Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Anne-Lena Kowalka

Analyse der Sichtbarkeitsberechnung anhand des View Frustum Cullings

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Anne-Lena Kowalka

Analyse der Sichtbarkeitsberechnung anhand des View Frustum Cullings

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jenke
Zweitgutachter: Prof. Dr. Fohl

Eingereicht am: 07. Juni 2016

Anne-Lena Kowalka

Thema der Arbeit
Analyse der Sichtbarkeitsberechnung anhand des
View Frustum Cullings

Stichworte
Sichtbarkeitsberechnung, View Frustum Culling, Culling Techniken, Kameramodell, Octree, Hierarchische Datenstruktur, Hüllkörper

Kurzzusammenfassung
Diese Ausarbeitung gibt eine Übersicht über Optimierungsansätze in der Computergrafik mittels Sichtbarkeitsberechnung, insbesondere des View Frustum Cullings, sowie hierarchischen Datenstrukturen. Um die genannten Optimierungen beziehungsweise Performanceverbesserungen der Framerate nachzuweisen, wurde das View Frustum Culling in das Computergrafik-Framework der Computergrafik-Gruppe der HAW implementiert. Die Implementierung wendet zusätzlich die hierarchische Datenstruktur des Octrees an, um den Performancegewinn zu maximieren. Anschließend wurden Performancetests mit und ohne Anwendung des Cullings durchgeführt, die in dieser Arbeit verglichen und diskutiert werden.

Anne-Lena Kowalka

Title of the paper
Analysis of hidden surface determination by reference of view frustum culling

Keywords
hidden surface determination, view frustum culling, culling techniques, camera model, octree, hierarchical data structure, bounding volume

Abstract
This document gives an overview on optimizations for computer graphics by use of hidden surface determination, especially view frustum culling, and also on hierarchical data structures. To substantiate the mentioned optimizations, view frustum culling was implemented into the computer graphics group´s framework of the HAW. This implementation makes use of the hierarchical data structure of the octree to maximize the performance. The following perfomance tests with and without the use of view frustum culling are compared and discussed.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Die Darstellung großer Szenen und komplexer Strukturen mit Hilfe der Computergrafik kann sehr viel Rechenzeit in Anspruch nehmen. Deshalb ist es wichtig, unnötiges Rendering zu vermeiden, um eine bessere Performance zu erreichen. Eine Möglichkeit hierfür ist die Analyse dessen, welche Teile einer Szene tatsächlich mit Hilfe der Rendering Pipeline verarbeitet werden müssen und welche Teile der Szene nicht von der Kamera erfasst und somit nicht gerendert werden müssen. Ziel dieser Analyse ist es, das Rendering auf das Nötigste zu minimieren, um überflüssige Rechenzeit zu vermeiden und eine höhere Framerate zu erreichen.
Eines der dazu verwendeten Mittel der Computergrafik ist die Sichtbarkeitsberechnung. Es gibt verschiedene Varianten der Sichtbarkeitsberechnung, unter Anderem das „Culling“, welches eine Klasse von Algorithmen aus der Sichtbarkeitsberechnung zusammenfasst. Culling hilft dabei, beispielsweise Verdeckung, Ausrichtung beziehungsweise Abwendung vom Betracher oder auch Position von Objekten außerhalb des sichtbaren Bereichs zu erkennen und bringt somit verschiedene Performanceoptimierungen mit sich.

1.2 Zielsetzung

Diese Arbeit stellt verschiedene Culling-Techniken vor, wobei besonders auf das View Frustum Culling eingegangen wird. Außerdem wird der Einsatz hierarchischer Strukturen mit dem Ziel der Performanceverbesserung präsentiert und es werden Verwendungsansätze hierarchischer Strukturen in Verbindung mit Culling-Techniken aufgezeigt.
Ziel dieser Arbeit ist es, die Sichtbarkeitsberechnung der Computergrafik zu evaluieren. Des Weiteren soll anhand einer Implementierung des View Frustum Cullings Bezug zur Praxis hergestellt sowie Tests bezüglich der genannten Performanceverbesserungen durchgeführt werden, sodass die Wirkung des Cullings hinsichtlich der Performance aufgezeigt werden kann.
Die Implementierung soll weiterhin in dem Computergrafik-Framework der Computergrafik-Gruppe der HAW eingesetzt werden können.

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich in folgende Kapitel:
In dem Kapitel „Analyse“ werden verschiedene Culling-Techniken sowie GPU-basierte Sichtbarkeitsberechnung erläutert. Außerdem werden Ansätze für Optimierungsmöglichkeiten mittels hierarchischer Strukturen erklärt und mathematische Grundlagen gegeben.
Das Kapitel „Kameramodell als Grundlage für das View Frustum Culling“ untersucht das Kameramodell sowie die Herleitung des Sichtbarkeitsbereichs aus diesem und nennt Schwächen sowie diverse Verbesserungsansätze des Kameramodells. Desweiteren werden Optimierungsansätze für die Positionserkennung von Objekten hinsichtlich des Sichtbarkeitsbereichs vorgestellt.
Im Kapitel „Implementierung“ werden die bei der Implementierung verwendeten hierarchischen Datenstrukturen und Algorithmen für den Aufbau sowie die Verwendung des Sichtbarkeitsbereichs hinsichtlich des View Frustum Cullings erläutert.
Es folgt eine Evaluation, in der die Testergebnisse präsentiert und diskutiert werden.
In der „Schlussbetrachtung“ wird ein Fazit gezogen sowie mögliche Ansätze zur Weiterentwicklung der Implementierung gegeben.

2 Analyse

2.1 Sichtbarkeitsberechnung

Mithilfe der Computergrafik können große, detailreiche Szenen erzeugt werden, beispielsweise für Computerspiele oder Animationsfilme. Die Berechnungen dafür können viel Zeit in Anspruch nehmen, was laut Pramberger während den Anfängen der Computergrafik zu einem der ersten Probleme geführt hat, dem Sichtbarkeitsproblem (vgl. Pramberger (2012)). Auch Cohen-Or u. a. bezeichnet das Sichtbarkeitsproblem als fundamentales Problem der Computergrafik, da es nicht nur für korrekte Bilderzeugung relevant ist, sondern auch beispielsweise für Schattenbildung sowie Beleuchtungsrechnung (vgl. Cohen-Or u. a. (2003).
Das Sichtbarkeitsproblem beschreibt die Problematik bei der Abbildung einer dreidimensionalen Szene in ein zweidimensionales Bild und die dabei entstehende Frage, welche Oberflächen in der Abbildung tatsächlich sichtbar sind. Idealerweise fallen Berechnungen für Oberflächen, die im Bild nicht zu sehen sein werden, weg, sodass eine höhere Performance erzielt werden kann.
Sichtbarkeitsberechnung kann in jeder Ebene der Rendering Pipeline und auch in der Hardware stattfinden. Der Entwickler hat dabei die größte Kontrolle, wenn er den Algorithmus selbst in der Anwendungsebene beziehungsweise auf der CPU umsetzt (vgl. Akenine-Möller u. a. (2008, p. 660-661)).

2.2 Culling-Techniken

2.2.1 View Frustum Culling

Beim View Frustum Culling werden Objekte, die nicht innerhalb des Sichtbarkeitsbereichs liegen, auch nicht durch die Rendering Pipeline geschickt. In Abbildung 2.2 werden diese Objekte durch das hellrote und grüne Dreieck sowie das rote Rechteck und den orangenen Kreis repräsentiert.
Um zu bestimmen, welche Objekte nicht gerendert werden müssen, werden ihre Hüllkörper gegen die sechs Ebenen, die das View Frustum eingrenzen, auf Schnittpunkte getestet.

Wenn ein Objekt komplett außerhalb des sichtbaren Bereichs liegt, wird dies nicht gerendert, während Objekte, die teilweise oder komplett im View Frustum positioniert sind, durch die Rendering Pipeline geschickt werden (vgl. Akenine-Möller u. a. (2008, p. 665-665)). Hier gibt es verschiedene Optimierungsansätze, die ich im Abschnitt 3.2.2 dieser Arbeit erläutern werde.

2.2.2 Backface Culling

Backface Culling basiert auf der Idee, dass diejenigen Oberflächen, die aus Kameraperspektive auf der Rückseite von Objekten liegen, nicht die Rendering-Pipeline durchlaufen (siehe Rückseiten des blauen und gelben Polygons in Abbildung 2.2). Dies verschiebt Berechnungen von der Rasterisierungs-Ebene in die Geometrie-Ebene, sodass zwar mehr Vorausberechnungen nötig, aber in der Abbildung der Szene mehr Frames pro Sekunde möglich sind. Voraussetzung für diese Technik ist, dass die Kamera außerhalb der Szene liegt und nicht durch die Objekte hindurchsehen kann.
Um festzustellen, in welche Richtung ein Polygon orientiert ist, wird seine Normale n berechnet:

$$n = (\vec{v}_1 - \vec{v}_2) \times (\vec{v}_0 - \vec{v}_2) \tag{2.1}$$

$v_{0,1,2}$ repräsentiert am Polygon anliegende Vertices. Um nun die Orientierung eines Polygons zu bestimmen, wird mit der Normalen und einem Vektor, der die Blickrichtung angibt, das Skalarprodukt gebildet. Dabei ist das Vorzeichen des Ergebnisses entscheidend: Ein negatives Vorzeichen deutet darauf hin, dass das Polygon in Richtung der Kamera orientiert und somit sichtbar ist, während ein positives indiziert, dass dieses Polygon von der Kamera weggehend ausgerichtet ist und folglich nicht die Rendering Pipeline durchlaufen muss. (vgl. Akenine-Möller u. a. (2008, p. 662)).
Die Technik des Backface Cullings kann auch auf gesamte Objektgruppen angewendet werden und wird dann „clustered backface culling“ (vgl. Akenine-Möller u. a. (2008, p. 662)) genannt. Das Basiskonzept dafür bildet ein Kegel, der aus den Normalen nebeneinander liegender Oberflächen besteht. Mithilfe dieses Kegels lässt sich schnell feststellen, ob diese Flächen zur Kamera gerichtet oder von ihr abgewandt sind. Zwar ist dafür eine teure Vorausberechnung für den Aufbau des Kegels nötig, doch Performancegewinn entsteht dadurch, dass diese nur einmal passieren muss und die anschließenden Berechnungen pro Frame schneller ablaufen können (vgl. Shirmun und Abi-Ezzi (1993)).
Für das Backface Culling gibt es weiterhin Optimierungsansätze. Mit dem hierarchischen Backface Culling nach Kumar, Manocha, Garret und Lin beispielsweise spiegelt sich der Per-

formancegewinn durch Optimierung in einer 30 bis 70 Prozent höheren Framerate wieder (vlg. Kumar u. a. (1996)).

2.2.3 Occlusion Culling

Beim Occlusion Culling werden Objekte, die hinter anderen liegen und von diesen verdeckt sind, nicht durch die Rendering Pipeline geschickt. In Abbildung 2.2 werden der orangenfarbene Kreis und das grüne sowie pinkfarbene Polygon von dem gelben und blauen Polygon verdeckt, sodass es nicht notwendig ist, diese zu rendern.
Das Ziel dieses Ansatzes ist, dass die resultierenden Bilder eine geringere Tiefe besitzen, also dass Pixel seltener bis nicht neu überschrieben werden. Ob ein Pixel überschrieben wird, hängt dabei von der Reihenfolge ab, in der die Objekte gezeichnet werden. Objekte, die näher am Betrachter sind, sollten zuerst dahingehend geprüft werden, ob sie andere überdecken, da bei nahen Objekten die Wahrscheinlichkeit größer ist, dass sie hinter ihnen liegende abdecken. Selbst sehr kleine Objekte können große, aber weiter hinten liegende komplett überdecken, wenn sie nah genug am Betrachter positioniert sind.
Für das Occlusion Culling gibt es zwei Ansätze: punkt- und zellbasiert (siehe Abbildung 2.1).

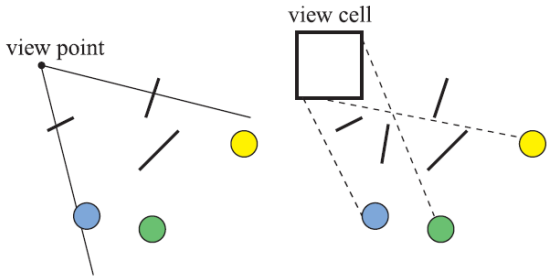

Abbildung 2.1: Punkt- und zellbasiertes Occlusion-Culling. (Aus Akenine-Möller u. a. (2008, p. 672))

Beim punktbasierten Occlusion Culling wird davon ausgegangen, dass der Betrachter an genau einem Punkt lokalisiert ist. Dies ist die übliche Annahme.
Beim zellbasierten Ansatz hingegen ist die Betrachterposition durch eine Zelle definiert. Folglich ist ein Objekt erst dann nicht sichtbar, wenn es von jedem Punkt der Zelle aus nicht sichtbar ist. Dieses Vorgehen ist von Vorteil, wenn sich der Betrachter ausschließlich innerhalb dieser Zelle bewegt, da die Vorausberechnungen dann durchgehend verwendet werden können

und keine neuen Berechnungen nötig sind.

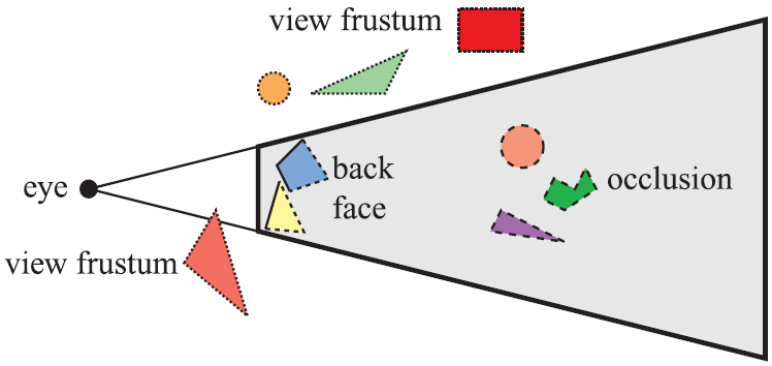

Abbildung 2.2: Verschiedene Culling-Techniken. (Aus Akenine-Möller u. a. (2008, p. 661))

2.2.4 Portal Culling

Portal Culling wird bei räumlichen Modellaufbauten eingesetzt. Es werden Räume, die von der Betracherposition aus nur teilweise, beispielsweise durch Türen oder Fenster, wahrgenommen werden, auf die sichtbaren Flächen reduziert. Diese Culling Technik ist sehr bedeutsam, da sie das View Frustum Culling und das Occlusion Culling zusammenfasst. Auf Räume, die durch Öffnungen sichtbar sind, wird das View Frustum Culling angewendet, und auf Objekte, die innerhalb des View Frustums liegen und andere überdecken, wird das Occlusion Culling angewendet.
Portal Culling erfordert Vorarbeit. Die räumliche Struktur muss aufgebaut werden, und zu jedem Raum müssen Öffnungen und Wände gespeichert werden. Außerdem müssen für jeden Raum Nachbarräume assoziiert werden. Dies geschieht oftmals per Hand, da dieser Prozess sehr komplex sein kann (vgl. Akenine-Möller u. a. (2008, p. 667)).
Airey, einer der ersten, die das Portal Culling implementiert haben, stellte den Nutzen dieser Technik in Frage. Zwar wurde sie bei kommerziellen Spielen leicht abgewandelt erfolgreich eingesetzt und erzielte eine erhöhte Framerate, doch der große Aufwand bei der Modellierung ist von Nachteil. Selbst kleine Änderungen im Aufbau der Räumlichkeit bringen großen Arbeitsaufwand mit sich, da die Modellierung angepasst werden muss (vgl. Airey (1990)).
Luebke und Georges haben einen Ansatz entwickelt, der diesen Aufwand verringern soll, indem der Aufbau des Modells automatisiert wird. Die Anwendung dieses Ansatzes hat im Test eine ein- bis zehnmal so hohe Framerate ergeben wie eine Anwendung ohne Culling und 20 bis 50 Prozent der Polygone entfernt. Sie weisen jedoch darauf hin, dass die Effektivität stark

von der Perspektive und dem vorgegebenen Raummodell abhängt (vgl. Luebke und Georges (1995)).

2.2.5 Detail Culling

Beim Detail Culling wird Performance durch Vernachlässigung der Detailgenauigkeit erreicht. Dies bedeutet, dass wenn sich der Betrachter bewegt, im gerenderten Bild auf kleine Details verzichtet wird. Dieser Ansatz wird auch „screen-size culling“ (Akenine-Möller u. a. (2008, p. 670)) genannt. Der Benutzer kann eine Grenze an Pixeln festlegen. Wenn ein Objekt im resultierenden Bild diese Anzahl an Pixeln erreicht, dann wird es gezeichnet. Ansonsten wird das Objekt verworfen.
Bei fixem Betrachtungspunkt wird das Detail Culling hingegen ausgeschaltet, sodass auch Details gezeichnet werden (vgl. ebd.).

2.3 Bedeutung des View Frustum Cullings

Bei Anwendungen, in denen die Kamera dynamisch durch eine Szene geschwenkt wird, ist das View Frustum Culling von großer Bedeutung. Beispielsweise bei Computerspielen besteht die Spielumgebung aus einer großen dreidimensionalen Szene, von welcher der Spieler meistens nur einen kleinen Ausschnitt sieht. Um zu verhindern, dass Ausschnitte, die außerhalb des sichtbaren Bereichs liegen, durch die Rendering Pipeline geschickt werden, wird das View Frustum Culling angewendet. Diese Technik ist in der Computergrafik essentiell, um komplexe Szenen schnell und performant rendern zu können (vgl. Akenine-Möller u. a. (2008, p. 771)).
Für das View Frustum Culling gibt es verschiedene Optimierungsansätze, beispielsweise für die Berechnungen, ob ein Objekt innerhalb des Frustums liegt. Des Weiteren gibt es Optimierungen, die mithilfe hierarchischer Strukturen arbeiten. Auf mögliche Optimierungen werde ich in einem späteren Teil der Arbeit (Abschnitt 3.2.2) eingehen.

2.4 GPU-basierte Sichtbarkeitsberechnung

Sichtbarkeitsberechnung wird auch von der Grafik-Hardware unterstützt. Im Folgenden werden Grundlagen sowie auf diese aufbauende Erweiterungen präsentiert.

2.4.1 Z-Puffer

Sichtbarkeitsberechnung findet auch in der Grafik-Hardware eines Computers statt, wie zum Beispiel im Z-Puffer. Dieser Puffer ist aufgebaut wie der Farb-Puffer und enthält für jeden Pixel des Bildes einen Z-Wert (vergleiche dazu Abbildung 2.3). Der Z-Wert gibt an, welches Objekt am nächsten am Betrachter positioniert ist (mit Hilfe der Position auf der Z-Achse) Wird ein Objekt gerendert, wird zunächst sein Z-Wert für das jeweilige Pixel berechnet. Wenn dieser Z-Wert kleiner als der aktuelle Wert im Z-Puffer ist, dann ist dieses Objekt näher am Betrachter positioniert. Folglich werden die Werte im Farb- und Z-Puffer mit den neuen Werten ersetzt. Ist der neue Z-Wert größer als der Wert im Z-Puffer, bleiben sowohl Farb- als auch Z-Puffer unberührt.
Der Vorteil dieses Algorithmus ist, dass er einfach ist. Weiterhin hat er ein $O(n)$, wobei n die Anzahl der Objekte ist. Einen weiteren Einfluss auf die Laufzeit hat allerdings noch die Anzahl der Pixel aller gerasterten Fragmente, sodass sich die Laufzeit schwer genau sagen lässt. Außerdem kann dieser Algorithmus auf alle Objekte angewendet werden, für die ein Z-Wert berechnet werden kann, wobei er unabhängig von der Reihenfolge der Objekte bleibt (vgl. Akenine-Möller u. a. (2008, p. 23-24)).
Catmull betont ebenfalls die Simplizität des Z-Puffers (vgl. Catmull (1974)). Sichtbarkeits- und Schnittberechnungen lassen sich mit Hilfe dieses Algorithmus trivial durchführen. Dabei kann der Aufbau der Szene beliebig komplex sein.
Laut Catmull gibt es aber auch diverse Nachteile, die der Z-Puffer mit sich bringt. Einer dieser Nachteile ist der hohe Speicherverbrauch, der zu Zeiten der Veröffentlichung seines Papers (1974) noch ein Problem dargestellt hat. Heutzutage dürfte dieser Aspekt aufgrund verbesserter Technik jedoch vernachlässigbar sein (vgl. ebd.).
Eine weitere und wichtigere Problemstellung ist das Vermindern des Treppen-Effekts („Aliasing“). Beim Treppen-Effekt erscheinen Kanten „treppenartig“, was an der endlichen Auflösung des Bildes liegt und sich somit nicht komplett verhindern lässt. Um diesen Effekt zu verringern, werden die Farbwerte sowohl des bedeckenden als auch des bedeckten Objekts kombiniert. Um zu verhindern, dass beispielsweise durch eine zufällige Render-Reihenfolge falsche Farbwerte vermischt werden, kann es von Nöten sein, die zu rendernden Objekte ihrer Entfernung zum Betrachter nach zu sortieren (vgl. ebd.).

