von Polygonen in den linken und rechten Unterhälften, sowie die Anzahl der von der Partitionsebene geschnittenen Polygone.

Diese Kostenfunktion liefert bereits sehr gute Ergebnisse kann aber noch verbessert werden. In [5] wird angesprochen, dass diese Kostenfunktion nicht berücksichtigt, dass auch die Kindknoten unterteilt werden und dadurch die tatsächlichen Kosten, abhängig von der Anzahl der Polygone in den Unterhälften, etwas geringer sind. Anstatt die tatsächliche Anzahl der Polygone zu verwenden wird vorgeschlagen die folgende Funktion auf die Anzahl der Polygone anzuwenden und diesen Wert in der obigen Formel einzusetzen:

$$f(n) = \begin{cases} n & \text{, falls } n \le n_0 \\ (n - n_0)^{1+\log_2 q} + n_0 & \text{, falls } n > n_0 \end{cases}$$

Die Variable n_0 gibt an, dass es sich nicht mehr lohnt einen Knoten aufzuteilen, wenn er bereits n_0 oder weniger Polygone enthält. Der Parameter $q \le 1$ ist ein Faktor der verwendet wird um die Effizienzsteigerung durch einen Schnitt anzugeben. Verwendet man $q = 1$, so ergibt sich keine Änderung zur linearen Kostenfunktion.

In den bei [5] durchgeführten Tests wurden für n_0 Werte von 4, 6 und 8 verwendet und für q die Werte 0.9, 0.95 und 0.98. Bei meinen Tests habe ich eine leichte Effizienzsteigerung im Vergleich zur linearen Kostenfunktion bei der Verwendung der Werte 0.95 und 0.98 für q feststellen können.

5.2.3 Speicherung des Kd-Tree

Da das Aufbauen des Kd-Tree für größere 3D Modelle ein recht aufwendiger Prozess ist, ist es sinnvoll diesen Prozess nur einmal auszuführen und das Ergebnis zu speichern. Da ein Kd-Tree mit 20 oder mehr Ebenen sehr viele Knoten hat und ich bei den beiden ersten misslungenen Versuchen festgestellt habe, dass der Bedarf an Speicherplatz für den Kd-Tree somit ziemlich groß werden kann habe ich versucht den Speicheraufwand für einen einzelnen Knoten gering zu halten. Für einen inneren Knoten benötige ich jetzt lediglich 32 Bit (2 Bits zur Angabe der Koordinatenachse und 30 Bits für die Position der Schnittebene). Blattknoten speichern neben den 32 Bits noch die Polygone.

5.3 Rendern

Nachdem der Kd-Tree erstellt und für den späteren Gebrauch gespeichert ist kann man sich nun an das Rendering wagen. Um ein polygonales 3D Modell mithilfe eines Kd-Trees mit dem in der ersten Bakkalaureatsarbeit erstellten Ray tracers rendern zu können musste dieser erweitert werden. Da ich den Ray tracer bereits mit dem Hintergedanken der möglichst einfachen Erweiterbarkeit entwickelt hatte war es nun nicht notwendig bestehenden Code zu ändern, sondern lediglich neuen Code hinzufügen indem ich eine neue Klasse erzeugte und von der IObject Schnittstelle ableitete. Für diese neue Klasse musste die Methode zum Berechnen eines Schnittpunktes mit einem Strahl implementiert werden.

Die Schnittpunktberechnung für ein Objekt welches mithilfe eines Kd-Trees beschleunigt wird läuft in drei Stufen ab. Zunächst wird geschaut ob der Strahl überhaupt die Axis-Aligned Bounding Box des gesamten Objektes trifft. Ist das nicht der Fall kann man mit den Berechnungen für den aktuellen Strahl bereits aufhören. Trifft der Strahl allerdings die AABB so muss man nun den Strahl durch den Kd-Tree verfolgen und immer wieder Schnitte mit den partitionierenden Ebenen berechnen, bis man an die Blattknoten des Baumes gelangt. Hier angelangt findet man nun eine Menge an Polygonen vor (in unserem Fall sind das ausschließlich Dreiecke), für welche man nun bestimmen muss ob diese vom Strahl getroffen werden. Befindet sich der, dem Ursprung des Strahls am nächsten liegende, Schnittpunkt innerhalb der Grenzen des Blattknotens, so hat man den Schnittpunkt gefunden und kann mit dem suchen aufhören, ansonsten muss man den nächsten Blattknoten im Baum finden und dort weitermachen.

