UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Soft Shadow Volumes

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Ilja Kipermann

Betreuer: Dipl.-Inform. Thorsten Grosch
(Institut für Computervisualistik, AG Computergrafik)

Koblenz, im Februar2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin mit der Veröffentlichung der Arbeit in der Universitätsbibliothek / in der Universitätsbibliothek und auf den Webseiten des Fachbereiches nicht einverstanden. *(Nichtzutreffendes bitte streichen)*

..... , den ....
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Bedeutung von Schatten 2

2 Wichtige Begriffe und Aspekte **3**
2.1 Wichtige Aspekte bei der Berechnung von Schatten 3
2.1.1 Überlagerung der Schatten 3
2.2 Hard Shadows vs. Soft Shadows 4
2.2.1 Exakte oder Fake-Schatten 4

3 Harte Schatten **5**
3.1 Shadow Mapping 5
3.2 Shadow Volumes 5

4 Soft-Shadows-Verfahren **7**
4.1 Bildbasierte Verfahren 7
4.1.1 Combination of several point-based shadow images 7
4.1.2 Layered Attenuation Maps 8
4.2 Objektbasierte Verfahren 8
4.2.1 Kombination mehrerer harter Schatten 8

5 Soft-Shadow-Volumes **8**
5.1 Berechnung der Penumbra-Wedges 9
5.2 Berechnung der Halbschattenwerte 10
5.2.1 Pass 1 10
5.2.2 Pass 2 11
5.2.3 Berechnung des verdeckten Bereiches 12

6 Implementierung **14**
6.1 createShadow 14
6.2 createWedges() 15
6.3 calculatePenumbra() 16
6.4 inShadow(segment s) 17
6.5 projectOnLight(Vector point,segment seg) 18
6.6 calculateArea(shortSegment clippedSeg) 19
6.7 drawVisibilityBuffer() 21

7 Beschleunigung des Algorithmus **22**

8 Ergebnisse **22**

1 Einleitung

Diese Studienarbeit beschäftigt sich mit der Berechnung von weichen Schatten. Der Fokus liegt dabei auf dem Soft-Shadow-Volumes Algorithmus von Ulf Assarsson und Tomas Akenine-Möller von der Chalmers University of Technology in Schweden. Zunächst stelle ich die Bedeutung und einige wichtige Aspekte im Zusammenhang mit der Berechnung von Schatten vor. Dann folgt eine Beschreibung der grundlegenden Verfahren zum erstellen von harten Schatten und einige alternative Verfahren für weiche Schatten. Im zweiten Teil der Studienarbeit gehe ich im Detail auf den Soft-Shadow-Volumes Algorithmus ein und stelle meine Implementierung vor.

1.1 Bedeutung von Schatten

Ein Schatten ist [6]

- ein Raum, der begrenzt wird, durch die von einer Lichtquelle abgewandte Seite eines lichtdämpfenden Gegenstandes und des durch dieselbe Lichtquelle erzeugte ungedämpfte Licht.
- das durch die Lichtquelle und den Gegenstand erzeugte Projektionsbild.

Schatten sind ein zentraler Faktor in der menschlichen Wahrnehmung ihrer Umgebung [3]. Sie helfen uns die relative Position und Größe eines Objektes in einer Szene zu verstehen (Abb. 1). Sie helfen uns die Geometrie und das Relief eines Objektes zu verstehen, auf welches sie fallen(Abb. 2) und sie helfen auch beim Verstehen der Geometrie des Schattenspenders, in dem sie Hinweise auf sonst vielleicht nicht sichtbare Bereiche geben (Abb. 3). Alles in allem können Schatten entscheidend für das Verständnis einer 3D-Szene sein.

Abbildung 1: Schatten geben Informationen über die relative Position eines Objektes

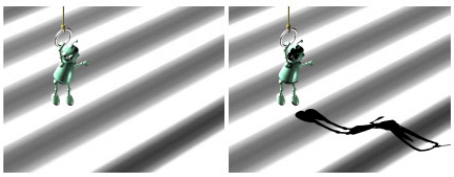


Abbildung 2: Schatten geben Hinweise auf die Geometrie des Receivers.

Abbildung 3: Schatten geben Informationen über die Geometrie des Occluders die sonst nicht zu sehen ist

2 Wichtige Begriffe und Aspekte

Zunächst möchte ich einige Begriffe erklären, auf die ich im weiteren Verlauf dieser Ausarbeitung zurückgreifen werde.

- Punktlichquelle – Eine Lichtquelle deren Lichtstrahlen aus einem einzigen Punkt ausgehen.
- Flächige Lichtquelle – Eine Lichtquelle deren Lichtstrahlen von einer Fläche aus ausgehen.
- Occluder – Ein Objekt dass die Lichtquelle verdeckt und damit einen Schatten wirft.
- Receiver – Das Objekt auf das der Schatten geworfen wird.
- Kernschatten(Umbra) – Der Bereich eines Schattens in dem die Lichtquelle vollständig Verdeckt ist.
- Halbschatten(Penumbra)– Der Bereich eines Schattens in dem die Lichtquelle nur teilweise Verdeckt ist.
- Sampling Points – In diesem Zusammenhang: Stichprobenartige Abtastpunkte auf der Lichtquelle.
- Bounding Volume - Ein geschlossenes Volumen welches ein oder mehrere Objekte einschließt.

2.1 Wichtige Aspekte bei der Berechnung von Schatten

2.1.1 Überlagerung der Schatten

Ein Problem bei der Schattenberechnung stellt die Kombination von Schatten von verschiedenen Occludern dar. Während wir bei Punktlichtquellen einfach eine Überlagerung der einzelnen Schatten verwenden können, ist es bei flächigen Lichtquellen komplizierter. Da wir bei flächigen Lichtquellen Halbschattenbereiche haben, kann man nicht mehr einfach eine Kombination der einzelnen Schatten verwenden, denn es kann auch Punkte geben, die für die einzelnen Occluder im Penumbra-Bereich liegen, zusammengenommen sich aber in der Umbra-Region befinden(siehe Abb. 4) [3].

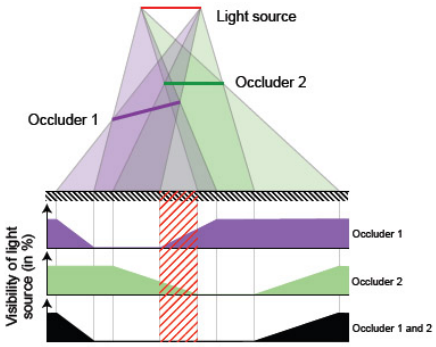


Abbildung 4: Der Schatten von zwei Occludern ist nicht einfach ihre Kombination.

2.2 Hard Shadows vs. Soft Shadows

Ein Schatten wird häufig als ein binärer Status verstanden. Ein Objekt ist entweder im Schatten oder nicht. Diese Wahrnehmung entspricht jedoch nur bei Punktlichtquellen der Wahrheit. In der Realität hat man es jedoch fast ausschließlich mit flächigen Lichtquellen zu tun.[3] So ist ja schon die Sonne, die wohl am häufigsten anzutreffende Lichtquelle, keine Punktlichtquelle. Auch jede Glühbirne ist eine flächige Lichtquelle und diese produzieren keine harten „binären“ Schatten, sondern weiche Verläufe an den Schattenrändern. Im Gegensatz zu einer Punktlichtquelle, die von jedem im Schatten liegenden Punkt aus, entweder sichtbar ist oder nicht, können flächige Lichtquellen auch teilweise sichtbar sein und einen Punkt schwächer beleuchten als einen anderen. Damit läge so ein Punkt im „Halbschatten“. Das berechnen des Halbschatten-Bereichs ist das Hauptproblem bei der Simulation von weichen Schatten.

