Shadow Mapping

Studienarbeit

vorgelegt von
Nico Hempe

Institut für Computervisualistik
Arbeitsgruppe Computergrafik

Betreuer: Dipl.-Inform. Thorsten Grosch
Prüfer: Prof. Dr.-Ing. Stefan Müller

Oktober 2004

Inhaltsverzeichnis

1.0 Einleitung

1.1 Motivation

Schon lange habe ich mich für die 3D-Computergrafik, insbesondere auch für Computerspiele begeistert. Die Vorstellung, immer realitätsnähere Szenen auf dem Computer in Echtzeit darstellen zu können, verblüfft mich jedes mal aufs Neue. Insbesondere die Grafik neuer Computerspiele und die immer authentischere Darstellungsqualität versetzen mich in Staunen. Dies ist auch einer der Gründe, der mich zu einer Studienarbeit im Bereich der Computergrafik veranlasste. Da Schatten eine der Grundvoraussetzungen sind, eine Szene realistisch wirken zu lassen, habe ich mich für ein Thema im Bereich Schattendarstellung entschieden. Schatten spielen gerade bei den aktuellen 3D-Spieletiteln wie z.B. Doom3 eine zentrale Rolle, um eine noch nie da gewesene Atmosphäre bei Computerspielen zu erzeugen. Viele aktuelle Spiele setzen zur Erzeugung von Schatten das Shadow Mapping Verfahren ein. Aber auch in Bereichen die nicht in Echtzeit berechnet werden, ist das Shadow Mapping als Schattenverfahren weit verbreitet. So setzen viele Hersteller von Animationsfilmen auf dieses Verfahren. Da ich bisher größtenteils nur theoretische Erfahrungen mit der Computergrafik gemacht habe, ist diese Studienarbeit für mich die Möglichkeit, theoretisch angeeignetes Wissen auch praktisch umzusetzen.

1.2 Schwerpunkt

Als Schwerpunkt dieser Arbeit habe ich mich für das Verfahren des „Shadow Mappings“ entschieden. Dies ist ein Verfahren, welches unabhängig von der Komplexität der Szene arbeitet und dadurch auch für die Schattenberechnung großer und komplizierter Szenen in Echtzeit bestens geeignet ist. Leider hat dieses Verfahren auch gravierende Nachteile, wie das Aliasing, auf die in dieser Arbeit näher eingegangen wird. Unter Aliasing versteht man einen Treppeneffekt an den Schattenkanten. Zu diesem Problem sollen auch neuartige Lösungen präsentiert und genauer beschrieben werden.

Ziel ist dabei die Entwicklung eines Programms, welches eine relativ große und komplexe Szene in Echtzeit schattiert. Dabei soll dies zum einen mit dem Standard Shadow Mapping Algorithmus, zum anderen durch die vorgestellten verbesserten Algorithmen geschehen, um einen Vergleich der verschiedenen Verfahren zu ermöglichen.

1.3 Übersicht

In Kapitel 2 werden zunächst verschiedene Schattenverfahren im Überblick vorgestellt und miteinander verglichen. In Kapitel 3 wird dann auf das Shadow Mapping genauer eingegangen und dessen Funktionsweise, sowie Vor- und Nachteile dieses Verfahrens erläutert. Kapitel 4 geht auf verschiedene Verbesserungen des Shadow Mappings ein. Dabei wird das Trapezoidal Shadow Mapping zur Reduzierung der Aliasing-Effekte bei gleicher Auflösung der Shadow Map vorgestellt und erläutert. Des Weiteren wird eine Modifikation vorgestellt, welche die Darstellung von weichen Schattenkanten ermöglicht. Kapitel 5 geht näher auf die Implementierung der hier vorgestellten Verfahren unter Verwendung von C++ und OpenGL ein. Außerdem findet sich darin eine Kommentierung des vorgestellten Quellcodes.

In Kapitel 6 werden die Ergebnisse der Verfahren aufgezeigt und miteinander verglichen bevor abschließend in Kapitel 7 ein Fazit aus den Erkenntnissen dieser Studienarbeit gezogen werden kann.

2.0 Schatten

2.1 Wozu Schatten?

Schon lange ist es das Ziel der 3D-Computergrafik, Szenen möglichst realitätsnah am Computer darstellen zu können. Hierbei spielen Schatten eine entscheidende Rolle. In der realen Welt wirft jedes Objekt Schatten und Szenen ohne Schatten wirken nicht reell, sondern flach und unschön. Aufgrund der 2D-Darstellung der Szene auf dem Monitor geht zudem die Dimension der Tiefe nahezu komplett verloren. Daher ist es umso wichtiger, möglichst viele sekundäre Tiefeninformationen in der grafischen Ausgabe zu integrieren.

Das menschliche Auge orientiert sich zur korrekten Wahrnehmung und Interpretation einer Szene am Schatten der Objekte. Bei einer Szene ohne Schatten kann das Auge nicht ermessen, wo genau sich ein bestimmtes Objekt oder die Lichtquelle im Raum befindet. Der Betrachter kann nicht erkennen, ob sich ein Gegenstand z.B. direkt über dem Boden befindet, oder ob er in einiger Entfernung über dem Boden schwebt. Auch die Größe des Objektes im Vergleich zu anderen kann nicht ermittelt werden. Erst durch die Darstellung der Szene mit Schattenwurf wird dies deutlich. Jetzt ist erkennbar, wo genau sich ein Objekt im Raum befindet und in welchem Größenverhältnis es zu anderen Objekten steht. Dieser Effekt wird in Bild 1 verdeutlicht. Anwendungen, die heute grafisch nahe an der Realität liegen sollen, sind ohne die Darstellung von Schatten nicht vorstellbar.

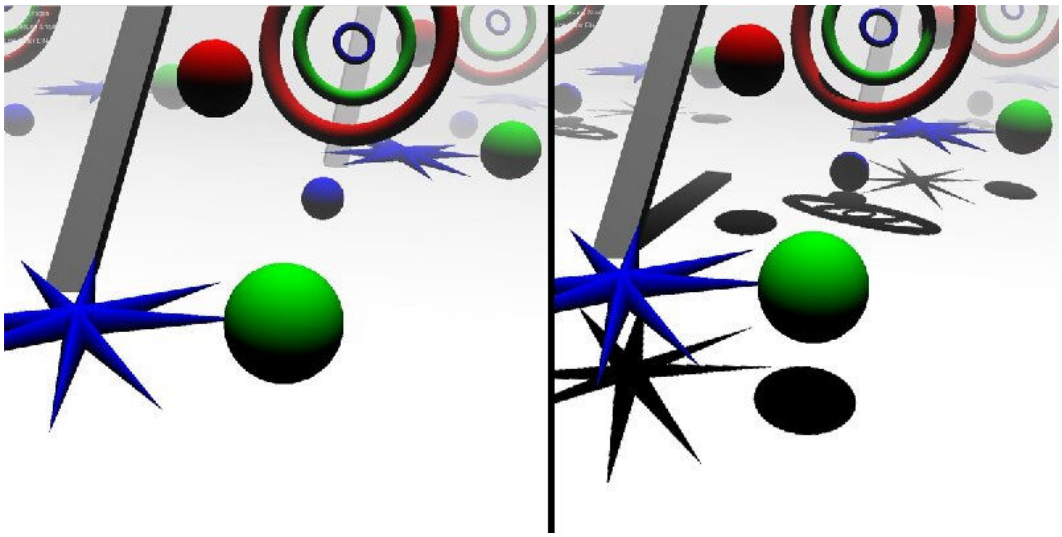


Bild 01 - Links: Szene ohne Schatten, Rechts: Szene mit Schatten

2.2 Schattenverfahren im Überblick

Zur Berechnung von Schatten gibt es in OpenGL leider keinen Aufruf wie z.B. glEnable(GL_SHADOWS). Das heißt, die Berechnung von Schatten muss komplett vom Programmierer übernommen werden. Hierzu sind im Laufe der Zeit viele verschiedene Verfahren entwickelt worden. Eines der bedeutendsten und das heute am weitesten verbreitete Verfahren ist das Shadow Mapping. Hierauf will diese Arbeit näher eingehen. Weitere Verfahren zur Berechnung von Echtzeitschatten sind z.B. Light Maps, Planare Schatten oder Shadow Volumes.

2.2.1 Light Maps

Light Maps eignen sich nur für statische Szenen, da hierbei die Schatten in den Texturen der Szene gespeichert und später einfach geladen werden. Aus diesem Grund ist dieses Verfahren für das Rendern dynamischer Szenen nicht geeignet. Die Qualität der Schatten hängt dabei von der Methode der ab, wie diese vor dem erstellen der Texturen berechnet wurden und wird theoretisch nur von der Auflösung der verwendeten Texturen eingeschränkt.

Bild 02 – Szene mit Light Maps

2.2.2 Planare Schatten

Die Idee der Planaren Schatten ist, dass diese Schatten als 2D-Projektion auf einer Ebene dargestellt werden. Dazu wird zunächst die Ebene gezeichnet, auf die der Schatten geworfen werden soll. Dann wird das Objekt in die Szene gesetzt, welches den Schatten werfen soll. Als letztes wird die Matrix auf eine 2D-Projektion gestellt, und das Objekt in der Schattenfarbe auf die Ebene projiziert.

Dieses Verfahren hat den Vorteil, dass es schnell und einfach funktioniert, und somit auch gut für das Rendern dynamischer Szenen in Echtzeit geeignet ist. Allerdings ist dieses Verfahren auch mit starken Einschränkungen behaftet. So funktioniert diese Methode nur mit planaren Ebenen und auch eine Selbstverschattung der Objekte ist nicht möglich.

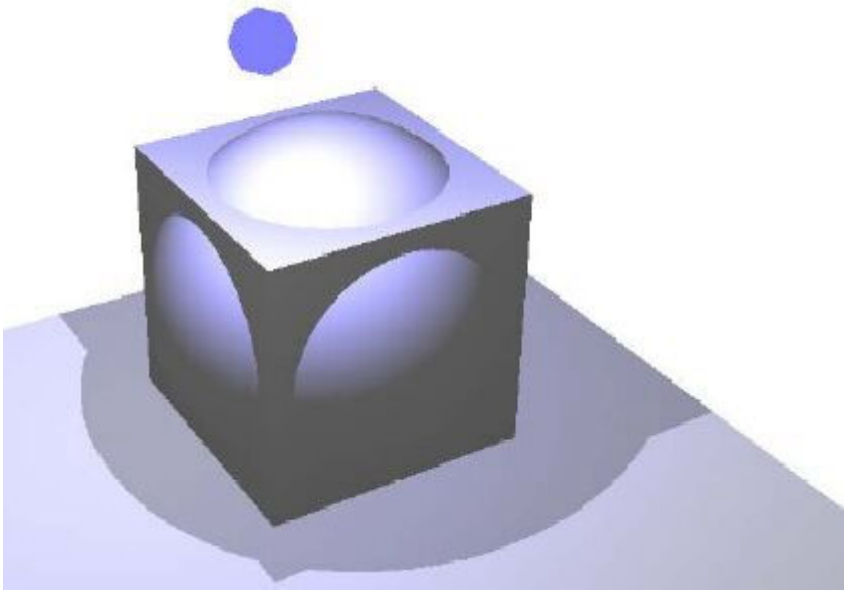


Bild 03 – Szene mit Planaren Schatten

2.2.3 Shadow Volumes

Shadow Volumes wurden erstmals 1977 von Frank Crow beschrieben. Dieses Verfahren nutzt den Stencil Buffer zur Schattendarstellung. Dabei müssen zunächst die Silhouetten-Eckpunkte der Objekte aus Sicht der Lichtquelle gefunden werden. Diese bilden die Shadow Volumes. Durch die Nutzung des Stencil Buffers können jetzt schattierte Bereiche der Szenen ausmaskiert werden. Mit diesem Verfahren ist auch eine Selbstverschattung der Objekte möglich.

Die Shadow Volumes haben zudem den Vorteil, dass mit ihnen sehr präzise Schatten gezeichnet werden können, die nicht wie die beim Shadow Mapping Verfahren Aliasing Effekte hervorrufen.

Nachteilig wirkt sich jedoch aus, dass die Silhouettenerkennung besonders bei komplexen Objekten und Szenen sehr viel Rechenleistung verbraucht und dadurch nur bedingt für die Echtzeitschattierung komplexer Szenen eingesetzt werden kann.

Bild 04 – Szene mit Shadow Volumes

2.2.4 Shadow Mapping

Lance Williams erfand 1978 das Shadow Mapping. Es setzte sich neben den Shadow Volumes schnell als der gebräuchlichster Schattenalgorithmus durch. Das Verfahren unterstützt ebenfalls die Selbstverschattung von Objekten und hat gegenüber den Shadow Volumes den Vorteil, dass es vollkommen unabhängig von der Geometrie der Szene arbeitet, was bedeutet, dass keinerlei Kenntnisse über die Geometrie der Szene oder Objekte benötigt werden.

Ein Beispiel für den Erfolg des Shadow Mappings stellt die Implementierung dieses Verfahrens in die Software *Renderman* der Firma Pixar dar. Später fand der Algorithmus auch in bekannten Kinofilmen, wie z.B. *Toy Story* Anwendung.

Aber gerade in Bereichen, in denen Schatten in Echtzeit berechnet werden müssen ist das Shadow Mapping eine beliebte Methode. So setzen auch viele aktuelle Videospiele diese Methode ein.

Bild 05 – Ausschnitt aus Toy Story

Bild 06 – Szene aus einem aktuellem PC-Spiel

3.0 Shadow Mapping im Detail

3.1 Funktionsweise des Shadow Mapping

Das Shadow Mapping basiert auf der Idee, dass die Szene zunächst aus Sicht der Lichtquelle gerendert wird. Damit dies möglich ist, muss es sich bei der Lichtquelle um ein Punktlicht oder direktionales Licht handeln. Das Standard Shadow Mapping funktioniert daher nur mit diesen Lichtquellenarten.

Aber wie kann man mit dem Verfahren herausfinden, ob es sich bei einem Bildpunkt um einen beleuchteten oder schattierten Punkt handelt? Grundlegend muss dazu geprüft werden, ob zwischen diesem bestimmten Punkt der Szene und der Lichtquelle ein anders Objekt liegt. Kann dieser Punkt aus Sicht der Lichtquelle gesehen werden, so ist der Punkt beleuchtet. Ist dieser Punkt nicht aus dem Blick der Lichtquelle zu sehen, so liegt der dieser im Schatten, denn das Licht der Lichtquelle kann diesen Punkt nicht erreichen.

Der Z-Buffer, oder auch Depth Buffer genannt, spielt bei diesem Test eine wesentliche Rolle, denn er enthält die Tiefeninformationen der Szene. Im ersten Durchlauf wird die Szene aus Sicht der Lichtquelle gerendert und nur die Tiefeninformation der Szene ausgelesen, die dann in einer 2D-Textur gespeichert wird. Diese Textur wird Shadow Map genannt. Sie besteht aus Grauwerten, welche die Tiefeninformation der jeweiligen Pixel zur Lichtquelle enthalten und dient später dem Test, ob ein Punkt im Schatten liegt oder nicht. Je dunkler ein Pixel in der Shadow Map ist, desto näher befindet sich dieser Pixel an der Lichtquelle.

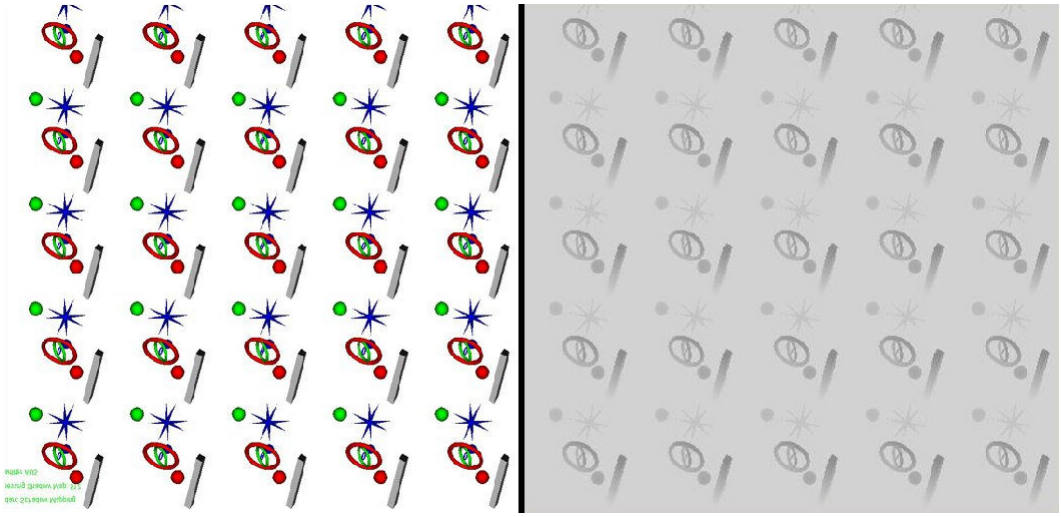


Bild 07 – Links: Szene aus Sicht der Lichtquelle, Rechts: Zugehörige Shadow Map

Im zweiten Durchlauf rendern wir die Szene aus Sicht der Kamera. Die im ersten Durchlauf erzeugte Shadow Map wird jetzt aus Sicht der Lichtquelle auf die Szene projiziert. Dazu transformieren wir jeden Pixel aus Kamerasicht in das Koordinatensystem der Shadow Map und vergleichen die Z-Werte der Szene mit den entsprechenden Tiefenwerten der Shadow Map. Dieser Test entscheidet darüber, ob der Pixel beleuchtet wird oder im Schatten liegt. Der Schattentest für einen bestimmten Punkt funktioniert dabei so, dass der Abstand eines Punktes aus Kamerasicht mit dem Wert der Shadow Map an der entsprechend projizierten Stelle verglichen wird. Bezeichnet man den Abstand des Punktes aus Kamerasicht zur Lichtquelle als R und den entsprechenden Wert der Shadow Map als D, so sind beim Schattentest die folgenden Ergebnisse möglich:

R = D: In diesem Fall ist der Abstand des Punktes gleich dem entsprechenden Wert in der Shadow Map. Das bedeutet, dass kein Objekt die Sichtlinie zur Lichtquelle blockiert. Der Punkt befindet sich nicht im Schatten und kann beleuchtet werden.

