

Robuste Echtzeitschatten für komplexe, dynamische Szenen

Diplomarbeit

vorgelegt von
Nico Hempe



U N I V E R S I T Ä T
K O B L E N Z · L A N D A U

Institut für Computervisualistik
Arbeitsgruppe Computergrafik

Betreuer: Dipl.-Inform. Johannes Behr, Yvonne Jung (Fraunhofer IGD, Darmstadt)
Prüfer: Prof. Dr.-Ing. Stefan Müller

Juni 2005



Aufgabenstellung für die Diplomarbeit
Herr Nico Hempe
(Matrikel-Nr. 200210086)

Thema: Robuste Echtzeitschatten für komplexe, dynamische Szenen

Im Rahmen dieser Arbeit soll ein Framework zur robusten Darstellung von Echtzeitschatten in komplexen, dynamischen Szenarien auf Basis eines bestehenden Szenengraphsystems entworfen und umgesetzt werden. Die Einbindung in die 3D-Welt sollte bzgl. der verwendbaren Verfahren parametrisierbar sein und dem Benutzer hinsichtlich der für immersive VR üblichen Problematiken wie Stereoprojektion oder Clustering transparent. Aufgrund der Größe typischer VR-Welten sollen adaptive Verfahren, wie zum Beispiel Perspective-Shadowmaps, ebenso Eingang finden wie echtzeitfähige Softshadowing-Algorithmen. Ziel dieser Arbeit ist es, nach einer ausgiebigen Analyse der bestehenden Verfahren, unterschiedliche geeignete Methoden zu entwickeln und als OpenSG-Komponente zu implementieren. Dabei soll ein Rahmensystem für Echtzeitschattenverfahren geschaffen werden, um diese vergleichbar und für unterschiedliche Anwendungsmodelle auswählbar zur Verfügung zu stellen.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Untersuchung bekannter Schattenverfahren
2. Einarbeitung in das Szenengraphsystem OpenSG
3. Entwurf eines Frameworks
4. Implementierung mehrerer Schattenverfahren für weiche und harte Schatten
5. Testen der Schattenalgorithmen mit verschiedenen Szenen
6. Dokumentation der Ergebnisse

Koblenz, den 2.6.2005

Betreuer: Dipl.-Inform. Johannes Behr, Yvonne Jung (Fraunhofer IGD, Darmstadt)
Dipl.-Inform. Thorsten Grosch (Universität Koblenz-Landau)

– Prof. Dr. Stefan Müller –

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel Verwendung fanden. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

....., den
Ort, Datum

.....
Unterschrift

Danksagung

Ich möchte mich hiermit bei der Abteilung für Virtuelle und Erweiterte Realität des Fraunhofer-Instituts für Graphische Datenverarbeitung in Darmstadt bedanken, in der ich diese Arbeit angefertigt habe. Mein besonderer Dank geht dabei an Yvonne Jung und Johannes Behr, die mich dort betreut haben und mir nicht nur bei Fragen jederzeit zur Verfügung standen, sondern mich auch beim Entwurf des Shadow-Viewports sowie dessen Implementierung tatkräftig unterstützten.

Des Weiteren danke ich Thorsten Grosch, der mich auf Seiten der Universität betreute und mir immer gern mit Tipps zur Seite stand.

Ein letzter Danke geht an alle, die mich beim Korrigieren dieser Arbeit unterstützt haben.

Inhaltsverzeichnis

1.0 EINLEITUNG	7
1.1 MOTIVATION	7
1.2 SCHWERPUNKT	7
1.3 ÜBERSICHT	8
2.0 OPENSF	9
2.1 WAS IST OPENSF	9
2.2 GRUNDLAGEN	10
2.3 LICHTQUELLEN	12
2.4 FIELD CONTAINER UND MULTITHREADING	14
2.5 CLUSTERING	18
2.6 VIEWPORTS UND KAMERAS	20
3.0 SCHATTEN	22
3.1 WOZU SCHATTEN?	22
3.2 SCHATTENVERFAHREN IM ÜBERBLICK	24
3.2.1 LIGHT MAPS	24
3.2.2 SCHATTENFLECKEN	25
3.2.3 PLANARE SCHATTEN	26
3.2.4 SHADOW VOLUMES	27
3.2.5 SHADOW MAPPING	28
4.0 SHADOW MAPPING IM DETAIL	30
4.1 FUNKTIONSWEISE	30
4.2 VOR- UND NACHTEILE	33
4.3 VERBESSERUNGEN	35
5.0 PERSPECTIVE SHADOW MAPPING	36
5.1 IDEE	36
5.2 FUNKTIONSWEISE	37
5.3 LIGHT SPACE PERSPECTIVE SHADOW MAPPING	42
5.4 ZUSAMMENFASSUNG	44
6.0 TRAPEZ SHADOW MAPPING	45
6.1 IDEE	45
6.2 FUNKTIONSWEISE	47
6.3 SPEZIALFÄLLE UND VERBESSERUNGEN	55
6.3.1 PROBLEM BEI DER TRAPEZBERECHNUNG	55
6.3.2 AUTOMATISCHE ANPASSUNG DER BASE- UND TOP-LINE	56
6.3.3 KONSTANTERE SCHATTENQUALITÄT	57
6.3.4 POLYGON OFFSET PROBLEM	59
6.4 ZUSAMMENFASSUNG	59

7.0 PCF SHADOW MAPPING	60
7.1 IDEE	61
7.2 FUNKTIONSWEISE	62
7.3 VERBESSERUNGEN	63
7.3.1 <i>PERFORMANCEERHÖHUNG</i>	63
7.3.2 <i>QUALITÄTSVERBESSERUNG</i>	64
7.4 ZUSAMMENFASSUNG	66
8.0 PERSPEKTIVISCHES PCF SHADOW MAPPING	67
8.1 IDEE	67
8.2 FUNKTIONSWEISE	68
8.2.1 <i>BLOCKERSUCHE</i>	68
8.2.2 <i>BESTIMMUNG DER PENUMBRA</i>	69
8.2.3 <i>VARIABLER PERCENTAGE CLOSER FILTER</i>	70
8.3 QUALITÄTSVERBESSERUNG	71
8.4 ZUSAMMENFASSUNG	71
9.0 IMPLEMENTIERUNG	72
9.1 SHADER IN OPENSGL	72
9.1.1 <i>PROBLEME UND LÖSUNGEN</i>	72
9.2 DER SHADOW VIEWPORT	74
9.2.1 <i>DAS MODUL STD_SHADOWMAP</i>	80
9.2.2 <i>DAS MODUL TRAPEZ_SHADOWMAP</i>	82
9.2.3 <i>DAS MODUL PERSPECTIVE_SHADOWMAP</i>	84
9.2.4 <i>DAS MODUL PCF_SHADOWMAP</i>	86
9.2.5 <i>DAS MODUL PERSPECTIVE_PCF_SHADOWMAP</i>	87
9.3 BENUTZUNG DES SHADOW VIEWPORTS	88
10.0 ERGEBNISSE UND VERGLEICH	90
10.1 SCHATTENQUALITÄT HARTE SCHATTEN	90
10.1.1 <i>VERSCHIEDENE AUFLÖSUNGEN DER SHADOW MAP</i>	91
10.1.2 <i>NAH-, MITTEL- UND FERNBEREICH</i>	95
10.1.3 <i>VERSCHIEDENE BLICKWINKEL</i>	102
10.1.4 <i>ERGEBNISSE DER VORGESTELLTEN VERBESSERUNGEN</i>	106
10.1.5 <i>DISKUSSION</i>	110
10.2 SCHATTENQUALITÄT WEICHE SCHATTEN	112
10.2.1 <i>DISKUSSION</i>	118
11 FAZIT	119
<u>ANHANG A</u> <u>GLSL QUELLCODE</u>	<u>121</u>
<u>ANHANG B</u> <u>QUELLENVERZEICHNIS</u>	<u>127</u>

1.0 Einleitung

1.1 Motivation

Bereits in meiner Studienarbeit habe ich mich mit der Thematik der Echtzeitschatten auseinandergesetzt. Schon hier hat mich die Tatsache fasziniert, wie maßgeblich Schatten daran beteiligt sind, eine Szene realistisch wirken zu lassen. Szenen, die heute optisch nah an der Wirklichkeit liegen wollen, sind ohne entsprechende Schatten nicht mehr vorstellbar. Bis vor einiger Zeit mussten simple „Schattenflecken“ für diese Zwecke, gerade im Bereich der Echtzeit, ausreichen. Heute ist mit gesteigerter Rechenleistung auch die Berechnung und Darstellung komplexerer Schatten in Echtzeit möglich und in reell wirkenden Szenen auch gefordert. Die Technik der „Shadow Maps“ macht diese Form wirklichkeitsnaher Schatten in Echtzeit möglich.

Im Verlauf dieser Arbeit werde ich verschiedene Verfahren des Shadow Mappings genauer untersuchen und auch auf aktuelle Beispiele in der Praxis eingehen, z.B. anhand aktueller Computerspiele, in denen bereits neuste Methoden des Shadow Mappings eingesetzt werden.

1.2 Schwerpunkt

Der Schwerpunkt dieser Arbeit liegt in der Implementation eines Shadow-Viewports in das Open Source Szenengraphensystem „OpenSG“. Dabei soll dieser Viewport die Möglichkeit bieten, aus verschiedenen Schattenverfahren zu wählen und diese auf eine gegebene Szene anzuwenden. Darunter sollen Verfahren zur Erzeugung von harten als auch weichen Schatten sein. Der Shadow-Viewport soll am Ende so einfach zu bedienen sein wie möglich, es soll also im Idealfall nur einen Aufruf „Schatten an“ geben. Die restlichen für die Schattenberechnung notwendigen Berechnungen und Parameter sollen, soweit dies möglich ist, vom Shadow-Viewport selbst vorgenommen werden.

Allgemein habe ich mich in meiner Arbeit auf Verfahren auf Basis des „Shadow Mappings“ bezogen, da diese universell einsetzbar sind und auch für sehr komplexe Szenen optimale Ergebnisse liefern. Wie alle Shadow-Mapping-Verfahren arbeiten auch die hier vorgestellten weitgehend unabhängig von der Komplexität der Szene, wodurch die Möglichkeit besteht, reell wirkende Schatten auch für komplexe, dynamische Szenen in Echtzeit zu berechnen.

Auf Basis dieses Viewports sollen die Ergebnisse der verschiedenen implementierten Schattenverfahren vergleichbar sein. Am Ende werden diese gesammelt und verglichen, um deren Vor- und Nachteile zu diskutieren.

1.3 Übersicht

Zu Beginn gehe ich näher auf das Open Source Szenegraphensystem OpenSG ein, in welches der Shadow-Viewport implementiert werden soll. Dazu thematisiere ich nur Aspekte von OpenSG, die für das Verständnis des Shadow-Viewports nötig sind.

Im Anschluss setzte ich mich mit der Frage auseinander, warum Schatten überhaupt für die Realitätsnähe so wichtig sind und stelle verschiedene Verfahren zur Schattengenerierung anhand einiger Praxisbeispiele vor. Dabei wird auch kurz auf die Vor- und Nachteile dieser Verfahren eingegangen und diese an entsprechenden Praxisbeispielen illustriert.

Kapitel 4 geht im Folgenden genauer auf die Funktionsweise des Shadow Mappings ein. Hier werden auch verschiedene allgemeine Vor- und Nachteile dieses Verfahrens genannt.

Die darauf folgenden Kapitel beziehen sich auf die verschiedenen Schattenverfahren, die von mir in den Shadow-Viewport implementiert werden. Diese stelle ich einzeln vor und gehe dabei auf ihre Funktionsweise ein. Zusätzlich erläutere ich die Probleme dieser Verfahren und bringe Lösungsvorschläge an. Dabei stelle ich drei Verfahren für harte, sowie zwei Verfahren für weiche Schatten vor.

Kapitel 9 beschäftigt sich im Anschluss mit der Implementierung. Es wird der Aufbau des Shadow-Viewports vorgestellt und auf verschiedene Probleme bei der Implementierung eingegangen, bevor in Kapitel 10 ein Vergleich der erzielten Ergebnisse vorgenommen wird.

Kapitel 11 fasst zum Schluss die Erkenntnisse aus dieser Arbeit in einem Fazit zusammen.

2.0 OpenSG

2.1 Was ist OpenSG

Der Name „OpenSG“ steht für „Open Szenegraph“. Es handelt sich dabei, wie der Name bereits sagt, um ein Open Source Echtzeitrendering Szenegraphensystem, welches auf OpenGL aufsetzt. Der Unterschied besteht darin, dass OpenGL an sich keine Kenntnis über die zu rendernde Szene besitzt, sondern lediglich übergebene Dreiecke zeichnet. Hier setzt OpenSG mit dem Szenegraphensystem an. Es speichert die komplette Szene in einem Graphen und optimiert die Reihenfolge, in der die zu zeichnenden Dreiecke an OpenGL übergeben werden. Des Weiteren ermöglicht dies, Objekte oder ganze Teile der Szene, welche von der Kamera aus nicht zu sehen sind, vom Renderingprozess auszuschließen. Dadurch kann man einen großen Performancegewinn erreichen.

Ein weiterer Vorteil von OpenSG liegt in der Verwendung des Clusterings und Multithreadings. Dies ermöglicht es, ein und dasselbe Programm beispielsweise über ein Netzwerk auf verschiedenen Computern, auch mit unterschiedlicher Hardwareausstattung, berechnen zu lassen. Zudem ist OpenSG so aufgebaut, dass es sich leicht zu erweitern ist. Diese Erweiterungsmöglichkeit wird in der vorliegenden Arbeit genutzt. Im Verlauf dieses Kapitels stelle ich OpenSG nur oberflächlich vor. Es werden die für diese Arbeit nötige Aspekte erläutert, um die Funktion des Shadow-Viewports zu verstehen. Ich zeige nur wenige Codebeispiele und keine Anwendungen auf. Daher empfehle ich das OpenSG Tutorial [Q01], falls Fragen zu OpenSG offen bleiben.

Ich erläutere zunächst die Grundlagen eines Szenegraphensystems, bevor ich im Folgenden auf einzelne Komponenten von OpenSG eingehe. Für das Verständnis dieser Arbeit werden Lichtquellen und Field Container vorgestellt. Am Ende thematisiere ich den Aspekt des Multithreadings und Clusterings, um eines der Haupteinsatzgebiete von OpenSG zu zeigen.

2.2 Grundlagen

OpenSG speichert die komplette Szene in einem Graphen. Die meisten Szenegraphensysteme benutzen dieselbe Datenstruktur, um diesen aufzubauen. Es gibt Knoten, welche beispielsweise die Daten für die Geometrie, Transformation oder sonstige Informationen beinhalten. Grundsätzlich können Szenegraphen nach zwei Arten aufgebaut sein. Zum einen sind dies Single-Parent-, zum anderen Multi-Parent Systeme. Die Unterschiede der beiden Arten lassen sich am besten anhand eines Beispiels erläutern: Stellen wir uns ein einfaches Modell eines Autos vor, so besteht dieses aus einer Karosserie und vier Reifen. Alle Reifen besitzen die gleiche Geometrie, es sollte also reichen sie nur einmal im Speicher zu haben. Bei einem Single-Parent-System muss jedoch die Geometrie aller vier Reifen im Speicher vorliegen. Bei einem Multi-Parent-System reicht es, die Geometrie des Reifens einmal im Speicher liegen zu haben und nur vier unterschiedliche Transformationen der Reifen in Bezug auf die Karosserie zu verwenden. Jede Transformation verweist dann auf dieselbe Geometrie, die danach an die richtige Stelle gezeichnet wird. Bild 01 zeigt den grundsätzlichen Aufbau dieses Szenegraphen. OpenSG, sowie nahezu alle Szenegraphensysteme unterstützen Multi-Parent.

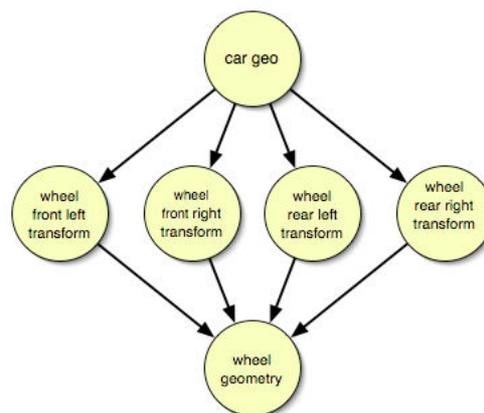


Bild 01 - Szenegraph eines einfachen Automodells [Q01]

In OpenSG werden Knoten nur dazu benutzt, die Hierarchie der Szene zu beschreiben. Sie besitzen nicht direkt Geometrien oder Transformationen, sondern lediglich Verweise auf ihre Kinder und einen Core, in dem die letztendlich wichtigen Informationen gespeichert sind.

Bild 02 verdeutlicht den Aufbau unseres Autobeispiels anhand des Systems, welches in OpenGL verwendet wird. Die Kreise sind dabei „normale“ Knoten, die Rechtecke Core-Knoten.

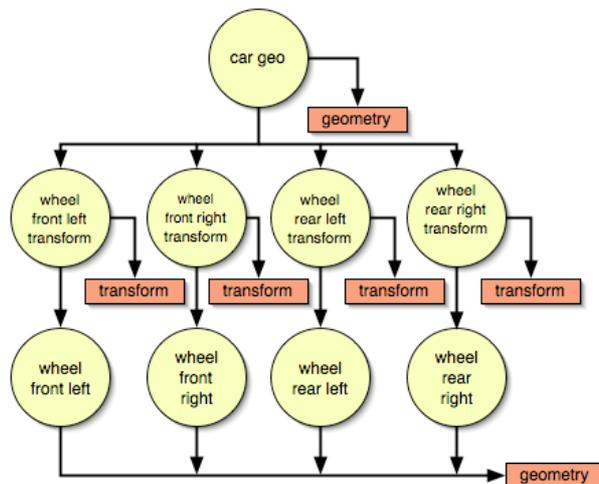


Bild 02 - Prinzip eines OpenGL-Szenegraphen [Q01]

Nur die Core-Knoten enthalten die eigentlichen Informationen. In OpenGL gibt es drei verschiedene Core-Knotentypen:

- Gruppenknoten
- Transformationsknoten
- Geometrieknoten

Gruppenknoten verweisen dabei auf eine neue Gruppe Knoten. Normalerweise kann dies jeder Knoten und es ist kein spezieller Core-Knoten notwendig, aber in OpenGL darf es keine Knoten ohne Core geben. Beim Traversieren des Graphen würde OpenGL diesen Prozess abbrechen, sobald es auf einen Knoten ohne Core trifft.

Transformationsknoten enthalten Informationen über Translationen, Rotationen oder Skalierungen, die auf alle Knoten unter diesem Transformationsknoten angewendet werden.

In Geometrieknoten ist schließlich, wie der Name bereits sagt, die eigentliche Geometrie gespeichert. Erreicht man beim Traversieren des Graphen einen Geometrieknoten, wird diese gezeichnet.

2.3 Lichtquellen

Lichtquellen erhellen eine Szene. Ohne Licht wäre alles schwarz, nur die Eigenemission der Objekte wäre zu sehen. Zusätzlich besitzen Lichtquellen die Eigenschaft, Schatten zu werfen, worum es in dieser Arbeit geht. Grundsätzlich unterscheidet man drei Arten von Lichtquellen:

- Direktionales Licht
- Spotlights
- Pointlights

Direktionales Licht kann man mit Sonnenlicht vergleichen. Dabei geht man davon aus, dass sich die Lichtquelle unendlich weit entfernt befindet, so dass die Lichtstrahlen praktisch in der gesamten Szene aus einer Richtung scheinen. Würde man die Szene aus Sicht dieser Lichtquelle rendern, käme dies einer orthographischen Sicht gleich. In OpenGL reicht zum Erstellen einer direktionalen Lichtquelle die Angabe einer Richtung, aus der das Licht kommen soll.

Spotlights sind vergleichbar mit einer Taschenlampe. Die Lichtquelle hat hierbei eine fest definierte Position, eine Lichtrichtung sowie einen Öffnungswinkel. Betrachtet man die Szene aus Sicht eines Spotlights, kommt dies einer normalen perspektivischen Sicht gleich. In OpenGL muss für ein Spotlight eine Lichtposition, eine Lichtrichtung und ein Öffnungswinkel angegeben werden.

Pointlights verdeutlicht man am besten anhand einer Glühbirne. Die Lichtquelle strahlt auch hier aus einer fest definierten Position, allerdings breitet sich das Licht von dieser Position kreisförmig in der ganzen Umgebung aus. Die Sicht eines Pointlights kann nicht so leicht dargestellt werden. Dadurch ist in vielen Shadow-Mapping-Verfahren die Schattierung der Szene anhand von Pointlights nicht ohne weiteres möglich. Um im Shadow-Viewport aber alle Arten von Lichtquellen behandeln zu können, bedient man sich hier eines kleinen Tricks. Das Pointlight wird hierzu in ein Spotlight umgewandelt. Den Öffnungswinkel wählt man so, dass aus der Position des Pointlights die komplette Szene zu sehen ist. Dies gewährleistet die Ausleuchtung bzw. Schattierung der gesamten Szene.

Auch Lichtquellen fügt man in OpenGL als Knoten in den Szenegraph ein. Eine Besonderheit ist, dass man die Angaben der Lichtquellen, also Position, Richtung, etc. nicht direkt vornimmt, sondern so genannte *Beacons* zum Einsatz kommen. Fügt man einen Lichtquellenknoten in einen Szenegraph ein, beeinflusst man damit zwei Dinge: Zum einen die Position und Richtung anhand der Transformationsknoten oberhalb des Lichtknotens, zum anderen aber auch den Einflussbereich des Lichts durch die folgenden Knoten. Es macht aber Sinn, diese beiden Angaben unabhängig voneinander zu kontrollieren. Die Position des eigentlichen Lichtknotens im Szenegraph beeinflusst den Bereich, den die Lichtquelle beleuchtet, also nur nachfolgende Knoten. Dadurch, dass eine Lichtquelle meistens viele Teile der Szene beeinflussen soll, stehen Lichtknoten oft an der Spitze eines Szenegraphen. Position und Richtung der Lichtquelle wird jedoch durch einen zweiten Knoten, dem Beacon, bestimmt.

Bild 03 zeigt, wie Lichtknoten allgemein in einen Szenegraphen eingebunden werden.

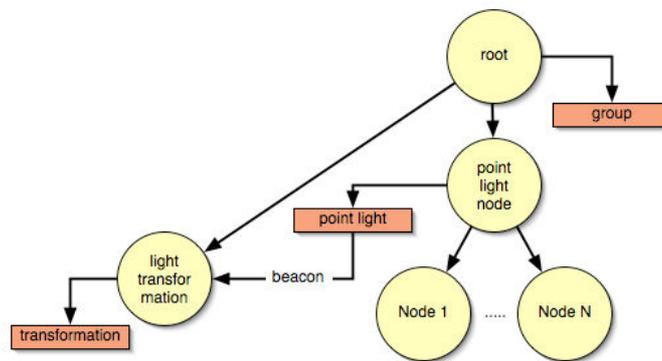


Bild 03 - Beacons im Szenegraphen [Q01]

2.4 Field Container und Multithreading

Unter Multithreading versteht man das Aufteilen des Programms auf mehrere unabhängige Prozesse. Multithreading bietet besonders auf Mehrprozessorsystemen oder modernen Prozessoren (z.B. Intel Pentium4 mit Multithreadingunterstützung sowie neue Mehrkernprozessoren) erhebliche Geschwindigkeitsvorteile.

Multithreading kann dazu genutzt werden, in einem Programm verschiedene Aufgaben unterschiedlichen Prozessen zuzuordnen. An einem bestimmten Punkt werden die Ergebnisse wieder zusammengefügt. Dabei spielt die Synchronisation der einzelnen Prozesse eine wesentliche Rolle. Ohne Synchronisation entscheidet das Betriebssystem, wann welcher Prozess gestartet oder angehalten wird. Durch diese Tatsache kann es passieren, dass ein Thread vom Betriebssystem noch lange ausgeführt wird, obwohl es bereits auf Daten eines anderen Prozesses wartet.

Eines der Probleme beim Multithreading stellt der gemeinsame Datenzugriff dar. Werden von einem Prozess Daten in einen Bereich geschrieben und versucht ein weiterer Prozess zur selben Zeit in diesen Bereich zu schreiben oder daraus zu lesen, führt dies meist zum Absturz des Programms. OpenSG löst das Problem durch den Einsatz von Field Containern, welche für die Speicherung nahezu aller Informationen genutzt werden. Die wesentliche Aufgabe eines Field Containers besteht darin, Daten vor gleichzeitigem Zugriff durch verschiedene Threads zu schützen. Nahezu alle Klassen in OpenSG leiten sich vom Typ Field Container ab. Jeder Field Container enthält zudem Daten über sich selbst, wie z.B. ein Name oder eine eindeutige Identifizierungsnummer, um auf ihn zugreifen zu können. Da in OpenSG diese Field Container benutzt, muss fast jedes Objekt per *Klasse::create()* erzeugen.

Field Container bestehen aus einem Single- oder Multi-Field. Single-Fields enthalten einzelne Werte, während Multi-Fields mit einem STL-Vektor vergleichbar sind, also eine dynamisch angepasste Größe haben. Multi Fields unterstützen auch den []-Operator, um auf bestimmte Elemente zugreifen zu können. Um Informationen in einen Field Container zu schreiben, wird der -> Operator benutzt, der eine Funktion des Objekts aufruft und meistens einen Wert übergibt. Zur Zuweisung eines Core-Knotens schreibt man beispielsweise:

```
beginEditCP(node1);
    node1->setCore(core);
endEditCP(node1);
```

node1 stellt einen Knoten und core einen Core-Knoten dar.

Beim Verändern von Field Containern muss man darauf achten, Änderungen vorher anzukündigen. Aus diesem Grund nimmt man Änderungen nur zwischen den beiden Anweisungen `beginEditCP(object)` und `endEditCP(object)` vor. Das Kürzel CP steht hierbei für Container Pointer. `beginEditCP()`- und `endEditCP()`-Anweisungen können auch gestaffelt werden, um mehrere Objekte gleichzeitig zu verändern. Werden EditCP's nur mit dem zu ändernden Objekt aufgerufen, geht OpenSG davon aus, dass alle Elemente des Objekts geändert werden sollen. Da dies besonders bei umfangreichen Objekten sehr zeitaufwendig ist, bieten diese Funktionen die Möglichkeit anzugeben, welche Änderungen durchgeführt werden sollen. Wollen wir beispielsweise an einem Knoten nur den Core-Knoten ändern, schreiben wir:

```
beginEditCP(node1, Node::CoreFieldMask);
node1->setCore(core);
endEditCP(node1, Node::CoreFieldMask);
```

Mehrere Änderungen trennt man durch ein Oderzeichen (|). Im Allgemeinen setzen sich die Namen für die Änderungen aus der zu ändernden Klasse gefolgt von `FieldMask` zusammen.

Die Ankündigung einer Änderung benötigt man für die Synchronisation beim Multithreading. Werden bei Änderungen editCP-Blöcke weggelassen, kann dies besonders bei Anwendungen, die auf mehreren Threads laufen, zum Absturz oder zu inkonsistenten Daten führen.

Standardmäßig wird beim Erstellen eines Field Containers nicht nur eine, sondern zwei Instanzen von ihm erzeugt. Die Instanzen heißen *Aspects*. OpenSG teilt jedem Thread einen eigenen Aspect zu. Möchte ein Thread Daten schreiben, wird nur der Aspect dieses Prozesses verändert (siehe Bild 04).

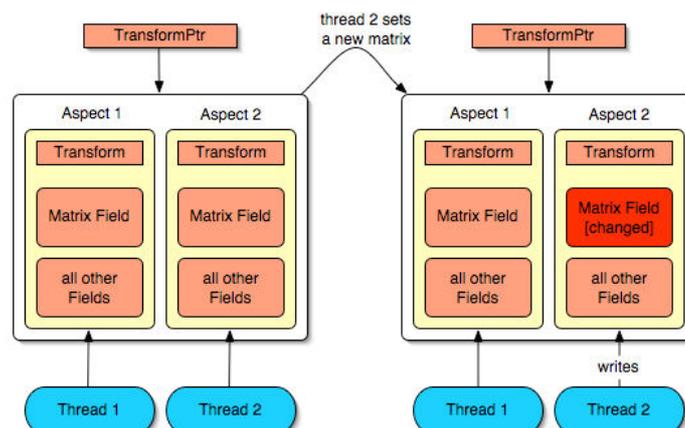


Bild 04 - Ein Aspect hat veränderte Daten [Q01]

Bei diesem Beispiel sieht man einen Field Container vom Typ Transform mit zwei Aspects, denen je ein eigener Thread zugeteilt ist. Schreibt Thread2 Daten, so wird nur der Thread2 zugeteilte Aspect verändert, der andere bleibt unberührt.

Wie unschwer zu erkennen ist, birgt diese Situation jedoch Probleme. Die Threads haben jetzt inkonsistente Daten, Thread 1 weiß nichts von der Veränderung der Daten von Thread 2. Zur Lösung verwendet man *Change Lists*. Jeder Thread besitzt eine eigene Change List. Werden von diesem Thread Daten geschrieben, wird seiner Change List ein Eintrag hinzugefügt. An einem bestimmten Programmpunkt müssen die Daten der Aspects synchronisiert werden. Dazu werden die Change Lists überprüft und bei Bedarf Daten aus einem Aspect in den anderen kopiert. Diese Synchronisation nimmt man manuell vor. Sie geschieht mit dem Aufruf

```
getChangeList()->applyAndClear()
```

Der Befehl bewirkt die Synchronisation der Threads durch Überprüfen der Change Lists und eventuellem Kopieren der veränderten Daten in alle Aspects (siehe Bild 05).

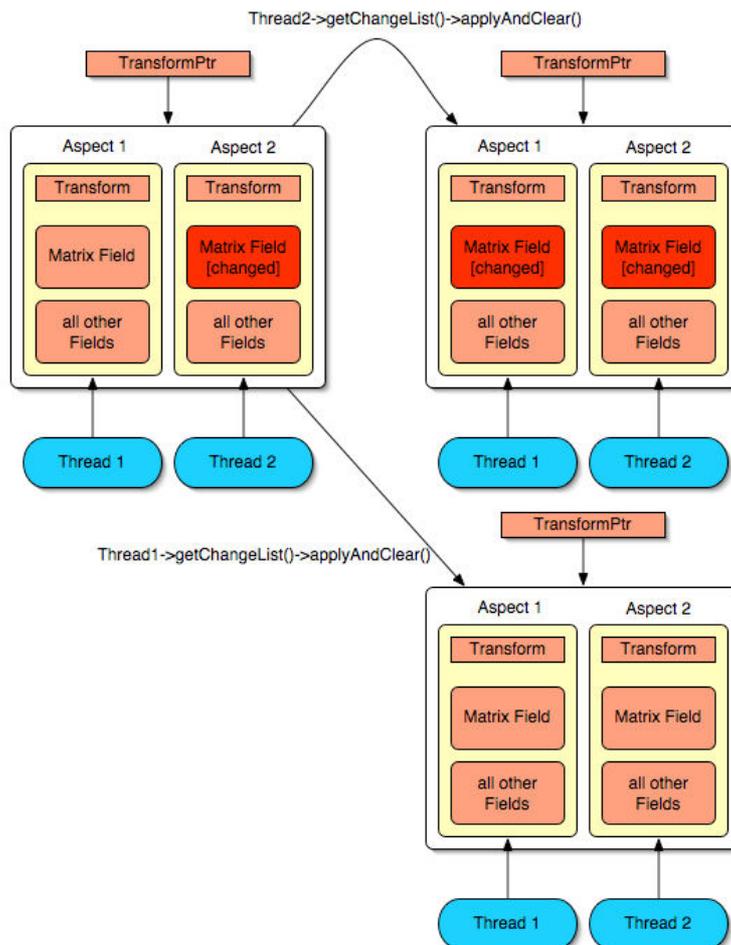


Bild 05 - Die Daten der Aspects werden synchronisiert [Q01]

Thread 2 berechnet in diesem Beispiel eine Matrix neu. Wird jetzt der Befehl *getChangeList()->applyAndClear()* von Thread 2 aufgerufen, kopiert OpenSG die veränderte Matrix in Aspect 1 von Thread 1. Danach haben beide Threads die veränderte Matrix und verfügen wieder über konsistente Daten. Erfolgt hingegen der Aufruf *getChangeList()->applyAndClear()* aus Thread 1, wird die von Thread 2 veränderte Matrix durch die unveränderte von Thread 1 überschrieben. Auch in dem Fall haben beide Threads wieder konsistente Daten, die Veränderung der Matrix durch Thread 2 ging jedoch verloren. Aus diesem Grund ist es immer wichtig zu wissen, welcher Thread wann welche Daten ändert. Nach dem Aufruf des Synchronisationsbefehls werden auch die Change Lists wieder gelöscht.

Dieses Vorgehen birgt jedoch eine Gefahr: Werden während der Synchronisation Daten geschrieben oder gelesen, kann dies wieder zu inkonsistenten Daten führen. Deshalb ist es wichtig, den Zugriff auf diese Daten während der Synchronisation zu blockieren. Hierfür dienen sogenannte Barriers. Barriers verfügen über eine Funktion namens *enter()*, der ein Integerwert übergeben wird. Trifft ein Thread auf diesen Aufruf im Programm, wird er solange angehalten, bis die übergebene Anzahl Threads diesen Punkt erreicht hat. In unserem Beispiel mit zwei Threads führt der Aufruf *barrier->enter(2)*; dazu, dass ein Thread hier wartet, bis der zweite diesen Punkt erreicht hat. Dies garantiert eine störungsfreie Synchronisation von veränderten Daten. Für eine Datensynchronisation hält man alle Threads durch einen doppelten *enter()*-Aufruf an. Nur der Thread, der veränderte Daten besitzt, synchronisiert diese hier durch den Aufruf *getChangeList()->applyAndClear()* mit den anderen Threads. Dieses Vorgehen gewährleistet, dass kein anderer Thread auf die Daten zugreift, während sie synchronisiert werden.

Die Verwendung von mehreren Threads macht nicht in jeder Situation Sinn. Eine Sinnvolle Verwendung für die Nutzung mehrere Threads ist, einen Thread beispielsweise für das darstellen einer Szene zu benutzen, während ein zweiter bald benötigte Objekte bereits in den Speicher lädt.