Wie man den Schnittpunkt mit einem AABB, einer Ebene und einem Dreieck berechnen kann wurde bereits in meiner ersten Arbeit besprochen. Wie man in der richtigen Reihenfolge durch den Kd-Tree navigiert um die Blattknoten zu finden kann im folgenden Abschnitt nachgelesen werden.

5.3.1 Schnittpunktberechnung mithilfe des Kd-Trees

Um mithilfe des Kd-Trees Schnittpunkte effizient berechnen zu können muss man den Baum zunächst rekursiv durchlaufen bis man an die Blattknoten kommt, welche die eigentlichen Geometriedaten enthalten. Bei der Implementierung bin ich dabei nach dem in [4] angegebenen Pseudocode vorgegangen.

```
bool Mesh3D::kd_tree_intersect(
   const Ray& ray,
   const vec2& ray_interval,
   const Node *node,
   IntersectionInfo *iinfo
)
const
{
   if ( interval_empty(ray_interval) || node == NULL )
      return false;

   if ( node->is_leaf() ) {
      return triangles_intersect(ray, ray_interval, node, iinfo);
   }
   else {
      const float t = plane_intersect(ray, ...);

      // determine the near and far nodes
      ...

      // interval clipped to near side
      vec2 new_interval(ray_interval[0], std::min(t, ray_interval[1]));

      bool isect_found = kd_tree_intersect(ray, new_interval, near_node_ptr, iinfo);

      if ( !isect_found ) {
         // interval clipped to far side
         new_interval[0] = std::max(t, ray_interval[0]);
         new_interval[1] = ray_interval[1];

         isect_found = kd_tree_intersect(ray, new_interval, far_node_ptr, iinfo);
      }

      return isect_found;
   }
}
```

Der oben dargelegte C++ Code ist eine leicht gekürzte Fassung des gesamten Codes, zeigt aber alle relevanten Punkte auf.

Die Routine erhält als Parameter den Strahl und den zu testenden Knoten zusammen mit einem Zeiger auf den Speicherort, an dem schnittpunktspezifische Daten abgespeichert werden sollen. Weiters wird noch die Variable *ray_interval* übergeben. Dieses Intervall wird bei jedem rekursiven Aufruf verkleinert und gibt den Suchbereich für den Schnittpunkt an. Nur Schnittpunkte, welche sich innerhalb dieses Intervalls befinden, werden als gültig betrachtet; dies adressiert den zweiten der angesprochenen Stolpersteine.

Als erstes wird überprüft ob das Intervall leer ist. Ist dies der Fall, so kann man für den entsprechenden Unterbaum die Suche abbrechen. Im anderen Fall ist das Intervall nicht leer und man unterscheidet zwischen einem Blattknoten und einem inneren Knoten. Ist der Knoten ein Blattknoten so müssen die Dreiecke auf einen Schnittpunkt überprüft werden. Ist es hingegen ein innerer Knoten müssen die Unterbäume überprüft werden. Zunächst bestimmt man den Schnittpunkt mit der Partitionsebene. Dieser Schnittpunkt wird zur Unterteilung des Suchintervalls verwendet. Das Suchintervall wird verkürzt und die Suche nach einem Schnittpunkt in dem, dem Ursprung des Strahls nahe liegenden, Unterknoten fortgesetzt. Konnte in diesem Knoten kein Schnittpunkt gefunden werden muss man die Suche im anderen Unterknoten mit dem entsprechenden Intervall fortsetzen.

5.4 Parallelisierung mit MPI

Im Rahmen dieser Bakkalaureatsarbeit habe ich den erweiterten Ray tracer auch mit MPI parallelisiert. MPI steht für *Message Passing Interface* und ist ein Protokoll, welches zur Kommunikation zwischen mehreren parallel ablaufenden Prozessen, welche sich in einem verteilten Rechensystem befinden können, verwendet wird. Das MPI API bietet verschiedene Funktionen um Informationen zwischen den einzelnen Prozessen, welche parallel und in Kooperation am selben Problem arbeiten, auszutauschen. MPI wird im High Performance Computing Bereich auf Clustern und Supercomputern eingesetzt.

Als Ausgangspunkt für die Parallelisierung mit MPI war der programmierte Ray tracer gegeben. Zunächst musste ich nun überlegen, wie man die auszuführenden Berechnungen aufteilen sollte, sodass diese von mehreren Prozessen parallel bearbeitet werden können. Eine Möglichkeit, die sich anbietet ist die Aufteilung der einzelnen zu berechnenden Pixel auf die verschiedenen Prozesse. Diese Möglichkeit funktioniert auch hervorragend gut, denn der Farbwert eines jeden Pixels kann unabhängig von allen anderen berechnet werden, sodass während der Berechnungen kein Austausch von Informationen notwendig ist. Die Aufteilung der Arbeit auf Pixelbasis skaliert auch besonders gut bei einer steigenden Anzahl von Rechnern, da es in einem Bild ausreichend viele Pixel gibt um die Arbeit auf viele Rechner zu verteilen und auch kein zusätzlicher Kommunikationsaufwand anfällt.

