TU DRESDEN

Fakultät Informatik

Proseminar Computergraphik

Dr. Mascolus

Belegarbeit

Optimierungsalgorithmen

– Effektives Rendern in der Computergraphik –

vorgelegt von:

Student:	Benjamin Schneider
Studiengang:	Informatik
Fachsemester:	4
Matrikelnummer:	3425954
E-Mail:	s9056596@mail.inf.tu-dresden.de

Dresden, den 14.07.2009

Inhalt

1 EINLEITUNG 1

2 GRUNDLAGEN 2

2.1 Der Sichtkegel 2

2.2 Geometrie aus Dreiecken 2

2.3 Die Grafikpipeline 3

3 ALGORITHMEN IM DETAIL 4

3.1 BSP-Trees 4

3.2 Quadtrees 6

3.3 Octrees 8

4 WEITERE TECHNIKEN 9

4.1 ROAM 9

4.2 Geometrical Clipmapping 10

4.3 Kd-Tree 10

5 VERBESSERUNGEN 11

5.1 GeoMipMapping 12

6 FAZIT 13

7 LITERATURVERZEICHNIS 15

Verzeichnis der Abbildungen

Abb. 1: Viewfrustum....................2
Abb. 2: Delphin als Dreiecksnetz....................3
Abb. 3: Grafikpipeline....................3
Abb. 4: Unser Ausgangsraum....................4
Abb. 5: Erster Schritt....................5
Abb. 6: Zweiter Schritt....................5
Abb. 7: Dritter Schritt....................6
Abb. 8: Unterteilung des Terrains mittels Quadtree....................7
Abb. 9: Sichtbarkeitstest an der Wurzel....................7
Abb. 10: Rekursive Sichtbarkeitsprüfung....................8
Abb. 11: Octree mit Viewfrustum....................9
Abb. 12: Geoclipmapping....................10
Abb. 13: kd-Tree....................11
Abb. 14: Viewfrustum-Culling auf dem Quadtree....................12
Abb. 15: GeoMipMapping....................13

Verzeichnis der Abkürzungen

CPU	Central Processing Unit
GPU	Graphics Processing Unit
KI	Künstliche Intelligenz
FPS	Frames per Second
BSP	Binary Space Partitioning
PVS	Potentially Visible Set
ROAM	Realtime Optimally Adapting Meshes
LOD	Level of Detail
DLOD	Discrete Level of Detail

1 Einleitung

Die Motivation für den Einsatz von Optimierungsalgorithmen ist im Allgemeinen das bestmögliche Ausnutzen von gegebenen Ressourcen. Auf dem Gebiet der Computergraphik sind diese Ressourcen in erster Linie Rechenleistung auf CPU und GPU, sowie Hauptspeicher und Festplattenkapazität. Um diese Ressourcen optimal zu nutzen versucht man CPU und GPU möglichst parallel arbeiten zu lassen und unnötige Berechnungen im Vorfeld auszuschließen. Erreicht wird dies heutzutage durch Techniken wie Threading – welches durch Mehrkernprozessoren noch unterstützt wird - und effiziente Optimierungsalgorithmen, die darauf abzielen unsichtbare Bildbereiche oder (3D-)Objekte vor dem Rendern auszusortieren. Somit wird Rechenleistung für andere Berechnungen - wie beispielsweise Physiksimulation oder KI - verfügbar.

Ich werde im Zuge dieser Ausarbeitung einige dieser Algorithmen vorstellen. Zunächst jedoch sollen ein paar grundlegende Kenntnisse, welche die Darstellung virtueller Welten in der Computergraphik betreffen, vermittelt werden.

2 Grundlagen

2.1 Der Sichtkegel

Vom sog. „Sichtkegel“ (engl. Viewfrustum) wird der Ausschnitt der virtuellen Welt abgegrenzt der später auf dem Bildschirm wirklich sichtbar sein wird.