Ein weiterer Nachteil ist, dass dieser Algorithmus nur beschränkt auf transparente Objekte angewendet werden kann. Diese müssen nach den undurchsichtigen Objekten gerendert werden und von hinten nach vorne (vgl. Akenine-Möller u. a. (2008, p. 23-24)).
Auch Cohen-Or u. a. erkennen die sehr gute Funktionalität des Z-Puffers bezüglich des Sichtbarkeitsproblems an. Als Schwachstelle benennen sie jedoch, dass jedes Pixel entsprechend seines Z-Wertes oft angefasst wird und somit unnötige Berechnungen durchgeführt und Speicher verschwendet werden (vgl. Cohen-Or u. a. (2003)). Um das Rendering mit Hilfe eines Z-Puffers zu beschleunigen, wird vorgeschlagen, zuvor einen Filter wie zum Beispiel das Backface Culling anzuwenden, oder den hierarchischen Z-Puffer zu verwenden, der im folgenden Kapitel (Abschnitt 2.4.2) genauer vorgestellt wird (vlg. ebd.).

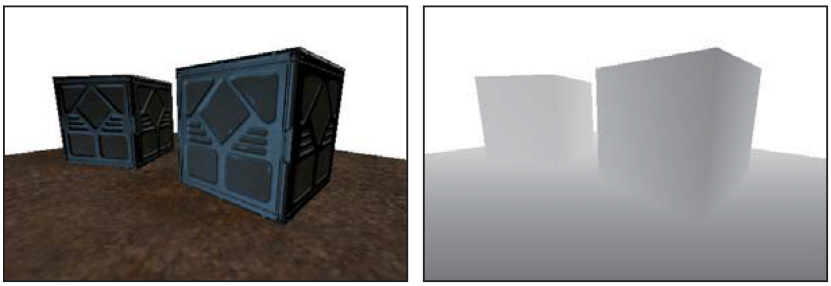

Abbildung 2.3: Eine gerenderte Szene und ihr zugehöriger visualisierter Z-Puffer. (Aus Hughes u. a. (1990, p. 1036))

2.4.2 Hierarchisches Z-Puffering

Die Verwendung des Z-Puffer-Algorithmus wie in Abschnitt 2.4.1 beschrieben kann problematisch werden, wenn es große, komplexe Szenen zu rendern gibt. Er führt pro Frame für jedes Polygon erneut Berechnungen durch, auch dann, wenn diese kompletten Polygone nicht sichtbar sind, weil sie beispielsweise von einem nahe liegenden Objekt abgedeckt sind.
Um diesen Worst Case zu vermeiden, haben Greene, Kass und Miller den Ansatz des hierarchischen Z-Puffers entwickelt. Er arbeitet mit Hilfe dreier Erweiterungen: hierarchische Unterteilung der Szene mittels eines Octrees, eine Z-Pyramide für die räumliche Struktur und eine Liste der Objekte, die im vorigen Frame sichtbar waren (vgl. Greene u. a. (1993)).
Auf die hierarchische Struktur des Octrees werde ich in einem späteren Teil dieser Arbeit (Abschnitt 2.5.4) noch genauer eingehen.
Der Octree wird um die komplette Szene gebildet. Wenn ein Octree-Würfel nicht im Frustum liegt, kann die mit ihm assoziierte Geometrie komplett ignoriert werden. Wird das Frustum

hingegen von einem Octree-Würfel geschnitten oder liegt der Würfel komplett innerhalb des Frustums, dann werden rekursiv seine Kind-Würfel geprüft. Dies hat zur Folge, dass Polygone in Würfeln, die außerhalb des Frustums liegen, auch nicht gerendert werden. Polygone, die in Würfeln liegen, die nur teilweise zu sehen sind, sind im schlimmsten Falle maximal eine Diagonalenlänge des beinhaltenden Würfels vom Frustum entfernt.
Der Octree kann den Nachteil mit sich bringen, dass Polygone, die mehrere Octree-Würfel schneiden, mehrfach gerendert werden, weil sie allen Würfeln zugeordnet sind. Dies kann jedoch vermieden werden, wenn das jeweilige Polygon beim ersten Mal als „schon gerendert“ markiert wird.
Die Z-Pyramide baut sich folgendermaßen auf: Auf unterster Ebene beinhaltet sie den normalen Z-Puffer. Die Werte der Ebenen dadrüber entstehen daraus, dass Flächen des Bildschirms in 2x2-Fenster (Quadranten) zusammengefasst werden und der Wert eingetragen wird, der am weitesten weg ist. Sollten sich am Z-Puffer Änderungen ergeben, werden diese nur dann eingetragen, sofern diese ein weiter weg liegendes Objekt indizieren. Daraus folgt, dass in der höchsten Ebene der Wert des Objekts steht, das am weitesten weg ist (siehe Kasten ganz rechts in Abbildung 2.4).

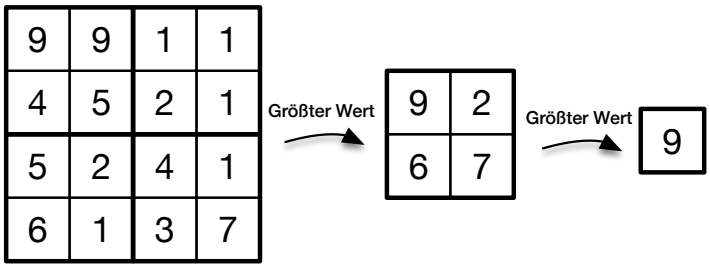

Abbildung 2.4: Aufbau eines hierarchischen Z-Puffers. (Aus Akenine-Möller u. a. (2008, p. 678))

Möchte man nun für ein Polygon herausfinden, ob es verdeckt ist, sucht man sich den zugehörigen Quadranten, der das Polygon beinhaltet. Ist der nächste Wert des Polygons weiter entfernt als der am weitesten entfernte des Quadranten, dann ist das Polygon verdeckt.
Um zu beweisen, ob ein Polygon verdeckt wird oder nicht, wird sein nächster Z-Wert mit dem zugehörigen Z-Wert der Z-Pyramide, beginnend bei der höchsten Ebene, verglichen. Sollte der Fall auftreten, dass Werte der Pyramide näher sind als das Polygon, kann abgebrochen

werden, da das Polygon dann überdeckt wird. Andernfalls wird der Z-Wert des Polygons mit dem jeweiligen Wert der Pyramide in der nächsttieferen Ebene verglichen. Wenn mit diesem Vorgehen die tiefste Ebene der Pyramide erreicht wird, ist das Polygon nicht verdeckt und muss gerendert werden.
Die Liste der bereits gerenderten Objekte basiert auf der Tatsache, dass die Darstellung von Szenen, sofern kein Szenewechsel oder Kameraschnitt passiert, sich von Frame zu Frame nicht stark verändert.
Angewendet bedeutet dies, dass Objekte, die aufgelistet sind, gerendert und markiert werden. Dann beginnt der eigentliche Algorithmus. Aus dem beim Rendering entstandenen Z-Puffer wird die Z-Pyramide gebildet. Wenn ausreichend Ähnlichkeit zum vorigen Frame vorliegt, sind im Z-Pyramiden-Test weniger Rekursionen nötig, da dies schon für den vorigen Frame passiert ist.
In der Praxis angewandt zeigte sich, dass bei einfachen Szenen das Rendering mit dem hierarchischen Z-Puffer länger dauert als das Rendering ohne den Puffer. Dies liegt an dem Rechenaufwand, der beim Aufbau der Z-Pyramide entsteht.
Bei komplexen Szenen hingegen ließ sich einen große Verbesserung feststellen: Während das Rendering einer Szene mit 538 Millionen Polygonen, von denen 59,7 Millionen Polygone im View Frustum liegen, mit dem Z-Puffer eine Stunde und 15 Minuten dauerte, brauchte es mit dem hierarchischen Z-Puffer 6,45 Sekunden (vgl. Greene u. a. (1993)).

2.4.3 Erweiterte Kollisionserkennung mit Hilfe der Grafik-Hardware

Die bereits genannten Algorithmen für GPU-basierte Sichtbarkeitsberechnung bringen einige Nachteile mit sich: das Auslesen des Z-Puffers ist ineffizient. Von Govindaraju (vgl. Govindaraju u. a. (2003)) durchgeführte Tests ergaben, dass das Auslesen des Z-Puffers bis zu 50 Millisekunden dauern kann, was die verbesserte Effizienz anderer Optimierungen relativiert. Des Weiteren sind sie auf geschlossene Objekte beschränkt, die bereits vorhandene Funktionen der grafischen Hardware verwenden, um Kollisionen zu entdecken. Außerdem sind diese Algorithmen nicht auf die Verwendung von großen, sich bewegenden Modellen ausgelegt.
Govindaraju et al. haben einen neuen Algorithmus entwickelt, der diese Nachteile ausgleichen soll. Er basiert darauf, dass „potentially colliding set[s]“ (PCS, vgl. ebd.) berechnet werden. Die PCS enthalten Objekte, die sich schneiden oder sehr nahe beieinander liegen. Der Algorithmus zum Aufbau der PCS sowie zur Kollisionserkennung gliedert sich in drei Phasen, von denen die ersten zwei im Image-Space, also auf der GPU stattfinden, und die dritte auf der CPU.
Um die PCS aufzubauen, wird wie folgt vorgegangen: zunächst wird der Z-Puffer geleert. Dann werden die Objekte zuerst von vorne nach hinten und dann, nach einer weiteren Leerung des

Z-Puffers, von hinten nach vorne gerendert. Dabei wird festgehalten, ob ein Objekt komplett sichtbar ist. Objekte, die bei beiden Renderdurchgängen komplett sichtbar sind, gehören nicht zum PCS.
Geläufige Hardware kann die Anfrage beantworten, ob ein Objekt sichtbar ist oder nicht, wobei die Performance abhängig von der verwendeten Hardware variiert. Da die Abfrage benötigt wird, ob alle Fragmente eines Objekts einen kleineren Z-Wert haben als im zugehörigen Z-Puffer, wird diese Gegebenheit dahingehend verändert, dass der Tiefentest jedes Fragment des Objekts mit dem Z-Puffer vergleicht. Wenn jedes Fragment einen kleineren Z-Wert hat als den aktuellen zugehörigen Wert im Puffer, dann ist das Objekt komplett nicht sichtbar.
Die zweite Stufe des Algorithmus teilt Objekte in nebeneinander liegende Dreiecke (Unterobjekte) auf und bildet für diese Unterteilung ein neues PCS. Schließlich wird für jedes einzelne Dreieck, welches das PCS der Unterobjekte beinhaltet, eine eigene Tiefenabfrage durchgeführt, sodass nur noch die Dreiecke übrig bleiben, die sich schneiden. Laut Govindaraju ist es performanter, nochmals ein PCS über die Unterobjekte zu bilden, da somit nicht für jedes Dreieck im PCS über die Objekte eine Tiefenabfrage durchgeführt werden muss (vgl. ebd.).
Die genauen Schnittpunkte der Dreiecke, die im letzten PCS enthalten sind, werden auf der CPU berechnet.
Die Performance dieses Ansatzes hängt von diversen Faktoren ab, beispielsweise der Anzahl der Objekte, der Tesselierung dieser und der Genauigkeit, welche wiederum von der Auflösung des Bildes abhängt. Der Ansatz bringt dafür, wie bereits erwähnt, den Vorteil mit sich, dass er auf große und sich bewegende Modelle anwendbar ist. Außerdem kann er auf alle Dreiecksnetz-Modelle verwendet werden (vgl. ebd.).

2.5 Optimierungsmöglichkeiten durch hierarchische Strukturen

Der Einsatz hierarchischer Strukturen in der Computergrafik kann große Vorteile mit sich bringen. Beim Rendern komplexer Szenen können Performanceprobleme auftreten, die sich mit Hilfe von beispielsweise Quad- oder Octrees reduzieren oder gar beheben lassen können. Eine hierarchische Unterteilung beginnt allerdings schon mit den Hüllkörpern einzelner Objekte. So ist eine Kollision zwischen Kugeln oder Würfeln, die als Hüllkörper von Objekten verwendet werden, deutlich einfacher festzustellen als eine Kollision zwischen zwei komplexen Dreiecksnetzen mit verschiedenen Ausdehnungen in alle Richtungen. Allerdings ist diese erste Stufe der hierarchischen Unterteilung bei sehr großen und vielschichtigen Szenen oft

nicht ausreichend. An dieser Stelle kommen hierarchische Strukturen wie Bounding Volume Hierarchie (siehe dazu Abschnitt 2.5.2) sowie Quad- und Octrees (Abschnitt 2.5.4) zum Einsatz.

2.5.1 Aufbau hierarchischer Strukturen

Hierarchische Strukturen werden häufig durch Bäume repräsentiert. Dabei gibt es drei Möglichkeiten, einen solchen Baum aufzubauen: *bottom-up*, *incremental tree-insertion* und *top-down*.
Die Bottom-up-Methode beginnt bei den Blattknoten, welche die Objekte repräsentieren. Es werden nah beieinander liegende Hüllkörper zusammengefasst und einem übergeordneten Knoten zugeordnet. Folglich sind Hüllkörper, die in der Szene nahe beieinander positioniert sind, in der Baumstruktur nebeneinander.
Der Top-down-Ansatz startet damit, dass ein Hüllkörper für alle Objekte gebildet wird, welcher dem Wurzelknoten zugeordnet wird. Anschließend wird mit Hilfe des Teile-und-herrsche-Verfahrens der Wurzelknoten aufgeteilt. Nun wird erneut für die Kindknoten ein Hüllkörper gebildet und diesem alle Objekte zugewiesen, dessen Hüllkörper eingeschlossen werden. Dieses Verfahren wird rekursiv fortgesetzt, bis die Blattknoten, also die Objekte, erreicht sind.
Bei der tree-insertion-Methode gibt es anfangs einen leeren Baum. Es werden die Blattknoten einzeln zum Baum hinzugefügt. Dabei wird das Volumen des gesamten Baumes möglichst klein gehalten (vgl. Akenine-Möller u. a. (2008, p. 804-805).

2.5.2 Bounding Volume Hierarchie

Bei der Bounding Volume Hierarchie gibt es einen Wurzelknoten und je Objekt einen Blattknoten. Dazwischen gibt es interne Knoten, die Zeiger auf ihre Kindknoten enthalten. Zu den internen Knoten gehört auch der Wurzelknoten. Jeder Knoten des Baumes hat einen Hüllkörper, der die gesamten Kindknoten enthält. Pro Ebene, die ein Knoten weiter vom Wurzelknoten entfernt ist, erhöht sich die Höhe des Baumes. So hat ein Baum, der nur einen Wurzel- und einen Blattknoten enthält, die Höhe eins. Bei einem balancierten Baum haben alle Blattknoten die Höhe h oder $h-1$, während bei einem vollem Baum alle Blattknoten dieselbe Höhe haben. Die Höhe lässt sich anhand der Anzahl der Knoten (sowohl interne als auch Blattknoten) berechnen: $\log_k(n)$, wobei k die Anzahl der Kindknoten jedes internen Knoten und n die gesamte Anzahl der Knoten ist (vgl. Akenine-Möller u. a. (2008, p. 649)).

Abbildung 2.5 zeigt eine Bounding Volume Hierarchie mit einem vollen Baum der Höhe zwei.

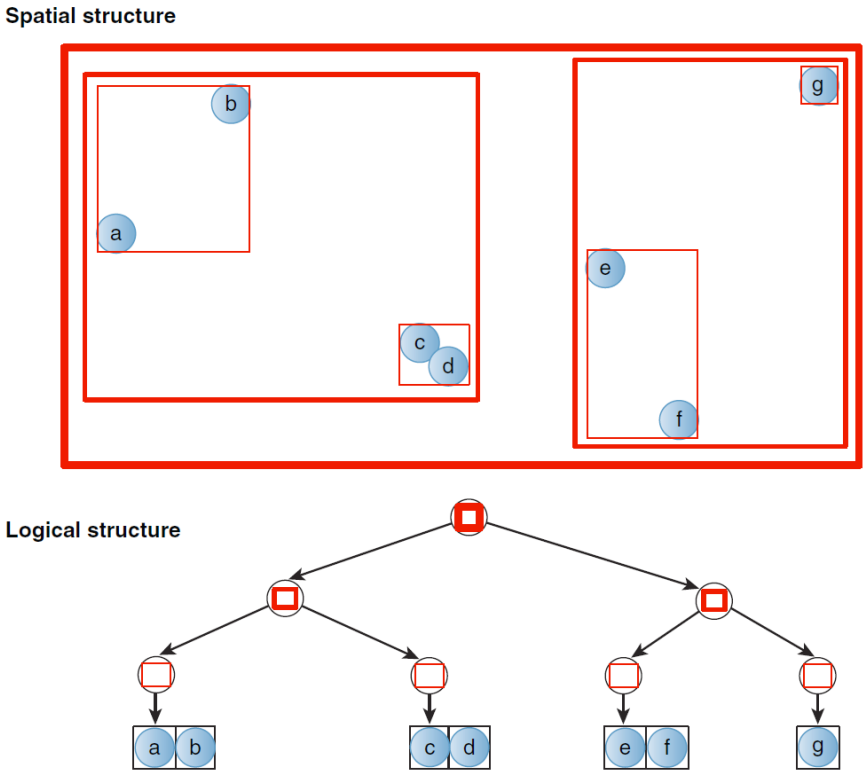

Abbildung 2.5: Darstellung einer zweidimensionalen Bounding Volume Hierarchie („Spatial structure“) und ihrem zugehörigen Baum („Logical Structure“). (Aus Hughes u. a. (1990, p. 1092))

In der Praxis angewandt beschleunigt die hierarchische Unterteilung viele Berechnungen. Wenn beispielsweise ein Strahl auf die Szene fällt und der Schnittpunkt berechnet werden soll, wird der Strahl zuerst gegen den Hüllkörper des Wurzelknotens getestet. Dieser enthält den kompletten Baum und somit alle Objekte, und wenn der Strahl ihn nicht trifft, kann die Prüfung beendet werden. Entsteht jedoch ein Schnittpunkt zwischen Strahl und dem Hüllkörper, werden rekursiv Schnittpunkttests mit den Hüllkörpern der Kindknoten durchgeführt. Sofern kein Schnittpunkt vorhanden ist, kann die Prüfung dieses Teilbaums beendet werden. Wird ein Schnittpunkt gefunden, werden rekursiv die Hüllkörper aller Kindknoten geprüft.
Bounding Volume Hierarchien können auch auf dynamische Szenen angewandt werden. Dabei gibt es zwei Möglichkeiten, den Baum umzustrukturieren, wenn sich ein Objekt bewegt und

dadurch den Hüllkörper des Elternknotens verlässt: entweder der Hüllkörper wird ausgeweitet oder der Blattknoten wird entfernt, der Hüllkörper des Elternknotens neu berechnet und das Objekt vom Wurzelknoten ausgehend neu eingefügt (vgl. Akenine-Möller u. a. (2008, p. 649-650)).
Gottschalk u. a. haben einen weiteren Ansatz entwickelt, um eine Szene hierarchisch zu unterteilen und Kollisionen effizient zu erkennen (vgl. Gottschalk u. a. (1996)). Er basiert auf der Verwendung orientierter Hüllkörper („Orientend Bounding Boxes “ oder auch „OBB “) und lässt sich auf alle Szenen anwenden, die durch Polygone dargestellt werden. Laut Gottschalk ist der Vorteil von OBB, dass sie Objekte enger umschließen und somit weniger Rekursionstiefe benötigt wird, um zu testen, ob Objekte sich schneiden. Dabei werden die Hüllboxen an deren längsten Achse geteilt und die Polygone auf die dabei entsteheneden Hüllkörper verteilt, indem sie den dabei entstehenden neuen Boxen zugeordnet werden, die ihren Mittelpunkt enthalten. Laut Gottschalk ist dieser Algorithmus asymptotisch schneller als vergleichbare Algorithmen, die Kugeln und an den Achsen ausgerichtete Boxen als Hüllkörper verwenden. Außerdem ist er dazu fähig, Schnittpunkte in komplexen Modellen mit einer hohen Anzahl an Polygonen zu finden (vgl. ebd.).