R > D: Der Abstand des Punktes zur Lichtquelle ist größer, als der entsprechende Wert in der Shadow Map. Hier muss demnach ein Objekt zwischen diesem Punkt und der Lichtquelle liegen, was bedeutet, dass dieser Punkt aus Sicht der Lichtquelle nicht sichtbar ist. Dieser Punkt liegt im Schatten.

Wie die Shadow Map aus Kamerasicht auf die Geometrie der Szene gelegt wird, verdeutlicht das folgende Diagramm:

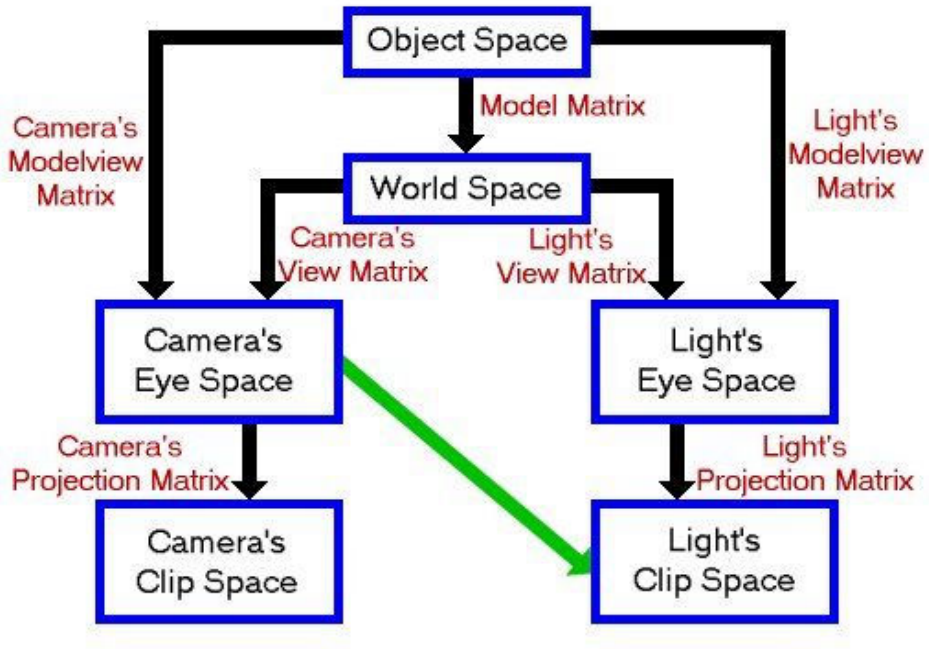


Bild 08: Aufgabe der benötigten Projektionsmatrix

Die Shadow Map ist ein Bild aus der Sicht des Lichtes, welches eine 2D Projektion ihres Clip Spaces ist. Um eine Texturprojektion vorzunehmen, wird in OpenGL die EYE_LINEAR Texturkoordinatenerzeugung verwendet. Sie erzeugt Texturkoordinaten für einen Vertex, der auf ihrer Kamera Koordinatenposition basiert. Diese erzeugten Texturkoordinaten müssen in die entsprechenden Koordinaten der Shadow Map transformiert werden. Hierfür findet die Texturmatrix Anwendung. Sie führt die Transformation aus, die in Bild 08 den diagonalen Pfeil darstellt.

Die Texturmatrix kann nach der folgenden Formel berechnet werden, wobei T die Texturmatrix, Pl und Vl die Projektions- bzw. Modelview-Matrix des Lichtes und Vc die Modelview Matrix der Kamera sind.

$$T = P_l \times V_l \times V_c^{-1}$$

Im nächsten Schritt müssen die Texturkoordinaten noch angepasst werden, da diese nach der perspektivischen Division entlang der X-, Y- und Z-Achse noch zwischen den Werten –1 und 1 liegen. Sie werden auf Werte zwischen 0 und 1 skaliert. Zu diesem Zweck verwendet man die folgende Matrix, welche mit der Texturmatrix T multipliziert wird.

$$\text{biasMatrix} = \begin{vmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0.5 & 0.5 & 0.5 & 1 \end{vmatrix}$$

Bedingt durch die Tatsache, dass nach dem Prinzip des Shadow Mappings für die Darstellung eines schattierten Bildes zwei Renderingdurchläufe benötigt werden - einmal aus Sicht der Lichtquelle und einmal aus Kamerasicht - handelt es sich beim Shadow Mapping um ein Two-Pass-Verfahren. Bei statischen Szenen reicht jedoch auch die einmalige Berechnung der Shadow Map aus.

3.2 Vor- und Nachteile des Shadow Mappings

Ein großer Vorteil des Shadow Mappings liegt in der Tatsache, dass es sehr schnell arbeitet und relativ einfach zu implementieren ist. Dadurch eignet sich dieses Verfahren seht gut für das Echtzeitrendering. Des Weiteren arbeitet dieses Verfahren im Gegensatz zu Shadow Volumes vollkommen unabhängig von der Geometrie der Szene. Es benötigt für die Schattendarstellung keine Kenntnisse über deren Geometrie. Dadurch kann im Gegensatz zu den Shadow Volumes eine einfache Szene genauso schnell schattiert werden, wie eine sehr komplexe mit vielen oder aufwändigen Objekten.

Allerdings hat das Shadow Mapping auch einige Nachteile. So kann es nur mit direktionalem Licht oder Spotlights realisiert werden. Spotlights werden definiert durch eine Position, eine Blickrichtung und einen Öffnungswinkel, direktionales Licht entsteht durch eine Lichtquelle, die theoretisch unendlich weit entfernt liegt und daher in der gesamten Szene im gleichen Winkel eintrifft, ist also vergleichbar mit Sonnenlicht. Diese Einschränkung ergibt sich daraus, dass die Szene aus Sicht der Lichtquelle gerendert werden muss. Mit omnidirektionalen Lichtquellen, also Lichtquelle, die in alle Richtungen Licht abgeben, ist dies nicht ohne weiteres möglich. Es existieren aber Ansätze, um diese Art Lichtquellen zu simulieren, indem sie diese in mehrere Punktlichtquellen aufteilen. Diese Möglichkeit soll hier aber nicht weiter betrachtet werden.

Eine weitere Herauforderung stellt das so genannte Polygon Offset Problem da. Dabei kommt es zu Darstellungsfehlern an Teilen der Objektoberflächen, die fälschlicherweise schattiert dargestellt werden. Dieser Fehler entsteht, da es sich beim Shadow Mapping um ein bildbasierendes Verfahren handelt und dadurch der für den Schattentest wichtige Z-Buffer nur mit einer endlichen Präzision berechnet wird.

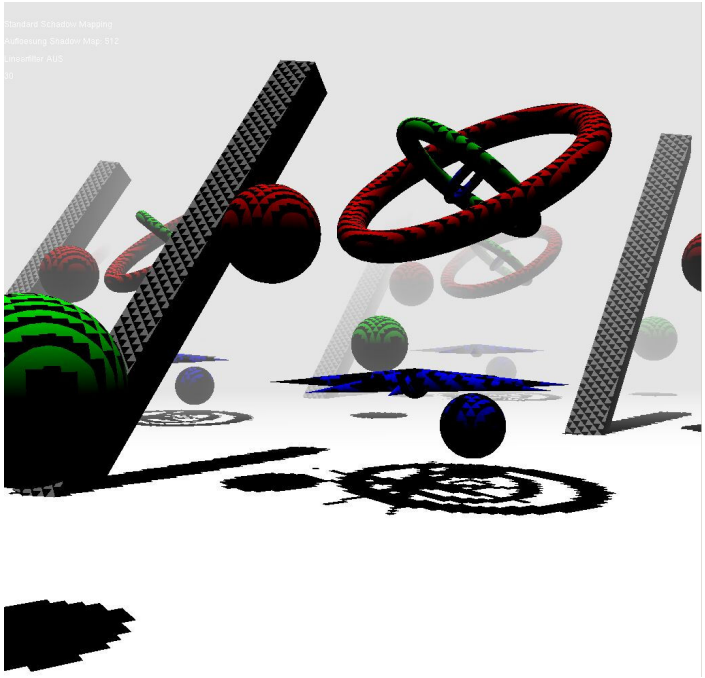


Bild 09 – Polygon Offset Problem

Das Polygon Offset Problem lässt sich jedoch relativ einfach beheben. Es muss nur immer ein bestimmter Wert zu den Tiefenwerten der Shadow Map hinzuaddiert werden. Dadurch werden die Objekte etwas von der Lichtquelle wegbewegt, was einem falschen Schattentest vorbeugt.

Ein anderer Lösungsansatz dieses Problems wäre, bei der Erstellung der Shadow Map die Front Faces der Objekte zu entfernen und nur die Back Faces zu rendern. Auch dadurch lassen sich in der Shadow Map die Z-Werte vergrößern. Zwar kann es nach dieser Methode zu Präzisionsproblemen bei der Schattenberechnung auf dem Back Face kommen, aber dies ist nicht weiter störend, da die Rückseite der Objekte standardmäßig von OpenGL nicht beleuchtet werden.

Das größte Problem des Shadow Mappings stellt allerdings das sogenannte Aliasing dar. Unter Aliasing versteht man einen Treppeneffekt an den Schattenkanten, wie er in Bild 10 zu erkennen ist. Das Aliasing entsteht durch die endliche Auflösung der Shadow Map. Dadurch werden verschiedene Pixel der Szene beim Schattentest auf denselben Punkt der Shadow Map transformiert und so mit dem gleichen Wert verglichen. Dadurch ergeben sich große quadratische Schattenareale in der Szene.

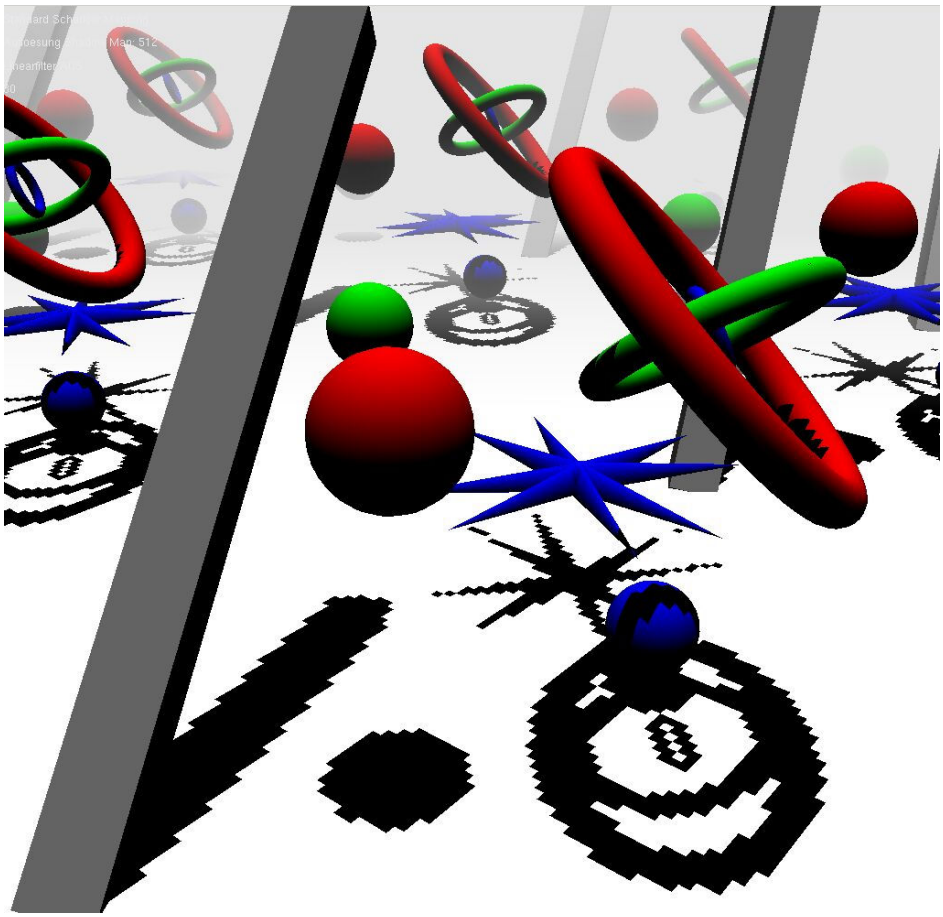


Bild 10 – Aliasing: Artefakte bilden sich an den Schattenkanten

4.0 Verbesserungen des Shadow Mappings

Die einfachste Art die Aliasing-Effekte des Shadow Mappings zu reduzieren, wäre, die Auflösung der Shadow Map zu vergrößern. Dies kann allerdings nur begrenzt geschehen, da auch moderne Grafikkarten oft nur Texturauflösungen von 2048x2048 Pixel unterstützen. Eine höhere Auflösung würde zudem zu einem deutlich spürbaren Geschwindigkeitsverlust führen. Aber gerade bei großen Szenen oder ungünstigem Verhältnis zwischen Lichtquelle und Kamera liefert auch eine hohe Auflösung nur unbefriedigende Ergebnisse.

Zur Reduzierung der Aliasing Effekte sind im Laufe der Zeit einige Verbesserungen des Shadow Mappings vorgestellt worden, welche Qualitätsverbesserungen auch ohne die Erhöhung der Shadow Map Auflösung erreichen. Dabei gibt es Ansätze die nicht in Echtzeit berechnet werden können, und solche die mit geringem Mehraufwand oder oftmals schneller als das Standard Shadow Mapping funktionieren. Diese Verfahren sind dann in Echtzeit einsetzbar. Als Beispiel für ein Nicht-Echtzeitverfahren mit guten Ergebnissen sei das Adaptive Shadow Mapping [Q09] genannt. Die Auflösung der Shadow Map wird dabei nach einer hierarchischen Baumstruktur in verschieden hoch aufgelöste Bereiche unterteilt. Bereiche an den Schattenkanten besitzen eine höhere Auflösung als Bereiche im Innern des Schattens.

In dieser Arbeit wird jedoch auf ein Echtzeitverfahren eingegangen, welches ähnlich gute Ergebnisse liefert. Hierbei handelt es sich um das Trapezoidal Shadow Mapping. Es reduziert die Aliasing Effekte auch bei geringen Auflösungen der Shadow Map auf ein Minimum und lässt sich mit geringem Mehraufwand im Vergleich zum Standard Shadow Mapping berechnen. Dieses Verfahren kann das Standard Shadow Mapping vollständig ersetzen.

4.1 Trapezoidal Shadow Maps

Die Trapezoidal Shadow Maps wurden erstmals 2004 von Tobias Martin und Tiow-Seng Tan der National University of Singapore beschrieben. Sie eignen sich besonders für die Darstellung von Schatten in sehr großen Szenen mit einer oder mehreren globalen Lichtquellen, bei denen das Standard Shadow Mapping auch mit hohen Auflösungen der Shadow Map nur unbefriedigende Ergebnisse liefert. Das Trapezoid Shadow Mapping baut dabei auf dem Prinzip des Perspektivischen Shadow Mapping auf, enthält aber einige Veränderungen, die zu deutlich besseren Ergebnissen führen. Das Perspektivische Shadow Mapping entwickelten Marc Stamminger und George Drettakis im Jahre 2003. Bei Interesse können Details zu diesem Verfahren in den Quellen [Q08] und [Q10] nachgelesen werden.

4.1.1 Idee

Das Trapezoidal Shadow Mapping baut auf der Grundidee auf, dass zur Schattenberechnung die Shadow Map nicht über die gesamte Szene erstellt werden muss, sondern nur über solche Bereiche, die tatsächlich auch von der Kamera gesehen werden können.
Schon allein durch diese Bedingung lässt sich die Qualität der Schatten deutlich erhöhen, da die Shadow Map nur für den Bereich erstellt wird, wo Schattenberechnungen auch tatsächlich benötigt werden. Das Trapezoidal Shadow Mapping geht jedoch noch weiter. Es berechnet ein Trapez, welches das Kamera Frustrum aus Sicht der Lichtquelle vollständig einschließt. Das Kamera Frustrum oder Kamera View Frustrum steht für den Sichtkegel der Kamera. Durch später beschriebene Transformationen wird dieses Trapez zu einem Einheitswürfel transformiert, was einen weiteren positiven Effekt mit sich bringt. Objekte der Szene, welche in der Nähe des Kameraursprungs, also nahe am Betrachter liegen, werden dadurch in einer höheren Auflösung schattiert als solche, die sich weiter entfernt befinden.

Das Trapezoidal Shadow Mapping behebt zudem ein bekanntes Problem der Perspektivischen Shadow Maps. Bei diesen kann es schon bei kleinen Bewegungen der Kamera oder des Lichtes von einem Frame zum nächsten zu deutlich erkennbaren Qualitätsunterschieden in der Schattendarstellung kommen. Dieser Effekt entsteht durch die quadratische Bounding Box, die bei den Perspektivischen Shadow Maps um das Kamera View Frustrum gelegt wird.

Das Trapezoidal Shadow Mapping basiert dagegen auf einem Trapez als Bounding Box. Es umschließt das View Frustrum der Kamera optimal, so dass dieser Qualitätsunterschied hier nicht zu beobachten ist. Im folgenden Bild erkennt man, wie sich bei einer Veränderung der Kamerasicht beim Perspektivischen Shadow Mapping die Fläche der nicht benötigten Teile der Shadow Map stark erhöht, wohingegen sie bei den Trapezoidal Shadow Maps nahezu konstant bleibt.