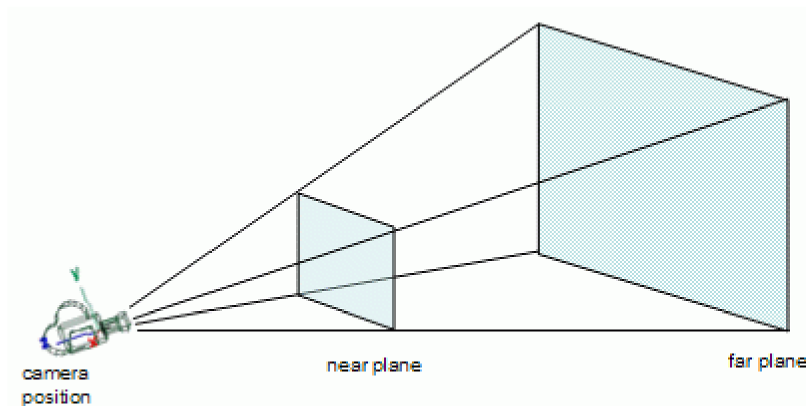

Abb. 1: Viewfrustum

Punkte bzw. Objekte die außerhalb des Sichtkegels liegen werden nach der Verarbeitung durch die Grafikkarte im finalen Bild nicht sichtbar sein. Sie können also schon frühzeitig von zeitraubenden Berechnungen wie Beleuchtung und Texturierung ausgeschlossen werden. Um festzustellen, ob ein Objekt außerhalb liegt, braucht man nur zu prüfen ob es hinter einer der sechs Ebenen liegt, die das Viewfrustum bilden. Um nicht jeden einzelnen Eckpunkt eines Objektes prüfen zu müssen, werden Bounding Volumes eingesetzt die das Objekt vollständig umschließen und die an Stelle des komplexen (3D-)Objektes auf Sichtbarkeit untersucht werden. Diesen ganzen Vorgang nennt man View Frustum Culling.

2.2 Geometrie aus Dreiecken

In der Computergraphik werden zur Darstellung von 3D-Objekten häufig Dreiecke verwendet, welche so zusammengesetzt werden, dass eine lückenlose Hülle entsteht. Mit Hilfe von Dreiecke lassen sich nahezu alle Objekte darstellen oder annähern. Ein Bild soll dies veranschaulichen:

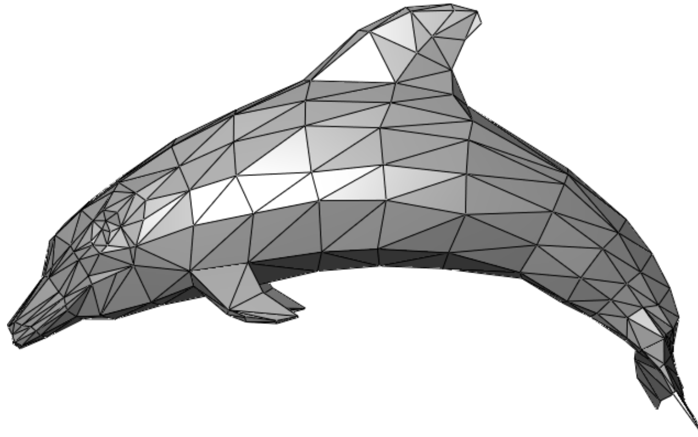

Abb. 2: Delphin als Dreiecksnetz

Die Dreiecke können nun transformiert, beleuchtet, schattiert und texturiert werden um das Objekt im virtuellen Raum zu bewegen und noch realistischer wirken zu lassen.

2.3 Die Grafikpipeline

Um deutlich zu machen wo die meisten der hier vorgestellten Algorithmen ansetzen, bedarf es einem Einblick in die Entstehung des fertigen Bildes auf dem Bildschirm. Die Grafikpipeline ist eine Modellvorstellung in der Computergrafik, die beschreibt, welche Schritte ein Grafiksystem zum Rendern, also zur Darstellung einer 3D-Szene auf einem Bildschirm, durchführen muss. Sie lässt sich in drei große Schritte einteilen: Anwendung, Geometrie und Rasterung.[1]

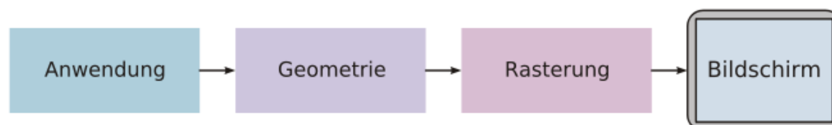

Abb. 3: Grafikpipeline

Die Anwendung schickt die zu zeichnenden 3D-Objekte an die Grafikkarte, wo der Geometrieschritt ausgeführt wird. In diesem Schritt werden die Dreiecke nun beleuchtet, texturiert und transformiert. Da dieser Schritt sehr rechenaufwändig ist, wird bei den hier vorgestellten Algorithmen versucht in der Anwendung selbst so viele Objekte

[1] Tomas Akenine-Möller, Eric Haines: *Real-Time Rendering,* S. 11

und damit Dreiecke wie möglich auszusortieren (engl. Culling). Somit werden nur noch die Dreiecke abgearbeitet, die auch wirklich sichtbar sind.