Einer der wichtigsten Gründe Multithreading zu verwenden, ist das Clustering, welches im folgenden Kapitel behandelt wird. Es stellt eine spezielle Art des Multithreading dar.

2.5 Clustering

Die Funktion des Clustering stellt eine herausragende Stärke dar, die auch dazu beitrug, dass OpenSG so weite Anwendung gefunden hat. Der Bedarf an Rechenleistung zum Rendern einer Szene kann je nach Qualität der Darstellung sehr hoch sein. Durch Clustering ist es möglich, den Berechnungsprozess auf mehrere Rechner zu verteilen und so nicht an die Rechenleistung eines einzelnen Rechners gebunden zu sein. Zu dem sind Supercomputer deutlich teurer als mehrer Einzelrechner, die beim Clustering gebündelt werden und nahezu gleiche Leistung bringen können. Die einzelnen Rechner koppelt man über ein Netzwerk zusammen.

Bild 06 zeigt beispielsweise einen einfachen Cluster für die Berechnung einer Stereoszene.

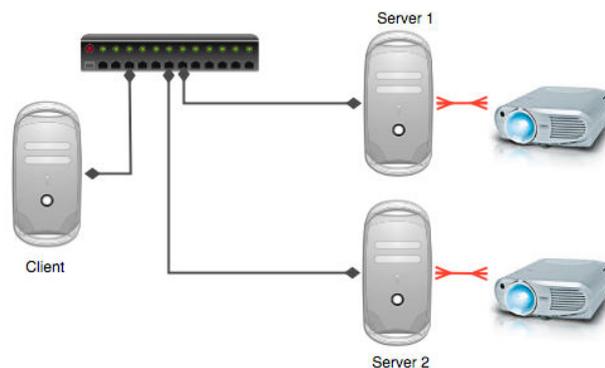


Bild 06 - Einfaches Clustering-Modell [Q01]

Auf dem Client läuft hier das Hauptprogramm. Es verändert zum Beispiel den Szenegraphen, lädt Dateien, verarbeitet Input und so weiter. Die beiden Server sind nur für das Rendern der Szene zuständig. Jeder Server rendert die Szene aus der Sicht, die für das jeweilige Auge bei einer Stereoprojektion nötig ist. Der Einsatz von Clustering eröffnet viele Möglichkeiten. So kann es dazu benutzt werden, ein großes Bild auf mehrere Ausgabegeräte aufzuteilen. Bild 07 zeigt die HeyeWall am Fraunhofer Institut in Darmstadt. Dies ist eine sechs-mal-vier Bildschirme große Projektionsfläche, die von einem Client und 48 Servern betrieben und zur Stereoprojektion benutzt werden kann.



Bild 07 - HeyeWall am Fraunhofer Institut Darmstadt [Q01]

Auf dem Client und auf den Servern laufen verschiedene Programme. Den Serverprogrammen teilt man einen bestimmten Name zu, der dem Client bekannt gegeben werden muss, um diese im lokalen Netzwerk zu finden.

Clustering verfährt im wesentlichen ähnlich wie Multithreading, mit dem Unterschied, dass OpenSG die Synchronisation anhand der *beginEditCP()*- und *endEditCP()*-Blöcke automatisch behandelt.

2.6 Viewports und Kameras

Die Bildschirmausgabe erfolgt in Fenstern. Jedes Fenster muss mindestens einen Viewport besitzen. Das Fenster kann aber auch auf mehrere Viewports aufgeteilt werden. Ein einfaches Beispiel für den Einsatz von mehreren Viewports in einem Fenster bieten viele Modellierungstools. Bei ihnen ist das Fenster oft in vier Viewports aufgeteilt, meistens eine Objektansicht und drei Sichten von oben, von vorn und von der Seite. Für diese Arbeit benötigen wir jedoch nur einen Viewport für das Fenster, da dieser lediglich Schatten in die Szene hinzufügen soll. Kurz gesagt füllt ein Viewport das Fenster mit Informationen, schreibt also in den ColorBuffer des ihm zugewiesenen Bereichs. Neben dem Standard-Viewport gibt es in OpenGL noch weitere Viewports für verschiedene Aufgaben, z.B. einen Stereo-Viewport für die Darstellung von 3D-Szenen für Shutterbrillen.

Damit der Viewport die Szene darstellt, benötigt er unter anderem einen Startknoten und eine Kamera. Als Startknoten fungiert meist der Root-Knoten des Szenegraphen. Er definiert, an welcher Stelle des Szenegraphen das Rendern begonnen wird.

Anhand der Kamera wird bestimmt, was letztendlich vom Szenegraphen gerendert wird und was weggelassen werden kann, weil es nicht zu sehen ist. Jeder Viewport kann auch auf mehrere Kameras zurückgreifen, muss aber mindestens eine Kamera besitzen. Kameras werden in OpenGL ähnlich wie Lichtquellen behandelt, das heißt auch hier kommen Beacons zum Einsatz, um die Kameraposition und Richtung zu bestimmen. Zur Bestimmung der Position und Ausrichtung einer Kamera benötigt man nur eine einzige Matrix, die sogenannte Modelviewmatrix. Da aber oft die Angaben über eine Kamera nicht in Matrizen gemacht werden, sondern mit Hilfe von zwei Punkten und einem Vektor, kann man die in OpenGL integrierte Hilfsfunktion *MatrixLookAt()* benutzen. Sie berechnet aus dem Kamerapunkt, dem LookAt-Punkt, und einem Up-Vektor die entsprechende Modelviewmatrix.

OpenGL stellt unter anderem zwei Arten von Kameras zur Verfügung:

- Perspektivische Kamera
- Matrixkamera

Zum Erstellen einer perspektivischen Kamera werden die Angaben über eine Near- und Far-Plane sowie eines Öffnungswinkels benötigt. Die Near-Plane gibt dabei an, ab welcher Entfernung zur Kameraposition die Szene gezeichnet werden soll. Die Far-Plane gibt an, bis zu welcher Sichtweite von der Kameraposition aus gerendert wird. Der Öffnungswinkel bestimmt, in welchem Winkel die Sehstrahlen die Kamera verlassen. Ähnlich wird auch in OpenGL vorgegangen, mit dem Unterschied, dass in OpenGL der Kamera zusätzlich ein Beacon zugewiesen werden muss.

Die Matrixkamera als spezielle Kameraart wird in dem hier vorgestellten Shadow Viewport auch oft benutzt. Sie wird allein durch zwei Matrizen bestimmt, eine Modelview- und eine Projektionsmatrix, ähnlich wie in OpenGL. Da sich die Matrizen für diese Kamera auch von Hand erstellen lassen, bietet sie deutlich mehr Möglichkeiten als die Perspektivische Kamera. Mit ihr kann beispielsweise eine orthographische Sicht simuliert werden, wie sie für das Shadow Mapping von direktionalem Licht benötigt wird.

Beide Kameraarten verfügen über weitere Möglichkeiten, spezielle Situationen zu simulieren. Hierfür können der Kamera sogenannte *Camera Decorators* zugefügt werden. So bietet OpenGL unter anderem einen `StereoCameraDecorator` an, der für eine 3D-Ausgabe Anwendung findet.

Im Shadow Viewport kommt ein weiterer Camera Decorator zum Einsatz, der *Tile Decorator*. Er bietet die Möglichkeit, einzelne Bereiche des Viewports zu rendern und wird benutzt, um Shadow Maps mit einer höheren Auflösung als der Bildschirmauflösung zu erstellen. Auf das genaue Vorgehen wird in Kapitel 9 näher eingegangen.

3.0 Schatten

3.1 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 08 verdeutlicht. Anwendungen, die heute grafisch nahe an der Realität liegen sollen, sind ohne die Darstellung von Schatten nicht vorstellbar.

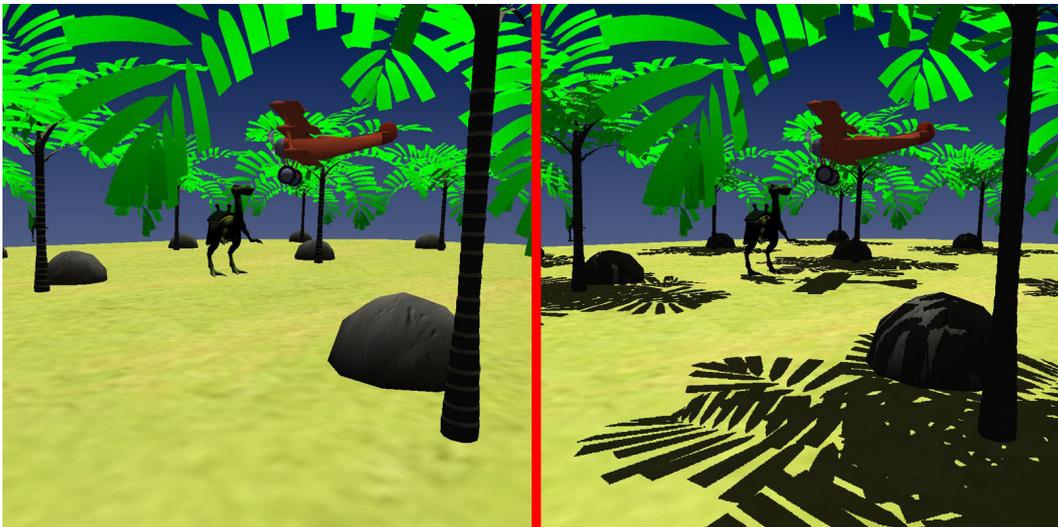


Bild 08 - Links: Szene ohne Schatten, Rechts: Szene mit Schatten

Noch einen Schritt hin zur Realitätsnähe führt der Einsatz von weichen Schatten. In der Realität wirft kaum ein Objekt nur harte Schatten, sondern die Schattenkanten verschwimmen, je weiter das Objekt von seinem Schatten entfernt ist. Nur bei flächenlosen Lichtquellen wären tatsächlich reine harte Schatten zu beobachten, diese sind in der Realität jedoch nicht gegeben. Durch die rasante Entwicklung der Hardware, ist die Darstellung realistischer weicher Schatten heute auch in Echtzeit möglich. Im Bild 2 sind die Unterschiede zwischen harten und weichen Schatten zu sehen.

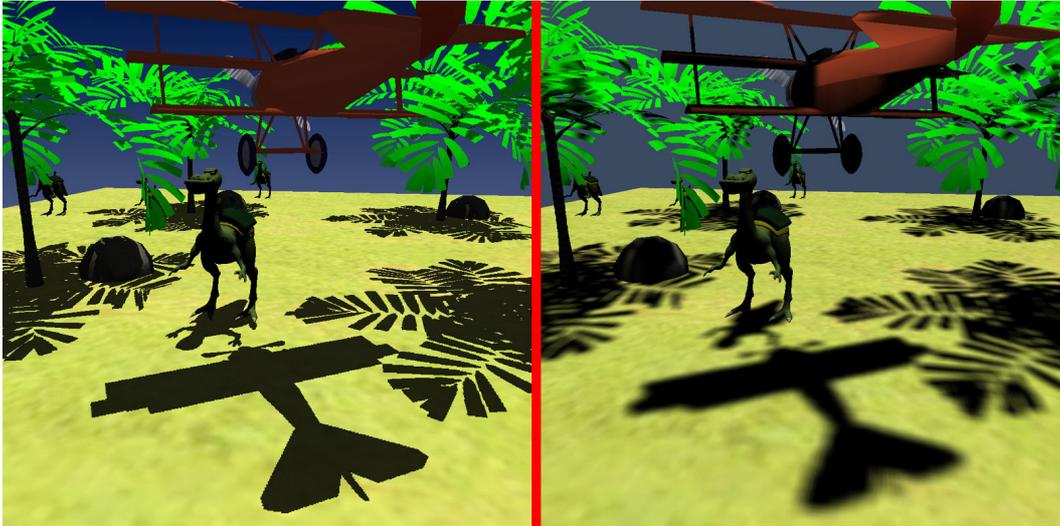


Bild09 – Links: Szene mit harten Schatten, Rechts: Szene mit weichen Schatten

3.2 Schattenverfahren im Überblick

Zur Berechnung von Schatten gibt es in OpenGL leider keinen Aufruf wie z.B. `glEnable(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 populäre Verfahren werden hier kurz in einer Übersicht erläutert.

3.2.1 Light Maps

Light Maps eignen sich nur für statische Szenen, da hierbei die Schatten für eine Szene vorberechnet und anschließend direkt in den Texturen der Szene gespeichert werden. Aus diesem Grund ist das Verfahren für das Rendern dynamischer Szenen nicht geeignet. Die Qualität der Schatten hängt dabei von dem Schattenverfahren ab, mit dem diese berechnet wurden. Theoretisch lassen sich damit Schattenkanten erzeugen, die nur durch die Auflösung der Textur, in der sie gespeichert werden, beschränkt sind. Ein populäres Beispiel für den Einsatz von Lightmaps ist das PC-Spiel „World of Warcraft“ [B01]. Der Schattenwurf aller statischen Objekte in der Welt wurde hier vorberechnet und in den Texturen gespeichert. Dies spart die Rechenzeit zum Erstellen dieser Schatten. Wie in Bild 10 auch zu erkennen ist, haben die Schatten eckige Kanten. Dies weist darauf hin, dass die vorberechneten Schatten auf Basis des Shadow Mappings berechnet wurden. Auf die Vor- und Nachteile des Shadow Mappings wird in Kapitel 4 noch näher eingegangen.



Bild10 – Szene aus „World of Warcraft“ [B01]

3.2.2 Schattenflecken

Eine der wohl einfachsten Arten von Echtzeitschatten bilden die Schattenflecken. Hierbei besitzt jedes schattenwerfende Objekt einen vorberechneten Schattenfleck, der auf die Szene projiziert wird. Je nachdem wie groß das Objekt ist, wird ein größerer oder kleinerer Schattenfleck auf die Szene projiziert. Dies ist eines der einfachsten und schnellsten Verfahren zur Schattendarstellung, bietet jedoch nur wenig Realitätsnähe. Für die Schattendarstellung dynamischer Objekte in dem PC-Spiel „World of Warcraft“ [B01] wird auch diese Art von Schatten eingesetzt. Bild 11 zeigt unterschiedlich große Objekte mit unterschiedlich großen Schattenflecken.



Bild11 – Szene aus „World of Warcraft“ [B01]

3.2.3 Planare Schatten

Die Idee der Planaren Schatten ist, dass diese Schatten als 2D-Projektion auf einer Ebene dargestellt werden. Dazu zeichnet man zunächst die Ebene, auf die der Schatten geworfen werden soll. Dann setzt man das Objekt in die Szene, welches den Schatten erzeugen 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 haften an dem Verfahren auch starke Einschränkungen. So funktioniert diese Methode nur mit planaren Ebenen und auch eine Selbstverschattung der Objekte ist nicht möglich. Bild 12 zeigt ein Objekt, welches seinen Schatten auf eine planare Fläche wirft



Bild 12 – Planare Schatten

3.2.4 Shadow Volumes

Frank Crow beschrieb 1977 erstmals das Shadow-Volume-Verfahren.. Es 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 kann man jetzt schattierte Bereiche der Szenen ausmaskieren. Das Verfahren ermöglicht auch eine Selbstverschattung der Objekte. Die Shadow Volumes haben zudem den Vorteil, dass man mit ihnen sehr präzise Schatten zeichnen kann, die nicht wie die beim Shadow-Mapping-Verfahren Kanteneffekte hervorrufen. Auf die Problematik der Shadow Maps gehe ich im Kapitel 4 noch näher ein.

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.

Eines der bekanntesten aktuellen Beispiele für den Einsatz von Shadow Volumes bieten die Computerspiele „Doom 3“ und „Quake 4“ [B02], die auf der selben Grafikenengine basieren. Gerade Doom3 wurde wegen der spektakulären Schattenwürfe und deren intensiven Nutzung zur Erzeugung einer spannenden Atmosphäre bekannt. Ein Nachteil dieser Methode ist jedoch, wie bereits erläutert, die hohe Hardwareanforderung, die zum Erscheinungstermin von Doom3 sogar aktuelle Hardware bei der Darstellung der Schatten in höchster Qualität überforderte. Die Leistung des Verfahrens hängt sehr stark von der Komplexität der Szene ab, da durch viele Vertices die Suche nach Objektkanten aufwändiger wird. Im Gegenzug erzeugen diese Schatten in jeder Situation ein optimales Ergebnis. Der Vorteil des Shadow-Volume-Verfahrens ist im Bild 13 gut zu erkennen.



Bild 13 – Links: Szene aus „Doom3“ [B02], Rechts: Szene aus „Quake 4“ [B02]

3.2.5 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. Das bedeutet, dass man keinerlei Kenntnisse über die Geometrie der Szene oder Objekte benötigt.

Ein Beispiel für den Erfolg des Shadow Mappings stellt die Implementierung dieses Verfahrens in die Software *Renderman* der Firma Pixar dar. Zunächst wurden Kurzfilme wie „Luxor Jr.“ [B03] mit diesem Verfahren produziert, später fand der Algorithmus auch in bekannten Kinofilmen, wie z.B. *Toy Story* [B03] Anwendung.



Bild 14 – Links: Ausschnitt aus „Luxor Jr.“, Rechts: Ausschnitt aus „Toy Story“ [B03]

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 die Methode ein.

Ein Beispiel für den Einsatz von Standard Shadow Mapping liefert das PC-Spiel „Warhammer 40k“ [B04], wie es in Bild 15 zu sehen ist. Auffällig sind hier die kantigen Schatten, die aus einem Nachteil des Shadow-Mapping-Verfahrens entstehen, auf das in Kapitel 4 noch eingegangen wird. Für kleinere Objekte werden auch in diesem Spiel die bereits vorgestellten Schattenflecken genutzt, wie im Hintergrund zu erkennen ist.



Bild 15 – Szene aus „Warhammer 40k – Dawn of War“ [B04]

Aktuelle Anwendungen müssen heute jedoch nahezu einwandfreie Schatten liefern, wodurch es mittlerweile viele verbesserte Verfahren auf Basis des Shadow Mappings gibt. Ziel dieser Verfahren ist eine Maximierung der Schattenqualität und Minimierung des dazu nötigen Mehraufwands an Rechenleistung. Ein aktuelles Beispiel für den Einsatz eines verbesserten Shadow-Mapping-Verfahrens, einem sogenannten *perspektivischen Shadow-Mapping-Verfahren*, liefert das Computerspiel „Battlefield 2“ [B05].



Bild 16 – Szene aus „Battlefield 2“, Links: Gute Qualität, Rechts: Schlechte Qualität [B05]

Beide Bilder zeigen des Schatten des selben Panzers. Wie im oberen Bild zu erkennen ist, liefert es nahezu kantenfreie Schatten. Nur in speziellen Situationen treten wieder verstärkt Schattenkanten auf, wie man im unteren Bild erkennt. Auf dessen Ursache und das hier eingesetzte Verfahren wird später eingegangen.

Das Shadow-Mapping-Verfahren lässt sich auch für die Darstellung von weichen Schattenkanten nutzen, wie sie heute für realitätsnahe Schatten gefordert werden. In den Kapiteln 7 und 8 finden Sie hierzu Näheres.

4.0 Shadow Mapping im Detail

4.1 Funktionsweise

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 Spotlight oder direktionales Licht handeln. Das Standard Shadow Mapping funktioniert daher nur mit diesen Lichtquellenarten. Für Punktlichtquellen kommen spezielle Verfahren zum Einsatz.

Wie kann man jetzt anhand des Shadow-Mapping-Verfahrens 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, liegt dieser im Schatten. 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 rendert man die Szene aus Sicht der Lichtquelle und liest nur die Tiefeninformation der Szene aus, die man dann in einer 2D-Textur speichert. 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.

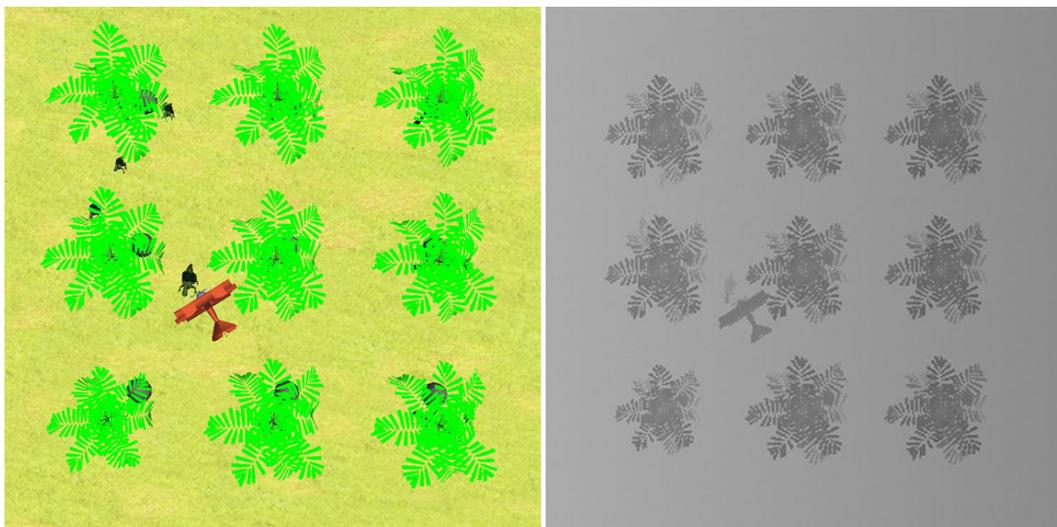


Bild 17 – Links: Szene aus Sicht der Lichtquelle, Rechts: Zugehörige Shadow Map

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:

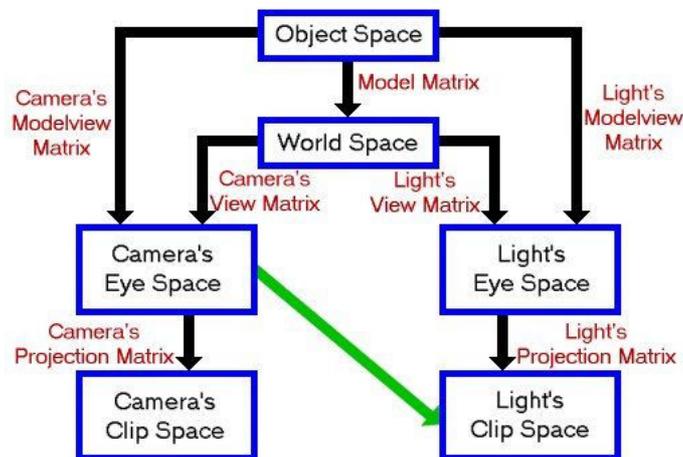


Bild 18 - Aufgabe der benötigten Projektionsmatrix

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. Die 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 18 den diagonalen Pfeil darstellt.

Die Texturmatrix kann man nach der folgenden Formel berechnen, wobei T die Texturmatrix, P_l und V_l die Projektions- bzw. Modelview-Matrix des Lichtes und V_c die Modelview Matrix der Kamera sind.

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

Im nächsten Schritt muss man die Texturkoordinaten noch anpassen, 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 Null und Eins 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.

4.2 Vor- und Nachteile

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 gut für das Echtzeitrendering. Des Weiteren arbeitet es, im Gegensatz zu den Shadow Volumes, vollkommen unabhängig von der Geometrie der Szene. Es benötigt für die Schattendarstellung keine Kenntnisse über deren Geometrie. Dadurch kann per Shadow Mapping 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 directionalem Licht oder Spotlights realisiert werden. Für den Einsatz von Punktlichtern sind spezielle Verfahren nötig. Spotlights werden definiert durch eine Position, eine Blickrichtung und einen Öffnungswinkel. Directionales Licht entsteht durch eine Lichtquelle, die theoretisch unendlich weit entfernt liegt und daher in der gesamten Szene im gleichen Winkel eintrifft, wie dies bei Sonnenlicht der Fall ist. 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 Ansätze, um diese Lichtquellen zu simulieren, indem sie diese in mehrere Punktlichtquellen aufteilen. Die Möglichkeit soll hier aber nicht weiter betrachtet werden. Wie Punktlichtquellen für das Shadow Mapping im Shadow Viewport angepasst werden wurde in Kapitel 2.3 beschrieben.

Eine weitere Herausforderung bei der Benutzung des Shadow Mappings stellt das so genannte Polygon-Offset-Problem dar. Dabei kommt es zu Darstellungsfehlern an Teilen der Objektoberflächen, die fälschlicherweise schattiert erscheinen. Der Fehler entsteht, da es sich beim Shadow Mapping um ein bildbasiertes Verfahren handelt und dadurch der für den Schattentest wichtige Z-Buffer nur mit einer endlichen Präzision berechnet wird. 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 bewegen sich die Objekte etwas von der Lichtquelle weg, was einem falschen Schattentest vorbeugt. Hierbei muss auf einen gut angepassten Wert geachtet werden. Ist der Wert zu klein, entstehen an manchen Stellen trotzdem Fehlerschatten. Ist der Wert zu groß, werfen Objekte keine Schatten in der nahen Umgebung (siehe Bild 19).

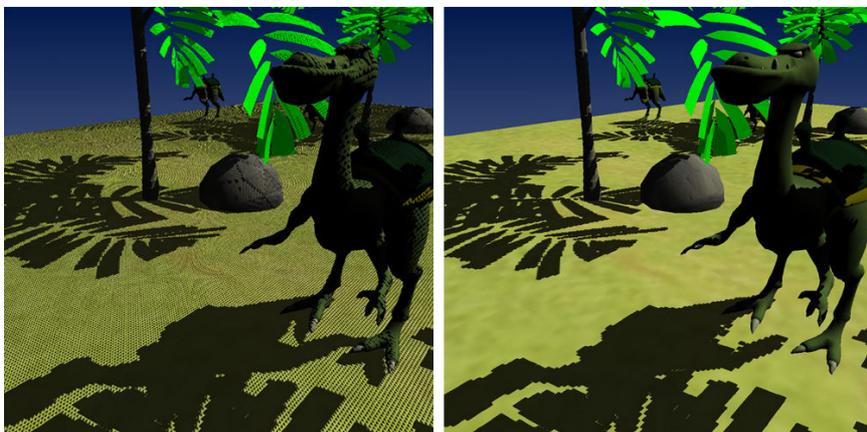


Bild 19 – Links: Offset zu klein, Rechts: Offset zu groß

Auf dem linken Bild ist das Bias zu klein gewählt- in der ganzen Szene treten fehlerhafte Schatten auf. Auch der Dinosaurier im Vordergrund weist falsche

Selbstverschattung auf. Auf dem Bild rechts dagegen wurde das Offset zu hoch gewählt, was ein Verschwinden von Schatten zur Folge hat. Achtet man auf die Füße des Dinosauriers fällt auf, dass dessen Zehen keine Schatten mehr werfen. Der Dinosaurier scheint zu schweben.

Ein anderer Lösungsansatz des Polygon Offset 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 das 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 20 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 20 – Aliasing: Artefakte bilden sich an den Schattenkanten

Dieses Problem lässt sich durch den Einsatz einer größeren Auflösung der Shadow Map entgegenwirken. Allerdings stößt dies bald an die Grenzen des Machbaren. Für sehr große Szenen, z.B. die Simulation von Sonnenlicht auf einer ganzen Szene, ist das Vorgehen nicht mehr praktikabel. Die Auflösung der Shadow Map müsste für das Erreichen einer guten Qualität so groß sein, wie sie keine Hardware unterstützt. Darum versucht man, Shadow-Mapping-Verfahren zu entwickeln, die mit gleicher Shadow-Map-Auflösung eine bessere Schattenqualität liefern, als das Standard-Shadow-Mapping. Auf diese Verfahren wird im Verlauf dieser Arbeit eingegangen.

4.3 Verbesserungen

Seit der Erfindung der Shadow Maps wurden zahlreiche Algorithmen erfunden, um die bekannten Probleme des Shadow Mappings zu reduzieren und die Darstellungsqualität zu verbessern. Dabei gehen viele Verbesserungsverfahren auf das Aliasingproblem des Shadow Mappings ein und versuchen es so weit wie möglich zu verhindern. Die einfachste Art den Aliasing-Effekt zu reduzieren wäre, die Auflösung der Shadow Map zu vergrößern. Das kann allerdings nur begrenzt geschehen, da auch moderne Grafikkarten nicht uneingeschränkt große Texturauflösungen unterstützen. Eine höhere Auflösung führt zudem zu einem deutlich spürbaren Geschwindigkeitsverlust durch den gestiegenen Mehraufwand.

Zur Reduzierung der Aliasing-Effekte sind im Laufe der Zeit einige Verbesserungen des Shadow Mappings vorgestellt worden, welche Qualitätsverbesserungen ohne die Erhöhung der Shadow-Map-Auflösung erreichen. Dabei gibt es Ansätze, die nicht in Echtzeit berechnet werden können und solche, die echtzeitfähig sind und mit teilweise nur geringem Mehraufwand eine deutlich bessere Schattenqualität liefern. Als Beispiel für ein Nicht-Echtzeitverfahren mit guten Ergebnissen sei das Adaptive Shadow Mapping [Q12] genannt. Die Auflösung der Shadow Map wird hier nach einer hierarchischen Baumstruktur in verschiedenen hoch aufgelöste Bereiche unterteilt. Bereiche an den Schattenkanten besitzen eine höhere Auflösung als Bereiche im Innern des Schattens.

Zu den echtzeitfähigen Verfahren können Shadow-Mapping-Verfahren gezählt werden, die auf Basis der Perspektivischen Shadow Maps [Q11] funktionieren. Bei diesen Verfahren wird vor der Erstellung der Shadow Map eine Perspektive anhand des Kamerafrustums in die Shadow Map einberechnet. Ziel ist eine höhere Auflösung in Bereichen, die nah an der Kamera liegen und eine Minimierung der ungenutzten Fläche auf der Shadow Map. Ungenutzte Bereiche entstehen dort, wo zwar die Lichtquelle hinscheint, die aber aus Kamerasicht nicht zu sehen sind. In den folgenden zwei Kapiteln gehe ich auf Verbesserungsverfahren auf dieser Basis ein. Ein weiterer Aspekt, auf den sich viele Verbesserungen des Shadow Mappings beziehen, ist die Darstellung von weichen Schatten, sogenannten Schlagschatten. Sie werden im Gegensatz zu harten Schatten von flächigen Lichtquellen erzeugt. In der Realität hat jede Lichtquelle auch eine Fläche, flächelose Lichtquellen kommen hier nicht vor. Demnach kann es in der Realität nur weiche Schatten geben. Die Möglichkeit weiche Schatten in eine Szene zu rendern hebt deren Realitätsnähe deutlich an. Die Umgebung wirkt natürlicher als mit harten Schatten.

Zwei Verfahren für die Darstellung von weichen Schattenkanten stelle ich in den Kapiteln 7 und 8 vor. Das erste Verfahren erzeugt uniforme Soft-Shadows, also Schatten, die an jeder Stelle am Schattenrand einen konstanten breiten Farbverlauf haben. Uniforme Soft-Shadows sehen zwar deutlich natürlicher aus als harte Schatten, sind aber noch nicht ganz authentisch. In der Realität hat ein Objekt, welches sich näher am Schattenempfänger befindet einen weniger verschwommenen Schattenrand als eines, welches weit von diesem entfernt ist. Die Ursache hierfür ist die Tatsache, dass Licht Welleneigenschaften besitzt und sich demnach auch wellenförmig ausbreitet.

Das zweite vorgestellte Verfahren beherrscht auch diese Aufgabe. Die damit erzeugten Schatten sehen sehr authentisch aus.

5.0 Perspective Shadow Mapping

Ich gebe zuerst eine kurze Einführung in das Prinzip des Perspective-Shadow-Mappings und gehe dann genauer auf das Light-Space-Perspective-Shadow-Mapping ein. Die Perspective-Shadow-Maps wurden erstmals von Marc Stamminger und George Drettakis im Jahr 2002 vorgestellt [Q11].

Die Light Space Perspective Shadow Maps wurden erstmals 2004 von Michael Wimmer, Daniel Scherzer und Werner Purgathofer präsentiert [Q19]. Sie enthalten einige Verbesserungen gegenüber den normalen Perspective-Shadow-Maps, auf die ich in diesem Kapitel ebenfalls eingehe.

5.1 Idee

Ziel dieser perspektivischen Verfahren ist die Reduzierung der Aliasing-Effekte durch eine perspektivische Transformation der Szene vor der Erzeugung der Shadow Map. Durch die Transformation sollen Bereiche, die sich nah an der Kamera befinden, vergrößert und solche, die weiter weg liegen, verkleinert werden. Dadurch steht Objekten nah an der Kamera eine größere Auflösung bei der Erzeugung der Shadow Map zur Verfügung als denen, die sich weiter weg befinden. Dies erreicht man durch die Berechnung der Schatten im post-perspektivischen Raum der Kamera. Bild 21 zeigt eine Szene vor der perspektivischen Transformation, sowie danach.

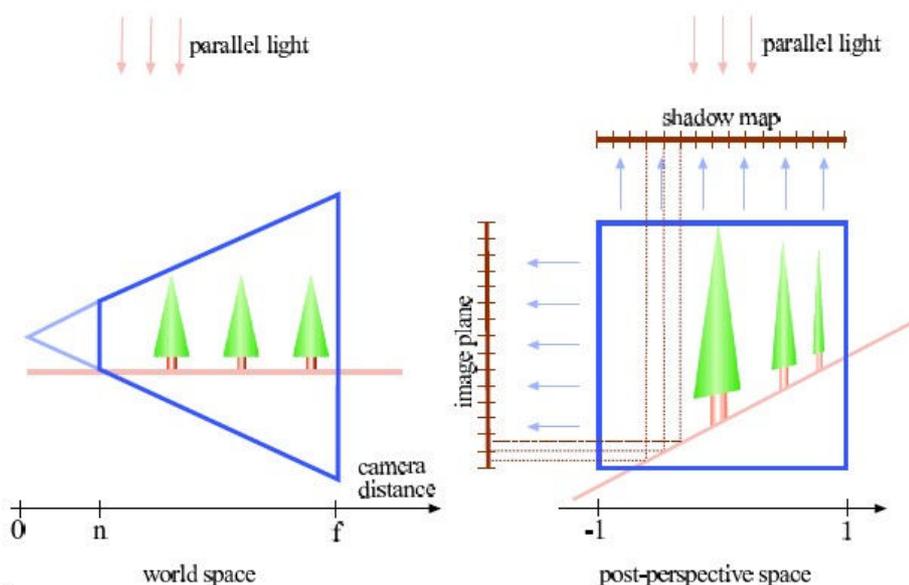


Bild 21 – Transformation in den post-perspektivischen Raum [Q10]

Bildlich kann man sich diese Transformation so vorstellen, dass der Pyramidenstumpf der Kamera an der Near-Plane so lange gestreckt wird, bis man einen Einheitswürfel erhält. Im folgenden Kapitel erkläre ich die einzelnen Schritte für die Berechnung der Perspective-Shadow-Maps. Daraufhin stelle ich Light-Space-Perspective-Shadow-Maps vor und erläutere dessen Unterschiede zum Perspektiv-Shadow-Mapping.