Chang u. a. haben für die hierarchische Unterteilung eine Strategie entwickelt, die zum Ziel hat, den Ansatz von Gottschalk u. a. zu optimieren, sodass sich Schnittpunkte noch effizienter finden lassen (vgl. Chang u. a. (2008)). Nach Chang hat die Art der hierarchischen Unterteilung einen sehr starken Einfluss auf die Performance der Kollisionsfindung. Die Enge der Hüllkörper und die Komplexheit der Schnittpunktberechnungen sind dabei die wichtigsten Faktoren. Aus diesem Grund wird der hierarchische Baum von Gottschalk dual weiterentwickelt, indem um jeden Knoten je zwei Hüllkörper, nämlich eine Kugel und eine orientierte Box, gebildet werden. Die Kugeln liegen dabei ebenfalls möglichst eng um das Objekt, das sie beinhalten. Dieser Aufbau nutzt die Vorteile beider Formen: Kugeln lassen sich einfacher auf Schnittpunkte testen, und orientierte Hüllboxen liegen enger an dem Objekt, das sie umhüllen. Somit werden nur Objekte auf Kollsion getestet, die sehr nahe beieinander liegen, und der kostenaufwändigere Schnittpunkttest der orientierten Hüllkörper findet nur dann statt, wenn sich die zugehörigen Hüllkugeln schneiden.

2.5.3 Binary Space Tree

Ein Binary Space Tree unterteilt eine Szene räumlich. Es wird eine Ebene berechnet, die einen Schnitt darstellt und den ursprünglichen Raum teilt. Die enthaltenen Objekte werden anschließend den Unterräumen zugeordnet. Diese Teilung wird bis zur gewünschten Granularität

rekursiv weitergeführt.
Es gibt zwei Ansätze, die räumliche Aufteilung vorzunehmen: achsenparallel oder polygonorientiert.
Ein achsenparalleler Baum basiert auf einem Hüllkörper, deren Kanten parallel zu den Achsen des Koordinatensystems sind. Dieser Körper wird mit Hilfe einer Ebene rekursiv in zwei neue Körper geteilt. Dabei können sowohl gleich als auch unterschiedlich große Boxen entstehen. Es gibt verschiedene Möglickeiten, Polygone, die von der Ebene geschnitten werden, den Boxen zuzuordnen. Das Polygon kann beispielsweise auf dieser Ebene des Baumes gespeichert werden. Des Weiteren kann man es beiden Kindboxen zuordnen oder es wird von der Ebene aufgeteilt in zwei neue, kleinere Polygone.
Bei einem polygonorientierten Baum werden die Ebenen als Teiler verwendet, auf denen Polygone liegen. Polygone, die von einer solchen Ebene geschnitten werden, lassen zwei neue Polygone entstehen. Dieses Vorgehen wird rekursiv fortgesetzt, bis alle Polygone im Baum enthalten sind (vgl. Akenine-Möller u. a. (2008, p. 650-654)).
Es ist wichtig, darauf hinzuweisen, dass es bei einem Baum keine Grenze für die Rekursionstiefe gibt. Deshalb lassen sich beliebig viele Ebenen hinzufügen. Dies ist von Vorteil, wenn später Objekte hinzukommen oder verschoben werden und diese aus Effizienzgründen nicht aus ihren alten Knoten gelöscht werden sollen.
Eine weitere Folge ist, dass bei Implementierungen, in denen Objekte von Ebenen geteilt werden, die Rekursionstiefe beziehungsweise der Speicherbedarf stark steigen kann. Die Platzierung der Teiler-Ebenen ist daher von großer Bedeutung (vgl. Hughes u. a. (1990, p. 1088)).
Weiterhin optimiert es im Allgemeinen die Bearbeitungzeit einer Anfrage, wenn der Baum balanciert ist, da ein balancierter Baum auch im Worst Case eine kurze Bearbeitungsdauer bietet, die sich nicht von anderen Anfragen abhebt. Es gibt allerdings Fälle, in denen es Sinn macht, den Baum unbalanciert aufzubauen, beispielsweise wenn bestimmte Anfragen im Vergleich zu anderen besonders häufig gestellt werden (vgl. ebd., p.1088-1089).

2.5.4 Quad- und Octree

Quad- und Octrees funktionieren nach dem gleichen Prinzip. Sie unterscheiden sich nur im Bereich ihrer Anwendung: Quadtrees werden im zweidimensionalen Bereich eingesetzt, während Octrees im dreidimensionalen Raum Verwendung finden. Die Verwendung eines Binary Space Trees kann zu der gleichen räumlichen Aufteilung führen wie die Verwendung eines Octrees. (vgl. Akenine-Möller u. a. (2008, p. 655)).
Beim Aufbau eines Quad- beziehungsweise Octrees wird folgendermaßen vorgegangen: zuerst

wird ein minimales Quadrat (in 2D) beziehungsweise ein minimaler Würfel (in 3D) um das Bild beziehungsweise die Szene oder das Objekt gebildet. Anschließend werden Ebenen parallel zu den Achsen des Koordinatensystems gebildet, die durch den Mittelpunkt des Quadrats/ des Würfels verlaufen. Die Ebenen unterteilen das gesamte Bild/ die komplette Szene in vier beziehungsweise acht neue, kleinere Quadrate oder Würfel. Dieser Vorgang wird rekursiv fortgesetzt, bis das Blattkriterium erfüllt ist. Ein solches Blattkriterium kann beispielsweise eine maximale Rekursionstiefe oder eine maximale Anzahl an beinhalteten Objekten (wie zum Beispiel Dreiecke eines Dreiecksnetzes) sein (vgl. Akenine-Möller u. a. (2008, p. 654)). Abbildung 2.6 zeigt den Aufbau eines Quadtrees, bei dem das Blattkriterium die maximale Anzahl von einem Objekt pro Quadrant ist, das heißt es werden so lange Rekursionsschritte zur Unterteilung durchgeführt, bis jeder Quadrant nur noch null oder ein Objekt beinhaltet. Bei der Aufstellung der Blattkriterien sollte berücksichtigt werden, dass die hierarchische Struktur des Quad- beziehungsweise Octrees zum Ziel hat, bestimmte Anfragen schneller bearbeiten zu können (vgl. Samet und Webber (1988)). Eine zu hohe Granularität des Quad- oder Octrees kann die Performance bestimmter Anfragen nämlich auch verlangsamen.

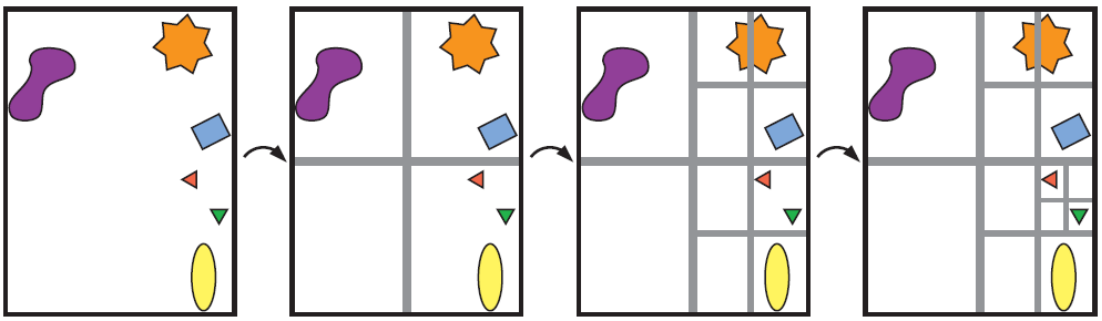

Abbildung 2.6: Rekursiver Aufbau eines Quadtrees. (Aus Akenine-Möller u. a. (2008, p. 655))

2.6 Mathematische Gegebenheiten und Grundlagen

Die Implementierung des View Frustum Cullings verwendet hauptsächlich zwei Funktionalitäten: die Bildung von Hüllkörpern, speziell Hüllboxen, und die Kollisionserkennung mit Ebenen. Die Verwendung von Octrees für Objekte und die gesamte Szene kann weitere Vorteile mitbringen. In diesem Kapitel werde ich die mathematischen Strukturen kurz erläutern.

2.6.1 Hüllkörper

Hüllkörper haben den Sinn, Kollisionen zwischen Objekten sowie Strahlen schneller und effizienter berechnen zu können und unnötige Kollisionstests mit den Objekten zu vermeiden. Hierbei ist es wichtig, dass der Hüllkörper möglichst eng am beinhalteten Objekt anliegt, da die Wahrscheinlichkeit, dass ein Strahl einen solchen Körper trifft, proportional zu der Oberfläche des Hüllkörpers steigt. Das Ziel solch enger Hüllkörper ist es, möglichst viele Kollisionsabfragen negativ ausfallen zu lassen, damit nicht unnötige Schnittpunktberechnungen gemacht werden wie beispielsweise dann, wenn ein Strahl zwar den Hüllkörper, aber nicht das beinhaltete Objekt schneidet.
Es gibt verschiedene Arten von Hüllkörpern: achsenparallele sowie orientierte Boxen und Kugeln, welche verschiedene Vor- und Nachteile mit sich bringen. Achsenparallele Boxen lassen sich leicht berechnen und führen zu einer einfachen Kollisionserkennung, umschließen ein Objekt dafür aber oft nicht so eng. Orientierte Boxen liegen eng am Objekt an, erschweren dafür aber die Berechnungen für die Kollisionserkennung. Kugeln bieten eine einfache Kollisionserkennung, bringen aber den Nachteil mit sich, dass häufig viel Platz zwischen dem beinhalteten Objekt und seinem Hüllkörper ist, was dazu führt, dass viele Kollisionstests mit der Kugel positiv ausfallen, mit dem entsprechenden Objekt jedoch nicht.
Um eine achsenparallele Hüllbox zu bilden, werden je Achse die Minima und Maxima aller Punkte des Objekts gesucht. Die linke untere Ecke wird durch die Minima, die rechte obere Ecke durch die Maxima definiert (vgl. Akenine-Möller u. a. (2008, p. 732)).
Bei der Bildung von Hüllkugeln gibt es verschiedene Ansätze. Häufig wird zunächst eine achsenparallele Box gebildet. Anschließend wird die Kugel mithilfe des Zentrums und der Diagonalen der Box definiert. Eine mögliche Optimierung besteht darin, ausgehend vom Zentrum der Box den am weitesten entfernten Punkt des Objekts zu finden und die Distanz zu ihm als Radius festzulegen.
Ritter hat einen Algorithmus entwickelt, um eine annähernd optimale Hüllkugel zu finden (vgl. Ritter (1990)). Er teilt sich in zwei Schritte: Zuerst wird über alle Punkte des Objekts iteriert und dabei festgehalten, welche Punkte für jede Achse Minimum und Maximum enthalten. Die

Punkte mit der größten Distanz auf der jeweiligen Achse werden im ersten Schritt für die Definition der Kugel verwendet. Im zweiten Schritt wird nochmals über jeden Punkt iteriert und die Kugel erweitert, sofern ein Punkt außerhalb von ihr liegt. Laut Ritter hat dieser Algorithmus eine Laufzeit von $O(n)$, wobei n die Anzahl der Punkte ist und erbringt eine Kugel, die 5% größer als die ideale Kugel ist (vgl. ebd.).
Die Berechnung orientierter Hüllboxen gestaltet sich wesentlich komplexer. Gottschalk hat einen Ansatz entwickelt, bei dem zuerst eine konvexe Hülle um alle Dreiecke des Dreiecksnetzes gebildet wird. Dafür wird angenommen, dass die Fläche der Abbildung eines Dreiecks in der Hülle durch folgende Formel berechnet werden kann (vgl. Gottschalk u. a. (1996)):

$$A_i = \frac{1}{2} \left| (\vec{p_i} - \vec{q_i}) \times (\vec{p_i} - \vec{r_i}) \right| \tag{2.2}$$

p_i, q_i, r_i sind die drei Eckpunkte des Dreiecks.
Die Gesamtoberfläche der konvexen Hülle ist dann die Summe aller Dreiecksflächen (vgl. ebd.):

$$A_H = \sum_{i}^{n} A_i \tag{2.3}$$

Der Mittelpunkt eines Dreiecks lässt sich wie folgt berechnen (vgl. ebd.):

$$\vec{c_i} = \frac{\vec{p_i} + \vec{q_i} + \vec{r_i}}{3} \tag{2.4}$$

Daraus lässt sich der Mittelpunkt der konvexen Hülle bestimmen (vgl. Akenine-Möller u. a. (2008, p. 734)):

$$\vec{c}_H = \frac{1}{A_H} \sum_{i=0}^{n-1} A_i \vec{c_i} \tag{2.5}$$

Somit lässt sich eine Kovarianzmatrix aufstellen (vgl. Gottschalk u. a. (1996)):

$$C_{jk} = \sum_{i=1}^{n} \frac{A_i}{12 A_H} (9 c_{j,i} c_{k,i} + p_{j,i} p_{k,i} + q_{j,i} q_{k,i} + r_{j,i} r_{k,i}) - c_{j,H} c_{k,H} \tag{2.6}$$

Die aus der Kovarianzmatrix berechneten normalisierten Eigenvektoren bilden die Richtungsvektoren der orientierten Hüllbox. Mithilfe der Minima und Maxima auf jeder Achse kann die Hüllbox anschließend eingegrenzt werden und ist somit klar definiert. Die Komplexität dieses Algorithmus beträgt $O(n \log n)$, wobei n die Anzahl der Dreiecke ist (vgl. Akenine-Möller u. a. (2008, p. 733-734)).

2.6.2 Ebene

Ebenen sind im Zusammenhang mit View Frustum Culling wichtig, weil sie das View Frustum eingrenzen. Mit ihrer Hilfe lässt sich außerdem feststellen, ob ein Objekt innerhalb oder außerhalb davon liegt oder es geschnitten wird.
Ebenen lassen sich durch zwei Formen darstellen: explizit und implizit (vgl. Akenine-Möller u. a. (2008, p. 908)).
Bei der expliziten Darstellung wird die Ebene durch einen Punkt ($\vec{o}$) und zwei Richtungsvektoren ($\vec{s}$ und $\vec{t}$) charakterisiert (siehe Formel 2.7):

$$\vec{p}(u,v) = \vec{o} + u\vec{s} + v\vec{t} \tag{2.7}$$

Die implizite Darstellung charakterisiert eine Ebene mit Hilfe eines Punktes ($\vec{p}$) auf dieser, ihrer Normalen ($\vec{n}$) und einer Konstanten (d), die ihre Position definiert (siehe Formel 2.8):

$$\vec{n} * \vec{p} + d = 0 \tag{2.8}$$

2.6.3 Normale einer Ebene

Die Normale einer Ebene gibt an, in welche Richtung die Ebene gerichtet ist. Dies ist beispielsweise relevant für die Beleuchtungsrechnung. Außerdem lässt sich mit Hilfe der Normalen berechnen, ob ein Punkt „vor“ oder „hinter“ der Ebene liegt. In der Implementierung des View Frustum Cullings ist diese Eigenschaft sehr wichtig für die Kollisionserkennung von Objekten mit dem View Frustum.
Die Normale einer Ebene lässt sich auf mehrere Wege berechnen. Zum einen ist es möglich, das Kreuzprodukt der beiden Richtungsvektoren der expliziten Darstellung zu berechnen, um die Normale zu erhalten (vgl. Akenine-Möller u. a. (2008, p. 908/909)):

$$\vec{n} = \vec{s} \times \vec{t} \tag{2.9}$$

Außerdem kann die Normale mit Hilfe dreier gegebener Punkte, die auf der Ebene liegen, berechnet werden. Dazu muss aus den Differenzen der Punkte das Kreuzprodukt gebildet werden (vgl. ebd.):

$$\vec{n} = (\vec{u} - \vec{w}) \times (\vec{v} - \vec{w}) \tag{2.10}$$

Die Ergebnisse von Gleichungen 2.9 sowie 2.10 müssen anschließend noch normiert werden.

3 Kameramodell als Grundlage für das View Frustum Culling

3.1 Vorstellung diverser Kameramodelle

3.1.1 Kameramodell der Implementierung

Die Kamera in der Computergrafik repräsentiert das „Auge“ des Betrachters. Sie bringt diverse Parameter mit, aus denen sich das View Frustum ableiten lässt (Skizze des Kameramodells mit seinen Parametern siehe Abbildung 3.1).
Mit Hilfe der Parameter kann zur Laufzeit und in Abhängigkeit von Kameraänderungen immer wieder ein neues View Frustum berechnet werden, sodass Objekte dynamisch sichtbar und unsichtbar geschaltet werden können. Die Parameter der Kamera werden im Folgenden erklärt.

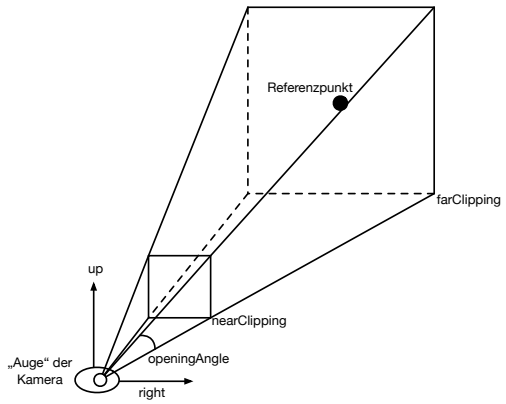

Abbildung 3.1: Kameramodell der Implementierung. (Eigene Skizze)

Das Auge der Kamera wird über einen Punkt in Form eines Vektors definiert. Es gibt die Position an, von der aus die Szene betrachtet wird. Das Resultat ist eine Art Lochkamera.