Der letzte Schritt – die Rasterung – dient nun dazu das fertig gezeichnete Bild auf die Bildschirmpixel zu projizieren.

3 Algorithmen im Detail

In diesem Kapitel werden nun drei Algorithmen etwas näher vorgestellt. Es handelt sich dabei um BSP-, Quad- und Octrees.

3.1 BSP-Trees

Die Abkürzung BSP steht für „Binary Space Partitioning“, was soviel bedeutet wie binäre Raumaufteilung. Angewandt wurde diese Technik schon vor einigen Jahren in Computerspielen wie „Doom“ und „Quake“. Sie dient dazu das Zeichnen von Indoor-Levels, also der Spielewelt, zu beschleunigen. Erreicht wird dies durch die rekursive Unterteilung des Levels in konvexe Teilräume. In diesen konvexen Teilräumen überdeckt nun kein Polygon mehr ein anderes. Bei dieser Unterteilung wird immer versucht möglichst gleichgroße Teilräume zu schaffen und keine Polygone zu zerschneiden, denn dies würde später den Renderaufwand wieder erhöhen. Die Teilung selbst erfolgt mit Hilfe von Ebenen, die meist so liegen, dass sie mit Polygonen der Levelgeometrie zusammenfallen.

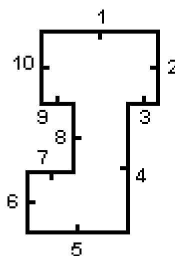

Abb. 4: Unser Ausgangsraum

In Abb. 4 sehen wir unseren Ausgangsraum. Dieser soll nun nach dem BSP-Algorithmus unterteilt werden. Die Polygone im Bild haben neben einer Nummer auch einen Normalenvektor – dargestellt durch kleine Striche - der angibt wo ihre Vorderseite ist.

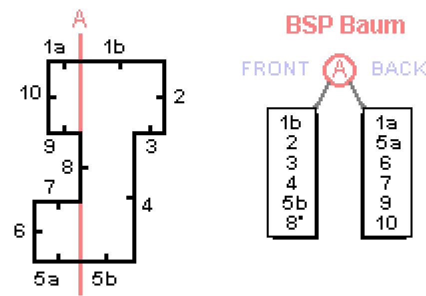

Abb. 5: Erster Schritt

Im ersten Schritt wir nun der Ausgangsraum in zwei Teilräume aufgesplittet. Dabei wird eine Teilungsebene durch das Polygon Nummer acht gelegt und alle Polygone auf der Vorderseite der Ebene kommen in die Frontliste inklusive des Teilungspolygons selbst. Die Orientierung der Ebene ist die gleiche wie die des Teilungspolygons. Alle anderen liegen auf dessen Rückseite und werden somit der Backliste zugeordnet. Man sieht das bei dieser ersten Unterteilung bereits zwei Polygone zerschnitten wurden. Dadurch entstehen vier neue Polygone, die entsprechend mit „a“ und „b“ beschriftet wurden. Der Grund für das Zerschneiden ist, dass eine möglichst gleichmäßige Teilung angestrebt wird. Man muss hier also abwägen zwischen dem Gleichgewicht von Front- und Backliste und dem Zerteilen von Polygonen (denn je ausbalancierter der Baum ist, desto schneller können später große Teile aussortiert werden).