5.2 Funktionsweise

Die Berechnung der Schatten verläuft nach dem folgenden Muster:

1. Erzeugung einer „Hülle“, die alle wichtigen Bereiche einschließt
2. Erzeugung einer neuen virtuellen Kamera
3. Perspektivische Transformation der Szene
4. Erzeugung der Shadow Map
5. Schattentest und Darstellung

Im ersten Schritt werden Bereiche von der Schattenberechnung ausgeschlossen, die im endgültigen Bild nicht sichtbar sind. Dazu erzeugt man eine „Hülle“, die alle Objekte einschließt, welche für die Schattenberechnung wichtig sind. Auf diesen Schritt sowie auf die Erzeugung der virtuellen Kamera gehe ich gleich genauer ein. Zunächst betrachten wir einen anderen wichtigen Punkt. Durch die Transformation der Szene in den post-perspektivischen Raum wird nicht nur die Szene verändert, sondern auch die Lichtquellen. Hier kann sich deren Position und Art ändern. Bild 22a zeigt die möglichen Resultate von direktionalen, Bild 22b die von Punktlichtquellen nach der perspektivischen Transformation.

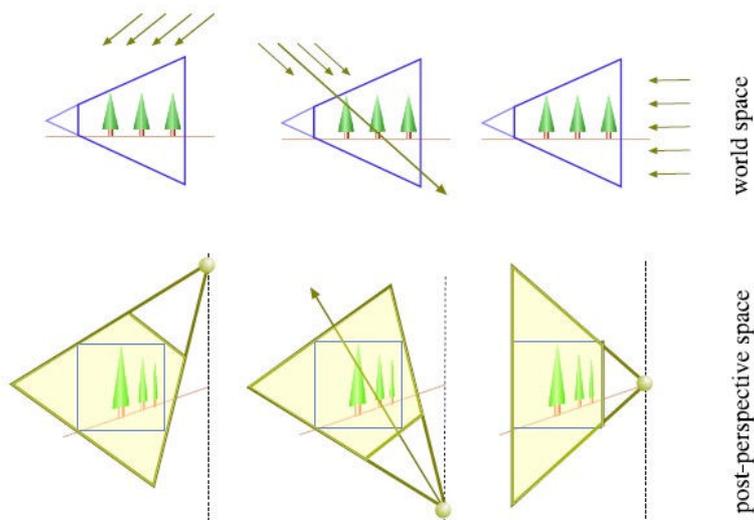


Bild 22a – Direktionale Lichtquellen im post-perspektivischen Raum [Q11]

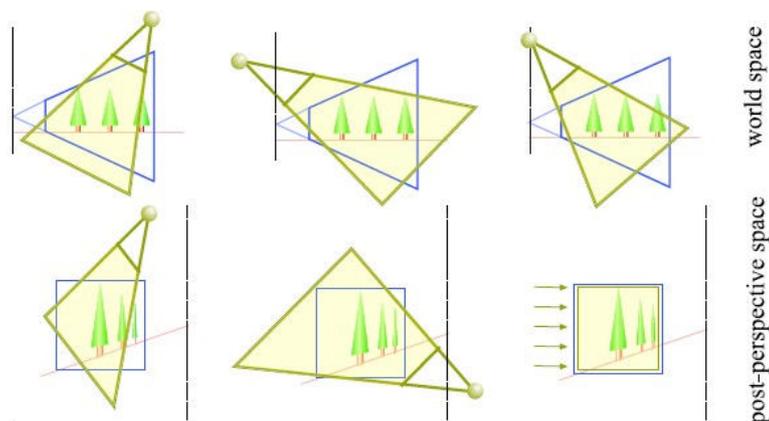


Bild 22b – Punktlichtquellen im post-perspektivischen Raum [Q11]

Man erkennt, dass aus direktionalem Licht Punktlichtquellen werden können und umgekehrt. Ausschlaggebend hierfür ist bei direktionalen Lichtquellen die Lichtrichtung und bei Punktlichtquellen die Lichtposition. Allgemein kann man feststellen, dass sich die Lichtrichtung nach der perspektivischen Transformation dann umkehrt, wenn diese in Richtung der Kamera verläuft. Bei Punktlichtquellen gibt es noch den Fall, dass die Punktlichtquelle zu einer direktionalen Lichtquelle wird, sobald sich deren Position auf einer Ebene mit der Kameraposition befindet.

Zurück zur Erzeugung der Hülle für die Schattenberechnung. Durch die Benutzung dieser Hülle sollen Bereiche, die zwar von der Lichtquelle, nicht aber von der Kamera zu sehen sind, entfernt werden. Nur über diesen Bereichen wird später die Shadow Map erzeugt. Beleuchtet die Lichtquelle beispielsweise eine große Szene, in der das Kamerafrustum relativ klein ist, wird beim Standard-Shadow-Mapping die Shadow Map über die gesamte Sichtweite der Lichtquelle erstellt. Das hat zur Folge, dass nur ein sehr kleiner Teil der Auflösung für die Schattentests gebraucht wird. Allein durch die Beschränkung auf diese relevanten Bereiche kann die Schattenqualität bereits maßgeblich verbessert werden.

Es liegt zunächst nahe, einfach das Kamerafrustum zur Beschreibung der Hülle zu verwenden, da sich alle Objekte, die in der Szene sichtbar sind, in diesem View-Frustum befinden müssen. Das ist zwar richtig, allerdings wurde hier ein Fall außer Acht gelassen. Auch Objekte außerhalb des View-Frustum können Schatten in die dargestellte Szene werfen. Bild 23 zeigt eine solche Situation.

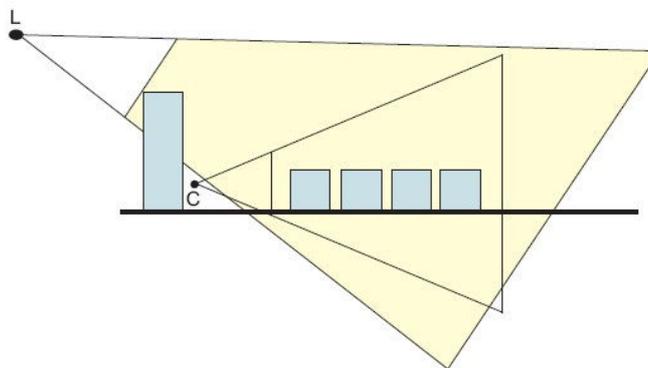


Bild 23 – Ein Objekt außerhalb des View-Frustums wirft Schatten in dieses

Wird in dem gezeigten Beispiel nur der Bereich des Kamerafrustums betrachtet, fehlt im letztendlichen Bild der Schatten des Objekts, welches diesen von außerhalb in das View-Frustum wirft. Darum muss der Bereich, über den die Shadow Map generiert wird, erweitert werden, um sämtliche Bereiche abzudecken, die Schatten in das View-Frustum werfen können. Bild 24 zeigt, wie man diesen Bereich ermittelt.

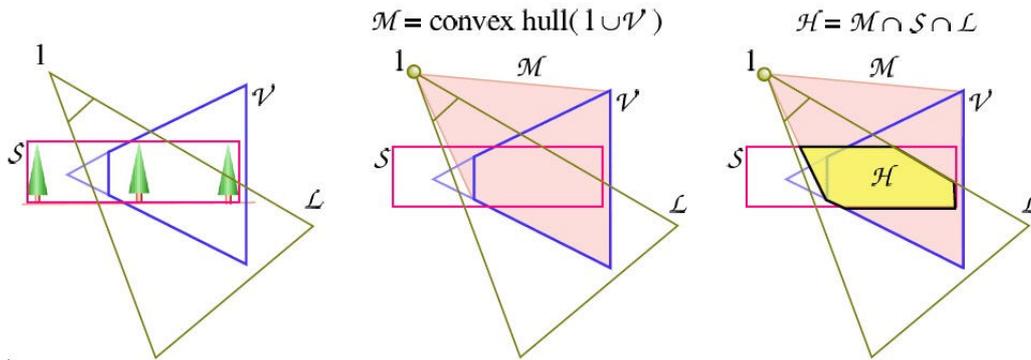


Bild 24 – Berechnung der konvexen Hülle [Q10]

Auf den Skizzen ist das View-Frustum des Lichts als I und das der Kamera als V dargestellt. S ist die Bounding Box der Szene, welche sämtliche Objekte einschließt. Als erstes wird die konvexe Hülle von V aus Sicht von I abgebildet, in der Skizze als M dargestellt. Bereits jetzt wären alle Objekte, die Schatten in die Szene werfen können, im fokussierten Bereich, allerdings kann dieser noch zu groß sein und Bereiche umfassen, in denen sich keine Objekte befinden können. Aus diesem Grund wird er noch mit der Bounding Box der Szene geschnitten. Der resultierende Bereich H beschreibt nun den Bereich, der für die Schattenberechnung interessant ist. Er wird von nun an als *Body* bezeichnet. Dieser *Body* kann jetzt in den post-perspektivischen Raum der Kamera transformiert werden, um die Shadow Map zu erzeugen.

Dabei kann jedoch ein weiteres Problem auftreten, welches mit den Eigenschaften des post-perspektivischen Raums zusammenhängt. Bild 25 zeigt eine solche Situation.

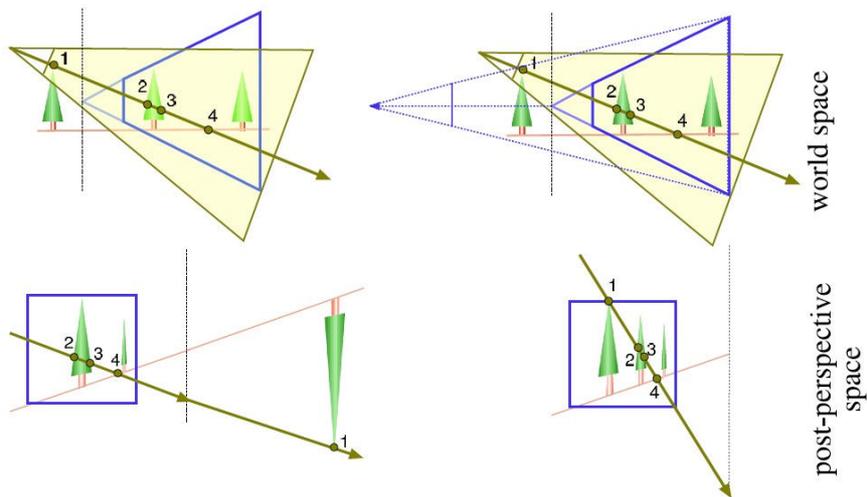


Bild 25 – Strahlen im post-perspektivischen Raum [Q10]

Links oben auf der Skizze ist zu erkennen, dass sich das Licht hinter der Kamera befindet und dessen View-Frustum bereits ein Objekt schneidet. Zur Verdeutlichung wurde eine Linie vom Lichtursprung über alle Objekte gezogen, welche diesen Strahl schneidet. Links unten ist das Ergebnis dieser Linie nach der Transformation in den post-perspektivischen Raum der Kamera aufgeführt. Man erkennt, dass die Linie zunächst nur die Objekte innerhalb des Kamera-View-Frustum schneidet und erst danach das Objekt, welches sich hinter der Kamera befand. Der Grund hierfür liegt in der Beschaffenheit des post-perspektivischen Raums. Die gepunktete Linie trennt die Szene in zwei Bereiche, einen, der vor dem Kameraursprung liegt und einen dahinter. Im post-perspektivischen Raum wird der Bereich vor der Kamera in den positiven Bereich abgebildet, wohingegen der dahinter in den negativen überführt wird. Hier stellt die Linie am Kameraursprung die Grenze von der positiven Unendlichkeit zur negativen dar. Die Linie schneidet also erst alle Objekte, die im positiven Bereich liegen, bevor sie nach dem Übertritt auf den Bereich hinter der Unendlichkeitslinie die Objekte in diesem Bereich trifft.

Um diese Situation zu vermeiden, wird vor der Erstellung der Shadow Map eine virtuelle Kamera erzeugt, bei der alle Objekte vor dem Kameraursprung liegen, wie es rechts oben in der Skizze gezeigt ist. Der Ursprung der neuen Kamera wird dabei nach hinten verlegt, so dass sie alle relevanten Objekte, also den Body, in dessen View-Frustum einschließt. Nun schneidet die gedachte Sichtlinie alle Objekte in der richtigen Reihenfolge, wie rechts unten in der Skizze zu sehen ist. Die Verlagerung des Kamerapunktes bezeichnet man als *virtual camera shift*.

Der virtual camera shift muss umso größer sein, je weiter die transformierte Lichtquelle hinter dem Kamerapunkt liegt, falls diese zu einem Point Light transformiert wurde. Dadurch verringert sich die Perspektive bzw. wird ganz aufgehoben, was dasselbe Ergebnis wie beim Standard-Shadow-Mapping zur Folge hätte. Ähnlich verhält es sich, wenn der Ursprung der transformierten Lichtquelle dem View-Frustum sehr nahe kommt, wie im Bild 26 zu erkennen ist.

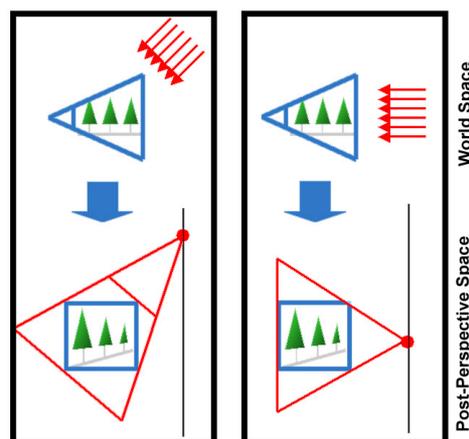


Bild 26 – Situationen die zu einer schlechten Qualität führen [Q10]

In diesem Fall wird die Perspektive durch die Transformation in den post-perspektivischen Raum der Kamera durch die Perspektive des entstandenen Point Lights wieder verringert (links) und im schlechtesten Fall komplett invertiert (rechts).

Die besten Ergebnisse erreicht man, wenn die Lichtquelle nach der Transformation zu einer direktionalen Lichtquelle wird, wie es in Bild 27 illustriert ist.

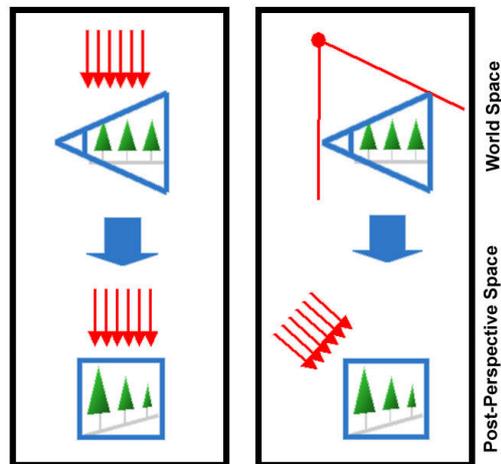


Bild 27 – Situationen die eine gute Qualität liefern [Q10]

In diesen Fällen wird die Perspektive, die durch die post-perspektivische Transformation entstanden ist, nicht verändert und liefert dadurch die besten Ergebnisse. Die Erzeugung dieser neuen virtuellen Kamera macht den Unterschied zwischen den Perspective-Shadow-Maps und den Light-Space-Perspective-Shadow-Maps [Q19] aus, die im nächsten Abschnitt genauer vorgestellt werden. Beim Perspective-Shadow-Mapping wählt man als Abstand für die Near-Plane der virtuellen Kamera denselben, wie er auch für die ursprüngliche Kamera gesetzt war. Nur der Abstand der Far-Plane wird angepasst, um das View-Frustum der ursprünglichen Kamera komplett einzufassen.

Da man die Shadow Map aus Sicht der Lichtquelle und nicht der Kamera erstellt, muss man das Koordinatensystem der virtuellen Kamera noch anpassen. Dafür bleibt der errechnete Kamerapunkt erhalten, man verändert nur den Blick- und Up-Vektor. Im neuen Koordinatensystem blickt die Kamera in Lichtrichtung und verwendet die Blickrichtung der alten Kamera als Up-Vektor. Im Anschluss transformieren wir die Szene in den post-perspektivischen Raum der virtuellen Kamera und erzeugen die Shadow Map. Beim folgenden Schattentest müssen wir diese Transformation vor dem Schattentest ebenfalls mit einberechnen, um die korrekten Texturkoordinaten für die post-perspektivische Shadow Map zu erhalten.

5.3 Light Space Perspective Shadow Mapping

In diesem Abschnitt gehe ich auf eine Verbesserung des eben beschriebenen Algorithmus des Perspective-Shadow-Mappings, die Light-Space-Perspective-Shadow-Maps ein [Q19]. Im Folgenden wird dieses Verfahren als *Lisp-Shadow-Mapping* bezeichnet.

Bei den Perspective-Shadow-Maps kommt es nach der Transformation in den post-perspektivischen Raum zu einer sehr starken perspektivischen Verzerrung. Dadurch werden nahe liegende Objekte sehr groß, während schon nach kurzer Blickweite deren Größe drastisch abnimmt. Dadurch lässt sich ein erheblicher Qualitätsverlust des Schattens bereits nach kurzen Distanzen feststellen. Aliasing tritt bei entfernten Objekten sogar deutlich stärker auf als beim Standard-Shadow-Mapping. Dieses Oversampling naher und Undersampling entfernter Objekte versuchen die Lisp-Shadow-Maps zu verhindern. Ein genauer Vergleich der einzelnen implementierten Schattenverfahren wird in Kapitel 10 durchgeführt.

Die Lisp-Shadow-Maps unterscheiden sich von dem Standardverfahren durch eine veränderte Berechnung der virtuellen Kamera. Wie bereits der Name des Verfahrens sagt, führen wir die Berechnungen für die virtuelle Kamera im Lichtraum durch. Dazu transformieren wir den Body durch die Multiplikation der Light-View-Matrix in Lichtkoordinaten. Diese Matrix errechnet sich aus der Kameraposition, der Lichtrichtung als Blickrichtung und der Blickrichtung der Kamera als Up-Vektor.

Nach der Transformation berechnen wir eine Bounding Box um den Body und bestimmen die beiden Extrempunkte. Anhand dieser Extrempunkte kann jetzt ein neuer Kamerapunkt sowie der neue Abstand der Near- bzw. Far-Plane berechnet werden. Bild X skizziert das View-Frustum der alten Kamera in blau sowie der neuen virtuellen Kamera in rot. V kennzeichnet dabei das View-Frustum der alten Kamera, P das der virtuellen.

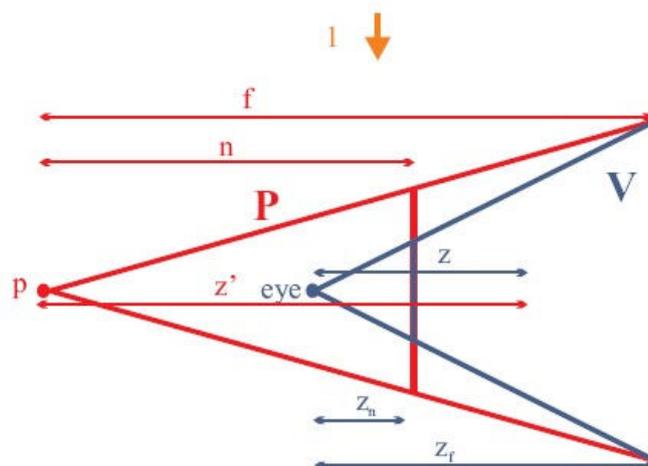


Bild 28 – Bestimmung des neuen Kamerapunkts [Q18]

Der virtuelle Kamerapunkt \mathbf{p} befindet sich im Abstand n zur Near-Plane der alten Kamera sowie im Abstand f zur alten Far-Plane. Daher müssen diese beiden Werte ermittelt werden, wozu die folgende Formel dient:

$$n_{opt} = z_n + \sqrt{z_f z_n}$$

z_n ist dabei Near-Plane Abstand der alten Kamera, z_f der Far-Plane Abstand.

Der virtuelle Kamerapunkt \mathbf{p} entscheidet dabei über die Stärke der Perspektive. Wird n so gewählt, dass es z_n entspricht, erhält man das Ergebnis des Standardverfahrens der Perspektivischen-Shadow-Maps. Geht \mathbf{p} hingegen gegen unendlich, wie es bei einer orthographischen Sicht der Fall ist, wird dasselbe Ergebnis wie beim Standard-Shadow-Mapping erreicht, da in diesem Fall das View-Frustum quadratisch wird, was keine perspektivische Verzerrung erzeugt.

Die View-Matrix für die virtuelle Kamera wurde bereits erstellt um die Berechnungen für n und f im Lichtraum durchzuführen. Diese Werte benutzen wir jetzt für die Berechnung der Projektionsmatrix für die virtuelle Kamera. Dafür verwenden wir die folgende Matrix:

$$\text{Lisp} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & a & 0 & b \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} \quad \text{mit} \quad \begin{aligned} a &= \frac{f+n}{f-n} \\ b &= \frac{-2 * f * n}{f-n} \end{aligned}$$

Die Matrix wird *Lisp-Matrix* genannt. Mit der Lisp-Matrix wird der Body perspektivisch verzerrt und wir errechnen uns wieder eine Bounding Box. Anhand der beiden Extrempunkte erstellen wir eine neue Projektionsmatrix, die die beiden Extrempunkte auf den Einheitswürfel projiziert. Die endgültige Projektionsmatrix für die virtuelle Kamera ergibt sich aus der Multiplikation mit der entstandenen Projektionsmatrix und der Lisp-Matrix.

Durch die oben aufgeführten Formeln soll ein optimales Ergebnis für die perspektivische Verzerrung erreicht werden, um Aliasing in wichtigen Bereichen zu verringern oder im Idealfall ganz zu entfernen. Einzelheiten können in [Q19] nachgelesen werden.

5.4 Zusammenfassung

In diesem Kapitel wurde das Grundprinzip hinter dem perspektivischen Shadow Mapping erklärt und zwei Verfahren vorgestellt, welche dies auf verschiedene Art und Weise versuchen umsetzen. Ich habe gezeigt, dass man durch die Einbeziehung einer Perspektive vor dem Erzeugen der Shadow Map wesentlich bessere Ergebnisse gegenüber dem Standard-Shadow-Mapping erreichen kann.

Die unterschiedlichen Ergebnisse der Perspective-Shadow-Maps und der Lisp-Shadow-Maps zeigen, wie wichtig dabei die Wahl einer optimalen Perspektive ist. Wird diese zu groß, sind Objekte bereits nach kurzer Entfernung geringer aufgelöst als beim Standard-Shadow-Mapping. Wird sie dagegen zu klein, sind keine wesentliche Verbesserung der Schattenqualität im Vergleich zum Standard-Shadow-Mapping zu erkennen.

Ein anders perspektivisches Shadow-Mapping-Verfahren, das Trapez-Shadow-Mapping, versucht ebenfalls, diese Perspektive optimal zu wählen, geht dabei jedoch anders an die Lösung dieses Problems heran. Auf das Trapez-Shadow-Mapping-Verfahren gehe ich im folgenden Kapitel ein.

6.0 Trapez Shadow Mapping

Die Trapezoidal-Shadow-Maps beschrieben erstmals Tobias Martin und Tiow-Seng Tan der National University of Singapore im Jahr 2004 [Q06]. Sie eignen sich, wie die Perspective-Shadow-Maps auch, 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 auf dem Prinzip des Perspective-Shadow-Mappings auf, enthält aber einige Veränderungen, die teilweise zu deutlich besseren Ergebnissen führen.

Ein weiterer Vorteil dieses Verfahrens liegt in der, im Vergleich zum Perspective-Shadow-Mapping, einfachen und leicht verständlichen Berechnung der perspektivischen Verzerrung. Diese kann durch die Änderung eines einzigen Parameters intuitiv an die Gegebenheiten der Szene angepasst werden.

6.1 Idee

Das Trapezoidal-Shadow-Mapping baut, die alle perspektivischen Shadow-Mapping-Verfahren auf der Idee auf, die Szene vor dem Erstellen der Shadow Map perspektivisch zu verzerren. Des weiteren wird diese nicht über die gesamte Szene erstellt, sondern nur über den Bereichen, die für die Schattenberechnung benötigt werden.

Das Trapezoidal-Shadow-Mapping berechnet ein Trapez, welches das Kamera-Frustrum aus Sicht der Lichtquelle vollständig einschließt. Durch später beschriebene Transformationen wird dieses Trapez zu einem Einheitswürfel transformiert, was den geforderten perspektivischen Effekt erzeugt. 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 bei manchen anderen perspektivischen Shadow-Mapping-Verfahren. 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 hier um das Kamera View Frustrum gelegt wird.

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

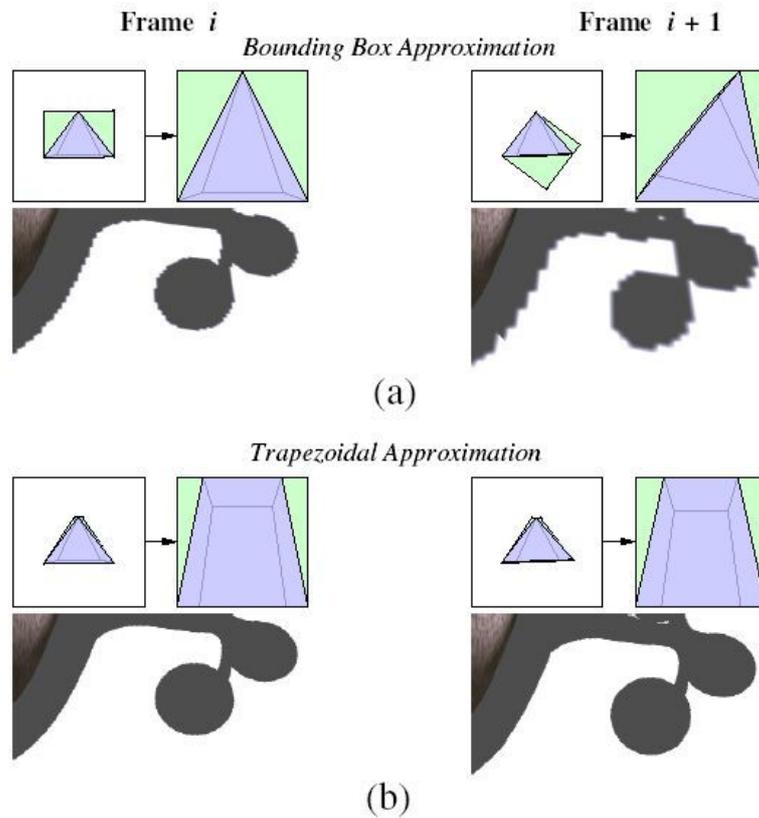


Bild 29 – Bounding Box um das Kamerafrustrum und entsprechende Schatten [Q05]

6.2 Funktionsweise

Die Initialisierung und Erzeugung der Shadow Map funktioniert beim Trapezoidal-Shadow-Mapping genauso wie beim Standard-Shadow-Mapping. Der Unterschied zu diesem liegt in der Fläche, über welche die Shadow Map generiert wird. Beim Trapezoidal-Shadow-Mapping wird diese 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 sowie 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.

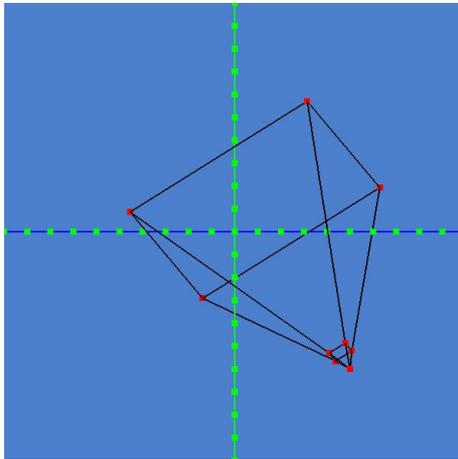


Bild 30a – Eckpunkte des Kamera Frustrums

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

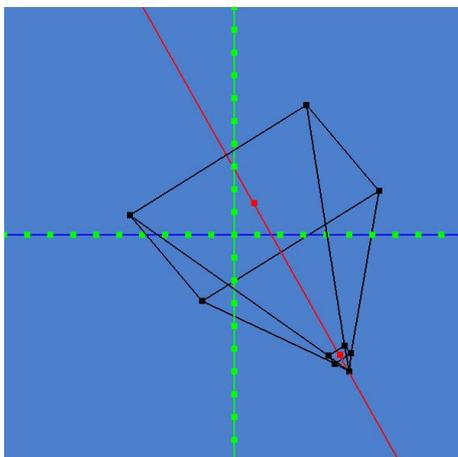


Bild 30b – Mittellinie durch die Planemittelpunkte

- An jedem der Eckpunkte legt man jetzt die Mittellinienorthogonale an und der jeweilige Schnittpunkt mit der Mittellinie wird berechnet. Aus diesen Schnittpunkten sucht man die zwei Punkte heraus, die zum Kameraursprung am nächsten und am weitesten entfernt liegen. Nur durch diese zwei Punkte zeichnet man 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*.

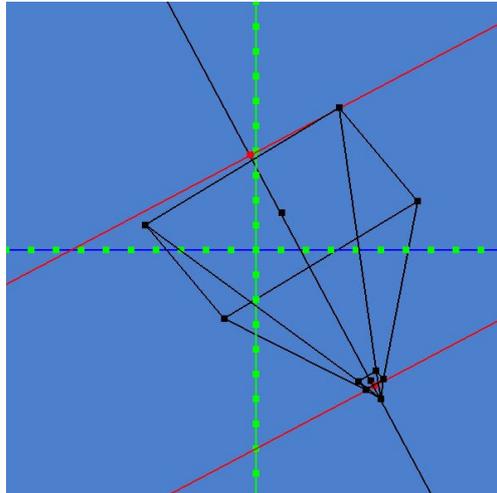


Bild 30c – Bestimmung der Base- und Top-Line

- 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 bestimmen, 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.

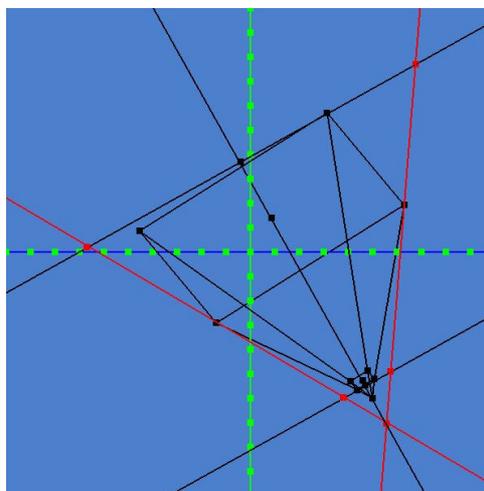


Bild 30d – Seitenlinien und Trapezpunkte

Die skizzierte Methode, die Seitenlinien zu bestimmen, liefert jedoch nur unbefriedigende Ergebnisse. In Bild 30d 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 deshalb unbefriedigende Ergebnisse, weil durch die Umrechnung des Trapezes zum Einheitswürfel der schmale Teil des Trapezes sehr stark perspektivisch verzerrt 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 31 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 vereinzelt Schattenblöcke zu erkennen.

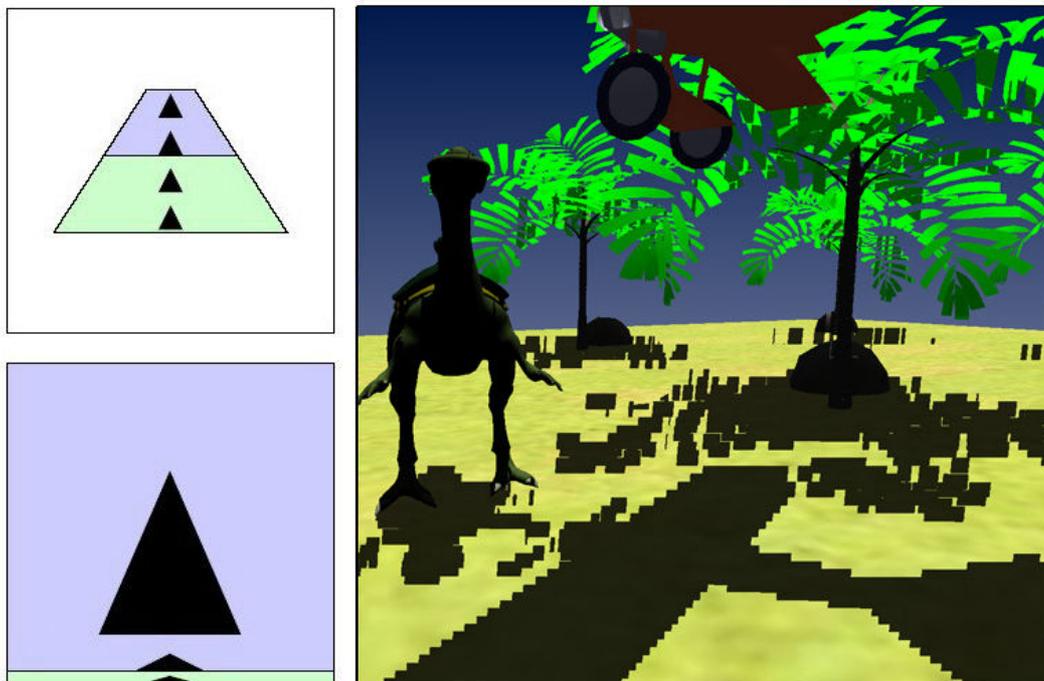


Bild 31 – Links Kamera Frustrum aus Sicht der Lichtquelle und Shadow Map Bereich
Rechts: Szene, die mit diesem Verfahren Schattiert wird.

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 perspektivische Verzerrung nicht so stark aus. Die gewünschte perspektivische Verzerrung soll bei diesem Verfahren so gewählt werden, dass 80% der Shadow-Map-Auflösung einem genau definiertem Bereich des Kamerafrustum zur Verfügung gestellt wird. Diese Region wird *Focus Region* genannt. Bild 32 verdeutlicht die Berechnung des gesuchten Punktes.

