Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marcel Kuhn

Echtzeit-Schatten für interaktive Visualisierungsanwendungen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marcel Kuhn

Echtzeit-Schatten für interaktive Visualisierungsanwendungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Philipp Jenke
Zweitgutachter : Prof. Dr. Klaus-Peter Kossakowski

Abgegeben am 5. April 2016

Marcel Kuhn

Thema der Arbeit
Echtzeit-Schatten für interaktive Visualisierungsanwendung

Stichworte
Computergrafik, Schattenalgorithmen, Schattenvolumen, echtzeitfähig

Kurzzusammenfassung
Diese Arbeit stellt verschiedene echtzeitfähige Schattenalgorithmen vor, sowie eine Alternative zu den klassischen Schattenalgorithmen, welche jedoch nicht echtzeitfähig ist. Einer der Algorithmen wurde prototypisch implementiert und evaluiert.

Marcel Kuhn

Title of the paper
Real-Time-Shadows for interactive visual applications

Keywords
Computer graphics, shadow algorithms, shadow volumes, real-time

Abstract
This thesis researches various real-time shadow algorithms and introduces an alternative approach, which is not capable of rendering shadows in real time. One of the algorithms was implemented and evaluated.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation 2
1.2 Zielsetzung 2
1.3 Aufbau der Arbeit 3
2 Stand der Technik 4
2.1 Begriffsklärung 4
2.1.1 Rendering-Pipline 4
2.1.2 Transformation 5
2.1.3 Clipping 5
2.1.4 Shading 5
2.1.5 Echtzeit 6
2.2 Hardwareaufbau 6
2.2.1 Framebuffer 6
2.2.2 Color Buffer 6
2.2.3 Depth/Z-Buffer 7
2.2.4 Stencil Buffer 7
2.3 Lokale Beleuchtungsmodelle 7
2.4 Globale Beleuchtungsmodelle 9
2.5 Alternative Verfahren zur Schattenberechnung 10
2.5.1 Shadow Mapping 10
2.5.2 Shadow Volumes 13

3 Konzept 16
3.1 Grundkonzept der Schattenvolumen 16
3.2 Z-Pass und Z-Fail 17
3.3 Eleminierung der Far-Plane 21
3.4 Unendliche Schattenvolumen 23
3.5 Bestimmung von Silhouettenkanten 24
3.6 Weiche Schatten 26
4 Implementierung 28
4.1 Das Framework 28
4.1.1 Allgemeines 28
4.1.2 Aufbau 29
4.2 Grundaufbau des Algorithmus 30
4.2.1 Erzeugung der Kollisionspyramide 33
4.3 Generierung von Schattenvolumen 35
4.3.1 Grundgerüst eines Datenstruktur-Renderers 35
4.3.2 Unterscheidung zwischen Z-Pass und Z-Fail 37
4.3.3 Zeichnen der Schattenvolumen 37
4.3.4 Bestimmung der Silhouettenkanten 39
4.3.5 Kanten 40
5 Evaluation 42
5.1 Aufbau 42
5.2 Stärken 42
5.3 Schwächen 45
5.4 Performance 47
5.4.1 Messungen 47
5.4.2 Z-Pass vs. Z-Fail 52
6 Fazit und Schluss 54
6.1 Fazit 54
6.2 Ausblick 55
Literaturverzeichnis 56

1 Einleitung

Licht und Schatten sind ein wichtiger Bestandteil der Raumwahrnehmung. Sie liefern dem menschlichen Gehirn wichtige Informationen über die Beschaffenheit der Oberfläche sowie dreidimensionale Lage des Objektes. Der Eigenschatten gibt einem Körper Volumen und Ausdehnung, während der Schlagschatten einen Bezug zu anderen Flächen und Körpern im Raum darstellt.

In der Computergrafik versucht man mit Hilfe von Algorithmen und sogenannten Shadern diese Eigenschaften zu reproduzieren. Hierbei wird zwischen Echtzeit- und Nicht-Echtzeitlösungen unterschieden. Für die Erzeugung von Eigenschatten wird in beiden Fällen meist ein Shader benutzt. Die eigentliche Problematik befindet sich bei den Schlagschatten. Eine Variante zur Generierung von Schlagschatten ist das Ray-Tracing, welches sich sehr nah an der Realität orientiert und aus diesem Grund einen physikalisch sehr korrekten Schatten generiert. Diese Methode ist wegen der aufwendigen Berechnungen nicht Echtzeit tauglich. Für Echtzeitanwendungen sind Algorithmen wie das Shadow Mapping oder die Shadow Volumes besser geeignet. Da sich diese Arbeit mit Echtzeitanwendungen auseinandersetzt, wird die Thematik des Ray-Tracings nicht weiter vertieft.

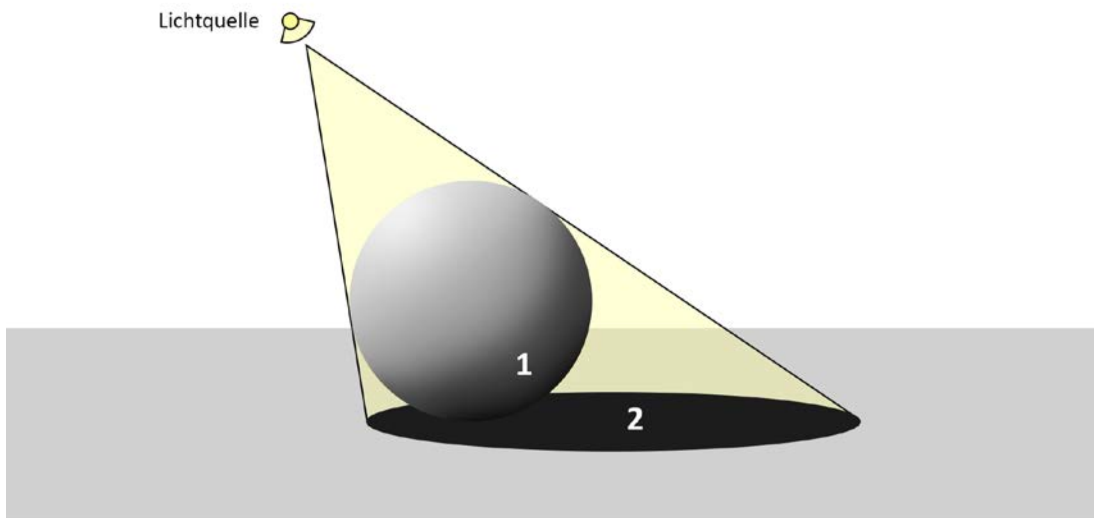


Abbildung 1.1: Eigenschatten (1) und Schlagschatten (2) anhand einer Kugel dargestellt.

1.1 Motivation

Da wie bereits zuvor beschrieben, Schatten eine wichtige Rolle spielen, um eine Szene realistisch wirken zu lassen, sind Schatten-Algorithmen auch heute noch ein interessantes und wichtiges Thema. Des Weiteren hat sich die Leistung der Prozessoren und besonders der Grafikarten in den letzten Jahren deutlich gesteigert. Es soll im Rahmen dieser Arbeit an Hand eines Prototypens geschaut werden, wie viel Leistung Schattenalgorithmen mit heutiger Hardware erzielen.

1.2 Zielsetzung

Das Hauptziel dieser Arbeit besteht in der Analyse der Möglichkeiten, Schatten in Echtzeitanwendungen zu generieren und in der prototypischen Umsetzung eines Ansatzes.
Der im Rahmen dieser Arbeit entstandene Prototyp soll folgende Kriterien erfüllen. Es soll möglich sein, Körper und Lichtquellen während der Laufzeit zu verändern. Des Weiteren soll das System in der Lage sein, weiche Schatten zu erzeugen. Die Performance der weichen Schatten liegt hierbei jedoch nicht im Vordergrund. Außerdem muss die Funktion der Schattengenerierung, während der Laufzeit für die einzelnen Körper dynamisch ein- und ausschaltbar sein.

1.3 Aufbau der Arbeit

Die Arbeit ist in mehrere aufeinander aufbauende Kapitel aufgeteilt. In dem Kapitel „Stand der Technik“ werden zuerst wichtige Begriffe, Komponenten und Techniken erklärt, die zum Verständnis des darauffolgenden Textes notwendig sind. Anschließend werden verschiedene Ansätze zum Lösen der Schattenproblematik sowie das in der Computergrafik zu Grunde liegende Beleuchtungsmodell vorgestellt. Nachfolgend wird der im Rahmen dieser Arbeit entwickelte Prototyp vorgestellt und evaluiert. Als Abschluss dieser Arbeit folgen eine Zusammenfassung der erlangten Ergebnisse sowie ein Ausblick für zukünftige Arbeiten, die auf dieser Arbeit aufsetzen können.

2 Stand der Technik

2.1 Begriffsklärung

2.1.1 Rendering-Pipline

Unter der Rendering-Pipeline (dt. Grafikpipline) versteht man den Ablauf, den ein Grafiksystem durchläuft, um die eingehenden Daten auf dem Bildschirm zu bringen. Was an dieser Stelle wichtig zu erwähnen ist, dass mehrere Durchläufe erfolgen können, bevor das Bild an den Monitor weitergegeben wird. Im Groben kann die Rendering-Pipline in drei Abschnitte unterteilt werden: Anwendung, Geometrie und Rasterung.

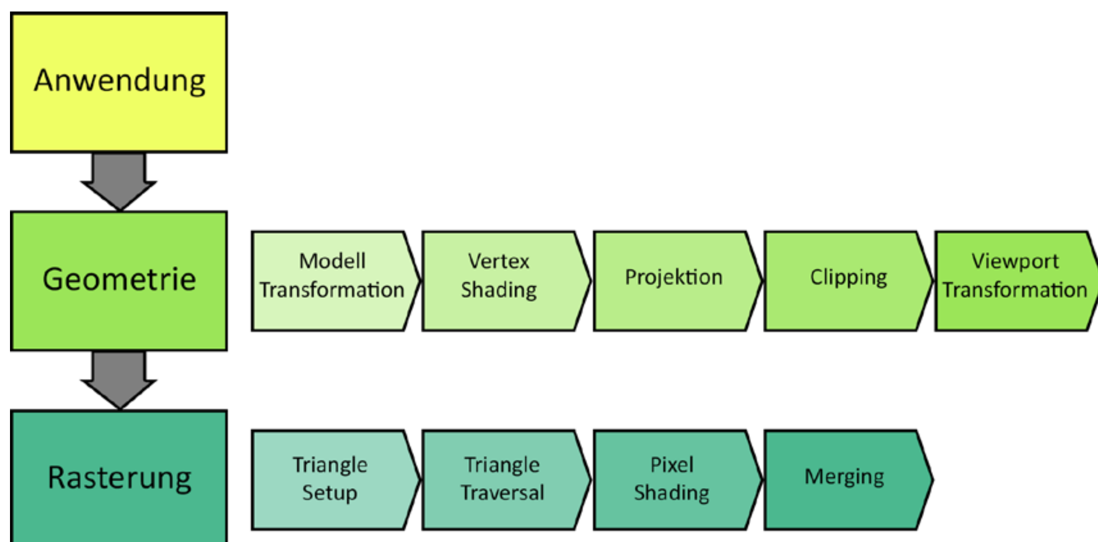


Abbildung 2.1: Die einzelnen Abschnitte der Rendering-Pipline (oben nach unten) und deren Komponenten (links nach rechts)

In dem Anwendungsabschnitt werden von dem Programmierer Datenstrukturen angelegt. Die Wahl dieser ist völlig frei. Typische Datenstrukturen in der Anwendungsschicht sind dabei Repräsentationen von Vertices, Triangles und Meshes.
In dem Abschnitt Geometrie werden geometrische Operationen an den einzelnen Polygonen (meist Dreiecke) vorgenommen. Dazu gehören Operationen wie Transformation von Modellen und Kamera, Vertex Shading, Projektion, Clipping sowie die Viewport-Transformation. Im letzten Abschnitt, der Rasterung, wird das Erzeugte aus Sicht der Kamera auf die einzelnen zur Verfügung stehenden Pixel übertragen.
Für diese Arbeit wichtig zu erwähnen sind die auf der Grafikkarte zur Verfügung stehenden Puffer, welche im Abschnitt 2.2 Hardwareaufbau genauer erklärt werden.

2.1.2 Transformation

Unter der Transformation versteht man, die neue Ausrichtung, sodass die Kamera entlang der z-Achse schaut. Objekte mit größeren z-Werten sind dabei weiter von der Kamera entfernt. Eine Transformation wird mit Hilfe von Translationen (Verschiebungen), Rotationen und Skalierungen erreicht.

2.1.3 Clipping

Als Clipping wird das Abschneiden (engl. Clipping) von nicht benötigten Körpern bzw. Polygone bezeichnet. Dies geschieht im Geometrieabschnitt. Dabei werden Polygone, die komplett im Sichtbarkeitsbereich (engl. Viewing frustum) liegen zur Rasterung weitergegeben. Polygone, die komplett außerhalb des Bereichs liegen, werden nicht weitergereicht. Das Abschneiden wird bei Polygonen durchgeführt, bei denen ein Teil sichtbar ist und ein anderer nicht. Es wird die Stelle ermittelt, wo das Polygon die Flächen des Sichtbarkeitsbereichs schneidet und ersetzt die außerhalb liegenden Vertices durch neue, die sich auf der Fläche befinden.

2.1.4 Shading

Mit Hilfe von Shading (dt. Schattierung) wird in der Computergrafik versucht eine möglichst realistische Darstellung der Körper, in Beziehung zu den im dreidimensionalen Raum existierenden Lichtquellen, zu schaffen. Dafür muss jeder Körper Informationen über sein Material enthalten. Typische Materialeigenschaften sind dabei der ambiente, diffuse und der spekularer Lichtanteil. Populäre Shading-Verfahren sind das Gouraud Shading von Henri Gouraud und das Phong Shading von Bui Tuong Phong. Hierbei darf das Phong Shading

nicht mit dem Phong-Beleuchtungsmodel verwechselt werden, welches in Kapitel 2.2 besprochen wird. Durch die genannten Shader erhalten Körper bereits Eigenschatten.

Es wird zwischen zwei verschiedenen Shadern unterschieden, zum einen der Vertex Shader, zum anderen der Pixel Shader. Der Vertex Shader wird in der Rendering-Pipline in dem Geometrieabschnitt ausgeführt und auf Basis von Vertices (dt. Knoten) berechnet. Der Pixel Shader hingegen wird im Rasterungsabschnitt für jeden Pixel ausgeführt.

2.1.5 Echtzeit

Allgemein in der Informatik bedeutet der Begriff Echtzeit, dass Signale garantiert in bestimmten Zeiten bearbeitet werden. In der Computergrafik hingegen handelt es sich bei Echtzeit um eine Bildwiederholrate, die interaktive Anwendungen erlaubt. Meistens wird eine Anwendung als echtzeitfähig angesehen, wenn die Bildwiederholrate mindestens 30 FPS beträgt.

2.2 Hardwareaufbau

2.2.1 Framebuffer

Bei dem Framebuffer (dt. Bildpuffer) handelt es sich eigentlich um den Color Buffer. Oftmals wird jedoch das Gespann der Puffer auf der Grafikkarte als Framebuffer bezeichnet. Das Gespann stellen dabei der Farbpuffer, Tiefenpuffer und der Stencil Buffer dar.

2.2.2 Color Buffer

Der Color Buffer (dt. Farbpuffer) enthält Informationen über die Farbwerte sowie die Transparenz des jeweiligen Pixels. Bei den Werten handelt es sich um die Anteile von Rot, Grün und Blau. Für die Transparenz wird ein weiterer Wert gespeichert, welcher meist Alpha genannt wird. Der Color Buffer ist in mancher Literatur auch als Framebuffer zu finden.

2.2.3 Depth/Z-Buffer

In dem Depth Buffer (dt. Tiefenpuffer) werden Informationen über die Sichtbarkeit der einzelnen Körper abgespeichert. Hierbei wird aus Sicht der Kamera überprüft, wie weit jedes Objekt entfernt ist.

2.2.4 Stencil Buffer

Der Stencil Buffer (dt. Schablonenpuffer) wird zum Abspeichern der Position von gerenderten Objekten benutzt. Meistens stehen dafür acht Bits pro Pixel im Puffer zur Verfügung. Die im Puffer gespeicherten Informationen können dazu verwendet werden um das Schreiben in den Color-Buffer und Z-Buffer zu steuern. Beispielsweise, im ersten Durchgang wird eine Szene komplett in roten Licht gerendert und im Stencil Buffer ist ein Kreis in der Mitte des Bildes abgespeichert. Im zweiten Renderdurchgang wird dieselbe Szene noch einmal mit normalen Licht (Rot-, Grün- und Blauanteile besitzen alle den gleichen Wert) gezeichnet, mit Berücksichtigung des Stencil-Buffer-Inhalts. Je nachdem, welche Operation verwendet wird, entsteht ein „normales“ Bild mit einem roten Kreis in der Mitte oder ein rotes Bild mit einem „normalen“ Kreis in der Mitte.

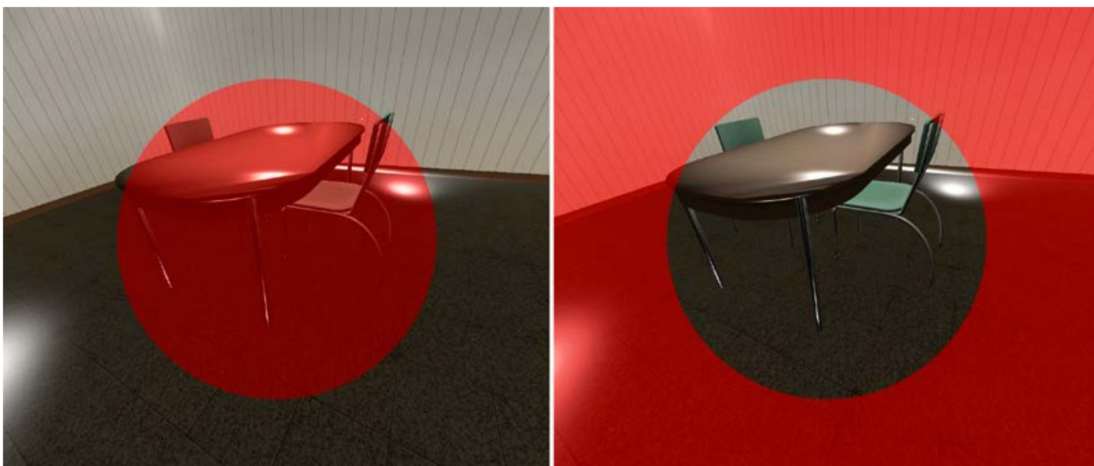


Abbildung 2.2: Resultate des in Kapitel 2.2.4 beschriebenen Beispiels

2.3 Lokale Beleuchtungsmodelle

Die Aufgabe eines Beleuchtungsmodells in der Computergrafik besteht in der vereinfachten Berechnung der Lichtdichte an Oberflächen. Es wird oft auch von lokalen Beleuchtungsmodellen gesprochen, da die Berechnung für jede Oberfläche einzeln

vorgenommen wird und somit unabhängig von dem Rest der Szene ist. Es wird zwischen drei verschieden Lichtarten unterschieden: dem ambienten Licht, dem diffusen Licht und der spekularen Reflexion.

Bei dem ambienten Lichtanteil handelt es sich um ein in der Realität nicht vorhanden Anteil. Dieser wird dennoch benutzt um überall in der Szene etwas Licht zu haben. Es simuliert die aus der Physik indirekten Beleuchtungen von der Umgebung. Als Beispiel, in einem Raum steht ein Tisch, dieser wirft einen Schatten auf dem Boden. In der Realität wäre nun der Bereich des Fußbodens, durch die indirekte Beleuchtung des Raumes dunkler, jedoch nicht komplett schwarz.