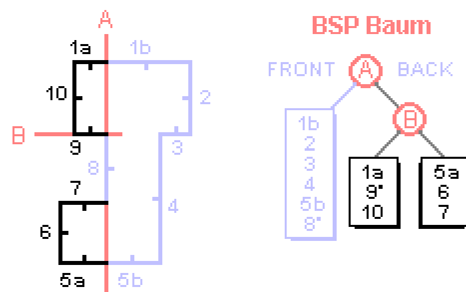

Abb. 6: Zweiter Schritt

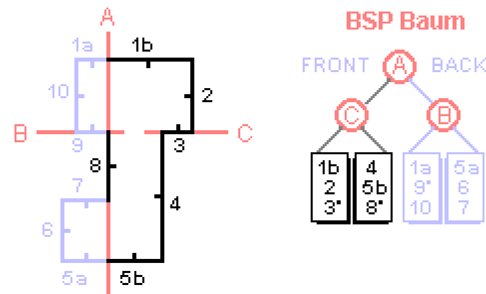

Abb. 7: Dritter Schritt

In den Schritten zwei und drei werden nun die beiden Teilräume aus dem ersten Schritt nochmals aufgeteilt und es entstehen konvexe Unterräume, welche man auch als die Blätter (engl. Leafs) des BSP-Baumes bezeichnet. Hingegen sind A, B und C die sog. Knoten (engl. Nodes) des Baumes.

Mit Hilfe des BSP-Baumes können nun Sichtbarkeitstests und Kollisionserkennung maßgeblich beschleunigt werden. Durch Bounding Box-Tests kombiniert mit einem PVS-System bzw. Portalen, welche die Leafs verbinden, kann schnell entschieden werden ob ein Leaf sichtbar ist oder ob evtl. ganze Teile des Baums hinter dem Betrachter liegen und nicht gerendert werden müssen. Ein solches PVS speichert in jedem Leaf eine Liste von allen von diesem Leaf aus sichtbaren Blättern. So werden Unterräume die durch Wände verdeckt sind aussortiert und müssen nicht von der Grafikkarte verarbeitet werden. Da die Berechnung und Speicherung des PVS im Vorfeld geschehen kann beeinflusst sie die Framerate der Anwendung nicht. Der Nutzer bzw. Spieler wird lieber ein paar Sekunden Wartezeit beim Programmstart in Kauf nehmen, statt beim spielen ein ständiges Ruckeln des Bildes hinnehmen zu müssen.

3.2 Quadtrees

Beim Quadtree handelt es sich um eine Technik die vorwiegend im Outdoor-Rendering, z.B. für die Darstellung von Landschaften (engl. Terrain), eingesetzt wird. Hier wird unsere Landschaft im Gegensatz zum BSP-Tree statt in zwei in vier Teilstücke aufgespalten. Somit entstehen nach jeder Teilung vier Kinder aus jedem Knoten des Baumes. Diese Unterteilung wird wiederum solange rekursiv fortgeführt, bis ein bestimmtes Abbruchkriterium erreicht ist. Solch ein Abbruchkriterium könnte beispielsweise eine festgelegte Mindestgröße der Blätter oder eine gewisse Tiefe des Baums

sein. In den Blättern des Baumes werden letztendlich die Geometriedaten der Landschaft gespeichert.

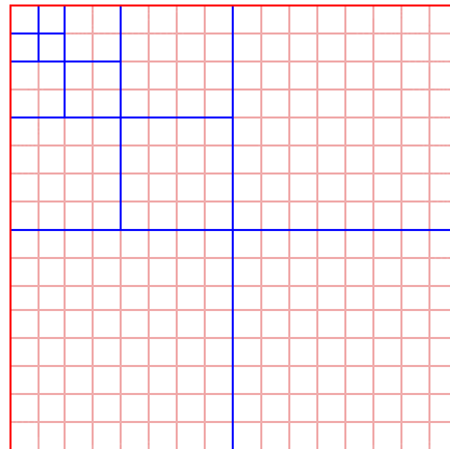

Abb. 8: Unterteilung des Terrains mittels Quadtree

In Abb. 8 sieht man wie das Terrain immer weiter unterteilt wird bis schließlich die Blätter erreicht sind. Will man das Terrain nun rendern können mittels Viewfrustum-Culling schnell große Teile aussortiert werden. Dies geschieht indem die Bounding Boxen der Baumknoten von der Wurzel an auf Sichtbarkeit geprüft werden.