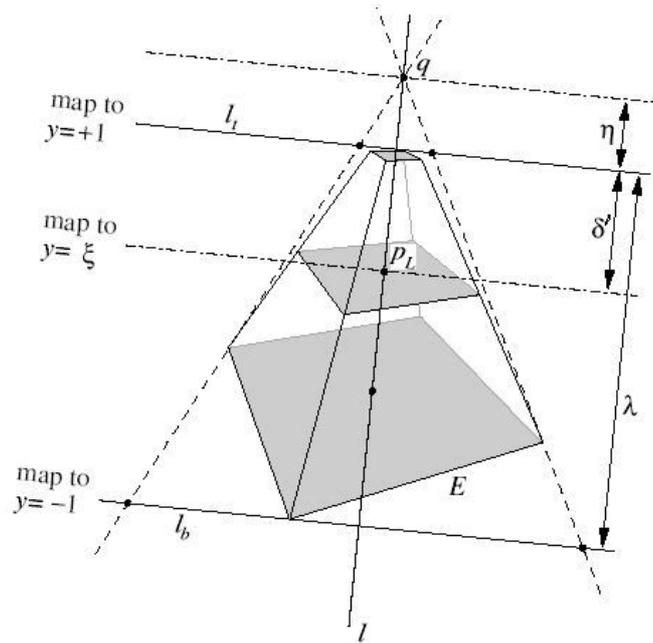


Bild 32 – Berechnung des 80%-Punktes [Q05]

Die Focus Region erstreckt sich über den Abstand δ vom Kameraursprung in Sichtrichtung. Der Punkt p markiert einen Punkt, der genau im Abstand δ vom Kameraursprung entfernt liegt. p_L 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 den Punkt projiziert, der der 80%-Line im Einheitswürfel entspricht. Dem Bereich vor diesem Punkt wird dadurch 80% der Fläche der Shadow Map zur Verfügung gestellt. Aus diesem Grund wird diese Regel *80%-Regel* genannt.

Man erreicht dies 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 projiziert. Die Base- und Top-Line werden bei der Berechnung auf $y = -1$ sowie $y = +1$ projiziert, wodurch die 80%-Line bei $y = -0.6$ liegt. 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 Projektion:

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

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

In der Implementation wurde die Berechnung so angepasst, dass der Benutzer die Focus Region durch einen Wert zwischen 0 und 1 festlegen kann. Eine Angabe von 0.5 würde bedeuten, dass die vordere Hälfte des Kamerafrustum als Focus Region genommen werden soll.

Die Seitenlinien ziehen wir vom Punkt q zu den äußeren Eckpunkten der Far-Plane. Die vier gesuchten Trapezpunkte ergeben sich jetzt aus den Schnittpunkten der Seitenlinien mit der Base- und Top-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.

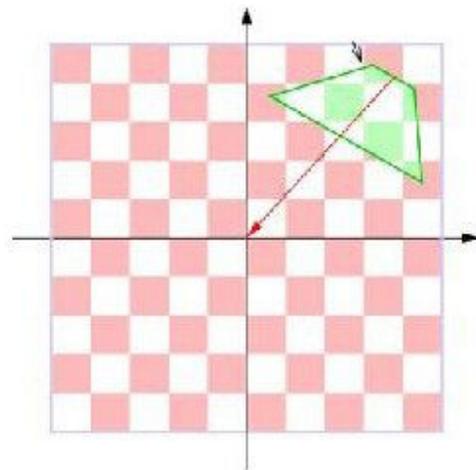
Ziel des nächsten Schrittes ist die Berechnung einer Matrix, die dieses Trapez zu einem Einheitswürfel transformiert. Diese Matrix wird als N_T bezeichnet. Im Folgenden wird veranschaulicht, wie eine solche Matrix erstellt werden kann [Q06].

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.

$$u = (t_2 + t_3) / 2$$

$$T_1 = \begin{vmatrix} 1 & 0 & 0 & -x_u \\ 0 & 1 & 0 & -y_u \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

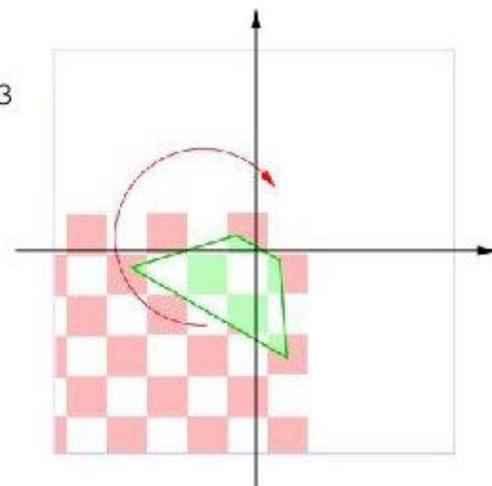


Schritt 2:

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

$$u = (t_2 - t_3) / |t_2 - t_3|$$

$$R = \begin{vmatrix} x_u & y_u & 0 & 0 \\ y_u & -x_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

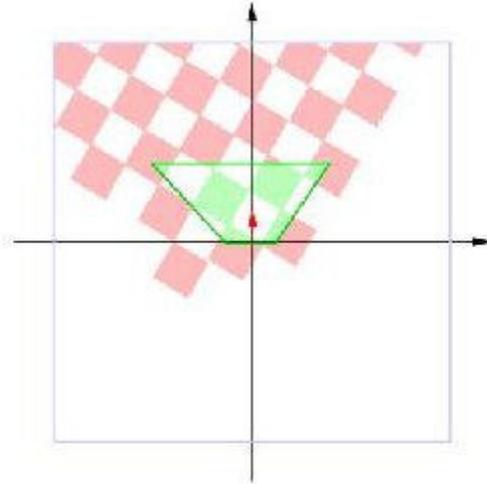


Schritt 3:

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. Das wird durch die Matrix T_2 erreicht.

$$u = R * T_1 * i$$

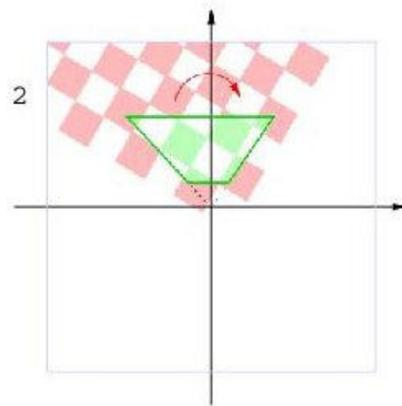
$$T_2 = \begin{vmatrix} 1 & 0 & 0 & -x_u \\ 0 & 1 & 0 & -y_u \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

**Schritt 4:**

Nun muss das Trapez mit der Matrix H gesichert werden, so dass es symmetrisch zur Y-Achse liegt.

$$u = (T_2 * R * T_1 * (t_2 + t_3)) / 2$$

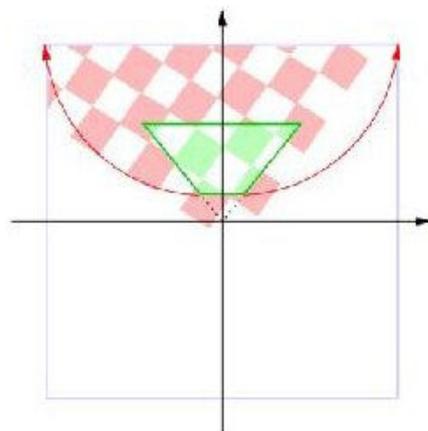
$$H = \begin{vmatrix} 1 & -x_u/y_u & 0 & 0 \\ 0 & 1/y_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

**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° .

$$u = H * T_2 * R * T_1 * t_2$$

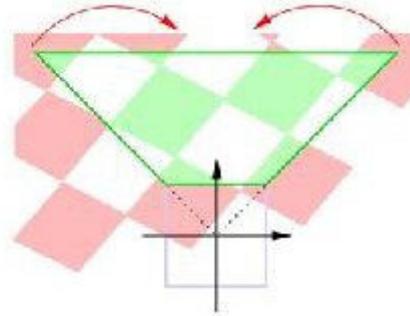
$$S_1 = \begin{vmatrix} 1/x_u & 0 & 0 & 0 \\ 0 & 1/y_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



Schritt 6:

Die folgende Matrix N formt das Trapez zu einem Rechteck.

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

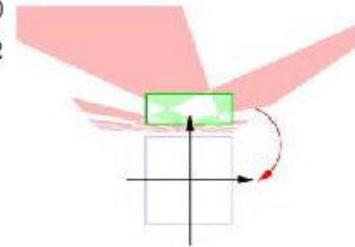
**Schritt 7:**

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.

$$u = N * S_1 * H * T_2 * R * T_1 * t_0$$

$$v = N * S_1 * H * T_2 * R * T_1 * t_2$$

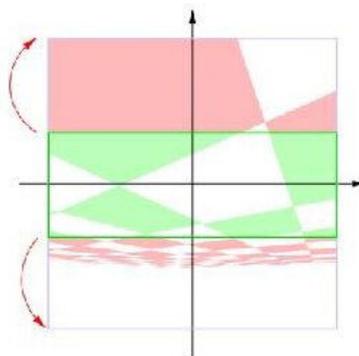
$$T_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -(y_u/w_u + y_v/w_v)/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**Schritt 8:**

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

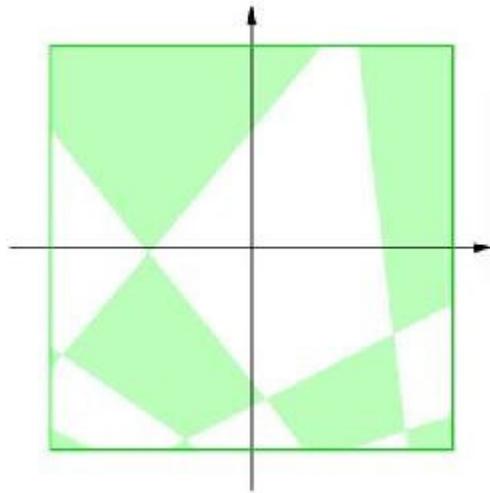
$$u = T_3 * N * S_1 * H * T_2 * R * T_1 * t_0$$

$$S_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -w_u/y_u & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



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 die Shadow Map generiert werden. Der anschließende Schattentest funktioniert genauso, wie beim Standard-Shadow-Mapping. Es wird jedoch zusätzlich die Projektionsmatrix N_T auf die Szene angewendet.

6.3 Spezialfälle und Verbesserungen

Die beschriebenen Verbesserungen beschreiben nur das Standardverfahren der Trapez-Shadow-Maps. Es gibt jedoch Situationen, in denen es auch hier zu deutlich spürbaren Qualitätsverlusten bei der Darstellung kommt oder aber eine Berechnung der Schatten nicht mehr nach dem genannten Algorithmus funktioniert. Im Rahmen dieser Arbeit wurden die zu Grunde liegenden Verfahren optimiert, um noch Schattenqualitäten zu erreichen. Diese Optimierungen werden in den folgenden Abschnitten aufgezeigt. Die Ergebnisse dieser Optimierungen werden in Kapitel 10 präsentiert.

6.3.1 Problem bei der Trapezberechnung

Die Situationen, dass kein Trapez nach dem genannten Verfahren berechnet werden kann tritt auf, sobald sich aus Sicht des Lichtes die Kameraposition bzw. deren Near-Plane innerhalb der Kamera Far-Plane befindet. In dieser Situation kann das Trapez nicht mehr mit dem beschriebenen Verfahren gebildet werden. Bild 33a verdeutlicht diese Situation.

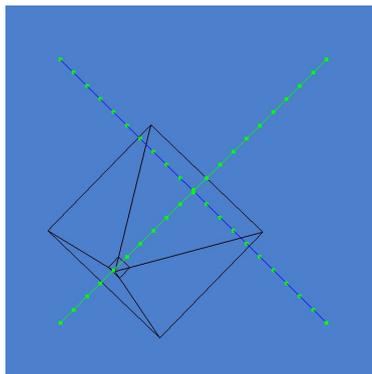


Bild 33a – Kein Trapez berechenbar

Die Lösung des Problems liegt darin zu überprüfen, wann diese Situation eintritt und in diesem Fall nicht zu versuchen, das Trapez zu berechnen, sondern statt dessen einen anderen Lösungsweg zu gehen. Dazu wird die Far-Plane der Kamera als Trapez verwendet. Diese schließt in der genannten Situation das Kamerafrustum vollständig ein. Nun ist eine korrekte Schattierung ohne Fehlstellen gewährleistet. Bild 33b zeigt dies anhand des roten Bereichs.

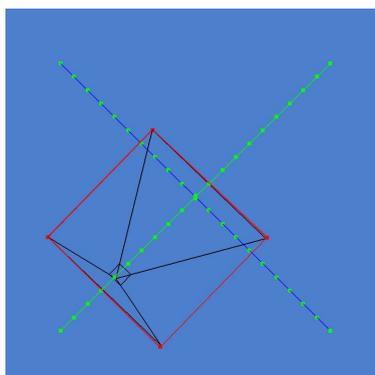


Bild 33b – Far-Plane als Trapez

6.3.2 Automatische Anpassung der Base- und Top-Line

Je nach gewählter Sichtweite der Kamera fällt die Far-Plane der Kamera sehr groß aus, was gerade im oben genannten Fall zu einer deutlich sichtbaren Verschlechterung der Schattenqualität führt. Auch im Standardfall wandert durch eine zu große Sichtweite die Top-Line sehr weit nach oben, was ebenfalls die Schattenqualität beeinträchtigt. So kann es sein, dass trotz einer großen Sichtweite nur nahe liegende Objekte sichtbar sind, weil sie weiter entfernte Objekte verdecken. Der gesamte Bereich hinter diesen Objekten ist in dieser Situation für den Schattenwurf irrelevant und somit verschwendete Auflösung auf der Shadow Map.

In vielen Situationen liegt in einem Abstand vor der Near-Plane kein Objekt. Da aber gerade dieser Bereich beim Trapez-Shadow-Mapping besonders viel Auflösung der Shadow Map zur Verfügung gestellt bekommt, fällt eine zu weit hinten liegende Near-Plane und daraus resultierende Base-Line besonders ins Gewicht. Wird der Abstand der Near- und Far-Plane optimal gewählt, führt dies zu einer spürbaren Verbesserung der Schattenqualität.

Für die optimale Anpassung der Base- und Topline benötigen wir demnach die Pixel mit dem geringsten und dem größten Tiefenwert aus Sicht der Kamera. Der Tiefenwert des am nächsten liegenden Pixel wird als Abstand der Near-Plane gewählt. Die Tiefe des am weitesten entfernt liegenden Pixel als Sichtweite. Die Base- und Top-Line wird auf diese Weise immer optimal an die Szene angepasst.

6.3.3 Konstantere Schattenqualität

Wie bei allen perspektivischen Shadow-Mapping-Verfahren liefert auch das Trapez-Shadow-Mapping keine konstant gleiche Schattenqualität. Je weiter sich die Blickrichtung der Kamera der des Lichtes annähert, desto schlechter wird die Schattenqualität. Der Grund für diese Verschlechterung wurde in den vorigen Kapiteln für die Perspective- und Lisp-Shadow-Maps bereits erklärt. Beim Trapez-Shadow-Mapping liegt das Problem jedoch in einem anderen Bereich. Die folgende Skizze zeigt eine Situation, in der sich die Blickrichtung der Lichtrichtung annähert:

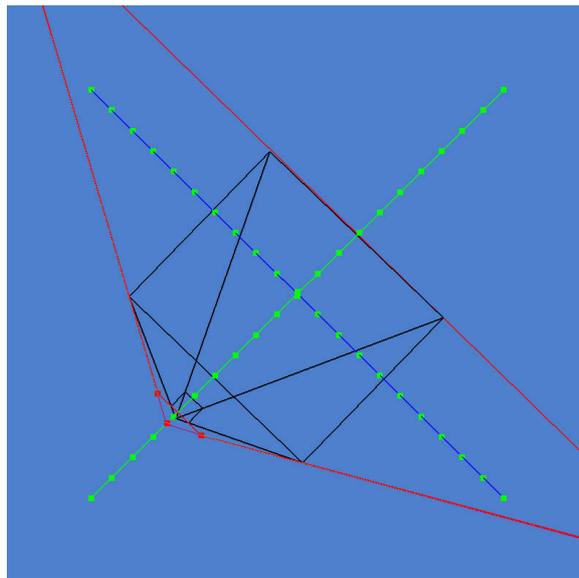


Bild 34a – Sehr großer Winkel zwischen den Seitenlinien

In rot ist der Trapezbereich markiert, der später auf den Einheitswürfel transformiert wird. Wie unschwer zu erkennen ist, fällt die kurze Seite des Trapezes in der Nähe der Near-Plane im Vergleich zur langen Seite sehr klein aus. Daraus resultiert durch die Transformation zum Einheitswürfel eine sehr starke perspektivische Verzerrung. Nur Objekte, die sehr nah an der Kamera liegen, liefern eine gute Schattenqualität, während entferntere schon nach kurzer Distanz auf einen sehr kleinen Bereich abgebildet werden und hier in Folge dessen starkes Aliasing auftritt. Eine zu starke perspektivische Verzerrung kann man vermeiden, wenn diese Situation erkannt und die perspektivische Verzerrung verringert wird. Da der Abstand des Stützpunktes direkten Einfluss auf die Stärke der Verzerrung hat, erreicht man durch die Verlagerung dieses Punktes nach hinten eine geringere Verzerrung.

Um die genannte Situation erkennen zu können, berechnet man den Winkel, den die beiden Seitenlinien aufspannen. Liegt dieser über einem festgelegten Grenzwert wird die Position des Stützpunktes nach hinten verlagert, bis der Winkel wieder im geforderten Bereich liegt. Durch eine solche Korrektur ist eine relativ gleichmäßige Schattenqualität gerade in den Situationen gewährleistet, in denen sich die Blickrichtung der Lichtrichtung annähert. Bild 34b zeigt das korrigierte Trapez.

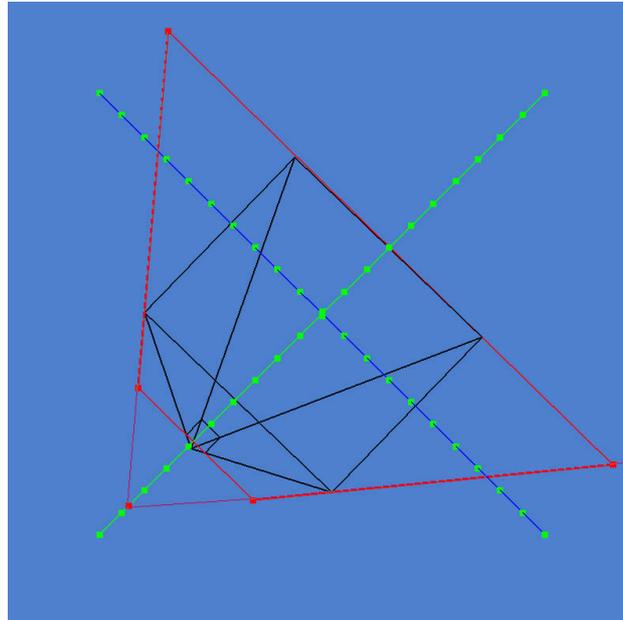


Bild 34b – Korrigiertes Trapez

6.3.4 Polygon Offset Problem

Das Polygon Offset Problem kennen wir vom Standard-Shadow-Mapping. Es wurde dort durch das addieren eines festgelegten Offsets behoben. Das Problem tritt jedoch bei der Benutzung des Trapezoidal-Shadow-Mappings verstärkt auf. Durch die perspektivische Verzerrung verstärkt sich dieser Effekt in gewissen Bereichen. Das Offset ist für Bereiche nah an der Kamera zu hoch, wodurch hier Schatten verschwinden, in Bereichen die weiter entfernt liegen zu gering, wodurch es zu fehlerhaften Schattendarstellungen kommt.

Das Problem lässt sich jedoch ohne großen Aufwand beheben. Die passende Lösung wird in [Q06] vorgestellt. Dazu muss nur eine kleine Veränderung an den Tiefenwerten vorgenommen werden. Im Fragment Shader werden die perspektivisch verzerrten Z-Werte durch die unverzerrten Z-Werte der Szene ersetzt. Nach der Veränderung tritt das verstärkte Polygon-Offset-Problem nicht mehr auf, ein statisches Polygon-Offset reicht für die Korrektur wieder aus.

6.4 Zusammenfassung

Das Trapezoidal-Shadow-Mapping zählt zu den perspektivischen Shadow-Mapping-Verfahren. Hierbei wird die Perspektive der Kamera genutzt, die Shadow Map optimal aufzubauen, also deren Auflösung so gut es geht an wichtigen Stellen zu konzentrieren und an weniger wichtigen Stellen zu verringern. Werden die Verfahren zur Schattierung großer Szenen benutzt, z.B. für die Simulation von Sonnenlicht, so wird die Darstellungsqualität im Vergleich zum Standard-Shadow-Mapping erheblich verbessert. Durch weitergehende Verbesserungen können nachteilige Effekte wie plötzlich nachlassende Schattenqualität oder ein verstärktes Polygon Offset Problem verringert werden.

Anwendungstechnisch liegt der Vorteil des Trapez-Shadow-Mappings in der intuitiven Bedienbarkeit. Hierbei ist besonders die verständliche Wahl der Focus Region hervorzuheben, die es dem Benutzer ermöglicht, ohne Kenntnis des Verfahrens die Schattenqualität auf die für ihm wichtigen Bereiche zu konzentrieren.

7.0 PCF Shadow Mapping

Das Percentage-Closer-Filter-Shadow-Mapping, kurz *PCF-Shadow-Mapping*, ist ein Verfahren zur Darstellung von weichen Schattenkanten. Dabei erzeugt das Verfahren uniforme Soft-Shadows. Die Schattenkanten verschwimmen über einen konstanten Abstand. Den Bereich, in dem die Schattenkanten verlaufen, nennt man *Penumbra*. Die Bereiche, die voll schattiert sind nennt man *Umbra*. Die Umbra entsteht nur in den Bereichen, in denen kein Teil der Fläche der Lichtquelle den entsprechenden Pixel sehen kann.

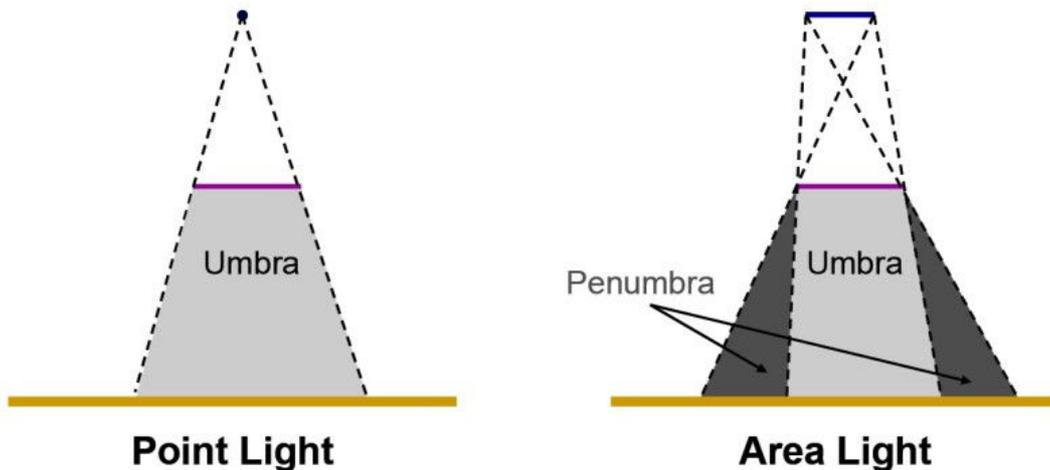


Bild 35 – Umbra und Penumbra [Q14]

Wie man in der Grafik erkennt, hängt die Größe der Penumbra von der Fläche der Lichtquelle ab. Da es im OpenGL / OpenSG keine flächigen Lichtquellen gibt, müssen sie simuliert werden.

7.1 Idee

Beim PCF-Shadow-Mapping wird für einen Pixel nicht nur ein Schattentest durchgeführt, sondern mehrere. Dazu wird der aktuelle Pixel zusätzlich mit dem umliegenden Tiefenwerten der Shadow Map verglichen und die Ergebnisse des Schattentests werden aufaddiert. Danach wird ein Durchschnittswert ermittelt, der als Schattierung für den Pixel gewählt wird. Schauen wir uns das folgende Beispiel an:

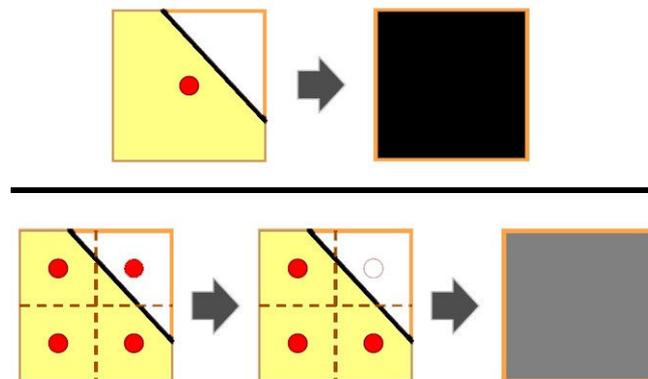


Bild 36 – Prinzip eines PCF-Filters [Q14]

In der oberen Reihe sehen wir einen normalen Schattentest. Der rote Punkt stellt den Mittelpunkt des Pixels dar. Der hier liegende Tiefenwert wird für den Schattentest dieses Pixels verwendet und mit dem entsprechenden Wert in der Shadow Map verglichen. Die beige Fläche zeigt den Bereich, in dem der Schattentest eine schattierte Fläche zurückliefern würde. Der Pixel ist in dem oben gezeigten Beispiel schattiert, also schwarz.

Die untere Reihe zeigt die Funktionsweise eines 4-Sample-PCF-Filters. Wir erweitern den Pixel um drei Bereiche und führen zu jedem Mittelpunkt der Flächen einen eigenen Schattentest durch. In unserem Beispiel liefern drei Tests eine Schattierung zurück, einer eine beleuchtete Fläche. Der entstandene Pixel ist demnach zu 75% schattiert, wird also dunkelgrau dargestellt.

Der Bereich, um den ein Pixel erweiterbar ist, ist natürlich veränderbar. Man kann auch mehr als nur die view Samples in die Berechnung einfließen lassen. Je größer man diesen Bereich wählt, desto mehr verlaufen die Schatten an den Kanten und umso größer werden die Penumbra und die daraus resultierende simulierte Fläche der Lichtquelle.

7.2 Funktionsweise

Wie beim Standard-Shadow-Mapping rendert man auch bei diesem Verfahren zuerst die Szene aus Sicht der Lichtquelle und schreibt die Tiefenwerte in die Shadow Map. Beim Standard-Shadow-Mapping würde jetzt für jedes Pixel nur ein einzelner Schattentest mit dem resultierenden Tiefenwert aus der Shadow Map durchgeführt und der Pixel schattiert oder beleuchtet dargestellt werden. Demnach kann der Pixel zwei Zustände einnehmen; schattiert oder beleuchtet. Für einen weichen Schattenübergang muss der Pixel mehr Zustände als nur die zwei genannten einnehmen können. Aus diesem Grund führt man den Schattentest für einen Pixel nicht nur mit dem entsprechendem Tiefenwert der Shadow Map durch, sondern auch mit den angrenzenden Tiefenwerten. Je größer der zu untersuchende Bereich gewählt ist, desto mehr Schattentests werden für den Pixel durchgeführt. Ein Zähler wird jedesmal um eins erhöht, sobald ein Schattentest eine schattierte Fläche liefert.

Nachdem alle Schattentests durchgeführt sind, teilt man den Zähler durch die Gesamtanzahl der durchgeführten Tests, wodurch man einen Durchschnittswert erhält. Dieser dient als Schattenfaktor für den Pixel. Er gibt an, wie stark der Pixel schattiert ist. Je mehr Schattentests fehlgeschlagen sind, desto höher ist der Durchschnittswert und der Pixel dementsprechend dunkler. Mit der Angabe der Suchreichweite kann man festlegen, wie weit die Schattenkanten verlaufen sollen. Allerdings muss man dabei beachten, dass die Anzahl der durchgeführten Schattentests im Bezug auf die Suchreichweite quadratisch ansteigt.

Durch diese Erkenntnis leuchtet es ein, dass nach diesem Verfahren nur begrenzte Schattenverläufe in Echtzeit realisierbar sind. Daher muss für starke Schattenverläufe ein Weg gefunden werden, die Anzahl der Samples drastisch zu reduzieren.

7.3 Verbesserungen

In diesem Abschnitt gehe ich auf die Erhöhung der Performance des PCF-Shadow-Mappings ein. Zudem zeige ich, wie die Qualität durch die Nutzung moderner Grafikhardware erhöht werden kann.

7.3.1 Performanceerhöhung

Sollen die Schatten weit verlaufen, muss die Suchreichweite entsprechend erhöht werden. Dies hat auch eine deutliche Erhöhung der durchzuführenden Schattentests zur Folge, die Framerate nimmt mit jeder Suchreichweitenerhöhung drastisch ab. Aus diesem Grund ist die simulierte Penumbra in ihrer Größe stark beschränkt. Der einzige Weg, die Bildrate auch bei einer hohen Penumbra auf ein akzeptables Maß zu bringen, liegt in der Reduzierung der durchzuführenden Schattentests. Daher gibt man die gewünschte Penumbra an und legt gleichzeitig die maximale Anzahl der zulässigen Schattentests fest, die pro Zeile durchgeführt werden dürfen.

Bild 37a zeigt das Ergebnis dieser Reduzierung. Hier wurde die Suchreichweite auf acht Pixel der Shadow Map festgelegt, die maximale Anzahl pro Samples aber von den erforderlichen acht Samples pro Zeile für eine akzeptable Qualität auf sechs Samples pro Zeile beschränkt. Dadurch ergibt sich eine Sampleanzahl von nur 36 pro Pixel statt 64, wie beim vollwertigen PCF-Verfahren. Die Framerate erreicht so wieder geeignete Werte. Links im Bild ist das komplett gesampelte, rechts das reduzierte Ergebnis dargestellt.

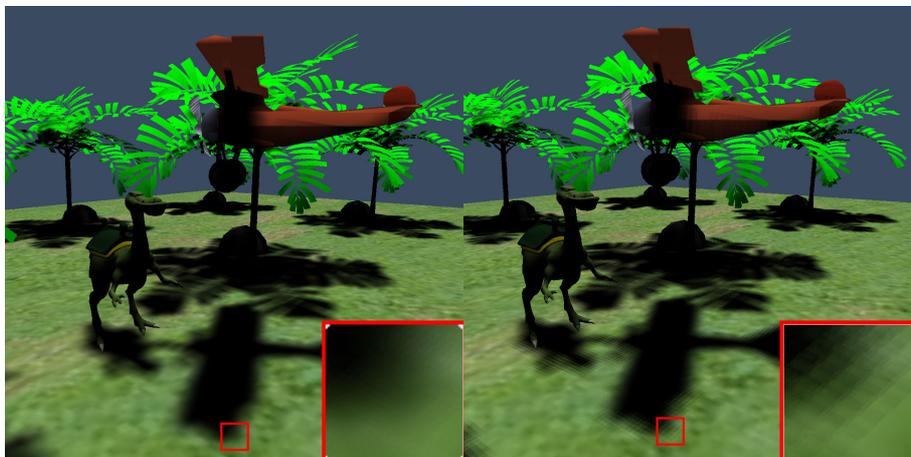


Bild 37a – Links: Ergebnis eines sehr hoch gesampelten Filters, Rechts: Ergebnis mit reduzierter Sampleanzahl

In dem vergrößerten Bereich des Bildes erkennt man, dass durch die Reduzierung der getesteten Samples auch die Qualität deutlich nachlässt. Die Übergänge sind nicht mehr fließend, die Schattenfarbe weist deutlich abgegrenzte Bereiche auf. Durch die reduzierte Anzahl der Samples lassen sich im endgültigen Pixel nicht mehr so viele Graustufen darstellen wie bei voller Sampleanzahl. Bei einem 8x8-PCF-Filter werden beispielsweise 64 Samples pro Pixel getestet. Es können also 64 Graustufen dargestellt werden. Reduziert man die Sampleanzahl auf sechs Pixel pro Zeile, werden nur noch 36 Samples getestet, wodurch maximal 36 unterschiedliche Grauwerte darstellbar sind. Im folgenden Abschnitt werden geeignete Schritte vorgenommen, um die Schattenqualität bei reduzierter Sampleanzahl wieder anzuheben.

7.3.2 Qualitätsverbesserung

Wie wir gerade festgestellt haben, liegt das Problem der Schattenqualität nach der Reduzierung der Samples in den fehlenden Übergängen der einzelnen Grauwerte. Hier bieten moderne Grafikkarten bereits eine einfache Lösung. Sie gestatten es, einen hardwareseitigen PCF-Filter zu verwenden, der deutlich schneller arbeitet als ein selbst programmierter Filter. Aktivieren wir den Filter für unser entstandenes Bild, ist bereits eine deutliche Erhöhung der Qualität zu erkennen (siehe Bild 37b).

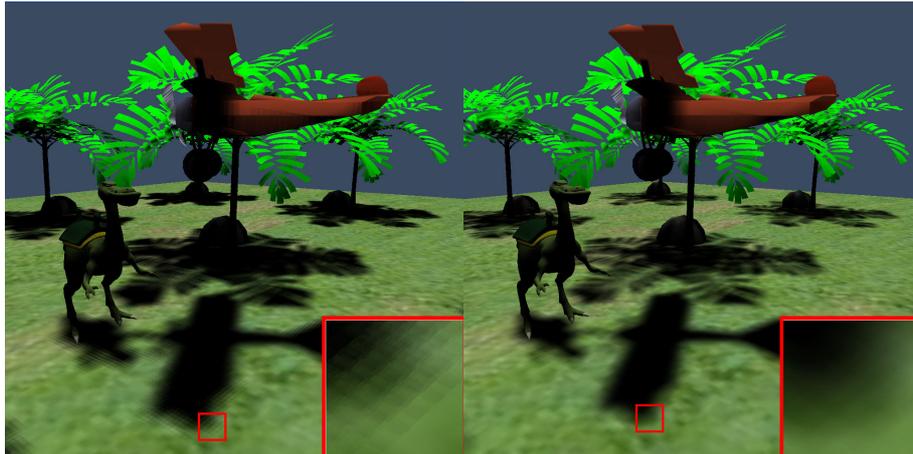


Bild 37b – Links: Schattenkante ohne Filter, Rechts Schattenkante mit aktiviertem Filter

Man erkennt in der Vergrößerung, dass die Übergänge nicht mehr so stark voneinander abgegrenzt sind. Die Qualität kommt dem voll gesampelten Schatten sehr nahe. Nur bei genauem Hinsehen kann man noch größere gleichfarbige Flächen erkennen. Die Framerate liegt jedoch weit über der des vollwertigen PCF-Filters. Steht der Hardware-PCF-Filter nicht zur Verfügung, muss nach anderen Möglichkeiten zur Qualitätsverbesserung gesucht werden.