Der diffuse Anteil sagt aus, wie viel Licht von der Oberfläche eines Objektes in alle Richtungen reflektiert wird. Dies ist abhängig vom Einfallswinkel des Lichtes. Strahlt das Licht senkrecht auf die Oberfläche, so wird diese stark beleuchtet. Umso größer der Winkel zwischen der Oberflächennormale und dem Einfallswinkel des Lichtes wird, umso weniger Licht wird von der Oberfläche reflektiert.

Der spekulare Anteil dient zur Erzeugung von Glanzpunkten auf der Oberfläche. In der Realität entspricht dies die ideale Reflexion auf der Oberfläche, also der Punkt bei dem gilt: Einfallswinkel gleich Ausfallswinkel. Als Beispiel, wenn man an einen sonnigen Tag auf eine Motorhaube schaut, sieht man dort oft ganz viele kleine glänzende Punkte. An diesen Stellen werden die Sonnenstrahlen vom Lack direkt in das menschliche Auge reflektiert.

Die Beleuchtungsmodelle in der Computergrafik berücksichtigen außerdem die Materialeigenschaften der jeweiligen 3D-Objekte. Dies liegt daran, dass wie in der Realität nicht jeder Gegenstand die gleichen Eigenschafften ausweist. Beispielsweise reflektiert ein Spiegel deutlich mehr Licht als ein schwarzer Pullover. Zu den bekanntesten Beleuchtungsmodellen in der Computergrafik zählen das Phong-Beleuchtungsmodell und Blinn-Beleuchtungsmodell.

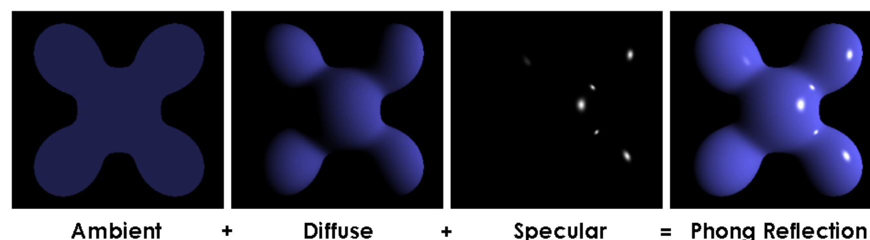


Abbildung 2.3: Hier als Beispiel das Phong-Beleuchtungsmodell: die einzelnen Lichtanteile sowie das fertige Bild [1]

---

[1] Brad Smith: *Illustration of the components of the Phong reflection model*, https://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell#/media/File:Phong_components_version_4.png

2.4 Globale Beleuchtungsmodelle

Anders als bei den lokalen Beleuchtungsmodellen, wird bei der Berechnung mit einem globalen Beleuchtungsmodell die ganze Szene betrachtet. Globale Beleuchtungsmodelle sind physikalisch wesentlich korrekter, dafür ist die Berechnung jedoch auch deutlich aufwendiger und meist nicht echtzeitfähig. Bekannte globale Beleuchtungsmodelle sind dabei das Ray-Tracing, Radiocity, Photon Tracing sowie das Path Tracing. Für einen detaillierteren Einblick in ein globales Beleuchtungsverfahren wurde im nachfolgenden Abschnitt das Ray-Tracing gewählt.

Bei dem Ray-Tracing handelt es sich um ein Verfahren zur Berechnung von Lichtausbreitung und nicht um einen Schattenalgorithmus. Das Ray-Tracing ist weitgehend physikalisch korrektes Verfahren. Weitgehend, da ebenfalls wie die Schattenalgorithmen, auch das Ray-Tracing ein physikalisch inkorrektes Beleuchtungsmodell als Grundlage benutzt. Durch die Funktionsweise des Ray-Tracings werden sowohl Schlag- als auch Eigenschatten erzeugt. Ray-Tracing-Algorithmen sind auf Grund des hohen Rechenaufwands nicht echtzeitfähig.
Um das Prinzip des Ray-Tracing zu verstehen, ist es wichtig zu wissen, wie sich das Licht im realen Leben ausbreitet. Eine Lichtquelle sendet einen Lichtstrahl (engl. ray) in den Raum. Zur Vereinfachung wird davon ausgegangen, dass es sich um ein perfektes Vakuum handelt, damit es sich bei dem Strahl um eine Gerade handelt. Der Strahl bewegt sich solange durch den Raum, bis er auf einen Gegenstand trifft. Je nach Eigenschaft der Oberfläche kann ein Prozentteil des Strahles absorbiert (engl. absorption), in eine andere Richtung reflektiert (enlg. reflection) oder von der Oberfläche gebrochen werden (engl. refraction). Des Weiteren kann eine Fluoreszenz auftreten (engl. fluorescence). Sollte der Gegenstand eine Transparenzeigenschaft aufweisen, wird ein Teil des Lichtstrahls in diesen gebrochen. Dadurch, dass nun einige Strahlen auf das menschliche Auge treffen, ist es für uns Menschen möglich zu sehen.

Das Grundkonzept des Ray-Tracing wurde 1968 von Arthur Appel [APP68] veröffentlicht. Es wird davon ausgegangen, dass jedes Modell in der Szene die bereits oben genannten Materialeigenschaften aufweisen kann. Anders als im realen Leben wird ein Strahl von der Position des Auges bzw. der Kamera für jeden Pixel in den Raum geschickt. Sollte der Strahl nun auf ein Objekt treffen, liegt dieses am Nächsten zur Kamera. Mit anderen Worten, es handelt sich um das Objekt, welches der Beobachter sieht. Mit Hilfe der Materialeigenschaften können nun die Schattierungen für dieses bestimmt werden.

Im Jahr 1979 veröffentlichte Turner Whitted [WHIT79] eine Erweiterung des Algorithmus, in dem er auftreffende Strahlen weiterverfolgt. Nach Whitted können nun beim Auftreffen drei neue Strahlen generiert werden. Dazu gehört ein Strahl für die Reflektion, Brechung sowie für den Schatten. Um den Schatten zu bestimmen wird ein Strahl zu jeder Lichtquelle im Raum gesendet. Mit diesem Strahl geprüft, ob sich ein nicht transparentes Objekt zwischen Licht und der betroffenen Oberfläche befindet.

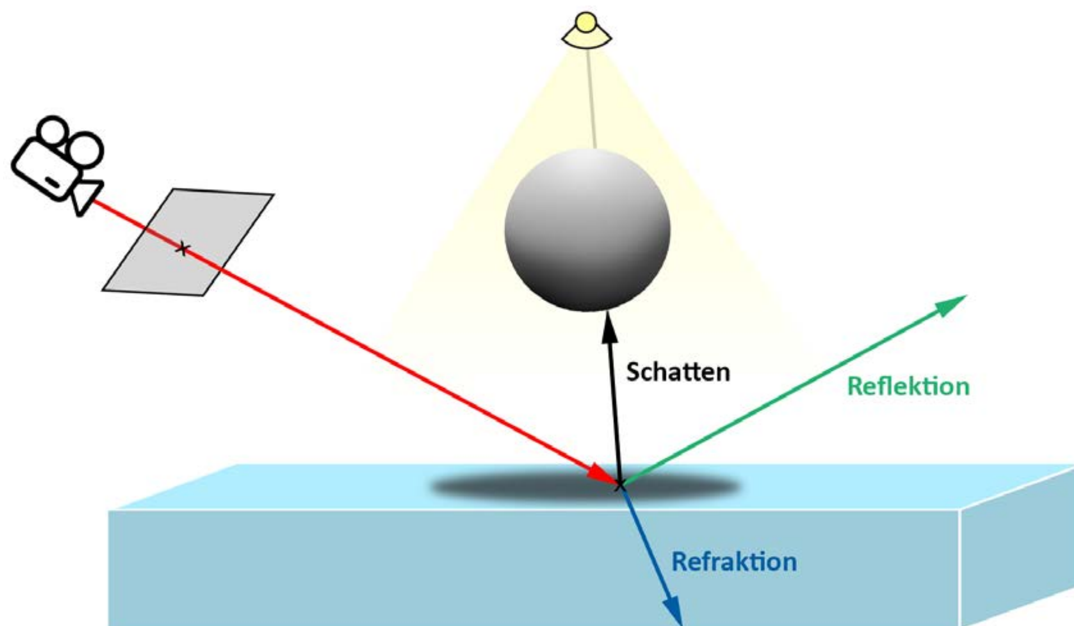


Abbildung 2.4: Ray-Tracing nach Whitted. Es wird ein Strahl in die Scene gesendet, dieser trifft auf den Quader. Drei neue Strahlen werden erzeugt und entsandt. Da die Kugel den Schattenstrahl zur Lichtquelle blockiert, liegt der Punkt im Schatten.

2.5 Alternative Verfahren zur Schattenberechnung

2.5.1 Shadow Mapping

Shadow Mapping (dt. Schattenkarte) ist ein Algorithmus zum Erzeugen von Schlagschatten und abhängig von der gewählten Implementierung auch Eigenschatten. Die Shadow Mapping-Algorithmen sind echtzeitfähig und werden sehr häufig in der Spieleentwicklung benutzt.

Das Grundkonzept wurde 1978 von Lance Williams [WILL78] vorgestellt. Williams Idee besteht darin, als erstes die Szene aus Sicht der schattenwerfenden Lichtquelle zu rendern und nur die Tiefenwerte in den Z-Buffer zu schreiben. Wichtig an dieser Stelle anzumerken ist, dass es sich um eine direktionale bzw. Scheinwerferlichtquelle (engl. Spotlight) handeln muss, da bei dem Ansatz davon ausgegangen wird, dass das Licht in eine Richtung „guckt“. Bei den im Z-Buffer abgespeicherten Werten handelt es sich bereits um die Shadow-Map. Diese ist lediglich eine Tiefenkarte aus Sicht der Lichtquelle. Um mit Hilfe der Shadow-Map Schatten in die Szene einzuzeichnen, wird diese ein zweites Mal aus Sicht der Kamera gerendert. Hierbei wird beim Rendern der Objekte, dessen Position für jeden benutzten

Pixel mit der Shadow-Map abgeglichen. Sollte sich der gerenderte Punkt weiterweg als der in der Karte hinterlegte befinden, so liegt der Punkt im Schatten.

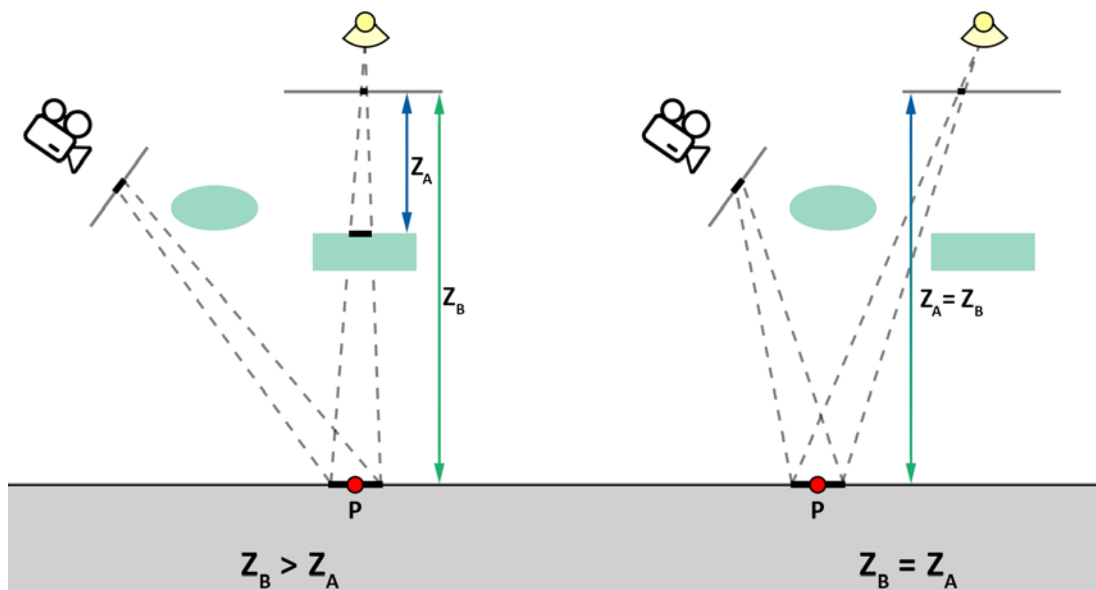


Abbildung 2.5: Auf der linken Seite ist der Z-Wert Z_B für Punkt P deutlich größer als der in der Shadow-Map hinterlegte Tiefenwert Z_A. Dem zu Folge wird P von einem Objekt verdeckt und liegt im Schatten. Auf der rechten Seite ist der Z-Wert Z_B gleich Z_A und P wird vom Licht angestrahlt.

Diese Technik ist sehr beliebt für Echtzeitanwendungen auf Grund des abschätzbaren Rechenaufwands zur Erzeugung der Shadow-Map. Dieser verhält sich linear zu den gerenderten Objekten in der Szene. Des Weiteren ist die Zugriffszeit auf die Werte in der Shadow-Map konstant. Ein großer Nachteil besteht darin, dass die Qualität der Schatten stark von der gewählten Größe der Map abhängen. Wird diese zu klein gewählt, so wirken die Schatten blockartig (siehe Abbildung 2.6). Dasselbe Problem kann ebenfalls auftreten, wenn die Kamera sich sehr nah an dem Objekt befindet und die Lichtquelle sehr weit weg ist. Das Letztere ist auch als „perspective aliasing“ bekannt. Ein weiteres Problem ist das „self aliasing“, hierbei erkennt der Algorithmus inkorrekter Weise ein Polygon als „befindet sich im Schatten“ an. Dieses Problem hat zwei Gründe. Zum einen die Ungenauigkeit der Datentypen in den Puffern des Prozessors, zum anderen der Fakt, dass ein Punkt zur Repräsentation von der Tiefe einer Fläche benutzt wird. Im Falle, dass der gespeicherte Tiefenwert des Lichts minimal kleiner ist als der der Fläche, entstehen Eigenschatten. Um dieses Problem zu umgehen, stellte Hourcade 1985 [HOUR85] ein Verfahren vor, bei dem IDs an Stelle von Tiefenwerte in Shadow Map abgespeichert werden. Hierbei erhält jedes Objekt oder Polygon eine ID. Wie bereits erwähnt, wird mit diesem Verfahren das Problem des „self aliasings“ umgangen. Jedoch ist das Erzeugen von Eigenschatten mit Hourcades Ansatz nicht möglich.

Eine Erweiterung dieses Algorithmus wurde von Dietrich [DIET01] sowie von Forsyth [FORS07] vorgestellt, welche eine Kombination aus Williams und Hourcades Ansatzes sind. Hierbei werden für die Schlagschatten eine ID-Karte und für die Eigenschatten eine Tiefenkarte benutzt. Es gibt noch weitere Ansätze um dieses Problem zu lösen, wie das Einfügen eines konstanten Tendenzwertes, welcher wiederum neue Probleme mit sich bringt, wenn dieser zu hoch gewählt wird.

Da in dieser Arbeit, die Schattenvolumen als Schwerpunkt gesetzt wurden, geht es hier lediglich um das Verständnis was Shadow-Maps sind und wie diese funktionieren. Aus diesem Grund, wird auf das Vertiefen von Verbesserungen, sowie das Erzeugen möglichst realistischer (weicher) Schatten verzichtet.

Abbildung 2.6: Eine mit dem reinen (ohne Smoothing) Shadow-Mapping-Algorithmus gerenderte Szene (Map-Auflösung von 256x256). An der Stuhllehne, sowie den Tischrundungen bei den Schatten ist die Blockbildung sehr deutlich zu erkennen.

2.5.2 Shadow Volumes

Als ein Schattenvolumen (engl. Shadow Volume) wird ein in der fertigen Szene nicht sichtbarer Körper bezeichnet, welcher die Szene in zwei Bereiche unterteilen soll: dem Schatten- und Nicht-Schattenbereich. Um ein Schattenvolumen zu erzeugen wird von der Lichtquelle ausgehend ein Tetraeder erzeugt. Hierbei ist die Position der Lichtquelle die Spitze und es wird über die Vertices eines Polygenes eine Gerade in die Unendlichkeit gezogen. Wird nun der Teil von der Lichtquelle zum Polygon bei dem Tetraeder weggelassen, bleibt das Schattenvolumen übrig. Mit Hilfe dieses Volumens kann nun bestimmt werden, welche Teile eines anderen sich in dem Volumen befindenden Körpers verdeckt werden.

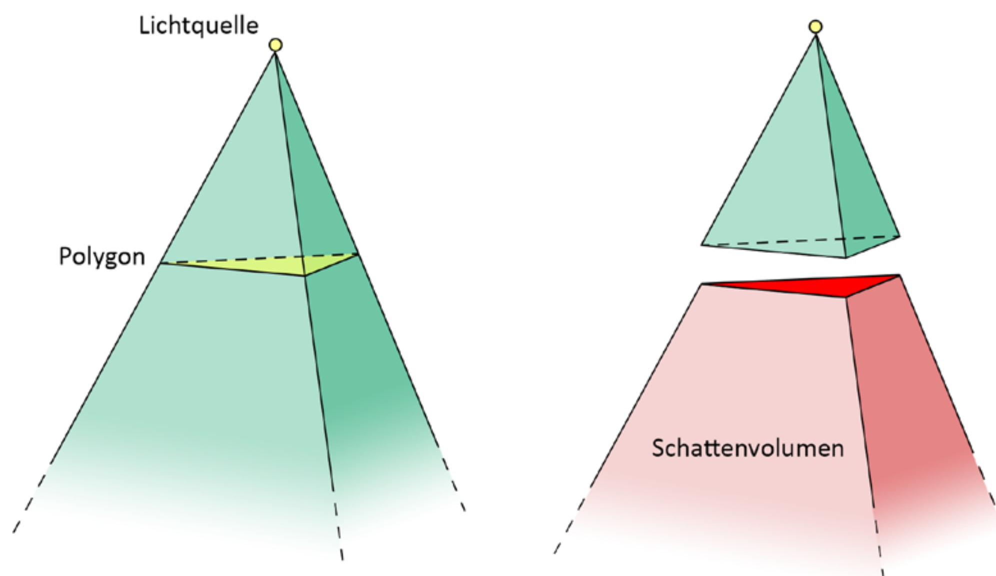


Abbildung 2.7: Erzeugung von Schattenvolumen, nachempfunden von [AM08, Seite 341]

Das Grundprinzip der Schattenvolumen wurde 1977 von Frank Crow [CROW77] vorgestellt. Crows ursprünglicher Algorithmus setzte zur Auswertung von Schattenvolumen auf Ray-Tracing. Für jeden Pixel aus Sicht des Beobachters wurde ein Strahl in den Raum entsandt und gezählt wie viele Schattenvolumen betreten und verlassen wurden, bis der Strahl auf das erste sichtbare Objekt im Raum trifft. Um zu erkennen ob ein Volumen betreten oder verlassen wurde, wird in Crows Ansatz überprüft ob es sich um eine Rückseite oder Vorderseite des betroffenen Schattenpolygons handelt. Crow wies ebenfalls schon auf die Problematik hin, dass das Zählen inkorrekt sei, wenn sich der Betrachter in einem Volumen befände.

Brotman und Badler [BRO84] bauten 1984 auf Corws Ansatz auf, in dem sie eine Implementierung mit Shading vorführten und zeigten, dass es möglich ist mit Hilfe von mehreren Lichtquellen, weiche Schatten zu erzeugen.

Im Jahr 1985 stelle Fuchs [FU85] die erste Hardwareimplementierung vor, die sogenannten Pixel-Planes. Anders als Crows Ansatz, funktionieren die Pixel-Planes ohne Ray-Tracing. Bei den Pixel-Planes wird für jeden Pixel und jedes schattenwerfende Polygon geprüft, ob der Pixel sich innerhalb des Volumens befindet. Das Volumen wird dabei durch das schattenwerfende Dreieck und die drei Schattenvolumenflächen (engl. Planes) gebildet. Das Auswerten der Volumen löste Fuchs mit Hilfe von Flächengleichungen.