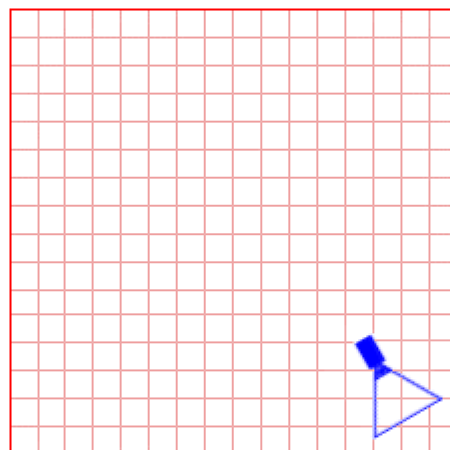

Abb. 9: Sichtbarkeitstest an der Wurzel

Sollte schon die Wurzel nicht im Sichtkegel liegen, so ist das gesamte Terrain nicht sichtbar und braucht nicht gerendert zu werden. In Abb. 9 jedoch sieht man, dass in unserem Fall der Sichtkegel einen Teil der Landschaft einschließt und somit auch die Bounding Box des Wurzelknotens (engl. Rootnode) mit dem Viewfrustum kollidiert. Also müssen nun die Kinder des Wurzelknotens geprüft werden.

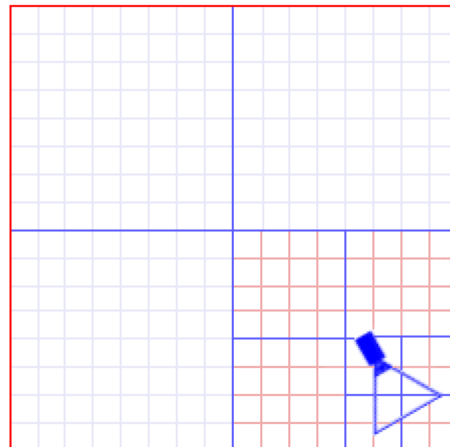

Abb. 10: Rekursive Sichtbarkeitsprüfung

Im ersten Schritt konnten nun schon dreiviertel des gesamten Terrains geculled werden und nur noch einer den Kindknoten des Wurzelknotens muss weiteruntersucht werden. Im nächsten Schritt können schließlich wiederum dreiviertel des verbleibenden Terrains vom Rendern ausgeschlossen werden und im letzten Schritt bleiben nur noch vier Leafs übrig, die wirklich sichtbar sind.

Man sieht also das es sich beim Quadtree um ein sehr effektives Verfahren zum Rendern großer Landschaften handelt. Mit diesem Vorgehen lässt sich beispielsweise auch die Kollisionserkennung mit der Terrainoberfläche stark beschleunigen. Anstatt jedes Dreieck aus dem das Terrain besteht zu überprüfen, kann man mit ein paar Schritten ein einzelnes Leaf auswählen, in dem die Kollision dann auf Dreiecksebene berechnet werden muss.

3.3 Octrees

Der Octree ist nun nichts anderes, als die Erweiterung des Quadtrees um die dritte Dimension. Er dient dazu einen dreidimensionalen Raum zu unterteilen. Hatte der BSP-Tree noch zwei Kinder je Knoten und der Quadtree vier, so besitzt der Octree nun acht Kinder je Knoten, die die Form von Quadern annehmen. Das Vorgehen beim Aufbau und beim Traversieren des Baumes ist jedoch dieselbe wie auch beim Quadtree. Ein Bild soll dies nochmals kurz veranschaulichen:

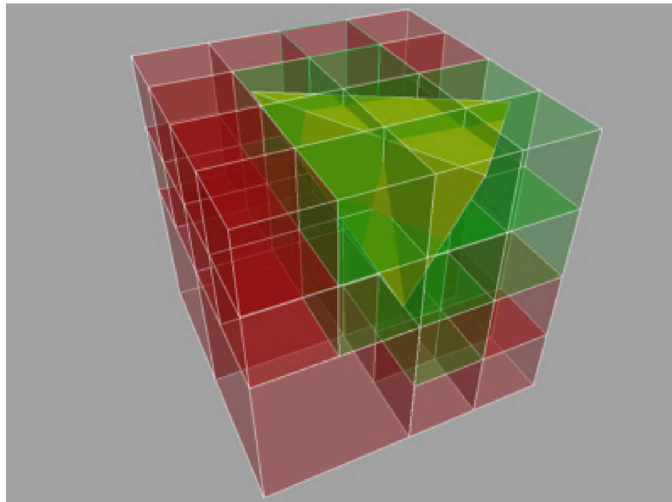

Abb. 11: Octree mit Viewfrustum

Man sieht wie das Viewfrustum die sichtbaren Leafs schneidet (grün) und unsichtbare Nodes direkt aussortiert werden (rot).