Eine solche Möglichkeit wäre das manuelle „verwischen“ dieser Grauwerte, z.B. mit Hilfe eines Gauß-Filters. Keine schlechte Idee, wenn man bedenkt, dass sich durch geeignete Filter auch große Penumbrae effizient erzeugen lassen. Es wäre möglich, z.B. mehrere Gauß-Filter mit relativ kleinen Suchregionen hintereinander zu schalten und dadurch auch mit wenigen Samples große Schattenverläufe zu simulieren. Das Problem hierbei ist, dass es keinen Sinn macht, die ganze Szene zu verwischen. Dadurch würden nicht nur die Schatten, sondern die ganze Szene unscharf werden. Auch das Filtern der Shadow Map bringt keinen Vorteil, da hier nur Tiefenwerte gespeichert sind. Darum muss nach einem anderen Weg gesucht werden.

Eine mögliche Lösung stelle ich jetzt vor. Diese basiert auf einem Verfahren, das in [Q17] genannt wird. Man rendert die Szene bei diesem Vorgehen zuerst ohne Schatten in eine Textur, die man *Color Map* nennt. Im Anschluss daran führt man die Schattentests durch, verrechnet die Ergebnisse aber nicht direkt mit der Szene, sondern speichert diese ebenfalls in einer Textur. In dieser Textur stehen nur die Werte, die man tatsächlich verwischen möchte, weshalb man hierauf jetzt einen Gauß-Filter anwenden kann, um die Schattenkanten verlaufen zu lassen. Am Ende verrechnet man die *Shadow Factor Map* mit der *Color Map* und gibt das Ergebnis aus.

Dieses Verfahren hat allerdings einen gravierenden Nachteil. Verdeckt ein Objekt einen schattierten Bereich, so scheint dieses Objekt zu leuchten, da die Schatten um das Objekt verschwinden. Der Grund hierfür ist schnell gefunden. Verdeckt ein ungeschattertes Objekt einen Bereich der normalerweise schattiert ist, schlägt der Schattentest hier nicht fehl, weil das Objekt das den Bereich verdeckt, nicht im Schatten liegt. Aus diesem Grund wird auf der Shadow Factor Map an der entsprechenden Stelle ein ungeschattierter Fleck eingetragen. Filtert man diesen Bereich, verschwinden auch die Schattenkanten am Rand des Objekts, die dann den gezeigten Leuchteffekt hervorrufen.

Um das Problem zu beheben erstellt man die Color Map und die Shadow Factor Map wie eben beschrieben, mit dem Unterschied, dass es sich bei dem verwendeten Schattenverfahren nicht um Standard-Shadow-Mapping handelt, sondern um einen reduzierten PCF-Filter. Der Leuchteffekt tritt bekanntermaßen nur an Schattenkanten auf, die neben einem ungeschatterten Objekt liegen. Aus diesem Grund wird eine zusätzliche Textur erstellt. Diese enthält alle Objektkanten, erzeugt durch einen Edge-Detection-Shader aus Kamerasicht.

Es folgt das Filtern der Shadow Factor Map. Dafür werden ein horizontal/vertikaler, sowie ein diagonaler Gauß-Filter hintereinander geschaltet. Zusätzlich übergibt man diesen Filtern die Edge Map. Das aktuelle Pixel wird jetzt nur verwischt, wenn sich in der Suchregion des Gauß-Filters keine Objektkante befindet. Dies gewährleistet, dass Schattenkanten an Objekten nicht verwischt werden.

Ergebnisse dieses Verfahrens kann man in Kapitel 10 betrachten.

7.4 Zusammenfassung

Es wurde die Funktionsweise eines PCF-Filters erklärt und dessen Funktion für das Erzeugen weicher Schatten aufgezeigt. Die erreichten Ergebnisse sehen deutlich realistischer aus als bei harten Schatten. Es wurde aber auch gezeigt, dass gerade für große Penumbra ein vollwertiger PCF-Filter aus Leistungsgründen nicht mehr praktikabel ist. Aus diesem Grund muss man einen Kompromiss zwischen Qualität und Leistung eingehen. Als Resultat wurde der reduzierte PCF-Filter vorgestellt, der zwar die Leistung deutlich erhöht, gleichzeitig aber die Schattenqualität erheblich verringert. Um die erreichten Ergebnisse wieder aufzuwerten wurde die Nutzung moderner Grafikhardware sowie das nachträgliche Filtern vorgestellt. Dadurch konnten annähernd die gleichen Ergebnisse wie bei einem vollwertigem PCF-Filter erreicht werden, die Framerate ist hier jedoch deutlich höher.

8.0 Perspektivisches PCF Shadow Mapping

Das Perspektivische-PCF-Shadow-Mapping, kurz *PPCF-Shadow-Mapping*, basiert auf der Grundlage des normalen PCF-Shadow-Mappings wie es im vorigen Kapitel vorgestellt wurde. Es werden einige Modifikationen vorgenommen, um perspektivisch korrekte Soft-Shadows, sogenannte Schlagschatten, erzeugen zu können. Das Verfahren wurde erstmals 2005 von Randima Fernando [Q15] vorgestellt

Schlagschatten sind Soft-Shadows, die je nach Größe der Lichtquelle und Abstand des Schattenempfängers vom Schattenerzeuger verschieden weit verlaufen.

8.1 Idee

Hinter dem PPCF-Shadow-Map-Verfahren steckt die Idee, dass kein fester Wert für den Suchbereich verwendet wird, sondern dieser Wert variabel gewählt wird. Die Größe der Umbra sowie Penumbra verändert sich abhängig von der Größe der Lichtquelle, sowie dem Abstand des Blockers und Schattenempfängers zur Lichtquelle.

8.2 Funktionsweise

Der PPCF-Algorithmus läuft im Allgemeinen in drei Schritten ab, um den perspektivisch korrekten Schatten für einen Pixel zu berechnen:

1. Blockersuche
2. Bestimmung der Penumbra
3. Anwendung eines variablen PCF-Filters

Die Aufgaben und das Vorgehen der einzelnen Schritte erläutere ich in den folgenden Abschnitten.

8.2.1 Blockersuche

Um den Abstand des Blockerpixels zur Lichtquelle zu bestimmen, erzeugt man zunächst eine normale Shadow Map aus Sicht des Zentrums der Lichtquelle. Als nächstes wird eine *Search Region* bestimmt, in der man die Shadow Map in der Umgebung des aktuellen Pixels nach Blockern durchsucht. Die Größe der Search Region hängt dabei von der Größe der Lichtquelle und dem Abstand des aktuellen Pixels zur Lichtquelle ab. Bild 38 verdeutlicht die Bestimmung der Search Region.

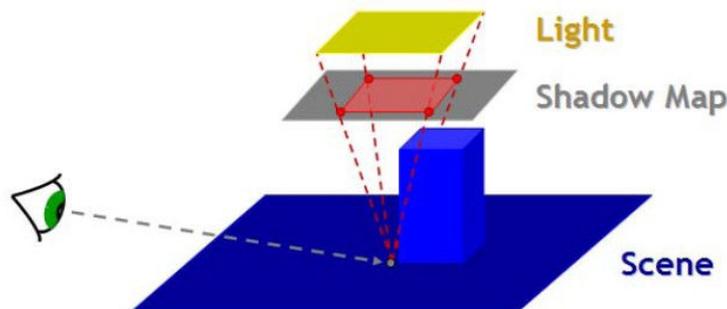


Bild 38 – Bestimmung der Search Region [Q14]

Danach sampelt man die Shadow Map in diesem Bereich und führt die Schattentests durch. Die Tiefenwerte der Shadow Map, bei denen der aktuelle Pixel schattiert wäre, werden addiert und nach dem Sampeln durch die Anzahl der gefundenen Blocker geteilt. Das Resultat ist ein durchschnittlicher Tiefenwert der gefundenen Blocker. Diesen Wert benötigen wir für die Berechnung der Penumbra. Der Durchschnittswert liefert dabei die besten Ergebnisse. Verwendet man stattdessen den kleinsten Tiefenwert der gefundenen Blocker, entstehen falsche Schatten, sobald sich mehrere Schatten überschneiden.

Bei der beschriebenen Blockersuche kann es natürlich auch passieren, dass in der Search Region keine Blocker gefunden werden, der Pixel also voll beleuchtet ist. In diesem Fall entfällt später die Berechnung für die Penumbra oder den PCF-Filter, der Pixel kann voll beleuchtet gezeichnet werden. Dieser Fall sollte also aus Performancegründen erkannt werden, um die dann überflüssigen Schritte von der Berechnung auszuschließen.

8.2.2 Bestimmung der Penumbra

In diesem Schritt berechnen wir die Größe der Penumbra.

Für die Berechnung benötigt man die folgenden drei Werte:

1. Abstand des Blockerpixels zur Lichtquelle
2. Abstand des Empfängerpixels zur Lichtquelle
3. Größe der Lichtfläche

Hat man all diese Werte, kann die Penumbra nach der folgenden Formel berechnet werden:

$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

$w_{Penumbra}$ ist dabei die Größe der Penumbra, w_{Light} die Größe der Lichtquelle, $d_{Blocker}$ und $d_{Receiver}$ der Abstand des Blocker- bzw. Empfängerpixels zur Lichtquelle. Die folgende Skizze verdeutlicht dies.

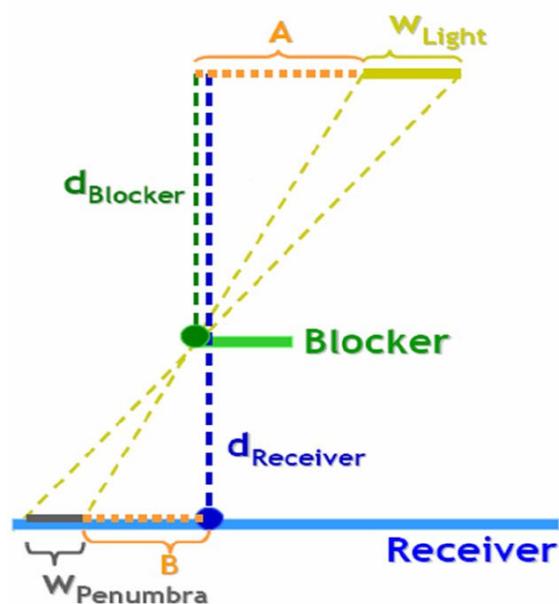


Bild 39 – Bestimmung der Penumbra [Q14]

Die genannte Formel basiert auf der Grundlage, dass Blocker, Empfänger und Lichtquelle parallel zueinander sind.

Die Größe der Lichtquelle wird dabei vom Benutzer festgelegt und dem Programm mitgeteilt. Den Abstand des Empfängerpunktes zur Lichtquelle ermitteln wir wie beim Standard-Shadow-Mapping durch den Z-Wert des aktuellen Pixels, transformiert in Lichtkoordinaten. Der Abstand des Blockers zur Lichtquelle wurde im vorherigen Schritt berechnet.

8.2.3 Variabler Percentage Closer Filter

Im letzten Schritt wenden wir einen variablen PCF-Filter für die Darstellung der weichen Schattenkanten an. Die Filtergröße wird dabei durch die Größe der im vorherigen Schritt berechneten Penumbra festgelegt. Hier verwenden wir den im vorigen Kapitel vorgestellten reduzierten PCF-Filter mit einer kleinen Modifikation. Während in dem dort vorgestellten PCF-Filter die Suchreichweite vorher vom Programm festgelegt wurde, wählen wir diese beim Perspektivischen-PCF-Filter von Pixel zu Pixel unterschiedlich groß. Für die Suchreichweite verwenden wir den im vorigen Abschnitt errechneten Wert für die Penumbra.

Aus Performancegründen legen wir auch bei diesem Filter eine maximale Anzahl an Samples pro Zeile fest, um Schwankungen der Framerate zu vermeiden. Durch diese Maßnahme müssen wir allerdings die maximale Ausdehnung der Penumbra begrenzen, um in jeder Situation eine akzeptable Schattenqualität zu garantieren.

8.3 Qualitätsverbesserung

Da dieses Schattenverfahren auf der selben Grundlage funktioniert, wie der im vorigen Kapitel vorgestellte PCF-Filter, können hier die selben Maßnahmen für die Qualitätsverbesserung Anwendung finden.

8.4 Zusammenfassung

Das PPCF-Shadow-Mapping ist ein Verfahren, um perspektivisch korrekte Schlagschatten per Shadow Mapping darstellen zu können. Die perspektivisch korrekte Penumbra wird dafür für jedes Pixel einzeln berechnet und die Suchreichweite des PCF-Filters am Ende entsprechend angepasst. Da dieses Verfahren zwei Samplingschritte durchführt - bei der Blockersuche und beim PCF-Filter am Ende -, ist die Samplingrate pro Pixel recht hoch. In [Q15] werden für eine akzeptable Framerate und Qualität für die Blockersuche sechs, für den PCF-Filter acht Samples pro Zeile vorgeschlagen. Dadurch kommt man auf 36 Samples für die Blockersuche sowie 64 für den PCF-Filter. Zusammen ergeben sich so 100 Samples, die pro Pixel betrachtet werden müssen. Dadurch ist dieses Verfahren das Aufwendigste von den hier vorgestellten. Im Gegenzug erzeugt es Schatten, die nah an der Realität liegen und sehr authentisch wirken. Mit der Wahl geeigneter Filtermethoden kann die Qualität der Schatten durch die Steigerung der maximal möglichen Penumbra weiter verbessert werden.

9.0 Implementierung

Das folgende Kapitel geht auf die Implementierung des Shadow Viewports und seiner einzelnen Komponenten ein. Dabei erläutere ich zunächst den grundlegenden Aufbau des Shadow Viewports, bevor ich näher auf die einzelnen Komponenten eingehe.

9.1 Shader in OpenSG

Für die implementierten Schattenverfahren für harte Schatten konnte die Fixed-Funktion-Pipeline benutzt werden, aber die Berechnung und Darstellung der Soft-Shadows musste per Vertex und Fragment Shader realisiert werden. Shader werden in OpenSG in *SHLChunks* geladen und dem Material des jeweiligen Objekts hinzugefügt. Dabei traten bei der Umsetzung Probleme auf, auf die ich im folgenden Abschnitt eingehe. Gleichzeitig stelle ich die in der Implementierung praktizierten Lösungen vor.

9.1.1 Probleme und Lösungen

Eine der optionalen Anforderungen an den Shadow Viewport war, dass die Darstellung von Schatten mit jeder Szene realisierbar sein soll, auch wenn diese bereits Shader benutzt. Der Benutzer sollte dafür keine Anpassungen in den eigenen Shadern vornehmen müssen. Das Vorhaben gestaltete sich zunächst schwierig, da hier zwei Probleme auftraten. Das erste Problem besteht darin, dass es OpenSG-Intern nicht ohne weiteres möglich ist, in einen vorhandenen Szenegraphen nachträglich Shader einzufügen. Das Problem liegt darin, dass SHLChunks als Material der entsprechenden Objekte gesetzt werden. Beim Erstellen eines Szenegraphen stellt das kein Problem dar. Da der Shadow Viewport aber einen fertigen Szenegraphen zum Rendern übergeben bekommt, ist es nicht ohne weiteres möglich, Shader nachträglich dem Geometriematerial hinzuzufügen.

Eine Möglichkeit bestand darin, den kompletten Szenegraphen nach Geometrienoten zu durchsuchen und deren Material durch die Schattenshader zu ersetzen. Ganz abgesehen davon, dass dieses Vorgehen wenig performant ist, gehen dadurch nicht nur die ursprünglichen Shader sowie Materialeigenschaften verloren, sondern auch das Entfernen der neu zugefügten Shader und die Wiederherstellung des ursprünglichen Materialparameter erweisen sich als äußerst schwierig.

Daher musste eine andere Möglichkeit gefunden werden. Dazu wurde die Baumstruktur von Szenegraphen genutzt. In OpenSG gibt es sogenannte MaterialGroups. Wie normale Materialparameter können auch MaterialGroups mit Materialeigenschaften versorgt und Objekten hinzugefügt werden. Der Unterschied zu normalen Material Chunks liegt darin, dass dieses Material für alle im Baum folgenden Knoten verwendet wird. Enthält ein Objekt bereits Materialparameter, werden diese ignoriert. Wenn wir diese Material Group als neuen Root-Knoten unseres Szenegraphen setzen und als Kind den ursprünglichen Root-Knoten zufügen, können wir Shader in die Szene einfügen, die für sämtliche Objekte genutzt werden. Nachdem alle Berechnungen durchgeführt sind, löschen wir den MaterialGroup-Knoten und stellen den

ursprünglichen Root-Knoten wieder her. Dadurch ist es ohne weiteres möglich, schnell und effektiv Shader für einen vorhandenen Szenegraphen zu verwenden.

Das zweite Problem liegt darin, dass vorhandene Shader nicht überschrieben werden sollen. Bei dem genannten Vorgehen ist das aber der Fall. Die Lösung des Problems gestaltet sich zum Glück weniger schwierig. Dazu wird die Szene vor der Manipulation gerendert und in einer Textur gespeichert, die wir *Color Map* nennen. Sie enthält jetzt die gerenderte Szene mit allen vorhandenen Shadereffekten. Danach fügen wir dem Szenegraphen die neuen Shader zu und alle erforderlichen Berechnungen werden durchgeführt. Statt die berechneten Schatten jetzt direkt zu verwenden, werden sie ebenfalls in einer Textur gespeichert, die ich *Shadow Factor Map* nenne. Die Shadow Factor Map enthält jetzt an den Pixelpositionen, an denen später Schatten in die Szene kommen, einen Faktor der kleiner 1 ist. Die Shadow Factor Map habe ich bereits in Kapitel 7 vorgestellt.

Sind alle Schattenfaktoren berechnet, multipliziert man diese Textur mit der Color Map, wodurch die Stellen, an denen sich Schatten befinden, abgedunkelt werden. Weil man die Color Map mit allen vorhandenen Shadern gerendert hat, enthält sie auch die jeweiligen Effekte.

9.2 Der Shadow Viewport

Die folgende Grafik zeigt den Aufbau des Standard-Viewports, wie er momentan in OpenGL implementiert ist.

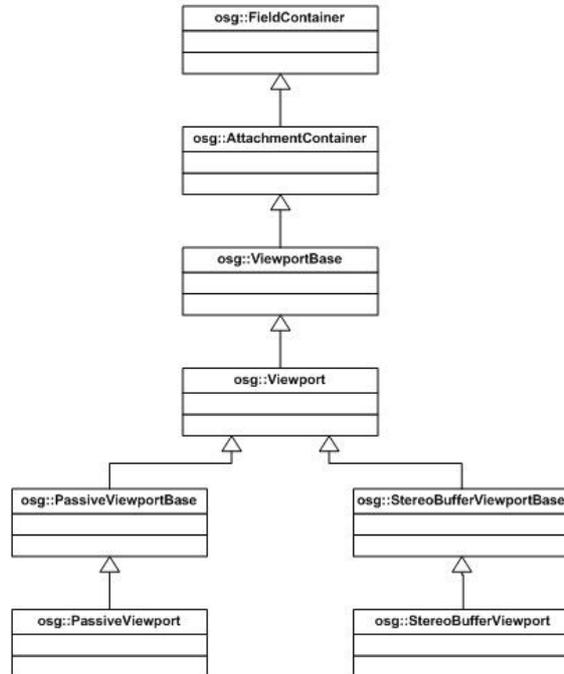


Bild 40 – Standard-Viewport

Wie man sieht, sind zwei Klassen von der Klasse Viewport abgeleitet, der `PassiveViewport` und der `StereoBufferViewport`. Diese stellen eine Erweiterung des Standard-Viewports dar. Den `StereoBufferViewport` benutzt man beispielsweise für die Darstellung einer Szene als Stereoprojektion.

Der in dieser Arbeit vorgestellte Viewport soll den Standard-Viewport einmal komplett ersetzen. Die folgende Grafik zeigt den Aufbau des Shadow-Viewports, wie er momentan implementiert ist.

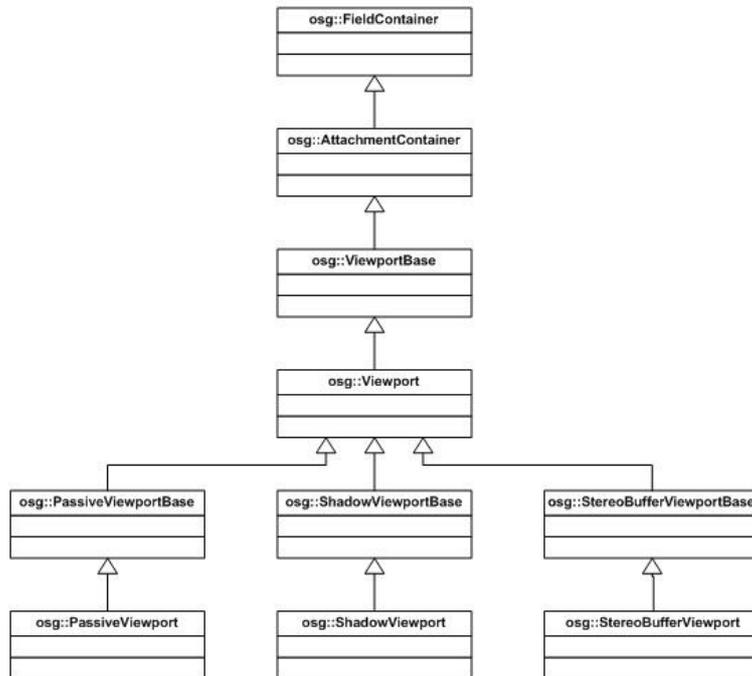


Bild 41 – Shadow-Viewport

Man sieht, dass der Shadow-Viewport momentan noch vom Viewport selbst abgeleitet ist und somit auch nur eine Erweiterung darstellt. Das soll jedoch in Zukunft geändert werden. Alle Module des bisherigen Viewports sollen dann vom Shadow-Viewport abgeleitet sein.

Ein wesentlicher Aspekt des Shadow Viewports sollte dessen Erweiterbarkeit darstellen. Neue Schattenverfahren sollten problemlos in den Shadow-Viewport integrierbar sein. Aus diesem Grund wurde er modular aufgebaut. Jedes implementierte Schattenverfahren ist in einem eigenen Modul integriert. Die folgende Grafik zeigt den Aufbau des Shadow-Viewports.

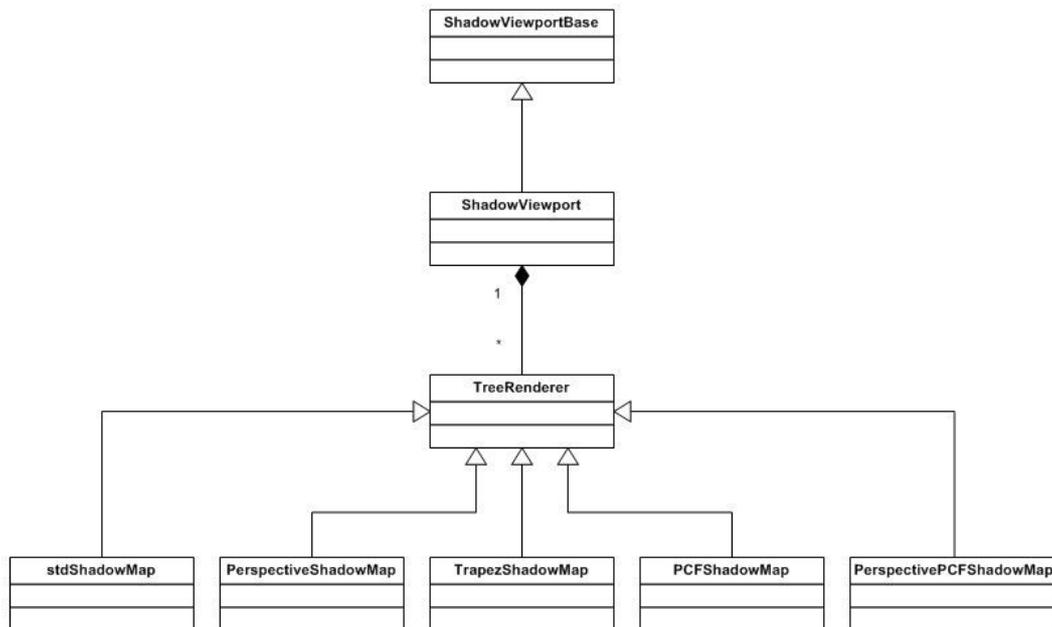


Bild 42 – Aufbau Shadow-Viewport

Wie man anhand des Klassendiagramms sieht, gibt es zur Zeit fünf verschiedene Schattenverfahren im Shadow-Viewport. Durch den modularen Aufbau ist es jederzeit möglich, weitere Module zu integrieren.

Je nach dem, welches Schattenverfahren man für die Darstellung auswählt, wird vom Shadow-Viewport ein Objekt der jeweiligen Klasse erzeugt. Gleichzeitig übergibt der Shadow Viewport diesem Objekt einen Zeiger auf sich selbst. Da die Klasse Treerenderer nicht vom Shadow-Viewport abgeleitet ist, wird diese Übergabe nötig, um von den einzelnen Modulen später auf die Daten der ShadowViewport-Klasse und deren FieldContainer zugreifen zu können.

Applikationen, die den Shadow-Viewport verwenden, steuern diesen durch die in der Klasse ShadowViewportBase zur Verfügung gestellten Variablen. Der FieldContainer dieser Klasse bildet demnach das Interface für den Shadow-Viewport, über den eine Applikation mit ihm kommunizieren kann. Variablen und Methoden der Klasse ShadowViewport selbst sind von außen nicht sichtbar und somit auch nicht veränderbar. Die Variablen der Base-Klasse stelle ich im Folgenden vor und beschreibe deren Funktion.

ShadowViewportBase
<pre> -offBias : Real32 = 4 -offFactor : Real32 = 10 -sceneRoot : NodePtr -mapSize : UInt32 = 512 -shadowColor : Color4f = (0,0,0,1) -shadowMode : UInt32 = 0 -range : Real32 = 0.3 -qualityMode : bool = false -shadowOn : bool = true -lightNodes : <vector>NodePtr -excludeNodes : <vector>NodePtr </pre>
<pre> +get/setOffBias() +get/setOffFactor() +get/setShadowColor() +...() </pre>

Listing 01 – Klasse *ShadowViewportBase*

Man kann erkennen, dass einige Attribute bereits mit Standardwerten versehen sind. Diese müssen daher nicht unbedingt von einer Applikation gesetzt werden. Nur Attribute ohne Vorbelegung müssen zwangsweise übergeben werden, um eine Funktionalität zu gewährleisten. Viele der Attribute erklären sich selbst, dennoch wird vollständigkeitshalber kurz auf jedes einzelne eingegangen. Die genaue Auswirkung der einzelnen Variablen auf die unterschiedlichen Schattenverfahren beschreibt Kapitel 9.3.

Die beiden Real32 Variablen *offBias* und *offFactor* benutzt man für die Wahl des Polygon-Offsets, durch das man fehlerhafte Schatten vermeiden kann.

Den Root-Knoten der zu rendernden Szene muss man dem Attribut *sceneRoot* übergeben. Ohne diese Angabe kann der Viewport nicht wissen, was er rendern soll.

shadowColor enthält die Schattenfarbe, während man mit dem Attribut *shadowMode* ein Schattenverfahren für die Darstellung wählen kann. Hier kann man sowohl Zahlen als auch enums angeben. Eine Liste der enums finden Sie in Kapitel 9.3.

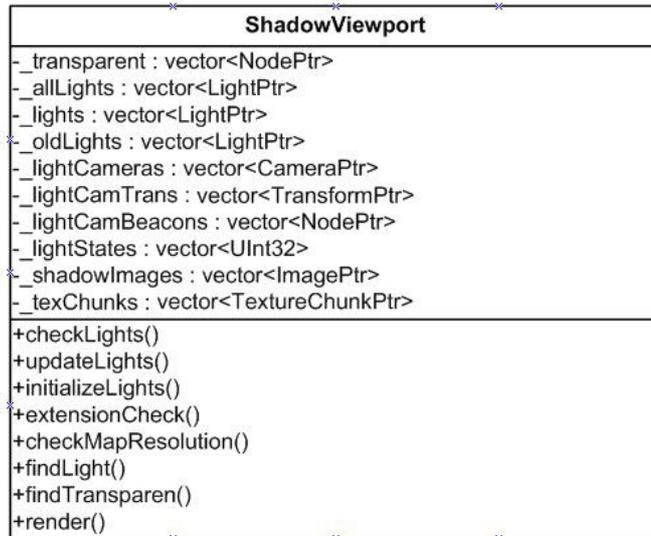
Das Attribut *range* ist nur im Modul *TrapezShadowMap* von Bedeutung. Man verwendet ihn dort zur Definition der Focus Region.

Die beiden bool_Attribute *shadowOn* und *qualityMode* geben an, ob man Schatten darstellen möchte und ob dafür eine besonders hohe Schattenqualität auf Kosten höherer Rechenleistung Anwendung finden soll.

Dem Vektor *lightNodes* kann man die Lichtquellen übergeben, die für die Schattendarstellung verwendet werden sollen. Auch wenn dieses Attribut keine Default-Werte hat, ist die Übergabe von Lichtknoten nicht zwingend notwendig. Werden keine Knoten übergeben, wird der Szenegraph traversiert und alle gefundenen Lichtknoten für die Schattendarstellung verwendet. Dem Vektor *excludeNodes* können Knoten des Szenegraphen übergeben werden, die keine Schatten werfen sollen.

Die Klasse `ShadowViewportBase` bietet für jede der genannten Variablen passende `get/set` Methoden, um deren Werte zu verändern. Allgemein besteht deren Name aus dem Prefix `get` oder `set`, gefolgt vom Namen des gewünschten Attributs. Zum Ändern des Attributs `mapSize` steht demnach die Methode `setMapSize(UInt32 size)` zur Verfügung.

Kommen wir zur `ShadowViewport`-Klasse. Das folgende Klassendiagramm zeigt deren Aufbau:



Listing 02 – Klasse `ShadowViewport`

Wie man sieht, enthält diese Klasse viele Methoden und Attribute, die etwas mit Licht zu tun haben. Dies ist auch eine der Hauptaufgaben dieser Klasse. Anhand eines Ausschnitts der `render()`-Methode, die von `OpenSG` für jedes Frame aufgerufen wird, lässt sich die Funktion dieser Klasse am besten beschreiben.

```

void ShadowViewport::render(RenderActionBase* action)
{
    ...
    if(!extensionCheck())
    {
        SWARNING << "No Shadowmap-Extensions available!"
        << endLog;
    }
    else
    {
        checkMapResolution();
        checkLights(action);
    }
    ...
    treeRenderer->render(action);
}
  
```

Listing 03 – Ausschnitt der `render()`-Methode der Klasse `ShadowViewport`

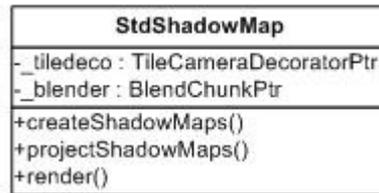
Anhand der Methode *extensionCheck()* überprüft man, ob die erforderlichen GL-Extensions vorhanden sind und von der Hardware unterstützt werden. Ist dies der Fall, legt man die Auflösung der Shadow Map durch den Aufruf der Methode *checkMapResolution()* fest. Da die Shadow Map eine Zweierpotenzauflösung haben muss, wird hier die größtmögliche Auflösung gewählt, die noch innerhalb der Fenstergröße darstellbar ist. Wird die Szene beispielsweise in einer Auflösung von 640x640 Pixel gerendert, legt man die Größe der Shadow Map auf 512x512 Pixel fest. Vom Benutzer können natürlich auch größere Auflösungen gewählt werden, aber diese Methode verhindert, dass man unnötig kleine Auflösungen wählt, die zu sehr schlechten Ergebnissen führen.

Im Anschluss behandelt man die Lichtquellen. Durch den Aufruf der *checkLights()*-Methode wird der Szenegraph mit der *findLight()*-Methode traversiert und alle gefundenen Lichtquellen in den Vektor *_lights* geschrieben. Hat man alle Lichtquellen gefunden, erzeugt die Methode *initializeLights()* für jede Lichtquelle eine Kamera, die im Kameravektor *_lightCameras* gespeichert wird. Zusätzlich erzeugt sie eine Textur für die Shadow Map. Zuletzt erfolgt der Aufruf *updateLights()*. Hier versorgt man die entsprechenden Lichtkamas mit den nötigen Parametern. Will man die Szene jetzt aus Sicht einer Lichtquelle rendern, z.B. um die entsprechende Shadow Map zu generieren, muss nur noch die zur Lichtquelle gehörende Lichtkamera aus dem Kameravektor *_lightCameras* gewählt werden.

Nach der Behandlung der Lichtquellen wird die *render()*-Methode des aktuell aktiven Moduls für die Schattengenerierung aufgerufen. Alles was ab hier passiert, gehört zur Aufgabe der einzelnen Module. Dadurch ist eine größtmögliche Freiheit für die Implementierung von verschiedenen Schattenverfahren gewährleistet. In den folgenden Kapiteln gehe ich auf die einzelnen Module ein. Dabei zähle ich nicht alle Attribute und Methoden der einzelnen Module auf, sondern nur die, die für das Verständnis wichtig sind. Die Punktkette „...“ bedeutet, dass die Klasse weitere Attribute oder Methoden enthält, die für interne Berechnungen verwendet werden.

9.2.1 Das Modul StdShadowMap

Das StdShadowMap-Modul ist folgendermaßen aufgebaut:



Listing 04 – Klasse StdShadowMap

Das Modul hat nur zwei Attribute. `_tiledeco` ist ein `TileCameraDecorator`, der in Kapitel 2 bereits kurz vorgestellt wurde. Er ist dafür zuständig, die Szene in einzelne Kacheln aufzuteilen, falls die Shadow Map mit einer höheren Auflösung als der Bildschirmauflösung gerendert werden soll. Das Bild wird sozusagen in einzelne Kacheln unterteilt, die man am Ende wieder zusammenfügt. Ohne diesen KameraDecorator entstehen in der Shadow Map schwarze Bereiche in den Regionen, die über die Bildschirmauflösung hinausgehen, weil in diesen Bereichen nichts im Color Buffer steht.