Weiche Schatten sind ganz offensichtlich wesentlich realistischer als harte Schatten und der „Weichheits“-Grad hängt ganz enorm vom Abstand des Occluders vom Receiver ab/citesurvey. Um den höchstmöglichen Realitätsgrad einer Szene zu erreichen ist es unbedingt erforderlich weiche Schatten simulieren und berechnen zu können. Mittlerweile gibt es Verfahren, die die Berecnung von Weichen Schatten, mit Hilfe der Grafikhardware, in Echtzeit ermöglichen und somit können diese in Spielen oder anderen Echtzeit-Anwendungen eingesetzt werden. Einige dieser Verfahren werde ich weiter unten vorstellen.

2.2.1 Exakte oder Fake-Schatten

Um genau berechnen zu können, welchen Bereich ein Schatten einnimmt, müssen wir genau wissen welches Licht der Occluder auf dem Weg zum Receiver blockiert. Dafür muss man alle Teile des Occluders kennen, die, von zumindest einem Punkt der flächigen Lichtquelle aus, zu sehen sind. Für Echtzeitanwendungen ist dieser Ansatz oft jedoch zu langsam, weswegen man meistens einfach das Zentrum der Lichtquelle für die Sichtbarkeitsberechnung verwendet, was zwar physikalisch nicht absolut korrekt ist, im Endergebnis aber normalerweise wenig auffällt. Es kann jedoch zu Problemen bei besonders großen oder weit entfernten Lichtquellen führen, die einen Occluder eventuell sogar von verschiedenen Seiten gleichzeitig beleuchten.

3 Harte Schatten

Die zwei Standard-Verfahren um harte Schatten darzustellen sind: das **Shadow-Mapping** und das **Shadow-Volumes** Verfahren. Beiden dient als Lichtquelle eine Punktlichtquelle und beide Verfahren sind mit der heutigen Hardware problemlos in Echtzeit realisierbar. Die später vorgestellten Soft-Shadow-Algorithmen bauen darauf auf.

3.1 Shadow Mapping

Das Verfahren besteht grundsätzlich aus zwei Pässen.

- Zunächst wird die Szene von der Lichtquelle aus gerendert und die Tiefeninformationen des Ergebnis-Bildes werden in einer *Shadow Map* gespeichert. (Abb. 3.1)
- Im zweiten Pass wird die Szene vom Betrachter aus gerendert. Für jedes Pixel wird die Distanz zur Lichtquelle berechnet und mit dem Tiefenwert in der Shadow Map abgeglichen. Wenn die Entfernung zwischen dem Objekt und der Lichtquelle größer ist, als die Distanz, die in der Shadow Map gespeichert ist, befindet sich das Objekt im Schatten und die Farbe wird entsprechend modifiziert. (Abb. 3.1)

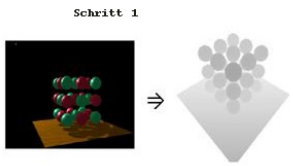


Abbildung 5: Schritt 1

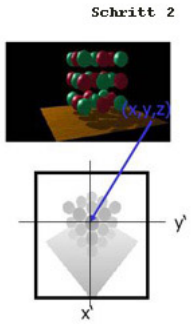


Abbildung 6: Schritt 2

3.2 Shadow Volumes

Das Shadow Volumes Verfahren ist ein rein geometrischer Ansatz. Es besteht aus zwei Schritten.

- Zunächst benötigt man die Silhouette des Occluders, von der Lichtquelle aus gesehen. Der einfachste Weg diese Silhouette zu bekommen, ist nach Nachbardreiecken zu suchen, die jeweils zur Lichtquelle hin und von der Lichtquelle weg zeigen. Die Silhouette bilden die Kanten zwischen diesen Dreiecken. Diese Silhouette wird entlang der Lichtrichtung extrudiert. Für jede Kante bilden wir somit eine Halbebene die von der Silhouettenkante und der Lichtquelle definiert wird. Und alle diese Halbebenen bilden das *Shadow Volume*. Um

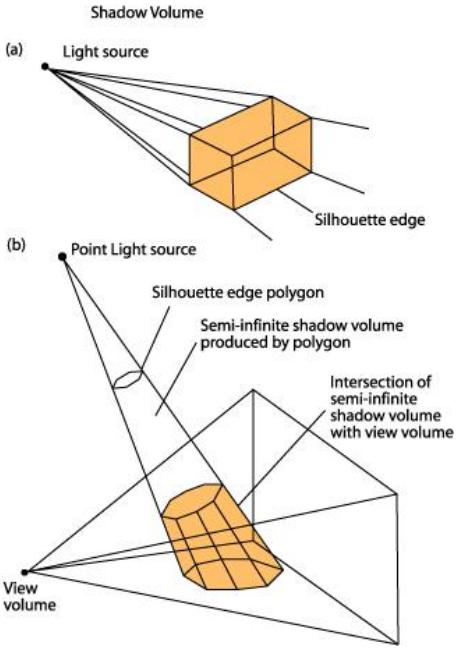
festzustellen, ob ein Punkt im Schatten liegt oder nicht, braucht man nur zu wissen ob ein Punkt innerhalb des Shadow Volume liegt oder nicht.

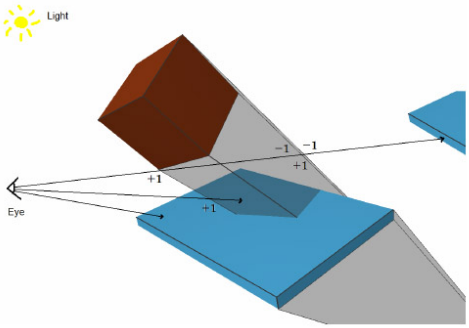
- Im zweiten Schritt zählen wir nun wie viele Flächen aus dem Shadow Volume zwischen jedem sichtbaren Punkt und dem Betrachter liegen. Dabei inkrementiert man einen Zähler für jede Vorderseite und dekrementiert ihn für jede Rückseite. Wenn der Zähler am Ende positiv ist befinden wir uns innerhalb des Schattens.

  Dieser Schritt lässt sich gut mit Hilfe des Stencil-Buffer realisieren in dem man Vorder- und Rückseiten des Shadow Volumes in den Stencil-Buffer rendert und dabei bei den Vorderseiten den Stencil-Buffer inkrementiert und bei Rückseiten dekrementiert. Am Ende hat man im Stencil-Buffer genau den Schattenbereich ausmaskiert.

Der kurz gefasste Algorithmus sieht folgendermaßen aus:

- Die Szene nur mit ambientem Licht rendern.
- Shadow Volume berechnen und in den Stencil-Buffer rendern.
- Die Szene mit dem Stencil-Test rendern und den maskierten Bereich abdunkeln.





4 Soft-Shadows-Verfahren

Es gibt grundsätzlich zwei Arten von schnellen(möglichst echtzeitfähigen) Algorithmen um weiche Schatten zu berechnen: objektbasierte Verfahren und Bild-basierte Verfahren. Bildbasierte Verfahren sind dabei auf dem Shadow-Mapping Ansatz aufgebaut, während objektbasierte Verfahren auf dem Shadow-Volumes Verfahren aufbauen. Einige dieser Verfahren möchte ich hier beispielhaft vorstellen.