4 Weitere Techniken

4.1 ROAM

Bei ROAM handelt es sich um eine ältere Technik, die meist zur Optimierung im Bereich des Terrain-Renderings eingesetzt wurde. Die Abkürzung ROAM steht hierbei für Realtime Optimally Adapting Meshes und verrät bereits worauf dieser Algorithmus basiert. Ziel ist es das Dreiecksnetz (engl. Terrain mesh), welches die Landschaft bildet, möglichst stark zu vereinfachen ohne deutlich sichtbare Detailverluste zu erzeugen. Es wird hierzu ausgehend von einer sehr groben, detailarmen Version des Terrains durch Hinzufügen von weiteren Eckpunkten (engl. Vertices) ein detailreicheres Dreiecksnetz erzeugt. Je weiter ein Teilbereich der Landschaft vom Betrachter bzw. der virtuellen Kamera entfernt ist, desto weniger Details werden hinzufügt. Somit entsteht eine optimale Balance zwischen Detailreichtum und Einsparung von Dreiecken die von der Grafikkarte gerendert werden müssen.

Der Nachteil dieser Vorgehensweise ist jedoch der hohe Aufwand an CPU-Rechenleistung, da in jedem Frame das Terrainmesh neu berechnet werden muss. Ein weiterer Nachteil der daraus folgt, ist das die hohe Leistungsfähigkeit moderner Grafikchips nicht genutzt wird.

4.2 Geometrical Clipmapping

Geometrical Clipmapping ist eine vergleichsweise neue Technik, die dem Texture Mipmapping ähnelt. Eingesetzt wird sie vor allem für die Darstellung großer Terrains, wie z.B. bei Geo-Simulationen. Die Idee hinter dieser Technik ist folgende:

Moderne Grafikkarten sind in der Lage viele Millionen Dreiecke pro Sekunde darzustellen. Sie können den gesamten Bildschirm mit pixelgroßen Dreiecken füllen und trotzdem eine hohe FPS-Rate garantieren. Über ein hochdetailiertes Dreiecksnetz wird eine Reihe von verschachtelten Gitternetzen gelegt die verschiedene Detailgrade aufweisen.

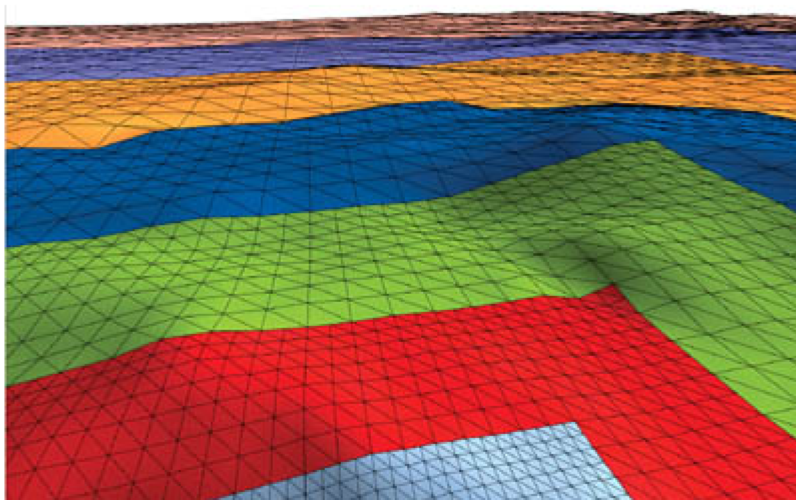

Abb. 12: Geoclipmapping

Diese ineinander eingebetteten Gitternetze werden stets mit der Kameraposition verschoben, sodass eine kontinuierliche Anpassung des Detailgrades um den Betrachter herum erfolgt. Man nennt dies auch CLOD. Das so vereinfachte Terrainmesh kann nun gerendert werden.

4.3 Kd-Tree

Der kd-Baum ist eine Datenstruktur um Punkte im k-dimensionalen Raum zu organisieren. Der Grundraum wird dabei immer wieder unterteilt, bis sich eine bestimmte Anzahl von Punkten in einem Teilraum befindet bzw. eine bestimmte Tiefe des Baumes oder ein anderes Abbruchkriterium erreicht ist.