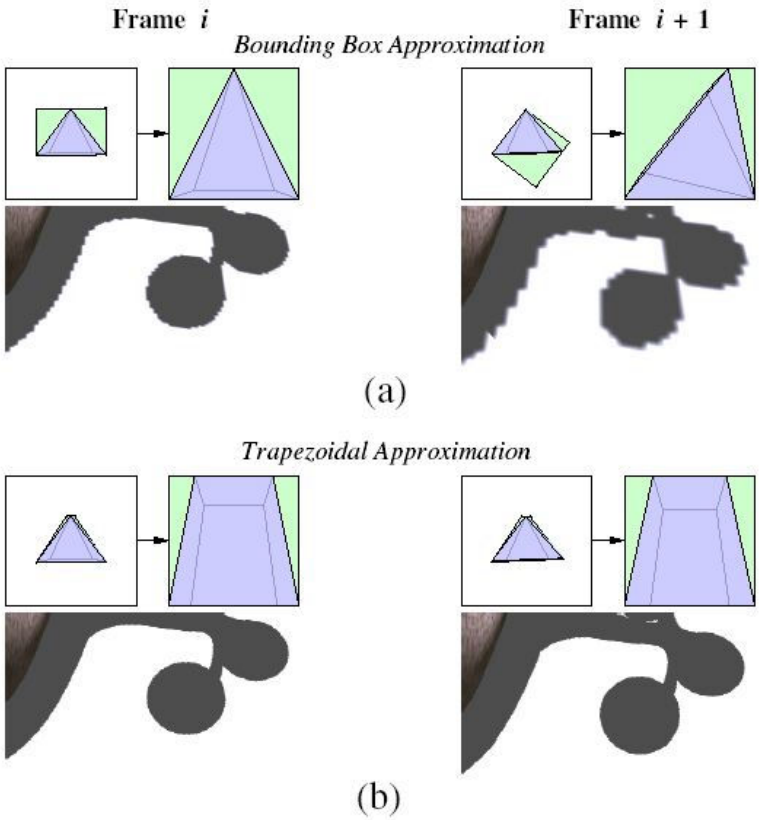


Bild 11 – Bounding Box um das Kamerafrustrum und entsprechende Schatten. Oben: Perspektivische Shadow Maps, Unten: Trapezoidal Shadow Maps

4.1.2 Funktionsweise

Die Initialisierung und Erzeugung der Shadow Map funktioniert beim Trapezoidal Shadow Mapping genauso wie beim Standard Shadow Mapping. Der Unterschied zum Standard Shadow Mapping liegt in der Fläche, über welches die Shadow Map erzeugt wird. Beim Trapezoidal Shadow Mapping wird die Shadow Map zu einem großen Teil nur über dem Gebiet erzeugt, welches auch tatsächlich von der Kamera überblickt wird. Dazu müssen als erstes die vier Trapezpunkte um das Kamera Frustrum aus Sicht der Lichtquelle bestimmt werden. Dies geschieht mit folgenden Schritten:

1. Zuerst berechnet man die Eckpunkte des Kamera View Frustrums und die Position der Kamera in Lichtkoordinaten. Dadurch ergeben sich je vier Punkte für die Near und Far Plane und ein Punkt, der den Kameraursprung darstellt.

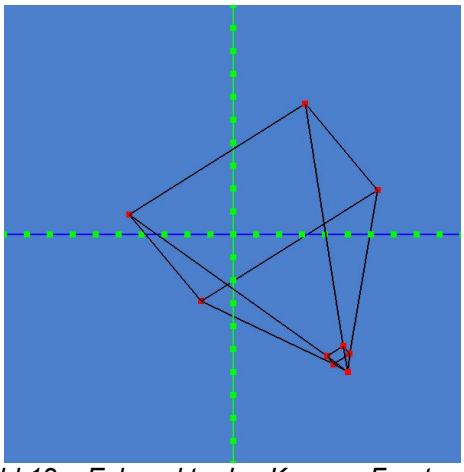


Bild 12 – Eckpunkte des Kamera Frustrums

2. Danach können die Mittelpunkte der Near und Far Plane berechnet und eine Gerade durch diese zwei Punkt gelegt werden. Diese Gerade wird als Mittellinie bezeichnet. Zu dieser Mittellinie bildet man die Orthogonale, welche ab jetzt Mittellinienorthogonale genannt wird.

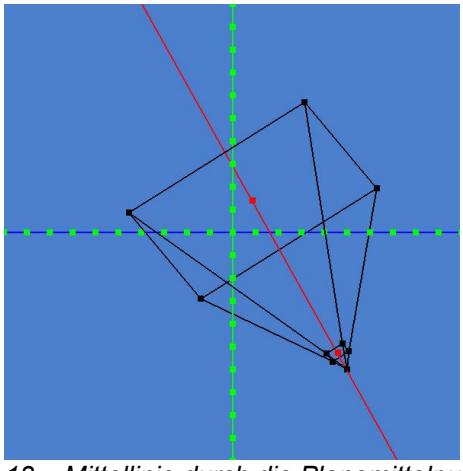


Bild 13 – Mittellinie durch die Planemittelpunkte

3. Die Mittellinienorthogonale wird jetzt an jedem der Eckpunkte angelegt und der entsprechende Schnittpunkt mit der Mittellinie berechnet. Aus diesen Schnittpunkten sucht man die zwei Punkte heraus, die am nächsten und am weitesten entfernt zum Kameraursprung liegen. Nur durch diese zwei Punkte zeichnet man jetzt eine Linie in Richtung der Mittellinienorthogonale. Die Linie mit dem geringsten Abstand zum Kameraursprung wird als Base Line bezeichnet, die Linie mit dem größten Abstand zum Kameraursprung heißt Top Line.

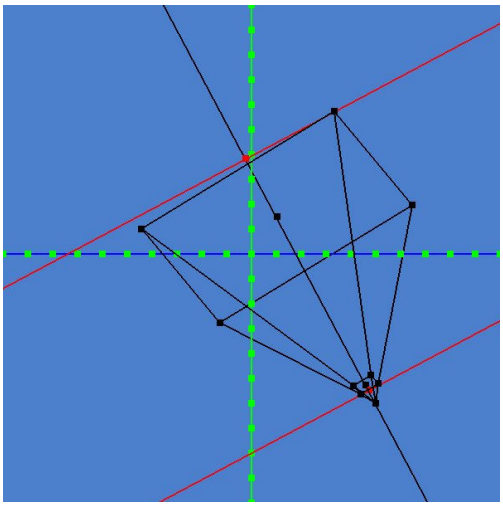


Bild 14 –Bestimmung der Base und Top Line

4. Als letztes müssen die Seitenlinien des Trapezes berechnet werden. Durch die Schnittpunkte der Base und Top Line mit den Seitenlinien kann man die vier gesuchten Trapezpunkte bestimmten, die das Kamera View Frustrum vollständig einschließen. Für die Konstruktion der Seitenlinien müssen die zwei Eckpunkte der Far Plane bekannt sein, die am weitesten links und rechts von der Mittellinie entfernt liegen. Durch sie verlaufen die Seitenlinien. Zur Konstruktion einer konvexen Hülle des Kamera View Frustrums würde es bereits ausreichen, die Seitenlinien einfach durch diese Punkte und den Kameraursprung zu ziehen.

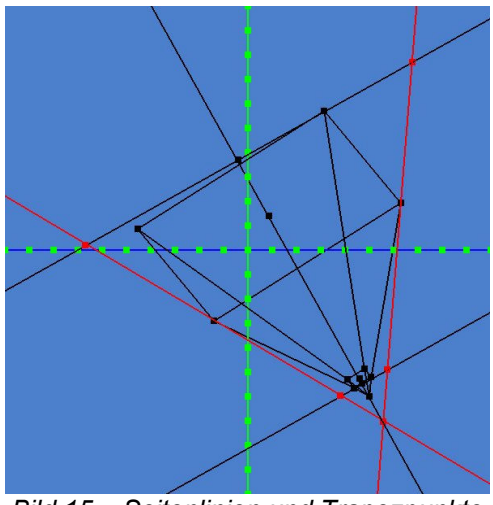


Bild 15 – Seitenlinien und Trapezpunkte

Die skizzierte Methode, die Seitenlinien zu bestimmen, liefert jedoch nur unbefriedigende Ergebnisse. In Bild 14 ist der optimierte Algorithmus bereits implementiert. Man erkennt es daran, dass die Seitenlinien nicht durch den Kameraursprung gezogen wurden, sondern durch einen Punkt, der etwas weiter dahinter liegt. Dieser Ansatz wird nachfolgend erläutert:

Die Verwendung des Kameraursprungs als Basispunkt der Seitenlinien liefert insoweit unbefriedigende Ergebnisse, da durch die Umrechnung des Trapezes zum Einheitswürfel der schmale Teil des Trapezes sehr stark gewichtet wird. Bereits nach kurzer Distanz nimmt die Gewichtung stark ab. Dies führt zu ausgeprägtem Oversampling in einem extrem schmalen Bereich an der Near Plane, während in dem Bereich dahinter deutliches Undersampling auftritt. Im Bild 15 erkennt man die Auswirkungen dieses Undersamplings. Während im vorderen Bereich die Schattenkanten noch relativ wenige Artefakte bilden, sind im hinteren Bereich der Szene nur noch einzelne Schattenblöcke zu erkennen. Den Teil, der in der Shadow Map am höchsten aufgelöst wird, sieht man hier nicht, da der Bereich so klein ist, dass er nur zwischen dem Kameraursprung und dem ersten Berühren des Kamera Frustrums mit dem Boden liegt.

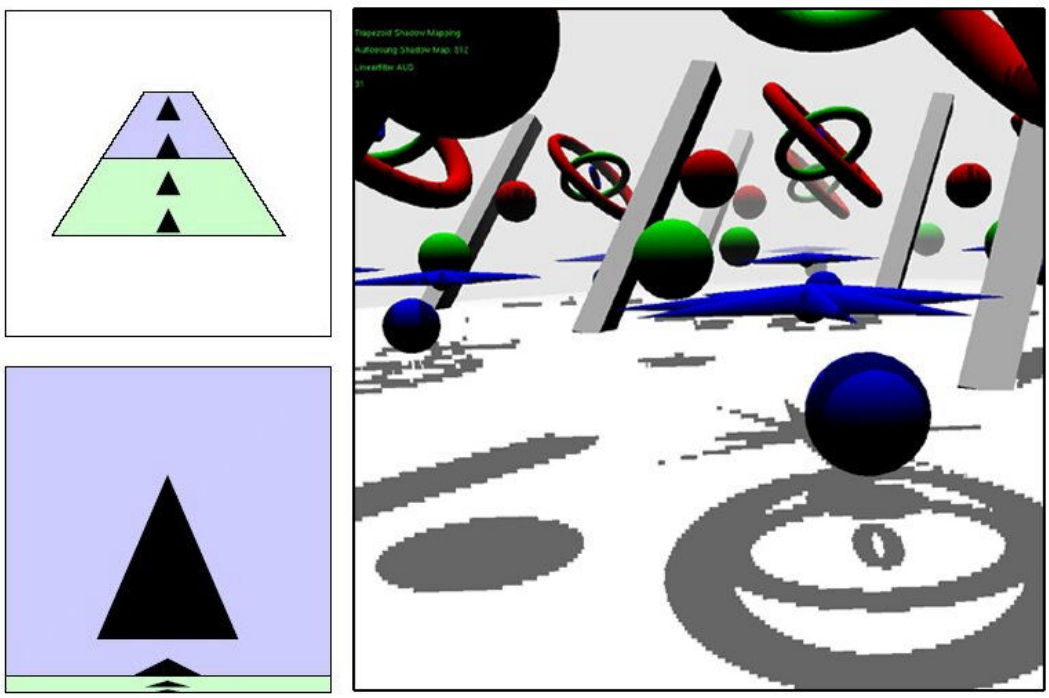


Bild 16 – Links Kamera Frustrum aus Sicht der Lichtquelle und Shadow Map Bereich Rechts: Szene, die mit diesem Verfahren Schattiert wird.

Aus diesem Grund führt man die Seitenlinien nicht durch den Kameraursprung, sondern durch einen Punkt in einem gewissen Abstand hinter der Kamera. Dadurch fällt die kurze Seite des Trapezes länger aus, und das Kamera Frustrum wird in diesem Bereich nicht so stark gestreckt. Zudem stellt man der Region, die für das Auge am interessantesten ist, insgesamt 80% der Shadow Map Auflösung zur Verfügung. Diese Region wird Focus Region genannt. Bild 17 verdeutlicht die Berechnung des gesuchten Punktes.

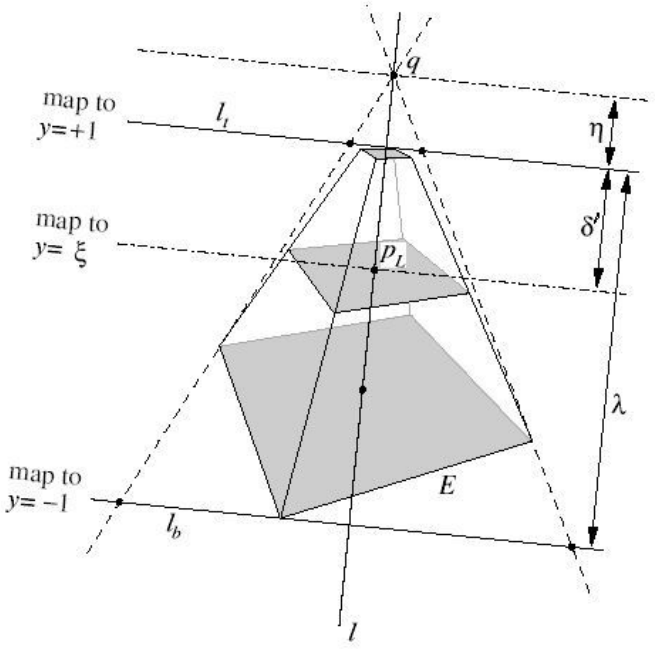


Bild 17 – Berechnung des 80%-Punktes

Die Focus Region erstreckt sich über den Abstand δ vom Kameraursprung in Sichtrichtung. Der Punkt p markiert dann einen Punkt, der genau im Anstand δ vom Kameraursprung entfernt liegt. pl bezeichnet diesen Punkt auf der Mittellinie. Der Abstand von der Top Line zu diesem Punkt wird als δ' bezeichnet. Der Punkt p_L wird dann durch die Transformation des Trapezes zum Einheitswürfel auf die 80% Linie projiziert. Deshalb wird diese Regel auch 80%-Regel genannt.

Dies geschieht durch die Beschreibung einer perspektivischen Projektion. Dadurch kann die Position eines Punktes q auf der Mittellinie berechnet werden, der den Punkt p_L auf die 80% Linie projizieren würde, wobei in diesem Fall y = -0.6 wäre, und die Base und Top Line auf y = -1 und y = +1 gesetzt werden würden. λ bezeichnet den Abstand zwischen der Base und Top Line. Der Abstand des Punktes q von der Top Line nennen η. Er berechnet sich durch folgende homogene 1D perspektivische Projektion:

$$\begin{pmatrix} \frac{-(\lambda+2\eta)}{\lambda} & \frac{2(\lambda+\eta)\eta}{\lambda} \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \delta'+\eta \\ 1 \end{pmatrix} = \begin{pmatrix} \tilde{\xi} \\ \omega \end{pmatrix},$$

$$\text{and} \quad \xi = \frac{\tilde{\xi}}{\omega}. \quad \text{So,} \quad \eta = \frac{\lambda\delta' + \lambda\delta'\xi}{\lambda - 2\delta' - \lambda\xi}.$$

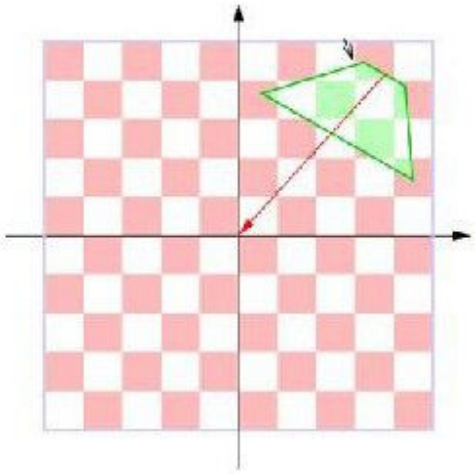
Die Seitenlinien ziehen wir jetzt vom Punkt q zu den äußeren Eckpunkten der Far Plane. Die vier gesuchten Trapezpunkte ergeben sich aus den Schnittpunkten der Seitenlinien mit der Top und Base Line. Diese Punkte werden als t_0 bis t_3 bezeichnet, wobei t_0 und t_1 die Trapezpunkte der längeren Seite sind, und t_2 und t_3 die der kürzeren Seite.

Der nächste Schritt ist die Berechnung einer Matrix, die dieses Trapez zu einem Einheitswürfel transformiert. Diese Matrix wird als N_T bezeichnet. Bei Interesse kann die im Beispielprogramm implementierte Funktionsweise in der Quelle [Q07] auf den Seiten 17-21 nachgelesen werden. Eine deutlich anschaulichere Berechnung dieser Matrix stellt jedoch die folgende Vorgehensweise dar. Sie vollzieht sich in 8 Schritten:

Schritt 1:
Die Matrix T_1 verschiebt die Mitte der Top Line in den Ursprung. Dabei setzen sich die Vektoren u und v aus den Werten x_u bis w_u und x_v bis z_v zusammen.

$$u = (t\_2 + t\_3)/2$$

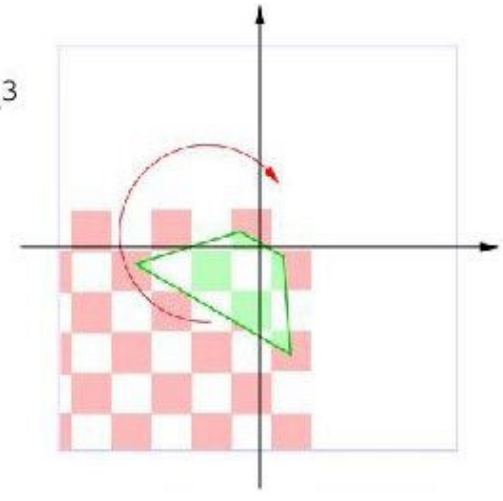
$$T\_1 = \begin{vmatrix} 1 & 0 & 0 & -x\_u \\ 0 & 1 & 0 & -y\_u \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 2:
Das Trapez mit der Matrix R wird so rotiert, dass die Top Line kollinear zur X-Achse liegt.

$$u = (t\_2 - t\_3) \;/\; |t\_2 - t\_3$$

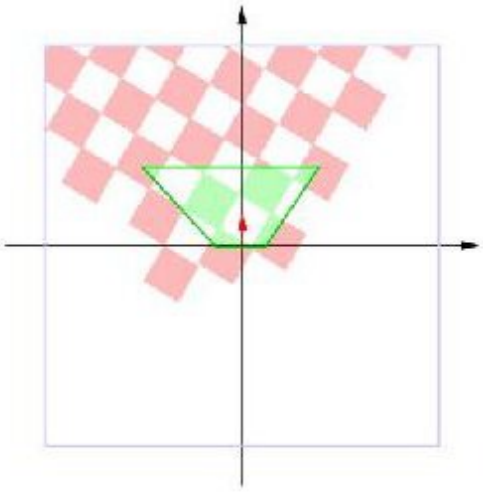
$$R = \begin{vmatrix} x\_u & y\_u & 0 & 0 \\ y\_u & -x\_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 3:
Danach wird das Trapez so verschoben, dass der Schnittpunkt i der beiden Seitenlinien von t_0 und t_3, sowie t_1 und t_2 im Ursprung liegt. Dies wird durch die Matrix T_2 erreicht.

$$u = R * T\_1 * i$$

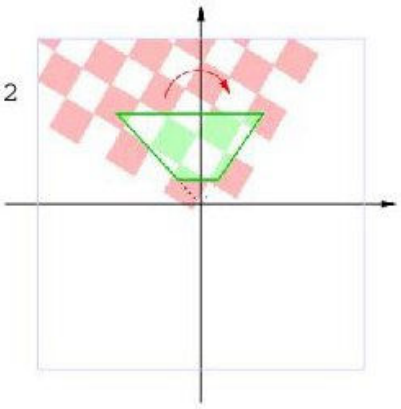
$$T\_2 = \begin{vmatrix} 1 & 0 & 0 & -x\_u \\ 0 & 1 & 0 & -y\_u \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 4:
Nun muss das Trapez mit der Matrix H geschert werden, so dass es symmetrisch zur Y-Achse ist.

$$u = (T\_2 * R * T\_1 * (t\_2 + t\_3)) / 2$$

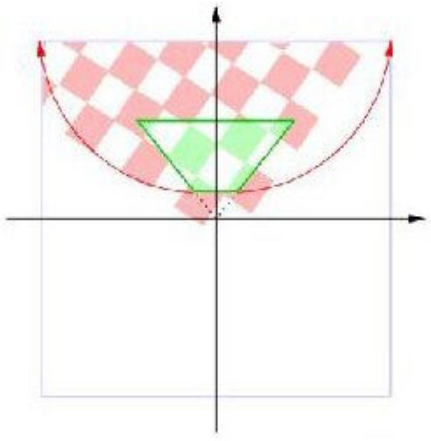
$$H = \begin{vmatrix} 1 & -x\_u/y\_u & 0 & 0 \\ 0 & 1/y\_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 5:
Durch die Matrix S_1 wird das Trapez so skaliert, dass der Abstand zwischen der Top Line und der X-Achse 1 beträgt. Des Weiteren beträgt der Winkel zwischen den beiden Seitenlinien nach der Skalierung 90°.