4.1 Bildbasierte Verfahren

4.1.1 Combination of several point-based shadow images

[4, 3]

Ein relativ nahe liegender Ansatz um einen weichen Schatten zu berechnen, ist ihn aus einer Kombination von harten Schatten zu gewinnen. Für die harten Schatten verwenden wir Punktlichtquellen. Weiche Schatten entstehen naturgemäß, bei flächigen Lichtquellen. Daher nehmen wir Sampling-Points auf der flächigen Lichtquelle, um mit Hilfe dieser, harte Schatten zu erstellen. Bei diesem Verfahren wird von vornherein bestimmt welche Objekte Occluder und welche Receiver sind. Für jeden Sampling-Point wird die Szene vom Sampling-Point aus in einen Buffer gerendert (Occlusion-Map), wobei für den Receiver eine 0 in den Buffer geschrieben wird und für den Occluder eine 1. Diese Buffer werden dann zu einer *Attenuation-Map* zusammengefügt, in der für jedes Pixel gespeichert ist, wie viele Sampling-Punkte der Lichtquelle für dieses Pixel verdeckt bzw. nicht Sichtbar sind. Abhängig von der Sichtbarkeit der Lichtquelle hat man in der Attenuation-Map aus den einzelnen harten Schatten kombiniert, einen Wert für den weichen Schatten stehen. Beim Rendern wird diese mit den Standard-Texturen kombiniert. (siehe Abb. 7)

Das Hauptproblem dieses Verfahrens ist, dass man P*S rendering Pässe benötigt. Wobei P die Anzahl der Receiver und S die Anzahl der Sampling-Punkte ist. Dabei nimmt die Qualität bei wenig Sampling-Punkten stark ab und man sieht die einzelnen harten Schatten. Von daher ist es in der Praxis am besten wenn man einen einzigen Receiver verwendet damit das verfahren in Echtzeit funktioniert.

Abbildung 7: links: Die Occlusion-Map für einen einzelnen Sampling-Point. mitte: Die Attenuation-Map mit 4 Samples. rechts: Die Attenuation Map mit 64 Samples.

4.1.2 Layered Attenuation Maps

[5, 3]

Ein weiteres bild-basiertes Verfahren ist das Layered-Attenuation-Maps Verfahren. Es basiert auf der in Kapitel 4.1.1 beschriebenen Methode und beginnt ebenso, indem man Sampling-Points auf der Lichtquelle nimmt und damit eine modifizierte Attenuation-Map vorberechnet:

Das Verfahren läuft wie folgt ab[3]:

- Für jeden Sampling-Point wird die Szene entlang der Lichtnormalen gerendert.
- Für diese Bilder wird dann die Verzerrung, die sich aus den unterschiedlichen Positionen der Sampling-Points ergibt, bezüglich eines zentralen Referenzpunktes, welcher das Zentrum der Lichtquelle ist, rausgerechnet.
- Nun werden die Bilder nacheinander in eine *Layered-depth image (LDI)* eingefügt, in dem Bereiche bezüglich ihrer Z-Buffer Tiefeninformation zu *Layern* zusammengefasst werden. Jeder Layer besitzt einen Tiefenwert und einen Integer-Zähler. Wenn ein Tiefenwert sich bereits in der LDI befindet wird der Zähler um eins erhöht. Damit wird die Anzahl der Sampling-Punkte gezählt, von denen aus ein Pixel zu sehen ist.
- Zuletzt werden die Werte der Zähler durch die Gesamtzahl der Sampling-Points geteilt und wir erhalten damit den Prozentsatz der Verschattung für diese Punkte und die Layerd-Attenuation-Map.

Das rendern der Szene wird danach in zwei Pässen durchgeführt:

- Zunächst wird die Szene mit der Standard-Beleuchtung und Texturen gerendert, womit alle für den Betrachter unsichtbaren Objekte eliminiert werden.
- Im zweiten Schritt wird jeder Pixel des Bildes aus dem ersten Pass mit dem korrespondierenden Punkt auf der Layered Attenuation Map abgeglichen. Wenn sich der Punkt darin befindet, wird er dem Verschattungswert entsprechend abgedunkelt, wenn nicht, ist er von der Lichtquelle aus nicht zu sehen und liegt im Kernschattenbereich.

4.2 Objektbasierte Verfahren

4.2.1 Kombination mehrerer harter Schatten

Die einfachste objektbasierte Methode um weiche Schatten zu zeichnen, ist mehrere Sampling-Points auf der flächigen Lichtquelle zu nehmen, für jeden dieser Punkte einen harten Schatten zu berechnen und dann zwischen den entstandenen Bildern zu interpolieren. Leider hat man hierbei das Problem das man viele Sampling-Punkte benötigt um ein zufriedenstellendes Ergebnis zu erhalten. Diese Methode ist nicht unbedingt effizient.

5 Soft-Shadow-Volumes

Der Fokus dieser Studienarbeit liegt auf dem Soft-Shadow-Volumes Verfahren, welches 2002 von Tomas Akenine-Möller und Ulf Assarsson in dem Paper „Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges“ [1] vorgestellt und

2003 weiterentwickelt [2] wurde. Der Algorithmus basiert auf dem Shadow-Volumes Verfahren, welches auch ein Teil des Soft-Shadow-Volumes Algorithmus ist und verwendet wird um den Umbra Bereich zu bestimmen. Die Idee ist, von einem einzelnen Sampling Punkt der Lichtquelle(meistens das Zentrum) aus, die Silhouette des Occluders zu berechnen und basierend auf der Silhouette *Penumbra-Wedges* zu bilden, welche den Halbschattenbereich markieren. Diese Penumbra-Wedges, die aus jeder Kante der Silhouette gebildet werden, stellen Keile dar, mit der Silhouetten-Kante als Spitze, die den Halbschattenbereich auf dem Receiver umfassen.

Die genauen Halbschatten-Werte werden dann aus der Projektion des Hard-Schadow-Volume auf die Lichtquelle für jede Penumbra-Wedge bestimmt und in einen Visibility-Buffer geschrieben in dem auch schon der Kernschatten gespeichert ist. Am ende des Prozesses befindet sich im Visibility-Buffer ein weicher Schatten.

Der Einfachheit halber verwenden wir rechtwinklige Lichtquellen, da damit die Projektion und Flächenberechnung einfacher sind, der Algorithmus funktioniert jedoch mit allen Lichtquellen-Formen.

5.1 Berechnung der Penumbra-Wedges

Wie beim Shadow-Volumes Algorithmus benötigt man zunächst die Silhouette des Occluders, um den Schattenbereich zu bestimmen. Diese kann man, ähnlich wie in Kapitel 3.1 beschrieben, finden, indem man nach Nachbardreiecken sucht, bei denen ein Dreieck zur Lichtquelle hin gewandt ist und das andere von der Lcihtquelle abgewandt. Das ist zum Beispiel über das Skalarprodukt der Normalen des Dreiecks mit der Lichtnormalen festzustellen. Die Kante zwischen diesen Dreiecken ist eine Kante der Silhouette. Wenn man die Silhouette der Objekte hat geht es nun darum, daraus die Penumbra-Wedges zu berechnen. Der exakte Halbschattenbereich zu einer Silhouettenkante wäre ein Projektionskegel der Lichtquelle durch die Kante (Abb.. 8).

Für schnelle bzw. Echtzeitanwendungen ist es zu komplex und ineffizient die exakte Wedge zu berechnen. Für den hier vorgestellten Algorithmus reicht es auch wenn wir ein Bounding-Volume nutzen, welches den Halbschattenbereich komplett umfasst. Ein solches Bounding-Volume wird durch vier Ebenen definiert *vorne*, *hinten*, *links* und *rechts*. Diese Bounding-Volumes für die exakten Wedges sind unsere Penumbra-Wedges und wesentlich leichter zu berechnen als eine exakte Wedge.

Eine Penumbra-Wedge berechnen wir aus jeder Silhouettenkante. Eine Silhouettenkante besitzt einen Anfangs- und Endpunkt e_0 und e_1. Wir finden zunächst denjenigen dieser zwei Punkte, welcher am nächsten zur Lichtquelle liegt. Sollte es der Punkt e_1 sein wird auf Verbindungslinie zwischen dem Sampling-Punkt unserer Lichtquelle und e_0 ein neuer Punkt e_0' konstruiert, welcher den selben Abstand von der Lichtquelle hat wie e_1. Falls e_0 am nächsten zur Lichtquelle liegt wird entsprechend ein Punkt e_1' konstruiert. Gehen wir im weiteren Verlauf davon aus, dass e_1 am nächsten zur Lichtquelle liegt. Dann bildet die Kante zwischen e_1 und e_0' die Spitze des Keils, der die Penumbra-Wedge darstellt. In dem wir einen Punkt der Silhouetten-Kante verschieben, umfasst die Penumbra-Wedge nicht mehr den exakten Halbschatten-Bereich sondern ein Bounding-Volume, welches diesen beinhaltet.