Das `BlendChunk`-Attribut `_blender` wird für das Rendern der Szene ohne Schatten benötigt. Schattierte Bereiche werden durch das aktivieren des Blenders ausgeblendet.

Durch einen Blick in die `render()`-Methode lässt sich der Renderablauf am Besten erklären.

```
void StdShadowMap::render(RenderActionBase* action)
{
    ...
    createShadowMaps(action);

    if(      !shadowVP->_lights.empty() &&
           !shadowVP->_lightCameras.empty() )
    {
        projectShadowMaps(action);
    }

    else
    {
        FDEBUG(("Rendering without Shadows\n"));
        shadowVP->Viewport::render(action);
    }

    for(UInt16 i=0; i < shadowVP->getForegrounds().size(); ++i)
    {
        shadowVP->getForegrounds(i)->draw(action, shadowVP);
    }
}
```

Listing 05 – Ausschnitt der `render()`-Methode der Klasse `StdShadowMap`

Die Methode *createShadowMaps()* erzeugt für jede Lichtquelle die passende Shadow Map und speichert sie im TextureChunk-Vektor *_texChunks* der ShadowViewport-Klasse. Die passenden Lichtkameran wurden im vorigen Schritt bereits vom Shadow-Viewport erzeugt und stehen im Camera-Vektor *_lightCameras* zur Verfügung.

Nachdem alle erforderlichen Shadow Maps erzeugt sind, zeichnet man die Szene in der Methode *projectShadowMaps()*. Hat man dem Shadow-Viewport keine Lichtknoten übergeben und wurden auch beim traversieren des Szenegraphen keine Lichtknoten gefunden, ruft diese Methode die *render()*-Methode des Standard-Viewports auf, da man ohne Lichtquellen keine Schatten darstellen muss. Da der Standard-Viewport in Zukunft komplett durch den Shadow-Viewport ersetzt werden soll, muss man ihn spätestens dann durch eine eigene *render()*-Methode erweitern. Diese soll die Szene dann ohne Schatten rendern.

Zum Schluss rendert man alle vorhandenen Vordergründe auf die Szene, bevor das fertige Bild am Bildschirm ausgegeben wird.

9.2.2 Das Modul TrapezShadowMap

Das folgende Bild zeigt den Aufbau dieses Moduls:



Listing 06 – Klasse TrapezShadowMap

Hier haben die Attribute `_tiledeco` und `_blender` dieselben Funktionen wie im `StdShadowMap` Modul. Neu ist die `MatrixCamera` `_matrixCam` sowie der Matrizenvektor `_NT`. Die Matrix Kamera sollte aus Kapitel 2 bekannt sein. Sie beschreibt eine Kamera, wie OpenGL, durch eine Projektions- und Modelviewmatrix. Im Matrizenvektor `_NT` speichert man die vom Trapez-Shadow-Mapping erzeugten Projektionsmatrizen für die perspektivische Verzerrung für alle Lichtquellen.

Auch hier lässt sich die Funktionsweise des Moduls am Besten anhand der `render()`-Methode erklären.

```
void TrapezShadowMap::render(RenderActionBase* action)
{
    ...
    Pnt3f t_0,t_1,t_2,t_3;

    for (int i=0; i<shadowVP->_lights.size(); i++)
    {
        calculateTrapezoid(action, t_0,t_1,t_2,t_3, shadowVP->_lights[i], i);
        Matrix N_T;
        map_Trapezoid_To_Square(N_T,t_0,t_1,t_2,t_3);
        _NT.push_back(N_T);
    }

    createShadowMaps(action);

    if ( !shadowVP->_lights.empty() && !shadowVP->_lightCameras.empty() )
    {
        projectShadowMaps(action);
    }
    else
    {
        FDEBUG(("Rendering without Shadows\n"));
        shadowVP->Viewport::render(action);
    }

    for(UInt16 i=0; i < shadowVP->getForegrounds().size(); ++i)
    {
        shadowVP->getForegrounds(i)->draw(action, shadowVP);
    }

    _NT.clear();
}
```

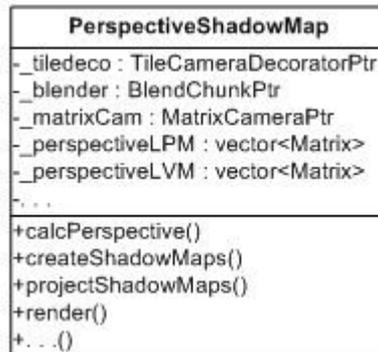
Listing 07 – Ausschnitt der `render()`-Methode der Klasse `TrapezShadowMap`

Zunächst erzeugt man vier Punkte, die man für jede Lichtquelle der Methode *calculateTrapezoid()* per Call by Reference übergibt. Sie berechnet aus der aktuellen Lichtkamera die Trapezpunkte, wie es im Kapitel 6.2 beschrieben wurde. Die Methode *map_Trapezoid_To_Square()* erstellt aus diesen Punkten die Projektionsmatrix, die das Trapez auf den Einheitswürfel transformiert. Für jede Lichtquelle schreibt man die entsprechende Matrix in den Matrizenvektor *_NT*. Anschließend verfährt man wie im Modul *StdShadowMap*. Für alle Lichtquellen werden die Shadow Maps erstellt, die Szene wird mit den Schatten und Foregrounds gezeichnet und schließlich am Bildschirm ausgegeben. Der einzige Unterschied liegt darin, dass man die neu berechneten Projektionsmatrizen vor dem Erstellen der Shadow Maps mit der Standard-Projektionsmatrix multipliziert. Aus diesem Grund wird auch die Matrix Kamera benötigt. Wir konfigurieren die neue Lichtkamera nicht mit den üblichen Parametern, sondern direkt über die Modelview- und Projektionsmatrix.

Nachdem alle Berechnungen durchgeführt sind und man die fertige Szene am Bildschirm ausgegeben hat, wird der Matrizenvektor *_NT* wieder gelöscht.

9.2.3 Das Modul PerspectiveShadowMap

Auch hier zunächst der Aufbau dieses Moduls :



Listing 08 – Klasse PerspectiveShadowMap

Die ersten drei Attribute *_tiledeco*, *_blender* und *_matrixCam* haben auch hier dieselbe Funktion wie im vorherigen Modul. Neu sind die beiden Matrizenvektoren *_perspectiveLPM* und *_perspectiveLVM*. Wurde im Modul TrapezShadowMap nur eine neue Projektionsmatrix für die Lichtkamera berechnet, erzeugen wir hier sowohl die Projektions- als auch Modelviewmatrix neu und speichern sie für jede Lichtquelle in dem entsprechenden Matrizenvektor. Die *render()*-Methode unterscheidet sich auch nicht sehr von der des TrapezShadowMap-Moduls.

```
void PerspectiveShadowMap::render(RenderActionBase* action)
{
    ...
    for (int i=0; i<shadowVP->_lights.size(); i++)
    {
        Matrix _LPM, _LVM;
        calcPerspective(_LPM, _LVM, i);
        _perspectiveLPM.push_back(_LPM);
        _perspectiveLVM.push_back(_LVM);
    }

    createShadowMaps(action);

    if(!shadowVP->_lights.empty() && !shadowVP->_lightCameras.empty())
    {
        projectShadowMaps(action);
    }
    else
    {
        FDEBUG(("Rendering without Shadows\n"));
        shadowVP->Viewport::render(action);
    }

    for(UInt16 i=0; i < shadowVP->getForegrounds().size(); ++i)
    {
        shadowVP->getForegrounds(i)->draw(action, shadowVP);
    }

    _perspectiveLPM.clear();
    _perspectiveLVM.clear();
}
```

Listing 09 – Ausschnitt der *render()*-Methode der Klasse PerspectiveShadowMap

Die beiden neuen Matrizen berechnet man in der Methode *calcPerspective()* für jede Lichtquelle und speichert sie in dem jeweiligen Matrizenvektor. Beim Erstellen der Shadow Maps werden die neuen Matrizen als Parameter für die Lichtkamera genutzt. Danach rendert man wie gehabt, bevor man am Ende die beiden Matrizenvektoren wieder löscht.

9.2.4 Das Modul PCFShadowMap

Da es sich bei diesem Verfahren um ein Soft Shadow Verfahren handelt, kommen hier Shader zum Einsatz. Aus diesem Grund führen wir das Rendern der Szene nach dem in Kapitel 9.1 vorgestelltem Muster durch. Das Bild zeigt den Aufbau der Klasse.



Listing 10 – Klasse PCFShadowMap

Wie im vorigen Modul gibt es auch hier die Attribute `_tiledeco`, `_blender` und `_matrixCam`. Sie haben hier die selben Aufgaben wie in der vorigen Klasse auch.

`_colorMap` ist der TextureChunk, in dem man später die Color Map speichert. `_colorMapImage` ist der passende Image-Pointer dazu. Die Shadow Factor Map wird im TextureChunk `_shadowFactorMap` gespeichert, bei dem `_shadowFactorMapImage` der zugehörige Image Pointer ist.

Die restlichen Attribute beziehen sich auf die Shader und deren Benutzung. Für die Berechnung der PCF-Soft-Shadows verwendet man zwei Shader, welche in den Attributen `_shadowSHL` sowie `_combineSHL` enthalten sind. `_shadowCmat` und `_combineCmat` sind die ChunkMaterials, denen man die Shader zugefügt. Diese ChunkMaterials können jedem beliebigen Objekt als Material übergeben werden, so dass dort die geladenen Shader verwendet werden. Wegen den Problemen, die im Abschnitt 9.1.1 beschrieben wurden, fügt man diese nicht den Objekten direkt hinzu, sondern bindet sie an eine MaterialGroup. Diese MaterialGroups fügt man den Knoten `_shadowRoot` bzw. `_combineRoot` zu, die man dann als neue Root-Knoten vor den Root-Knoten des Szenegraph hängt.

Die *render()*-Funktion gibt Aufschluss über den Ablauf der Berechnungen.

```
void PCFShadowMap::render(RenderActionBase* action)
{
    ...
    createColorMap(action);
    createShadowMaps(action);

    for(int i = 0; i<shadowVP->_lights.size();i++)
    {
        createShadowFactorMap(action, i);
    }

    createCombineMap(action);

    drawTextureBox(action, _colorMap);
}
```

Die Methode *createColorMap()* rendert die Szene in eine Textur und erstellt so die Color Map. In *createShadowMaps()* erzeugt man wie gehabt die Shadow Maps für alle Lichtquellen.

Im folgenden Schritt erzeugt man die Shadow Factor Map durch die Methode *createShadowFactorMap()* und aktualisiert diese für jede Lichtquelle, bevor sie in der Methode *createCombineMap()* mit der Color Map verrechnet und wieder in der Textur *_colorMap* gespeichert wird. Als letztes gibt man durch die Methode *drawTextureBox()* die modifizierte Color Map auf dem Bildschirm aus.

Als Shadersprache habe ich GLSL benutzt, die verwendeten Shader können Sie im Anhang einsehen.

9.2.5 Das Modul PerspectivePCFShadowMap

Dieses Modul ist genauso aufgebaut, wie das PCFShadowMap-Modul. Sämtliche Attribute und Funktionen sind gleich, lediglich die Shader wurden modifiziert. Aus diesem Grund führe ich den Klassenaufbau hier nicht erneut auf.

Die GLSL-Shader dieses Moduls können Sie im Anhang einsehen.

9.3 Benutzung des Shadow Viewports

In diesem Abschnitt gehe ich auf die Verwendung des Shadow-Viewports in einem OpenSG-Programm ein. Ich setze dafür OpenSG Grundkenntnisse voraus, da ich nur auf die Einbindung des Shadow-Viewports eingehe, nicht aber auf das Erstellen einer Szene oder anderer benötigter Komponenten.

Zunächst binden wir die Headerdatei des Shadow-Viewports ein und deklarieren eine Variable.

```
#include <OSGShadowViewport.h>
...
ShadowViewportPtr svp;
```

Bei der Initialisierung erzeugen wir den Shadow-Viewport.

```
svp = ShadowViewport::create();
```

Das folgende Listing zeigt, wie man die Parameter des Shadow-Viewports manipulieren kann. Alle in diesem Listing aufgeführten Parameter müssen initialisiert werden, damit der Shadow-Viewport funktioniert.

```
beginEditCP( svp );
    svp->setBackground( bg );
    svp->setRoot( rootNode );
    svp->setSize( 0,0,1,1 );
endEditCP( svp );
```

Per *setBackground()* übergeben wir dem Viewport einen Background. Dies kann jeder beliebige in OpenSG zur Verfügung gestellte Background sein. Über *setRoot()* teilen wir dem Viewport mit, ab welchem Knoten er die Szene rendern soll. Dies ist normalerweise der Root-Knoten des Szenegraphen. Die Größe des Viewports geben wir über *setSize()* an. Die ersten beiden Parameter geben den X/Y Wert der linken unteren Ecke an, die letzten beiden die Koordinaten der oberen rechten Ecke. Werte zwischen Null und Eins beziehen sich auf die relative Fenstergröße, alles darüber gibt Pixelwerte an.

Im Folgenden führe ich alle Parameter auf, die man optional ändern kann.

```
BeginEditCP(svp);
    svp->setShadowMode( STD_SHADOW_MAP );
    svp->setOffBias( 4 );
    svp->setOffFactor( 10 );
    svp->setShadowColor( Color4f(0.1,0.1,0.1,1.0) );
    svp->setMapSize( 1024 );
    svp->setRange( 0.4 );
    svp->setShadowOn( true );
    svp->setQualityMode( true );
    svp->getLightNodes().addValue( lightNode );
    svp->getExcludeNodes().addValue( excludeNode );
EndEditCP(svp);
```

Durch den Parameter *setShadowMode* lässt sich das verwendete Schattenverfahren auswählen. Hier stehen die folgenden Parameter zur Auswahl:

1. NO_SHADOW
2. STD_SHADOW_MAP
3. TRAPEZ_SHADOW_MAP
4. PERSPECTIVE_SHADOW_MAP
5. PCF_SHADOW_MAP
6. PPCF_SHADOW_MAP

Über *setOffBias()* sowie *setOffFactor()* lässt sich das verwendete Polygon Offset ändern.

Mit *setShadowColor()* kann man einen Color4f-Wert übergeben, der als Schattenfarbe verwendet wird.

setMapSize() legt fest, wie hoch die Auflösung der zu verwendenden Shadow Map sein soll. Durch die verwendeten TileCameraDecorator in den Schattenmodulen kann die Auflösung auch höher sein als die Bildschirmauflösung.

Der Wert, der über *setRange()* übergeben wird, hat momentan nur Einfluss auf das Trapez-Shadow-Mapping. Man legt durch diesen Parameter fest, welcher Bereich der Szene von der Kameraposition auf die 80%-Linie projiziert wird. Dabei sind Werte größer Null und kleiner 0.8 zulässig, wobei ein Wert von 0.8 dem Ergebnis des Standard-Shadow-Mappings entspricht. In diesem Fall projiziert man die ersten 80% der Szene auf den Bereich der Shadow Map, der 80% der Auflösung zur Verfügung gestellt bekommt.

Die Schattendarstellung kann man über *setShadowOn()* aktivieren und deaktivieren, mit *setQualityMode()* kann man den Qualitätsmodus aktivieren. Beim TrapezShadowMap-Modul wird hier die Suche nach dem kleinsten und grössten Tiefenwert aktiviert, um die Base- und Top-Line optimal anzupassen. Dieses Verfahren benötigt jedoch einen zusätzlichen Renderingdurchlauf. Bei den Soft-Shadow-Verfahren wird durch die Aktivierung des Qualitätsmodus die maximale Penumbra erhöht, wodurch die Anzahl der genommenen Samples steigt.

Mit der Methode *svp->getLightNodes().addValue()* teilt dem Shadow Viewport mit, für welche Lichtquellen er Schatten erzeugen soll. Außerdem muss durch diese Übergabe nicht der Szenegraph nach Lichtquellen durchsucht werden, was Rechenzeit einspart.

Knoten, die nicht dargestellt werden sollen, können dem Shadow-Viewport per *getExcludeNodes().addValue()* übermittelt werden.

Als letztes muss man dem Fenster für die Ausgabe am Bildschirm noch mitteilen, dass es den Shadow-Viewport verwenden soll. Das geschieht mit dem folgenden Aufruf:

```
windowPointer->addPort (svp);
```

Ab jetzt übernimmt der Shadow-Viewport das Rendern der Szene.

10.0 Ergebnisse und Vergleich

In diesem Kapitel vergleiche ich die verschiedenen Schattenverfahren anhand von Screenshots miteinander und erörtere deren Vor- und Nachteile. Dabei gehe ich zunächst auf die Schattenqualität ein, bevor die Performance der Verfahren verglichen wird. Im Anschluss folgt eine kurze Zusammenfassung der gewonnenen Ergebnisse.

10.1 Schattenqualität harte Schatten

In diesem Abschnitt vergleiche ich die Schattenqualität der einzelnen Hard-Shadow-Verfahren in unterschiedlichen Situationen miteinander. Bei den harten Schatten vergleiche ich zunächst die Schattenqualität anhand verschieden hoch aufgelöster Shadow Maps, bevor ich auf die Unterschiede bei verschieden entfernt liegenden Schatten eingehe. Im Anschluss vergleiche ich die Schattenqualität aus verschiedenen Blickwinkeln relativ zur Lichtrichtung.

10.1.1 Verschiedene Auflösungen der Shadow Map

Gehen wir zunächst auf das Standard-Shadow-Mapping ein. Die folgende Bildreihe zeigt die Ergebnisse dieses Verfahrens mit unterschiedlichen Auflösungen der Shadow Map. Das Bild links oben zeigt den Bereich, über dem die Shadow Map erzeugt wird. Dieser entspricht der Sicht der Lichtquelle.

Standard-Shadow-Mapping

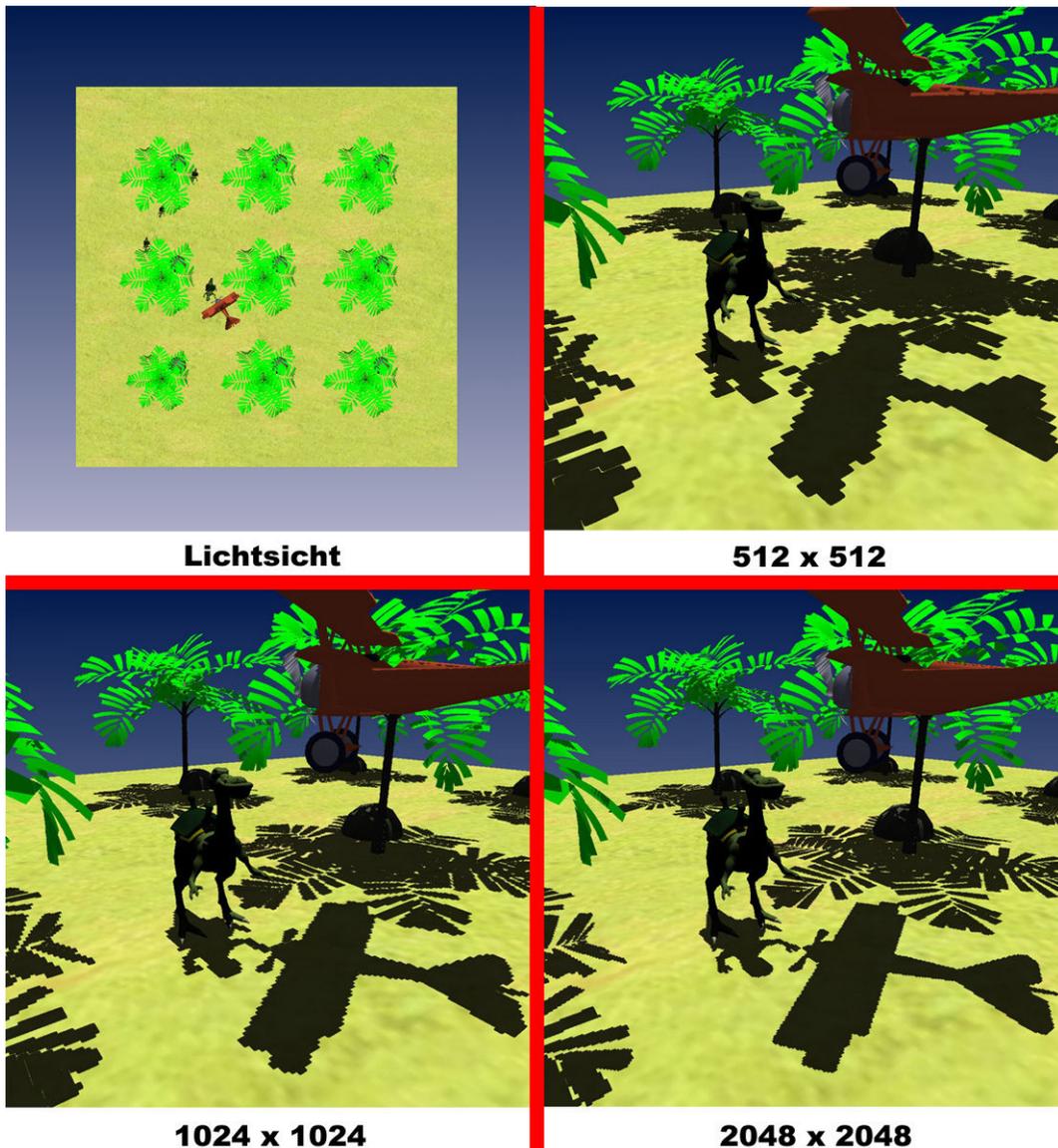


Bild 43 – Unterschiedliche Shadow-Map-Auflösungen beim Standard-Shadow-Mapping

Man kann erkennen, dass der Schatten bei allen Auflösungen starke Aliasingeffekte aufweist. Erst bei einer Auflösung von 2048x2048 Pixel ist die Qualität annehmbar.

Als nächstes sehen wir uns die Ergebnisse des Trapez Shadow Mappings an.

Trapez-Shadow-Mapping

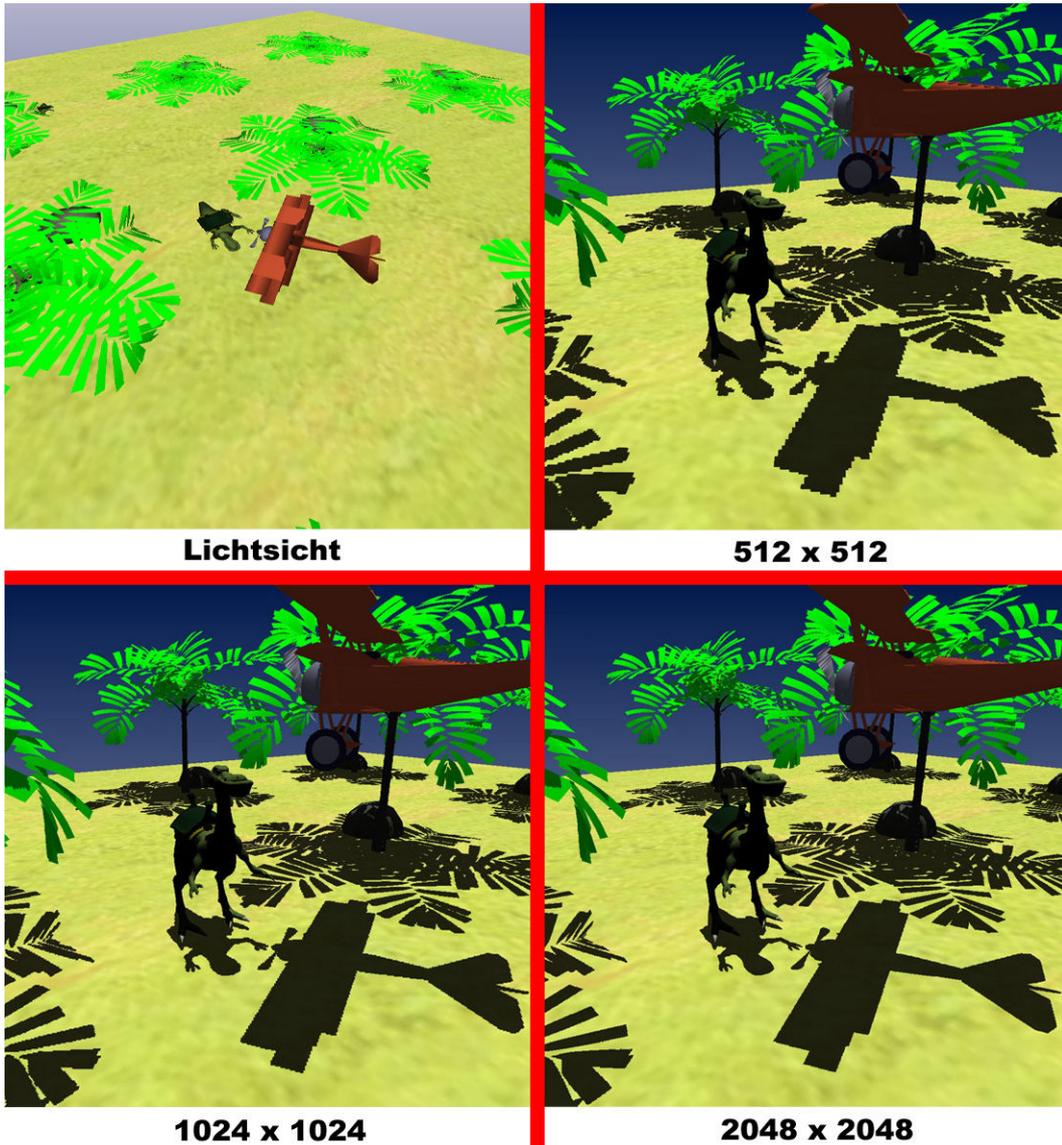


Bild 44 – Unterschiedliche Shadow-Map-Auflösungen beim Trapez-Shadow-Mapping

Schon beim ersten Hinsehen kann man das deutlich reduzierte Aliasing im Vergleich zum Standard-Shadow-Mapping erkennen. Sogar mit einer Auflösung von nur 512^2 Pixel der Shadow Map wird hier ein Ergebnis erzielt, welches qualitätsmäßig zwischen der 1024^2 und 2048^2 Shadow Map des Standard-Shadow-Mappings anzusiedeln ist. Schaut man sich die Sicht der Lichtquelle an, also den Bereich, über dem die Shadow Map erzeugt wird, erkennt man sofort, woher die Qualitätsverbesserung kommt. Der Dinosaurier und das Flugzeug, welche sich aus Kamerasicht am Nächsten an der Kamera befinden, sind hier deutlich größer als beispielsweise die Palmen im Hintergrund. Dies wird durch die neue Projektionsmatrix erreicht, die die Szene innerhalb des berechneten Trapez auf den Einheitswürfel projiziert.

Das dritte Implementierte Verfahren für harte Schatten ist das Lisp-Shadow-Mapping. Die Ergebnisse dieses Verfahrens sind im folgenden Bild zu sehen.

Lisp-Shadow-Mapping

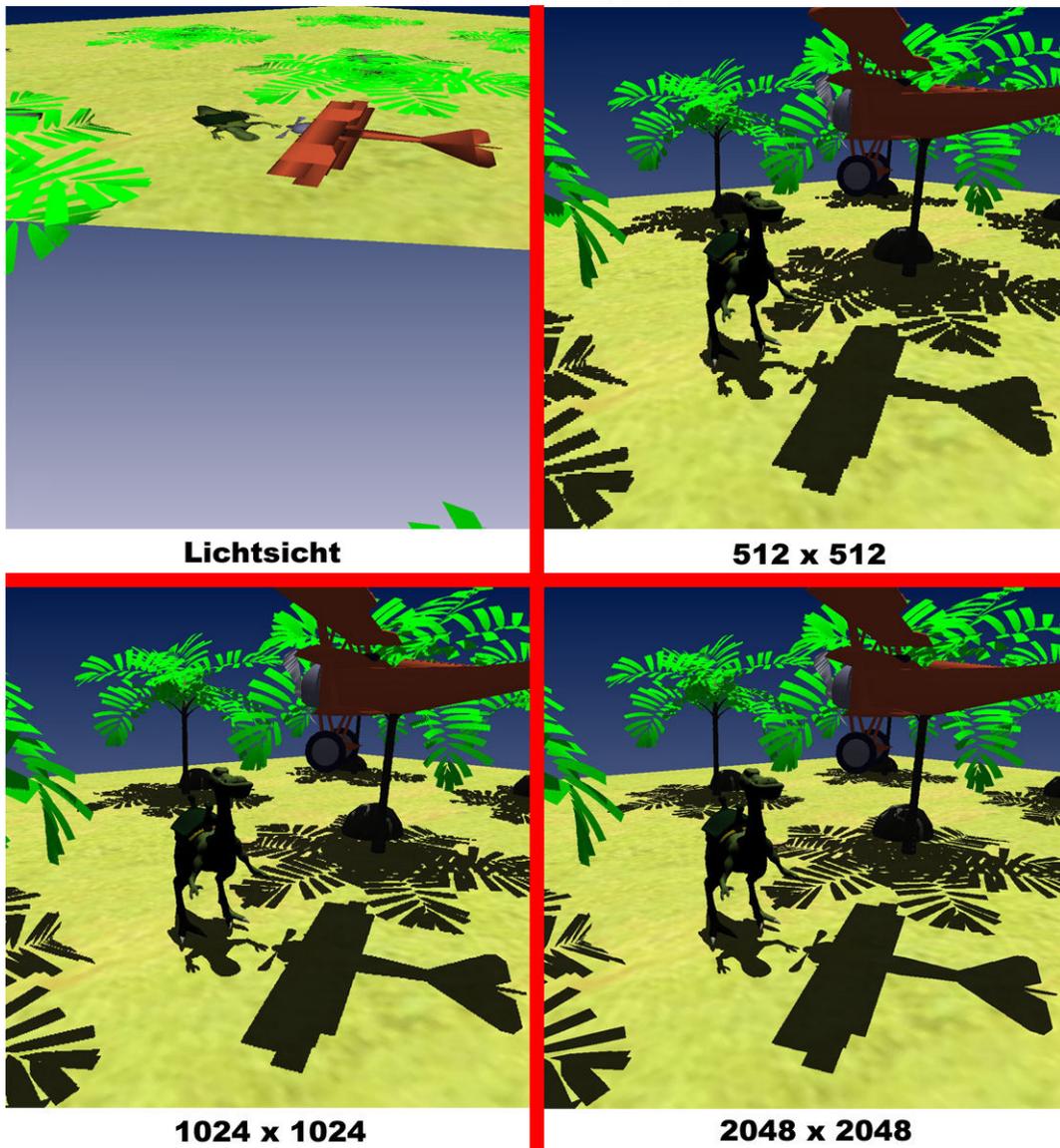


Bild 45 – Unterschiedliche Shadow-Map-Auflösungen beim Lisp-Shadow-Mapping

Auch die Ergebnisse des Lisp-Shadow-Mapping sehen deutlich besser aus, als die des Standard-Shadow-Mappings. Die Schattenqualität ist hier ebenfalls bei 512² Pixeln bereits mit der des Standard Shadow Mappings zwischen 1024² und 2048² vergleichbar. Die Ergebnisse scheinen sogar noch etwas besser zu sein als beim Trapez-Shadow-Mapping. Bei der Lichtsicht fällt auf, dass Objekte im Vordergrund noch etwas größer wirken als beim Trapez-Shadow-Mapping, wodurch nahe liegende Objekte eine noch höhere Auflösung zur Verfügung gestellt bekommen als beim Trapez-Shadow-Mapping. Das Aliasing naher Objekte wird dadurch noch weiter verringert. Die blaue Fläche in der Lichtsicht entsteht durch das Clippen des Kamera-View-Frustum mit der Szene. Der Bereich, in dem der Untergrund in Lichtsicht beginnt, stellt den Schnitt des Kamerafrustum mit dem Untergrund dar.

Aber welches Verfahren macht jetzt effektiv die besseren Schatten? Ist es wirklich von Vorteil im Vordergrund liegende Objekte perspektivisch immer mehr zu verzerren, damit sie immer größer werden? Zur Beantwortung dieser Frage schauen wir uns das Perspective-Shadow-Mapping an.