Strahlen, die von der Kamera aufgenommen werden, müssen alle diesen Punkt passieren (vgl. Hughes u. a. (1990, p. 301)).
Der Bezug- oder Referenzpunkt gibt die Blickrichtung der Kamera an. Wenn ein Strahl genau vom Augpunkt startend mit Richtung zum Referenzpunkt ein Objekt trifft, ist davon auszugehen, dass dieses in der Mitte des Sichtbarkeitsbereichs liegt (vgl. ebd., p. 302).
Der Öffnungswinkel gibt in Grad an, wie weit das Sichtfeld der Kamera aufgespannt ist beziehungsweise wie weit sie vom Bezugspunkt aus sehen kann. In der Implementierung zu dieser Arbeit beträgt der Winkel 45°. Das Verhältnis von Höhe zu Breite ist 1.0, sodass das View Frustum gleich hoch wie breit ist. Der Winkel sowie das Verhältnis zwischen Höhe und Breite sind variabel, es bietet sich aber an, diese Parameter auf die genannten Werte zu setzen, um einen realistischen Sichtbarkeitsbereich ähnlich der Pyramidenform zu erhalten.
Der Up-Vektor gibt relativ zur Kamera an, in welche Richtung es „nach oben“ geht. Anfangs ist dieser Vektor durch $up = (0, 1, 0)^T$ definiert, das heißt die Höhe erstreckt sich entlang der Y-Achse. Dies kann sich - in Ahängigkeit der Kamerabewegung- ändern.
Des Weiteren gibt es noch einen Vektor, der relativ zur Kamera angibt, in welche Richtung die Rechtsausrichtung geht. Anfangs ist dieser Vektor auf $right = (1, 0, 0)^T$ gesetzt, er muss sich aber ebenfalls dynamisch zu Kamerabewegungen mitverändern.
Mit diesen Parametern ist die Kameraperspektive über ihre Position sowie die Blickrichtung und Spannweite klar definiert. Es fehlen noch die Distanzen zur nahen und fernen Clipping-Ebene, um das View Frustum komplett einzugrenzen.
Die Distanzen der nahen und fernen Clipping-Ebene geben an, von wo bis wohin der Sichtbarkeitsbereich sich erstreckt. In der Implementierung zu dieser Arbeit erstreckt sich diese Distanz im initialen Zustand entlang der Z-Achse. Somit ist das View Frustum komplett definiert, sodass sich die Ebenen, die es eingrenzen, berechnen lassen.
Die genannten Parameter lassen sich in intrinsische und extrinsische unterteilen: Die extrinsischen sind durch Bewegung der Kamera veränderbar (wie beispielsweise die Orientierung), während die intrinsischen (wie beispielsweise der Öffnungswinkel) unabhängig von Position und Bewegungen sind.
Laut Hughes u. a. (vgl. ebd., p. 302) kann es sinnvoll sein, die Distanz zur nahen Ebene so dicht zu setzen, wie das Objekt von Interesse entfernt ist, um zu verhindern, dass es im resultierenden Bild von anderen Objekten verdeckt ist. Weiterhin macht es bei der Wahl der Distanz zur fernen Ebene Sinn, diese aufgrund der begrenzten Auflösung des Z-Puffers nicht zu groß zu wählen. Somit sollten Objekte, die aufgrund ihrer Entfernung in der Szene keine Relevanz haben, ignoriert werden (vgl. ebd., p. 302/303). In der Implementierung ist die freie Wahl dieser Distanzen nicht möglich, da hier - wie alle anderen intrinsischen Kameraparameter - die Near-

und Far-Clipping-Distanzen vom Kameraobjekt bestimmt sind.

3.1.2 Schwächen des Kameramodells und Verbesserungsansätze

Laut Kolb u. a. können aktuelle Kameramodelle der Computergrafik das Verhalten einer Linsenkamera in Bezug auf Räumlichkeit nicht korrekt simulieren (vgl. Kolb u. a. (1995)). Der Fokus dieser Modelle liegt vermehrt auf korrekter Lichtberechnung statt auf den Berechnungen der Geometrie und Räumlichkeit. Dabei gibt es nach Kolb diverse Situationen, in denen ein physikalisch korrekt arbeitendes Kameramodell von Bedeutung ist, beispielsweise bei der Zusammensetzung von Bildern aus echten und synthetischen Teilstücken oder bei der Anwendung von dreidimensionalen Simulationsanwendungen, welche häufig von Personen verwendet werden, die auch den Umgang mit „echten“, nicht digitalen Kameras gewohnt sind (vgl. ebd.).
Die Implementierung seines Kameramodells basiert auf den physikalischen Gegebenheiten, die von der Kameralinse und der Räumlichkeit der Szene bedingt werden. Ziel dieser Implementierung ist es, die Simulation des Kameramodells von Räumlichkeit beziehungsweise Geometrie sowie die Strahlenverfolgung und -ausbreitung zu verbessern. Um dies zu erreichen, verwendet er das Prinzip der Strahlenverfolgung („Ray Tracing“).
Die Implementierung soll gegenüber den Standard-Kameramodellen eine Verbesserung in verschiedenen Bereichen bringen, die im Folgenden kurz erläutert werden.
Das Verhältnis zwischen Kameralinse, aufgenommenen Objekten und der Filmebene, auf der das Aufgenommene abgebildet wird, ist präziser. Somit sind der Sichtbarkeitsbereich und die Tiefenrelationen dichter an denen von realen Kameras.
Außerdem werden Lichtstrahlen durch die gesamte Szene verfolgt. Dies hat zur Folge, dass die Räumlichkeit korrekt wiedergegeben werden kann. Zusätzlich können Verzerrungen simuliert werden, wie zum Beispiel bei Verwendung einer Fischaugen-Linse (leicht kugelförmige Abbildung) oder einer anamorphischen Linse (Stauchung des Bildes).
Eine weitere Verbesserung ist bei der Beleuchtung zu beobachten, da Lichtstrahlen bei ihrer Verfolgung gewichet werden.
Das von Kolb u. a. entwickelte Modell bringt dennoch Bereiche mit sich, die Verbesserung bedürfen. So wurde bei der Umsetzung beispielsweise davon ausgegangen, dass die Linsenübertragung perfekt ist. Weiterhin wurden wellenlängen-abhängige Effekte wie Sensor-Sensitivität ignoriert (vgl. ebd.).

Auch Barsky u. a. stellt fest, dass das menschliche Auge den Unterschied zwischen von Com-

putern berechneten und fotografisch aufgenommenen Bildern sehr leicht erkennt (vgl. Barsky u. a. (2003a)). Dies führt er darauf zurück, dass in computergrafisch berechneten Bildern keinerlei Fokus beziehungsweise Unschärfe zu erkennen ist. Als Grund hierfür nennt er, dass die Transformation einer dreidimensionalen Szene in ein zweidimensionales Bild sehr komplex ist. Im Folgenden werden die von ihm vorgestellten Kameramodelle aufgeführt.
Das Standard-Kameramodell in der Computergrafik simuliert eine Lochkamera. Alle Lichtstrahlen der Szene passieren eine Linse mit einer unendlich kleinen Öffnung. Bei diesem Modell erscheint die komplette Szene scharf, da von allen Punkten, unabhängig von ihrer Position in der Szene, ein Lichtstrahl losgeschickt wird. Laut Barsky u. a. ist dieses Modell mit einer punktförmigen Linse in der Realität nicht umsetzbar, da das resultierende Bild zu trübe wäre, um etwas erkennen zu können.
Ein weiteres Modell ist die „Thin Lens Approximation“ (vgl. ebd.). Bei diesem Modell wird angenommen, dass zwar die Linsenöffnung endlich, dafür aber ihre Dicke unendlich klein ist. Auf der optischen Achse, welche durch die Mitte der Linse verläuft, liegt der Blickpunkt F („focal point“, vgl. ebd.). Lichstrahlen, die diesen Punkt passieren beziehungsweise auf ihn zulaufen, besitzen die Eigenschaft, dass sie parallel zu der optischen Achse verlaufen, nachdem sie durch die Linse gebrochen wurden. Der Abstand zwischen dem Punkt F und der Mitte der Linse heißt fokale (zentrale) Länge („focal length“, vgl. ebd., siehe f in Abbildung 3.2).

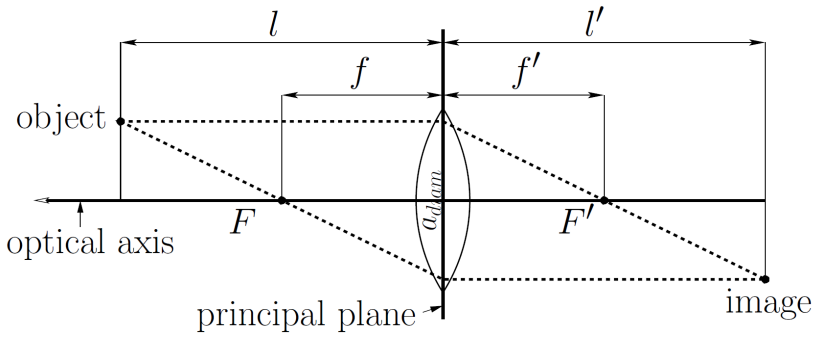

Abbildung 3.2: Thin Lens Approximation. (Aus (Barsky u. a., 2003a, p. 3))

Das resultierende Bild bei der Aufnahme durch dieses Modell wird durch folgendes Gesetz beschrieben (vgl. ebd.):

$$\frac{1}{l} + \frac{1}{l'} = \frac{1}{f} \tag{3.1}$$

l ist der Abstand zwischen Objekt und Linse, l' ist die Distanz von der Linse zu dem resultierenden Bild (beziehungsweise zu der Bildebene) und f ist die fokale Länge. Diese Formel nach l' aufgelöst ergibt (vgl. ebd.):

$$l' = \frac{fl}{l - f} \tag{3.2}$$

Mit dieser Gleichung lässt sich der Abstand für ein Objekt berechnen, bei dem seine Projektion konvergiert.
Um nun auf eine bestimmte Distanz d_{focus} zu fokussieren, muss die Bildebene, auf der das resultierende Bild entsteht, ebenfalls einen bestimmten Abstand d'_{focus} zur Linse haben. Dieser Abstand lässt sich folgendermaßen berechnen (vgl. ebd.):

$$d'_{focus} = \frac{fd_{focus}}{d_{focus} - f} \tag{3.3}$$

Gleichung 3.3 entspricht Gleichung 3.2, nur dass l sowie l' durch d_{focus} sowie d'_{focus} ersetzt wurden. Punkte, die näher an der Linse liegen als d_{focus}, werden hinter die Bildebene projiziert, während Punkte, die weiter weg von der Linse liegen, vor der Bildebene abgebildet werden. Daraus folgt, dass die Projektion der Punkte, die nicht die Distanz d_{focus} zur Bildebene haben, einen Kreis entstehen lässt, an dem sich die „Defokussierung“ (vgl. ebd.) dieses Bildpunktes feststellen lässt.
Die „Thick Lens Approximation“ (vgl. ebd.) wird für Fälle angewendet, bei denen die Dicke der Linse im Verhältnis zur fokalen Länge klein ist. Eine dicke Linse hat zwei konvexe Oberflächen, die eine bestimmte Distanz voneinander entfernt sind. Das Verhalten einer solchen Linse ist ähnlich zu dem einer dünnen Linse, bis auf dass auf die Lichstrahlen beim Passieren der Linse eine Translation parallel zur optischen Achse erfolgt. Diese Transformation muss berücksichtigt werden, damit die Gleichungen wie bei Thin Lens Approximation gelten, um berechnen zu können, auf welcher Entfernung das resultierende Bild entsteht.
In einem „Full Lens System“ (vgl. ebd.) gibt es mehrere Linsen mit konvexen Oberflächen. Bei Bewegung der Bildebene bewegen sich eine oder mehrere Linsen mit, sodass sich der Blickpunkt dynamisch mitverändert. Dieses Modell bringt den Vorteil mit sich, dass sich durch die Bewegung der Linsen die Platzierung der Bildebene mitverändert, sodass sich der Sichtbarkeitsbereich ähnlich wie in einem physikalischen System ebenfalls mitverändert. Bei den zuvor vorgestellten Kameramodellen hingegen befindet sich die Bildebene unabängig von Kamerabewegungen immer an demselben Platz.
Die vorgestellten Kameramodelle repräsentieren „object space techniques“ (vgl. ebd.), das heißt sie arbeiten im dreidimensionalen Raum. Sie nehmen Komplexität und weniger Effizienz in

Kauf, um höhere Genauigkeit und verbesserten Realismus zu erreichen.

Es gibt aber auch Techniken, die das bereits gerenderte Bild bearbeiten, um Fokussierung und somit höheren Realismus zu erreichen. Diese Techniken werden bildbasiert („image based“ (vgl. Barsky u. a. (2003b))) genannt, da sie mit dem zweidimensionalen Bild arbeiten .
Eine dieser bildbasierten Techniken ist der von Potmesil u. a. entwickelte „Image-Based Blur“ (vgl. Potmesil und Chakravarty (1982)) . Hierbei wird von dem bereits gerenderten Bild jedes Pixel eingetrübt, indem ein Zerstreuungskreis („circle of confusion“, vgl. ebd.) über es gelegt wird. Der Radius dieses Kreises hängt von der Position beziehungsweise der Tiefe des Pixels ab, die es wiedergibt.
Im Jahre 1982, als der Algorithmus von Potmesil u. a. entwickelt und getestet wurde, hat diese Bildbearbeitung auf einem 512x512 großen Bild zwischen drei und 125 Minuten gedauert (vgl. Potmesil und Chakravarty (1982)). Folglich muss abgewogen werden, ob eine solche Performance-Einbußung durch stärkeren Realismus im Bild ausgeglichen werden kann.
Weitere Nachteile des Algorithmus von Potmesil u. a. sind laut Barsky et. al, dass durch die Anwendung verschiedener Zerstreuungskreise das Gesamtbild aufgehellt wird und dass womöglich Objekte teils getrübt, teils fokussiert dargestellt werden (vgl. Barsky u. a. (2003b)).
Eine weitere bildbasierte Technik ist der von Shinya entwickelte „Ray Distribution Puffer“ (oder RDB-Puffer, vgl. Shinya (1994)). Dieser Algorithmus gliedert sich in mehrere Schritte: Zuerst wird das Bild inklusive zugehörigem Z-Puffer gerendert. Anschließend wird für jedes Pixel ein RDB-Puffer angelegt. In diesen Puffer werden Farb- und Z-Werte eingetragen, welche entstehen, wenn von diesem Pixel abgehende Lichtstrahlen andere Pixel treffen. Dabei ist es wünschenswert, einheitliche Lichtstrahlen zu haben (siehe Abbildung 3.3). Deshalb wird für jeden Strahl ein Richtungsvektor definiert, der ebenfalls im RDB-Puffer gespeichert wird. Nun wird für jedes Pixel p der Zerstreuungskreis berechnet. Für jedes Pixel q, das in diesem Zerstreuungskreis liegt, wird sein Lichstrahlrichtungsvektor berechnet und der zugehörige Slot im RDB-Puffer gesucht. Dann werden die Z-Werte von p und q verglichen, und der kleinere Wert sowie der zugehörige Farbwert wird für Pixel q eingetragen. Abschließend wird für jedes Pixel der Durchschnitt der Farbwerte im RDB-Puffer gebildet und für dasjenige Pixel gespeichert.
Dieser Algorithmus bringt eine Rechenzeit mit sich, die nahezu unabhängig von der Größe des RDB-Puffers und der Szenenkomplexität ist (vgl. ebd.).

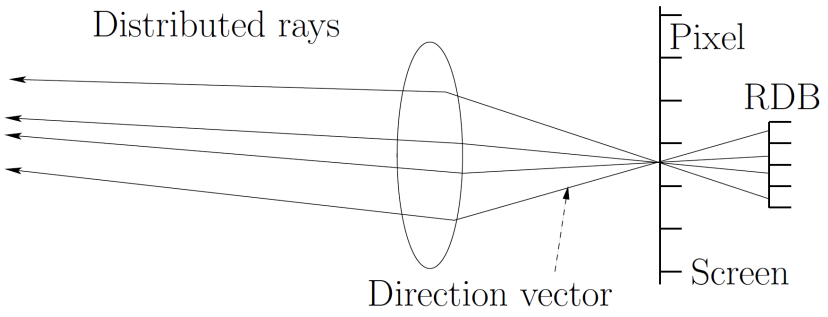

Abbildung 3.3: Ray Distribution Puffer. (Aus (Barsky u. a., 2003b, p. 3)

Beim „Blurring by Discrete Depth“ (vgl. Barsky u. a. (2003b)) wird das Bild in Teilstücke aufgeteilt. Diese Teilstücke werden anhand ihrer Distanz zur Kamera voneinander abgegrenzt. Jedes dieser Teilstücke wird anschließend mit einem bestimmten Filter eingetrübt, um abschließend das Gesamtbild aus den einzelnen Teilstücken wieder zusammenzusetzen. Wenn der Fall auftritt, dass verschiedene Eintrübungen im zusammengesetzten Bild aufeinandertreffen, sodass „scharfe“ Grenzen entstehen, kommt lineare Interpolation zum Einsatz, um die Schärfe der Grenzen abzuschwächen. Im Test von Barsky hat dieser Algorithmus für die Bearbeitung eines Bildes mit elf Tiefengraden 156 Sekunden gebraucht und war somit schneller als Distributed Ray Tracing (vgl. ebd.).
Weiterhin gibt es diverse „Light Field Techniques“ (vgl. ebd.) oder auch Lichtfeldrendering. Hierbei wird ein bereits vorhandenes Bild aus der Sicht mehrerer Perspektiven reproduziert, um verschiedene Lichteinfälle zu simulieren. Der Vorteil dieser Techniken ist, dass keine Informationen über die Geometrie und Räumlichkeit der Szene vorliegen müssen, damit neue Abbildungen entstehen können. Stattdessen kann man Informationen über die Räumlichkeit indirekt über die verschiedenen Perspektiven erhalten (vgl. ebd.).

3.2 Erkennung der Position von Objekten bezüglich des View Frustums

Es gibt diverse Ansätze und Optimierungen für das Erkennen der Position von Objekten bezüglich des View Frustums. Im Folgenden werden die Grundlagen hierfür präsentiert sowie einige Optimierungsansätze vorgestellt.

3.2.1 Grundlagen und Gegebenheiten zur Bestimmung der Position eines Objektes

Das View Frustum wird mit Hilfe von sechs Ebenen definiert: nah und fern, links und rechts sowie oben und unten. Wie bereits im vorigen Kapitel beschrieben, lassen sich diese Ebenen mit Hilfe der gegebenen Kameraparameter eindeutig bestimmen.
Die eingrenzenden Ebenen sind über einen Punkt auf dieser und eine Normale klar definiert. Beim Aufbau der Ebenen ist es wichtig darauf zu achten, dass sich die Ebenen und ihre Normalen immer relativ zur Kamera verhalten müssen. Dies hat zur Folge, dass zum Beispiel die Distanz zwischen rechts und links sich initial auf der X-Achse erstreckt, aber nach einer Änderung des Augpunktes der Kamera sich auch über andere beliebige Achsen erstrecken kann.

```
UP = ( 0,000 1,000 0,000 )
RIGHT = ( 1,000 0,000 0,000 )
REF = ( 0,000 0,000 0,000 )
FBL = ( -5,579 -5,579 -5,000 )
FBR = ( 5,579 -5,579 -5,000 )


Kamera-Auge auf (5,0,0) verschoben:
UP = ( 0,000 1,000 0,000 )
RIGHT = ( 0,000 0,000 -1,000 )
REF = ( 0,000 0,000 0,000 )
FBL = ( -5,000 -5,579 5,579 )
FBR = ( -5,000 -5,579 -5,579 )
```

Abbildung 3.4: Ausgabe der Kameraparameter up, $right$ und ref sowie zweier Eckpunkte eines View Frustums vor und nach einer Verschiebung des Augpunktes der Kamera auf $(5, 0, 0)$. (Ausgabe des Programms)

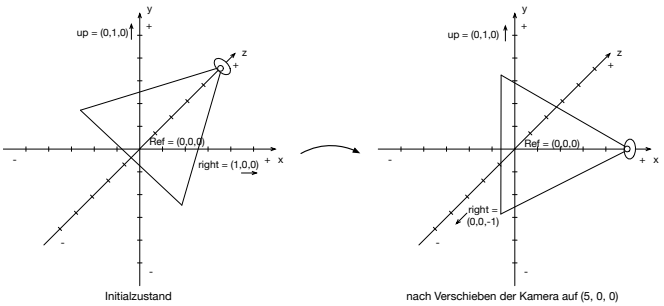

Abbildung 3.5: Skizzierter Sichtbarkeitsbereich der Kamera im Initialzustand, in dem sich die Rechts-Links-Distanz auf der X-Achse erstreckt, und nach Verschiebung. Nach der Verschiebung erstreckt sich die Rechts-Links-Distanz auf der Z-Achse. (Eigene Skizze)

In Abbildung 3.4 ist die Ausgabe der Kameraparameter up, $right$ und des Referenzpunktes ref zu sehen sowie zweier Eckpunkte des View Frustums („Far Bottom Left“ (FBL), sowie „Far Bottom Right“ (FBR)), vor und nach einer Verschiebung der Kamera auf den Punkt $(5, 0, 0)$. Es wird deutlich, dass sich im Initialzustand, in welchem die Kamera auf $(0, 0, 5)$ positioniert ist, der Up-Vektor entlang der Y-Achse und der Right-Vektor entlang der X-Achse erstreckt. Nach der Kameraverschiebung hingegen erstreckt sich die Rechtsausrichtung entlang der Z-Achse. Dies ist auch bei den Eckpunkten nachvollziehbar. Nachdem sich im Initialzustand die Distanz zwischen „FBL“ (links) und „FBR“ (rechts) auf der X-Achse erstreckt hat (Distanz : $|FBL_x| + |FBR_x| = 11,158$), befindet sich nach Verschiebung diese Distanz auf der Z-Achse ($|FBL_z| + |FBR_z| = 11,158$). Auch Abbildung 3.5 verdeutlicht, dass sich unter anderem die Rechtsausrichtung der Kamera nach einer Verschiebung verändern kann.
Des Weiteren müssen die Normalen immer nach innen gerichtet sein, also in das View Frustum zeigen (siehe Abbildung 3.6), um später korrekt bestimmen zu können, ob ein Objekt innerhalb oder außerhalb liegt oder ob es vom View Frustum geschnitten wird. Wie die Kamera-Parameter korrekt in die Berechnungen eingesetzt werden, um Ebenen und Normalen zu definieren, wird später noch genauer beschrieben.