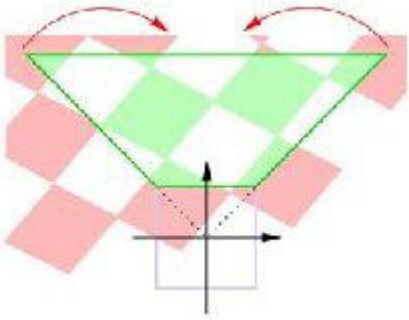
$$u = H * T\_2 * R * T\_1 * t\_2$$

$$S\_1 = \begin{vmatrix} 1/x\_u & 0 & 0 & 0 \\ 0 & 1/y\_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 6:
Die folgende Matrix N formt das Trapez zu einem Rechteck.

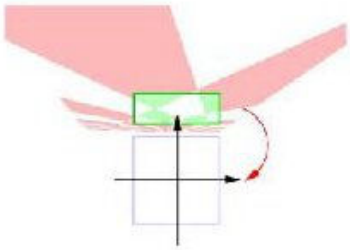
$$N = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix}$$



Schritt 7:
Anschließend wird das Rechteck entlang der Y-Achse verschoben, bis der Mittelpunkt des Rechtecks im Ursprung liegt. Danach ist das Rechteck auch symmetrisch zur X-Achse.

$$u = N * S\_1 * H * T\_2 * R * T\_1 * t\_0$$
$$v = N * S\_1 * H * T\_2 * R * T\_1 * t\_2$$

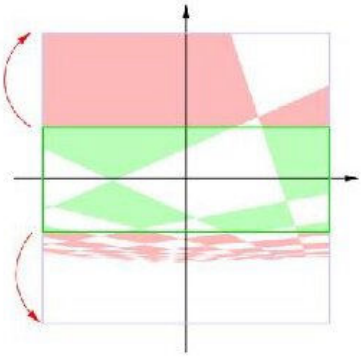
$$T\_3 = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -(y\_u/w\_u+y\_v/w\_v)/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 8:
Zuletzt skalieren wir das Rechteck mit der Matrix S_2 entlang der Y-Achse bis es den Einheitswürfel ausfüllt.

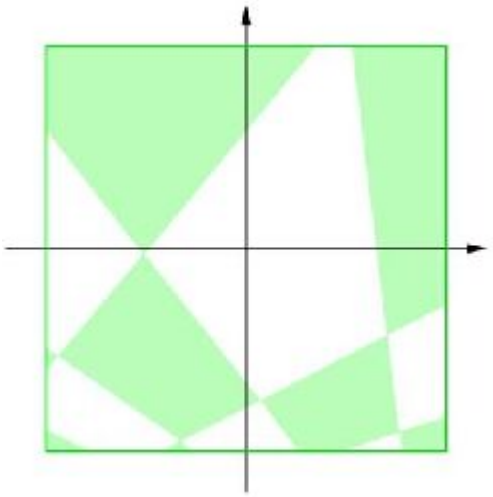
$$u = T\_3 * N * S\_1 * H * T\_2 * R * T\_1 * t\_0$$

$$S\_2 = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & -w\_u/y\_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Die Matrix N_T lässt sich jetzt anhand der eben errechneten Matrizen bestimmen.

$$N\_T = S\_2 * T\_3 * N * S\_1 * H * T\_2 * R * T\_1$$



Weil das Trapez zu einem Einheitswürfel transformiert wurde, kann darauf jetzt die Shadow Map gebildet werden. Der anschließende Schattentest funktioniert genauso, wie beim Standard Shadow Mapping. Es wird jedoch zusätzlich die Matrix N_T auf die Szene angewendet.

4.2 Weiche Schattenkanten

4.2.1 Idee

Weiche Schatten entstehen nur durch flächige Lichtquellen, nicht jedoch durch Punktlichtquellen. Da das Standard Shadow Mapping oder Trapezoidal Shadow Mapping jedoch nur Punktlichtquellen oder direktionale Lichtquellen unterstützt, erscheint die Erzeugung weicher Schatten mit diesem Verfahren unmöglich. Mit einem kleinen Trick kann man jedoch eine flächige Lichtquelle mit Punktlichtquellen simulieren.

Dazu wird jede Ecke der flächigen Lichtquelle als Punktlichtquelle angesehen. Je größer die Leuchtfläche ist, desto ungenauer wird diese Simulation. Deshalb sollten die Kanten der Leuchtfläche aus möglichst vielen Punktlichtquellen bestehen, die alle im gleichen Abstand zueinender liegen. Je mehr Punktlichtquellen hierzu benutzt werden, desto besser wird das Ergebnis ausfallen. Nun muss die Szene für jede Lichtquelle gerendert und schattiert und die entstandenen Bilder miteinander verrechnet werden. Viele Punktlichtquellen bedeuten allerdings auch viel Rechenaufwand, so dass bei dieser Methode ein Kompromiss gefunden werden muss, um Echtzeitwiedergabe zu ermöglichen.

4.2.2 Funktionsweise

Die Methode verwendet den Accumulation Buffer unter OpenGL. Der Buffer kann mehrere Bilder zu einem verrechnen. Dabei wird bei einer Anzahl X von Bildern, jedes durch X geteilt und zu den bisherigen addiert. Dadurch entsteht am Ende ein Bild, welches zu gleichen Teilen aus den verschiedenen Bildern zusammengestellt wurde. Im Effekt erscheinen Bereiche, die nur in wenigen Bildern schattiert waren heller als solche, die in sehr vielen Bildern schattiert wurden.

Während der Accumulation Buffer früher nur Softwareseitig unterstützt wurde und damit sehr langsam war, wird dieser heute von modernen Grafikkarten auch Hardwareseitig unterstützt, was zu hohen Frameraten führt und dadurch akzeptable Ergebnisse liefert.

In der vorliegenden Arbeit wird die flächige Lichtquelle durch 16 Punktlichtquellen ersetzt. Dabei ist zu beachten, dass die Szene für das endgültige Bild jedoch doppelt so oft gerendert werden muss, da Shadow Mapping ein Two-Pass-Verfahren darstellt. Zur Anzeige eines Bildes sind in diesem Beispiel also 32 Renderingdurchläufe notwendig. Nur neuste Grafikkarten können solch hohe Anforderungen in Echtzeit berechnen. Auf diese Weise kann jedoch jedes Verfahren das harte Schatten erzeugt dazu genutzt werden, auch weiche Schatten zu simulieren.

5.0 Implementierung

Das folgende Kapitel geht auf die Implementierung der vorgestellten Shadow Mapping Verfahren ein. Die Funktionsweise kann anhand des Quellcodes erklärt werden.

5.1 Standard Shadow Mapping

Das Standard Shadow Mapping ist in der Klasse STDSM implementiert. Die Benutzung und Funktionsweise der von der Klasse zur Verfügung gestellten Methoden werden in den folgenden Kapiteln beschrieben.

5.1.1 Die Klasse STDSM

```
class STDSM
{
public:
    void init(MATRIX4X4 *lightProjectionMatrix, MATRIX4X4 *lightViewMatrix, MATRIX4X4 *cameraProjectionMatrix,
              MATRIX4X4 *cameraViewMatrix, int *width, int *height);
    void createShadowMap(int shadowsize);
    void drawCamera();
    void drawLight();
    void end();

private:
    MATRIX4X4 *LPM;
    MATRIX4X4 *LVM;
    MATRIX4X4 *CPM;
    MATRIX4X4 *CVM;
    GLuint shadowtex;
    int *width;
    int *height;
};
```

Listing 1 – Die Klasse STDSM

Die privaten Variablen sind Zeiger, die im Hauptprogramm an die Klasse übergeben werden. *LPM, LVM, CPM* und *CVM* sind dabei die verschiedenen Modelview und Projektions Matrizen. Die Variable *shadowtex* enthält später die Shadow Map und *width* und *height* die Höhe und Breite des Fensters.

5.1.2 Die Methode *init*

```
void STDSM::init(MATRIX4X4 *lightProjectionMatrix, MATRIX4X4 *lightViewMatrix, MATRIX4X4 *cameraProjectionMatrix,
                 MATRIX4X4 *cameraViewMatrix, int *width, int *height)
{
   float shadowAmbient[] = {0.4f, 0.4f, 0.4f, 1.0f};

   glGenTextures( 1, &shadowtex );
   glBindTexture( GL_TEXTURE_2D, shadowtex );

   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );

   glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
   glTexParameteri( GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE_ARB, GL_LUMINANCE );

   glTexParameteri( GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE_ARB, GL_COMPARE_R_TO_TEXTURE_ARB );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL );
   glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FAIL_VALUE_ARB,shadowAmbient);

   LPM = lightProjectionMatrix;
   LVM = lightViewMatrix;
   CPM = cameraProjectionMatrix;
   CVM = cameraViewMatrix;
   this->width=width;
   this->height=height;
}
```

Listing 2 - Die Methode init der Klasse STDSM

In der Initialisierung des Hauptprogramms wird die Methode *init* der Klasse aufgerufen, und damit die benötigten Variablen per Referenz an die Klasse übergeben. Hierbei kann je nach belieben die Projektionsmatrix als perspektivische oder orthogonale Sicht übergeben werden, da Shadow Mapping sowohl mit direktionalem Licht, also einer orthografischer Sicht der Lichtquellen, als auch mit Punktlichtquellen, also einer perspektivische Sicht, funktioniert. Im Beispielprogramm wird bewusst eine orthografische Sicht verwendet, da auch das Trapezoidal Shadow Mapping diese Sicht benutzt und damit besser Qualitätsvergleiche der verschiedenen Schattenverfahren möglich sind.

Die Methode *init* speichert nicht nur die Adressen der übergebenen Variablen in den lokalen Variablen der Klasse, sondern initialisiert auch die Textur für die Shadow Map. Mit *GL_TEXTURE_COMPARE_FAIL_VALUE_ARB* wird die Schattenfarbe angegeben. Ohne diese Angabe wird der Schatten immer komplett schwarz dargestellt.

5.1.3 Die Methode createShadowMap

```
void STDSM::createShadowMap(int shadowsize)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(*LPM);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(*LVM);

    glViewport( 0, 0, shadowsize, shadowsize );
    glDisable( GL_LIGHTING );
    glColorMask(0,0,0,0);
    glEnable( GL_POLYGON_OFFSET_FILL );
    drawScene();

    glBindTexture( GL_TEXTURE_2D, this->shadowtex);
    glCopyTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 0, 0, shadowsize, shadowsize, 0 );
    glEnable( GL_LIGHTING );
    glColorMask( 1,1,1,1 );
    glDisable( GL_POLYGON_OFFSET_FILL );
    glViewport(0, 0, *width, *height );
}
```

Listing 3 – Die Methode createShadowMap der Klasse STDSM

Die Methode *createShadowMap* erzeugt die Shadow Map und führt dadurch den First Pass des Shadow Mappings aus. Als Parameter bekommt die Methode dabei die Größe der Shadow Map übergeben. Die Methode muss in jedem Frame aufgerufen werden, da die Shadow Map bei dynamischen Szenen für jedes Bild neu berechnet werden muss.

Zuerst wird der Color und Depth Buffer gelöscht und die Projektions und Modelview Matrix auf die der Lichtquelle gesetzt.

Anschließend setzt man den Viewport auf die Auflösung der Shadow Map, schaltet die Beleuchtung aus und setzt die Color Mask auf Null, um nur in den Depth Buffer zu rendern. Erst dann wird die Szene gezeichnet. Durch *GL_POLYGON_OFFSET_FILL* addiert OpenGL den angegebene Offset zu den Tiefenwerten hinzu, um die in Kapitel 3.2 beschriebenen Artefakte zu vermeiden.

Nach dem Rendern wird der Tiefenbuffer ausgelesen und in der Shadow Map Textur gespeichert. Zum Schluss aktiviert man die Beleuchtung und den Color Buffer wieder, deaktiviert das *GL_POLYGON_OFFSET_FILL* und setzt den Viewport wieder auf den ursprünglichen Wert.

5.1.4 Die Methode drawCamera

```
void STDSM::drawCamera()
{
    glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
    glMatrixMode( GL_PROJECTION );
    glLoadMatrixf(*CPM);
    glMatrixMode( GL_MODELVIEW );
    glLoadMatrixf(*CVM);

    static MATRIX4X4 biasMatrix(0.5f, 0.0f, 0.0f, 0.0f,
                                0.0f, 0.5f, 0.0f, 0.0f,
                                0.0f, 0.0f, 0.5f, 0.0f,
                                0.5f, 0.5f, 0.5f, 1.0f);

    MATRIX4X4 textureMatrix=biasMatrix*(*LPM)*(*LVM);

    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_S, GL_EYE_PLANE, textureMatrix.GetRow(0));
    glEnable(GL_TEXTURE_GEN_S);

    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_T, GL_EYE_PLANE, textureMatrix.GetRow(1));
    glEnable(GL_TEXTURE_GEN_T);

    glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_R, GL_EYE_PLANE, textureMatrix.GetRow(2));
    glEnable(GL_TEXTURE_GEN_R);

    glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
    glTexGenfv(GL_Q, GL_EYE_PLANE, textureMatrix.GetRow(3));
    glEnable(GL_TEXTURE_GEN_Q);

    glBindTexture(GL_TEXTURE_2D, shadowtex);
    glEnable(GL_TEXTURE_2D);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB, GL_COMPARE_R_TO_TEXTURE_ARB);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL);

    glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE_ARB, GL_INTENSITY);

    glEnable(GL_FOG);
    drawScene();
    glDisable(GL_FOG);
}
```

Listing 4 – Die Methode drawCamera der Klasse STDSM

Die Methode *drawCamera* zeichnet schließlich die Szene mit Schatten, führt also den „Second Pass“ des Shadow Mappings aus. Auch hier löscht man zunächst den Color und Depth Buffer. Anschließend wird die Projektions und Modelview Matrix der Camera geladen.

Als nächstes erzeugt man eine 4x4 Matrix, die die Textur von der Dimension [-1, 1] auf [0, 1] umwandelt. Die 4x4 Matrix *textureMatrix* enthält diese Matrix zur Umrechnung der Szene in die [0, 1] Texturkoordinaten.

Im nächsten Schritt wird die Generierung der Texturkoordinaten vorgenommen.

Dann bindet man die Shadow Map, aktiviert den Schattentest und zeichnet die Szene. Dabei wird jeder Pixel der den Schattentest nicht besteht schattiert dargestellt, da dieser sich im Schatten befindet. Zum zeichnen der Szene wird zusätzlich noch der Nebel aktiviert.

5.1.5 Die Methode drawLight

```
void STDSM::drawLight()
{
    glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(*LPM);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(*LVM);
    drawScene();
}
```

Listing 5 – Die Methode drawLight der Klasse STDSM

Die Methode *drawLight* zeichnet die Szene einfach aus Sicht der Lichtquelle, ohne jedoch den Schattentest durchzuführen und Schatten zu erzeugen. Dies würde hier auch nicht viel Sinn ergeben, da aus Sicht der Lichtquelle keine Schatten zu sehen sind. Diese Ansicht soll lediglich zeigen, über welchem Gebiet die Shadow Map aufgebaut wird.

Wie in *drawCamera* wird auch hier zunächst der Color und Depth Buffer gelöscht. Dann wird die Projektions und Modelview Matrix für die Lichtquelle geladen und die Szene gezeichnet.

5.1.6 Die Methode end

```
void STDSM::end()
{
    glDisable( GL_TEXTURE_2D );
    glMatrixMode( GL_TEXTURE );
    glLoadIdentity();
    glDisable( GL_TEXTURE_2D );
    glDisable( GL_TEXTURE_GEN_S );
    glDisable( GL_TEXTURE_GEN_T );
    glDisable( GL_TEXTURE_GEN_R );
    glDisable( GL_TEXTURE_GEN_Q );
}
```

Listing 6 – Die Methode end der Klasse STDSM

Die Methode *end* wird nach dem zeichnen der Szene mit Schatten aufgerufen. Sie setzt die Texturmatrix zurück und deaktiviert die Texturen wieder.