Da wir den Punkt immer nur näher zur Lichtquelle hin verschieben, ist dieser Bereich auf jeden Fall größer als der exakte und beinhaltet diesen von daher. In einem späteren Schritt wird, bei der Berechnung der Verschattung, der zusätzlich innerhalb der Wedge eingeschlossene Bereich aber dennoch nicht beschattet. Ein

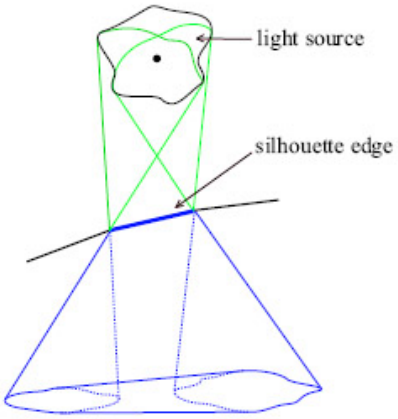


Abbildung 8: Exakter Halbschattenbereich.

Problem kann allerdings entstehen wenn der Abstand der beiden Silhouettenkantenenden von der Lichtquelle sich sehr stark unterscheidet. Dann werden besonders viele zusätzliche Pixel in die Penumbra-Wedge mit einbezogen für die letztendlich unnötige Berechnungen durchgeführt werden müssen.

Nachdem wir die modifizierte Silhouettenkante e_1e_2 bestimmt haben, können wir die vier Ebenen der Penumbra-Wedge bilden. Die vordere und hintere Ebene werden durch die Silhouettenkante gelegt, so, dass sie die Lichtquelle gerade eben am Rande, jeweils auf der einen und auf der anderen Seite der Lichtquelle, berühren. Die rechte Ebene wird so definiert´, dass e_0' und der Vektor der orthogonal zum Vektor e_1e_0' und dem Verbindungsvektor zwischen dem Lichtquellen-Zentrum und e_0' ist, in ihr liegt. Die linke Ebene wird genau so definiert nur entsprechend auf der anderen Seite durch den Punkt e_1. Wir erstellen Polygone, die die Bereiche der Ebenen abdecken die wir benötigen. Nach unten hin muss unsere Wedge weit Genug ausgedehnt sein um wirklich sicher zu stellen dass alle Objekte die im Halbschatten liegen umfasst werden. Damit haben wir zu jeder Silhouettenkante eine Penumbra-Wedge die als Bounding-Volume für den Halbschattenbereich dient (Abb. 9).

5.2 Berechnung der Halbschattenwerte

Für die Pixel die sich innerhalb der Penumbra-Wedge befinden müssen wir nun den Grad der Verschattung für jeden einzelnen sichtbaren Pixel bestimmen. Die Berechnung wird in zwei Pässen durchgeführt und wir benötigen einen zusätzlichen Buffer, den Visibility-Buffer. Dieser Buffer wird zunächst überall auf 0 gesetzt, was volle Beleuchtung bedeutet(Akenine-Möller und Assarsson verwenden an dieser Stelle eine 1 für volle Beleuchtung und 0 für vollen Schatten, was jedoch am Endergebnis nichts ändert).

5.2.1 Pass 1

Als erster Pass wird der harte Schatten in den V-Buffer gerendert. Wir verwenden den Shadow-Volumes Algorithmus und nehmen das Zentrum der Lichtquelle als

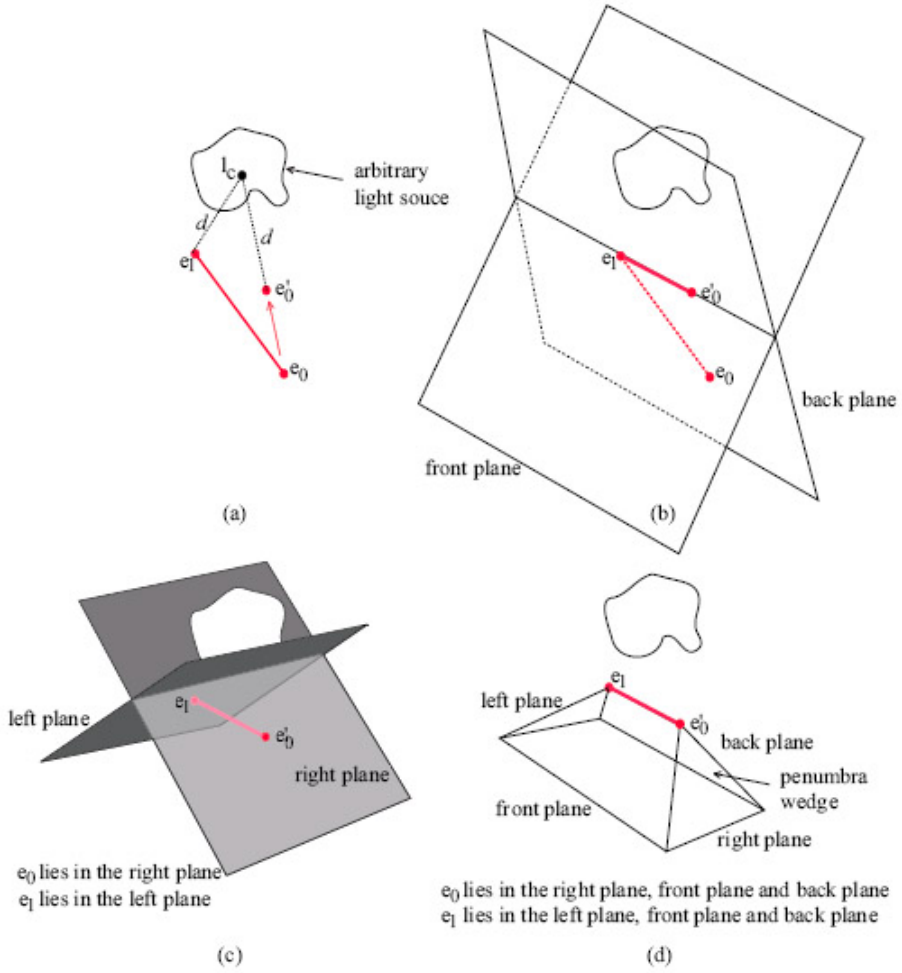


Abbildung 9: Berechnung der Penumbra-Wedge.

Punktlichtquelle für den Algorithmus. Dabei wird für die Vorderseiten eine 1 addiert und für die Rückseiten subtrahiert. Am Ende des Prozesses hat man dann in dem Bereich des harten Kernschattens, für das Zentrum als Punktlichtquelle, überall eine 1 im Visibility-Buffer stehen, volle Verschattung. Es sollte erwähnt werden dass man diesen Bereich nicht als Umbra verwenden kann, denn dieser harte Schatten überdeckt Teile der Penumbra die wir bekommen wenn wir die ganze Lichtquelle und nicht nur das Zentrum verwenden. Die Überbeschattung wird jedoch an anderer Stelle kompensiert.