Bergeron [BERG86] erweiterte 1986 den Grundalgorithmus von Crow, so dass dieser auch mit nicht-planaren Polygonen arbeiten konnte. Des Weiteren führte Bergeron die sogenannten Caps ein, also das Schließen der Volumen „bei Unendlich“, damit korrekt bestimmt werden kann in wie vielen Schattenvolumen sich der Betrachter befindet.

Im Jahre 1988 veröffentlichten Fournier und Fussell [FOUR88] einen Ansatz, bei dem der Frambuffer zur Berechnung der Schattenvolumen benutzt wurde. Dabei erhielt jeder Pixel im Framebuffer einen Tiefen- sowie Schattenzählwert. Die Idee von Fournier und Fussell diente als Grundlage des später eingeführten Stencil Buffers.

Heidmann [HEID91] stellte 1991 den ersten, auf den Stencil Buffer basierenden Ansatz vor, welcher als Grundlage der modernen Implementierung gilt. Als erstes wird die Szene nur mit ambienten Licht und Tiefentests gerendert. Die Tiefentests sind von daher notwendig, damit die Informationen über das am nahe liegenste Objekt für jeden Pixel vorhanden sind. Im zweiten Renderdurchlauf werden nach Heidmann die Schattenvolumen eingezeichnet, jedoch nur in den Stencil Buffer. Ansonsten wären die Schattenkörper in der fertigen Szene sichtbar. Hierbei werden die Volumen in zwei Durchgängen gezeichnet, als erstes werden nur die Vorderseiten gezeichnet und der Zähler im Stencil Buffer erhöht, wenn der Test erfolgreich war. Im zweiten Durchgang werden die Rückseiten gezeichnet und der Zähler jeweils verringert. Anschließend wird die Szene erneut gerendert, diesmal jedoch mit eingeschalteter Lichtquelle und Berücksichtigung der Werte im Stencil Buffer. Ist der Wert größer als null, so liegt der Pixel im Schatten. Dieser Prozess kann für beliebig viele Lichtquellen wiederholt werden.

Im Jahr 1996 deutete Diefenbach [DIEF96] auf ein Problem mit der Near- und Far-Clipping Plane hin, in dem ein Teil der Schattenvolumen durch eine der Planes abgeschnitten wird und das Zählen inkorrekt sei. Für das Near-Clipping Problem stellte er eine Lösung vor, welche in einigen besonderen Fällen jedoch immer noch fehlerhaft war.

Im Jahr 1999 stellten Bilodeau [BIL99] und ein Jahr später Carmack [CARM00] fest, dass die Formulierung der Tiefentests auch umgedreht funktioniert. Hierbei wird der Zähler für die Vorderseiten runter- und für die Rückseiten hochgezählt, wenn der Test fehlschlägt. Es wird bei diesem Ansatz so zu sagen von hinten nach vorne gezählt. Dieses Prinzip wird oft auch als Z-Fail bezeichnet, während der ursprüngliche Ansatz unter den Namen Z-Pass bekannt ist.

2003 veröffentlichten Everitt und Kilgard [KILG02] die erste robuste, fehlerfreie Umsetzung der Stencil Schattenvolumen.

3 Konzept

3.1 Grundkonzept der Schattenvolumen

Um die nachfolgenden Kapitel verstehen zu können, sollte zuerst einmal das Grundprinzip der Schattenvolumen erläutert werden. Die Idee ist, dass mit Hilfe der Schattenvolumen eine Schablone erzeugt wird, mit der bestimmt werden kann, ob der Farbwert von Bild A oder Bild B für einen bestimmten Pixel verwendet werden soll. Bild A stellt dabei die Szene komplett im Dunkeln dar, während Bild B dieselbe Szene mit Beleuchtung wiedergibt.

Der grobe Ablauf des Shadow Volume-Algorithmus sieht wie folgt aus. Zunächst muss die Szene komplett im Dunkeln gerendert werden. Es darf nur der ambiente Lichtanteil eingeschaltet sein. Anschließend werden für alle Lichtquellen der Szene, die resultierenden Schattenvolumen erzeugt (siehe 3.4) und in den Stencil-Buffer geschrieben, wo diese ausgewertet werden (siehe 3.2). Ist dieser Vorgang abgeschlossen enthält der Stencil-Buffer die benötigte Schablone. Zum Schluss wird die Szene erneut gerendert, dieses Mal mit eingeschalteter Lichtquelle. Es werden jedoch nur die Pixel im Framebuffer überschrieben, die nicht im Schatten liegen.

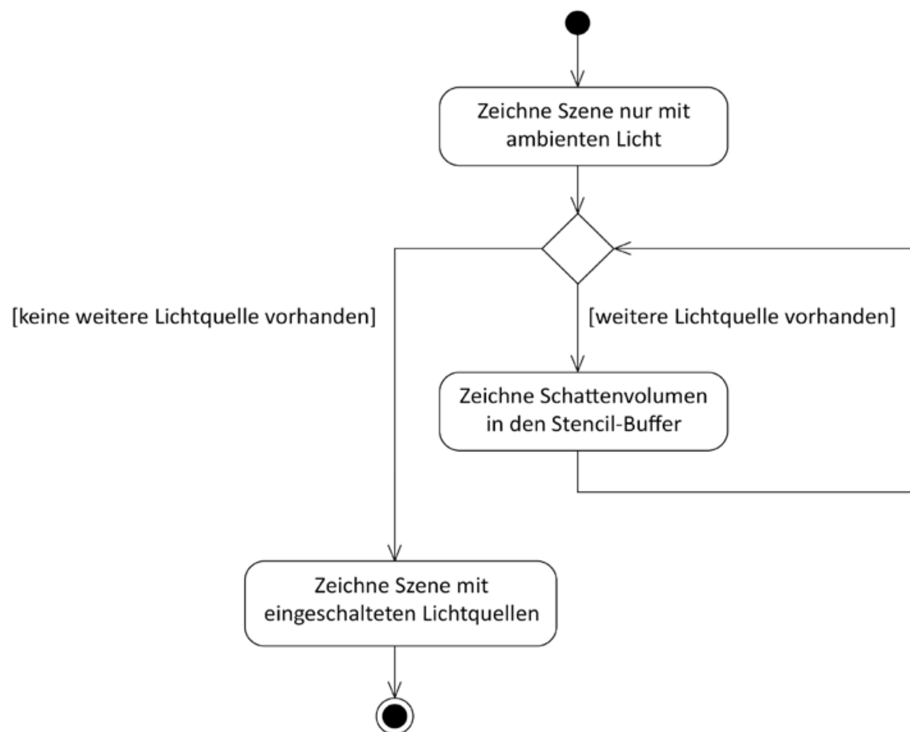


Abbildung 3.1: Grundliegender Ablauf des Shadow Volume-Algorithmus

3.2 Z-Pass und Z-Fail

Das Zählen bzw. Auswerten, ob ein Punkt im Schatten liegt, passiert bei modernen Ansätzen mit Hilfe des Stencil-Buffers. Wie bereits kurz in Kapitel 2.6 angesprochen, stellt der Stencil-Buffer dementsprechende Funktionen zur Verfügung. Anders als im originalen Ansatz von Crow wird in dem modernen Ansatz nicht für jeden Pixel ein Strahl entsandt, sondern ein Tiefentest durchgeführt. Während dessen wird im Stencil-Buffer gezählt, wie viele Schattenvolumen betreten und verlassen wurden bis der Tiefentest auf ein Objekt der Szene trifft. Dabei wird beim Betreten eines Volumens der Wert im Puffer inkrementiert und beim Verlassen dekrementiert. Konkret müssen für das Zählen zwei Durchgänge erfolgen. Im ersten Durchlauf werden mit Hilfe des „Face Cullings“ die Rückseiten und im zweiten die Vorderseiten der Schattenpolygone eliminiert. Der erste Durchlauf dient zur Überprüfung, wie viele Volumen betreten wurden. Hingegen wird im zweiten geprüft, wie viele wieder verlassen wurden. Beträgt das Endresultat null, wird der Punkt nicht von einem Objekt der Szene verdeckt. Bei allen anderen Werten befindet sich dieser im Schatten. Dieses Verfahren wird im Allgemeinen als Z-Pass bezeichnet. Der Ausdruck Pass (engl. für passieren/erfolgreich) kommt hierbei zu Stande, weil in diesem Ansatz jeweils gezählt wird, wenn der Tiefentest gelingt.

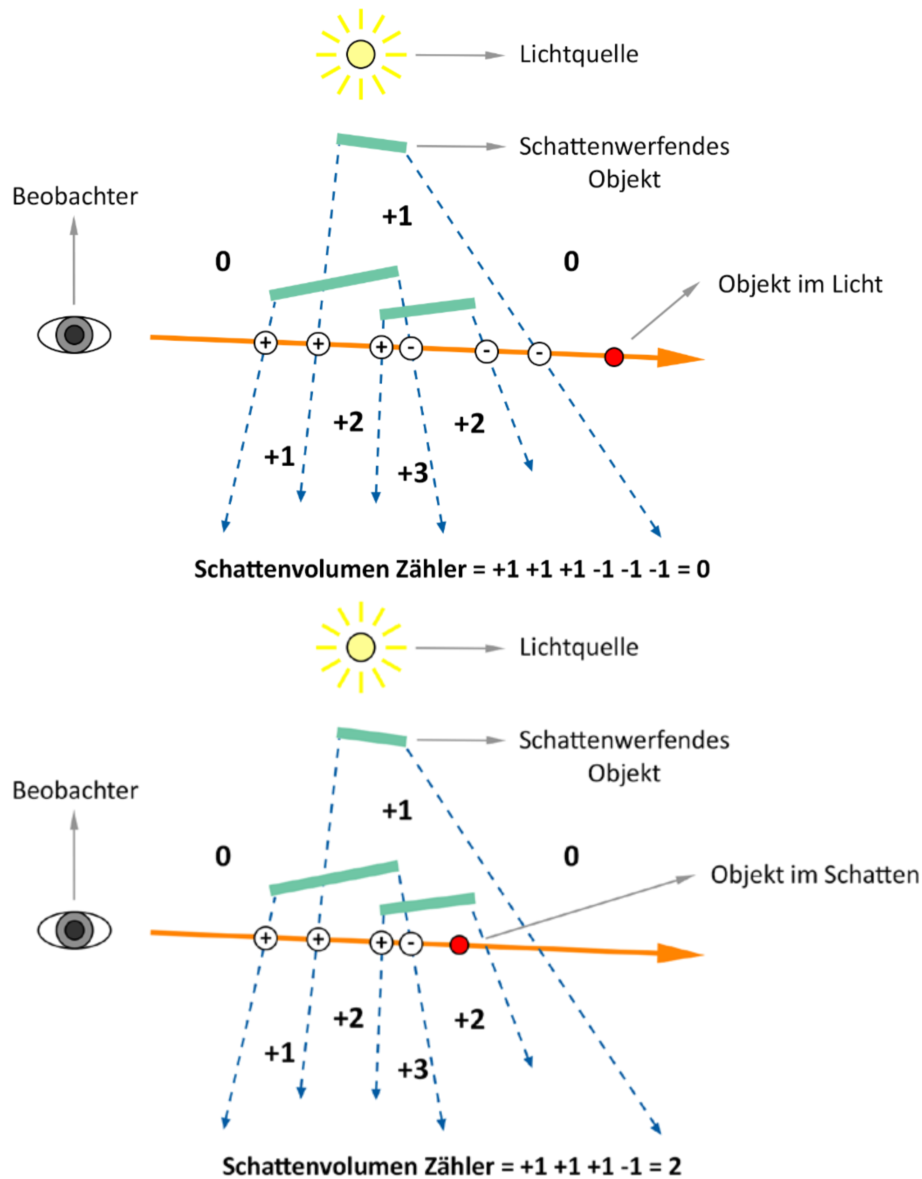


Abbildung 3.2: Bestimmung ob ein Punkt im Schatten liegt mit Z-Pass. Oben liegt dieser Außerhalb, da der Wert null beträgt. Unten befindet sich der Punkt im Schatten, da der Wert ungleich null ist. Nachempfunden von [KILG04, Seite 26, 27]

Das Z-Pass Verfahren funktioniert weitgehend korrekt, es sei denn der Beobachter selbst befindet sich in einem Schattenvolumen oder ein Teil eines Schattenvolumens wird von der Near-Clipping Plane abgeschnitten. Damit der Algorithmus in diesem Fall ebenfalls korrekt zählen kann, muss das sogenannte Z-Fail Verfahren verwendet werden.

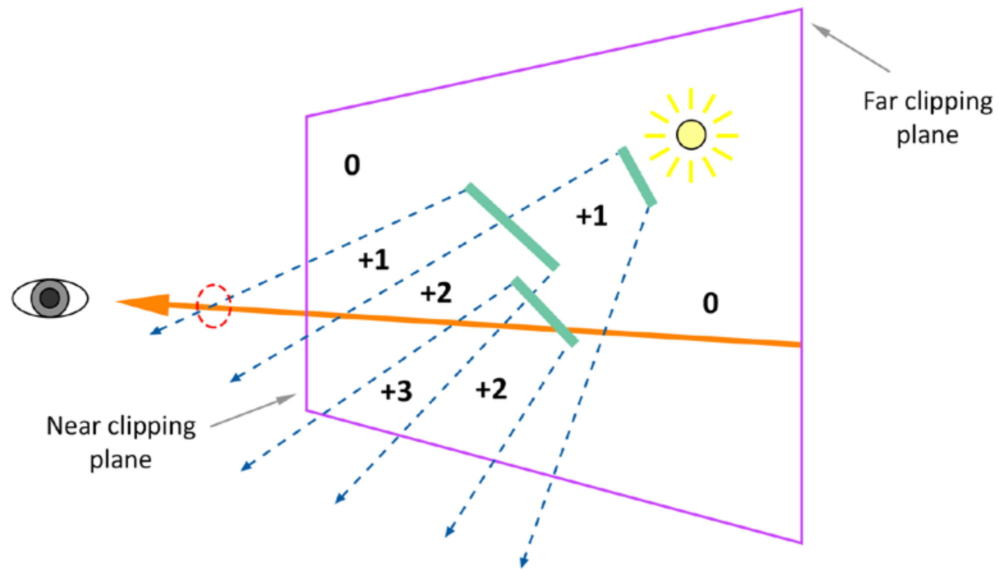


Abbildung 3.3: Z-Pass Zählproblem, wenn ein Schattenvolumen von der Near-Clipping Plane abgeschnitten wird. Nachempfunden von [KILG04, Seite 31]

Bei Z-Fail handelt es sich um die Umkehrung des Z-Pass Verfahren. Dies bedeutet, der Wert im Stencil-Buffer wird jedes Mal erhöht bzw. verringert, wenn der Tiefentest fehlschlägt. Ebenfalls anders durch die Umkehrung ist, dass für jedes Betreten einer Rückseite der Zähler erhöht und für jedes Verlassen der Vorderseiten verringert wird. Bildlich gesehen, wird bei dem Z-Pass von vorne nach hinten und beim Z-Fail von hinten nach vorne gezählt. Durch die Umkehrung wird das Problem mit der Near-Clipping-Plane auf die Far-Clipping-Plane verschoben. Eine detaillierte Beschreibung zur Behebung des Problems beim Z-Fail Verfahren ist in Abschnitt 3.3 geschildert.

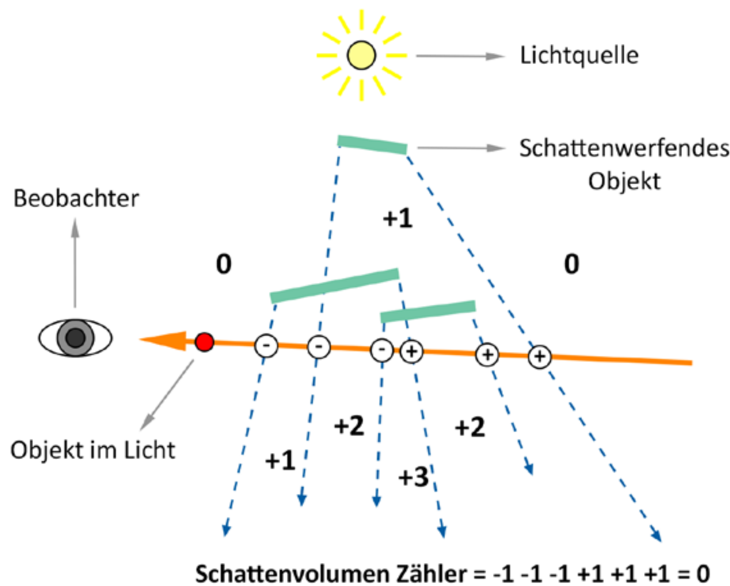


Abbildung 3.4: Auswertung der Schattenvolumen mit dem Z-Fail Verfahren, nachempfunden von [KILG04, Seite 35]

Das Z-Fail Verfahren ist zwar robuster, aber dafür auch langsamer. Das Zählen pro Pixel mit dem Z-Fail Verfahren dauert bereits länger. Besonders merkbar wird dies, bei Körpern mit großer Polygonanzahl, da das Z-Fail Verfahren ein komplett geschlossenes Schattenvolumen benötigt. Eine detaillierte Beschreibung zum Erstellen der Volumen ist in Kapitel 3.3 zu finden. Wichtig an dieser Stelle ist es zu wissen, dass die Volumen mit der Rückseite des Körpers geschlossen werden. Das bedeutet, umso mehr Polygone der Körper besitzt, umso mehr Polygone müssen zum Schließen der Rückseite des Volumens gezeichnet werden. Bei dem Z-Pass Verfahren ist das Schließen der Schattenvolumen hingegen nicht notwendig.

Die Grundidee ist, das Z-Fail Verfahren nur dann zu benutzen, wenn das Z-Pass Verfahren fehlerhaft zählt. Ein guter Ansatz zur Unterscheidung wurde 2003 von Morgan McGuire [MGU03] vorgestellt. McGuire wählt dabei für alle Körper in der Szene, das dementsprechend benötigte Verfahren aus. Zunächst wird eine Pyramide von dem Viewport zur Lichtquelle erstellt. Anschließend wird geprüft, ob der jeweilige Körper bzw. ein Teil des Körpers sich innerhalb dieser Pyramide befindet. Ist dies nicht der Fall, kann das effiziente Z-Pass Verfahren verwendet werden. Ansonsten muss auf das Z-Fail Verfahren umgeschaltet werden.

3.3 Eleminierung der Far-Plane

Wie bereits in Kapitel 2.6 kurz erläutert, müssen die Schattenvolumen ins Unendliche reichen. Ein großes Problem stellt dabei das Clipping dar. Um nun das Abschneiden der Volumen in der Ferne zu verhindern, wird in der Regel die Far-Clipping-Plane in die „Unendlichkeit“ verschoben. Wird ein Teil des Volumens abgeschnitten, so treten Fehler beim Zählen auf, wie in Abbildung 3.5 gezeigt wird. Der in dieser Arbeit erstellte Prototyp verwendet ebenfalls diese Technik.