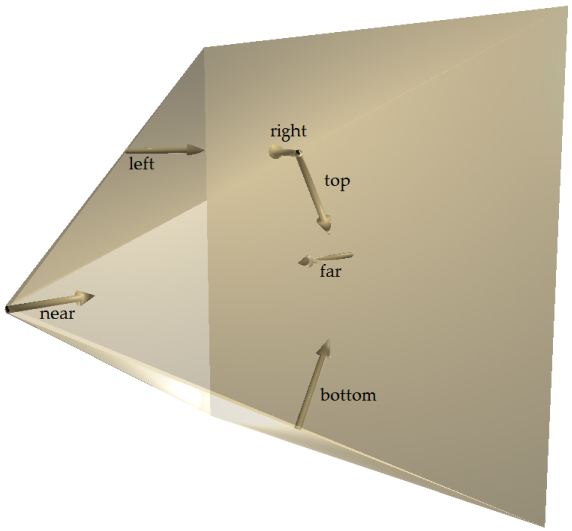

Abbildung 3.6: Ansicht eines Sichtbarkeitsbereichs beziehungsweise View Frustums und den zugehörigen (beschrifteten) Normalen der Ebenen. (Ausgabe des Programms)

Nachdem das View Frustum aufgebaut wurde, lässt sich die Position eines Objekts bezüglich des View Frustums bestimmen. Hierfür gibt es verschiedene Vorgehensweisen und Optimie-

rungen, welche diverse Vor- und Nachteile hinsichtlich Genauigkeit und Effizienz mitbringen. Die Schnittpunkte sowie die Position unterschiedlicher Hüllkörper (Box, Kugel) mit dem View Frustum lassen sich unterschiedlich berechnen. Im Folgenden werde ich mich auf achsenparallele Hüllboxen beschränken, da in der Implementierung ausschließlich diese als Hüllkörper verwendet werden.
Unabhängig vom verwendeten Hüllkörper sollte der Test für die Position von Objekten so früh wie möglich terminieren, um möglichst hohe Performance zu erreichen. Liegt ein Hüllkörper komplett innerhalb oder außerhalb des View Frustums, sind weitere Schnittpunkttests auf tieferer Ebene der Hüllkörperhierarchie nicht mehr notwendig und die beinhalteten Objekte müssen komplett oder gar nicht gerendert werden. Wird ein Hüllkörper jedoch geschnitten, können weitere Schnittpunkttests auf tieferer Hierarchieebene des Hüllkörpers sinnvoll sein, um die (teilweise) sichtbaren Objekte (oder Objektteile) zu extrahieren. Die weitere Behandlung von Hüllkörpern, die geschnitten werden, kann allerdings teuer und aufwändig sein. Eine Möglichkeit dieser Behandlung ist es, potenziell geschnittene Hüllkörper als komplett im View Frustum liegend zu betrachten. Dies hat möglicherweise zur Folge, dass auch nicht sichtbare Objekte (oder Objektteile) gerendert werden.

Eine andere Möglichkeit, einen Hüllkörper, der nicht als komplett außerhalb liegend definiert werden kann, zu behandeln, ist es, auf tiefere Hierarchiebenen zu gehen und wiederum Schnittpunkttests durchzuführen. Somit können kleinere Teile des Hüllkörpers, die innerhalb des View Frustums liegen, extrahiert werden.
Die beiden genannten Ansätze - die Annahme, dass ein Objekt komplett innerhalb des View Frustums liegt, sofern es nicht komplett außerhalb liegt, sowie die Durchführung weiterer Schnittpunkttests auf tieferen Hierarchieebenen - haben Mehraufwand zur Folge, sodass fall- und andwendungsabhängig entschieden werden muss, wie mit geschnittenen Hüllkörpern umgegangen werden soll.
Des Weiteren ist anzumerken, dass es nicht zu signifikanten Folgen oder Verschlechterungen der Performance kommt, wenn die Tests für innerhalb und außerhalb dahingehend vereinfacht werden, dass wenige Objekte fälschlicherweise als innerhalb erkannt werden, auch wenn sie außerhalb liegen, sofern ihre Position speziell ist. Dieser Ansatz kann im schlimmsten Fall zur Folge haben, dass Objekte gerendert werden, die nicht sichtbar sind. Hingegen gilt es zu verhindern, dass Objekte innerhalb des Frustums als außerhalb bestimmt werden, da dies zu Fehlern beim Rendering führen kann (vgl. Akenine-Möller u. a. (2008, p. 771-772)).

3.2.2 Optimierungsansätze zur Positionsbestimmung von Objekten bezüglich des View Frustums

Assarson u. a. haben diverse Ansätze entwickelt, um die Schnittpunktfindung zwischen dem View Frustum und Hüllboxen zu verbessern beziehungsweise zu optimieren (vgl. Assarsson und Moller (2000)).
Die erste Methode besteht darin, den Test für die Bestimmung der Position eines Objektes auf zwei Eckpunkte seiner achsenparallelen Hüllbox zu reduzieren. Die Wahl der Punkte fällt dabei auf die beiden, deren Verbindungsdiagonale der Normalen der zu testenden Ebene am nächsten kommt und durch den Mittelpunkt der Box verläuft. Assarson u. a. haben diese Eckpunkte „*p-vertex*“ und „*n-vertex*“ genannt (vgl. ebd.). Im ersten Schritt wird überprüft, ob der *n-vertex* außerhalb jeder Ebene des Frustums liegt (siehe der Würfel oben rechts in Abbildung 3.7). Sobald dieser Test für eine Ebene positiv ist, sind laut Assarson weitere Tests überflüssig, da ab dieser Stelle sicher ist, dass die Hüllbox komplett außerhalb des Frustums liegt. Sind hingegen alle Tests negativ, wird damit fortgesetzt, den *p-vertex* gegen jede Ebene zu testen. Wenn in diesem Schritt alle Tests anzeigen, dass der *p-vertex* innerhalb jeder Ebene liegt, befindet sich laut Assarson u. a. die Hüllbox komplett im Frustum (siehe der Würfel unten links in Abbildung 3.7). Andernfalls wird die Hüllbox vom View Frustum geschnitten (vgl. ebd.).

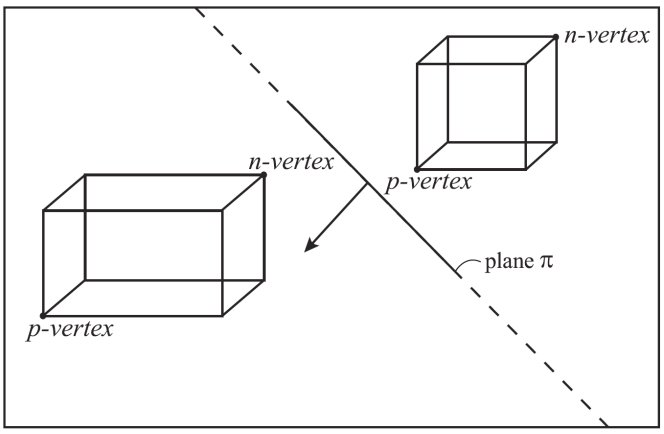

Abbildung 3.7: Optimierungsansatz von Assarson u. a. (Aus Assarsson und Moller (2000, p. 4))

Der zweite Optimierungsweg besteht darin, zeitliche Kohärenz zu nutzen. Wenn eine Hüllbox in einem Frame außerhalb des Frustums liegt, ist die Wahrscheinlichkeit bei kleiner Kamerabewegung relativ groß, dass diese Hüllbox im folgenden Frame immer noch außerhalb des View Frustums liegt. Liegt eine Hüllbox also außerhalb einer Ebene, wird in ihrer Datenstruktur ein

Index gespeichert, der auf diese Ebene zeigt, sodass für den nächsten Frame zuerst gegen diese Ebene getestet wird. Dies soll zu möglichst schneller Terminierung des Tests führen. Ist der Test gegen die gespeicherte Ebene hingegen positiv, werden weitere Tests gegen die anderen Ebenen durchgeführt (vgl. ebd.).
Eine andere Variante ist Kohärenz, die auf Rotationen und Translationen von bewegbaren Objekten anwendbar ist. Dabei gilt, dass ein Objekt rechts außerhalb vom View Frustum bei beispielsweise einer Rotation nach links, die kleiner als 180 Grad ist, immer noch außerhalb liegen muss (vgl. ebd).
Ein weiterer Ansatz spaltet das View Frustum entlang jeder Achse in acht Teile, ähnlich wie bei der ersten Rekursionsstufe eines Octrees. Anders als beim Octree sind hier die entstehenden Oktanten jedoch nicht würfelförmig. Mit diesem Ansatz lässt sich die Anzahl der Tests reduzieren, sofern die Distanz vom Hüllkörpermittelpunkt zum Hüllkörperrand kleiner ist als der kleinste Abstand zwischen View Frustum-Mittelpunkt und -Ebenen. Somit sind nur noch Tests gegen die äußeren drei Ebenen des Oktants nötig, in dem sich der Mittelpunkt des Hüllkörpers befindet (vgl. Abbildung 3.8, wo für die Kugel in der 2D-Abbildung nur Tests gegen π_a und π_b notwendig sind). Liegt der Hüllkörper innerhalb dieser zwei beziehungsweise drei Ebenen (in 3D), ist davon auszugehen, dass die Kugel komplett im View Frustum ist. Wenn er hingegen außerhalb einer dieser Ebenen positioniert ist, steht fest, dass die Kugel komplett außerhalb des View Frustums liegt, andernfalls wird sie vom View Frustum geschnitten (vgl. ebd.).

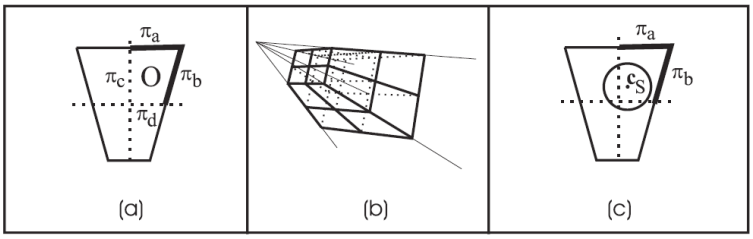

Abbildung 3.8: Aufteilung des View Frustums in Oktanten. (Aus Assarsson und Moller (2000, p. 5))

Beim Maskieren wird bei der Traversierung über den Szenegraph bei jedem Rekursionsschritt eine Bitmaske an die Kindknoten übergeben. Diese Bitmaske enthält so viele Bits wie das Frustum Ebenen hat. Jedes Bit steht für eine Ebene und gibt an, ob der Elternknoten innerhalb oder außerhalb jener Ebene liegt. Dies ist von Vorteil, da weitere Tests gegen eine bestimmte Ebene nicht mehr notwendig sind, sofern ein Elternknoten innerhalb jener Ebene liegt, da dann

auch seine gesamten Kindknoten innerhalb dieser Ebene liegen. Somit können bei tieferen Rekursionsschritten für diese Ebene weitere Tests weggelassen werden (vgl. ebd.).
Die beschriebenen Optimierungen für den Schnittpunkttest von Assarson u. a. haben im Durchschnitt eine 1,4-fache Verbesserung der Performance erreicht, teilweise wurde die Performance um das Dreifache beschleunigt (vgl. ebd.).

3.3 Verwendung von Octrees als Optimierungsansatz für Schnittpunktbestimmung

Durch den Einsatz von Octrees als hierarchische Struktur sowohl für Objekte als auch für die gesamte Szene ist es möglich, Performanceverbesserungen zu erzielen und Objekte, die nur mit bestimmten Teilen im View Frustum liegen, auch nur teilweise zu rendern.

3.3.1 Octree der Szene

Der Octree für die Szene umfasst diese komplett mit allen Objekten unabhängig davon, ob diese im View Frustum liegen oder nicht (siehe Abbildung 3.9). Er bringt insofern Performancegewinn, als dass anfangs diejenigen Octree-Würfel, die das View Frustum schneiden oder innerhalb liegen, vom gesamten Octree extrahiert werden. Dies führt dazu, dass nur noch die Hüllkörper der Dreiecksnetze auf Schnittpunkt mit dem View Frustum getestet werden müssen, die auch die extrahierten Octree-Würfel der Szene berühren. Liegt eine Berührung zwischen Hüllkörper des Dreiecksnetzes und Octree-Würfel der Szene vor, werden genauere Schnittpunkttests zwischen Hüllkörper und View Frustum durchgeführt, um zu bestimmen, ob die Objekte gerendert werden müssen oder nicht. Ist der Test auf Berührung zwischen View Frustum und Octree-Würfel des Szene-Octrees hingegen negativ, fallen weitere Schnittpunkttests zwischen Hüllbox des Dreiecksnetzes und View Frustum weg, was einen Performancegewinn mit sich bringt.

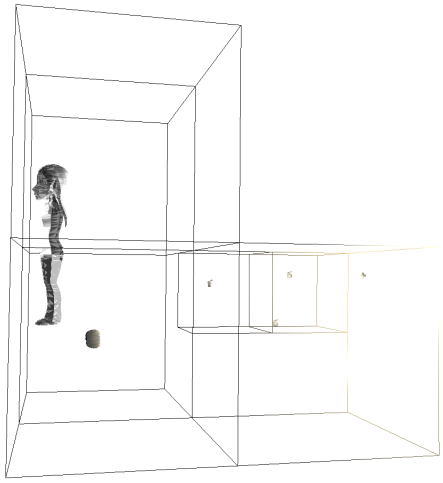

Abbildung 3.9: Octree für eine beispielhafte Szene mit sechs Dreiecksnetzen. (Ausgabe des Programms)

Bei der Anwendung des Octrees für die gesamte Szene ist zu beachten, dass die Rekursionstiefe nicht zu niedrig und nicht zu hoch gewählt werden darf. Eine zu niedrige Rekursionstiefe führt dazu, dass die Octree-Würfel zu groß werden, sodass zu viele Schnittpunkttests zwischen extrahierten Octree-Würfeln und Hüllboxen von Dreiecksnetzen positiv ausfallen und deshalb unnötig viele genauere Schnittpunkttests zwischen View Frustum und Hüllboxen durchgeführt werden.
Eine zu hohe Rekursionstiefe des Octrees wiederum führt zu sehr feiner Granularität und somit einer höhren Anzahl an Knoten des Octrees, was die Anzahl der Tests auf Überschneidung von Octree-Würfeln und Hüllboxen erhöht. Dies kann zu einer Relativierung des Performancegewinns führen, der durch die Anwendung des Octrees für die Szene erreicht werden soll.
Messungen der Framerate, die mit Hilfe der Implementierung zu dieser Arbeit durchgeführt wurden, haben gezeigt, dass die ausschließliche Verwendung der Blattknoten für die Überschneidungstests zwischen Octree-Würfeln und Hüllboxen performanter ist als die Verwendung von Octree-Würfeln auf niedrigeren Rekursionsebenen. In der Implementierung zu dieser Arbeit werden daher nur Blattknoten des Octrees verwendet. Deshalb sollten die Blattkriterien für den Octree der Szene nicht zu klein gewählt werden, um zu große Octree-Würfel zu vermeiden, und nicht zu hoch, um eine zu feine Granularität des Octrees zu verhindern.

3.3.2 Octrees für Dreiecksnetze

Octrees für Dreiecksnetze können dafür eingesetzt werden, diese nur teilweise zu rendern, sofern sie nicht komplett im Sichtfeld liegen. Dafür wird beim Schnittpunkttest zwischen View Frustum und Octree-Würfel auf die tiefstnotwendige Rekursionstiefe gegangen, um festzulegen, ob ein Octree-Würfel komplett innerhalb oder außerhalb des View Frustums liegt. Weist der Test darauf hin, dass das Objekt geschnitten wird, werden rekursiv alle Kindknoten des Octree-Würfels getestet. Wird bei dieser Rekursion die höchste Rekursionsstufe erreicht und der betreffende Octree-Würfel wird geschnitten, kann das Objekt gerendert werden oder nicht. Es empfiehlt sich, in einem solchen Fall ds Objekt zu rendern, um Rendering-Fehler zu vermeiden.

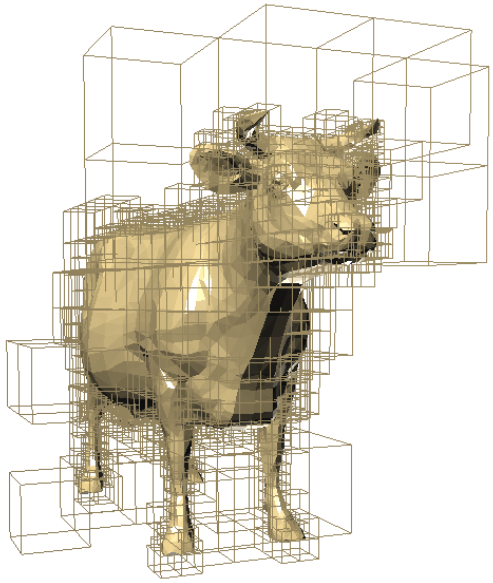

Abbildung 3.10: Octree für das Dreiecksnetz einer Kuh. Die Octree-Würfel auf der tiefsten Rekursionsstufe referenzieren die Dreiecke des Meshes, die sie beinhalten. (Ausgabe des Programms)

Es besteht die Möglichkeit, Meshes, die geschnitten werden, nur teilweise zu rendern. Dafür werden rekursiv alle Kindknoten auf Schnittpunkte getestet. Anschließend werden von den Octree-Würfeln der tiefsten Rekursionsstufe, die im Sichtbarkeitsbereich liegen, die beinhalteten Triangles extrahiert und auf sichtbar beziehungsweise unsichtbar getoggelt.
Bei solchen Meshes, welche vom View Frustum geschnitten werden, gilt es abzuwägen, ob der Gewinn, der dabei entsteht, wenn das Dreiecksnetz nur teilweise gerendert wird, den

Mehraufwand relativiert. Dafür müssen rekursiv die Kindknoten des Octrees auf Schnittpunkte getestet werden. Bei der zugehörigen Implementierung hat sich gezeigt, dass das Umschalten der Sichtbarkeit einzelner Dreiecke mit Hilfe von Rekursion deutlich langsamer ist als nur teilweise sichtbare Objekte komplett zu rendern. Deshalb wird die Möglichkeit, Objekte nur teilweise sichtbar zu machen, nicht genutzt.