5.2 Trapezoidal Shadow Mapping

Das Trapezoidal Shadow Mapping ist in der Klasse TRAPSM implementiert. Die Benutzung und Funktionsweise der von der Klasse zur Verfügung gestellten Methoden werden in den folgenden Kapiteln beschrieben.

5.2.1 Die Klasse TRAPSM

```
class TRAPSM
{
public:
    void init(MATRIX4X4 *lightProjectionMatrixOrtho, MATRIX4X4 *lightViewMatrix, MATRIX4X4 *cameraProjectionMatrix,
              MATRIX4X4 *cameraViewMatrix, int *sichtweite, int *width, int *height);
    void doCalculations();
    void createShadowMap(int shadowsize);
    void drawCamera();
    void drawLight();
    void end();

private:
    MATRIX4X4 *LPMO;
    MATRIX4X4 *LVM;
    MATRIX4X4 *CPM;
    MATRIX4X4 *CVM;
    MATRIX4X4 N_T;
    int *width;
    int *height;
    int *sichtweite;
    GLuint shadowtex;
    double RQ[3][3];
    double TR[3][3];

    void calculateTrapezoid(VECTOR3D &t_0,VECTOR3D &t_1,VECTOR3D &t_2,VECTOR3D &t_3);
    void map_Square_To_Quad(double t0[3],double t1[3],double t2[3],double t3[3]);
    void map_Trapezoid_To_Square(double t0[3], double t1[3], double t2[3], double t3[3]);
    void Adjoint();
    float det2(float a,float b, float c, float d);
};
```

Listing 7 – Die Klasse TRAPSM

Die privaten Variablen sind Zeiger, die im Hauptprogramm an die Klasse übergeben werden. *LPM, LVM, CPM* und *CVM* sind dabei die verschiedenen Modelview und Projektions Matrizen. Die Variable *shadowtex* enthält später die Shadow Map und *width* und *height* die Höhe und Breite des Fensters.

Die 4x4 Matrix *N_T* enthält die Matrix, welche die trapezförmige Shadow Map in einen Einheitswürfel transformiert.

Die Variablen *RQ* und *TR*, sowie alle privaten Methoden werden für die Berechnung dieser Matrix benötigt. Sie werden hier nicht näher beschrieben. Bei Interesse kann die Funktionsweise [Q07] auf den Seiten 17-21 nachgelesen werden. Der Source Code ist dort im Appendix A.2.2 auf den Seiten 70-72 zu finden.

5.2.2 Die Methode *init*

```
void TRAPSM::init(MATRIX4X4 *lightProjectionMatrixOrtho, MATRIX4X4 *lightViewMatrix,
                  MATRIX4X4 *cameraProjectionMatrix, MATRIX4X4 *cameraViewMatrix,
                  int *sichtweite, int *width, int *height)
{
   float shadowAmbient[] = {0.4f, 0.4f, 0.4f, 1.0f};

   glGenTextures( 1, &shadowtex );
   glBindTexture( GL_TEXTURE_2D, shadowtex );

   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );

   glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
   glTexParameteri( GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE_ARB, GL_LUMINANCE );

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB, GL_COMPARE_R_TO_TEXTURE_ARB );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL );
   glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FAIL_VALUE_ARB,shadowAmbient);

   LPMO = lightProjectionMatrixOrtho;
   LVM = lightViewMatrix;
   CPM = cameraProjectionMatrix;
   CVM = cameraViewMatrix;
   this->sichtweite=sichtweite;
   this->width = width;
   this->height = height;
}
```

Listing 8 – Die Methode init der Klasse TRAPSM

Im Hauptprogramm werden der Methode *init* die Adressen der verschiedenen Modelview und Projektions Matrizen, sowie die Sichtweite der Kamera und die Höhe und breite des Fensters übergeben. Diese werden in den lokalen Variablen der Klasse gespeichert. Die Projektionsmatrix der Lichtquelle muss dabei orthografisch sein, da nach der Berechnung der Trapezpunkte diese durch die Maße der orthografischen Projektion geteilt werden müssen, um Koordinaten im Einheitswürfel zu erhalten. Die Sicht der Lichtquelle, und damit die berechneten Trapezpunkte in Lichtkoordinaten müssen also in den postperspektivischen Raum transformiert werden.

Des weiteren initialisiert diese Methode die Shadow Map Textur. Per *GL_TEXTURE_COMPARE_FAIL_VALUE_ARB* kann angegeben werden, welche Farbe der Schatten haben soll. Ohne diese Angabe wird der Schatten immer schwarz gezeichnet.

5.2.3 Die Methode doCalculations und calculateTrapezoid

```
void TRAPSM::doCalculations()
{
    VECTOR3D t_0, t_1, t_2, t_3;
    calculateTrapezoid(t_0,t_1,t_2,t_3);

    double t0[3];
    t0[0] = t_0.x;
    t0[1] = t_0.y;
    t0[2] = t_0.z;

    double t1[3];
    t1[0] = t_1.x;
    t1[1] = t_1.y;
    t1[2] = t_1.z;

    double t2[3];
    t2[0] = t_2.x;
    t2[1] = t_2.y;
    t2[2] = t_2.z;

    double t3[3];
    t3[0] = t_3.x;
    t3[1] = t_3.y;
    t3[2] = t_3.z;

    map_Trapezoid_To_Square(t0,t1,t2,t3);

    GLdouble N_T2[16];
    N_T2[0] = TR[0][0]; N_T2[4] = TR[1][0]; N_T2[ 8] = 0.0; N_T2[12] = TR[2][0];
    N_T2[1] = TR[0][1]; N_T2[5] = TR[1][1]; N_T2[ 9] = 0.0; N_T2[13] = TR[2][1];
    N_T2[2] =     0.0; N_T2[6] =                  0.0; N_T2[10] = 1.0; N_T2[14] =     0.0;
    N_T2[3] = TR[0][2]; N_T2[7] = TR[1][2]; N_T2[11] = 0.0; N_T2[15] = TR[2][2];

    N_T = MATRIX4X4(N_T2[0],N_T2[1],N_T2[2],N_T2[3],
                    N_T2[4],N_T2[5],N_T2[6],N_T2[7],
                    N_T2[8],N_T2[9],N_T2[10],N_T2[11],
                    N_T2[12],N_T2[13],N_T2[14],N_T2[15]);
}
```

Listing 9 – Die Methode doCalculations der Klasse TRAPSM

Die Methode *doCalculations* erzeugt zuerst die Variablen *t_0* bis *t_3*, die an die Methode *calculateTrapezoid* übergeben werden. Nach diesem Aufruf enthalten sie die Eckpunkte des Trapezes aus Sicht der Lichtquelle, umgerechnet auf Werte zwischen –1 und 1. Auf diese Methode gehen wir nach diesem Abschnitt ein.

Anschließend rechnet man die Vektoren in Arrays um, da die Methode *map_Trapezoid_to_Square* Arrays erwartet. In dieser Studienarbeit wird nicht näher auf die diese Methode eingegangen. Es genügt zu wissen, dass diese anhand der Trapezpunkte eine Matrix erzeugt, die das Trapez zu einem Einheitswürfel transformiert. Diese Transformationsmatrix speichert man in der Variablen *N_T*. Bei Interesse kann die Funktionsweise in der Quelle [Q07] auf den Seiten 17-21 nachgelesen werden. Der Source Code ist im Appendix A.2.2 auf den Seiten 70-72 zu finden.

Eine weitere Möglichkeit, die die Erzeugung der Transformationsmatrix *N_T* sehr anschaulich erläutert, kann in dieser Studienarbeit im Kapitel 4.1.2 nachgelesen werden.

```
void TRAPSM::calculateTrapezoid(VECTOR3D &t_0,VECTOR3D &t_1,VECTOR3D &t_2,VECTOR3D &t_3)
{
    float t = -1.0f * tan(PI/8);
    float b = -t;
    float l = b;
    float r = t;
    VECTOR3D c0_C(0,0,0);
    VECTOR3D v0_C(l,t,-1);
    VECTOR3D v1_C(l,b,-1);
    VECTOR3D v2_C(r,b,-1);
    VECTOR3D v3_C(r,t,-1);

    float t2 = *sichtweite*t;
    float b2 = -t2;
    float l2 = b2;
    float r2 = t2;
    VECTOR3D v4_C(l2,t2,-*sichtweite);
    VECTOR3D v5_C(l2,b2,-*sichtweite);
    VECTOR3D v6_C(r2,b2,-*sichtweite);
    VECTOR3D v7_C(r2,t2,-*sichtweite);

    MATRIX4X4 invCamMatrix(*CVM);
    invCamMatrix.Invert();
    MATRIX4X4 invLightMatrix(*LVM);
    invLightMatrix.Invert();

    VECTOR3D c0_W(invCamMatrix*c0_C);
    VECTOR3D v0_W(invCamMatrix*v0_C);
    VECTOR3D v1_W(invCamMatrix*v1_C);
    ...
    //Berechnung für v2_W bis v7_W analog

    VECTOR3D c0_L(*LVM*c0_W);
    VECTOR3D v0_L(*LVM*v0_W);
    VECTOR3D v1_L(*LVM*v1_W);
    …
    //Berechnung für v2_L bis v7_L analog

    VECTOR3D c0_L_2D = VECTOR3D(c0_L.x, c0_L.y, -1);
    VECTOR3D v0_L_2D = VECTOR3D(v0_L.x, v0_L.y, -1);
    VECTOR3D v1_L_2D = VECTOR3D(v1_L.x, v1_L.y, -1);
   …
    //Berechnung für v2_L_2D bis v7_L_2D analog

    VECTOR3D mitteNear_C(0,0,-1);
    VECTOR3D mitteNear_W(invCamMatrix*mitteNear_C);
    VECTOR3D mitteNear_L(*LVM*mitteNear_W);
    VECTOR3D mitteNear_L_2D = VECTOR3D(mitteNear_L.x, mitteNear_L.y, -1);

    VECTOR3D mitteFar_C(0,0,-1*(*sichtweite));
    VECTOR3D mitteFar_W(invCamMatrix*mitteFar_C);
    VECTOR3D mitteFar_L(*LVM*mitteFar_W);
    VECTOR3D mitteFar_L_2D = VECTOR3D(mitteFar_L.x, mitteFar_L.y, -1);

    VECTOR3D richtungGerade_C(mitteFar_C-mitteNear_C);
    VECTOR3D richtungGerade_W(mitteFar_W-mitteNear_W);
    VECTOR3D richtungGerade_L(mitteFar_L-mitteNear_L);
    VECTOR3D richtungGerade_L_2D = VECTOR3D(richtungGerade_L.x, richtungGerade_L.y, 0);

    VECTOR3D up_C(0,1,0);
    VECTOR3D up_W(invLightMatrix*c0_C);
    VECTOR3D up_L(0,0,-1);
    VECTOR3D sichtlinie_L(0,0,-1);

    VECTOR3D richtungOrthogonale_C(up_C.CrossProduct(richtungGerade_C));
    VECTOR3D richtungOrthogonale_W(up_W.CrossProduct(richtungGerade_W));
    VECTOR3D richtungOrthogonale_L(up_L.CrossProduct(richtungGerade_L));
    VECTOR3D richtungOrthogonale_L_2D = VECTOR3D(richtungOrthogonale_L.x,richtungOrthogonale_L.y,0);

    // ... wird fortgesetzt
```

Listing 10a – Die Methode clalulateTrapezoid der Klasse TRAPSM, Teil 1

Die Methode *calculateTrapezoid* berechnet die vier Trapezpunkte aus Sicht der Lichtquelle und liefert sie an die übergebenen Variablen *t_0* bis *t_3* zurück. Dazu werden zunächst die Punkte des Kameraursprungs und des Kamerasichtvolumens in Kamerakoordinaten berechnet. Die Variable *c0_C* ist dabei der Kameraursprung, *v0_C* bis *c3_C* die Punkte der Near Plane, *v4_C* bis *v7_C* die Punkte der Far Plane.

Anschließend wird die inverse Matrix der Kameramatrix und der Lichtmatrix berechnet. Durch die Multiplikation eines Kamerapunktes mit der inversen Kameramatrix kann dieser in das Weltkoordinatensystem transformiert werden, durch die Multiplikation eines Lichtpunktes mit der inversen Lichtmatrix kann auch dieser in das Weltkoordinatensystem transformiert werden. Weltkoordinaten können mit der entsprechenden Kamera- oder Lichtmatrix in Kamera- bzw. Lichtkoordinaten umgerechnet werden.

Die Variable *c0_W* enthält nach der Multiplikation des Kameraursprungs in Kamerakoordinaten mit der inversen Kameramatrix den Kameraursprung in Weltkoordinaten. Analog werden so die Punkte *v0_W* bis *v7_W* berechnet, welche dann die Eckpunkte der Near und Far Plane in Weltkoordinaten enthalten.

Genauso kann man diese Koordinaten in Lichtkoordinaten transformieren, indem man die Weltkoordinaten mit der Lichtmatrix multipliziert. Die Variablen *c0_L* und *v0_L* bis *v7_L* enthalten nach der Multiplikation die Koordinaten des Kameraursprungs, sowie die Eckpunkte der Near und Far Plane des View Frustrums in Lichtkoordinaten. Die Eckpunkte des View Frustrums werden von nun an nur noch Eckpunkte genannt.

Alle Variablen in dieser Klasse, die die Endung *_C* besitzen, sind Punkte oder Vektoren in Kamerakoordinaten, alle Variablen mit der Endung *_W* sind Punkte oder Vektoren in Weltkoordinaten und alle Variablen mit der Endung *_L* sind Punkte oder Vektoren in Lichtkoordinaten.

Da für die Berechnung des Trapezes teilweise auch 2D Koordinaten benötigt werden, erzeugen wir noch die Variablen *c0_L_2D* und *v0_L_2D* bis *v7_L_2D*. Bei ihnen wird die Z-Koordinate auf einen festen Wert gesetzt. Auch Richtungsvektoren werden wie die Variable *richtungOrthogonale_L_2D* in 2D benötigt, also der Z-Wert auf 0 gesetzt.

Jetzt müssen die Mittelpunkte der Near und Far Plane bestimmt werden. Am einfachsten geschieht dies, indem man in Kamerakoordinaten vom Kameraursprung um eins entlang der Blickrichtung, also der negativen Z-Achse geht. Hier liegt der Mittelpunkt der Near Plane. Analog bekommt man den Mittelpunkt der Far Plane, indem man um die Länge der Sichtweite entlang der Blickrichtung geht. Durch die entsprechenden Matrixmultiplikationen können diese Punkte wieder in die relevanten Koordinatensysteme transformiert werden.

Die *richtungGerade*-Variablen enthalten die Richtungsvektoren der Gerade durch die Planemittelpunkte in den verschiedenen Koordinatensystemen. Diese Gerade wird von nun an Mittellinie genannt.

Im Kamerakoordinatensystem kann durch das Kreuzprodukt des Up-Vektors und der *richtungGerade_C* ein orthogonaler Vektor bestimmt werden, welcher in *richtungOrthogonale_C* gespeichert wird. Genauso wird auch *richtungOrthogonale_W* bestimmt, nur dass hier als Up-Vektor der Vektor vom Weltkoordinatenursprung zur Lichtquelle gewählt wird.
Diese Orthogonalen werden von nun an Mittellinienorthogonalen genannt.

```
VECTOR3D schnitt1_L,schnitt2_L,schnitt3_L,schnitt4_L,schnitt5_L,schnitt6_L,schnitt7_L,schnitt8_L,pb;

VECTOR3D punktOrthogonaleRechts1_L(v0_L.x+richtungOrthogonale_L.x,v0_L.y+richtungOrthogonale_L.y,
                                   v0_L.z+richtungOrthogonale_L.z);
LineLineIntersect3D(v0_L,punktOrthogonaleRechts1_L,mitteNear_L,mitteFar_L,schnitt1_L,pb);
// Berechnung für schnitt2_L bis schnitt8_L analog

VECTOR3D schnittdown1_L(schnitt1_L - sichtlinie_L);
LineLineIntersect3D(mitteNear_L, mitteFar_L, schnitt1_L, schnittdown1_L, schnitt1_L, pb);
// Berechnung für schnitt2_L bis schnitt8_L analog

VECTOR3D schnitt1_L_2D,schnitt2_L_2D,schnitt3_L_2D,schnitt4_L_2D,schnitt5_L_2D,schnitt6_L_2D,
         schnitt7_L_2D,schnitt8_L_2D;

VECTOR3D punktOrthogonaleRechts1_L_2D(v0_L_2D.x+richtungOrthogonale_L_2D.x,
                                      v0_L_2D.y+richtungOrthogonale_L_2D.y,-1);
LineLineIntersect3D(v0_L_2D,punktOrthogonaleRechts1_L_2D,mitteNear_L_2D,mitteFar_L_2D,schnitt1_L_2D,pb);
// Berechnung für schnitt2_L_2D bis schnitt8_L_2D analog

VECTOR3D schnitt1_W(invLightMatrix*schnitt1_L);
VECTOR3D schnitt2_W(invLightMatrix*schnitt2_L);
// Berechnung für schnitt1_W bis schnitt8_W analog

VECTOR3D schnittMin_L, schnittMax_L, schnittMax_L_2D, schnittMin_L_2D;
schnittMin_L_2D = schnitt1_L_2D;
schnittMin_L = schnitt1_L;
VECTOR3D richtungC0S1(schnitt1_L_2D-c0_L_2D);
VECTOR3D richtungC0S2(schnitt2_L_2D-c0_L_2D);
VECTOR3D richtungC0S3(schnitt3_L_2D-c0_L_2D);
VECTOR3D richtungC0S4(schnitt4_L_2D-c0_L_2D);

if(richtungC0S2.GetLength() < richtungC0S1.GetLength())
{
    schnittMin_L = schnitt2_L;
    schnittMin_L_2D = schnitt2_L_2D;
}
if(richtungC0S3.GetLength() < richtungC0S1.GetLength() && richtungC0S3.GetLength() < richtungC0S2.GetLength())
{
    schnittMin_L = schnitt3_L;
    schnittMin_L_2D = schnitt3_L_2D;
}
if(richtungC0S4.GetLength() < richtungC0S1.GetLength()  && richtungC0S4.GetLength() < richtungC0S2.GetLength() &&
   richtungC0S4.GetLength() < richtungC0S3.GetLength())
{
    schnittMin_L = schnitt4_L;
    schnittMin_L_2D = schnitt4_L_2D;
}
// Berechnung für schnittMax_L und schnittMax_L_2D analog, nur Test ob Länge größer, nicht kleiner

VECTOR3D schnittMin_W(invLightMatrix*schnittMin_L);
VECTOR3D schnittMin_W_2D(invLightMatrix*schnittMin_L_2D);
VECTOR3D schnittMax_W(invLightMatrix*schnittMax_L);
VECTOR3D schnittMax_W_2D(invLightMatrix*schnittMax_L_2D);

// … wird fortgesetzt
```

Listing 10b – Die Methode clalulateTrapezoid der Klasse TRAPSM, Teil 2

Das nächste Ziel zur Konstruktion des Trapezes ist, die zwei Mittellinienorthogonalen zu finden, die das gesamte View Frustrum einschließen und durch die äußeren zwei Eckpunkte gehen. Dazu wird zunächst an jeden der acht Eckpunkte die Mittellinienorthogonale angelegt und der Schnittpunkt mit der Mittellinie berechnet. Hierfür wird die Methode *LineLineIntersect3D* verwendet. Die ersten zwei Parameter die die Methode übergeben bekommt, sind der Start- und Endpunkt der ersten Geraden, die nächsten zwei der Start- und Endpunkt der zweiten Geraden. Als fünfter Parameter wird eine Variable übergeben, die nach dem Methodenaufruf den Schnittpunkt der Geraden, oder einen Punkt, von dem aus der Abstand der beiden Geraden minimal ist, enthält. Der sechste Parameter spielt hier keine Rolle und kann ignoriert werden.