Als ich mich nun entschieden hatte die Rechenarbeit auf Pixelbasis aufzuteilen blieb noch die Frage offen wie man entscheidet, welcher Prozess welche Pixel zu berechnen hat. Hierfür gibt es mehrere Lösungsansätze:

- *Aufteilung des Bildes in Streifen*

  Eine Möglichkeit die Arbeit auf n Prozesse aufzuteilen ist es das Bild in n gleich große Streifen (horizontal oder vertikal) zu unterteilen und jeden Prozess die Pixel von jeweils einem solchen Streifen berechnen zu lassen. Eine weitere denkbare Möglichkeit wäre die Aufteilung des Bildes anhand eines $n \times m$ Gitters.

  Damit dieser Lösungsansatz gut funktioniert muss man annehmen, dass alle Prozesse auf gleich schnellen Rechnern ablaufen und die Berechnungsdauer für jedes Pixel und somit auch jedes Streifens gleich ist. Ob die erste Annahme gilt kann leicht überprüft werden, doch dass die Zweite trifft in der Praxis nie zu, da die Berechnungsdauer für jedes Pixel von den getroffenen Objekten und deren Materialien abhängt (Reflexionen zu berechnen dauert z.B. länger als Berechnungen für matte Oberflächen).

- *Jedes n-te Pixel von Prozess mit Nummer n berechnen lassen*

  Diese Art der Aufteilung funktioniert in der Praxis um einiges besser als die zuvor besprochene. Die Annahme von vorher, dass jedes Pixel die gleiche

Berechnungsdauer erfordert hält in der Praxis nicht, doch dass benachbarte Pixel meist einen ähnlichen Zeitaufwand zur Berechnung benötigen trifft in der Praxis zu. Dies deshalb, da benachbarte Pixel meist zum selben Objekt mit denselben Materialeigenschaften gehören und somit einen ähnlichen Berechnungsaufwand erfordern. Genau diese Eigenschaft macht es auf, dass diese Art der Aufteilung so viel besser funktioniert.

- *Dynamische Arbeitsverteilung*

  Ein Problem der bisherigen Ansätze war die Annahme, dass alle Rechner die Prozesse gleich schnell abarbeiten. Sollte dies nicht der Fall sein, so ist die gleichmäßige Aufteilung der Arbeit nicht geeignet, da manche Prozesse nun mit der Arbeit früher fertig sind als andere und auf diese warten müssen.

  Dieses Problem kann nur durch dynamische Arbeitsverteilung gelöst werden, was es, im Unterschied zu den vorher besprochenen Möglichkeiten, jedoch erforderlich macht, dass die einzelnen Prozesse während der Berechnungen Informationen über den Berechnungsfortschritt austauschen und Arbeit wenn notwendig umverteilen.

Für meine Bakkalaureatsarbeit habe ich die ersten beiden Aufteilungsmöglichkeiten implementiert.

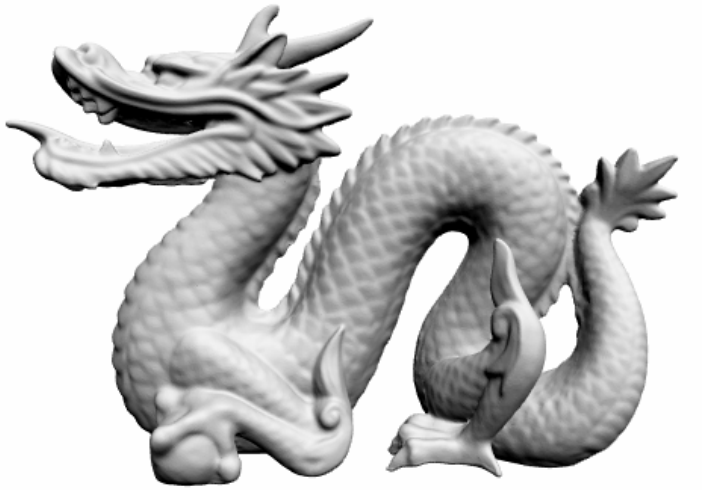
Nachdem die einzelnen Prozesse ihren Teil des Bildes berechnet haben bleibt ein letzter Arbeitsschritt übrig. Die berechneten Daten müssen gesammelt und abgespeichert werden. In meiner Implementierung senden deshalb alle Prozesse nach Abschluss der Berechnungen ihre Daten an einen Hauptprozess, welcher die Daten von den einzelnen Prozessen empfängt, in einem Bild zusammenstellt und abspeichert.