4 Implementierung

In diesem Kapitel wird die prototypische Implementierung des Konzepts „View Frustum Culling“ vorgestellt. Sie basiert auf der hierarchischen Unterteilung der gesamten Szene mit Hilfe eines Octrees. Anschließend wird die Sichtbarkeit einzelner Dreiecksnetze an- oder ausgeschaltet abhängig von ihrer Position bezüglich des View Frustums (innerhalb oder außerhalb). Dabei wird die hierarchische Unterteilung der Szene mit dem Ziel verwendet, rechenaufwändige Schnittpunkttests zwischen Dreiecksnetzen und View Frustum möglichst zu minimieren. Dieses Vorgehen soll die Framerate erhöhen, indem weniger Dreiecksnetze durch die Rendering Pipeline verarbeitet werden.
Im Folgenden werde ich die Implementierung genauer beschreiben und erläutern.

4.1 Hierarchische Unterteilung der Szene

Wie in Kapitel 3.3.1 bereits angesprochen, besteht der erste signifikante Schritt der Anwendung darin, die komplette Szene inklusive aller Dreiecksnetze hierarchisch zu unterteilen. Dabei wird auf das Modell des Octrees zurückgegriffen.

4.1.1 Aufbau des Octrees

Das Vorgehen für die Unterteilung gliedert sich in zwei Schritte: zuerst muss eine Hüllbox um die gesamte Szene gebildet werden. Dafür werden für jede Achse Minima („lower left“, der vordere, linke, untere Eckpunkt) sowie Maxima („upper right“, der hintere, rechte, obere Eckpunkt) der Hüllboxen aller Dreiecksnetze gespeichert. Es ist anzumerken, dass die Hüllboxen - unabhängig von durchgeführten Transformationen (auch Rotationen) auf sie - immer achsenparallel sind.
Nach der Berechnung der Hüllbox für die Szene erfolgt ihre hierarchische Unterteilung. Da der Octree auf kubischen Formen basiert, wird zunächst mit Hilfe der Hüllbox ein Würfel gebildet, der die gesamte Szene mit allen Dreiecksnetzen beinhaltet. Hierfür wird auf jede Koordinate des Mittelpunkts der Hüllbox eine Subtraktion der Hälfte der maximalen Kantenlänge der

Hüllbox ausgeführt, sodass ein neuer unten-links-Eckpunkt entsteht. Oben-rechts ergibt sich nun aus Addition der maximalen Kantenlänge der Hüllbox auf unten-links. Damit ist der Wurzelknoten erzeugt und die erste Stufe des Octrees erreicht. Abhängig der Blattkriterien für den Octree (maximale Rekursionstiefe sowie maximale Anzahl an beinhalteten Elementen, hier: Dreiecksnetze) müssen nun weitere Stufen gebildet beziehungsweise die Szene weiter unterteilt werden. Ist eine weitere Rekursionsstufe notwendig, müssen die Dreiecksnetze den neu entstehenden Octree-Würfeln erneut zugeordnet werden, da nur die Blattknoten des Octrees Dreiecksnetze referenzieren. Diese Unterteilung beziehungsweise die Zuordnung der Dreiecksnetze zu den Kindwürfeln stellte bei der Umsetzung die erste relevante Schwierigkeit dar.

4.1.2 Zuordnung der Dreiecksnetze zu Octree-Würfeln

Um die Dreiecksnetze den Octree-Würfeln neu zuzuordnen, wird nach einem Ausschlussverfahren vorgegangen. Das Verfahren prüft, ob die Hüllboxen der Dreiecksnetze „neben“ dem Octree-Würfel liegen oder ihn schneiden. Dabei werden die entgegensetzten Eckpunkte von Octree- Würfel und Hüllbox gegeneinander getestet. Wenn beispielsweise die X-Koordinate von „lower left“ der Hüllbox größer als die X-Koordinate von „upper right“ des Octree-Würfels ist, lässt sich sagen, dass die Hüllbox „rechts“ vom Octree-Würfel liegt und ihn somit nicht schneiden kann.
Trifft hingegen keine der Bedingungen zu, liegt eine Überschneidung zwischen Hüllbox und Octree-Würfel vor, sodass das Dreiecksnetz dem Würfel zugeordnet wird. Dies hat (zum Beispiel bei großen Dreiecksnetzen) zur Folge, dass ein Netz möglicherweise mehreren Octree-Würfeln zugeordnet wird.

4.1.3 Wahl der Rekursionstiefe des Octrees

Wie bereits erwähnt, ist bei der Wahl der Blattkriterien des Octrees zu beachten, dass sowohl große als auch kleine Rekursionstiefen Vor- und Nachteile mit sich bringen können. Eine hohe maximale Rekursionstiefe kann hohe Granularität des Octrees mit sich bringen, sodass kleinere Octree-Würfel mit weniger leerem Raum entstehen. Dafür müssen die Überschneidungstests zwischen Hüllkörpern der Dreiecksnetze und Octree-Würfeln aufgrund der höheren Anzahl der Octree-Würfel gegebenenfalls öfters ausgeführt werden. Eine niedrige Rekursionstiefe hingegen hat große Octree-Würfel zur Folge, sodass mehr Überschneidungstests zwischen Octree-Würfeln und Dreiecks-Hüllkörpern positiv ausfallen, was mehr genauere Schnittpunkttests nach sich zieht.

4.2 Aufbau des View Frustums

Der Aufbau des Sichtbarkeitsbereichs oder „View Frustum“ ist der entscheidende Schritt der Anwendung. Der aus den Kameraparametern berechnete Sichtbarkeitsbereich muss genau dem tatsächlichen, in der Animation sichtbaren Bereich entsprechen, um ein sinnvolles An- und Ausschalten der Sichtbarkeit der Dreiecksnetze zu ermöglichen.

4.2.1 Berechnung der Eckpunkte

Das View Frustum wird durch sechs Ebenen eindeutig begrenzt. Die Eckpunkte des Frustums wiederum ergeben sich aus den Schnittpunkten dieser Ebenen. Um das View Frutum aufzubauen, müssen zuerst die Eckpunkte bestimmt werden. Dafür werden zunächst die Mittelpunkte von naher (n) und ferner (f) Ebene berechnet:

$$\vec{center}_{n,f} = \vec{eye} + ((\vec{ref} - \vec{eye}) * distance_{n,f}) \tag{4.1}$$

Die Parameter eye sowie ref und $distance$ in Gleichung 4.1 sind Kameraeigenschaften und über das Kamera-Objekt abrufbar. Der Differenzwert von Referenzpunkt und Auge der Kamera gibt die Blickrichtung der Kamera an, welche multipliziert mit den Distanzen zum Near- und Far-Clipping und anschließend zum Augpunkt addiert die Mittelpunkte von naher und ferner Ebene liefert.
Anschließend wird die Höhe des View Frustums berechnet (aus Lighthouse3d.com (a)):

$$height_{n,f} = 2 * \tan\left(\frac{openingAngle}{2}\right) * distance_{n,f} \tag{4.2}$$

Der Tangens in Gleichung 4.2 gibt das Seitenverhältnis zwischen Höhe und Distanz zurück, welches benötigt wird, um den halben Öffnungswinkel zu erlangen. Die Relationen zwischen Öffnungswinkel sowie Höhe und Distanz werden in Abbildung 4.1 verdeutlicht.

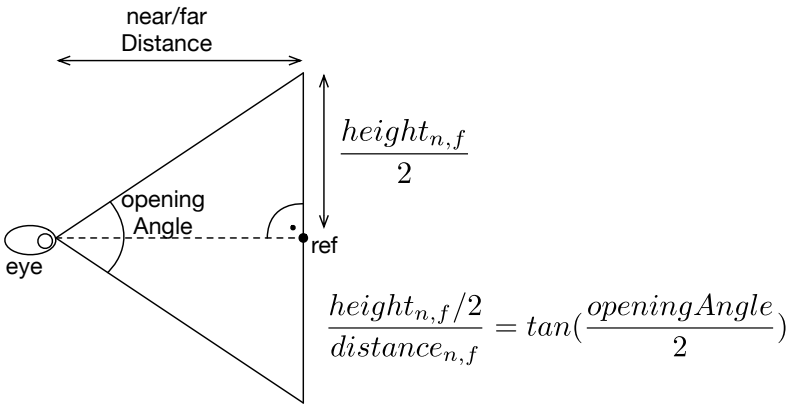

Abbildung 4.1: Das Verhältnis zwischen der halben Höhe der nahen beziehungsweise fernen Ebene zur nahen beziehungsweise fernen Distanz entspricht dem Tangens des halben Öffnungswinkels. (Eigene Skizze)

Dieser Wert wird zunächst mit 2 multipliziert, da sich die Höhe in positive und negative Richtung erstreckt, und dann mit naher beziehungsweise ferner Distanz multipliziert.
Mit Hilfe der Höhe der nahen und fernen Ebene sowie eines Parameters, der das Größenverhältnis zwischen Höhe und Breite angibt, lässt sich nun die Breite berechnen. Dieses Verhältnis ist variabel. In der Implementierung beträgt das Verhältnis $1:1$, die Ebenen haben also eine quadratische Form.
Nachdem die Mittelpunkte der nahen und fernen Ebene sowie ihre Höhe und Breite berechnet wurden, lassen sich die je vier Eckpunkte dieser Ebenen beziehungsweise die acht eingrenzenden Eckpunkte des View Frustums bestimmen. Hierfür werden zu den Mittelpunkten der nahen und fernen Ebene jeweils die halbe Höhe sowie Breite addiert oder subtrahiert, abhängig davon, welcher Eckpunkt berechnet werden soll (links, rechts, oben, unten). So wird der Eckpunkt der fernen Ebene, der an der oberen rechten Ecke liegt („far top right“, ftr), beispielsweise mit dieser Formel berechnet:

$$\vec{corner}_{ftr} = \vec{center}_f + \vec{up} * (\frac{height_f}{2}) + \vec{right} * (\frac{width_f}{2}) \tag{4.3}$$

Der Up-Vektor in Gleichung 4.3 ist über das Kamera-Objekt abrufbar, der Right-Vektor ergibt sich aus dem Kreuzprodukt des up-Vektors sowie der invertierten Blickrichtung der Kamera (siehe Gleichung 4.4):

$$\vec{right} = \vec{up} \times (\vec{eye} - \vec{ref}) \tag{4.4}$$

Mit Hilfe der acht Eckpunkte lassen sich jetzt die eingrenzenden Ebenen bestimmen.

4.2.2 Berechnung der Ebenen

Die Ebenen werden jeweils über ihre Normale und einen Punkt definiert. Somit lassen sich alle Ebenen über die Eckpunkte und den Augpunkt der Kamera generieren.
Da die nahe und ferne Ebene parallel zueinander und senkrecht zum Augpunkt sind, sind für diese Ebenen - im Gegensatz zu der linken, rechten, oberen und unteren Ebene - keine weiteren Berechnungen für die Normalen notwendig. Die Normale dieser Ebenen ist nämlich die (invertierte) Blickrichtung der Kamera, sodass sie sich wie folgt berechnen lassen:

$$\vec{normal}_n = \frac{(\vec{ref} - \vec{eye})}{||(\vec{ref} - \vec{eye})||} \tag{4.5}$$

$$\vec{normal}_f = \frac{(\vec{eye} - \vec{ref})}{||(\vec{eye} - \vec{ref})||} \tag{4.6}$$

Für die linke, rechte, obere und untere Ebene muss die Normale mit Hilfe zweier Eckpunkte sowie einem Kreuzprodukt berechnet werden (zur Berechnung der Normale einer Ebene mit Hilfe von drei Punkten siehe Gleichung 2.10). Dabei muss darauf geachtet werden, dass Vektor $\vec{w}$ (siehe Gleichung 2.10) immer $\vec{eye}$, also der Augpunkt der Kamera ist. Somit kann die Normale der linken Ebene beispielsweise durch die Punkte „far top left“ ($\vec{ftl}$), „far bottom left“ ($\vec{fbl}$) sowie $\vec{eye}$ berechnet werden.
Des Weiteren müssen die Normalen in das View Frustum zeigen. Da das Kreuzprodukt antikommutativ ist, müssen die Faktoren teilweise vertauscht werden (siehe Pseudocode 1).

Algorithm 1 Normalenberechnung

```
1: function NORMALCALCULATION(firstPlaneCorner, secondPlaneCorner, aCrossB)
2:     a, b, c
3:     a ← (firstPlaneCorner - eye)
4:     b ← (secondPlaneCorner - eye)
5:     if !aCrossB then
6:         c ← b × a
7:     else
8:         c ← a × b
9:     c ← c / ||c||
   return c
```

Sobald die eingrenzenden Eckpunkte und Ebenen erzeugt sind, ist das View Frustum eindeutig definiert, sodass nun mit der Positionserkennung der Dreiecksnetze begonnen werden kann.

4.3 Erkennung der Position von Dreiecksnetzen

Die Erkennung der Position von Dreiecksnetzen und daraus folgendes Umschalten der Sichtbarkeit dieser ist der wesentliche Schritt des View Frustum Cullings, um durch den Einsatz der Implementierung die Framerate zu erhöhen. Er unterteilt sich in zwei Teilschritte, um den Performancegewinn zu maximieren. Dabei hat der erste Teilschritt zum Ziel, Dreiecksnetze, die gar nicht in der Nähe des View Frustums liegen, schon auszusortieren, um aufwändigere, zeitintensivere Schnittpunkttests zu vermeiden. Dies geschieht mit Hilfe des Octrees der ganzen Szene.
Der zweite Teilschritt bezieht sich dann auf die tatsächliche Positionserkennung der Dreiecksnetze. Im Folgenden werde ich die beiden Teilschritte genauer erläutern.

4.3.1 Octree der gesamten Szene

Der Octree für die gesamte Szene wird dafür eingesetzt, diejenigen Octree-Würfel, die innerhalb des Sichtbarkeitsbereichs liegen oder ihn schneiden, vom gesamten Octree zu extrahieren (siehe Abbildung 4.2).

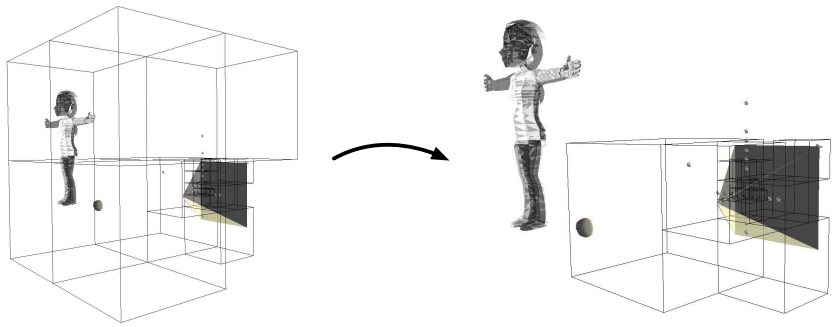

Abbildung 4.2: Der Octree der gesamten Szene und das View Frustum vor (links) und nach Extraktion der Octree-Würfel, die innerhalb des View Frustums liegen oder es schneiden. (Ausgabe des Programms)

Hierfür wird dasselbe Prinzip angewendet wie bei der Schnittpunkterkennung zwischen View Frustum und Hüllboxen der Dreiecksnetze, welches später noch in Abschnitt 4.1 genauer erklärt wird.
Die Extraktion (teilweise) sichtbarer Octree-Würfel bringt den Vorteil mit sich, dass vor den Schnittpunkttests zwischen View Frustum und Hüllbox des Dreiecksnetzes geprüft wird, ob die Hüllbox einen solchen vom Octree der gesamten Szene extrahierten Octree-Würfel be-

rührt, um eindeutig außerhalb liegende Dreiecksnetze schon im Vorfeld auszusortieren. Liegt keine Berührung zwischen der Hüllbox und den extrahierten Octree-Würfeln vor, ist davon auszugehen, dass der Inhalt der Hüllbox nicht im Sichtbarkeitsbereich liegt, sodass weitere Schnittpunkttests überflüssig sind und das Dreiecksnetz unsichtbar geschaltet wird. Gibt es hingegen eine Überschneidung zwischen extrahierten Octree-Würfeln und Hüllbox, liegt das Dreiecksnetz möglicherweise im Sichtbarkeitsbereich, sodass der Schnittpunkttest zwischen View Frustum und Hüllbox durchgeführt wird, um genau festzustellen, ob das Dreiecksnetz innerhalb des View Frustums liegt und sichtbar geschaltet werden muss oder nicht.

4.3.2 Positionserkennung einzelner Dreiecksnetze

Nachdem festgestellt wurde, dass zwischen Octree-Würfeln und Hüllbox des Dreiecksnetzes eine Überschneidung besteht, lässt sich sagen, dass das Dreiecksnetz potenziell innerhalb des View Frustums liegt. Daher ist nun der genauere Schnittpunkttest notwendig.
Dieser Schnittpunkttest stellte eine weitere nennenswerte Schwierigkeit bei der Umsetzung des View Frustum Cullings dar. Die naheliegende und triviale Lösung, alle Eckpunkte einer Hüllbox gegen alle Ebenen des View Frustums zu testen und sie nur dann als innerhalb zu klassifizieren, sofern auch alle Eckpunkte innerhalb liegen, arbeitet zum Beispiel bei Hüllboxen, die größer als das View Frustum sind, nicht korrekt. Eine weitere Überlegung bestand darin, eine Hüllbox als innerhalb liegend zu klassifizieren, sobald mindestens ein Eckpunkt innerhalb des View Frustums liegt. Dieser Algorithmus hingegen arbeitet nicht richtig bei dem Sonderfall, der in Abbildung 4.3 zu sehen ist, da hier zwar kein Eckpunkt der Hüllbox innerhalb des View Frustums liegt, das Dreiecksnetz aber trotzdem geschnitten wird.

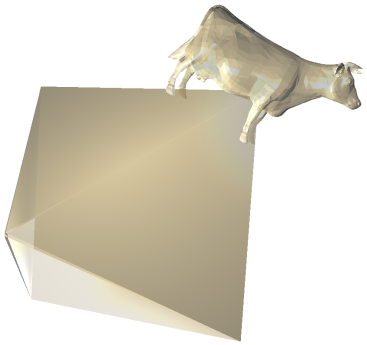

Abbildung 4.3: Ein Dreiecksnetz, welches vom View Frustum geschnitten wird. (Ausgabe des Programms)

Der schließlich implementierte Ansatz mit akzeptabler Funktionalität arbeitet nach dem Prinzip, eine Hüllbox genau dann als außerhalb liegend zu klassifizieren, sobald alle ihre Eckpunkte außerhalb der selben Ebene des View Frustums liegen (vgl. Lighthouse3d.com (b)). Alle anderen Hüllboxen werden als innenliegend klassifiziert. Dieser Ansatz gibt auch für den Sonderfall, der in Abbildung 4.3 zu sehen ist, das korrekte Ergebnis zurück.

4.3.3 Sonderfälle

Für Hüllboxen, die komplett außerhalb, aber nicht außerhalb von nur einer Ebene positioniert sind, sodass nicht alle Eckpunkte außerhalb der selben Ebene liegen, gibt dieser Algorithmus innenliegend zurück. Somit wird ein Dreiecksnetz mit einer solchen Hüllbox fälschlicherweise gerendert. Dieser Fall ist ähnlich zu dem in Abbildung 4.3, wobei das Dreiecksnetz dann etwas höher ist und das View Frustum nicht mehr berührt. Wie in Abschnitt 3.2 bereits erwähnt, ist diese Tatsache allerdings akzeptabel, da bei einem solchen Sonderfall zwar mehr gerendert wird, dafür aber weniger aufwändige Schnittpunkttests durchgeführt werden. Dies hat zur Folge, dass der Mehraufwand, welcher beim Rendering entsteht, durch den Verzicht auf genauere Schnittpunkttests ausgeglichen wird. Assarson und Möller konnten durch Verzicht auf weitere Schnittpunkttest und somit mögliches Rendern von Dreiecksnetzen außerhalb des View Frustums ebenfalls keine signifikanten Performanceeinbußen feststellen (vgl. Assarsson und Moller (2000)). Des Weiteren wird durch den ersten Teilschritt, nämlich den Test auf Überschneidung zwischen Hüllbox und extrahierten Octree-Würfeln, die Wahrscheinlichkeit des Auftretens eines solchen Sonderfalles bereits reduziert.