Zunächst werden die Variablen *schnitt1_L* bis *schnitt8_L* initialisiert, die die Schnittpunkte erhalten sollen. *schnitt1_L* bis *schnitt4_L* beziehen sich auf die Eckpunkte der Near Plane, *schnitt5_L* bis *schnitt8_L* auf die Eckpunkte der Far Plane. Die *punktOrthogonaleRechts*-Variablen sind je ein Punkt rechts des aktuellen Eckpunktes auf der Mittellinienorthogonalen durch diesen Eckpunkt.

Jetzt kann der Schnittpunkt zwischen der Mittellinie und den entsprechenden Mittellinienorthogonalen durch die Eckpunkte berechnet werden. Da sich diese Geraden jedoch nicht schneiden, liefert die Methode *LineLineIntersect3D* nur einen Punkt zurück, von dem aus der Abstand der beiden Geraden minimal ist. Um den tatsächlichen Schnittpunkt noch zu finden, wird jetzt der Schnittpunkt der Mittellinie und der Geraden vom jeweiligen Schnittpunkt in Sichtrichtung der Lichtquelle berechnet. Jetzt sind die tatsächlichen Schnittpunkte in den Variablen *schnitt1_L* bis *schnitt8_L* gespeichert.

Auch hier werden die 2D-Schnittpunkte noch benötigt. Sie errechnen sich analog zu den 3D-Schnittpunkten, nur dass hier die 2D-Variablen benutzt werden. Zudem ist hier eine weitere Schnittpunktberechnung mit der Sichtlinie nicht nötig.

Durch die Multiplikation mit der inversen Lichtmatrix können die Punkte in Weltkoordinaten bestimmt werden, die in den Variablen *schnitt1_W* bis *schnitt8_W* gespeichert werden.

Um nun herauszufinden, welcher Schnittpunkt der unterste auf der Mittellinie ist, werden die Richtungsvektoren vom Kameraursprung zu den eben errechneten 2D-Schnittpunkten der Near Plane (*schnitt1_L_2D* bis *schnitt4_L_2D*) gebildet und deren Länge bestimmt. Dies geschieht durch den Aufruf *getLength()*. Die folgenden if-Abfragen bestimmen den kleinsten Abstand und weist der Variablen *schnittMin_L* und *schnittMin_L_2D* den entsprechenden Schnittpunkt zu.

Das gleiche Verfahren wird auch für die Bestimmung des obersten Schnittpunktes auf der Mittellinie benutzt. Dazu werden auch hier die Richtungsvektoren vom Kameraursprung zu den eben errechneten Schnittpunkten mit der Far Plane (*schnitt5_L_2D* bis *schnitt8_L_2D*) gebildet und auch deren Länge mit *getLength()* bestimmt. Allerdings ermitteln die folgenden IF-Abfragen nun den größten Abstand und weisen der Variablen *schnittMax_L* und *schnittMax_L_2D* den entsprechenden Schnittpunkt zu.

Am Ende werden diese Schnittpunkte durch Multiplikation mit der inversen Lichtmatrix in Weltkoordinaten umgerechnet. Diese Punkte bilden jetzt mit der Mittellinienorthogonalen die Base und Top Line des Trapezes.

```
VECTOR3D mittelLinieMinMax_L(schnittMax_L-schnittMin_L);
float lambda = 1.0f;
float focusRegion = 0.4f;
float deltaStrich = focusRegion*lambda;
float n = (lambda*deltaStrich + lambda*deltaStrich*-(1-focusRegion))/(lambda-2*deltaStrich-lambda*-(1-focusRegion));

VECTOR3D nDot_L(schnittMin_L - (n*mittelLinieMinMax_L));
VECTOR3D nDot_W = invLightMatrix*nDot_L;
VECTOR3D nDot_L_2D = VECTOR3D(nDot_L.x,nDot_L.y,0);

//... wird fortgesetzt
```

Listing 10c – Die Methode clalulateTrapezoid der Klasse TRAPSM, Teil 3

Nun optimieren wir die Focus Region. Dazu werden die Seitenlinien des Trapezes nicht durch den Kameraursprung gezogen, sondern durch einen Punkt, der hinter dem Kameraursprung liegt. Das hat zur Folge, dass bei der Umrechnung des Trapezes zum Quadrat die vordere Region nicht zu stark oversampled, und damit die hinteren Regionen zu große Artefakte erzeugen.

Die Entfernung dieses Punktes vom Punkt *schnittMin_L* bzw. *schnittMin_L_2D* wird in der Variable *n* gespeichert. *Ndot* ist der gesuchte Punkt, der in diesem Abstand vom Punkt *schnittMin_L* bzw. *schnittMin_L_2D* entfernt liegt. Die Berechnung geschieht im Beispielprogramm dabei nach einer Formel, die von der in der Veröffentlichung zum Trapezoidal Shadow Mapping angegebenen leicht abweicht und hier bessere Ergebnisse liefert.

Durch den Wert *focusRegion* kann bestimmt werden, wie groß die Focus Region ausfallen soll. Wird der Wert erhöht, so vergrößert sich die Focus Region entlang der Sichtlinie. Eine größere Focus Region bringt bei einem Kameraursprung, der weiter vom Boden entfernt ist, bessere Ergebnisse reduziert aber gleichzeitig die Schattenqualität, wenn sich die Kamera dem Boden nähert.

Hier wäre ein dynamischer Ansatz denkbar, der je nach Situation diesen Wert berechnet, um stets die besten Ergebnisse zu erhalten. In der Szene aus dieser Arbeit hat sich ein Wert von 0.4 als Focus Region als sehr gut erwiesen.

```
VECTOR3D sl1 = VECTOR3D(nDot_L - v4_L);
VECTOR3D schnittSL1;
LineLineIntersect3D(nDot_L,nDot_L+sl1,schnittMax_L,schnittMax_L+richtungOrthogonale_L,schnittSL1,pb);

VECTOR3D sl1_2D = VECTOR3D(nDot_L_2D - v4_L_2D);
VECTOR3D schnittSL1_2D;
LineLineIntersect3D(nDot_L_2D,nDot_L_2D+sl1_2D,schnittMax_L_2D,schnittMax_L_2D+richtungOrthogonale_L_2D,
                    schnittSL1_2D,pb);
// Berechnung von schnittSL2 bis schnittSL4 und schnittSL2_2D bis schnittSL4_2D analog

VECTOR3D schnittSL1_W(invLightMatrix*schnittSL1_2D);
// Berechnung schnittSL2_W bis schnittSL4_W analog

VECTOR3D schnittSL1_C(*CVM*schnittSL1_W);
//Berechnung schnittSL2_C bis schnittSL4_C analog

if(schnittSL1_C.x > schnittSL2_C.x && schnittSL1_C.x > schnittSL3_C.x && schnittSL1_C.x > schnittSL4_C.x)
t_1 = schnittSL1_2D;
else if(schnittSL2_C.x > schnittSL1_C.x && schnittSL2_C.x > schnittSL3_C.x && schnittSL2_C.x > schnittSL4_C.x)
t_1 = schnittSL2_2D;
else if(schnittSL3_C.x > schnittSL1_C.x && schnittSL3_C.x > schnittSL2_C.x && schnittSL3_C.x > schnittSL4_C.x)
t_1 = schnittSL3_2D;
else t_1 = schnittSL4_2D;

if(schnittSL1_C.x < schnittSL2_C.x && schnittSL1_C.x < schnittSL3_C.x && schnittSL1_C.x < schnittSL4_C.x)
t_0 = schnittSL1_2D;
else if(schnittSL2_C.x < schnittSL1_C.x && schnittSL2_C.x < schnittSL3_C.x && schnittSL2_C.x < schnittSL4_C.x)
t_0 = schnittSL2_2D;
else if(schnittSL3_C.x < schnittSL1_C.x && schnittSL3_C.x < schnittSL2_C.x && schnittSL3_C.x < schnittSL4_C.x)
t_0 = schnittSL3_2D;
else t_0 = schnittSL4_2D;

VECTOR3D punktLinks_W(schnittMax_L_2D – 10000 * richtungOrthogonale_L_2D);
VECTOR3D laengeLinks1(t_0 - punktLinks_W);
VECTOR3D laengeLinks2(t_1 - punktLinks_W);
float abstand1 = laengeLinks1.GetLength();
float abstand2 = laengeLinks2.GetLength();

if(abstand1 < abstand2)
{
    VECTOR3D temp(t_0);
    t_0 = t_1;
    t_1 = temp;
}

VECTOR3D RichtungSL1_L = nDot_L_2D - t_0;
VECTOR3D RichtungSL2_L = nDot_L_2D - t_1;

LineLineIntersect3D(nDot_L_2D, nDot_L_2D+RichtungSL2_L, schnittMin_L_2D, schnittMin_L_2D+richtungOrthogonale_L_2D,t_2,pb);
LineLineIntersect3D(nDot_L_2D, nDot_L_2D+RichtungSL1_L, schnittMin_L_2D, schnittMin_L_2D+richtungOrthogonale_L_2D,t_3,pb);

// … wird fortgesetzt
```

Listing 10d – Die Methode clalulateTrapezoid der Klasse TRAPSM, Teil 4

Jetzt hat man alle benötigten Werte berechnet, um die Seitenlinien zu konstruieren. Dazu bildet man zunächst alle Richtungsvektoren vom Punkt *nDot_L* zu den vier Eckpunkten der Far Plane und berechnet anschließend deren Schnittpunkte mit der Top Line. Diese werden in den Variablen *schnittSL1* bis *schnittSL4* gespeichert. Zusätzlich werden noch die 2D-Schnittpunkte mit den entsprechenden 2D-Variablen berechnet.

Diese werden anschließend noch in Welt- und Kamerakoordinaten umgerechnet und in den entsprechenden Variablen gespeichert.

Als nächste Aufgabe stellt sich, herauszufinden, welche zwei der vier Schnittpunkte außen liegen und welcher der linke bzw. rechte Punkt ist. Dazu wird einfach der X-Wert der Schnittpunkte in Kamerakoordinaten verglichen. Der Punkt, dessen X-Wert am kleinsten ist liegt links und wird als *t_0* bezeichnet, der Punkt, dessen X-Wert am größten ist liegt rechts und wird als *t_1* bezeichnet.

Sollte die Kamera jedoch auf dem Kopf stehen, funktioniert dieser Test nicht mehr korrekt. Darum wird noch ein Punkt in Lichtkoordinaten angenommen, der sehr weit links liegt. Er wird gebildet, indem man vom *schnittMax_L_2D* sehr weit in Richtung der Mittellinienorthogonalen geht.
Ist die Länge des Vektors von diesem Punkt zu dem Punkt *t_0* größer als die Länge des Vektors von diesem Punkt zum Punkt *t_1*, so müssen *t_0* und *t_1* vertauscht werden.

Da jetzt bekannt ist, dass die Linien, die durch den Punkt *nDot* und *t_0* bzw. *t_1* gehen die Seitenlinien bilden, kann mit deren Schnittpunkt mit der Base Line *t_2* und *t_3* bestimmt werden. Auch hier ist auf die richtige Reihenfolge zu achten. Der Schnittpunkt der Seitenlinie über *t_1* mit der Base Line bildet den Punkt *t_2*, der Schnittpunkt der Seitenlinie über *t_0* mit der Base Line bildet den Punkt *t_3*.

Durch diese Maßnahmen liegen die Punkte *t_0* bis *t_3* jetzt im Uhrzeigersinn aus Kamerasicht, beginnend mit dem Punkt links auf der Far Plane.

```
// Test, ob FAR-Plane als Trapezoid genommen wird.
VECTOR3D richtungO1 = v4_L - v5_L;
VECTOR3D richtungO2 = v5_L - v6_L;
VECTOR3D richtungO3 = v6_L - v7_L;
VECTOR3D richtungO4 = v7_L - v4_L;

richtungO1 = VECTOR3D(-richtungO1.y,richtungO1.x,0);
richtungO2 = VECTOR3D(-richtungO2.y,richtungO2.x,0);
richtungO3 = VECTOR3D(-richtungO3.y,richtungO3.x,0);
richtungO4 = VECTOR3D(-richtungO4.y,richtungO4.x,0);

VECTOR3D richtungC1a = v4_L - v0_L;
VECTOR3D richtungC2a = v5_L - v0_L;
VECTOR3D richtungC3a = v6_L - v0_L;
VECTOR3D richtungC4a = v7_L - v0_L;

VECTOR3D richtungC1b = v4_L - v1_L;
VECTOR3D richtungC2b = v5_L - v1_L;
VECTOR3D richtungC3b = v6_L - v1_L;
VECTOR3D richtungC4b = v7_L - v1_L;

VECTOR3D richtungC1c = v4_L - v2_L;
VECTOR3D richtungC2c = v5_L - v2_L;
VECTOR3D richtungC3c = v6_L - v2_L;
VECTOR3D richtungC4c = v7_L - v2_L;

VECTOR3D richtungC1d = v4_L - v3_L;
VECTOR3D richtungC2d = v5_L - v3_L;
VECTOR3D richtungC3d = v6_L - v3_L;
VECTOR3D richtungC4d = v7_L - v3_L;

if (richtungO1.DotProduct(richtungC1a) > 0 && richtungO2.DotProduct(richtungC2a) > 0 && richtungO3.DotProduct(richtungC3a) > 0
   && richtungO4.DotProduct(richtungC4a) > 0 || richtungO1.DotProduct(richtungC1b) > 0 && richtungO2.DotProduct(richtungC2b) > 0
   && richtungO3.DotProduct(richtungC3b) > 0 && richtungO4.DotProduct(richtungC4b) > 0 || richtungO1.DotProduct(richtungC1c) > 0
   && richtungO2.DotProduct(richtungC2c) > 0 && richtungO3.DotProduct(richtungC3c) > 0 && richtungO4.DotProduct(richtungC4c) > 0
   || richtungO1.DotProduct(richtungC1d) > 0 && richtungO2.DotProduct(richtungC2d) > 0 && richtungO3.DotProduct(richtungC3d) > 0
   && richtungO4.DotProduct(richtungC4d) > 0)
{
            t_0 = v4_L;
            t_1 = v5_L;
            t_2 = v6_L;
            t_3 = v7_L;
}

t_0 = t_0/40;
t_1 = t_1/40;
t_2 = t_2/40;
t_3 = t_3/40;

}
```

Listing 10e – Die Methode clalulateTrapezoid der Klasse TRAPSM, Teil 5

Da das Trapez jedoch nicht gebildet werden kann, sobald die Kamera direkt in die Lichtquelle oder genau entgegengesetzt der Lichtquelle blickt, muss diese Situation abgefangen und extra behandelt werden. Allgemein tritt dieser Fall ein, wenn sich aus Sicht der Lichtquelle einer der Eckpunkte der Near Plane innerhalb der Far Plane befindet. Um herauszufinden, ob diese Situation eingetreten ist, müssen zunächst die Richtungsvektoren der Far Plane Eckpunkte entgegengesetzt des Uhrzeigersinns gebildet werden. Im nächsten Schritt werden daraus die Orthogonalenvektoren errechnet und in den Variablen *richtungO1* bis *richtungO4* gespeichert.
In den Variablen *richtungC1a* bis *richtungC4d* werden die Richtungsvektoren aller Far Plane Eckpunkte zu allen Near Plane Eckpunkten gespeichert.

In der if-Abfrage wird jetzt das Punktprodukt aus der Orthogonalen dem entsprechenden Richtungsvektor des Far Plane Eckpunktes zum Near Plane Eckpunkt berechnet. Wird hier auch nur ein einziger Wert kleiner 0, so kann davon ausgegangen werden, dass sich alle vier Near Plane Eckpunkte innerhalb der Far Plane befinden. In diesem Fall werden die Far Plane Eckpunkte als Trapezpunkte benutzt.

In einigen Fällen funktioniert dieser Test jedoch nicht korrekt, da zu früh auf die Far Plane als Trapez umgeschaltet wird, und daher Bereiche entstehen, für die keine korrekte Shadow Map entsteht. Da zudem bei großen Sichtweiten die Far Plane sehr groß ausfällt, verringert sich in diesen Fällen die Qualität der Schatten zusätzlich. Hier wäre ein besserer Ansatz denkbar.

Als letzter Schritt werden die eben bestimmten Trapezpunkte noch durch die seitliche Sichtweite der Lichtquelle geteilt, um auf einen Wert zwischen –1 und 1 zu kommen. In diesem Programm müssen die Trapezpunkte durch 40 geteilt werden, da das Sichtvolumen der Lichtquelle mit *glOrtho(-40,40,-40,40, 1.0f, 200.0f);* angegeben war.