Perspective-Shadow-Mapping

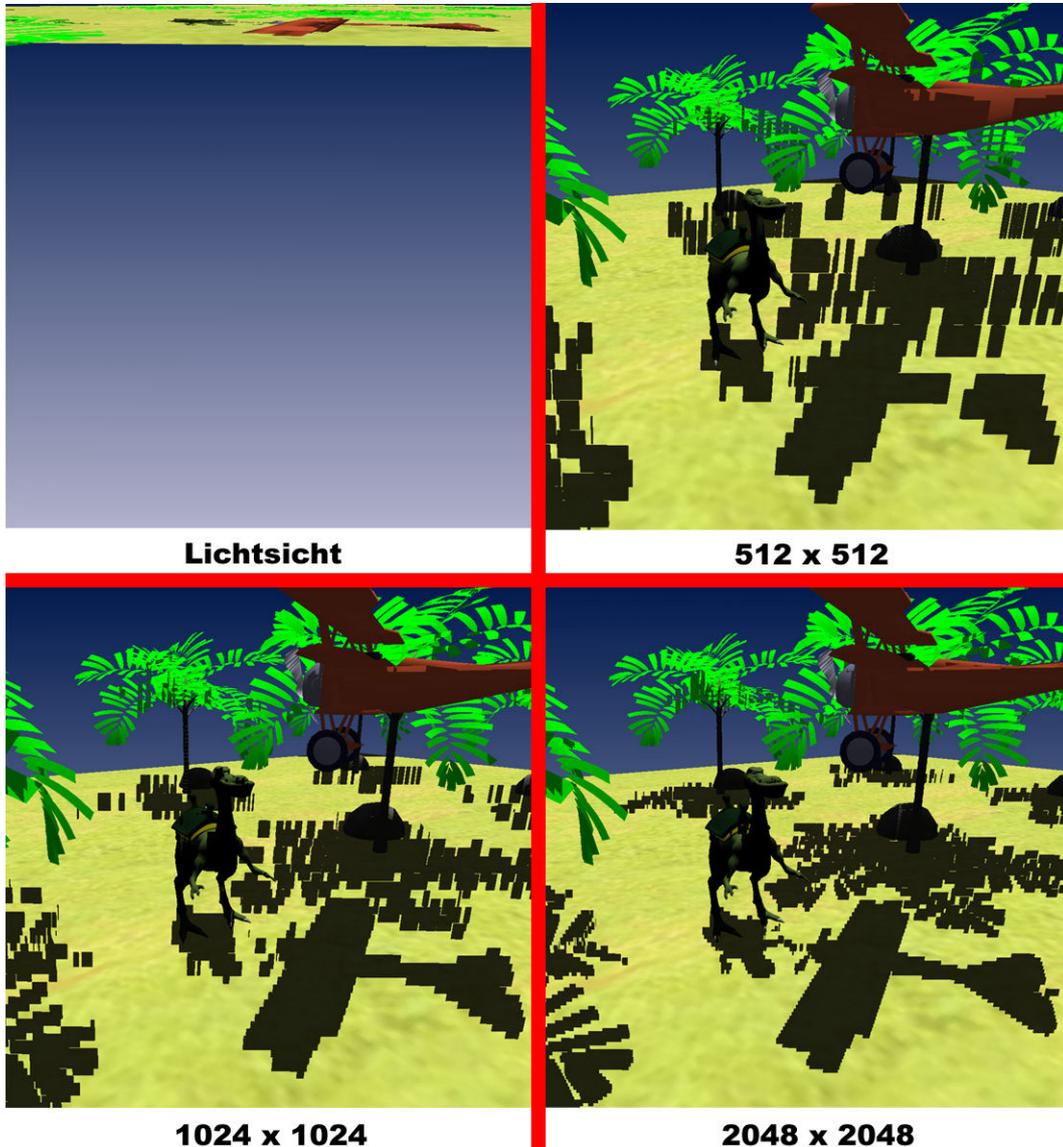


Bild 46– Unterschiedliche Shadow-Map-Auflösungen beim Perspective-Shadow-Mapping

Es fällt auf, dass die Qualität bei allen Shadow-Map-Auflösungen deutlich unter der des Standard-Shadow-Mapping liegt. Eine seltsame Erkenntnis, wo Perspective-Shadow-Mapping die Schattenqualität doch verbessern sollte. Schaut man sich die Lichtsicht an, kann man bereits erahnen, wie es zu diesem Qualitätsverlust kommt.

Für einen weiteren Vergleich betrachten wir im folgenden Kapitel verschiedene Bereiche des Bildes. Dabei gehen wir auch dieser Beobachtung auf den Grund.

10.1.2 Nah-, Mittel- und Fernbereich

Wir sehen uns in diesem Abschnitt die Schattenqualität in verschiedenen Bereichen des Bildes genauer an. Im vorigen Kapitel wurde mehr auf den Gesamteindruck des Bildes und weniger auf die Details geachtet. Diesen Details widmen wir uns in diesem Kapitel. Dazu vergrößern wir zu jedem Bild die Schattenkanten von drei Bereichen. Der erste Bereich liegt direkt vor der Kamera, der zweite etwa in der Mitte und der letzte am Ende der Szene. Für alle Bilder wurde eine Shadow Map Auflösung von 512^2 Pixel gewählt, um die Qualitätsunterschiede deutlich sichtbar zu machen. Da man normalerweise eine höhere Auflösung für die Shadow Map wählt, sehen die hier gezeigten Ergebnisse relativ schlecht aus. Beginnen wir wieder mit dem Standard Shadow Mapping.

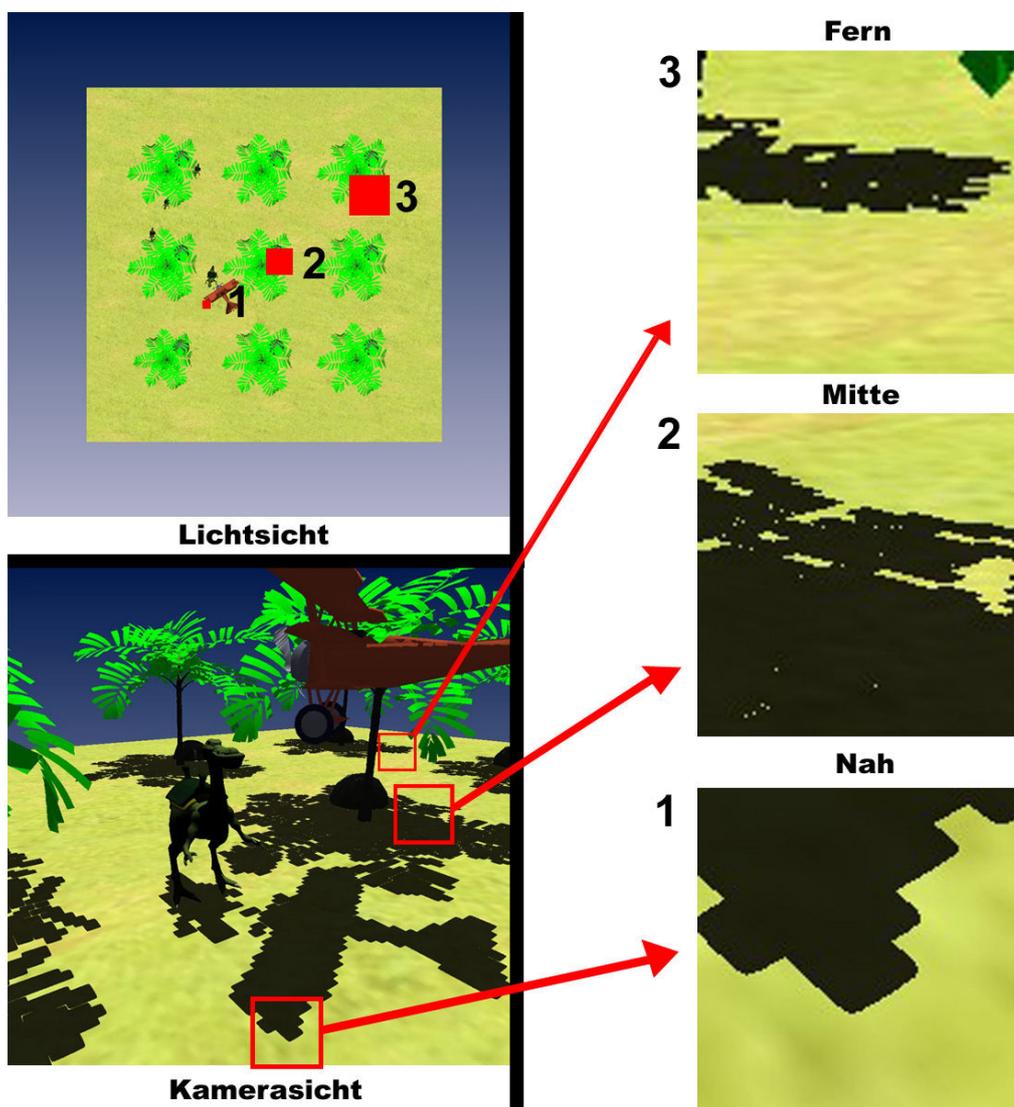


Bild 47 – Detailansicht beim Standard-Shadow-Mapping

In dem Bild habe ich die drei untersuchten Bereiche sowohl in der Kamera-, als auch der Lichtsicht markiert. Dabei fällt auf, dass alle drei untersuchten Bereiche aus Sicht des Lichtes unterschiedlich groß sind. Der Bereich nah an der Kamera erhält die geringste Auflösung, der hintere die größte, der mittlere etwa die Hälfte des hinteren. Berücksichtigt man die Perspektive der Kamera, so erhält jeder Bereich bezüglich seiner Gesamtfläche die gleiche Auflösung zugeteilt. Dies erkennt man daran, dass alle Palmen in der Lichtsicht gleich groß sind, egal an welcher Stelle sie stehen.

Anhand der Vergrößerungen sieht man, dass unter diesem Aspekt alle Schattenkanten in etwa gleich starkes Aliasing aufweisen. Natürlich wirken Schattenkanten, die nah an der Kamera liegen, deutlich gröber als solche, die weiter entfernt liegen.

Schauen wir uns die verschiedenen Bereiche beim Trapez-Shadow-Mapping an.

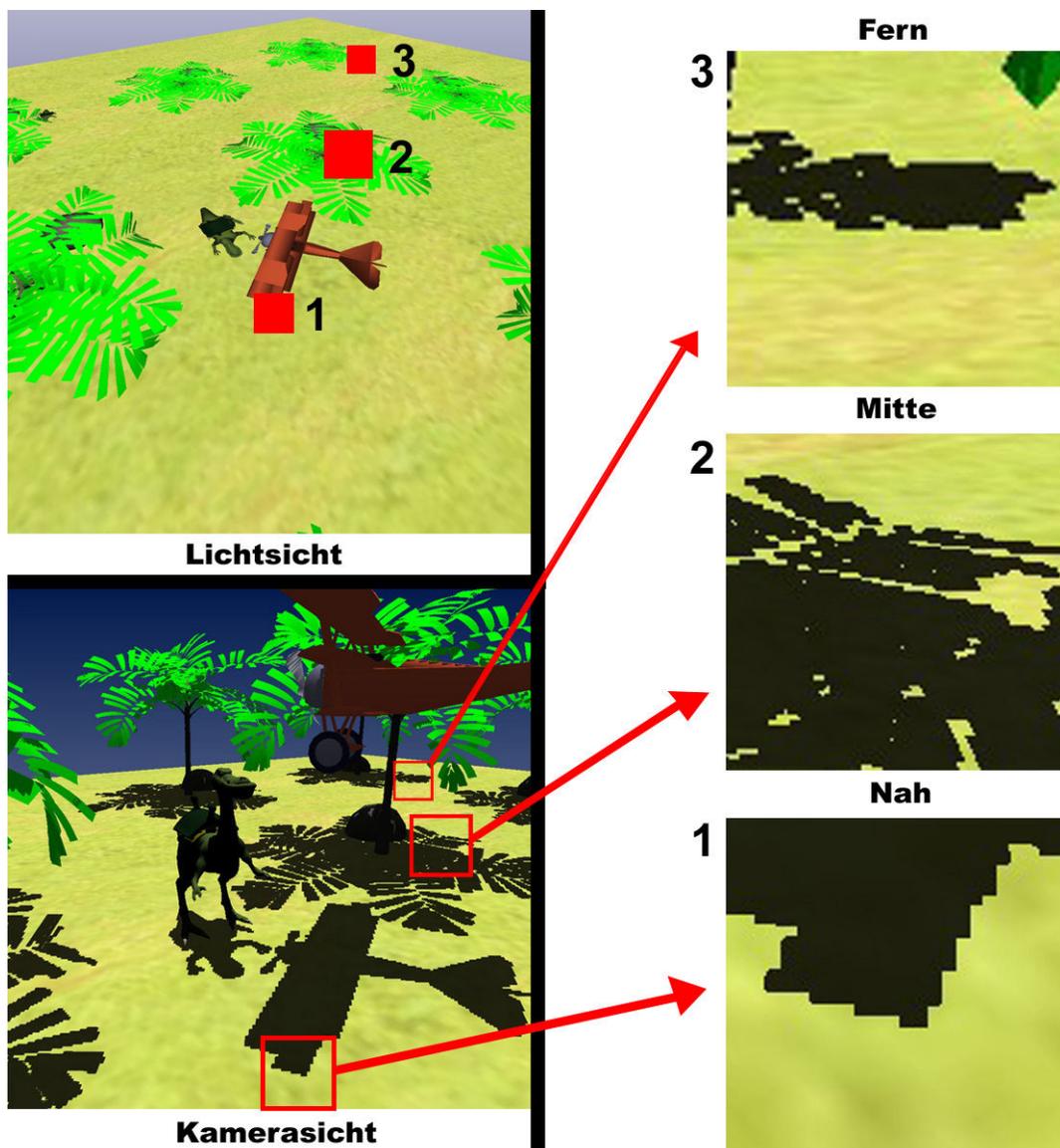


Bild 48 – Detailansicht beim Trapez-Shadow-Mapping

In der Vergrößerung fällt sofort die deutlich bessere Schattenqualität im Nahbereich auf. Schaut man sich die Größe der Fläche bei der Kamerasicht an, wird der Grund hierfür klar. Da diese Fläche in der Lichtsicht deutlich größer ist als die beim Standard-Shadow-Mapping, bekommt sie auch mehr der Auflösung. Dadurch reduziert sich das Aliasing.

Der Mittelbereich sieht in der Vergrößerung nur ein wenig besser aus als beim Standard-Shadow-Mapping. Auch das liegt daran, dass der entsprechende Bereich auf der Shadow Map etwas größer ist.

Anders verhält es sich im Fernbereich. Die Schattenqualität in der Vergrößerung sieht etwas schlechter aus, als beim Standard-Shadow-Mapping. Tatsächlich ist der entsprechende Bereich der Shadow Map hier kleiner als beim Standard-Shadow-Mapping. Dies ist genau der Effekt, den die perspektivische Verzerrung erreichen soll. Ein Teil der Auflösung wird sozusagen von hinten nach vorn verlagert. Aus Übersichtsgründen wurde für die Markierungen in der Lichtsicht keine Perspektive eingerechnet, so dass diese immer rechteckig sind.

Da das Trapez-Shadow-Mapping die Möglichkeit bietet, die Focus Region selber anzugeben, wurde sie für alle hier gezeigten Tests auf einen Wert von 0.3 gesetzt, was bedeutet, dass die ersten 30% der Kamerasicht insgesamt 80% der Shadow-Map-Auflösung zur Verfügung gestellt bekommt. Auf die Vorteile dieser Möglichkeit gehe ich in der Diskussion im Abschnitt 10.1.5 noch genauer ein.

Vergleichen wir diese Ergebnisse mit denen des Lisp-Shadow-Mappings.

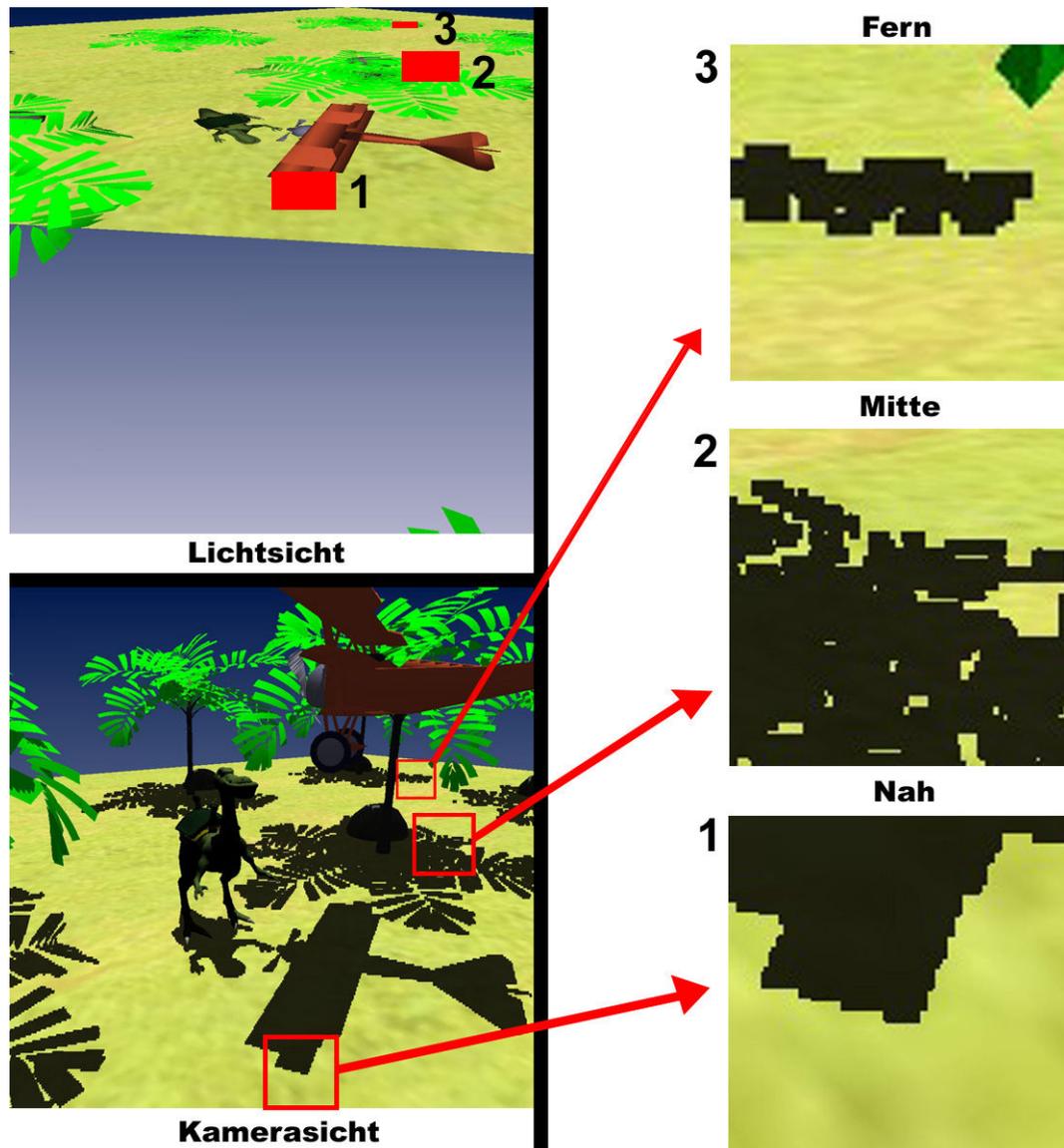


Bild 49 – Detailansicht beim Lisp-Shadow-Mapping

Auch hier erkennen wir sofort das verringerte Aliasing im Nahbereich. Es ist sogar noch weiter reduziert als beim Trapez-Shadow-Mapping. Wie erwartet ist wegen dieser Beobachtung der Bereich auf der Shadow Map noch etwas größer, als beim Trapez-Shadow-Mapping.

Der Mittelbereich sieht in der Vergrößerung schlechter aus als beim Trapez-Shadow-Mapping. Die Fläche in der Lichtsicht ist hier entsprechend kleiner. Ähnlich verhält es sich beim Fernbereich. Er ist deutlich kleiner als beim Trapez-Shadow-Mapping und sogar kleiner als beim Standard-Shadow-Mapping. Die Schattenqualität in diesem Bereich verringert sich im Vergleich zu den anderen Verfahren entsprechend, was in der Vergrößerung auffällt. Der Grund hierfür ist derselbe wie beim Trapez-Shadow-Mapping. Auflösung, die im vorderen Bereich mehr gewährt wird, muss an anderer Stelle eingespart werden. Vergleicht man die Lichtsicht des Trapez- und Lisp-Shadow-Mappings, fällt die etwas stärkere Verzerrung beim Lisp-Shadow-Mapping auf. Die Stärke der perspektivischen Verzerrung hat also nachweislich direkten Einfluss auf die Verteilung der Shadow Map Auflösung.

Aber noch haben wir die Frage nicht beantwortet, warum Perspective-Shadow-Maps in der Testszene so schlechte Ergebnisse erzielen. Schauen wir und dazu die verschiedenen Bereich des Bildes an.

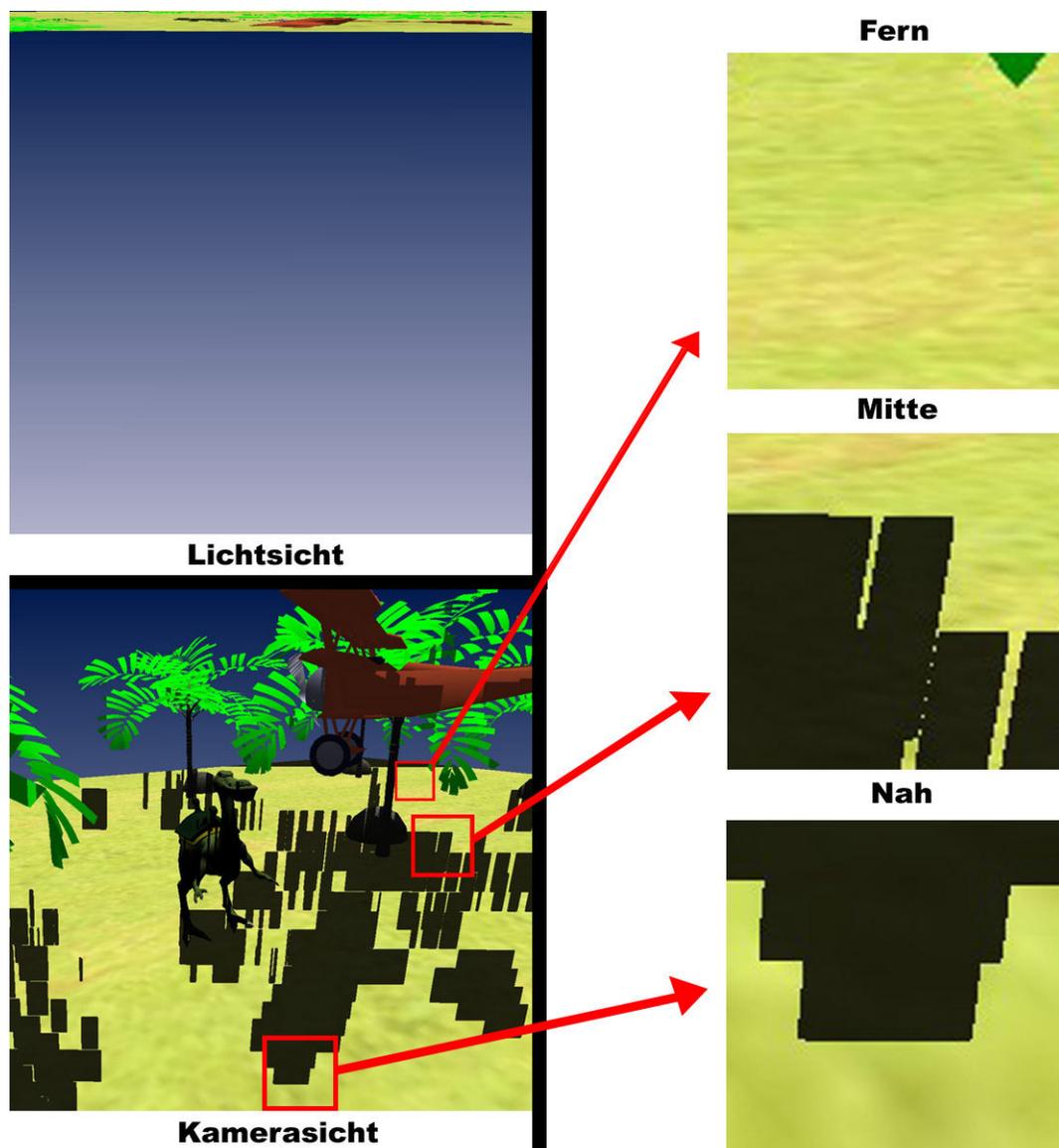


Bild 50 – Detailansicht beim Perspective-Shadow-Mapping

In der Lichtsicht ist deutlich zu sehen, dass die komplette Szene auf einen winzigen Bereich projiziert wurde. Der Bereich ist so klein, dass es mir nicht möglich war die entsprechenden Markierungen einzuzeichnen. Im Fernbereich fällt die Schattendarstellung sogar komplett aus. Auf einem Großteil der Lichtsicht ist nichts zu sehen. Genau hier liegt das Problem. Die Perspektive ist beim Perspective-Shadow-Mapping so stark, dass ein winziger Bereich vor der Kamera einen Großteil der Shadow-Map-Auflösung zur Verfügung gestellt bekommt. In unserer Testszene befindet sich in diesem Bereich jedoch kein Objekt. Die meiste Auflösung der Shadow Map wird in diesem Fall für unwichtige Bereiche der Kameransicht zur Verfügung gestellt. Im folgenden Bild habe ich die Kamera so eingestellt, dass sich im unmittelbaren Nahbereich ein Objekt befindet.



Bild 51 – Sehr starke perspektivische Verzerrung

Man sieht dass dieses Objekt jetzt sehr groß an der Stelle abgebildet wird, auf der vorher nichts zu sehen war. Demnach bekommt dieses Objekt auch eine sehr hohe Shadow-Map-Auflösung zugeteilt, wodurch die Schattenkanten nahezu Aliasingfrei dargestellt werden. Genau für diese starken Verzerrungen ist das Perspective-Shadow-Mapping gedacht. Hat die Kamera eine sehr große Sichtweite und es sollen relativ kleine Objekte im Vordergrund schattiert werden, bietet dieses Verfahren die optimale Lösung.

Zum Vergleich zeige ich ein Ergebnisbild aus dem Paper [Q11] zu diesem Verfahren.



Bild 52 – Per Perspective-Shadow-Mapping schattierte Szene [Q10]

Im Vordergrund sind sehr filigrane Objekte zu erkennen, die für eine gute Schattenqualität eine hohe Auflösung der Shadow Map benötigen. Die Kühe und Bäume im Hintergrund müssen im Gegensatz dazu nur relativ ungenau schattiert werden.

10.1.3 Verschiedene Blickwinkel

Bei der Erklärung der einzelnen Verfahren haben wir festgestellt, dass die erreichte Qualität stark vom Blickwinkel in Relation zur Lichtposition bzw. Lichtrichtung abhängig ist. In diesem Abschnitt sehen wir uns die Szene aus zwei zusätzlichen Blickwinkeln an und vergleichen die Ergebnisse mit denen von oben, die fast den Idealfall darstellten. Der Idealfall tritt dann ein, wenn die Blickrichtung senkrecht zur Lichtrichtung liegt.

In der ersten zusätzlichen Situation nähert sich die Blickrichtung der Lichtrichtung an, in der zweiten schauen wir mit der Kamera genau in Lichtrichtung. Da im letzten Fall die Objekte der Szene den Schattenwurf verdecken würden, schauen wir von unten gegen Lichtrichtung auf die Szene, wir sehen die Schatten also praktisch durch den Boden.

Die folgende Bildserie zeigt die Ergebnisse beim Standard-Shadow-Mapping. In der unteren Reihe ist die dazugehörige Lichtsicht abgebildet.

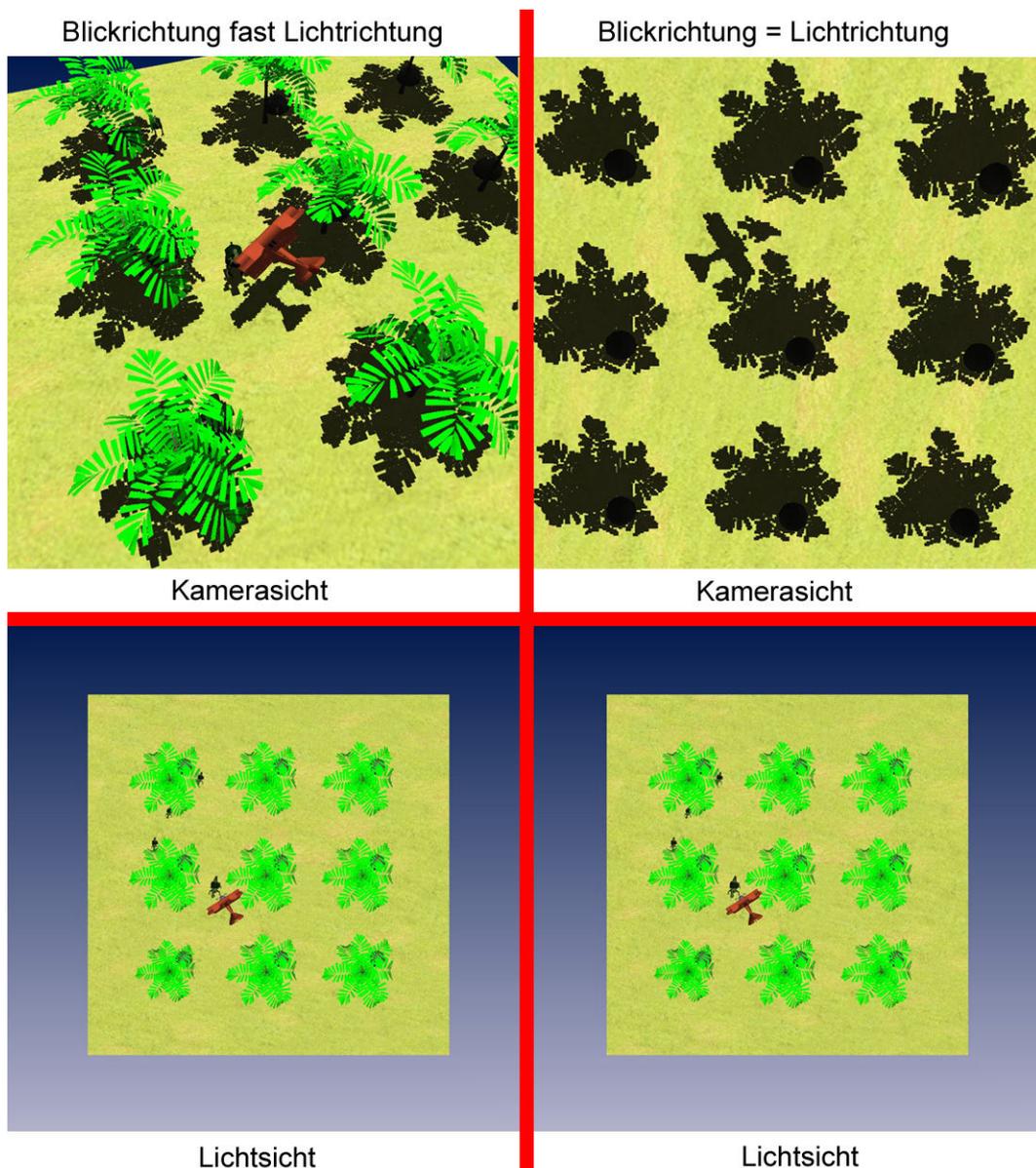


Bild 53 – Ungünstiger Blickrichtung beim Standard-Shadow-Mapping

Man kann erkennen, dass die Schattenqualität in jeder Situation konstant ist. Da die Lichtsicht unabhängig vom Kamerablickwinkel immer gleich ist, wird auch jeder Bereich mit einer gleich hohen Shadow-Map-Auflösung versorgt. Anders sieht dies bei den perspektivischen Verfahren aus. Schauen wir uns das Trapez-Shadow-Mapping an.

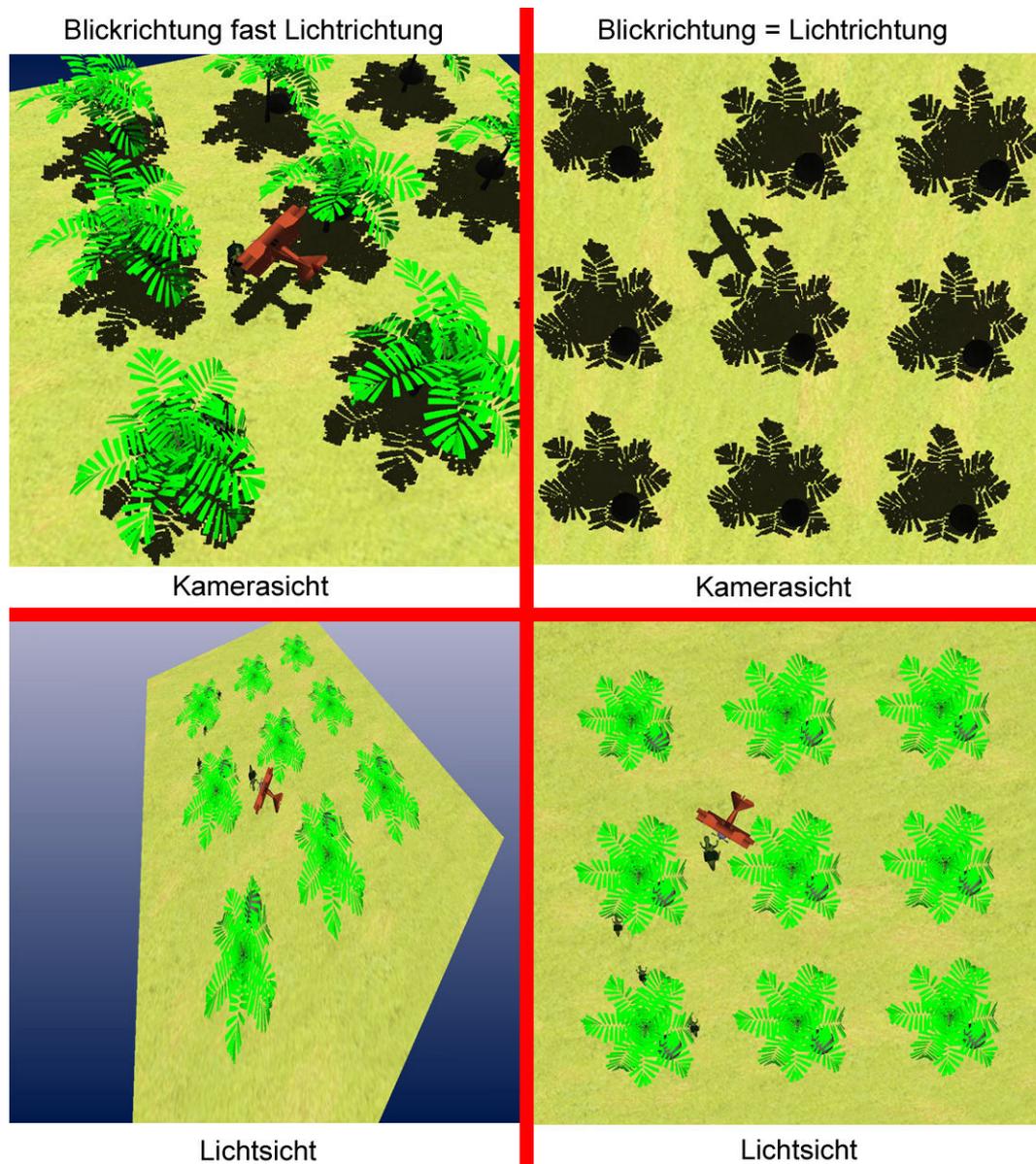


Bild 54 – Ungünstiger Blickrichtung beim Trapez-Shadow-Mapping

Hier erkennt man, dass die Schattenqualität, im Vergleich zu den bisher gezeigten Ergebnissen, sehr stark nachgelassen hat. Die perspektivische Verzerrung ist links für einen Großteil der Szene sehr gering geworden, während sie rechts nicht mehr vorhanden ist. Im linken Bild kann man erkennen, dass die Schatten in den Bereichen, in denen noch eine Verzerrung vorhanden ist, weniger Aliasing aufweisen als die im Hintergrund.