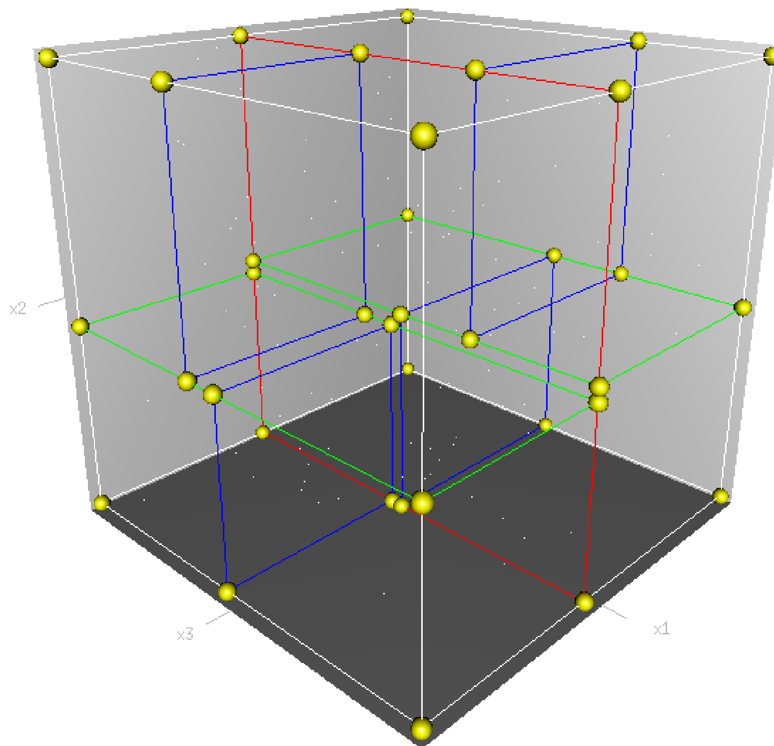

Abb. 13: kd-Tree

Abbildung 13 zeigt einen 3d-Tree. Als erstes wird der Grundraum durch die rote Ebene geteilt. Anschließend werden die Unterräume durch die grünen und nochmals durch die blauen Ebenen geteilt.

5 Verbesserungen

Die hier vorgestellten Algorithmen ermöglichen bereits eine flüssige Darstellung komplexer Szenen. Oft werden sie allerdings mit anderen Techniken kombiniert um noch bessere Frameraten zu erzielen bzw. Ressourcen für andere Berechnungen freizuhalten.

Eine sehr häufig verwendete Technik ist das sog. Backface-Culling. Hierbei wird davon ausgegangen, dass es in einer Szene nur geschlossene, massive Objekte gibt. Ist dies der Fall so müssen von den Dreiecken aus denen ein Mesh besteht nur die Vorderseiten gerendert werden, da die Rückseiten (engl. Backfaces) schließlich verdeckt sind.

Eine andere Möglichkeit (nahezu) unsichtbare Objekte auszusortieren ist das Contribution-Culling. Objekte die sehr weit vom Betrachter entfernt sind und deren Beitrag am Gesamtbild nur wenige oder gar keine Pixel am Bildschirm wären, werden

durch das Contribution-Culling verworfen. Meist wird hier zusätzlich Distanznebel eingesetzt um keine Lücken im Bild zu erzeugen.

Eine andere Technik die darauf beruht weit entfernte Objekt mit einem geringeren Detailgrad darzustellen und erst in der Nähe vom Betrachter in voller Auflösung zu renderen ist das sog. LOD. Dies soll nun am Beispiel des GeoMipMapping verdeutlicht werden.

5.1 GeoMipMapping

Das GeoMipMapping baut auf dem Quadtree-Algorithmus auf und erweitert ihn noch um den Einsatz eines LOD, genauer gesagt DLOD (Discrete Level of Detail).

Zuerst wird mit Hilfe des Quadtrees ein Viewfrustum-Culling durchgeführt um die sichtbaren Blättern des Baumes zu bestimmen.