5.2.4 Die Methode drawCamera

```
void TRAPSM::drawCamera()
{
    glMatrixMode( GL_PROJECTION );
    glLoadMatrixf(*CPM);
    glMatrixMode( GL_MODELVIEW );
    glLoadMatrixf(*CVM);

    glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );

    static float genS[] = { 1.0f, 0.0f, 0.0f, 0.0f };
    static float genT[] = { 0.0f, 1.0f, 0.0f, 0.0f };
    static float genR[] = { 0.0f, 0.0f, 1.0f, 0.0f };
    static float genQ[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    const GLfloat bias[] = {0.5, 0.0, 0.0, 0.0,
                            0.0, 0.5, 0.0, 0.0,
                            0.0, 0.0, 0.5, 0.0,
                            0.5, 0.5, 0.5, 1.0};

    glEnable( GL_TEXTURE_2D );
    glEnable( GL_TEXTURE_GEN_S );
    glEnable( GL_TEXTURE_GEN_T );
    glEnable( GL_TEXTURE_GEN_R );
    glEnable( GL_TEXTURE_GEN_Q );
    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGeni( GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGeni( GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
    glTexGenfv( GL_S, GL_EYE_PLANE, genS );
    glTexGenfv( GL_T, GL_EYE_PLANE, genT );
    glTexGenfv( GL_R, GL_EYE_PLANE, genR );
    glTexGenfv( GL_Q, GL_EYE_PLANE, genQ );

    glMatrixMode(GL_TEXTURE);
    glLoadMatrixf(bias);
    glMultMatrixf(N_T);
    glMultMatrixf(*LPMO);
    glMultMatrixf(*LVM);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(*CPM);
    glLoadMatrixf(*CVM);

    glBindTexture( GL_TEXTURE_2D, shadowtex);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE_ARB,GL_COMPARE_R_TO_TEXTURE_ARB);

    glEnable(GL_FOG);
    drawScene();
    glDisable(GL_FOG);
}
```

Listing 11 – Die Methode drawCamera der Klasse TRAPSM

Die Methode *drawCamera* zeichnet schließlich die Szene mit Schatten, führt also den „Second Pass“ des Shadow Mappings aus. Zunächst löschen wir wieder den Color und Depth Buffer. Anschließend wird die Projektions und Modelview Matrix der Kamera geladen, die Texturkoordinaten und die Matrix initialisiert, welche die Texturkoordinaten mit der Dimension [-1,1] in die Dimension [0,1] transformiert. Die Texturmatrix wird aktiviert und die benötigten Matrizen miteinander verrechnet. Die Shadow Map wird geladen, auf die Szene projiziert und der Schattentest durchgeführt. Abschließend wird die Szene mit Schatten gezeichnet.

5.2.5 Die Methode drawLight

```
void TRAPSM::drawLight()
{
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(N_T);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(*LPMO);
    glMultMatrixf(*LVM);
    drawScene();
}
```

Listing 12 – Die Methode drawLigth der Kalsse TRAPSM

Die Methode *drawLight* zeichnet die Szene, über die die Shadow Map erzeugt wird, ohne jedoch den Schattentest durchzuführen und Schatten zu erzeugen.

Dazu wird die Transformationsmatrix *N_T* als Projektionsmatrix benutzt und die Projektions und Modelview Matrix der Lichtquelle geladen. Anschließend wird die Szene gezeichnet.

In dieser Sicht kann man deutlich die verschieden hohe Gewichtung der Objekte in der Shadow Map erkennen. Objekte, die näher an der Kamera liegen werden größer dargestellt (oberer Bereich) und erhalten somit eine höhere Auflösung als Objekte die weiter entfernt liegen (unterer Bereich) und kleiner dargestellt sind.

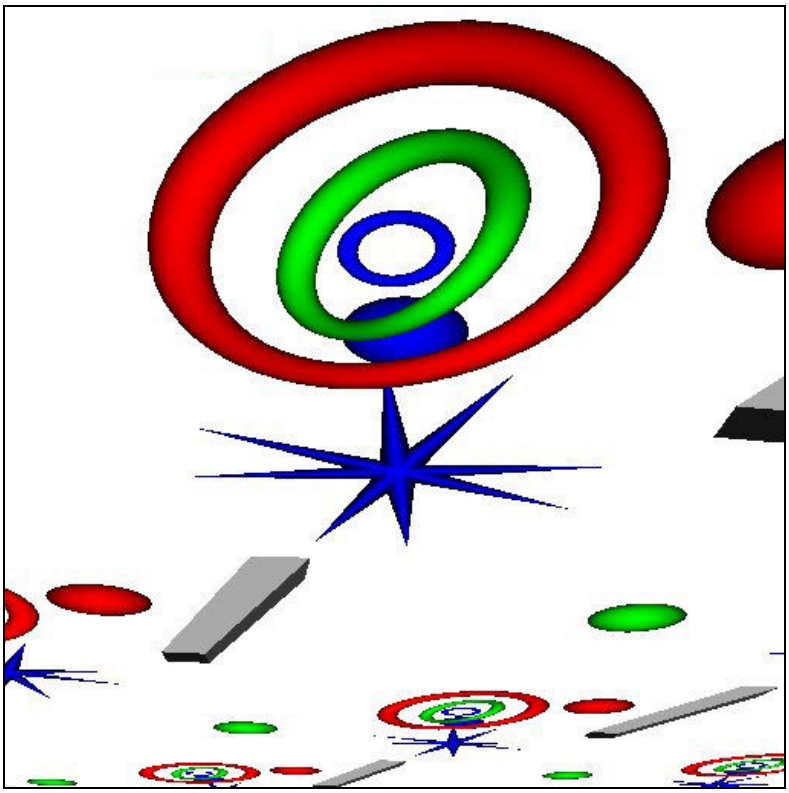


Bild 18 – Bereich, über den beim Trapezoidal Shadow Mapping die Shadow Map aufgebaut wird

5.2.6 Die Methode end

```
void TRAPSM::end()
{
    glDisable( GL_TEXTURE_2D );
    glMatrixMode( GL_TEXTURE );
    glLoadIdentity();
    glDisable( GL_TEXTURE_2D );
    glDisable( GL_TEXTURE_GEN_S );
    glDisable( GL_TEXTURE_GEN_T );
    glDisable( GL_TEXTURE_GEN_R );
    glDisable( GL_TEXTURE_GEN_Q );
}
```

Listing 13 – Die Methode end der Klasse TRAPSM

Die Methode *end* wird nach dem zeichnen der Szene mit Schatten aufgerufen. Hiermit wird die Texturmatrix zurückgesetzt und die Texturen deaktiviert.

5.3 Weiche Schattenkanten

Die Methoden zur Erzeugung weicher Schattenkanten wurden im Hauptprogramm implementiert. Ihre Funktionsweise wird im folgenden Kapitel beschrieben.

5.3.1 Die Methode softTSM

```
void softTSM()
{
    glClear(GL_ACCUM_BUFFER_BIT);
    MATRIX4X4 lightViewMatrixBak;
    lightViewMatrixBak = lightViewMatrix;

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt( lightPositionVec.x-softShadowDistTSM, lightPositionVec.y, lightPositionVec.z-softShadowDistTSM,
                lightDirectionVec.x, lightDirectionVec.y, lightDirectionVec.z,
                0.0f, 1.0f, 0.0f);
    glGetFloatv(GL_MODELVIEW_MATRIX, lightViewMatrix);
    TSM();
    glAccum (GL_ACCUM, 1.0/16.0);

    lightViewMatrix = lightViewMatrixBak;
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt( lightPositionVec.x-softShadowDistTSM, lightPositionVec.y, lightPositionVec.z+softShadowDistTSM,
                lightDirectionVec.x, lightDirectionVec.y, lightDirectionVec.z,
                0.0f, 1.0f, 0.0f);
    glGetFloatv(GL_MODELVIEW_MATRIX, lightViewMatrix);
    TSM();
    glAccum (GL_ACCUM, 1.0/16.0);

    // weitere 14 Schritte analog mit unterschiedlicher Verschiebung der Lichtquellenposition

    glAccum (GL_RETURN, 1.0f);
}
```

Listing 14 – Die Methode softTSM der Klasse TRAPSM

Zur Erzeugung von weichen Schattenkanten verwenden wir den Accumulation Buffer, den wir zunächst löschen. Anschließend wird die original *lightViewMatrix* in *lightViewMatrixBak* gespeichert, um sie nach jeder Verschiebung wieder herstellen zu können.

Jetzt verändern wir die Position der Lichtquelle mit *glLookAt* in jedem Schritt leicht. Dazu wird die Variable *softShadowDistTSM* verwendet, welche den Maximalabstand zwischen den Verschiebungen enthält. Nach der Verschiebung wird die Szene mit dem Aufruf *TSM()* gerendert, welche im Hauptprogramm für die Darstellung des Trapezoidal Shadow Mapping verantwortlich ist. Dann wird der Color Buffer in den Accumulation Buffer addiert, was mit dem Aufruf *glAccum(GL_ACCUM,1.0f/16.0f)* geschieht. Dabei muss dieser durch die Anzahl der Gesamtbilder geteilt werden, die gemeinsam in den Accumulation Buffer geschrieben werden sollen. In diesem Programm zeichnen wir 16 Szenen mit verschiedenen Verschiebungen.

Nachdem alle Szenen in den Accumulation Buffer geschrieben wurden, wird dieser mit dem Aufruf *glAccum(GL_RETURN, 1.0f);* wieder in den Color Buffer zurück geschrieben.

Auf diese Weile lassen sich relativ einfach weiche Schattenkanten erzeugen. Je größer die Anzahl der Verschiebungen, desto besser wird das Ergebnis. Allerdings muss dabei berücksichtigt werden, dass das Shadow Mapping ein Two-Pass-Verfahren ist, die Szene also pro Bild zweimal gerendert wird. Das bedeutet bei 16 Verschiebungen ganze 32 Rendering Passes, bevor ein Bild gezeichnet wird.

5.3.2 Die Methode softSM

Der Aufbau der Methode *softSM* ist genauso aufgebaut wie Methode *softTSM*, allerdings benutzt diese nicht das Trapezoidal Shadow Mapping, sondern das Standard Shadow Mapping. Die Methode dient dem Vergleich der Darstellung von weichen Schattenkanten mit Trapezoidal Shadow Mapping und Standard Shadow Mapping. So kann eine Szene mit Trapezoidal Shadow Mapping bei einer deutlich geringeren Auflösung der Shadow Map dieselben Ergebnisse liefern, wie die Darstellung mit dem Standard Shadow Mapping. So sieht die Szene zum Beispiel mit Trapezoidal Shadow Mapping mit einer Shadow Map Auflösung von nur 128x128 Pixeln fast genauso gut aus, wie die Szene mit Standard Shadow Mapping und einer Shadow Map Auflösung von 512x512 Pixeln.

6.0 Ergebnisse und Vergleich

In diesem Kapitel werden die verschiedenen Schattenverfahren anhand von Screenshots miteinander verglichen und auch deren Vor- und Nachteile bildlich dargestellt

6.1 Schattenqualität

Im folgenden Bild sind die Ergebnisse des Standard Shadow Mappings und des Trapezoidal Shadow Mappings bei verschieden hohen Auflösungen der Shadow Map zu erkennen. Die obere Reihe beinhaltet dabei die Ergebnisse des Standard Shadow Mappings, die untere Reihe die des Trapezoidal Shadow Mappings.

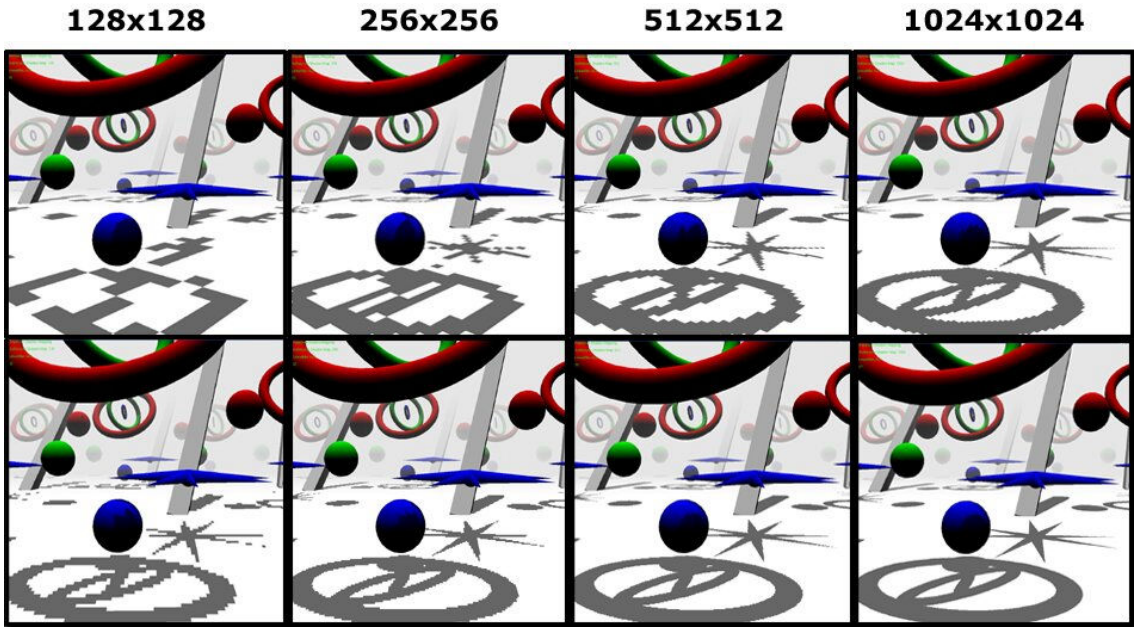


Bild 18 – Ergebnisse vom Standard Shadow Mapping und Trapezoidal Shadow Mapping

Bei einer sehr geringen Shadow Map Auflösung von 128x128 Pixeln sind beim Standard Shadow Mapping nur grobe Schattenblöcke zu erkennen. Auch bei 256x256 und 512x512 Pixeln ist das Ergebnis noch unbefriedigend und es bilden sich große Artefakte an den Schattenkanten. Sogar bei einer sehr hohen Auflösung von 1024x1024 Pixeln sind noch Artefakte zu erkennen.

Beim Trapezoidal Shadow Mapping bilden sich bei der geringsten Shadow Map Auflösung von 128x128 Pixeln zwar auch starke Artefakte, aber die Konturen der Objekte, die den Schatten werfen sind im Gegensatz zu denen beim Standard Shadow Mapping bereits deutlich zu erkennen. Bei einer Auflösung von 256x256 ist das Ergebnis mit dem des Standard Shadow Mappings bei einer Auflösung von 1024x1024 Pixeln vergleichbar. Bereits hier sind nur noch kleine Artefakte an den Schattenkanten erkennbar.

Ab einer Auflösung der Shadow Map von 512x512 Pixeln erhält man ein optimales Ergebnis. Die Schattenkanten sind klar erkennbar und zeigen keinerlei Artefakte mehr. Nur in den hinteren Bereichen sind noch kleinere Kanten erkennbar, die allerdings bei einer Auflösung von 1024x1024 auch komplett verschwunden sind.

Das Trapezoidal Shadow Mapping bringt aber gerade bei kleinen Objekten, oder Objekten mit vielen geometrischen Details seine Überlegenheit zum Standard Shadow Mapping deutlich zur Geltung, wie Bild 19 eindrucksvoll beweist.

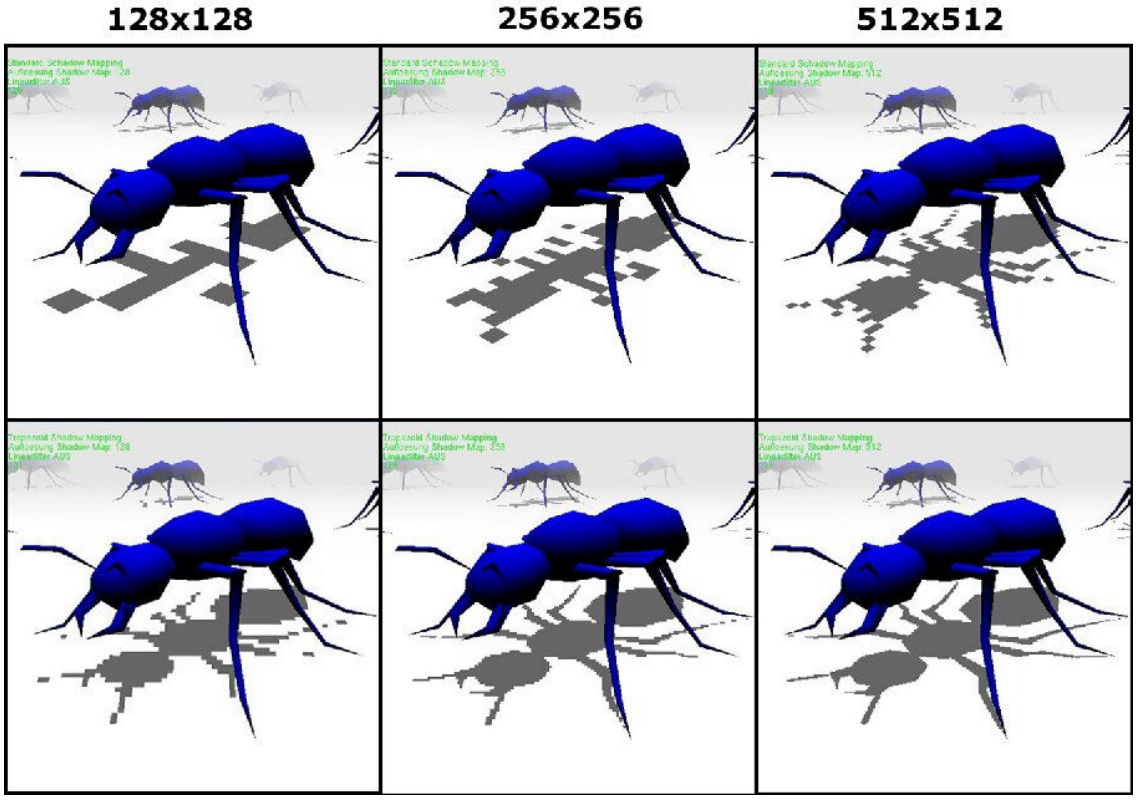


Bild 19 – Objekt mit kleinen geometrischen Details bei verschiedenen Schattenverfahren. Oben: Standard Shadow Mapping. Unten: Trapezoidal Shadow Mapping

Auch hier kann man bereits bei einer geringen Shadow Map Auflösung von nur 128x128 Pixeln das Objekt anhand des Schattenwurfs klar erkennen, während dies beim Standard Shadow Mapping erst ab einer Auflösung von 512x512 Pixeln möglich ist. Bei dieser Auflösung erzeugt das Trapezoidal Shadow Mapping bereits perfekte Ergebnisse ohne Artefakte.