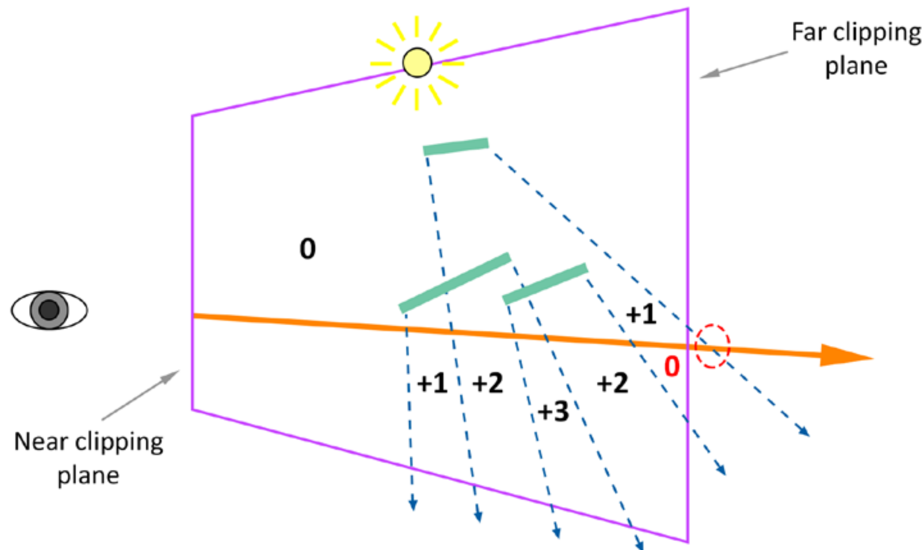


Abbildung 3.5: Ein Schattenvolumen wurde von der Far-Clipping Plane abgeschnitten. Der ausgehende Strahl durchläuft von daher nicht das letzte Schattenvolumen. Nachempfunden von [KILG04, Seite 36]

Das Near- und Far-Clipping wird über die Projektionsmatrix gesteuert.
Zunächst sollte erst einmal ein Blick auf die normale Projektionsmatrix geworfen werden.

$$\mathbf{P} = \begin{bmatrix} \frac{2 \times Near}{Right - Left} & 0 & \frac{Right + Left}{Right - Left} & 0 \\ 0 & \frac{2 \times Near}{Top - Bottom} & \frac{Top + Bottom}{Top - Bottom} & 0 \\ 0 & 0 & -\frac{Far + Near}{Far - Near} & -\frac{2 \times Far \times Near}{Far - Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Wie hier bereits entnommen werden kann, taucht die Far-Clipping-Plane (Far) nur in Reihe drei und vier auf. Wird der Wert, nach Blinn [BLI93], für die Far-Clipping-Plane nun auf unendlich gesetzt, erhält man folgende Matrix.

$$\lim_{Far \to \infty} \mathbf{P} = \mathbf{P}_{\mathbf{inf}} = \begin{bmatrix} \frac{2 \times Near}{Right - Left} & 0 & \frac{Right + Left}{Right - Left} & 0 \\ 0 & \frac{2 \times Near}{Top - Bottom} & \frac{Top + Bottom}{Top - Bottom} & 0 \\ 0 & 0 & -1 & -2 \times Near \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Da es sich bei Projektionsmatrix, sowie den Vertices um homogene Koordinaten handelt, reicht es aus, die w-Koordinate auf null zu setzten. Dies bezweckt, dass der Vertex mit P_{inf} ins unendliche projiziert wird.

Beim Verwenden der unendlichen Projektionsmatrix, anstelle der „normalen“ Projektionsmatrix, verliert der Tiefenpuffer geringfügig an Genauigkeit. Dies kann jedoch in den meisten Anwendungsfällen vernachlässigt werden. Als Beispiel, bei einer Near-Plane von 1 und einer Far-Plane von 100, würde der Tiefenpuffer ca. 1% seiner Genauigkeit verlieren, wenn die Far-Plane auf Unendlich gesetzt wird.

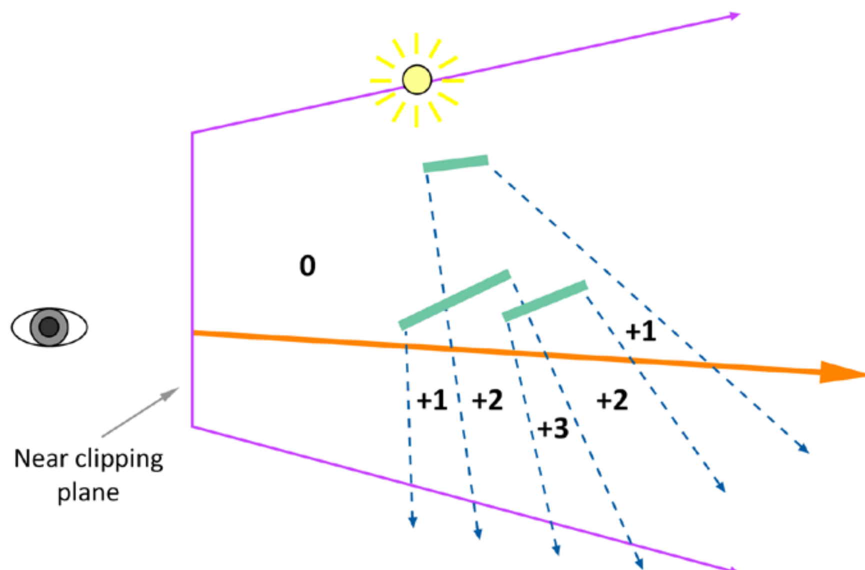


Abbildung 3.6: Zählen der Schattenvolumen mit der Projektionsmatrix $\mathbf{P}_{\mathbf{inf}}$, nachempfunden von [KILG04, Seite 37]

3.4 Unendliche Schattenvolumen

Die Idee zur Generierung von Schattenvolumen besteht darin, ein Volumen von der Rückseite eines Objektes in die „Unendlichkeit“ zu projizieren. Bei der Rückseite handelt es sich um die Seite des Objektes, die vom Licht weg zeigt. Bei diesem Körper muss es sich um ein geschlossenes Modell handeln. Sollte dieses Risse aufweisen, so entstehen Fragmente in der Luft und der Schatten selbst ist unvollständig.
Um nun ein komplettes Schattenvolumen zu bilden, werden folgende Polygone benötigt: Alle Polygone der Vorderseite des Körpers für den Boden des Schattenvolumens. Dann die Seiten die vom Boden aus in die Unendlichkeit reichen sollen, sowie bei dem Z-Fail Verfahren benötigten „Deckel“, welcher das Volumen in der Unendlichkeit schließt. Bei dem „Deckel“ handelt es sich letztendlich um die Rückseite des Körpers, die in die Unendlichkeit projiziert werden soll. Es wird im folgenden Text davon ausgegangen, dass es sich bei den Polygonen des Models um Dreiecke handelt.

Die Bodenpolygone lassen sich sehr einfach bestimmen. Wie bereits erwähnt, handelt es sich hierbei um alle Polygone, die die Vorderseite des Körpers bilden. Um diese zu finden wird geprüft, welche Polygone des Körpers zum Licht zeigen. Deren Vertices werden genommen und erneut gezeichnet.

$$A = \{A_X, A_Y, A_Z, A_W\}$$
$$B = \{B_X, B_Y, B_Z, B_W\}$$
$$C = \{C_X, C_Y, C_Z, C_W\}$$

Nachdem alle Vorderseitenpolygone gezeichnet wurden, ist eine geschlossene Oberfläche entstanden.

Um die Flächen bzw. Seiten des Volumens zeichnen zu können, wird zunächst die Silhouette des Körpers oder genauer gesagt, die möglichen Silhouettenkanten benötigt. Eine mögliche Silhouettenkante ist eine Kante, bei der eines der anliegenden Dreiecke vom Licht weg- und das andere zum Licht hinzeigt. Eine genauere Beschreibung zur Findung von Silhouettenkanten wird in Kapitel 3.4 erläutert. Bei den Seiten handelt es sich um Rechtecke, also um vier Vertices. Pro Rechteck sind die ersten zwei Vertices bereits durch die Silhouettenkante gegeben, nämlich A und B. Bei den beiden fehlenden Vertices C und D handelt es sich um die Differenzen A-L und B-L. Damit diese korrekt in die Unendlichkeit projiziert werden, muss wie in Kapitel 3.3 beschrieben, die w-Koordinate auf null gesetzt werden. Daraus ergeben sich dann folgende vier Vertices:

$$B = \{B_X, B_Y, B_Z, B_W)$$
$$A = \{A_X, A_Y, A_Z, A_W\}$$
$$D = \{A_X L_W - L_X A_W, A_Y L_W - L_Y A_W, A_Z L_W - L_Z A_W, 0\}$$
$$C = \{B_X L_W - L_X B_W, B_Y L_W - L_Y B_W, B_Z L_W - L_Z B_W, 0\}$$

Der “Deckel” lässt sich ähnlich wie bereits der Boden bestimmen. Anders als beim Boden werden hier die Polygone benötigt, welche von der Lichtquelle wegzeigen. Da der „Deckel“ das Volumen in der Unendlichkeit schließen soll, muss dieser ebenfalls, wie bereits die beiden Vertices C und D, bei den Seiten in die Unendlichkeit projiziert werden. Das bedeutet die neuen Vertices A, B und C bilden sich ebenfalls aus den Differenzen zwischen den drei Vertices des Polygons und der Position des Lichts L.

$A = \{A_X L_W - L_X A_W, A_Y L_W - L_Y A_W, A_Z L_W - L_Z A_W, 0\}$
$B = \{B_X L_W - L_X B_W, B_Y L_W - L_Y B_W, B_Z L_W - L_Z B_W, 0\}$
$C = \{C_X L_W - L_X C_W, C_Y L_W - L_Y C_W, C_Z L_W - L_Z C_W, 0\}$

3.5 Bestimmung von Silhouettenkanten

Wie bereits in Kapitel 3.3 erwähnt, ist es wichtig die Silhouette des schattenwerfenden Körpers zu bestimmen, da es ansonsten nicht möglich ist zwischen Vorder- und Rückseite zu unterscheiden. Die Operation der Bestimmung der Kanten ist sehr zeitaufwändig und sollte gut optimiert sein, da dieser Vorgang auf der CPU und nicht auf der GPU stattfindet.
Wie ebenfalls in Kapitel 3.3 erwähnt, müssen die Kanten gefunden werden, bei den eine Seite zum und die andere vom Licht weg zeigt. Es lässt sich relativ einfach bestimmen, ob ein Dreieck zum Licht zeigt oder nicht. Zunächst muss der Richtungsvektor L, der vom Dreieck zum Licht zeigt bestimmt werden. Für die Berechnung kann ein beliebiger Vertex des Dreiecks verwendet werden. Anschließend muss das Skalarprodukt N•L gebildet werden, wobei N die Dreiecksnormale ist. Ist das Skalarprodukt der beiden Vektoren grösser als null, zeigt das Dreieck zum Licht, ansonsten davon.

Ein Algorithmus zur Erkennung von Silhouettenkanten wurde 2002 von Hun Yen Kwoon [YEN02] im Artikel „The Theory of Stencil Shadow Volumes“ vorgestellt. Der Ablauf sieht wie folgt aus:

Algorithm 1 Silhouette Determination (Yen Kwoon)

```
1: Initialize edge list
2: for all triangles t in mesh m do
3:   if t not faces the light then
4:     for all edges e of t do
5:       if edge list contains e or its reverse then
6:         remove e from edge list
7:       else
8:         add e to edge list
9:       end if
10:    end for
11:  end if
12: end for
```

Es werden alle Dreiecke betrachtet und geprüft, ob diese zum Licht zeigen. Sollte dies der Fall sein, so werden nach Yen Kwoon alle Kanten des Dreiecks einer Liste hinzugefügt. Anschließend wird ebenfalls noch geprüft, ob die drei neuen Kanten bzw. die Umkehrung bereits in der Sammlung der Silhouettenkanten vorhanden sind und gegebenenfalls entfernt. Durch den letzten Schritt werden alle redundanten Kanten entfernt. Dies hat zur Folge, dass nur noch die eigentlichen Silhouettenkanten verbleiben. Dieses Vorgehen funktioniert, da bei einem geschlossenen Modell logischer Weise jede Kante genau von zwei Dreiecken benutzt wird. Ein Vorteil des Algorithmus ist, dass die Prozedur nur einmal durchlaufen werden muss. Ein großes Makro an Yen Kwoons Algorithmus ist jedoch das Entfernen redundanter Kanten. Da bei jedem Hinzufügen einer Kante, die Kantenliste immer wieder durchsucht wird, kann dies bei besonders polygonreichen Modellen sehr kostspielig werden.

Eric Lengyel [LENG02] hatte einen weiteren Ansatz in seinem Artikel „The Mechanics of Robust Stencil Shadows“ vorgestellt. Anders als Yen Kwoon durchläuft dieser zweimal alle Dreiecke eines Körpers. Des Weiteren arbeitet Lengyels Ansatz nur mit der Dreiecksliste und deren Vertices. Hierbei werden die Vertices gegen den Uhrzeigersinn durchlaufen und die Kanten erzeugt.
Der grundlegende Algorithmus sieht wie folgt aus: beim ersten Durchlauf werden alle Kanten erzeugt und das jeweilige Dreieck, bei dem die Kante erzeugt wurde, dieser hinzugefügt. Im zweiten Durchlauf, wird das Nachbardreieck, welches die jeweilige Kante mit dem hinterlegten teilt, zu geordnet. Erst zum Schluss überprüft Lengyel, ob es sich um eine Silhouettenkante handelt. Bedingung hierfür ist, dass ein Dreieck vom Licht weg und das andere zum Licht zeigt.

Kilgards Ansatz [KILG02] zur Bestimmung der Silhouettenkanten ist relativ simpel gehalten. Dieser setzt voraus, dass jedes Dreieck seine Kanten und dessen Nachbarn kennt. Der Algorithmus sieht wie folgt aus:

Algorithm 1 Silhouette Determination (Kilgard)

```
1: Initialize edge list
2: for all triangles t in mesh m do
3:   if t not faces the light then
4:     for all edges e of t do
5:       if neighbour triangle faces the light then
6:         add e to edge list
7:       end if
8:     end for
9:   end if
10: end for
```

Die Dreiecksliste wird wie bei Yen Kwoon nur einmal durchlaufen. Dabei überprüft Kilgard nur die Dreiecke die vom Licht weg zeigen, damit keine Duplikate entstehen. Anschließend wird geprüft, ob es einen Nachbarn gibt, der zum Licht zeigt. Sollte ein solches Dreieck gefunden werden, so handelt es sich bei der jeweiligen Kante um eine Silhouettenkante.
Wie hier deutlich wird, speichert Kilgard mehr Informationen ab, um eine bessere Performance zu erhalten.

Der in dieser Arbeit entstandene Algorithmus zur Erkennung von Silhouettenkanten ist ebenfalls sehr simpel und performanceorientiert gehalten. Es wird davon ausgegangen, dass alle Kanten eines Körpers bekannt sind. Des Weiteren müssen die Kanten beide anliegende Dreiecke kennen. Der Algorithmus selbst läuft lediglich nur einmal über alle Kanten hinweg und prüft, ob eines der Dreiecke zum Licht und das andere vom Licht zeigt. Ist diese Bedingung erfüllt wurde eine Silhouettenkante gefunden. Eine detailliertere Beschreibung zur Umsetzung und der Erzeugung der Kanten folgt in Implementierungskapitel 4.3.5.

3.6 Weiche Schatten

Alle bisher erläuterten Algorithmen erzeugen sogenannte harte Schatten. Bei einem harten Schatten gibt es lediglich nur „ist im Schatten“ oder „ist nicht im Schatten“. Es gibt keine Zwischenzustände. In der Realität gibt es diese Art von Schatten nicht, dort kommen nur weiche Schatten vor. Dies liegt daran, dass in der Computergrafik mit Punktlichtquellen gearbeitet wird, während es sich in der Realität immer um Flächen handelt, die das Licht ausstrahlen.

Ein weicher Schatten unterteilt sich in zwei Bereiche: dem Umbra- und Penumbrabereich, letzterer auch als Halbschatten bekannt. Als Umbrabereich wird die Fläche bezeichnet, die komplett von einem Objekt verdeckt ist. Der Penumbrabereich beschreibt die Fläche bei der immer noch etwas Licht hinkommt. Also den Übergang von hell nach dunkel.

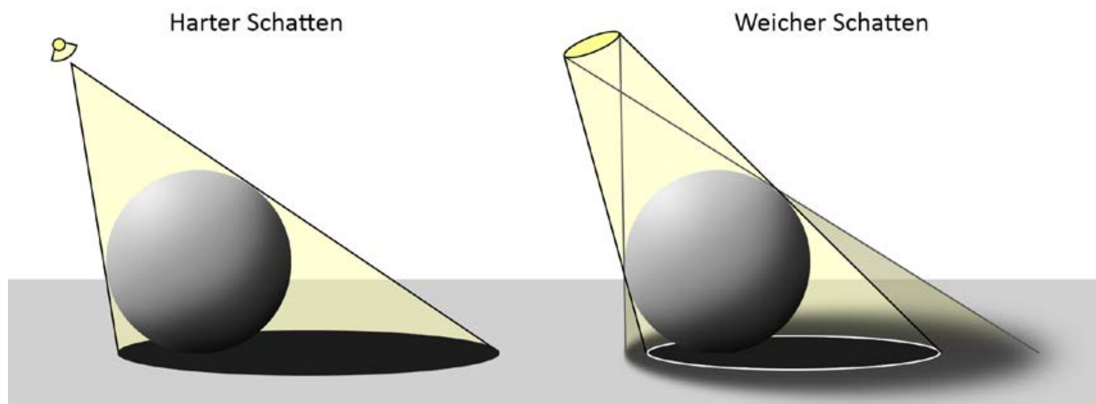


Abbildung 3.7: Unterschied eines harten und weichen Schattens. Der Umbrabereich ist in der rechten Abbildung mit einer weißen Umrandung hervorgehoben.

In der Computergrafik gibt es mehrere Ansätze, weiche Schatten in eine Szene zu bringen. Ein relativ simpler Ansatz ist, mehrere Lichtquellen sehr nah nebeneinander zu positionieren. Bei dem Prototyp in dieser Arbeit wurde dieser Ansatz ebenfalls gewählt mit ein paar kleinen Änderungen. Da wie bereits in Kapitel 3.3 erwähnt, die Schattenvolumen in Beziehung zur Position der Lichtquelle erzeugt werden, wird im Prototyp lediglich die Position der Lichtquelle mehrfach verschoben und die Prozedur zum Rendern der Schattenvolumen demensprechend oft wiederholt. Mit dieser Technik lassen sich glaubwürdige Bilder erzeugen, wenn der Abstand zur ursprünglichen Lichtquelle richtig gewählt wurde. Sollte dieser zu groß sein, wirkt die Szene unglaubwürdig und wenn dieser zu klein gewählt ist, unterscheiden sich die generierten Schatten nur sehr geringfügig von den harten Schatten. Des Weiteren ist dieser Ansatz eher ungeeignet für Echtzeitanwendung bzw. größere Szenen mit mehreren Lichtquellen, auf Grund des hohen Rechenaufwands.

Ein effizienter Algorithmus zur Berechnung von weichen Schatten wurde von Thomas Akenine-Moeller und Ulf Assarsson [ASSA03] im Jahr 2002 vorgestellt, welcher unter dem Namen „Penumbra Wedges" bekannt ist. In späteren Veröffentlichungen stellte Assarsson [ASSA03] verbesserte Varianten seines Ursprungsalgorithmus vor. Alle Algorithmen basieren jedoch auf das von Everitt und Kilgard [KILG02] vorgestellte Grundprinzip. Assarsson versucht in seinem Algorithmus aus einer Punktlichtquelle eine Art Strahler zu simulieren. Zunächst wird mit Hilfe des Shadow Volume-Ansatzes von Everitt und Kilgard der Umbrabereich generiert. Anschließend werden entlang der Silhouettenkanten Penumbra-Wegdes gebildet, die den Halbschattenbereich definieren. Bei den Wedges handelt es sich um keilförmige Körper, bei denen die demensprechende Silhouettenkante die Spitze darstellt. Zum Schluss werden die genauen Halbschattenwerte pro Pixel bestimmt. Assarsson benutzt hierfür einen weiteren Puffer, den sogenannten Visibility-Buffer (dt. Sichtbarkeitspuffer). Dieser enthält dann nach Beendung des Prozesses den weichen Schatten.