5.2.2 Pass 2

Der zweite Durchgang widmet sich nun der Penumbra. Wir müssen jetzt die Überbeschattung in dem Bereich des berechneten harten Schattens, der in der Penumbra-Region liegt kompensieren und die Verschattung für die übrigen Punkte der Penumbra berech-

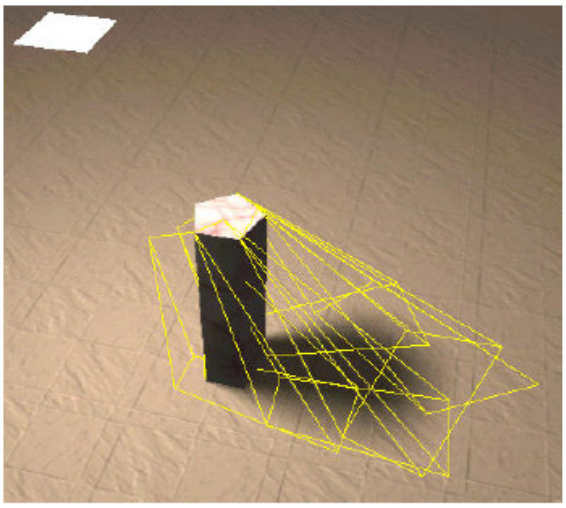


Abbildung 10: Penumbra-Wedges für eine einfache Szene

nen. Für jeden Punkt $\mathbf{p} = (x,y,z)$, der sich innerhalb einer Penumbra-Wege befindet, führen wir eine Sichtbarkeitsberechnung durch. Nehmen wir an ein Betrachter befindet sich im Punkt P und schaut auf die Lichtquelle. Wenn wir uns nun die Seite des Shadow Volumes vorstellen aus Pass 1, würde es einen Teil der Lichtquelle verdecken. Die Sichtbarkeit für den Punkt P ist der von der vom Shadow Volume nicht verdeckte Bereich der Lichtquelle geteilt durch die Gesamtfläche der Lichtquelle.

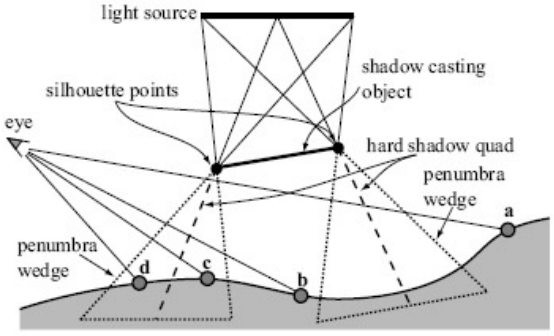


Abbildung 11: Zweidimensionales Beispiel für den Aufbau des Verfahrens

5.2.3 Berechnung des verdeckten Bereiches

Diesen verdeckten Bereich berechnen wir für jede Silhouettenkante einzeln. Für jede Silhouettenkante kann man eine Ebene konstruieren, welche genau den harten Schatten wie im Shadow-Volumes Algorithmus abgrenzt. Sie entspricht dem Teil des Shadow Volume für diese Silhouettenkante, wie es bereits in Pass 1 (Siehe Kap. 5.2.1) berechnet wurde. Diese *Hard-Shadow-Quad* betrachten wir nun vom

Punkt P auf dem Receiver aus, in Richtung der Lichtquelle. Die Hard-Shadow-Quad verdeckt einen Teil der Lichtquelle. Man kann es als eine Projektion der Hard-Shadow-Quad auf die Lichtquelle von Punkt P aus sehen, die die Lichtquelle verdeckt. Für Pixel P wird diese Abdeckung für jede Silhouettenkanten berechnet. Dabei wird der Visibility-Buffer immer aktualisiert.

Man muss allerdings unterscheiden auf welcher Seite der Penumbra-Wedge, bezüglich der aktuellen Silhouettenkante, sich P befindet. Wir teilen die Wedge in einen positiven und in einen negativen Halbraum mit der Hard-Shadow-Quad als Trennebene. Den überschatteten Halbraum, in dem sich die Umbra befindet definieren wir als positiv. Wenn sich Pixel P nun bezüglich der Hard-Shadow Quad im positiven, also überschatteten Halbraum befindet wird der Visibility-Buffer an der Stelle P um die Abdeckung reduziert, befindet er sich im negativen Halbraum wird der Visibility-Buffer um die Abdeckung an der Stelle P erhöht. (Siehe Abb. 12)

So verfährt man mit jedem sichtbaren Pixel innerhalb der Penumbra-Wedge, für jede Silhouettenkante. Damit wird die Überschattung kompensiert und der Halbschatten für die gesamte Penumbra Region berechnet.

Ein Pseudo-Code für die Berechnung der Halbschatten-Werte für eine Wedge sieht wie folgt aus:

```
rasterizeWedge(wedge W, hard shadow quad Q, light L)
for jedes pixel(x,y) innnerhalb der wedge
  p=point(x,y,z)  //z ist der z-Buffer Wert
  if p ist innerhalb der Wedge
     v(p) = projeziereHardShadowQuadUndBerechneAbdeckung(W,p,Q);
     if p im positiven Halbraum von Q
        VisibilityBuffer(x,y) = VisibilityBuffer(x,y) + v(p);
     else
        VisibilityBuffer(x,y) = VisibilityBuffer(x,y) - v(p);
     end;
   end;
end;
```

6 Implementierung

Die Implementierung basiert auf dem Shadow-Volumes Programm von Thorsten Grosch, in dem der standard Shadow-Volumes Algorithmus für harte Schatten umgesetzt wurde. Diesen habe ich um die notwendigen Methoden erweitert, um weiche Schatten mit Hilfe der Penumbra-Wedges (siehe Kap.5) darstellen zu können. Das ganze wurde mit c++ und OpenGL implementiert.

Die grobe Grundstruktur des Programms besteht aus 3 Funktionen

- *createShadow()*
- *createWedges()*
- *calculatePenumbra()*

Innerhalb von *createShadow()* wird der harte Schatten mit dem Shadow-Volumes Algorithmus berechnet und in den Visibility-Buffer geschrieben. Des weiteren werden dort die Silhouettenkanten in einer Liste gespeichert. Dann werden in *createWedges()* die Penumbra Wedges gebildet und in *calculatePenumbra()* werden dann die weichen Schattenwerte berechnet und der weiche Schatten gezeichnet. Der Occluder wird dem Programm bereits triangluiert als ein Array von Dreiecken mitgegeben.

6.1 createShadow

Dieser Teil der Implementierung war bereits vorhanden und wurde nur etwas ergänzt. Hier wird der harte Schatten berechnet. Im ersten Durchgang werden alle Seiten des Schattenvolumens, die zum Betrachter zeigen in den Stencil-Buffer gezeichnet. Dazu werden über Culling die Rückseiten abgeschaltet und der Stencil-Buffer wird für jedes gezeichnete Pixel um eins erhöht. Im zweiten Durchgang werden alle Seiten des Schattenvolumens die vom Betrachter weg zeigen in den Stencil Buffer gezeichnet und die Vorderseiten werden abgeschaltet. Der Stencil-Buffer wird dann für jedes gezeichnete Pixel um eins reduziert. Im Stencil Buffer bleibt am ende eine Maske für den harten Schatten stehen und man braucht nur noch ein bildschirm-füllendes Rechteck zu zeichnen mit dem Stencil Buffer als Maske.

Zum zeichnen des Schattenvolumens wird die Methode *DrawShadowVolume()* verwendet. Dort werden wie in Kap. 3.2 beschrieben die Silhouettenkanten bestimmt. Die Methode wurde modifiziert um die gefundenen Silhouettenkanten in einer Liste zu speichern. Dafür habe ich die Klasse *Segment* erstellt. Die Datenstruktur für eine Silhouettenkante sieht wie folgt aus:

```
class segment{

public:
Vector P1;
Vector P2;
Vector endP1A;
Vector endP2A;
Vector endP1B;
Vector endP2B;
Vector hardShadowP1;
Vector hardShadowP2;
```

```
};
```

hier werden alle benötigten Informationen für die Kante gespeichert. Die einzelnen Segmente werden dann in einer Liste *silhouette* gespeichert. In diesem Schritt haben wir jedoch bislang nur den Anfangs- und Endpunkt P1 und P2. Später werden die Vektoren für die Penumbra-Wedge und Hard-Shadow-Quad eingetragen.

6.2 createWedges()

In dieser Funktion werden die Penumbra-Wedges zu jeder Silhouetten-Kante berechnet und in der *silhouette* gespeichert. Zunächst wird der zur Lichtquelle nächstliegende Punkt bestimmt und der andere wird dann entsprechend auf diese Entfernung zur Lichtquelle hin verschoben. (Zeile 12-29). Nun besitzen wir die Spitze des Keils der Penumbra Wedge. Es sind die Punkte **newP1** und **newP2**. Um die Wedge konstruieren zu können benötigen wir zwei Punkte auf der Lichtquelle so, dass die vordere und die hintere Ebene der Penumbra Wedge die Lichtquelle an den Seiten berührt (siehe Abb. 10.