4.4 Konzept des View Frustum Cullings im Test- und Live-Modus

Nachdem eine Testszene aufgebaut wurde, lässt sich das implementierte View Frustum Culling in zwei Modi anwenden: Test-Modus (siehe Abschnitt 4.4.1) und Live-Modus (siehe Abschnitt 4.4.2).

4.4.1 Test-Modus

Im Test-Modus wird das View Frustum einmalig berechnet und ist dann statisch und unabhängig von weiteren Kamerabewegungen. Danach wird der Octree für die Szene aufgebaut und die (teilweise) sichtbaren Octree-Würfel werden einmalig extrahiert. Anschließend werden die dem Szenegraph hinzugefügten Dreiecksnetze abhängig von ihrer Position bezüglich des

View Frustums sichtbar oder unsichtbar geschaltet. Dies geschieht zunächst mit Hilfe des Überschneidungstests zwischen den extrahierten Octree-Würfeln der Szene und anschließend - sofern eben genannter Test positiv ist - mittels genauerem Schnittpunkttest zwischen Hüllboxen der Dreiecksnetze und View Frustum. Nun ist die Anwendung durchgelaufen, sodass man das View Frustum und die (un-)sichtbar geschalteten Dreiecksnetze aus allen Perspektiven betrachten kann. Abbildung 4.4 zeigt die gesamte Testszene und ein View Frustum, bevor das Culling beziehungsweise das Sichtbar-/Unsichtbar-Schalten der Dreiecksnetze durchgeführt wird. In Abbildung 4.5 hingegen wurde das Culling angewendet, sodass nur die innerhalb des View Frustums liegenden Dreiecksnetze sichtbar geschaltet sind.
Der Test-Modus wird verwendet, um zu verifizieren, dass das Culling korrekt arbeitet und die richtigen Dreiecksnetze auf sichtbar beziehungsweise unsichtbar schaltet.

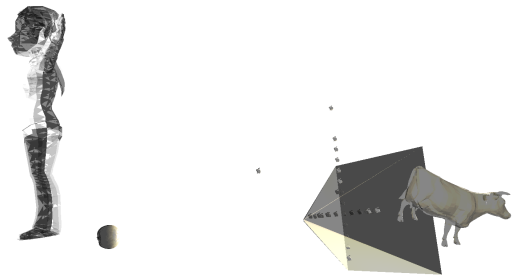

Abbildung 4.4: Die gesamte Szene mit einem View Frustum

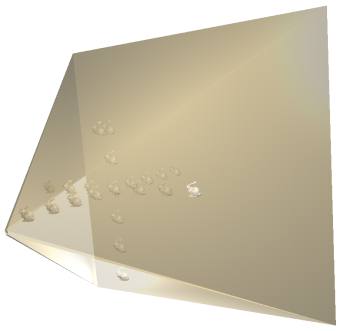

Abbildung 4.5: Ein View Frustum mit innerhalb liegenden Dreiecksnetzen nach Anwendung des Cullings

4.4.2 Live-Modus

Ähnlich wie im Test-Modus wird im Live-Modus zu Anfang das View Frustum berechnet sowie der Octree für die Szene gebildet. Das View Frustum entspricht dabei genau dem tatsächlich in der Animation sichtbaren Bereich. Danach findet die Extraktion der (teilweise) im Sichtbarkeitsbereich liegenden Octree-Würfel statt. Nun werden - wie im Test-Modus - die Dreiecksnetze (un-)sichtbar geschaltet.
Im Gegensatz zum Test-Modus allerdings reagiert das View Frustum Culling im Live-Modus auf Veränderungen der Kameraposition. Sobald die Kamera bewegt wird, wird der Sichtbar-

keitsbereich neu berechnet, was zur Folge hat, dass die Octree-Würfel der Szene, die den Sichtbarkeitsbereich berühren, neu extrahiert werden müssen. Anschließend werden die Dreiecke erneut (un-)sichtbar geschaltet. Dabei kann es passieren, dass im neuen Frame auf Grund der Kamerabewegung andere Dreiecksnetze sichtbar sind.
Der Live-Modus implementiert das Observer-Pattern. Das View Frustum Culling gleicht somit einem Beobachter der Kamera, welcher auf Kameraveränderung mit einer neuen Berechnung des View Frustums, erneuter Extraktion sichtbarer Octree-Würfel sowie darauf folgendem Toggeln der Sichtbarkeit der Dreiecksnetze reagiert.

5 Evaluation

Das Ziel der Bachelor-Arbeit war die Evaluierung der Sichtbarkeitsberechnung und die Anwendung derer auf die Praxis anhand einer Implementierung des View Frustum Cullings für das Java-Computergrafik-Framework der Computergrafik-Gruppe der HAW. Das erhoffte Ergebnis der Implementierung war ein Performancegewinn, der sich in einer erhöhten Framerate widerspiegelt. Dieser sollte dadurch erreicht werden, dass durch das View Frustum Culling Dreiecksnetze, die außerhalb des Sichtbarkeitsbereichs liegen, nicht über die Rendering Pipeline verarbeitet werden. Das Resultat entspricht in gewissem Maße den Erwartungen, da bei steigender Anzahl an Dreiecksnetzen außerhalb des Sichtbarkeitsbereichs mit Anwendung des View Frustum Cullings die Framerate stagniert, während sie ohne das View Frustum Culling sinkt. Auch ist ein leichter Performancegewinn festzustellen bei Szenen mit komplexen, außerhalb des Sichtbarkeitsbereichs liegenden Dreiecksnetzen.
Im Folgenden werde ich die Testszenarien sowie die Messergebnisse beschreiben und erläutern.

5.1 Testumgebung

Die Tests wurden unter Windows 8.1 (64 Bit) auf einer Intel(R) Core(TM) i7-3537U CPU mit 2.50 GHz durchgeführt. Die Grafikkarte ist eine Intel(R) HD Graphics 4000. Die verwendete Java-Runtime (JRE) ist 1.8.0 (Update 66), es wurden keine weiteren Parameter an die JVM übergeben.

5.2 Vorstellung der Testszenen

Es wurde mit diversen Testszenen gemessen. Eine dieser Testszenen besteht ausschließlich aus Hasen-Dreiecksnetzen mit je 4968 Dreiecken und 2503 Vertices. Die Dreiecksnetze sind entlang der Achsen positioniert (siehe Abbildung 5.1). Die Anzahl der Dreiecksnetze wurde bei den Tests variiert. Diese Testszene hatte zum Ziel, das Verhalten einer Anwendung mit View Frustum Culling bei steigender Anzahl an Dreiecksnetzen außerhalb des Sichtbarkeitsbereichs wiederzugeben.

Abbildung 5.1: Testszene mit Hasen-Dreiecksnetzen, rotiert und zentriert zur besseren Ansicht. Die Anzahl der Hasen ist variabel. (Ausgabe des Programms)

Die zweite Testszene besteht aus acht Dreiecksnetzen unterschiedlicher Größe und Komplexität, von denen einige außerhalb des (initialen) Sichtbarkeitsbereichs positioniert sind (siehe Abbildung 5.2). Zum Aufbau der Dreiecksnetze siehe Tabelle 5.1. Diese Testszene sollte verdeutlichen, dass insbesondere bei großen, komplexen Dreiecksnetzen außerhalb des Sichtbarkeitsbereichs unter Verwendung des View Frustum Cullings eine Performanceverbesserung festzustellen ist.

Dreiecksnetz	Anzahl	Dreiecke	Vertices
Kuh	1	5804	2903
Hase	4	4968	2503
Drache	1	47794	22998
Person	1	4184	2095
Kürbis	1	10000	5002

Tabelle 5.1: Verschiedene Dreiecksnetze mit Dreiecks- und Vertice-Anzahl

Abbildung 5.2: Gemischte Testszene; rotiert und zentriert zur besseren Ansicht. (Ausgabe des Programms)

Die dritte Testszene zeigt eine Stadt mit 100 Häusern, von denen ebenfalls einige innerhalb sowie außerhalb des initialen Sichtbarkeitsbereichs liegen (siehe 5.3). Der Aufbau der Szene ist zufällig, sodass die Anzahl der Dreiecke je Haus zwischen sieben und 184 und der Vertices zwischen 118 und 2026 schwankt. Diese Testszene sollte das Verhalten des View Frustum Cullings bei Anwendung auf eine große Szene simulieren.

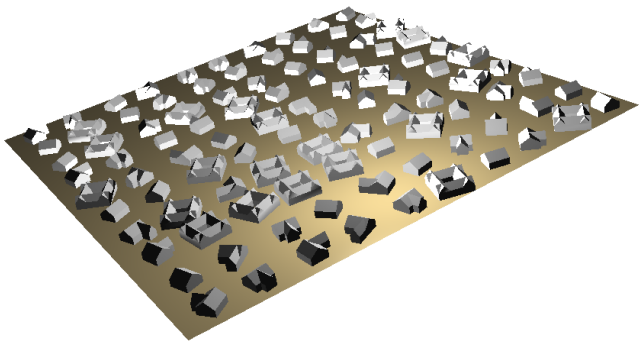

Abbildung 5.3: Testszene der Stadt; rotiert und zentriert zur besseren Ansicht. (Ausgabe des Programms)

5.3 Messergebnisse

Die Messungen wurden jeweils fünf Mal über fünf Sekunden sowohl mit als auch ohne View Frustum Culling sowie mit und ohne Kamerabewegung durchgeführt. Dabei wurde die erreichte Framerate gemessen und anschließend der Mittelwert berechnet.
Bei den Messungen der gemischten und der Stadtszene haben sich keine signifikanten Unterschiede in der Framerate abhängig des View Frustum Cullings gezeigt. Auch scheint es keine

deutlichen Abhängigkeiten von der Kamerabewegung zu geben. So liegt die durchschnittliche Framerate der gemischten Szene ohne Kamerabewegung und ohne View Frustum Culling bei 61,78 FPS (Frames Per Second) und mit View Frustum Culling bei 63,22 FPS (siehe Tabelle 5.3), was einer leichten Steigerung entspricht. Dies liegt an dem geringeren Render-Aufwand aufgrund außerhalb des Sichtbarkeitsbereichs liegender Dreiecksnetze (siehe Abbildung 5.4 sowie Abbildung 5.5), da diese bei Anwendung des View Frustum Cullings nicht gerendert werden. Bei Kamerabewegung ergibt sich für diese Testszene ohne View Frustum Culling eine Framerate von 58,92 FPS und mit View Frustum Culling 66,21 FPS (siehe Tabelle 5.2), was ebenfalls eine Steigerung ist. Die Standardabweichung ist bei sich bewegender Kamera und eingeschaltetem View Frustum Culling allerdings höher (steigt von 8,05 auf 15,91, siehe Tabelle 5.2). Dieser deutliche Anstieg der Standardabweichung lässt sich auf die Berechnungen zurückführen, die von der Kamerabewegung ausgelöst werden und beispielsweise die Java-Garbage-Collection und somit größere Schwankungen der Framerate zur Folge haben.

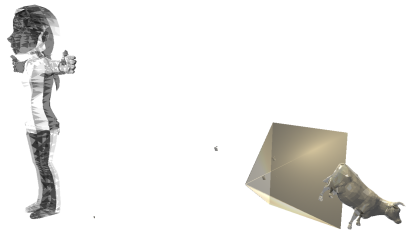

Abbildung 5.4: Gemischte Testszene vor Culling, zu sehen sind alle Dreiecksnetze der Szene sowie das Sichtbarkeitsvolumen. (Ausgabe des Programms)

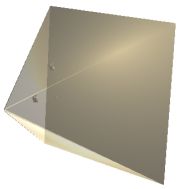

Abbildung 5.5: Gemischte Testszene nach Culling, zu sehen sind innerhalb des Sichtbarkeitsvolumens liegende oder von diesem geschnittene Dreiecksnetze der Szene sowie das Sichtbarkeitsvolumen. (Ausgabe des Programms)

	ohne VFC	mit VFC
FPS	58,92	66,21
Varianz	65,25	260,78
Standardabweichung	8,06	15,91

Tabelle 5.2: Gemischte Szene mit Bewegung der Kamera.

	ohne VFC	mit VFC
FPS	61,78	63,22
Varianz	70,51	46,41
Standardabweichung	8,39	6,72

Tabelle 5.3: Gemischte Szene ohne Bewegung der Kamera.

Die Messungen der Stadtszene haben folgende Ergebnisse gebracht:

	ohne VFC	mit VFC
FPS	64,53	65,11
Varianz	86,8	133,61
Standardabweichung	9,25	11,21

Tabelle 5.4: Stadtszene mit Bewegung der Kamera.

	ohne VFC	mit VFC
FPS	64,91	64,03
Varianz	121,5	108,3
Standardabweichung	10,87	9,88

Tabelle 5.5: Stadtszene ohne Bewegung der Kamera.

Erstaunlicherweise hat die Messung der Stadtszene ohne Kamerabewegung und mit View Frustum Culling eine leicht niedrigere Framerate ergeben als ohne View Frustum Culling (sinkt von 64,91 FPS auf 64,03 FPS, siehe Tabelle 5.5). Dies kann zum einen daran liegen, dass die zu rendernde Szene keine komplexen Dreiecksnetze beinhaltet, sodass bei Einschalten des View Frustum Cullings kein großer Performancegewinn durch Unsichtbarschalten solcher Dreiecksnetze passiert, obwohl von der gesamten Szene relativ wenig den Sichtbarkeitsbereich berührt (siehe Abbildung 5.6 sowie 5.7). Näher liegt allerdings, dass Zeit abhängige Messungen mit Java aufgrund von unsteuerbaren Teilprozessen wie zum Beispiel der Garbage Collection schwer unter immer gleichen Voraussetzungen zu wiederholen sind, sodass in den Ergebnissen auch Messungenauigkeiten enthalten sein könnten. Die Messung der Stadtszene mit Bewegung hingegen hat das erhoffte Ergebnis erbacht: Mit View Frustum Culling ist die Framerate leicht höher als ohne (steigt von 64,53 FPS auf 65,11 FPS, siehe Tabelle 5.4), was damit zusammenhängen könnte, dass nur ein relativ kleiner Teil der Szene den Sichtbarkeitsbereich berührt, sodass unter Verwendung des View Frustum Cullings ein signifikanter Renderingaufwand wegfällt.

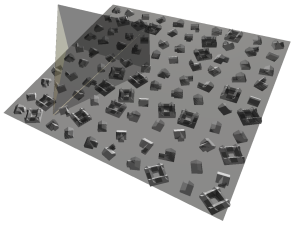

Abbildung 5.6: Stadtszene vor Culling, zu sehen sind alle Dreiecksnetze der Szene sowie das Sichtbarkeitsvolumen. (Ausgabe des Programms)

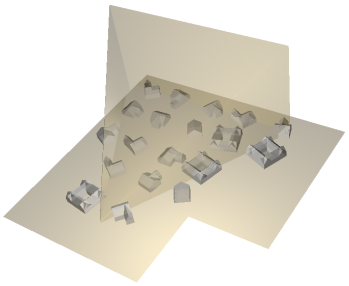

Abbildung 5.7: Stadtszene nach Culling, zu sehen sind innerhalb des Sichtbarkeitsvolumens liegende oder von diesem geschnittene Dreiecksnetze der Szene sowie das Sichtbarkeitsvolumen. (Ausgabe des Programms)

Bei den durchgeführten Messungen mit der ausschließlich aus Hasen-Dreiecksnetzen bestehenden Szene mit variabler Anzahl hat sich das erwartete Ergebnis gezeigt. Während bei den Tests ohne das View Frustum Culling die Framerate bei steigender Anzahl der Dreiecksnetze sinkt (siehe Tabellen 5.6 sowie 5.8 beziehungsweise Abbildungen 5.8 sowie 5.10), stagniert sie bei eingeschaltetem View Frustum Culling (siehe Tabellen 5.7 sowie 5.9 beziehungsweise Abbildungen 5.9 sowie 5.11). Dies liegt daran, dass ab einem bestimmten Punkt fast ausschließlich Dreiecksnetze außerhalb des Sichtbarkeitsbereichs hinzukommen. Während ohne das View Frustum Culling weiterhin alle Dreiecksnetze inklusive neu hinzugekommener gerendert werden, werden mit View Frustum Culling diese hinzukommenden Dreiecksnetze unsichtbar geschaltet und somit nicht gerendert.

	14 Hasen	28 Hasen	56 Hasen	112 Hasen	252 Hasen	546 Hasen
FPS	63,19	63,31	59,93	36,68	19,43	9,95
Varianz	45,13	56,32	14,81	4,26	0,49	0,03
Standardabweichung	6,62	7,45	3,84	2,05	0,65	0,17

Tabelle 5.6: Hasenszene mit steigender Anzahl Dreiecksnetzen ohne Bewegung der Kamera ohne View Frustum Culling.

	14 Hasen	28 Hasen	56 Hasen	112 Hasen	252 Hasen	546 Hasen
FPS	64,89	64,09	65,85	63,77	63,23	51,81
Varianz	152,49	76,4	84,89	72,47	84,52	90,57
Standardabweichung	12,23	8,48	9,2	8,51	9,19	9,44

Tabelle 5.7: Hasenszene mit steigender Anzahl Dreiecksnetzen ohne Kamerabewegung mit View Frustum Culling.

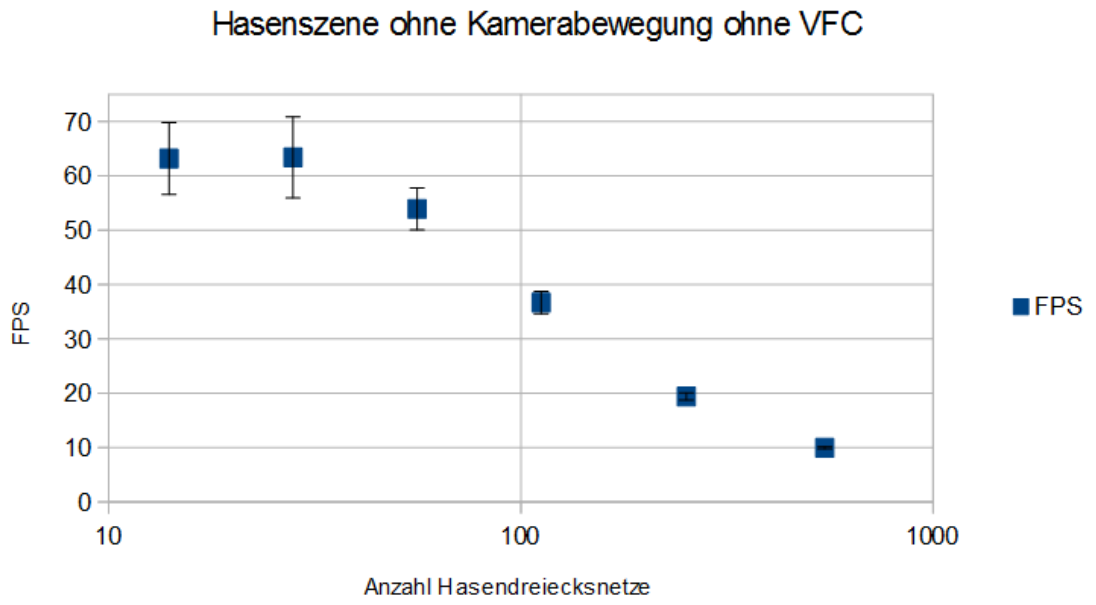

Abbildung 5.8: Verlauf der Framerate der Hasenszene ohne Bewegung der Kamera ohne View Frustum Culling; die X-Achse ist logarithmisch skaliert.