4 Implementierung

4.1 Das Framework

4.1.1 Allgemeines

Als Grundlage für die Implementierung des Stencil Shadow Volume Algorithmus wurde das Computer Graphics Research (kurz CG Research) Framework der HAW-Hamburg verwendet. Das Framework selbst basiert auf der Programmiersprache Java und benutzt OpenGL als Grafikkartenschnittstelle. Als Grunddatenstruktur innerhalb des Frameworks wird ein sogenannter Szenengraph verwendet. Das bedeutet, für jede Operation (z.B. Transformationen) und für alle Objekte (z.B. Triangle Meshes) wird ein Knoten in dem Graphen eingefügt.

Die Grundidee des Szenengraphen lässt sich am besten an Hand eines Beispiels erklären. Angenommen man möchte ein Auto rendern, dann gibt es einen Knoten (engl. Node) der die Karosserie enthält. Die Tür, das Dach sowie der Kotflügel des Autos werden nun als Unterknoten der Karosserie hinzugefügt. Möchte man das Auto nun aus der Szene entfernen, reicht es lediglich die Karosserie „auszuschalten“. Bei einer platten Repräsentation müssten ebenfalls die Tür, das Dach sowie der Kotflügel deaktiviert werden. Das Beispiel noch einmal kurzgefasst: Wenn die Karosserie nicht gerendert werden soll, macht es wenig Sinn die Tür oder das Dach des Autos zu rendern.
Ein wichtiger Bestandteil des Szenengraphen in dem Framework ist die Root-Node (dt. Wurzelknoten). Bei der Root-Node handelt es sich um einen speziellen Knoten, der wie der Name bereits aussagt, die Wurzel des Graphen repräsentiert. Der Knoten enthält außerdem wichtige Flags (z.B. ob Schatten verwendet werden sollen) und die Lichtquellen der Szene. Hinzukommt, dass die Root-Node der Applikation bekannt ist und als Einstiegspunkt für Veränderungen am Szenengraphen dient.

4.1.2 Aufbau

Der grundliegende Aufbau des Frameworks ist in Abbildung 4.1 dargestellt. Dabei wird im groben zwischen den *Apps*, dem *Graphics Core* und den *JOGL/JME3-Renderern* unterschieden. Unter *Apps* befinden sich die ausführbaren Demo-Klassen. In der *Graphics Core*-Komponente befinden sich die Datenstrukturen zur Repräsentation der 3D-Objekte, während die *Render*-Komponenten die Schnittstelle zur Grafikhardware darstellen. Die Datenstrukturen selbst sind zunächst unabhängig von den Renderer-Klassen. Ein Umstieg auf z.B. Direct3D wäre damit möglich. Das Framework bietet zwei verschiedene Rendermethoden: zum einen JMonkeyEngine3 und zum anderen JOGL (Java OpenGL). Für die Implementierung der Schattenvolumen wurde der JOGL-Renderer benutzt. Von der Applikation wird der *Renderer3D* aufgerufen, welcher den Szenengraphen rekursiv abarbeitet. Der *Renderer3D* wiederum ruft die dementsprechenden Renderer für die jeweiligen Nodes auf (bspw. *RenderTriangleMesh* für die Trinagle Mesh-Datenstruktur).

Für den Stencil Shadow Volume Algorithmus stellt diese Repräsentation eine Herausforderung dar. Anders als in einer platten Repräsentation, wo alle Informationen bekannt sind, muss das Erzeugen und Auswerten der Schattenvolumen für jede Datenstruktur explizit in dessen Renderer implementiert werden. Da es sich bei der Implementierung des Algorithmus in dieser Arbeit nur um einen Prototyp handelt, wurde zunächst nur die Triangle Mesh-Datenstruktur vollständig implementiert.

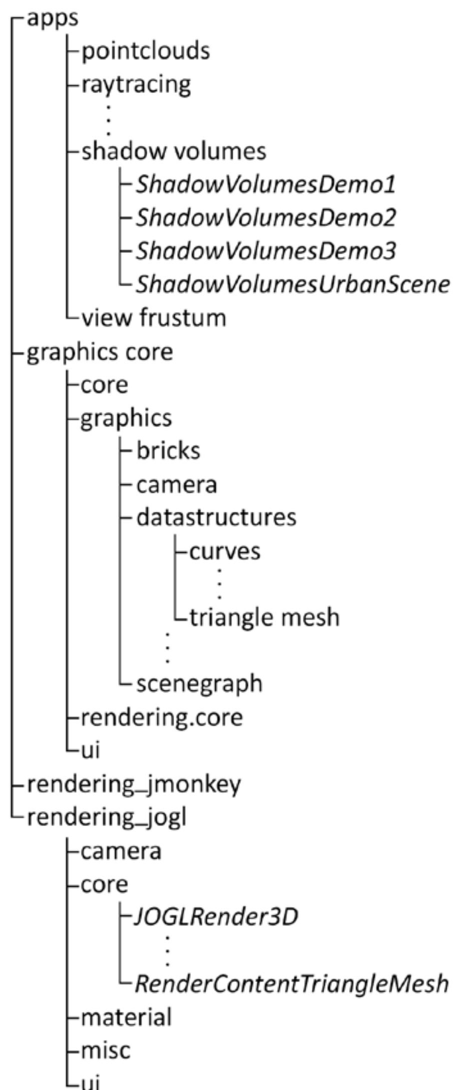


Abbildung 4.1: Grundliegender Aufbau des Frameworks mit dessen Komponenten (ohne *SmartHome*) und die wichtigsten *Klassen*

4.2 Grundaufbau des Algorithmus

Dieses Kapitel befasst sich mit dem Grundgerüst des Algorithmus im *Render3D*. Im *Renderer3D* wird am Anfang zwischen zwei Renderabläufen unterschieden: Ein „normaler" Ablauf ohne Schatten und der andere mit. Der gewünschte Ablauf kann an Hand eines globalen Flags *drawShadows* in der Root-Node von der Applikation übergeben werden.

Der Ablauf welcher ebenfalls Schatten in die Szene mit einzeichnet sieht wie folgt aus:

Algorithm 1 Renderer3D drawWithShadows()

```
1:  clearBuffers()
2:  setupPinf()
3:  for all lightsources do
4:    disableLightSource()
5:  end for
6:  drawScene()
7:  for all lightsources do
8:    drawShadowVolumes(lightsource)
9:    if shadow is soft then
10:     generateAdditionalLights()
11:     for all additional lights do
12:       drawShadowVolumes(additional lightsource)
13:     end for
14:   end if
15: end for
```

Zu Beginn des Algorithmus wird der Farb- und Tiefenpuffer zurückgesetzt. Des Weiteren wird die Far-Clipping Plane mit Hilfe der Projektionsmatrix Pinf eliminiert (Zeile 1,2). Anschließend wird die Szene nur mit ambienten Licht gezeichnet (Zeile 3-6). Dazu muss, wie in Kapitel 3.1 angesprochen, der diffuse und spekulare Lichtanteil jeder Lichtquelle deaktiviert werden. Dies geschieht in dem Aufruf `disableLightSource()` in der Zeile 4. Nachdem die Szene komplett im Dunkeln gerendert wurde, werden die Schattenvolumen für alle Lichtquellen in die Szene gezeichnet und ausgewertet (Zeile 7-15). Das Zeichnen, sowie das Auswerten geschehen in der `drawShadowVolumes()`-Methode. An dieser Stelle ist anzumerken, dass wenn der Two-Sided-Stencil-Buffer (dt. doppelseitiger Stencil-Buffer) nicht zum Auswerten verwendet wird, bei der `drawShadowVolumes()`-Methode eine Kopie der Lichtquelle übergeben werden muss. Bei dem Two-Sided-Stencil-Buffer werden beide Tests, also das Betreten und Verlassen der Volumen, bei nur einem Renderdurchgang durchgeführt. Wird dieser nicht verwendet, müssen für beide Tests die Volumen gerendert werden. In diesem Fall wird eine Kopie der Lichtquelle benötigt, da sich zwischen den beiden Tests die Position der Lichtquelle ändern kann, durch z.B. einer Animation. Zum Schluss wird noch geprüft, ob ein weicher Schatten erzeugt werden soll (Zeile 9-14). Es wird hier zwischen fünf Lichtarten für weiche Schatten unterschieden: ein harter Schatten (also keine weiteren Lichtquellen), eine Lichtfläche auf der X, Y oder Z-Koordinate, sowie eine kugelförmige Anordnung der Lichter. Für die Flächen werden jeweils acht und für die Lichtkugel 14 weitere Lichtquellen hinzugefügt (siehe Abbildung 4.2). Die Lichtart kann in der Applikation für jede einzelne Lichtquelle individuell eingestellt werden. In der Methode `generateAdditionalLights()` werden eine Reihe von zusätzlichen Lichtern erzeugt, um die jeweilige Lichtart darstellen zu können. Damit die fertige Szene nicht zu hell ist, wird

der diffuse, sowie der spekulare Anteil aller generierter Lichtquellen durch deren Anzahl dividiert. Anschließend werden die weiteren Schattenvolumen für zusätzliche Lichtquellen eingezeichnet und ausgewertet (Zeile 11-13).

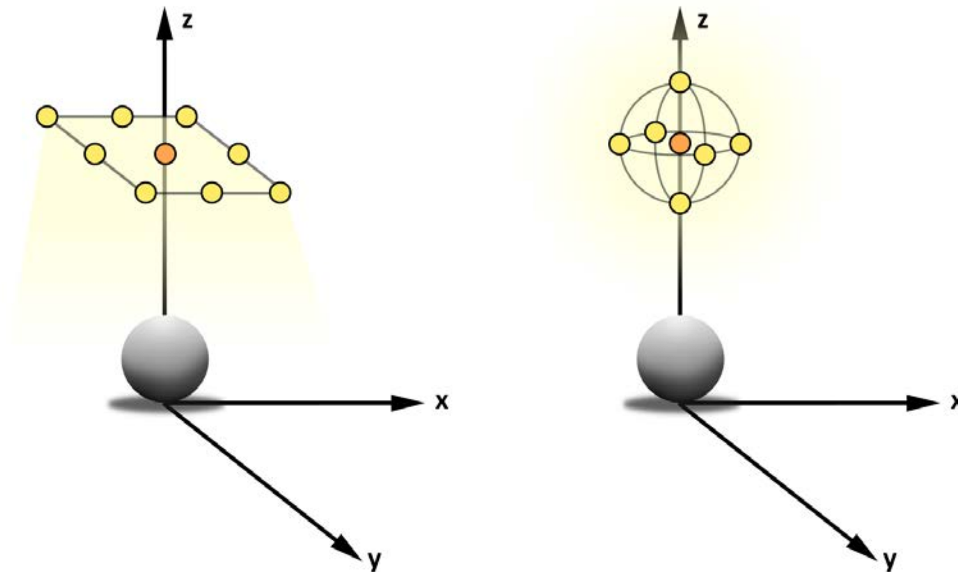


Abbildung 4.2: Weiche Schatten durch Hinzufügen weiterer Lichtquellen. Links eine Fläche bei der alle Lichtquellen die Z-Koordinate teilen und rechts (leicht vereinfacht) die kugelförmige Anordnung. Die ursprüngliche Lichtquelle ist dabei orange und die zusätzlichen gelb markiert.

Algorithm 1 Renderer3D drawShadowVolumes()

1: clearStencilBuffer()
2: disableColorWrites()
3: Setup stencil buffer for counting
4: generateShadowVolumes(lightSource)
5: enableLightSource()
6: enableColorWrites()
7: Setup stencil buffer to only render pixels with a value of 0
8: drawScene()

In der `drawShadowVolumes()`-Methode werden die Schattenvolumen generiert und ebenfalls ausgewertet. Zu Beginn wird der Stencil-Buffer zurückgesetzt und das Schreiben in den Frame-Buffer abgeschaltet (Zeile 1,2). Das Abschalten des Schreibens in dem Frame-Buffer ist notwendig, damit die Schattenvolumen selbst in der fertigen Szene nicht sichtbar sind.

Anschließend wird der Stencil-Buffer initialisiert. Da jede moderne Grafikkarte die Two-Sided-Stencil-Buffer Erweiterung unterstützt, wird ausschließlich nur diese für eine bessere Performance verwendet. Das weiter oben angesprochene Problem bezüglich einer Kopie der Lichtquelle, kann hiermit vernachlässigt werden.
Danach werden die Schattenvolumen erzeugt (Zeile 4). Hierfür wird der Szenengraph rekursiv abgearbeitet. Das eigentliche Generieren der Volumen sowie das Zählen und Auswerten findet in den jeweiligen Renderern der Datenstrukturen statt. Damit die Prozedur in den jeweiligen Renderern korrekt funktioniert, müssen während des rekursiven Abarbeitens des Szenengraphen gefundene Transformationen mit weitergereicht werden. Das ist wichtig, da die Renderer nur das Objekt selbst kennen, aber nicht dessen Transformation in der Szene. Während der Erzeugung der Schattenvolumen in den einzelnen Renderern werden die Vertices dann mit der mitgereichten Transformation multipliziert. Die Multiplikationen der einzelnen Vertices wurden zur besseren Lesbarkeit des Pseudocodes in dem Kapitel 4.3 weggelassen. Des Weiteren erwartet die `generateShadowVolumes()`-Methode eine Lichtquelle, sowie die Kollisionspyramide, die sich von der Near-Clipping-Plane zur Lichtquelle erstreckt. Diese ist zur Bestimmung des Zählverfahrens nötig. Die Pyramide wird in Form von Vertices übergeben.
Nachdem alle Schatteninformationen in dem Stencil Buffer eingetragen wurden, wird die Szene erneut gerendert (Zeile 5-8). Dieses Mal jedoch mit eingeschalteter Lichtquelle. Des Weiteren wird der Stencil Buffer so eingestellt, dass nur Pixel gerendert werden, die sich nicht im Schatten befinden. Also bei denen der Wert im Stencil Buffer null beträgt (Zeile 7).

4.2.1 Erzeugung der Kollisionspyramide

Wie bereits angesprochen, wird die Kollisionspyramide von der Near-Clipping-Plane zur Lichtquelle benötigt. Die Position der Lichtquelle ist bereits gegeben. Problematisch ist jedoch die Bestimmung der vier Eckpunkte der Near-Clipping-Plane. Die Idee ist, aus den vom Framework bereits gegebenen Informationen die Eckpunkte der Near-Clipping-Plane zu bestimmen. Folgende Werte sind von der Kamera gegeben: die Position der Kamera c_p, der Referenzpunkt zu dem die Kamera ausgerichtet ist c_{ref}, der senkrechte Öffnungswinkel sowie der Vektor der nach Oben zeigt c_{up}. Der Ablauf zur Berechnung der Eckpunkte sieht dabei wie folgt aus: Als erstes wird der Richtungsvektor d_{ref} benötigt, welcher die Distanz zwischen Kamera und dem Referenzpunkt beschreibt.

$$\vec{d}_{ref} = \vec{c}_{ref} - \vec{c}_{pos}$$

Anschließend wird der Mitelpunkt n_m bestimmt, welcher sich aus der Addition von c_p und d_{near} ergibt.

$$\vec{n}_m = \vec{c}_{pos} + \vec{d}_{near}$$

Wobei d_{near} die Distanz zwischen Kamera und der Near-Clipping-Plane beschreibt.

$$\vec{d}_{near} = \vec{d}_{ref} \cdot \frac{Near\ Clipping\ Plane\ distance}{normalize(\vec{d}_{ref})}$$

Im nächsten Schritt werden die halbe Höhe sowie die halbe Breite der Near-Clipping-Plane bestimmt. Für die Breite wird ebenfalls noch der Richtungsvektor c_{right} benötigt, welcher sich sehr einfach mit Hilfe des Kreuzproduktes aus d_{ref} und c_{up} herleiten lässt. Des Weiteren wird das sogenannte Field of View in X-Richtung benötigt, welches sich wie folgt berechnen lässt:

$$fovX = 2 \cdot \arctan(fovY \cdot 0.5) \cdot aspect\ ratio$$

Für die Höhe wird das Field of View in Y-Richtung benötigt, dies ist jedoch bereits durch den Öffnungswinkel der Kamera gegeben.

$$halfHeight = \tan(\frac{fovY}{2}) \cdot NCP\ distance$$

$$halfWidth = \tan(\frac{fovX}{2}) \cdot NCP\ distance$$

Nachdem die Längen berechnet wurden, müssen nun die Richtungsvektoren erzeugt werden, welche in die dementsprechenden Richtungen *Links*, *Rechts*, *Oben* und *Unten* zeigen. Um diese zu erhalten, werden die Richtungsvektoren c_{up} und c_{right} mit den zuvor berechneten Halblängen *halfHeight* und *halfWidth* multipliziert. Die beiden Vektoren *Links* und *Unten* resultieren dabei aus der Umkehrung von *Rechts* und *Oben* (Multiplikation mit -1).

$$\vec{d}_{up} = \vec{c}_{up} \cdot halfHeight$$

$$\vec{d}_{down} = -1 \cdot \vec{d}_{up}$$

$$\vec{d}_{right} = \vec{c}_{right} \cdot halfWidth$$

$$\vec{d}_{left} = -1 \cdot \vec{d}_{right}$$

Mit Hilfe der Richtungsvektoren und dem Mittelpunkt n_m können nun die vier Eckpunkte *Unten Links, Oben Links, Unten Rechts sowie Oben Rechts* bestimmt werden.

$$\vec{n}_{ll} = \vec{h}_{down} + \vec{w}_l + \vec{n}_m$$

$$\vec{n}_{ul} = \vec{h}_{up} + \vec{w}_l + \vec{n}_m$$

$$\vec{n}_{lr} = \vec{h}_{down} + \vec{w}_r + \vec{n}_m$$

$$\vec{n}_{ur} = \vec{h}_{up} + \vec{w}_r + \vec{n}_m$$

Diese Berechnung muss jedes Mal wiederholt werden, wenn sich die Position oder der Referenzpunkt der Kamera verändert. Um dieses Problem zu lösen, wurde die `updateCamera()`-Methode um die oben aufgeführte Berechnung erweitert. Diese Methode wird jedes Mal bei einer Veränderung der Kamera aufgerufen.

4.3 Generierung von Schattenvolumen

4.3.1 Grundgerüst eines Datenstruktur-Renderers

Wie im vorigen Kapitel bereits erwähnt, findet die Erzeugung, Auswertung und Wahl des Verfahrens in den jeweiligen Renderern der einzelnen Datenstrukturen statt. Es wurde zunächst nur die Triangle-Mesh Datenstruktur komplett implementiert. Das Grundgerüst sollte jedoch für alle anderen Datenstrukturen wie z.B. Punktwolken dasselbe sein. Die Bestimmung der Silhouettenkanten müsste beispielsweise für die Punktwolken-Datenstruktur angepasst werden, da die hier in der Arbeit vorgestellten Ansätze so nicht funktionieren. Das Grundgerüst des Renderers für Triangle-Meshes sieht wie folgt aus:

Algorithm 1 RenderContentTriangleMesh generateShadowVolumes

```
1:  if object throws shadow and is in range then
2:     if scene has changed then
3:        setupTestingMethod() #determine if zFail is required
4:     end if
5:     updateBackfaceInformation()
6:     if zFail is required then
7:        Setup stencil buffer to use zFail
8:        Increment for back facing polygons
9:        Decrement for front facing polygons
10:    else
11:       Setup stencil buffer to use zPass
12:       Increment for front facing polygons
13:       Decrement for back facing polygons
14:    end if
15:    drawShadowPolygons()
16: end if
```

In dem Prototyp handelt es sich um ein und dieselbe Methode (`draw3D()`) zum Zeichnen der Meshes sowie der Schattenvolumen. Ob nun die Mesh oder die Volumen gezeichnet werden, wurde mit Hilfe einer Fallunterscheidung gelöst. Zur Vereinfachung wurde die Methode zum Zeichnen der Schattenvolumen hier `generateShadowVolumes()` genannt.
Um Zeit und Rechenleistung zu sparen wurden zwei Eigenschaften hinzugefügt. Zum einen kann in der Applikation eingestellt werden, ob das Objekt einen Schatten werfen soll, zum anderen kann jedem Licht eine bestimmte Reichweite zugeordnet werden. Letzteres ist physikalische nicht korrekt, da das Licht nicht über eine bestimmte Reichweite an Stärke verliert. Stattdessen sagt die Eigenschaft nur aus, dass nach X-Einheiten das Licht diesen Punkt nicht mehr erreicht. Sollte keine Reichweite definiert sein, so wird das Objekt immer vom Licht erreicht. Um zu prüfen, ob sich ein Objekt in Reichweite des Lichts befindet, wird dessen BoundingBox verwendet. Es wird geprüft, ob einer der acht Eckpunkte oder der Mittelpunkt der Box sich im Radius des Lichts befindet.
Das Flag, welches definiert ob das Objekt einen Schatten wirft oder nicht, wurde aus Performance Gründen eingefügt. Angenommen bei der Szene handelt es sich um einen Raum. In diesem Fall macht es wenig Sinn die Schatten für die Raum-Mesh zu bestimmen, da sich außerhalb des Raumes nichts befindet.
Im nächsten Schritt muss bestimmt werden, welches Verfahren verwendet werden soll (Zeile 2-4). Dieser Schritt kann übersprungen werden unter folgenden Bedingungen. Es befinden sich keine Animationen in der Szene, also die Szene ist statisch. Außerdem darf sich die Position der Kamera sowie die des Lichtes nicht verändert haben. Diese Informationen werden mit Hilfe eines Flags der `generateShadowVolumes()`-Methode vom *Renderer3D* übergeben. Ein genauerer Einblick in die Methode `setupTestingMethod()` ist in dem Abschnitt „Wahl des Zählverfahrens“ zu finden.