Diese Punkte bestimme ich durch die Projektion des Verbindungsvektors zwischen dem Lichtzentrum und dem Segmentzentrum auf die Lichtquelle(Zeile 31-35). Der normalisierte Vektor wird dann einfach in beide Richtungen vom Lichtzentrum aus um die halbe Lichtbreite verlängert und wir bekommen, zwar approximierte, aber hinreichend genaue Punkte auf der Lichtquelle(Zeile 37-40). Durch diese beiden Punkte und die Endpunkte der modifizierten Silhouettenkante werden die vordere und die hintere Ebene der Penumbra-Wedge gelegt. Die Ebenen werden als viereckige Polygone gespeichert, wobei die unteren Punkte entsprechend weit nach unten skaliert werden müssen, um die ganze Szene zu erfassen(Zeile 42-46). Aus den unteren Punkten des vorderen und hinteren Polygons und den Endpunkten des Silhouetten-Segments lassen sich nachher auch die Seiten-Polygone der Penumbra Wedge bestimmen.

```
void createWedges()
{
Vector longerDistToLight;
Vector dist1;
Vector dist2;
Vector newP1;
Vector newP2;
float scale = 100;

for( int i=1; i<silhouette.size();i++)
{
        dist1 = silhouette[i].P1-lightPositionModelview;
        dist2 = silhouette[i].P2-lightPositionModelview;

        if(dist1.length() <= dist2.length())
        {
                float length1 = dist1.length();
                dist2.normalize();
                newP1 = silhouette[i].P1;
                newP2 = lightPositionModelview + (length1*dist2);
```

```
        }

        else
        {
            float length2 = dist2.length();
            dist1.normalize();
            newP1 = lightPositionModelview + (length2*dist1);
            newP2 = silhouette[i].P2;
        }

        Vector segmentMiddle =newP1 + ((newP2−newP1)*0.5);
        Vector middleLightDistance = segmentMiddle − lightPositionModelview;

        middleLightDistance[Y]=0;
        middleLightDistance.normalize();

        Vector lightEdgePointA = lightPositionModelview +
        (middleLightDistance * lightWidth * 0.5);
        Vector lightEdgePointB = lightPositionModelview −
        (middleLightDistance * lightWidth * 0.5);

        Vector endP1A = newP1 + (newP1 − lightEdgePointA)*scale;
        Vector endP2A = newP2 + (newP2 − lightEdgePointA)*scale;

        Vector endP1B = newP1 + (newP1 − lightEdgePointB)*scale;
        Vector endP2B = newP2 + (newP2 − lightEdgePointB)*scale;

        Vector hardShadowEnd1 = silhouette[i].P1 +
        (silhouette[i].P1 − lightPositionModelview)*scale;
        Vector hardShadowEnd2 = silhouette[i].P2 +
        (silhouette[i].P2 − lightPositionModelview)*scale;

        silhouette[i].hardShadowP1 = hardShadowEnd1; //Hard-Shadow Grenze
        silhouette[i].hardShadowP2 = hardShadowEnd2;

        silhouette[i].endP1A = endP1A; // WedgeSeite A
        silhouette[i].endP2A = endP2A;

        silhouette[i].endP1B = endP1B; // WedgeSeite B
        silhouette[i].endP2B = endP2B;
    }

    }
```

6.3 calculatePenumbra()

Hier geht man nun über alle Penumbra-Wedges, berechnet die Halbschatten-Werte und schreibt diese in den Visibility-Buffer. Nachdem der Visibility-Buffer mit den endgültigen Werten belegt ist, muss man ihn nur noch darstellen, was die Funktion drawVisibilityBuffer() übernimmt. Die Hauptarbeit wird durch den Aufruf der Funktion *inShadow(segment s)* ausgeführt.

6.4 inShadow(segment s)

Diese Funktion kriegt ein Segment aus der Silhouette mit der dazugehörigen Penumbra-Wedge übergeben. Hier werden die Halbschattenwerte berechnet und in den Visibility-Buffer geschrieben. Um herauszufinden welche Pixel sich innerhalb der Penumbra-Wedge befinden verwenden wir einfach das selbe Prinzip wie bei Shadow-Volumes in dem man die Vorder- und Rückseiten der Wedge in den Stencil-Buffer rendert. Am ende ist im Stencil-Buffer genau der Halbschattenbereich für diese Wedge gespeichert. Der wird ausgelesen und in den *wedgeBuffer* kopiert (Zeilen 4-20).

Damit haben wir im wedgeBuffer die Pixel stehen, für die wir nun die Abdeckung der Lichtquelle 5.2.3 berechnen müssen. Um das zu tun werden für jedes Pixel, bei dem eine 1 im wedgeBuffer eingetragen ist, die zugehörigen Weltkoordinaten berechnet, denn im wedgeBuffer haben wir nur die Bildschirmkoordinaten. Diese werden aus den Pixelkoordinaten und dem zugehörigen Z-Buffer Wert berechnet(Zeilen 38-50).

Um für diesen Punkt dann die Abdeckung zu berechnen werden zwei Funktionen ausgeführt: *projectOnLight(Vector point, segment seg)* und *calculateArea(shortSegment clippedSeg)*. In *projectOnLight* wird das Segment s für jedes Pixel, das im wedge-Buffer steht, auf die Lichtquelle projiziert und in *calculateArea* wird dann die entsprechende Projektion des Hard-Shadow-Quad auf die Lichtquelle bestimmt und die Fläche, die sie dort einnimmt berechnet,(Zeilen 53-57). Anschliessend müssen nur noch die Werte im Visibility-Buffer aktualisiert werden. Um zu wissen ob sich der Pixel im positiven oder negativen Halbraum bezüglich der Hard-Shadow-Quad befindet, nehme ich einfach eine Kopie des Stencil-Buffer, nach dem der harte Schatten im ersten Pass berechnet wurde, als Referenz (siehe 6.1). Dann werden die Werte im Visibility-Buffer entsprechend erhöht oder reduziert (Zeilen 60-65).

```
... hier wird der Bereich innerhalb der Penumbra−Wedge in den Stencil−Buffer geschrieben...

float wdata[width*height];
GLubyte wdata2[width*height];

// den Stencil Buffer auslesen
glReadPixels(0,0,width,height,GL_STENCIL_INDEX,GL_UNSIGNED_BYTE,wdata2);

int counter = 0;

// den Bereich innerhalb der Wedge aus dem Stencil Buffer in den wedgeBuffer kopieren.
for(int i=0;i<height;i++)
{
        for(int j=0;j<width;j++)
        {
                wedgeBuffer[i][j] = wdata2[counter];
                counter++;
        }
}

Vector testpoint;
float percentage;
shortSegment clippedPoints;

```

```
GLint viewport[4];
GLdouble mvmatrix[16], projmatrix[16];
GLdouble wx,wy,wz;

float zdata[1];

for(int i = 0; i<height; i++)
{
        for (int j = 0; j<width; j++)
        {
        if(wedgeBuffer[i][j] != 0)
                {
                //z-Buffer auslesen
                glReadPixels(i,j,1,1,GL_DEPTH_COMPONENT,GL_FLOAT,zdata);

                glGetIntegerv (GL_VIEWPORT, viewport);
                glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
                glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);

                //Weltkoordinaten für die Bildschirmkoordinaten des Pixels i,j bestimmen
                gluUnProject ((GLfloat) j, (GLfloat) i, zdata[0], mvmatrix, projmatrix,
                                viewport, &wx, &wy, &wz);
                testpoint[X] = wx;
                testpoint[Y] = wy;
                testpoint[Z] = wz;

                //Segment aud die Lichtquellenebene projezieren
                clippedPoints = projectOnLight(testpoint,s);

                //Falls die Projektion nicht komplett ausserhalb der Lichtquelle liegt die Verdeckung berechnen
                if (clippedPoints.out == false) {percentage = calculateArea(clippedPoints);}
                else percentage = 0;

                // Visibility-Buffer aktualisieren
                if(hardShadowBuffer[i][j]==1)
                //falls im überbestimmten Bereich subtrahieren sonst addieren
                visibilityBuffer[i][j] -= percentage;
                else
                visibilityBuffer[i][j] += percentage;
                }
        }
}
...hier werden openGL-States zurückgesetzt...
}
```