6 Ergebnisse

6.1 Kd-Tree Performance

Um die Performance des erweiterten Ray tracers und des implementierten Kd-Trees zu testen habe ich neue Testszenen angefertigt und den Kd-Tree für die 3D Modelle mit verschiedenen Parametereinstellungen erstellt und jeweils die Renderzeiten gemessen.

Für die erste Testszene habe ich das 3D Modell einer Drachenstatue mit 871414 Polygonen verwendet und von verschiedenen Ansichten gerendert. Gerendert wurde bei einer Auflösung von 640x480 Bildpunkten und 8x Antialiasing.



		Größe	Ansichtrenderzeit [s]				Anzahl Schnitte			
	q	KB	A	B	C	D	Median	Center	Cut-off	Bemerkung
1	1,00	25619	5,453	7,344	6,219	6,672	0	?	0	Erste Implement.
2	0,98	26586	4,656	6,344	5,266	5,734	0	75579	32433	Kein Median Split
3	0,98	37649	6,438	10,063	7,172	8,906	168164	0	63584	Kein Center Split
4	0,90	24097	5,000	6,734	5,687	6,516	15382	34788	31852	q = 0.90
5	0,95	25730	4,671	6,359	5,297	5,797	19741	48187	33744	q = 0.95
6	0,98	26471	4,656	6,313	5,203	5,797	22447	53794	32914	q = 0.98
7	1,00	27400	4,703	6,391	5,187	5,750	27018	60027	34188	q = 1.00

Die obige Tabelle zeigt verschiedene Werte wie Renderzeiten, Anzahl und Art der durchgeführten Schnitte beim Erstellen des Kd-Trees und auch die Größe der resultierenden Datei, welche die Geometriedaten zusammen mit dem Kd-Tree speichert. Für die obige Testszene wurde für den Parameter n_0 (siehe 5.2.2) der Wert 5 gewählt. Die Baumtiefe wurde auf 20 Ebenen beschränkt.

Für eine zweite Testszene habe ich ein aus 10 Millionen Polygonen bestehendes 3D Modell einer asiatischen Statue verwendet. Mehrere verschiedene Ansichten wurden dabei wieder mit einer Auflösung von 640x480 Bildpunkten und 8x Antialiasing gerendert. Für den Parameter n_0 wurde der Wert 4 verwendet.

		Größe	Ansichtrenderzeit [s]				Anzahl Schnitte			
	q	KB	A	B	C	D	Median	Center	Cut-off	Bemerkung
1	0,90	252473	4,703	6,782	6,031	8,109	76957	268333	223381	Baumtiefe 24
2	0,95	261143	4,406	6,328	5,625	7,516	93096	337919	235783	Baumtiefe 24
3	0,98	265009	4,360	6,266	5,594	7,422	105559	367431	230412	Baumtiefe 24
4	1,00	268162	4,438	6,328	5,563	7,609	120452	380783	252995	Baumtiefe 24
5	0,98	205333	12,844	17,047	17,203	23,828	12968	55786	31008	Baumtiefe 20

Betrachtet man die Renderzeiten in den beiden Tabellen, so kann man feststellen, dass der Ray tracer die besten Ergebnisse bei einem Wert von 0,98 für den Parameter q liefert.

In dieser Arbeit wurde auch angesprochen, dass ein balancierter Baum für das Rendern nicht unbedingt ideal ist. Dies wird ersichtlich, wenn man die Zeilen 2, 3 und 6 aus der ersten Tabelle miteinander vergleicht. Sucht man sich immer nur den Median einer jeweiligen Koordinate aus und verwendet diesen für die Schnittebene so ergeben sich wesentlich schlechtere Renderzeiten, als wenn man einfach immer

nur in der Mitte der AABB teilt. Sogar die vom Aufbauen des Kd-Trees resultierende Datei ist beim der Wahl des Medians größer. Betrachtet man bei der Wahl der Schnittebene sowohl die Möglichkeit des Teilens am Median der Koordinaten sowie des Teilens in der Mitte und wählt dann mithilfe der in 5.2.2 erwähnten Kostenfunktion die günstigere Möglichkeit ergibt sich Zeile 6 in der ersten Tabelle. Vergleich man diese mit Zeile 2 kann man aber nur einen insignifikanten Unterschied bei den Renderzeiten feststellen.