Anschließend werden die Informationen aktualisiert bzw. beim ersten Durchlauf erstellt, in welcher Beziehung jedes Dreieck zum Licht steht. Also ob es zum Licht zeigt oder nicht. Dies geschieht in Zeile 5.
Zum Schluss werden die Zähleigenschaften des Stencil Buffers eingestellt. Abhängig davon, welches Verfahren zuvor gewählt wurde. Anschließend werden die Schattenvolumen gezeichnet (Zeile 6-16).

4.3.2 Unterscheidung zwischen Z-Pass und Z-Fail

In diesem Unterkapitel wird die Methode `setupTestingMethod()` aus Kapitel 4.3.1 erläutert. Ebenfalls wie bei der Bestimmung ob ein Körper in Reichweite des Lichts liegt, wird hier mit der BoundingBox der Objekte gearbeitet. Allerdings reicht es hier nicht aus, nur die Eckpunkte und den Mittelpunkt zu betrachten, da so in vielen Fällen nicht das Z-Fail Verfahren verwendet wird und inkorrekte Schatten entstehen. Aus diesem Grund wurde hier das Schnittpunkt-Verfahren (engl. Intersection) angewandt. Dabei wird geprüft, ob sich zwei Flächen im dreidimensionalen Raum schneiden. Dieses Verfahren ist sehr viel genauer, aber dafür auch leistungsaufwendiger. Wie bereits zuvor erwähnt, wird hier geprüft, ob das Objekt eine Seite oder den Boden der Pyramide vom Viewport zum Licht schneidet. Die Pyramide wird vom *Renderer3D* mit durch den Szenengraphen gereicht. Für die konkrete Umsetzung wurde die bereits existierende *Intersection*-Klasse aus dem CG Research-Framework verwendet. Dementsprechend ob es einen Schnitt zwischen Boundingbox und der Pyramide gibt, wird eine Variable gesetzt, die später beim Einstellen des Stencil Buffers abgefragt wird. Sollte es einen Schnitt geben muss das Z-Fail Verfahren verwendet werden, ansonsten reicht das Z-Pass Verfahren aus.

4.3.3 Zeichnen der Schattenvolumen

In diesem Kapitel wird das konkrete Zeichnen der Schattenvolumen detailliert beschrieben, was in der Methode `drawShadowPolygons()` geschieht.

Algorithm 1 RenderContentTriangleMesh drawShadowPolygons

```
1: Lw ⇐ 1 #init light w-coordinate
2: if light is directional then
3:   Lw ⇐ 0
4: end if
5: for all Edges e of triangle mesh do
6:   t1 ⇐ e.getTriangle()
7:   t2 ⇐ e.getOtherTriangle()
8:   if isBackfacing(t1) ≠ isBackfacing(t2) then
9:     if isBackfacing(t1) then
10:      vA ⇐ getVertex(e.getB())
11:      vB ⇐ getVertex(e.getA())
12:    else
13:      vA ⇐ getVertex(e.getA())
14:      vB ⇐ getVertex(e.getB())
15:    end if
16:    vC ⇐ {vB.x · Lw − L.x, vB.y · Lw − L.y, vB.z · Lw − L.z, 0}
17:    vD ⇐ {vA.x · Lw − L.x, vA.y · Lw − L.y, vA.z · Lw − L.z, 0}
18:    drawVertex4(vB)
19:    drawVertex4(vA)
20:    drawVertex4(vD)
21:    drawVertex4(vC)
22:  end if
23: end for
24: if Z-Fail is required then
25:   for all triangles t in triangle mesh do
26:     for all vertices v in t do
27:       if isBackfacing(t) then
28:         vInf ⇐ {v.x · Lw − L.x, v.y · Lw − L.y, v.z · Lw − L.z, 0}
29:         drawVertex4(vInf)
30:       else
31:         drawVertex4(v)
32:       end if
33:     end for
34:   end for
35: end if
```

Zunächst einmal erwartet die Methode ein Flag, welches signalisiert ob es sich um ein direktionales Licht handelt. Punkt-sowohl als auch Strahler-Lichtquellen können gleichbehandelt werden. Das Flag wird an Hand einer Typenüberprüfung der jeweiligen Lichtquelle in der `generateShadowVolumes`-Methode bestimmt. Im Falle einer

direktionalen Lichtquelle muss für die W-Koordinate des Lichts eine 0 ansonsten eine 1 verwendet werden. Die Überprüfung sowie die Initialisierung der W-Koordinate geschehen in Zeile 1-4 des Pseudocodes.
Anschließend werden die Seiten des Volumens gezeichnet, welche unabhängig vom gewählten Verfahren sind. Hierzu wird einmal über alle Kanten iteriert. Wichtig ist an dieser Stelle, die Kanten werden in der bereits erwähnten `updateBackfacingInformation()`-Methode, beim ersten Durchlauf erzeugt und abgespeichert. Eine genauere Erklärung folgt in Abschnitt 4.3.4.
Danach wird geprüft, ob eines der beiden anliegenden Dreiecke zum und das andere vom Licht weg zeigt (Zeile 6-8). Ist dies der Fall handelt es sich um eine mögliche Silhouettenkante.
Das bedeutet für den Algorithmus, dass sich an dieser Stelle eine Seite des Volumens befindet bzw. gezeichnet werden muss. Bei den Seiten des Volumens handelt es sich um sogenannte Quads, also Rechtecke. Zeile 9-15 sind notwendig, damit die Normalen der gezeichneten Seitenteile alle nach außen zeigen. Im Fall, dass eine Seite nach innen zeigt, würden inkorrekte Schatten entstehen. Die vier Punkte des jeweiligen Rechtecks werden, wie in Kapitel 3.4 beschrieben, gebildet. Dabei bilden die zwei Vertices der Kante die Punkte A und B. Die Punkte C und D werden aus den Differenzen A-L und B-L gebildet (Zeile 16, 17). Wichtig ist, dass die w-Koordinate C und D null beträgt, damit mit Hilfe von P_{inf} die beiden Punkte korrekt in die Unendlichkeit projiziert werden können.
Nachdem alle vier Punkte bestimmt wurden, werden diese an OpenGL weitergereicht (Zeile 18-21). Der nachfolgende Abschnitt von Zeile 24-34 wird nur für das Z-Fail Verfahren benötigt. In diesem Abschnitt werden der „Deckel“ und Boden des Volumens erzeugt. Die Generierung ist sehr simpel. Es wird über alle Dreiecke der Mesh iteriert und jeweils geprüft, ob das Dreieck zum Licht zeigt. Im diesem Fall bildet es einen Teil des Bodens. Dies bedeutet, dass alle drei Vertices unverändert gezeichnet werden (Zeile 34). Im Fall, dass es vom Licht weg zeigt, muss für jeden Vertex des Dreiecks die Differenz V-L gebildet werden (Zeile 28-29). Ebenfalls wie schon bei den Seiten ist es hier wichtig, die w-Koordinate auf null zu setzen. Nach Abschluss dieser Prozedur befinden sich die Schatten bereits im Stencil Buffer.

4.3.4 Bestimmung der Silhouettenkanten

In diesem Unterkapitel geht es mehr um die zu Grunde liegende Datenstruktur als die eigentliche Bestimmung der Silhouettenkanten, da die entscheidende Überprüfung in der `drawShadowPolygones()`-Methode geschieht (Kapitel 4.3.3, Zeile 8). Wie bereits zuvor erwähnt, werden die benötigten Informationen in der `updateBackfacingInformation()`-Methode bestimmt. Der Ablauf der Methode sieht dabei wie folgt aus:

Algorithm 1 RenderContentTriangleMesh updateBackfacingInformation

```
1: if first run then
2:    initialize backfacing map
3: end if
4: for all triangles t in mesh do
5:    l ← L.getPosition() − t.getA() #Direction vector to the light source
6:    n ← t.getNormal() #Normal of the triangle
7:    result ← (multiply(l, n) < 0)
8:    backfacingMap.put(t, result)
9: end for
```

Als grundliegende Datenstruktur wurde eine Map gewählt, wobei die Dreiecke die Schlüssel und ein Boolean die Werte bilden. Der Boolean spiegelt die Information wieder, ob ein Dreieck vom Licht weg zeigt. Der Funktionsaufruf `isBackfacing(Triangle t)` in Kapitel 4.3.3 ist lediglich ein Zugriff auf die Map. Da die Map, genauso wie die Collection der Kanten nur bei dem Durchlauf mit Schatten benötigt wird, werden diese während des ersten Renderdurchlaufs initialisiert (Zeile 1,3).
Anschließend wird für alle Dreiecksnormalen das Skalarprodukt gebildet und geprüft ob dieses kleiner als null ist (Zeile 7). In dies der Fall, wird ein *true* in der Map gespeichert, ansonsten *false* (Zeile 8). Damit das Skalarprodukt gebildet werden kann, wird die Normale des Dreiecks sowie der Richtungsvektor vom Dreieck zum Licht benötigt. Die Normale ist bereits gegeben und kann aus dem Dreieck abgefragt werden (Zeile 6). Um den Richtungsvektor zu berechnen muss ebenfalls nur die Subtraktion L-V erfolgen. Dabei gilt: L ist der Ortsvektor vom Licht und V einer der drei Vertices vom Dreieck. In dem Pseudocode wurde der Eckpunkt A gewählt (Zeile 5).

4.3.5 Kanten

Angenommen es wären nur die Dreiecke ohne eine Kantenbeziehung gegeben. Im schlimmsten Fall müsste dann über alle Dreiecke iteriert werden, um das Nachbardreieck zu finden. Das ist sehr kostenintensiv und verringert die Performance drastisch.
Aus diesem Grund wurde eine Hilfskantenstruktur verwendet. Die Kanten enthalten dabei Indices der beiden Vertices, welche die Kante bilden, sowie eine Referenz zu den zwei anliegenden Dreiecken. Ebenfalls wie die Backfacing-Map wird die Kantenliste zur Renderlaufzeit erzeugt. Dies geschieht beim ersten Durchlauf, noch vor dem Aufruf der `generateShadowVolumes()`-Methode. Eine grundliegende Annahme ist dabei, dass die Mesh nicht während der Laufzeit verändert wird. Also es werden keine weiteren Dreiecke hinzugefügt oder bestehende Dreiecke gelöscht. Das Erstellen der Kantenliste ist sehr simpel. Es wird lediglich einmal über alle Dreiecke der Mesh iteriert und alle drei Kanten erzeugt. Für jede Kante wird geprüft, ob diese oder die Umkehrung bereits in der Liste

enthalten ist. Sollte diese noch nicht enthalten sein, so wird die Referenz des aktuellen Dreiecks in der Kante hinterlegt und die Kante der Liste hinzugefügt. Im Fall, dass die Kante oder die Umkehrung bereits existiert, wird das Dreieck bei der existierenden Kante als Nachbardreieck hinterlegt.

Algorithm 1 RenderContentTriangleMesh createEdgeList

```
1:  initialize list
2:  for all triangles t in mesh do
3:    for i := 0 to 3 do
4:      e ← createEdge(t.get(i), t.get((i + 1) mod 3))
5:      if list not contains e then
6:        e.addTriangle(t)
7:        list.add(e)
8:      else
9:        existingEdge ← getExistingEdge(list, e)
10:       existingEdge.addTriangle(t)
11:     end if
12:   end for
13: end for
```

Eine Alternative zu der meist verwendeten TriangleMesh-Datenstruktur mit einer Vertex- und Index-Liste, wäre die Halbkantendatenstruktur. Bei dieser Struktur ist bereits für jeden Vertex eine Kante hinterlegt, welche wiederum den Vertex kennt. Des Weiteren sind in der Kante die gegenüberliegende Halbkante, die nachfolgende Halbkante sowie die sich links befindende Facette bekannt. Diese Datenstruktur erlaubt es problemlos und effizient Nachbardreiecke zu finden. Sollte diese Datenstruktur dem Algorithmus zu Grunde liegen, kann auf die oben erwähnte Hilfsstruktur verzichtet werden.

5 Evaluation

5.1 Aufbau

Die Evaluation unterteilt sich in drei Unterkapitel: die Stärken und Schwächen des Schattenvolumen-Algorithmus sowie die Performance in Testszenen. Dabei werden in dem Kapitel „Stärken“ größtenteils die Vorteile gegenüber Shadow-Maps erläutert, während sich das Kapitel „Schwächen“ überwiegend mit allgemeinen Problemen und Nachteilen der Schattenvolumen auseinander setzt. Das Kapitel „Performance“ unterteilt sich in zwei weitere Unterkapitel. Im ersten werden Messung an Hand einer generierten Stadtszene und verschiedener Einstellungen durchgeführt. Im zweiten wird an Hand mehrerer Szenen geprüft, wie viel Leistung sich mit dem Z-Pass Verfahren einsparen lässt.

5.2 Stärken

Einer der größten Vorteile von Schattenvolumen sind die scharfen Kanten. Anders als das Shadow-Mapping werden die Schattenvolumen auf Geometriebasis berechnet. Das Shadow-Mapping ist hingegen ein Pixel-basierter Algorithmus. Der mit größte Nachteil bei der Verwendung von Shadow-Maps ist, dass die Qualität der erzeugten Schatten stark von der gewählten Größe der Karte abhängt. Des Weiteren ist die Qualität von der Entfernung zwischen dem schattenwerfenden Objekt und dem beschatteten Objekt abhängig. Sollte die Karte zu klein oder die Entfernung zu weit sein, wirken die Schatten leicht bis stark pixelig. Mit Schattenvolumen besteht dieses Problem nicht, da diese Pixel unabhängig sind. Die Kanten eines Schattenvolumens sind immer scharf, wie es sehr gut in Abbildung 5.1 zu sehen ist. Für die Szene wurde eine Punktlichtquelle verwendet, welche sich direkt mittig unter der Decke des Raumes befindet.

Außerdem erzeugen Schattenvolumen Self-Shadows. Der Schattenvolumen-Algorithmus erzeugt sowohl Schlag- als auch Eigenschatten. Im Fall, dass ein Objekt Teile von sich selbst verdeckt, werden diese korrekt im Schatten dargestellt.

Ein weiterer Vorteil von Schattenvolumen ist die Unterstützung von Punktlichtquellen. Im Vergleich: der reine Shadow-Map-Algorithmus funktioniert nur mit direktionalen Lichtquellen, da zur Erstellung der Shadow-Maps die Szene einmal aus Sicht des Lichts gerendert werden muss. Direktionale Lichtquellen weisen diese Eigenschaft der „Sicht“ auf. Bei Punktlichtquellen hingegen wird das Licht in alle Richtungen gleichmäßig verteilt. Die Ausbreitung findet kugelförmig statt. Da die Erzeugung von Schattenvolumen richtungsunabhängig ist, lassen sich diese für Shadow-Maps schwierigen Szenen mit Punktlichtquellen sehr leicht und problemlos rendern. Ein sehr gutes Beispiel ist an Hand eines beleuchteten Halloween-Kürbisses mit einer Punktlichtquelle in Abbildung 5.2 zu finden. Die Szene ist aus mehreren Gründen schwierig. Zum einen liegt der Großteil der Szene im Schatten und nicht umgekehrt, was meist der Fall ist (z.B. Sonne). Zum anderen kann das Licht in zwei Richtungen entweichen, einmal nach vorne durch das Gesicht des Kürbisses und einmal nach oben, da dieser dort ebenfalls eine Öffnung aufweist.

Mit Hilfe von Schattenvolumen lassen sich auch sehr gute Soft-Shadows erzeugen, wenn die Performance jedoch nicht im Vordergrund steht. Bereits mit nur sehr wenigen zusätzlichen Lichtquellen wirken die entstandenen Soft-Shadows glaubwürdig. In Abbildung 5.3 wurde die in Kapitel 4.2 vorgestellte Lichterfläche auf der Y-Achse mit acht zusätzlichen Lichtquellen verwendet. Es wurde ebenfalls ein Stuhl weiter Außen positioniert damit die Soft-Shadows besser zu erkennen sind. Ansonsten ist die Szene unverändert im Vergleich zu Abbildung 5.1.

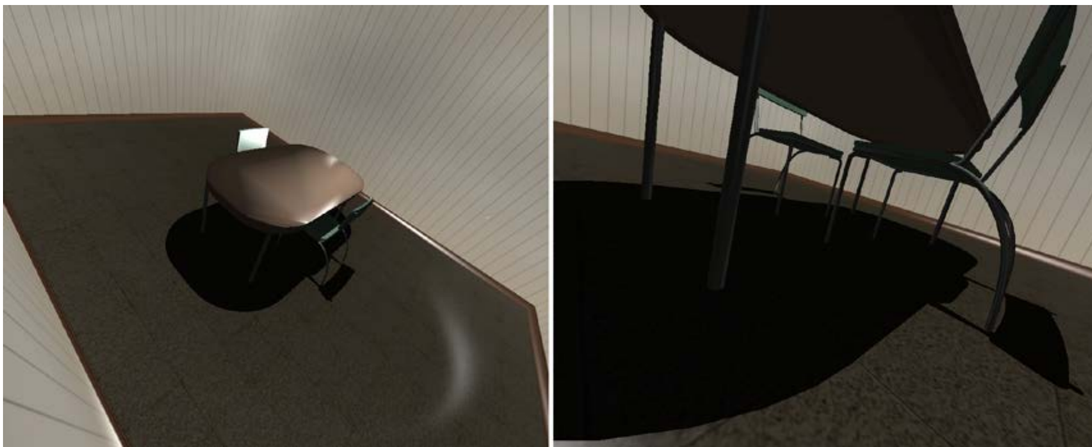


Abbildung 5.1: Szene mit Schattenvolumen. Es ist deutlich zu erkennen, dass die Kanten der Schatten aus der Ferne (links) sowie vom nahen (rechts) nicht pixelig sind.

Abbildung 5.2: Punktlichtquelle innerhalb eines Körpers. Der Großteil der Szene befindet sich im Dunkeln, die Szene wird lediglich durch die obere Öffnung und dem Gesicht des Kürbisses beleuchtet.