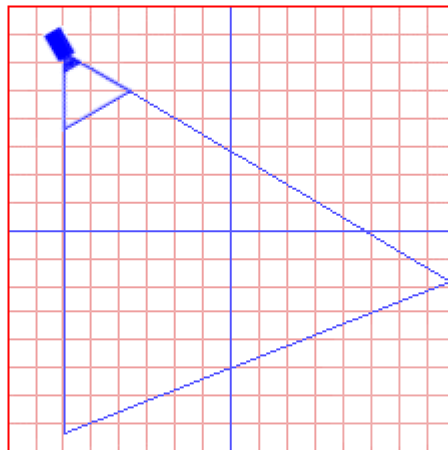

Abb. 14: Viewfrustum-Culling auf dem Quadtree

Nachdem nun die sichtbaren Leafs bekannt sind wird für jedes einzelne die Distanz zur virtuellen Kamera berechnet. Anhand dieses Abstandes wird nun ein Detailgrad ausgewählt, mit dem das jeweilige Terrainstück (also ein Blatt des Baumes) gerendert werden soll. In der folgenden Abbildung sind die dunkelgrünen Bereiche mit den meisten Details versehen, während die hellgrünen weniger Details aufweisen.

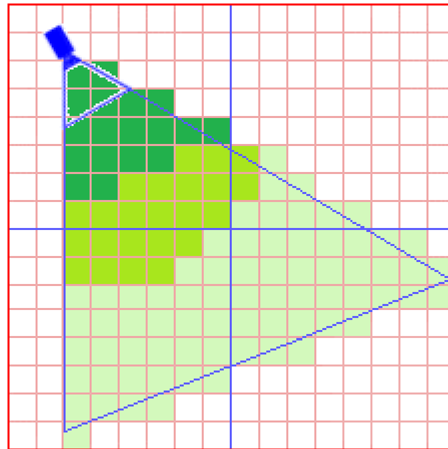

Abb. 15: GeoMipMapping

Die Detailstufen der Leafs können vor der eigentlichen Laufzeit berechnet werden und nehmen damit keine zusätzliche Rechenzeit in Anspruch (dafür extra Speicherplatz).

6 Fazit

Der Einsatz der hier vorgestellten Algorithmen und Techniken kann dazu beitragen die anfangs erwähnten Ressourcen optimal auszunutzen und unnötige Berechnungen einzusparen. Man sollte jedoch darauf achten Optimierungen nur dort einzusetzen, wo sie auch wirklich sinnvoll sind. Beispielsweise ist ein LOD für so einfache Formen wie eine Tür oder eine Wand wirkungslos bzw. würde die Framerate eher bremsen statt beschleunigen.

Ebenso lässt sich ein Indoor-Level (also eine virtuelle Welt die nur aus zusammenhängenden, geschlossen Räumen besteht) schlechter mit einem Quadtree verwalten als mit einem BSP-Tree.

Abschließend ist noch zu sagen, dass Optimierungsalgorithmen in der Regel unsichtbar sein sollten. Im Gegensatz zu Algorithmen für die Lichtberechnung o.ä. Man möchte nicht den Übergang zwischen verschiedenen Detailgraden des LOD sehen, wie es beim GeoMipMapping beispielsweise passieren kann, dass markante Berge oder Täler aus der Landschaft verschwinden wenn man sich weiter von ihnen entfernt.

Trotz der sich stetig steigernden Leistung von Mikrochips und den immer größer werdenden Speichern, sind Optimierungsalgorithmen ein wichtiges Mittel um den Bedarf an Rechen- und Speicherkapazität moderner Software zu stillen.

Software is a gas; it expands to fill its container.

- Nathan's First Law

7 Literaturverzeichnis

Quellen

Zerbst, Stefan (2002): 3D Spieleprogrammierung mit DirectX in C/C++ - Band II, Braunschweig.

Asivatham and Hoppe. Terrain rendering using gpu-based geometry clipmaps. GPU Gems 2. 2005.

Losasso and Hoppe. Geometry clipmaps: terrain rendering using nested rectangular grids. ACM Trans. Graph., 23(3):769-776, 2004.

Jacob E. Goodman, Joseph O'Rourke and Piotr Indyk (Ed.) (2004). "Chapter 39: Nearest neighbors in high-dimensional spaces". *Handbook of Discrete and Computational Geometry* (2nd ed.). CRC Press.