6.2 Parallelisierung

Da ich den Ray tracer für diese Bakkalaureatsarbeit auch parallelisiert hatte war es mir möglich diesen auf dem Institutscluster zu testen. Am Cluster konnte ich 8 Pentium III Rechner mit 1.26 Ghz und 1GB Ram für meine Berechnungen verwenden. Damit der Cluster auch wirklich ordentlich ausgenutzt werden konnte habe ich das äußerst rechenintensive *Monte Carlo Path Tracing* implementiert mit welchem es möglich ist globale Beleuchtungseffekte wie beispielsweise die diffuse Reflexion zu simulieren. Mit dem Monte Carlo Path Tracing kann eine Approximation für die, in der ersten Bakkalaureatsarbeit besprochene, *Rendering Equation* gefunden werden. Der Nachteil ist allerdings, dass es eine sehr hohe Anzahl an Samples pro Pixel benötigt (was es so zeitintensiv macht) um dass Bildrauschen, welches bei diesem Verfahren auftritt zu reduzieren.

Die Pixel wurden an die einzelnen Prozesse so aufgeteilt, sodass jedes *n*-te Pixel vom Prozess mit der Nummer *n* berechnet wird. Da alle verwendeten Rechner des Clusters gleich schnell sind skaliert diese Aufteilungsart ausgezeichnet; alle Rechner benötigen in etwa gleich lange um ihre Arbeit zu erledigen.

Das oben dargestellte Bild wurde in einer Auflösung von 640x480 Bildpunkten mit dem Monte Carlo Path Tracing berechnet und stellt eine Statue bestehend aus 10 Millionen Polygonen dar. Als Lichtquelle dient eine von Paul Debevec's *Light Probes* [7], welche Informationen der Beleuchtungsverhältnisse der Umgebung enthält. Gut zu erkennen sind die weichen Schatten am Fuß der Statue. Das Berechnen des Bildes bei 4800 Samples pro Pixel (so viele, damit das Rauschen gering genug ist) dauerte am Cluster etwa 50 Minuten. Diese Zeit könnte allerdings noch wesentlich verringert werden, indem z.B. nicht für alle Pixel 4800 Samples berechnet werden, sondern nur für jene, bei denen die statistische Varianz der einzelnen Samples größer als ein eingestellter Wert ist.

Referenzen

[1] Andrew S. Glassner. „An Introduction To Ray Tracing". Academic Press Inc., 1989

[2] Tomas Akenine-Möller und Eric Haines. „Real-Time Rendering" 2. Ausgabe. A.K. Peters, Ltd. 2002

[3] Paul S. Heckbert. „Writing a ray tracer". In An Introduction to Ray Tracing von Andrew S. Glassner. Academic Press Inc., 1989

[4] James Arvo und David Kirk. „A Survey of Ray Tracing Acceleration Techniques". In An Introduction to Ray Tracing von Andrew S. Glassner. Academic Press Inc., 1989

[5] László Szécsi und Balázs Benedek. „Improvements on the kd-tree". http://www.iit.bme.hu/~szirmay/katt/Szecsi.pdf

[6] Eric Haines. "Essential Ray Tracing Algorithms". In An Introduction to Ray tracing von Andrew S. Glassner. Academic Press Inc., 1989

[7] Paul Debevec. Light Probe Image Gallery. http://www.debevec.org/Probes/

[8] Stanford 3D Scanning Repository http://graphics.stanford.edu/data/3Dscanrep/

[9] The PLY File Format http://www.cc.gatech.edu/projects/large_models/ply.html

[10] 3D Object Intersections. http://www.realtimerendering.com/int/

[11] Recursive Ray Tracing. http://www.cs.fit.edu/wds/classes/adv-graphics/raytrace/raytrace.html

[12] The Recursice Ray Tracing Algorithm
http://www.geocities.com/jamisbuck/raytracing.html

[13] Some Mathematics for Advanced Graphics
http://www.cl.cam.ac.uk/Teaching/1999/AGraphHCI/SMAG/

[14] Introduction to Computer Graphics
http://medialab.di.unipi.it/web/IUM/Waterloo/html.html

[15] Exposure Control
http://freespace.virgin.net/hugo.elias/graphics/x_posure.htm

[16] Ray Tracing: Graphics for the Masses
http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

[17] Monte Carlo Ray Tracing
http://www.math.hmc.edu/~wchang/graphics_papers/Monte%20Carlo%20Ray%20Tracing%20-%20Siggraph%20Course%202003.pdf