6.5 projectOnLight(Vector point,segment seg)

Um das Segment auf die Lichtquelle zu projizieren bildet man zwei Geraden durch den Punkt P auf dem Receiver und die Enden des Segmentes. Dann werden die Schnittpunkte der Geraden mit der Ebene, in der das Licht liegt, gebildet und

wir haben die Projektion des Segments. Da wird nur die Punkte des projizierten Segments brauchen die in der Lichtquelle liegen wird das Segment noch entsprechend „beschnitten“. Dafür habe ich das Cohen-Sutherland Clipping Verfahren verwendet[7].

```
shortSegment projectOnLight(Vector point,segment seg)
{
        Vector lightNormal;
        lightNormal[X] = 0;
        lightNormal[Y] = -1;
        lightNormal[Z] = 0;

        //Projeziere die Punkte auf die Decke mit Hilfe von Ebenen-Gerade-Schnitt
        Vector projP1 = intersectPlaneLine(lightPositionModelview,lightNormal,point,seg.P1);
        Vector projP2 = intersectPlaneLine(lightPositionModelview,lightNormal,point,seg.P2);

        //die Projezierten Punkte werden an der Lichtquelle mit Cohen-Sutherland geclippt
        shortSegment clippedSeg;
        Clipping clipObject;
        clippedSeg = clipObject.clip(projP1,projP2,lightPositionModelview,lightWidth);

        return clippedSeg;
}
```

6.6 calculateArea(shortSegment clippedSeg)

Die Funktion bekommt das geclippte projizierte Segment aus *projectOnLight* übergeben und es muss damit der Prozentsatz der Fläche berechnet werden, die das projizierte Hard-Shadow-Quad auf der Lichtquelle bedeckt. Es ist allerdings nicht nötig wirklich die Hard-Shadow-Quad zu projizieren, sondern es reicht schon die Projektion des Segments zu haben, was in *projectOnLight* ja schon berechnet wurde. Bei der Berechnung des harten Schattens sind wir von der Mitte der Lichtquelle ausgegangen. Deswegen kann man die Projektion der Hard-Shadow-Quad bilden, in dem man Verbindungsgeraden von der Mitte der Lichtquelle durch die Endpunkte des projizierten Segments bildet. Diese Verbindungsgeraden schliessen eine Fläche ein. Die Schnittfläche dieses Bereiches und der Lichtquelle ist unsere gesuchte Abdeckung (Abb. 12).

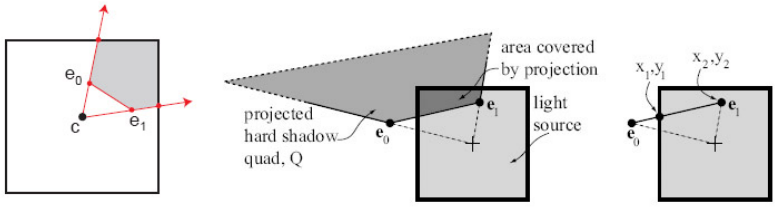


Abbildung 12: Die Projektion der Hard-Shadow-Quad auf die Lichtquelle

Um die Größe der Fläche zu bestimmten ist es am einfachsten sie zu triangulieren und die Summe der Flächen der Dreiecke zu bestimmen, an die man ja über das Kreuzprodukt sehr leicht ran kommt. Das Hauptproblem liegt darin, herauszufinden wie die verdeckte Fläche innerhalb der Lichtquelle liegt, damit wir die Geometrie dieser Fläche bestimmen können um sie dann zu triangulieren. Es sind verschiedene Fälle möglich wie die geclippten projizierten Punkte des Segments auf der Lichtquelle liegen könnten:

- Beide Punkte liegen innerhalb des Segments.
- Ein Punkt liegt am Rand einer innerhalb des Segments.
  - so dass die Schnittpunkte mit den Rändern der Lichtquelle auf der selben Seite liegen
  - auf angrenzenden Seiten liegen
  - auf entgegengesetzten Seiten liegen
- Beide Punkte liegen an den Rändern des Segments. Für diesen Fall muss man noch weiter unterscheiden
  - Beide liegen an entgegengesetzten Seiten
  - Beide liegen an angrenzenden Seiten.

Von der Lage des Segments innerhalb der Lichtquelle und der Schnittpunkte der Verbindungsgeraden mit den Rändern ist die Geometrie der Schnittfläche abhängig. Um immer die Lage eines Punktes bezüglich der Lichtquelle bestimmen zu können verwende ich die Funktion lightSideCode(Vector clippedPoint). Diese bestimmt durch Segment-Schnitttests wie die Lage des Punktes bezüglich der vier Seiten der Lichtquelle ist und gibt einen Code zurück. 12, 23, 34, 41 für die vier Seiten und 111, falls der Punkt innerhalb der Lichtquelle liegt. Damit kann man hinterher auch bestimmen ob die Punkte auf angrenzenden , entgegegengesetzten oder auf der selben Seite liegen und die Fälle unterscheiden. Falls LSC1 der Code von einem und LSC2 von einem anderen Punkt ist gilt:

- if (LSC1 == LSC2 == 111) beide Punkte liegen innerhalb.
- else if (LSC1 + LSC2 > 111) ein Punkt liegt innerhalb.

- else if (LSC1 == LSC2) Punkte liegen auf der selben Seite.
- else if (LSC1 + LSC2 = gerade) Punkte liegen auf entgegengesetzten Seiten.
- else if (LSC1 + LSC2 = ungerade) Punkte liegen auf angrenzenden Seiten.

Falls ein oder zwei Punkte innerhalb der Lichtquelle liegen bildet man, wie beschrieben, den Schnittpunkt der Verbindungsgeraden zur Mitte mit dem Rand der Lichtquelle, für den man dann auch den Light-Side-Code bestimmt. Als nächstes ist es nötig für die einzelnen Fälle zu überprüfen wie viele (keine, eine oder zwei) Ecken der Lichtquelle innerhalb des verdeckten Bereiches liegen, da diese für die Geometrie der Schnittfläche ebenfalls essentiell sind. Das kann man mit einem Segment-Schnittest lösen. Man bildet Segmente von der Mitte zu den Ecken der Lichtquelle und macht einen Schnitttest mit dem Segment zwischen den geclippten Punkten. Falls es einen Schnittpunkt gibt liegt die Ecke innerhalb des abgedeckten Bereiches. Am Ende hat man die gesamte Geometrie des Abgedeckten Bereiches (Die geclippten Silhouetten-Kante-Punkte, die eventuellen Schnittpunkte mit den Rändern und die Lichtquellen-Ecken). Damit lässt sich diese Fläche nun ganz leicht in Dreiecke zerlegen und ihre Größe berechnen.