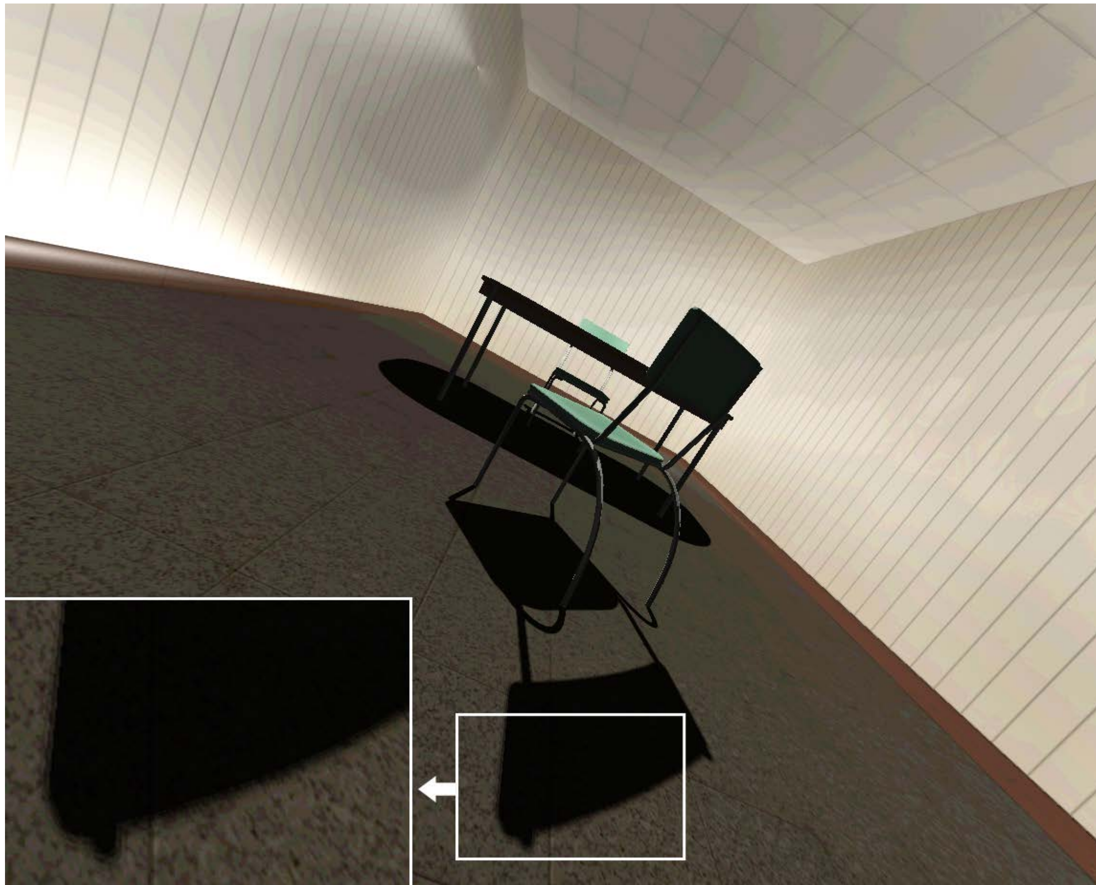


Abbildung 5.3: Soft-Shadows generiert mit der Lichterfläche auf der Y-Achse

5.3 Schwächen

Der große Vorteil der geometriebasierten Schatten ist ebenfalls die größte Schwäche des Algorithmus. Die Performance des Algorithmus ist stark von der Beschaffenheit der Objekte in der Szene abhängig. Bei sehr polygonreichen, aber auch bei polygonärmeren Objekten kann dies schnell bemerkbar werden. Diese Problematik wird bei den Messungen in Kapitel 5.4.1 noch einmal sehr deutlich gezeigt. Ebenfalls die Anzahl der Lichtquellen kann die Performance stark beeinträchtigen. Ein gutes Beispiel dafür sind die Soft-Shadows des Prototypens, da diese durch mehrere nebeneinanderpositionierten Lichtquellen erzeugt werden. Die Szene in Abbildung 5.3, die aus drei schattenwerfenden Objekten und einer Lichtquelle besteht, erreicht mit harten Schatten eine durchschnittliche Framerate von 64 FPS (Frames per second). Bei eingeschalteten Soft-Shadows, also mit 8 zusätzlichen Lichtquellen, liegt die durchschnittliche Framerate bei 18 FPS.

Der Schattenvolumen-Algorithmus funktioniert außerdem nur mit komplett geschlossenen Modellen. Sollte ein Modell in der Szene beispielsweise beabsichtigt ein Loch im Boden haben, so ist es nicht möglich korrekte Schatten mit dem Algorithmus zu generieren. Eine andere Möglichkeit wäre, dass ein Modell ungewollte Risse aufweist, welche auf Grund schlechter Modellierung vorhanden sind. In beiden Fällen, werden Fragmente der Schattenvolumen selbst in der finalen Szene sichtbar, welche jedoch im eigentlichen Schatten fehlen. Abbildung 5.4 zeigt ein solches Beispiel.

Abbildung 5.4: Auftretende Probleme, wenn ein Modell Risse aufweist. In diesem Beispiel am Kopf als auch bei den Beinen zu erkennen. [1]

[1] Hulk Modell: http://tf3dm.com/3d-model/hulk-70258.html (3dregenerator)

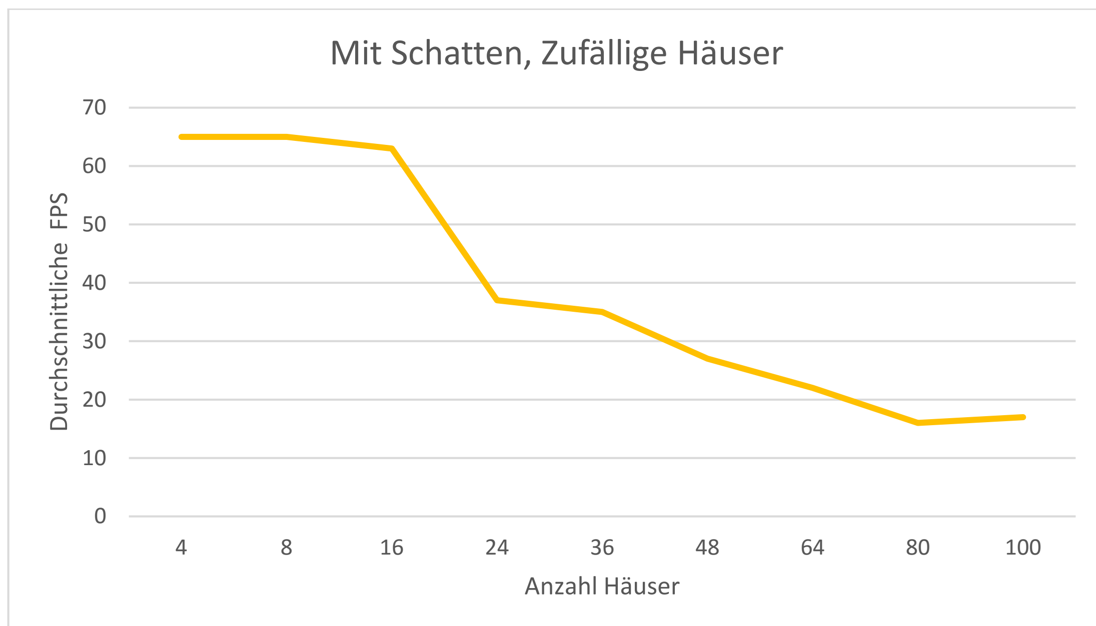
5.4 Performance

5.4.1 Messungen

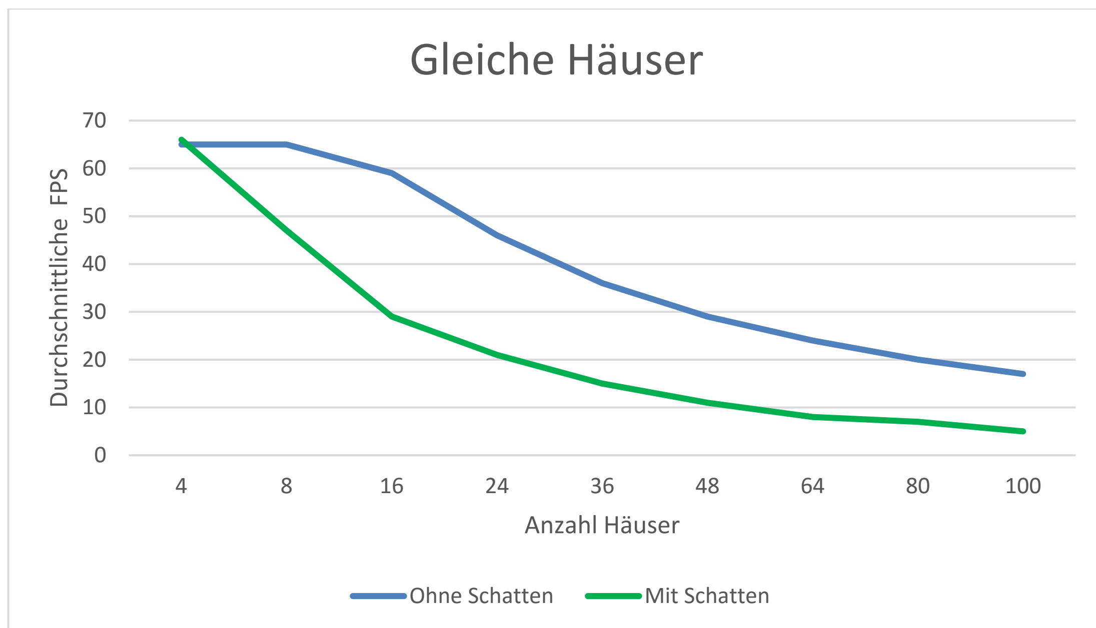
Für die Messungen der FPS (Frames per second) wurde der *UrbanScene*-Generator verwendet, welcher bereits im Framework vorhanden war. Mit Hilfe des Generators lassen sich beliebig viele Häuserblöcke erzeugen. Ein Block besteht dabei aus vier zufällig ausgewählten Häusern. Bei den Häusern handelt es sich um sehr einfache, polygonarme Modelle. Es wurden insgesamt drei Messungen vorgenommen, um zu testen wie sich die Framerate zum einen bei Erhöhung der Anzahl der Objekte und zum anderen bei Erhöhung der Anzahl der Lichtquellen verhält. Aus diesem Grund sehen die Tests wie folgt aus.

Im ersten Test wurde nur eine Lichtquelle verwendet und stätig die Anzahl der Blöcke erhöht. Dieser wiederum besteht aus zwei Messungen. Als erstes wurde die Framerate mit zufälligen Häusern gemessen. Da aber, wie bereits in Kapitel 5.3 erwähnt wurde, der Schattenvolumen-Algorithmus stark von den Objekten abhängig ist, wurde der Zufallsfaktor bei den zweiten Messungen entnommen. Es wurde nur ein Haus-Modell verwendet.

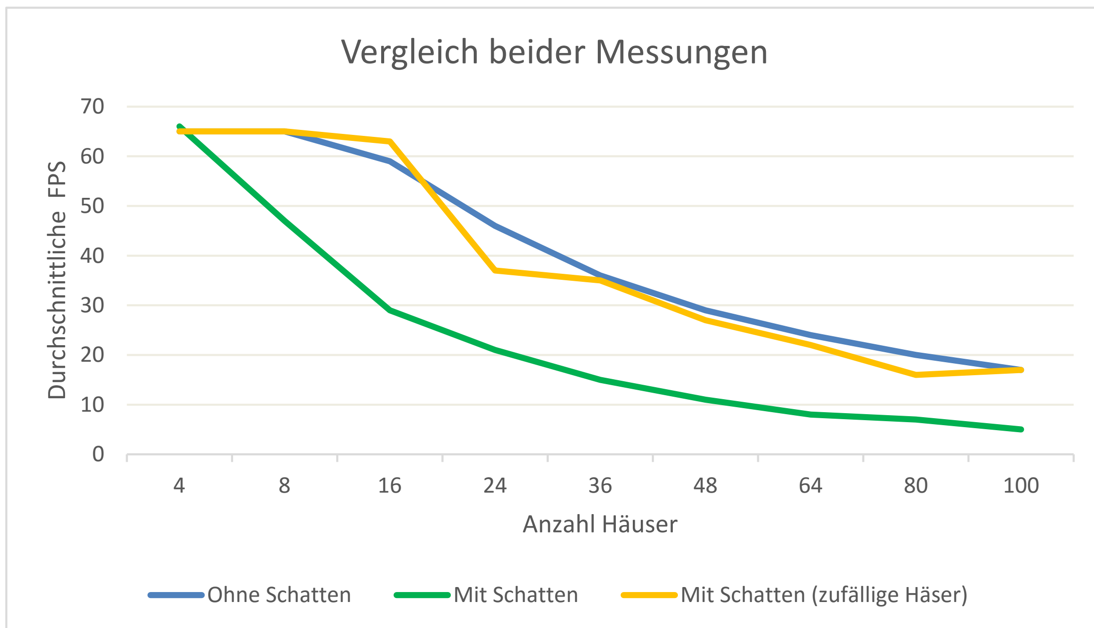
Im zweiten Test hingegen wird eine Stadt-Szene mit 16 Häusern benutzt, die mit nur einer Lichtquelle konstante 66 FPS liefert. Bei dieser Szene werden dann weitere Lichtquellen hinzugefügt. Für alle Messungen wurde eine Auflösung von 800x640 gewählt.



Bei dieser Messung war es nicht möglich einen direkten Vergleich zu der Szene ohne Schatten herzustellen, da die Wahl der Häuser zufällig ist. Dennoch hat die Messung einige interessante Ergebnisse gezeigt. Bei einer generierten Stadt mit bis zu 20 Häusern läuft die Szene problemlos mit konstanten 60 FPS. Wird die Anzahl der Häuser verdoppelt liefert die diese weiterhin noch 30 FPS. Interessant jedoch sind die Messungen bei 16 und 24 Häusern. Während das Testsystem bei 16 Häusern noch gute 63 Bilder pro Sekunde zeichnen kann, bricht die Framerate bei 24 rapide auf 37 FPS ein. Ebenfalls interessant sind die Messungen mit 80 und 100 Häusern. Bei 80 Häusern wird eine Framerate von 16 FPS und bei 100 eine von 18 FPS erreicht. Beide Messungen deuten darauf hin, dass einmal mehr Häusermodelle zu Gunsten und einmal mehr Modelle zum Nachteil des Schattenvolumen-Algorithmus gewählt wurden sind.



Bei der zweiten Messung wurde nur ein Hausmodell (*Haus01*) verwendet, um einen Vergleich zwischen der Szene mit und ohne Schatten herzustellen. Die Ergebnisse der Messung zeigen, dass mit dem verwendeten Hausmodell maximal nur 16 Häuser dargestellt werden sollten, da bereits hier nur noch 30 Bilder pro Sekunde gezeichnet werden. Im Vergleich: bei dem ersten Test mit 16 zufälligen Häusern wurden noch 63 FPS erzielt.
Des Weiteren ist zu erkennen, dass der Verlust bei eingeschalteten Schatten immer stärker zunimmt. Während die Verlustrate bei 8 Häusern nur 28% beträgt, sind es bei 16 Häusern bereits 51%. Sollte die Szene aus 100 Häusern bestehen, liegt die Verlustrate sogar bei 71%.



Wenn nun die Ergebnisse beider Messungen verglichen werden, fällt auf, dass die Ergebnisse der Messung mit zufälligen Häusern und Schatten sehr nahe bei den mit dem gleichen Haus ohne Schatten liegen. Dies lässt darauf schließen, dass das gewählte Modell für die Messungen mit demselben Haus generell relativ viel Leistung benötigt.

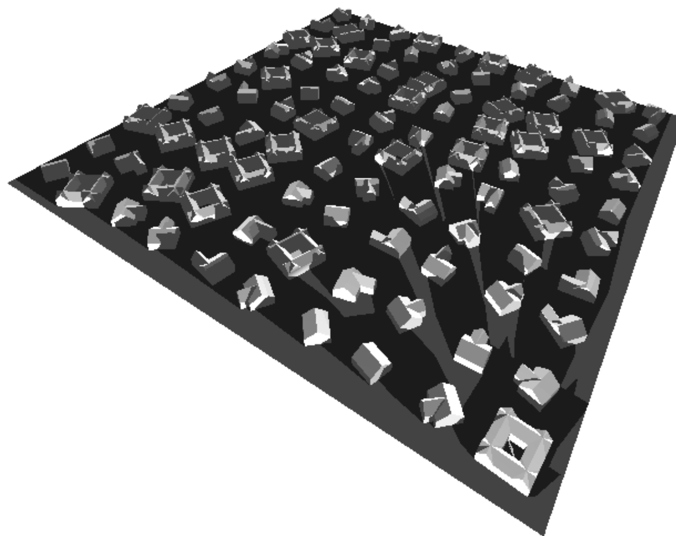


Abbildung 5.5: Stadtszene mit 100 zufälligen Häusern und Schatten

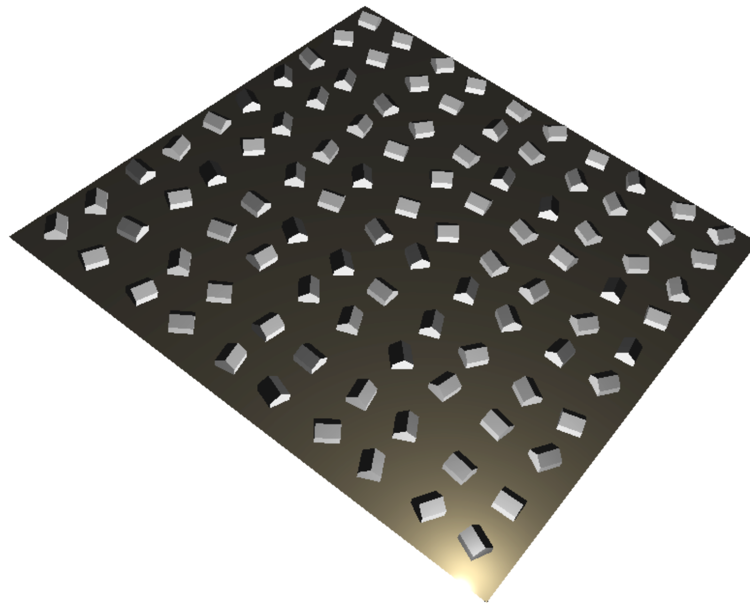


Abbildung 5.6: Stadtszene mit 100 gleichen Häusern ohne Schatten

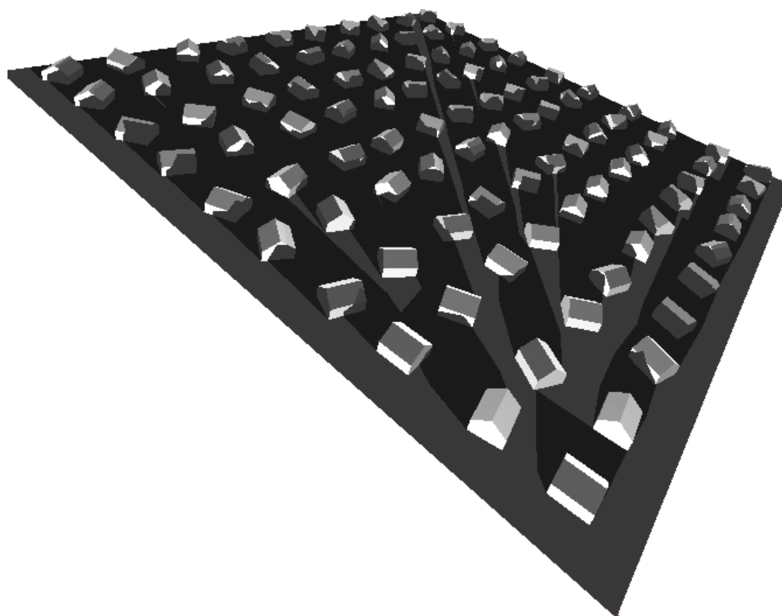
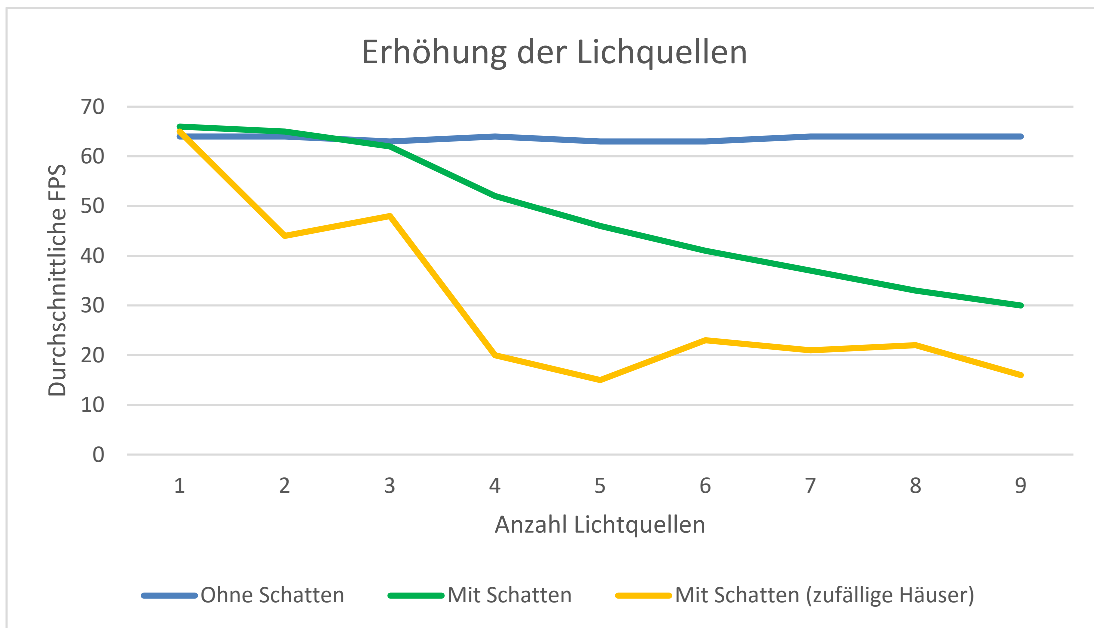