Wie die gezeigten Bilder verdeutlichen, erreicht das Trapezoidal Shadow Mapping bereits ab einem Viertel der Shadow Map Auflösung des Standard Shadow Mappings vergleichbare Ergebnisse. Bei höheren Auflösungen erreicht man damit Ergebnisse, die mit dem Standard Shadow Mapping nur mit noch höheren Auflösungen oder gar überhaupt nicht zu erreichen wären.

Um noch einmal zu verdeutlichen, wie es schon bei so geringen Auflösungen des Shadow Map zu so guten Ergebnissen kommt, sind in Bild 20 die Shadow Maps des Standard Shadow Mappings und des Trapezoidal Shadow Mappings zu sehen.

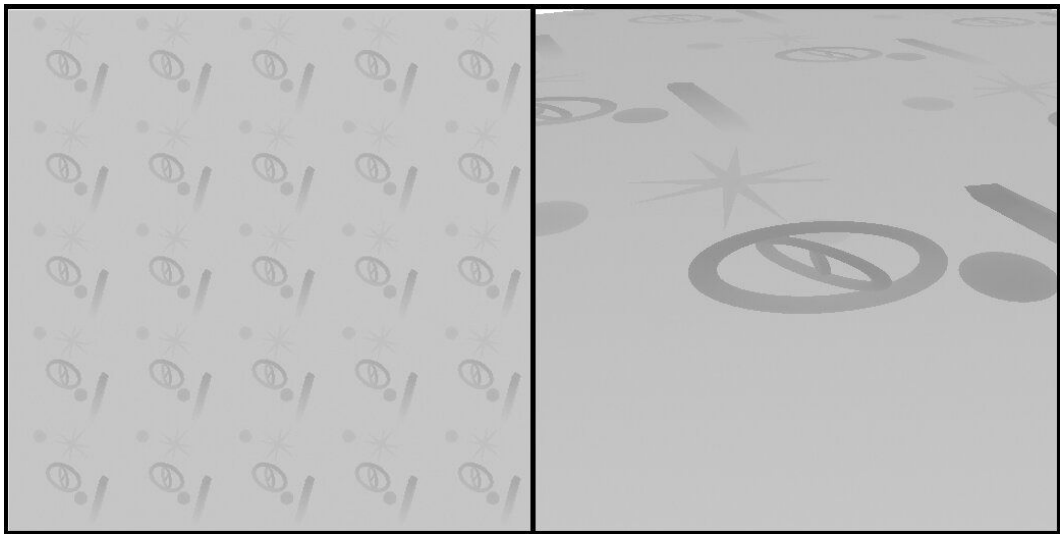


Bild 20 – Shadow Map des Standard Shadow Mappings und Trapezoidal Shadow Mapping

Im linken Bild ist die Shadow Map der Szene des Standard Shadow Mappings zu erkennen, im rechten Bild die des Trapezoidal Shadow Mappings.

Während das Standard Shadow Mapping diese über die ganze Szene aufbaut, erzeugt das Trapezoidal Shadow Mapping die Shadow Map nur über einem kleinen Teil der Szene, die vom Kamera Frustrum aus Sicht der Lichtquelle erkennbar ist. Durch diesen Unterschied wird klar, warum schon bei niedrigen Auflösungen das Trapezoidal Shadow Mapping deutlich bessere Ergebnisse als das Standard Shadow Mapping liefert.

Auch bei den weichen Schattenkanten liefert das Trapezoidal Shadow Mapping eine deutlich bessere Qualität, wie Bild 20 und 21 zeigt. Links ist jeweils die Szene ohne Linearfilter, rechts mit Linearfilter.

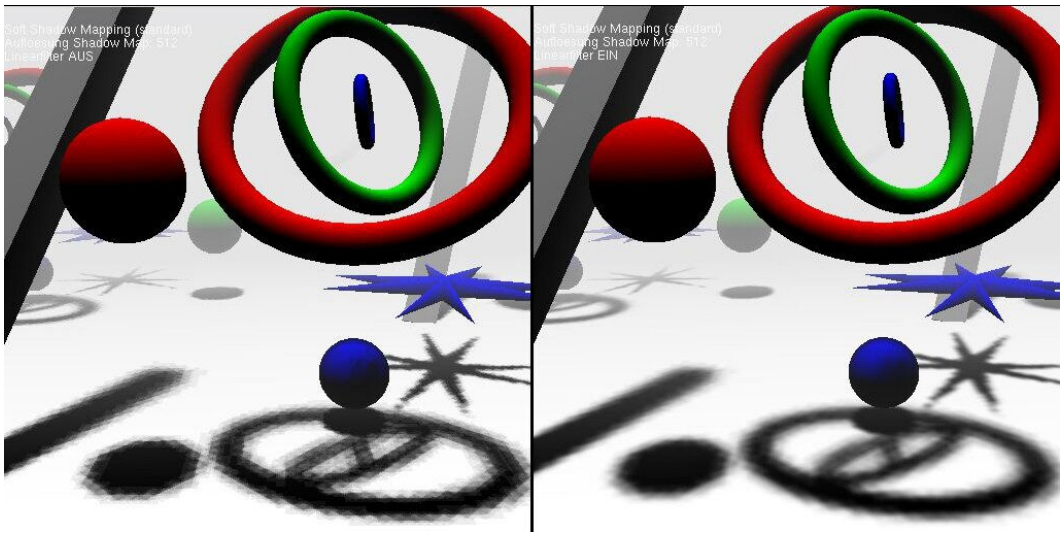


Bild 21 – Soft Shadows mit Standard Shadow Mapping (16 Samples)

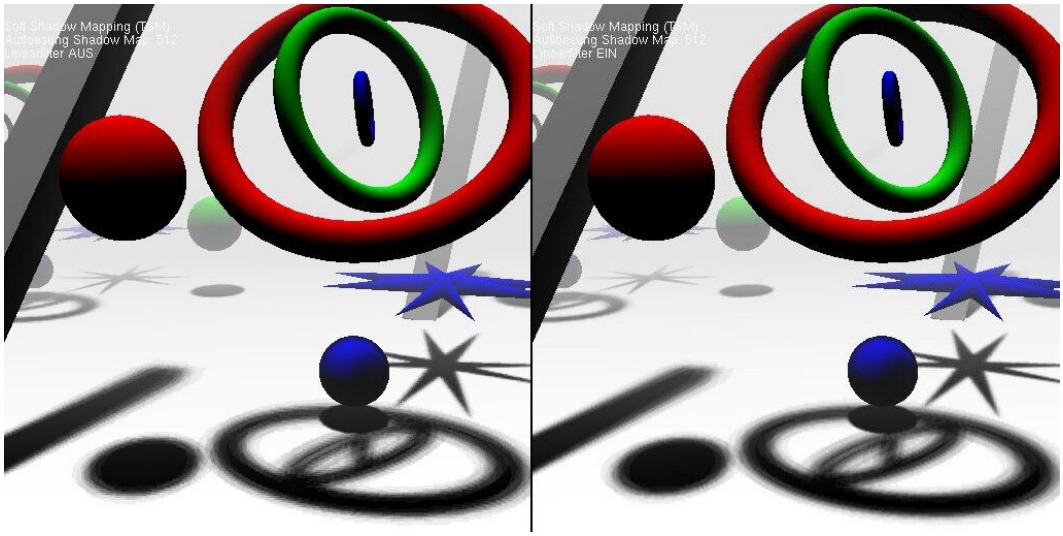


Bild 22 – Soft Shadows mit Trapezoidal Shadow Mapping (16 Samples)

Die Shadow Map Auflösung betrug 512x512 Pixel. Beim Standard Shadow Mapping sieht man ohne Linearfilter deutliche Artefakte, aber auch mit Filter sieht der Schatten nicht sehr realistisch aus. Beim Trapezoidal Shadow Mapping hingegen sehen die Soft Shadows auch ohne Linearfilter deutlich realistischer aus.

6.2 Vorteile des Trapezoidal Shadow Mappings

Wie man im vorherigen Kapitel deutlich erkennen konnte, ist das Trapezoidal Shadow Mapping in Punkto Schattenqualität dem Standard Shadow Mapping weit überlegen. Allerdings ist auch dieses Schattenverfahren nicht für alle Situationen gleich gut geeignet.

Die besten Ergebnisse gegenüber dem Standard Shadow Mapping liefert das Trapezoidal Shadow Mapping, wenn:

- die Szene relativ groß ist und eine oder mehreren globalen Lichtquellen besitzt (Sonnenlicht),
- das Kamera View Frustrum aus Sicht der Lichtquelle relativ klein ist,
- die Lichtquelle senkrecht zur Szene steht,
- sich die Kamera immer in Bodennähe befindet.

In diesen Situationen kann das Standard Shadow Mapping problemlos durch das Trapezoidal Shadow Mapping ersetzt werden, zumal dieses nahezu gleich schnell arbeitet und deutlich bessere Ergebnisse liefert.

Aus diesen Situationen heraus würde sich das Verfahren am besten für die Schattierung einer Szene aus der Ich-Perspektive eignen. Auch große Außenareale mit simuliertem Sonnenlicht können damit qualitativ hochwertig schattiert werden.

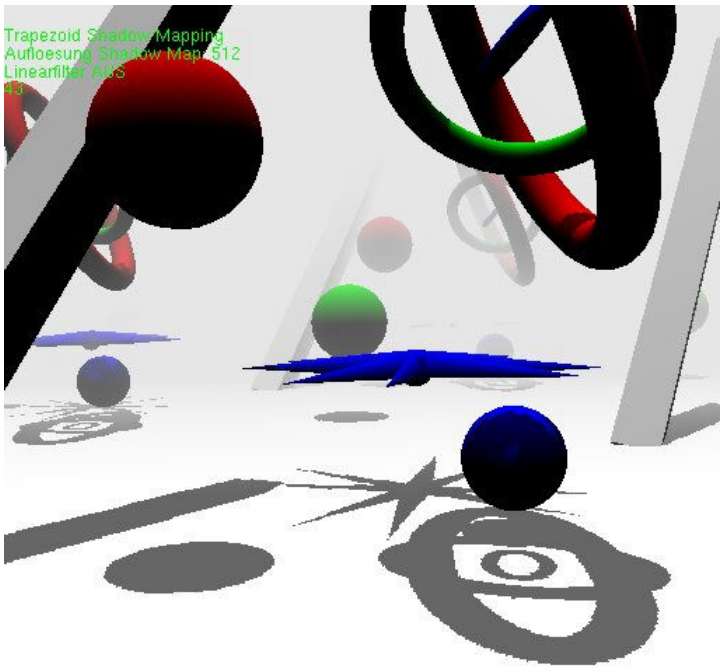


Bild 23 – Gute Schattenqualität beim Trapezoidal Shadow Mapping

6.3 Nachteile des Trapezoidal Shadow Mappings

Leider erreicht das Trapezoidal Shadow Mapping nicht in jeder Situation gleich gute Ergebnisse.

Relativ schlechte Ergebnisse, die nicht deutlich besser als die des Standard Shadow Mappings sind stellen sich ein, wenn:

- die Szene aus Sicht der Lichtquelle relativ klein im Vergleich zum Kamera View Frustrum ist,

- die Kamera in oder gegen die Blickrichtung der Lichtquelle sieht,

- die Kamera über dem Boden schwebt oder von schräg oben auf die Szene blickt.

Auf Grund dieser Einschränkungen ist das Trapezoidal Shadow Mapping nicht gut für Szenen geeignet, die in Draufsicht oder Flugansicht schattiert werden müssen. Dieser Nachteil entsteht durch die Focus Region. Hierbei wird die Szene direkt vor der Kamera mit einer sehr hohen Auflösung schattiert. Bei einer Draufsicht ist dieser Bereich allerdings leer, dass heißt der Großteil der Shadow Map Auflösung geht verloren. Hier kann durch eine große Focus Region die Qualität jedoch noch verbessert werden und ein Ansatz zur dynamischen Bestimmung der Focus Region könnte die Qualität in diesen Situationen verbessern. Das Trapezoidal Shadow Mapping liefert jedoch in keiner Situation Ergebnisse, die Qualitativ unter denen des Standard Shadow Mappings liegen.

Ein Vergleich der Bilder 21 und 22 macht die Qualitätsunterschiede bei gleicher Shadow Map Auflösung und unterschiedlichen Situationen deutlich.

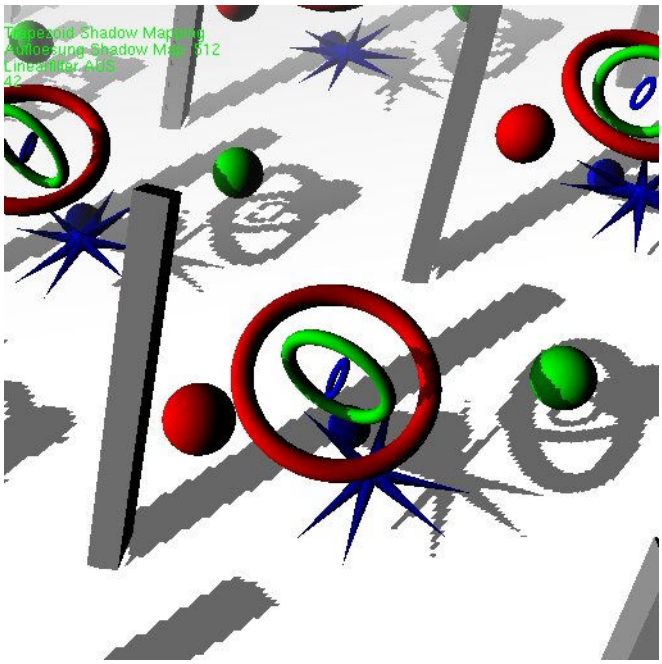


Bild 24 – Schlechte Ergebnisse beim Trapezoidal Shadow Mapping

7.0 Fazit

In dieser Studienarbeit wurden die Unterschiede des Standard Shadow Mappings und des Trapezoidal Shadow Mappings verdeutlicht und detailliert auf diese Verfahren eingegangen.

Wir haben gesehen, dass das Trapezoidal Shadow Mapping sich als vollwertiges Schattenverfahren herausgestellt hat, welches das Standard Shadow Mapping problemlos komplett ersetzten kann. In vielen Situationen ergeben sich qualitativ deutlich bessere Ergebnisse als beim Standard Shadow Mapping. Aber auch in Situationen, für die das Trapezoidal Shadow Mapping nicht so gut geeignet ist, sinkt die Schattenqualität nie unter die des Standard Shadow Mappings.

Die Geschwindigkeit des Trapezoidal Shadow Mappings ist zudem nahezu gleich schnell als die des Standard Shadow Mappings. Der Geschwindigkeitsverlust ist dabei so gering, dass dieser meist vernachlässigt werden kann.

Das Trapezoidal Shadow Mapping kann in vielen Situationen auch dort eingesetzt werden, wo bisher nur Shadow Volumes in Echtzeit gute Schattenergebnisse lieferten. Durch den Einsatz der Trapezoidal Shadow Maps können zudem die Objekte und die Szene deutlich komplexer aufgebaut sein, da beim Shadow Mapping keine Beschränkung der Objektkomplexität nötig ist.

Allerdings hat auch das Trapezoidal Shadow Mapping noch mit Nachteilen zu kämpfen, zu denen Verbesserungen nötig wären. So könnte statt der statischen Festlegung der Focus Region eine dynamische Anpassung die Schattenqualität in vielen Situationen deutlich verbessern. Dadurch könnte das Trapezoidal Shadow Mapping auch in Situationen besser eingesetzt werden, in denen es bisher nicht so gute Ergebnisse lieferte, wie z.B. der Draufsicht.

Auch der Lösungsansatz, welcher die Far Plane als Trapezpunkte nimmt, wenn die Kamera in oder gegen die Lichtrichtung sieht ist noch nicht optimal. Gerade bei einer großen Sichtweite kommt es hier zu starkem Aliasing. Auch hier wäre ein besserer Ansatz empfehlenswert.

Abschließend ist zu sagen, dass das Trapezoidal Shadow Mapping trotz der noch bestehenden Nachteile eines der besten Shadow Mapping Verfahren ist, die es zurzeit im Bereich des Echtzeitrenderings gibt.

Anhang A Bildverzeichnis

Anhang B Listings

Anhang C Quellenverzeichnis

[Q01] „Schatten“, Marc Stamminger 2003
http://www9.informatik.uni-erlangen.de/Teaching/SS2004/InCG/incg06_6up.pdf

[Q02] „Projizierte Schatten in OpenGL“, Stephan Drab 2003
http://webster.fh-hagenberg.at/statt/haller/cgr2_20032004/tutorial12/12-shadow-presentation.pdf

[Q03] „Practical and Robust Shadow Volumes“ Nvidia 2002
http://developer.nvidia.com/object/robust_shadow_volumes.html

[Q04] “Shadow Mapping: Casting curved shadows on curved surfaces”, Paul Baker
http://paulsprojects.net/tutorials/smt/smt.html

[Q05] “Anti-aliasing and Continuity with Trapezoidal Shadow Maps”, Tobias Martin und Tiow-Seng Tan 2004
http://www.comp.nus.edu.sg/~tants/tsm.pdf

[Q06] “Trapezoidal Shadow Maps (TSM) – Recipe”, National University of Singapore 2004
http://www.comp.nus.edu.sg/~tants/tsm/TSM_recipe.html

[Q07] “Fundamentals of Texture Mapping and Image Warping“, Paul Heckbert's Master Thesis

[Q08] “Perspective Shadow Maps”, Marc Stamminger und George Drettakis 2002
http://www-sop.inria.fr/reves/publications/data/2002/SD02

[Q09] “Adaptive Shadow Maps”, Randima Fernando, Sebastian Fernandez, Kavita Bala, Donald P. Greenberg 2001
http://citeseer.ist.psu.edu/fernando01adaptive.html

[Q10] “GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics”, Randima Fernando 2004