6.7 drawVisibilityBuffer()

In dieser Funktion muss der, bereits mit Werten aufgefüllte, Visibility-Buffer in den Frame-Buffer geschrieben werden. Mit glReadPixels() wird der Color-Buffer ausgelesen, dann mit den Werten aus dem Visibility-Buffer verrechnet und mit glDrawPixels in den Frame-Buffer geschrieben.

```
void drawVisibilityBuffer(){

unsigned char* screen = new unsigned char[height*width*4];

glReadPixels(0,0,width,height,GL_RGBA,GL_UNSIGNED_BYTE,screen);

int counter = 3;

for(int i=0;i<height;i++)
        {
        for(int j=0;j<width;j++)
                {
                if(visibilityBuffer[i][j]<0){visibilityBuffer[i][j]=0;}
                screen[counter-1]=screen[counter-1]*(1-visibilityBuffer[i][j]);
                screen[counter-2]=screen[counter-2]*(1-visibilityBuffer[i][j]);
                screen[counter-3]=screen[counter-3]*(1-visibilityBuffer[i][j]);
                counter+=4;
                }

        }
glDrawPixels(width,height,GL_RGBA,GL_UNSIGNED_BYTE,screen);
}
```

7 Beschleunigung des Algorithmus

Der Algorithmus kann mit Hilfe vorberechneter Texturen wesentlich effizienter gemacht werden[2]. Der Verschattungs-Wert im Visibility-Buffer für einen Punkt **P**und eine Penumbra-Wedge hängt von der Projektion der Silhouettenkante auf die Lichtquelle ab (siehe 5.2.3 und 6.6). Wenn wir die Kante projiziert und gegen die Lichtquelle geclippt haben, sind die beiden Endpunkte der geclippten Kante (x_1, y_1) und (x_2, y_2). Mit diesen Koordinaten ist es möglich eine vierdimensionale Lookup-Tebelle zu erstellen so, dass $f(x_1, y_1, x_2, y_2)$ die Verschattung bezüglich einer bestimmten Wedge zurückgibt.

Wir diskretisieren die Lichtquelle in $n \times n$ Felder. Ein Ende der projizierten Kante liegt dann in einem dieser Felder($x_1 = a, y_1 = b$). Im nächsten Schritt wird zu Feld (a,b) eine $n\times$ Sub-Textur erstellt, in der jedes Feld die Position des anderen Endes der Kante repräsentiert(x_1, y_1, x_2, y_2). Für jedes der Felder wird dann, für die entsprechende x_1, y_1, x_2, y_2 Position, die Sichtbarkeit vorberechnet. Insgesamt werden $n \times n$ solcher $n \times n$ Texturen vorberechnet und in einer einzigen zweidimensionalen Textur gespeichert. Zur Echtzeit braucht man dann nur die vorberechneten Werte für eine Kante aus der Textur auszulesen und hat die Verschattungs-Werte für den Visibility-Buffer. Assarsson und Akenine-Möller schlagen vor n=32 zu benutzen was eine $1024n \times 1024$ Textur ergibt. Mit einem Pentium4 1.7 GHz Prozessor dauerte die Vorberechnung dieser Texturen etwa 3 Minuten.

Um den Soft-Shadow-Volumes Algorithmus Echtzeitfähig zu machen ist es allerdings unvermeidlich Teile der Berechnung auf die Grafik-Hardware auszulagern, was mir leider im Zeitrahmen dieser Studienarbeit nicht mehr möglich war.

8 Ergebnisse

Assarsson und Akenine-Möller haben den Algorithmus mit einem 1024-Sampling-Points Bild verglichen, welches mit dem Verfahren, wie in Kap. 4.2.1 beschrieben, erstellt wurde und bei der hohen Anzahl an Sampling-Points eine gute Qualität besitzt. Wie man sehen kann ist das Ergebniss sehr zufriedenstellend.

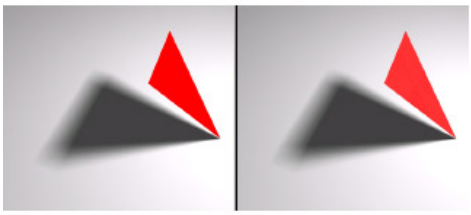
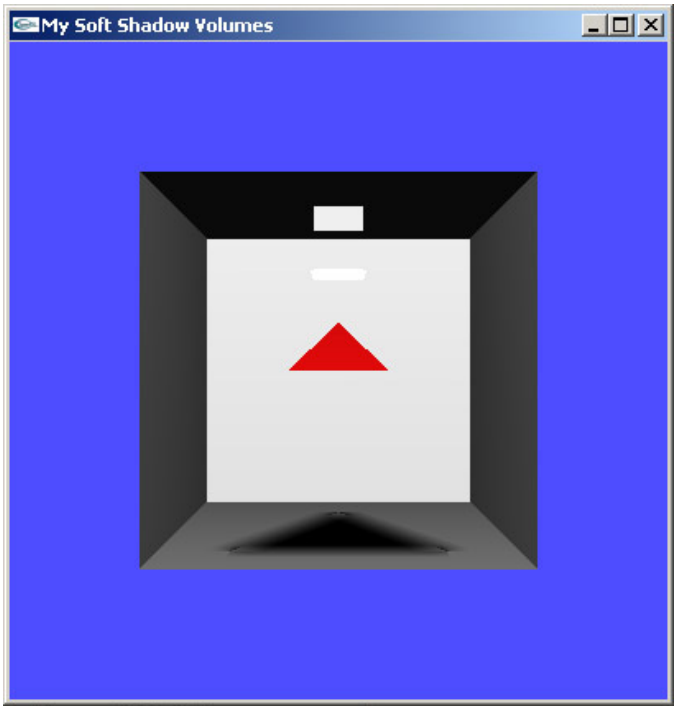
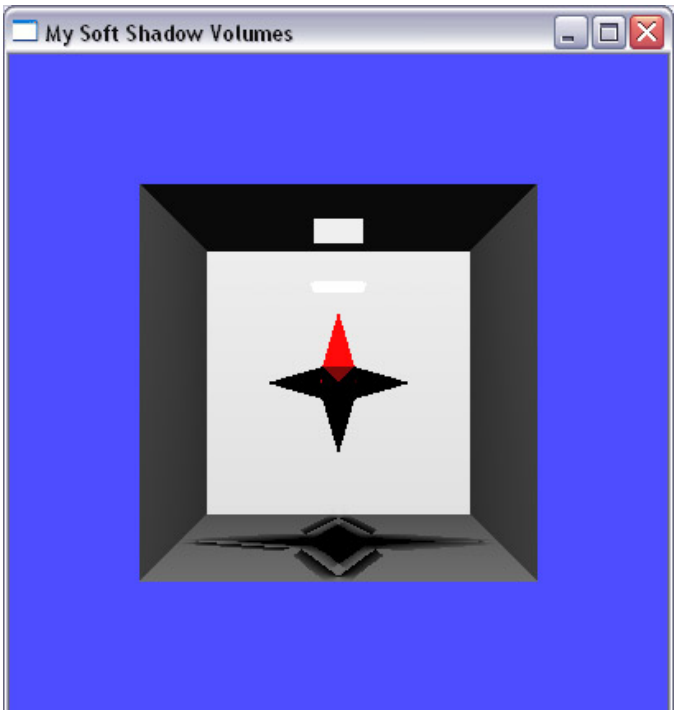


Abbildung 13: links: das Ergebnis mit dem Hard-Shadow-Volumes Algorithmus rechts: das 1024-Sampling-Points Bild

Bei meiner Implementierung kann man bei einem einfachen Objekt, wie einem Tetraeder, Artefakte in den Ecken des Schattens erkennen, wo sich die Wedges überschneiden. Bei dem komplexeren Stern sieht man auf dem Ergebnisbild zwar klar den Verlauf des weichen Schattens, leider entstehen um den weichen Schatten herum Artefakte.


My Soft Shadow Volumes


My Soft Shadow Volumes

Literatur

[1] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. *Rendering Techniques (13th Eurographics Workshop on Rendering).*

[2] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics (SIGGRAPH 2003).*

[3] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22(4):753–774, Dec. 2003. State-of-the-Art Reviews.

[4] Michael Herf. Efficient generation of soft shadow textures. 1997.

[5] Alan Heirich Maneesh Argavala, Ravi Ramamoorthi and Lauren Moll. Efficient image-based methods for rendering soft shadows. *Computer Graphics(SIGGRAPH 2000), Annual Conferences Series, ACM SIGGRAPH.*

[6] Roger McLassus. Wikipedia, Dec. 2005.

[7] Prof. Dr. Stefan Müller. Folien computergrafik 1, (5) clipping, 2005.