Abbildung 5.7: Stadtszene mit 100 gleichen Häusern und Schatten



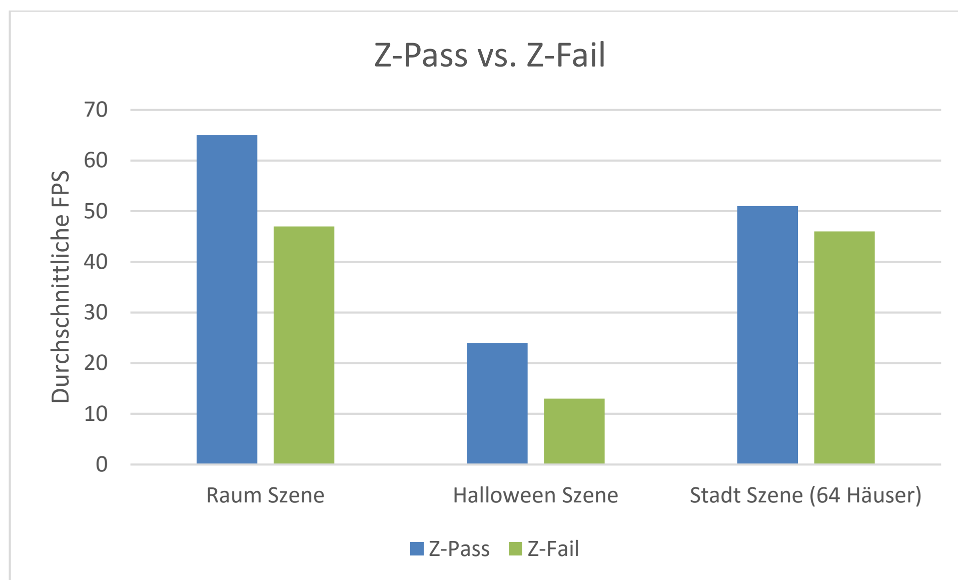
Bei diesem Test wurde stetig die Anzahl der Lichtquellen erhöht. Die Position der Lichtquellen bleibt dabei immer dieselbe, da hier lediglich die Leistung gemessen werden soll. Es wurde einmal mit zufällig gewählten Häusern und einmal mit nur einem Modell gemessen. Beim Letzteren jeweils einmal mit und ohne Schatten. Auf Grund der niedrigen Performance im vorigen Test wurde für den zweiten ein anderes Modell (*Haus02*) verwendet.

Wie die Ergebnisse der Messungen zeigen, hat die Erhöhung der Anzahl der Lichtquellen keinen großen Einfluss auf die Performance, wenn keine Schatten verwendet werden. Die durchschnittliche Framerate liegt immer im 60 FPS Bereich. Dies könnte daran liegen, dass mit der Erhöhung der Anzahl der Lichtquellen lediglich die Berechnungen in den Shadern mehr werden. Anders als bei der Erhöhung der Anzahl der Häuser, wo die Anzahl der Draw-Calls steigt. Auch die Ergebnisse der Messung mit Schatten liegen immer über 30 FPS, wobei die Framerate mit bis zu drei Lichtquellen sogar noch über 60 FPS liegt.
Aber auch hier steigt die Verlustrate stetig an, während der Verlust bei vier Lichtquellen nur 18% beträgt, sind es bei neun Lichtquellen bereits 53%.
Ebenfalls interessant ist die Messung mit zufälligen Häusern, wie bereits im vorigen Test, wird auch hier sehr deutlich, dass der Algorithmus stark von den Objekten abhängig ist. Es lässt sich aus den Ergebnissen schließen, dass für die Messung mit nur einem Haus, ein für den Algorithmus sehr leichtes Modell verwendet wurde.

5.4.2 Z-Pass vs. Z-Fail

In diesem Kapitel wird überprüft wie viel Leistung sich mit Hilfe des Z-Pass Verfahrens in der Praxis einsparen lässt. Wie bereits im Konzept erläutert wurde, muss das Z-Fail Verfahren benutzt werden, wenn sich der Beobachter innerhalb eines Schattenvolumens befindet. In diesem Fall würde das Z-Pass Verfahren inkorrekt zählen. Ein konkretes Beispiel an Hand einer Szene dafür ist in Abbildung 5.8 dargestellt.

In diesem Test wurden drei Szenen jeweils mit Z-Pass und Z-Fail gerendert und die durchschnittliche Framerate gemessen. Dabei wurde vernachlässigt, ob das Z-Pass Verfahren korrekt zählt, da dies in der Halloween-Szene nicht möglich ist. Des Weiteren wurde bei beiden Messungen die Überprüfung, welches Verfahren verwendet werden muss, abgeschaltet.



Bei den drei Szenen handelt es um die Raum-, die Halloween- und die Stadt-Szene. Die Raum-Szene besteht aus einen einfachen Raum mit einem Tisch und zwei Stühlen (siehe Abbildung 5.1). Bei der Halloween-Szene handelt es sich um einen Hauseingang mit sieben Kürbissen, einer davon ist ausgehöhlt und dient als Lichtquelle (siehe Abbildung 5.2). Die letzte Szene wurde, wie in den vorigen Tests auch, mit Hilfe des UrbanScene-Generators erstellt (siehe Abbildung 5.7). Dabei wurden 64 gleiche Häuser (*Haus02*) verwendet.
Die Ergebnisse der Messungen zeigen deutlich, dass in allen drei Fällen das Z-Pass Verfahren mehr FPS liefert. Ebenfalls ersichtlich wird, dass die Komplexität der Modelle eine Rolle spielt. Umso komplexer die Modelle, desto mehr Leistung kann das Z-Pass Verfahren erzielen. Während das Z-Pass Verfahren bei der Stadt-Szene 9.8% mehr Frames liefert, erzielt es bei der Raum-Szene 27% und bei der Halloween-Szene sogar 45,8% mehr

Frames. Diese Ergebnisse sind jedoch nicht überraschend, da man bedenken muss, dass bei dem Z-Fail Verfahren jedes Modell zweimal gezeichnet werden muss, da Vorder- und Rückseite des Modells den Boden und „Deckel“ des Schattenvolumens bilden.

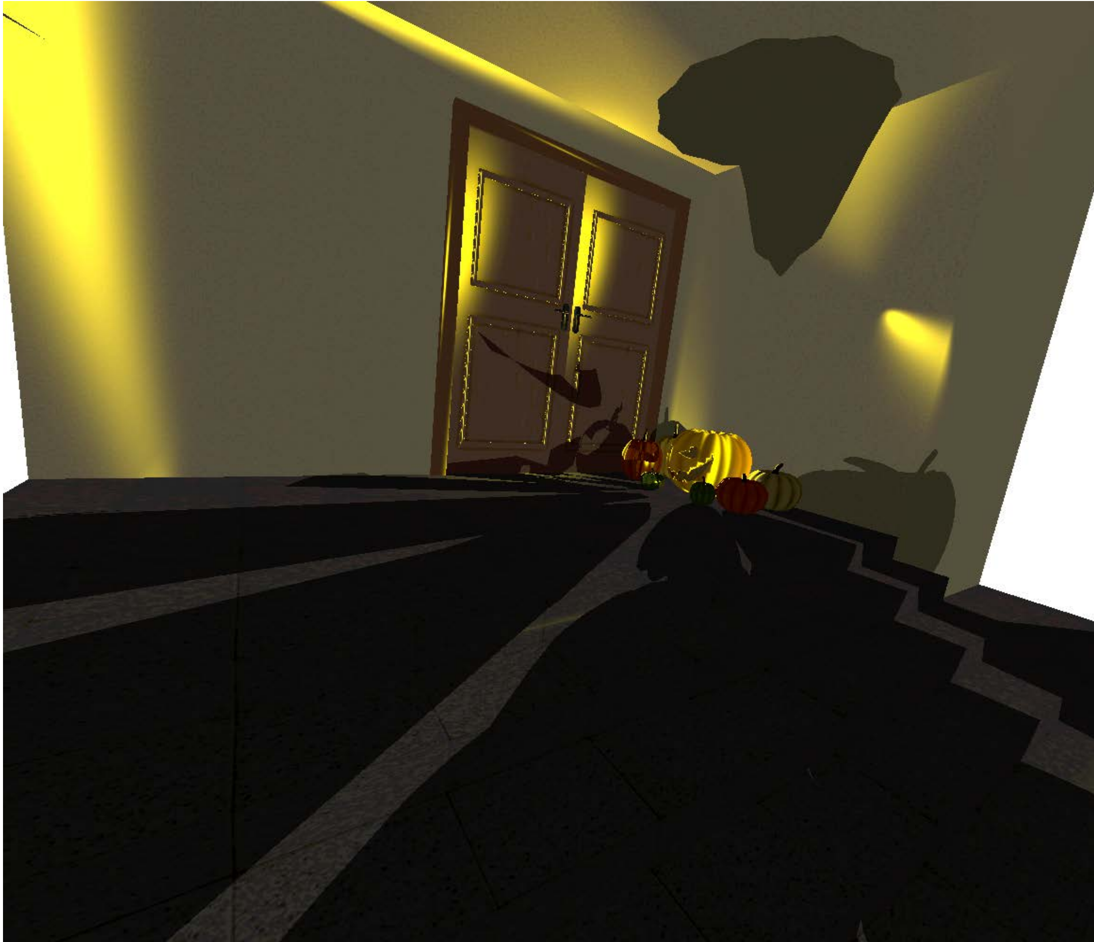


Abbildung 5.8: Z-Pass Zählproblem am Beispiel der Halloween-Szene aus Abbildung 5.2, wenn der Beobachter sich im Schatten befindet

6 Fazit und Schluss

6.1 Fazit

Ziel dieser Arbeit bestand in der Recherche und Analyse von Echtzeitschatten-Algorithmen. Es wurde das Shadow-Mapping Verfahren vorgestellt, ein pixelbasierendes Verfahren, welches auf Grund der guten Performance-Abschätzungen in den meisten Echtzeitanwendungen verwendet wird. Außerdem wurden die Schattenvolumen vorgestellt. Eine Alternative, die anders als die Schattenkarten auf die geometrische Grundlage basiert. Zum Vergleich wurde auch das Ray-Tracing vorgestellt. Ein Modell zur Lichtausbreitung, welches durch seine Funktionsweise ebenfalls Schatten erzeugt. Anders als die beiden Schattenalgorithmen ist dieses Modell nicht echtzeitfähig, dafür jedoch aus physikalischer Sicht am korrektesten. Es wurde an Hand des Prototypens gezeigt, dass sich mit Hilfe der Schattenvolumen sehr gute und glaubwürdige Schatten erzeugen lassen. In der Evaluation wurde ebenfalls deutlich, dass Schattenvolumen stark von der geometrischen Beschaffenheit der Modelle abhängig sind. Auf Grund dieser Abhängigkeit ist die Performance einer Szene mit Schattenvolumen nur sehr schwer abzuschätzen.

Aus den gesammelten Erkenntnissen lässt sich schließen, dass bei einer kleineren Szene und besonders kleinere Szenen mit Punktlichtquellen, Schattenvolumen sehr gut geeignet sind. Sollte die Szene zu groß sein oder die Modelle zu komplex, wäre das Shadow-Mapping die bessere Wahl. Für Bilder, wo physikalische Korrektheit wichtig ist und Echtzeit keine Rolle spielt, kann das Ray-Tracing verwendet werden, anstelle eines Schatten-Algorithmus.

6.2 Ausblick

Schattenalgorithmen werden auch in der Zukunft, besonders in der Spieleindustrie, eine bedeutende Rolle spielen. Auch wenn die Leistung der Grafikhardware in den letzten Jahren deutlich angestiegen ist, reicht die momentane Leistung nicht aus um beispielsweise Verfahren wie das Ray-Tracing in Echtzeitanwendungen zu benutzten, welches die Schattenalgorithmen ablösen könnte.

Der Prototyp dieser Arbeit lässt sich in zwei Richtungen erweitern. Zum einen die Erweiterung der unterstützten Datenstrukturen. Im Rahmen dieser Arbeit wurde lediglich die TriangleMesh-Struktur implementiert. Andere Strukturen wie z.B. die Punktwolken stellen eine Herausforderung dar, weil es sich ausschließlich nur um Punkte handelt. Es müsste ein komplett neues Verfahren entwickelt werden, um die SIlhouettenkanten zu bestimmen und zu generieren. Für das Z-Pass Verfahren würde dies bereits ausreichen. Um ebenfalls das Z-Fail Verfahren verwenden zu können, müsste man sich einen Weg überlegen, wie die Vorder- und Rückseite des Objektes bestimmt werden kann. Eine weitere Überlegung wäre, wie die beiden Seiten dann in eine TriangleMesh-Struktur überführt werden können, um den Boden und „Deckel“ des Volumens zu bilden.

Die anderen Erweiterungen wären am Algorithmus selbst. Wie in Kapitel 3.6 erläutert, gibt es eine deutlich effizientere Methode weiche Schatten zu erzeugen. Damit weiche Schatten auch bei größeren Szenen in Echtzeit zum Einsatz kommen können, wäre es eine Überlegung wert, den Penumbra-Wedges-Algorithmus von Akenine-Möller und Assarsson [ASSA03] zu implementieren. In diesem Fall würde sich diese Arbeit perfekt als Grundlage eignen, da der Penumbra-Wedges auf den Stencil Shadow Volumes-Algorithmus aufbaut. Um den Umbrabereich zu bestimmen, kann der in dieser Arbeit entstandene Prototyp verwendet werden. Die konkrete Erweiterung bestünde hier in der Erzeugung der Penumra-Wedges bei den Silhouettenkanten und dessen Auswertung.

Es können ebenfalls Performance-Optimierungen durchgeführt werden. Eine davon wäre das Scissoring. Dies bedeutet, dass vor dem Zeichnen ein Bereich festgelegt wird, indem die Volumen gezeichnet und ausgewertet werden. Eine konkrete Umsetzung dafür wurde von McGuire [MGU03] vorgestellt. Eine weitere Optimierung wäre das Auslagern der Bestimmung von Silhouettenkanten sowie das Zeichnen der Schattenvolumen auf die GPU. Das kann mit Hilfe eines Geometry-Shaders realisiert werden [WÄCH07].

Literaturverzeichnis

[AM08] Tomas Akenine-Möller, Eric Haines, Naty Hoffman: *Real-time Rendering*, Third edition, Taylor & Francis Ltd, 2008

[APP68] Arthur Appel: *Some Techniques for Shading Machine Renderings of Solids*, AFIPS Press, Arlington, 1968

[ASSA03] Ulf Assarsson: *A Real-Time Soft Shadow Volume Algorithm*, Chalmers University of Technology Göteborg, 2003

[BERG86] Philippe Bergeron: *A General Version of Crow's Shadow Volumes*, IEEE Computer Graphics and Applications, 1986

[BIL99] Bill Bilodeau, Mike Songy, Creative Labs sponsored game developer conference, unpublished slides, 1999

[BLI93] Jim Blinn: *A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform*, IEEE Computer Graphics and Applications, 1993

[BRO84] Lynne Brotman, Norman Badler: *Generating Soft Shadows with a Depth Buffer Algorithm*, IEEE Computer Graphics and Applications, 1984

[CARM00] John Carmack, unpublished correspondence, 2000

[CROW77] Franklin Crow: *Shadow Algorithms for Computer Graphics*, ACM SGGRAPH Computer Graphics, 1977

[DIEF96] Paul Diefenbach: *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions*, University of Pennsylvania, 1996

[DIET01] Sim Dietrich: *Practical Priority Buffer Shadows*, in Mark DeLoura, ed., Game Programming Gems2, Charles River Media, 2001

[FOLEY95] James D. Foley, Andries van Dam, Steven K. Feiner: *Computer Graphics: Principles and Practice in C*, Second Edition, Addison Wesley, 1995

[FORS07] Tom Forsyth: *Shadowbuffers*, Game Developers Conference, March 2007

[FOUR88] Alain Fournier, Donald Fussell: *On the Power of the Frame Buffer*, ACM Transactions on Graphics, 1988

[FU85] Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan, Spach, John Austin, Frederick Brooks, John Eyles, John Poulton: *Fast Spheres, Shadows, Textures, Transparencies, and Image Enchancements in Pixel-Planes.*, Proceedings of SIGGRAPH, 1985

[HEID91] Tim Heidmann: *Real shadows, real time*, IRIS Universe, 1991

[HOUR85] J.C. Hourcade, A. Nicolas: *Algorithms for Antialiased Cast Shadows*, Computers and Graphics, 1985

[KILG02] Mark J. Kilgard, Cass Everitt: *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*, NVIDIA Corporation, 2002

[KILG04] Mark J. Kilgard: *Real-time Shadowing Techniques: Shadow Volumes*, ACM SIGGRAPH Computer Graphics, 2004

[LENG02] Eric Lengyel: *The Mechanics of Robust Stencil Shadows,* http://www.gamasutra.com/view/feature/131351/the_mechanics_of_robust_stencil_.php, 2002 (letzter Aufruf am 7.3.2016)

[MGU03] Morgan McGuire, John F. Hughes, Kevin T. Egan: *Fast, Practical and Robust Shadows*, Brown University, 2003

[WÄCH07] Carsten Wächter, Alexander Keller, Matrin Stich: *Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders*, NVIDIA Corporation, 2007

[WHIT79] Turner Whitted: *An Improved Illumination Model for Shaded Display*, Communications of the ACM, 1979

[WILL78] Lance Willams: *Casting curved shadows on curved surfaces*, ACM SIGGRAPH Computer Graphics, 1978

[YEN02] Hun Yen Kwoon: *The Theory of Stencil Shadow Volumes*, http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/the-theory-of-stencil-shadow-volumes-r1873, 2002 (letzter Aufruf am 7.3.2016)

Versicherung über Selbstständigkeit

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

Hamburg, den ________________ ____________________________