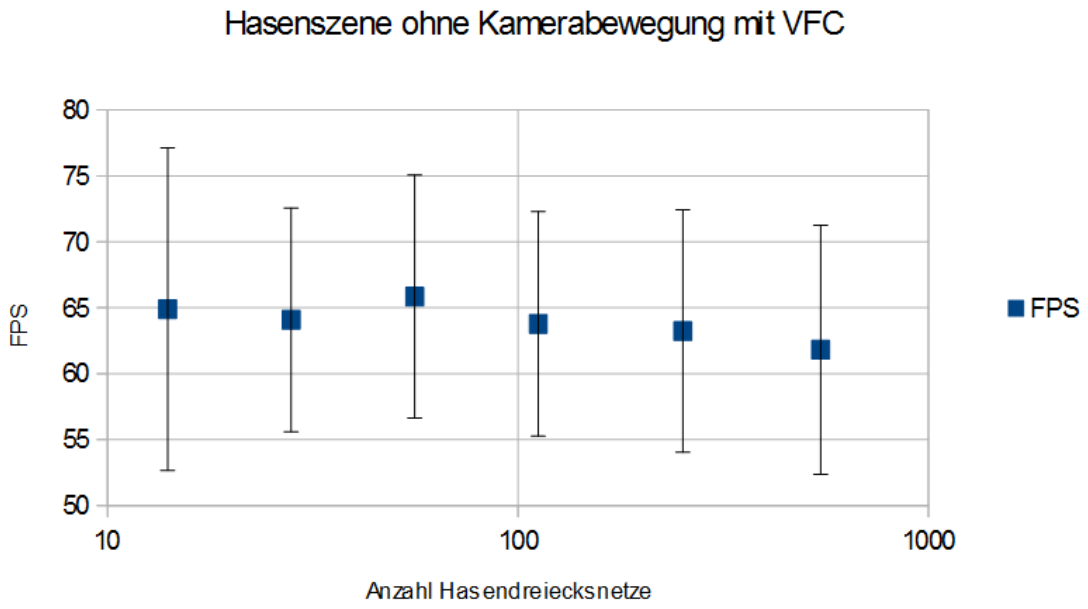

Abbildung 5.9: Verlauf der Framerate der Hasenszene ohne Bewegung der Kamera mit View Frustum Culling; die X-Achse ist logarithmisch skaliert.

	14 Hasen	28 Hasen	56 Hasen	112 Hasen	252 Hasen	546 Hasen
FPS	67,64	65,31	48,95	34,47	18,22	9,76
Varianz	180,37	89,45	60	24,23	17,25	0,03
Standardabweichung	13,27	9,41	7,74	4,92	4,14	0,16

Tabelle 5.8: Hasenszene mit steigender Anzahl Dreiecksnetzen mit Bewegung der Kamera ohne View Frustum Culling.

	14 Hasen	28 Hasen	56 Hasen	112 Hasen	252 Hasen	546 Hasen
FPS	66,27	66,13	75,61	74,33	75,89	78,64
Varianz	224,22	181,86	489,26	503,84	443,49	599,42
Standardabweichung	14,82	13,17	22,06	22,41	19,19	24,27

Tabelle 5.9: Hasenszene mit steigender Anzahl Dreiecksnetzen mit Bewegung der Kamera mit View Frustum Culling.

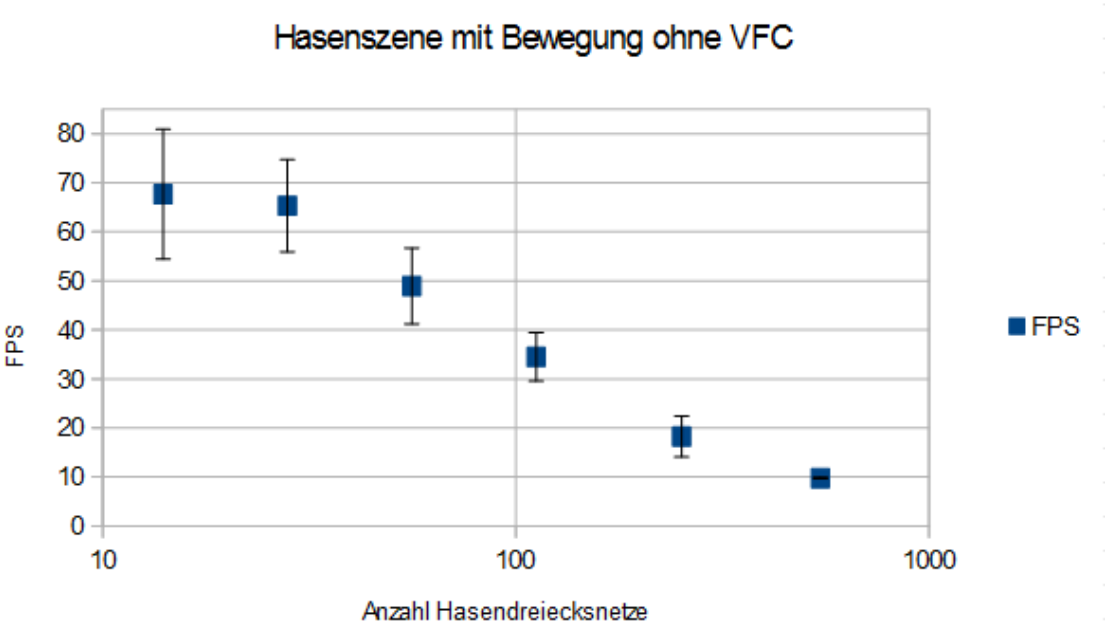

Abbildung 5.10: Verlauf der Framerate der Hasenszene mit Bewegung der Kamera ohne View Frustum Culling; die X-Achse ist logarithmisch skaliert.

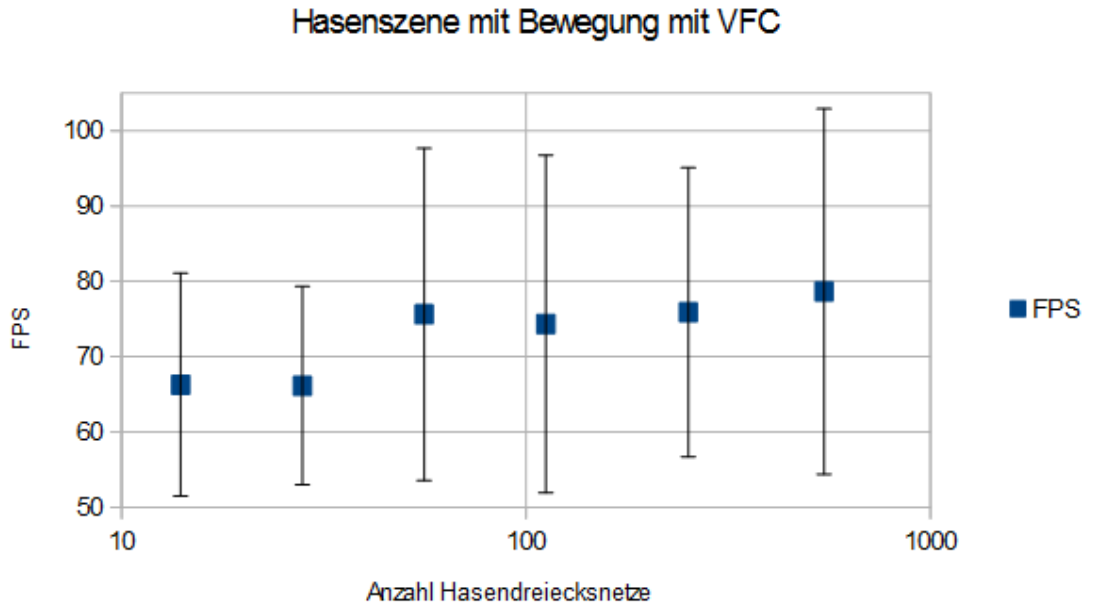

Abbildung 5.11: Verlauf der Framerate der HasenSzene mit Bewegung der Kamera mit View Frustum Culling; die X-Achse ist logarithmisch skaliert.

Tabellen 5.6, 5.7, 5.8 und 5.9 zeigen die Framerate, Varianz und Standardabweichung sowie Abbildungen 5.8, 5.9, 5.10 und 5.11 die Framerate und Standardabweichung (Fehlerbalken) in Abhängigkeit von der Anzahl der Hasen-Dreiecksnetze. Zum einen wird deutlich, dass die Framerate ohne View Frustum Culling bei steigender Anzahl der Dreiecksnetze sinkt. Mit View Frustum Culling hingegen ist die Framerate konstant und unabhängig von der steigenden Anzahl der Dreiecksnetze. Dies untermauert, dass bei Hinzukommen von Dreiecksnetzen außerhalb des Sichtbarkeitsbereichs die Framerate unter Anwendung des View Frustum Cullings konstant bleibt, da diese Dreiecksnetze unsichtbar geschaltet und somit nicht gerendert werden.
Zum anderen sinkt ohne View Frustum Culling die Standardabweichung deutlich mit steigender Anzahl Dreiecksnetzen (von 6,62 bei 14 Hasen-Dreiecksnetzen auf 0,17 bei 546 Hasen-Dreiecksnetzen ohne Bewegung (siehe Tabelle 5.6) beziehungsweise von 13,27 bei 14 Hasen-Dreiecksnetzen auf 0,16 bei 546 Hasen-Dreiecksnetzen mit Kamerabewegung (siehe 5.8)). Dies kann daran liegen, dass durch die hohe Auslastung bei der Messung, die durch den steigenden Rendering-Aufwand entsteht, deutlich weniger Messwerte zur Verfügung standen und somit ein kleinerer Schwankungsraum entstehen konnte.
Mit View Frustum Culling hingegen bewegt sich die Standardabweichung zwischen ~ 8 und ~ 12 (ohne Bewegung der Kamera, siehe Tabelle 5.7) beziehungsweise steigt von 14,82 bei 14 Hasen-Dreiecksnetzen auf 24,27 bei 546 Hasen-Dreiecksnetzen (mit Bewegung der Kamera, siehe Tabelle 5.9). Diese Tatsache kann damit zusammenhängen, dass bei Anwendung des View Frustum Cullings zwischen den Frames mehr Berechnungen (insbesondere bei Kamerabewegung) sowie beispielsweise unsteuerbare Java Garbage Collection ausgeführt wird, sodass die Auslastung unausgeglichen ist und ein größerer Schwankungsraum entstehen kann.

5.4 Zusammenfassung der Messergebnisse

Im Allgemeinen entsprechen die Messergebnisse den Erwartungen. Tendenziell ist unter Verwendung des View Frustum Cullings eine höhere Framerate zu beobachten. Insbesondere die relativ konstante Framerate bei steigender Anzahl an Dreiecksnetzen unter Verwendung des View Frustum Cullings erfüllt die erhofften Erwartungen und deutet auf einen deutlichen Performancegewinn hin. Die höhere Standardabweichung unter Verwendung des View Frustum Cullings lässt sich auf die zusätzlichen Berechnungen zurückführen.
Erstaunlich ist allerdings die, wenn auch nur leicht, steigende Framerate der Hasen-Szene bei Kamerabewegung und mit View Frustum Culling. Dies lässt sich eventuell auf Messungenauigkeiten zurückführen. Den Erwartungen nach sollte in diesem Fall die Framerate nämlich sinken,

da bei Bewegung und einer steigenden Anzahl Dreiecksnetze mehr Kollisionsberechnungen notwendig sind, was Performanceeinbußen zur Folge hat.

6 Schlussbetrachtung

6.1 Fazit

Das Ziel dieser Bachelor-Arbeit war die Evaluation der Sichtbarkeitsberechnung der Computergrafik sowie Verbesserungsmöglichkeiten derer mit Hilfe hierarchischer Strukturen hinsichtlich Performanceverbesserungen. Dabei wurde auf diverse Culling-Techniken sowie Anwendungsmöglichkeiten hierarchischer Strukturen eingegangen. Des Weiteren wurde das Kameramodell der Computergrafik präsentiert. Anhand einer Implementierung des View Frustum Cullings in das Computergrafik-Framework der Computergrafik-Gruppe der HAW mit Optimierung mittels Einsatz eines Octrees sollte Bezug zur Praxis hergestellt und der tatsächlich erreichte Performancegewinn gemessen werden.
Die Messungen der Framerate mit verschiedenen Testszenen unter Verwendung des View Frustum Cullings ergaben im Vergleich zu Messungen ohne View Frustum Culling einen Performancegewinn bei einer festen Anzahl und teils außerhalb des Sichtbarkeitsbereichs positionierten Dreiecksnetzen, der sich in einer leicht höheren Framerate widerspiegelte. Bei Szenen mit wachsender Anzahl Dreiecksnetzen außerhalb des Sichtbarkeitsbereichs hingegen wurde eine deutliche Performanceverbesserung festgestellt. Ohne View Frustum Culling sank die Framerate stetig bei steigender Anzahl an Dreiecksnetzen, während sie unter Verwendung des View Frustum Cullings schließlich stagnierte. Dieses Messergebnis unterstreicht die Funktionalität des View Frustum Cullings, eine höhere Framerate durch Vermeidung von unnötigem Rendering zu erreichen.

6.2 Ausblick

Bei der Umsetzung zeigten sich Schwierigkeiten bei der Positionsbestimmung von Dreiecksnetzen bezüglich des View Frustums. Die aktuelle Lösung der Positionsbestimmung deckt einen Großteil der Positionen ab, allerdings gibt es Sonderfälle, die fälschliches Rendering von Dreiecksnetzen außerhalb des Sichtbarkeitsbereichs zur Folge haben. An dieser Stelle wäre eine mögliche Ansatzstelle, diese Sonderfälle korrekt zu bearbeiten.

Ein weiterer möglicher Ansatzpunkt ist die Berechnung des Sichtbarkeitsbereichs. Aktuell werden aus den Kameraparametern die Eckpunkte und Ebenen immer wieder neu und unabhängig vom vorherigen Sichtbarkeitsbereich berechnet. Denkbar wäre hier der Einsatz einer Matrix, die die Kamerabewegung relativ zur alten Position abbildet und mit welcher die alten Eckpunkte multipliziert werden.

Außerdem wurde prototypisch das (Un-)Sichtbarschalten einzelner Dreiecke implementiert mit dem Ziel, nur teilweise innerhalb des Frustums liegende Dreiecksnetze auch nur teilweise zu rendern. Dieser Prototyp zeigte sich allerdings als sehr unperformant, sodass sich in der aktuellen Implementierung die Positionsbestimmung von Dreiecksnetzen auf komplett innerhalb oder komplett außerhalb des Sichtbarkeitsbereichs liegend beschränkt. So werden Dreiecksnetze, die möglicherweise nur minimal den Sichtbarkeitsbereich berühren, komplett gerendert, was unnötigen Renderaufwand zur Folge haben kann. Das nur teilweise Sichtbarschalten von Dreiecksnetzen bildet somit einen weiteren Ansatzpunkt für mögliche Performanceverbesserungen.

Literaturverzeichnis

[Airey 1990] Airey, John M.: Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations / DTIC Document. 1990. – Forschungsbericht

[Akenine-Möller u. a. 2008] Akenine-Möller, Tomas ; Haines, Eric ; Hoffman, Naty: *Real-Time Rendering 3rd Edition*. Natick, MA, USA : A. K. Peters, Ltd., 2008. – 1045 S. – ISBN 987-1-56881-424-7

[Assarsson und Moller 2000] Assarsson, Ulf ; Moller, Tomas: Optimized view frustum culling algorithms for bounding boxes. In: *Journal of Graphics Tools* 5 (2000), Nr. 1, S. 9–22

[Barsky u. a. 2003a] Barsky, Brian A. ; Horn, Daniel R. ; Klein, Stanley A. ; Pang, Jeffrey A. ; Yu, Meng: Camera models and optical systems used in computer graphics: part I, object-based techniques. In: *Computational Science and Its Applications - ICCSA 2003*. Springer, 2003, S. 246–255

[Barsky u. a. 2003b] Barsky, Brian A. ; Horn, Daniel R. ; Klein, Stanley A. ; Pang, Jeffrey A. ; Yu, Meng: Camera models and optical systems used in computer graphics: part II, image-based techniques. In: *Computational Science and Its Applications - ICCSA 2003*. Springer, 2003, S. 256–265

[Catmull 1974] Catmull, Edwin: A subdivision algorithm for computer display of curved surfaces / DTIC Document. 1974. – Forschungsbericht

[Chang u. a. 2008] Chang, Jung-Woo ; Wang, Wenping ; Kim, Myung-Soo: Efficient collision detection using a dual bounding volume hierarchy. In: *Advances in Geometric Modeling and Processing*. Springer, 2008, S. 143–154

[Cohen-Or u. a. 2003] Cohen-Or, Daniel ; Chrysanthou, Yiorgos L. ; Silva, Cláudio T ; Durand, Frédo: A survey of visibility for walkthrough applications. In: *Visualization and Computer Graphics, IEEE Transactions on* 9 (2003), Nr. 3, S. 412–431

[Gottschalk u. a. 1996] Gottschalk, Stefan ; Lin, Ming C. ; Manocha, Dinesh: OBBTree: A hierarchical structure for rapid interference detection. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* ACM (Veranst.), 1996, S. 171–180

[Govindaraju u. a. 2003] Govindaraju, Naga K. ; Redon, Stephane ; Lin, Ming C. ; Manocha, Dinesh: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In: *Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware* Eurographics Association (Veranst.), 2003, S. 25–32

[Greene u. a. 1993] Greene, Ned ; Kass, Michael ; Miller, Gavin: Hierarchical Z-buffer visibility. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* ACM (Veranst.), 1993, S. 231–238

[Hughes u. a. 1990] Hughes, John F. ; Van Dam, Andries ; McGuire, Morgan ; Sklar, David F. ; Foley, James D. ; Feiner, Steven K. ; Akeley, Kurt: *Computer Graphics: Principles and Practice (2Nd Ed.)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1990. – ISBN 0-201-12110-7

[Kolb u. a. 1995] Kolb, Craig ; Mitchell, Don ; Hanrahan, Pat: A realistic camera model for computer graphics. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* ACM (Veranst.), 1995, S. 317–324

[Kumar u. a. 1996] Kumar, Subodh ; Manocha, Dinesh ; Garrett, Bill ; Lin, Ming: Hierarchical back-face culling. In: *7th Eurographics Workshop on Rendering*, 1996, S. 231–240

[Lighthouse3d.com a] Lighthouse3d.com: *Geometric Approach - Extracting the Planes*. http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/. – Accessed April 15, 2016

[Lighthouse3d.com b] Lighthouse3d.com: *Geometric Approach - Testing Boxes*. http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-boxes/. – Accessed April 21, 2016

[Luebke und Georges 1995] Luebke, David ; Georges, Chris: Portals and mirrors: Simple, fast evaluation of potentially visible sets. In: *Proceedings of the 1995 symposium on Interactive 3D graphics* ACM (Veranst.), 1995, S. 105–ff

[Potmesil und Chakravarty 1982] Potmesil, Michael ; Chakravarty, Indranil: Synthetic image generation with a lens and aperture camera model. In: *ACM Transactions on Graphics (TOG)* 1 (1982), Nr. 2, S. 85–108

[Pramberger 2012] Pramberger, Ralf: Räumliche Datenstrukturen. (2012)

[Ritter 1990] Ritter, Jack: An efficient bounding sphere. In: *Graphics gems* Academic Press Professional, Inc. (Veranst.), 1990, S. 301–303

[Samet und Webber 1988] Samet, Hanan ; Webber, Robert E.: Hierarchical data structures and algorithms for computer graphics. I. Fundamentals. In: *Computer Graphics and Applications, IEEE* 8 (1988), Nr. 3, S. 48–68

[Shinya 1994] Shinya, Mikio: Post-filtering for depth of field simulation with ray distribution buffer. In: *Graphics Interface* Canadian Information Processing Society (Veranst.), 1994, S. 59–59

[Shirmun und Abi-Ezzi 1993] Shirmun, Leon A. ; Abi-Ezzi, Salim S.: The cone of normals technique for fast processing of curved patches. In: *Computer Graphics Forum* Bd. 12 Wiley Online Library (Veranst.), 1993, S. 261–272

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 07. Juni 2016 Anne-Lena Kowalka