In Kapitel 6.3.1 haben wir die Situation, dass man beim Trapez-Shadow-Mapping in oder gegen Lichtrichtung schaut, als Problemfall eingestuft. Weil hier kein Trapez mehr berechenbar ist, nimmt man die Eckpunkte der Far-Plane als Trapezpunkte. Das Ergebnis kommt, je nach Sichtweite, dem Standard-Shadow-Mapping gleich. Ist die Sichtweite hingegen sehr groß gewählt, kann dies zu einer schlechteren Schattenqualität führen. Auf eine Verbesserung dieser Situation wird im nächsten Abschnitt eingegangen. Auf dem Bild rechts ist die Far-Plane etwas kleiner als die komplette Lichtsicht beim Standard-Shadow-Mapping, wodurch es zu einer etwas besseren Schattenqualität kommt.

Schauen wir uns jetzt das Lisp-Shadow-Mapping an.

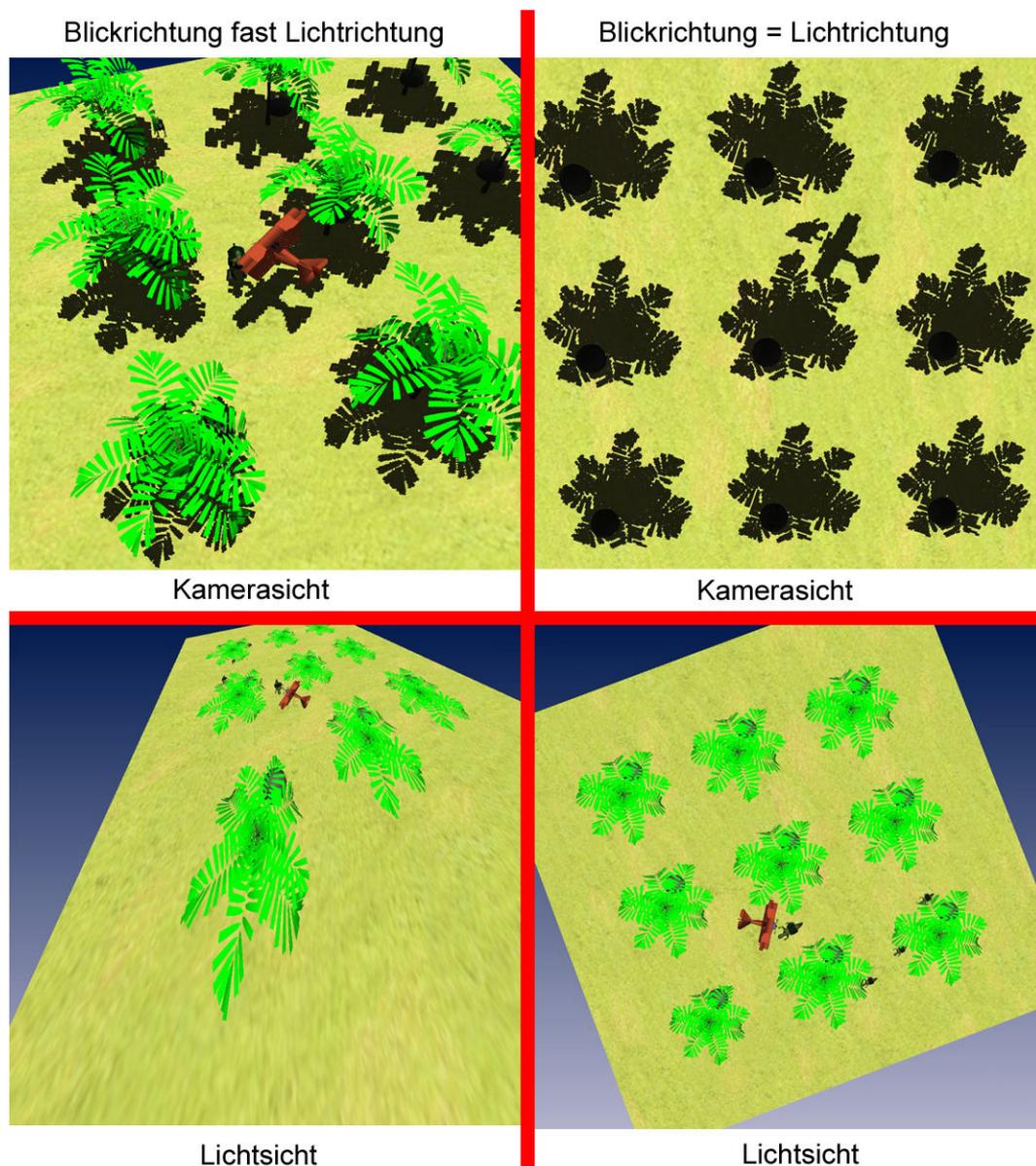


Bild 55 – Ungünstige Blickrichtung beim Lisp-Shadow-Mapping

Auch hier ist die perspektivische Verzerrung in der Situation, dass die Blickrichtung der Lichtrichtung gleicht, nicht mehr vorhanden. Das Ergebnis entspricht hier immer genau dem des Standard-Shadow-Mappings, und es kann nie schlechter werden.

Links ist die perspektivische Verzerrung noch stärker als beim Trapez-Shadow-Mapping in der Situation. Im resultierenden Kamerabild kann man erkennen, dass die Palme, die der Kamera am nächsten ist, eine bessere Schattenqualität hat, als beim Trapez-Shadow-Mapping. Im Gegenzug sind die Schatten im Hintergrund deutlich schlechter. Diese Ergebnisse sind mit denen des Perspective-Shadow-Mappings vergleichbar.

10.1.4 Ergebnisse der vorgestellten Verbesserungen

Den Vergleich des Perspective-Shadow-Mappings mit dem Lisp-Shadow-Mapping habe ich bereits im letzten Abschnitt durchgeführt. Hier gehe ich auf die in Kapitel 6.3 vorgestellten Verbesserungen ein. Schauen wir, was die Winkelkorrektur aus Kapitel 6.3.3 leistet. Betrachten wir zunächst die Ergebnisse ohne die Korrektur.

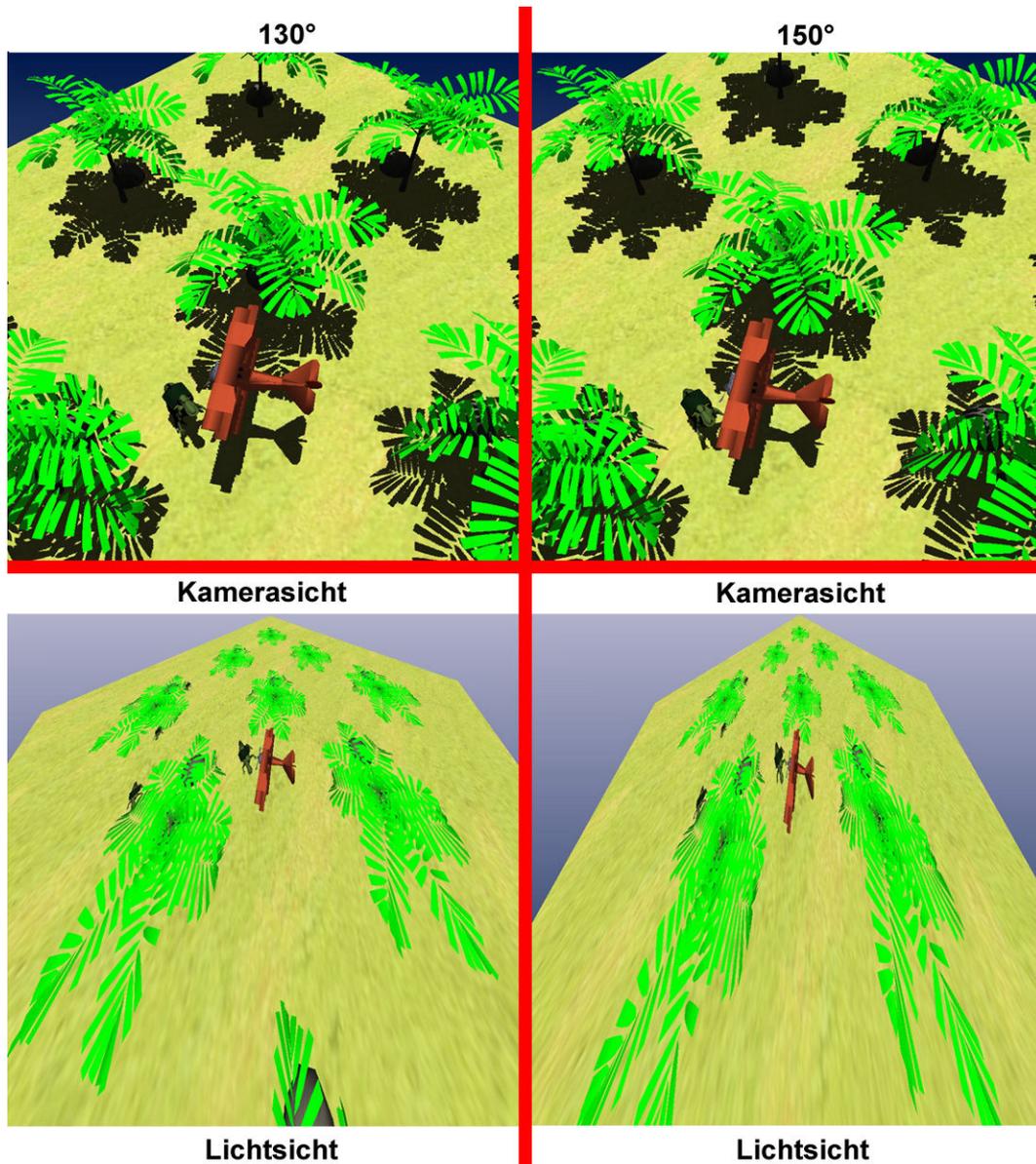


Bild 56 – Ergebnisse bei einem großen Trapezwinkel ohne Korrektur

Die Bilder zeigen die Kamera- sowie die dazugehörige Lichtsicht. Dabei wurden Blickwinkel ausgewählt, die noch nicht die Verwendung der Far-Plane als Trapez aus Kapitel 6.3.1 zur Folge haben. In den jeweiligen Bildern wurde der Winkel mit angegeben, der von den Seitenlinien aufgespannt wird. Man erkennt, dass die Qualität gerade im Hintergrund mit steigendem Winkel stark abnimmt. Diesen schnellen Qualitätsverlust versucht die genannte Verbesserung auszugleichen.

Das folgende Bild zeigt die Ergebnisse mit Korrektur.

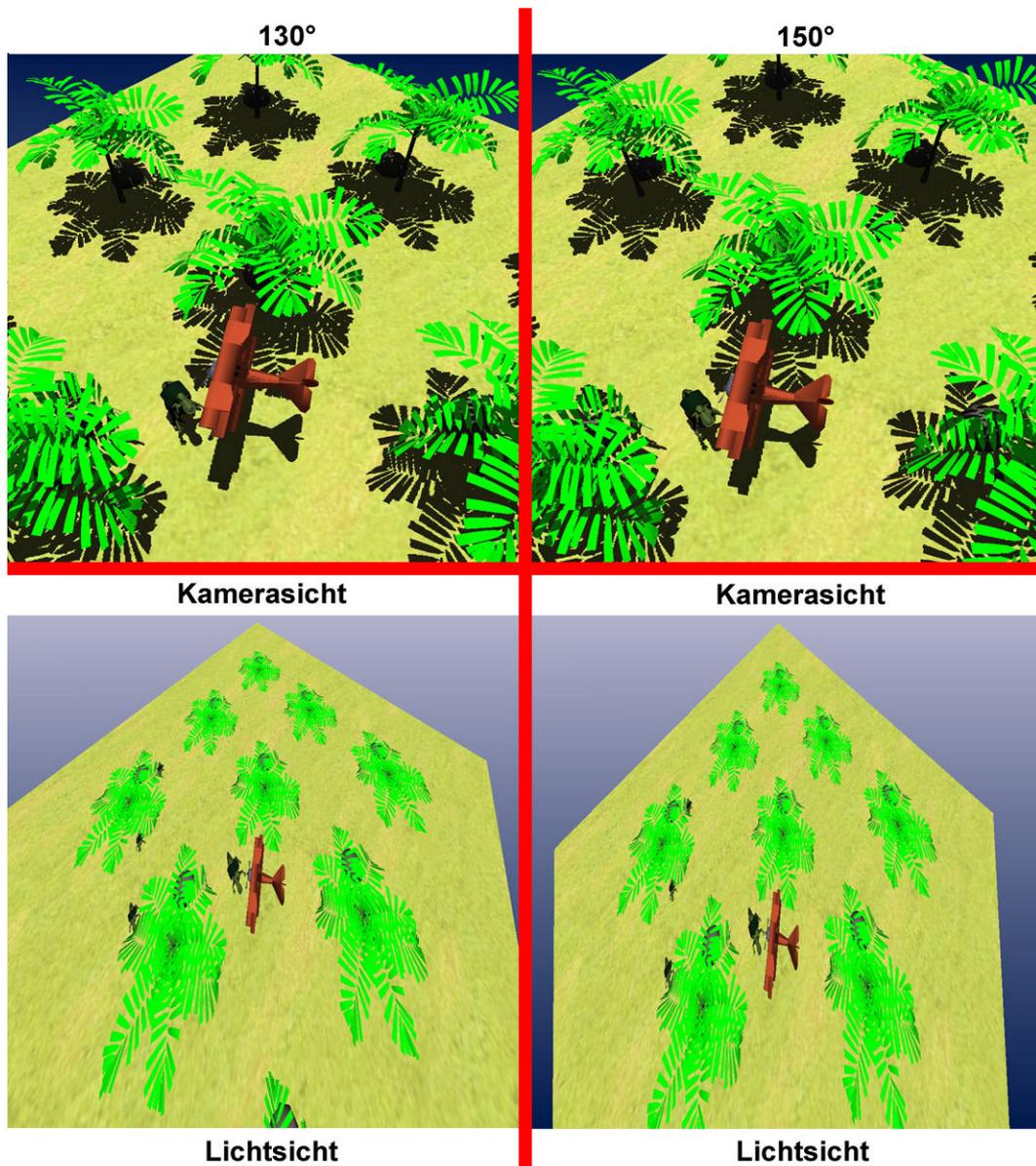


Bild 57 – Ergebnisse bei einem großen Trapezwinkel mit Korrektur

Man sieht, dass die Schattenqualität gerade im Hintergrund deutlich besser ist. Sogar die Schatten der einzelnen Palmwedel sind zu erkennen. Zwar wird die Schattenqualität im unteren Bereich der Kamerasicht etwas schlechter, dafür verteilt sich die Shadow-Map-Auflösung relativ gleichmäßig auf der gesamten Szene. Durch die Korrektur lässt die perspektivische Verzerrung, im Vergleich zu den Ergebnissen ohne die Korrektur, nach. Dies ist gewollt, da in der gezeigten Situation eine perspektivische Verzerrung das Schattenergebnis ungleichmäßig werden lässt, wie man im obigen Bild 57 erkennen konnte. Das Ziel, eine gleichmäßiger Schattenqualität zu erhalten, wurde erreicht.

Als nächstes untersuchen wir, ob auch die automatische Anpassung der Base- und Top-Line, die in Kapitel 6.3.2 vorgestellt wurde, eine Qualitätsverbesserung bringt.

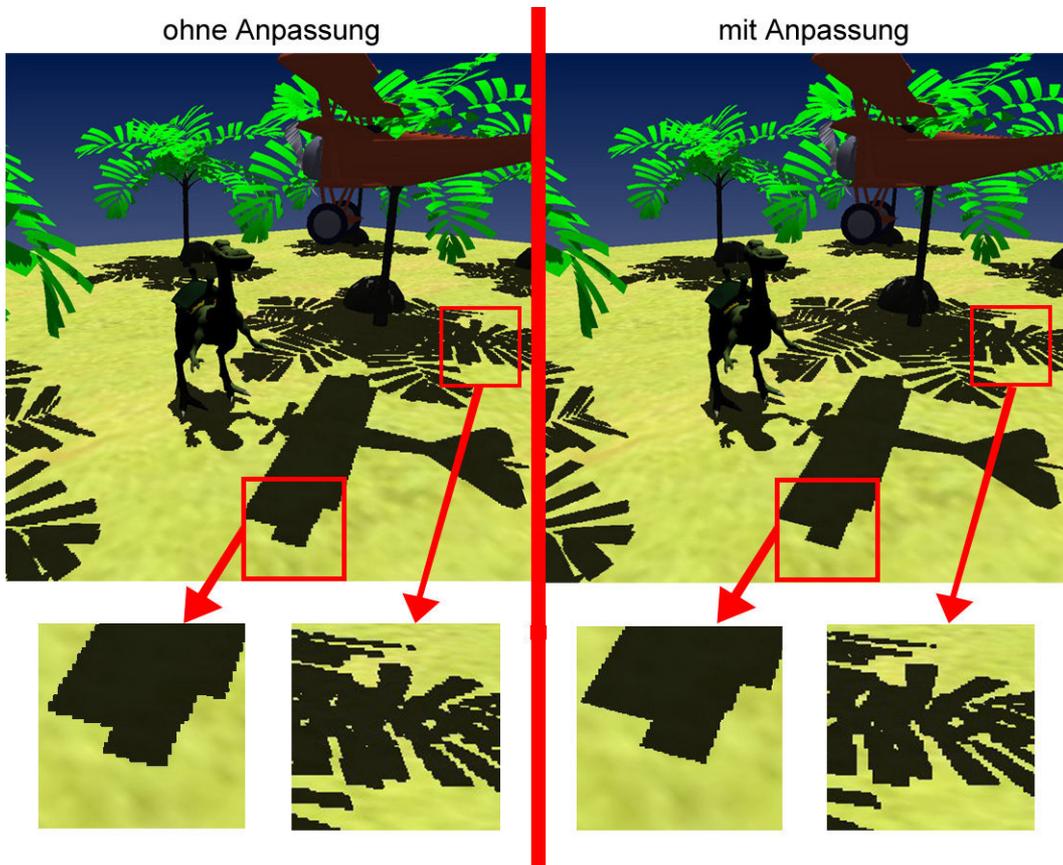


Bild 58 – Links: Schatten ohne Anpassung, Rechts: Schatten mit Anpassung

Anhand der Vergrößerungen kann auch hier eine Qualitätssteigerung in Normalsituationen festgestellt werden. Ein weitaus höherer Qualitätsunterschied ergibt sich aber dann, wenn die Far-Plane als Trapez verwendet wird und sich nur in einem kleinen Bereich des Kamerafrustums Objekte befinden.

Bild 59 demonstriert diese Situation.

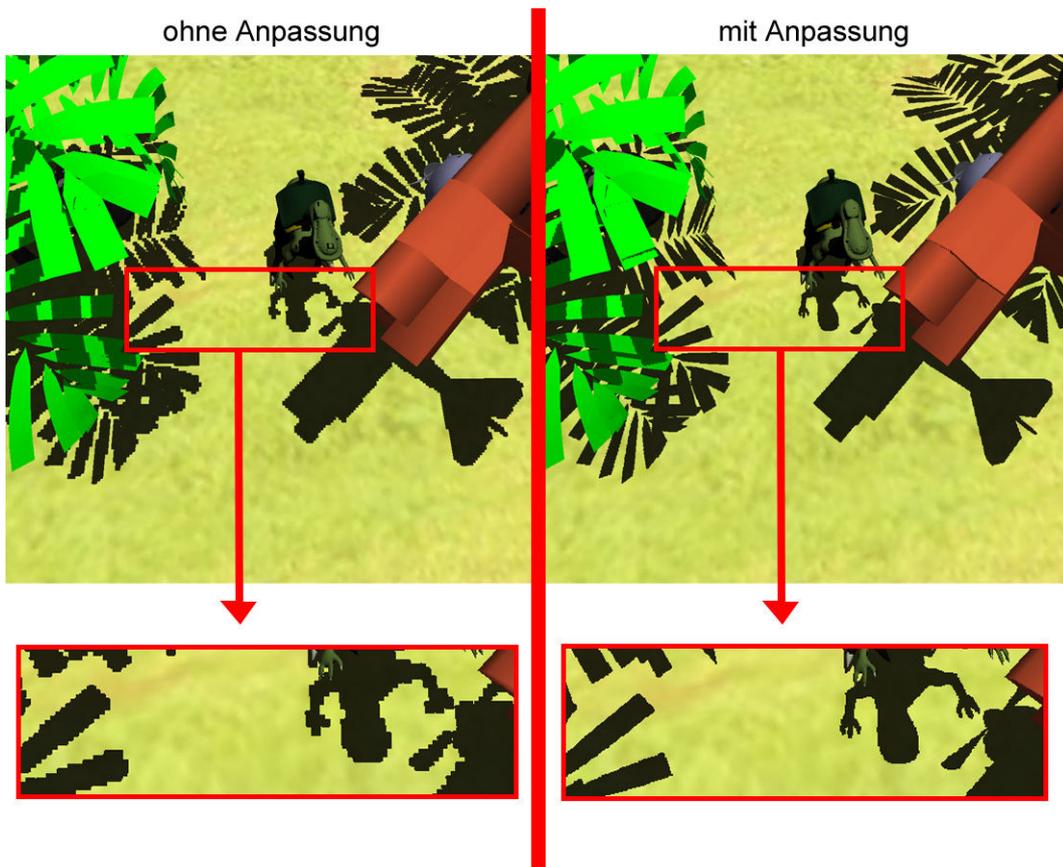


Bild 59 – Far-Plane wird als Trapez verwendet, Links: ohne Anpassung, Rechts: mit Anpassung

Dadurch, dass die Far-Plane im Vergleich zum tatsächlich gesehenen Bereich der Szene sehr groß ist, wird die Shadow Map ebenfalls über einem zu großen Bereich gebildet. Durch die Sichtweitenanpassung der genannten Verbesserung wird der Bereich für die Shadow Map auf den gesehenen Bereich beschränkt.

10.1.5 Diskussion

In diesem Abschnitt werden die gewonnenen Erkenntnisse diskutiert und die Vor- und Nachteile der einzelnen Verfahren in Hinsicht auf ihre erreichte Schattenqualität aufgeführt.

Wir haben im ersten Abschnitt festgestellt, dass mit steigender Auflösung der Shadow Map auch die Schattenqualität gesteigert wird. Die perspektivischen Verfahren erreichten hier bereits bei weniger als der Hälfte der Auflösung ähnliche oder sogar bessere Ergebnisse als das Standard-Shadow-Mapping. Dies ist ein wichtiger Punkt für die Performance des Verfahrens, wie wir im Kapitel 10.3 sehen werden. Beim Gesamteindruck lieferten das Trapez- sowie das Lisp-Shadow-Mapping ähnlich gute Ergebnisse. Das Perspektivischem-Shadow-Mapping lieferte jedoch im Vergleich zum Standard-Shadow-Mapping sehr schlechte Ergebnisse. Wir haben festgestellt, dass dies mit der sehr starken Perspektive zusammenhängt, die bei diesem Verfahren verwendet wird. In der Detailbetrachtung konnte sowohl beim Trapez-, als auch beim Lisp-Shadow-Mapping nachgewiesen werden, dass die Schattenqualität zwar im Frontbereich stark verbessert wird, diese aber im Gegenzug in den hinteren Regionen schlechter ist als beim Standard-Shadow-Mapping.

Damit haben wir einen direkten Zusammenhang der perspektivischen Verzerrung und der erreichten Schattenqualität nachgewiesen. Je stärker diese Verzerrung ist, umso mehr Shadow-Map-Auflösung wird dem vorderen Kamerabereich zur Verfügung gestellt und dem hinteren abgezogen. Hier kommt die Stärke des Trapez-Shadow-Mappings zur Geltung. Durch die einfache und verständliche Definition des Bereichs, der besonders gute Schattenqualität liefern soll, kann dieses Verfahren für nahezu jede Situation optimal angepasst werden. Zwar lässt sich die Perspektive beim Perspective- und Lisp-Shadow-Mapping auch beeinflussen, hierzu ist aber eine genaue Kenntnis der Funktionsweise des Verfahrens vorausgesetzt. Die folgende Bildserie zeigt weitere Ergebnisse des Trapez-Shadow-Mappings mit unterschiedlich definierter Focus Region.

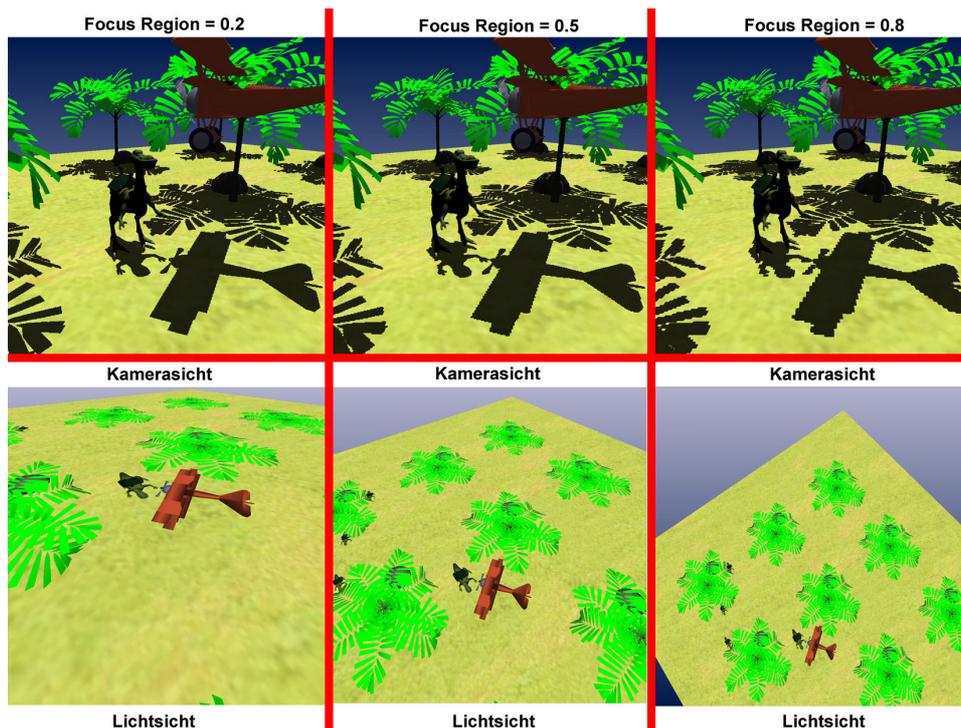


Bild 60 – Ergebnisse von unterschiedlich gewählten Focus Regions

Vergleicht man diese Ergebnisse mit den Ergebnissen der anderen Verfahren, so kann man feststellen, dass das Trapez-Shadow-Mapping genau dieselben Ergebnisse erreichen kann wie diese. Aus diesem Grund ist dieses Verfahren das universell einsetzbarste der hier vorgestellten Hard-Shadow-Verfahren.

Die Nachteile der perspektivischen Schattenverfahren habe ich im Kapitel 10.1.3 aufgezeigt. Die Schattenqualität kann, je nach Relation der Blickrichtung zur Lichtrichtung, stark schwanken. Zu diesem Problem wurden für das Trapez-Shadow-Mapping Lösungen vorgestellt, die die Qualität auch hier weiter verbessern.

Hinsichtlich der Schattenqualität kann man abschließend festhalten, dass für die meisten Situationen das Lisp-Shadow-Mapping und das Trapez-Shadow-Mapping ähnlich gute Ergebnisse liefern. Durch die aufgezeigten Verbesserungen und die universelle sowie leicht verständliche Konfiguration des Trapez-Shadow-Mappings ist dieses Verfahren dem Lisp-Shadow-Mapping etwas überlegen.

Ich habe zudem die Leistung des Lisp-, sowie Trapez-Shadow-Mappings mit der des Standard-Shadow-Mappings verglichen. Dabei habe ich festgestellt, dass der Mehraufwand dieser Verfahren minimal ist. Die Frameraten lagen bei beiden Verfahren nie mehr als 5-10% unter der des Standard-Shadow-Mappings. Dieser Wert kann durch effizientere Berechnungen sicher noch weiter reduziert werden, so dass es keinen Grund gibt, aus Performancegründen zum Standard-Shadow-Mapping statt zu einem perspektivischen Shadow-Mapping-Verfahren zu greifen.

10.2 Schattenqualität weiche Schatten

In diesem Kapitel betrachten wir die Ergebnisse der beiden vorgestellten Soft-Shadow-Verfahren. Da sie auf dem Standard-Shadow-Mapping basieren, arbeiten sie unabhängig vom Blickwinkel und liefern an jeder Stelle eine konstante Schattenqualität. Eine Kopplung der perspektivischen Verfahren mit den vorgestellten Soft-Shadow-Verfahren ist nicht ohne weiteres möglich, da der verwendete PCF-Filter davon ausgeht, dass die Shadow-Map-Auflösung an jeder Stelle gleich verteilt ist.

Schauen wir uns die Ergebnisse des PCF-Shadow-Mappings bei unterschiedlichen Auflösungen der Shadow Map an.

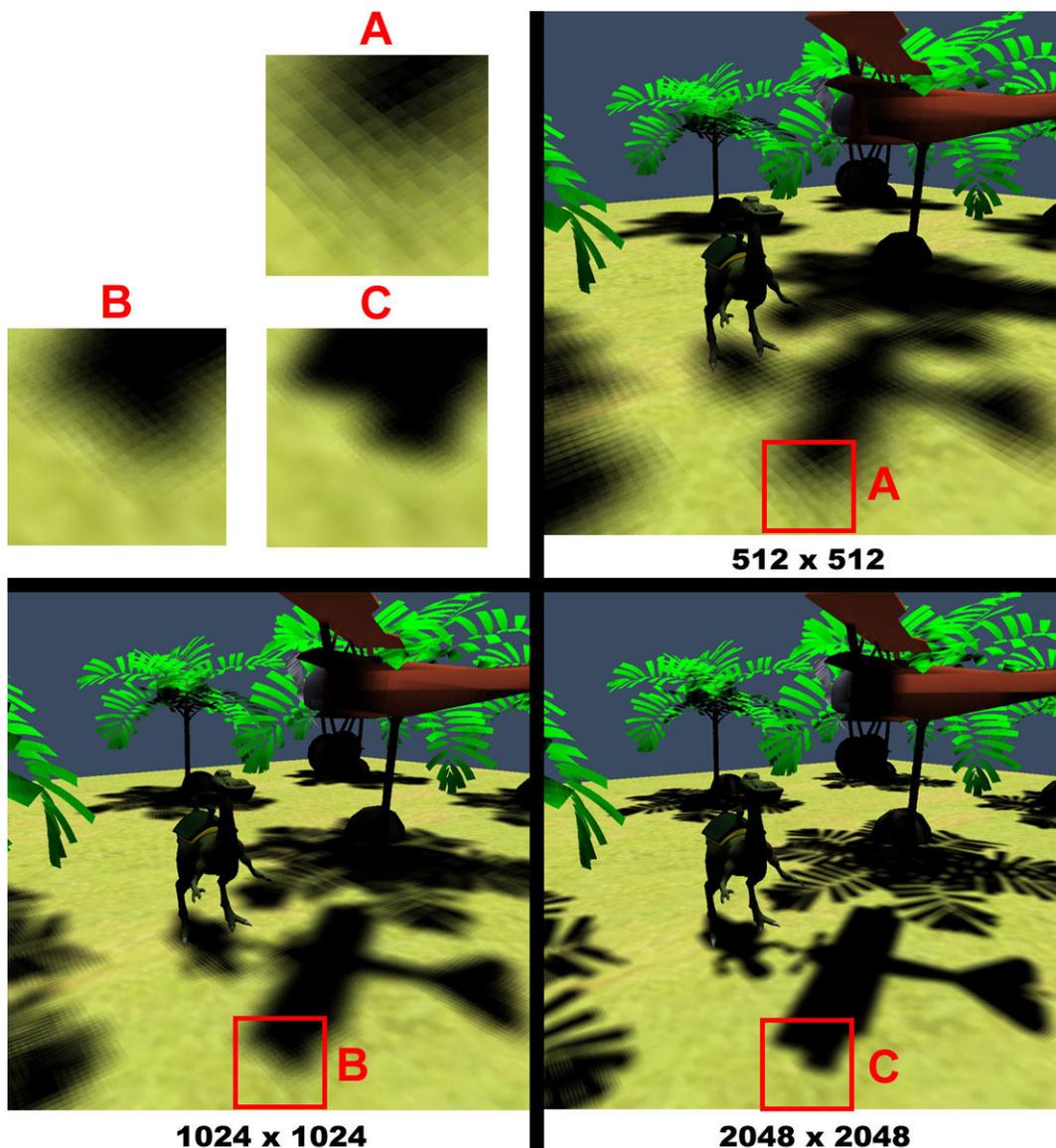


Bild 61 – PCF-Soft-Shadows bei unterschiedlichen Shadow-Map-Auflösungen

Schon beim Gesamteindruck fällt auf, dass weiche Schatten die Szene deutlich realistischer wirken lassen als harte. Zudem haben Soft-Shadows nach dem Prinzip des PCF-Shadow-Mappings einen weiteren positiven Effekt: Sie verdecken Aliasingeffekte. Im Bild kann man erkennen, dass auch bei niedrigen Shadow-Map-Auflösungen im Bezug auf Aliasing eine gute Qualität erreicht wird,

der Treppcheneffekt fällt so gut wie gar nicht auf. In der Vergrößerung ist jedoch zu erkennen, dass die Qualität der Farbverläufe an den Schattenkanten zu wünschen übrig lässt. Da hier ein 8x8-PCF-Filter eingesetzt wurde, wurde aus Performancegründen die maximale Sampleanzahl von 64 auf 36 gesenkt. Dadurch kommt es zu diesem Qualitätsverlust.

Bei einer geringeren Auflösung der Shadow Map verlaufen die Schattenkanten mehr als bei einer hohen. Das liegt daran, dass als Maß für den PCF-Filter die Pixelgröße der Shadow Map in Kamerakoordinaten benutzt wird. Je geringer die Auflösung der Shadow Map ist, desto größer ist der Bereich in Kamerakoordinaten, der auf einen einzelnen Pixel der Shadow Map abgebildet wird. Diese Erkenntnis muss man bei der Wahl einer geeigneten Suchregion beachten.

Im nächsten Bild wurde zur Qualitätsverbesserung der Hardware-PCF-Filter aktiviert.

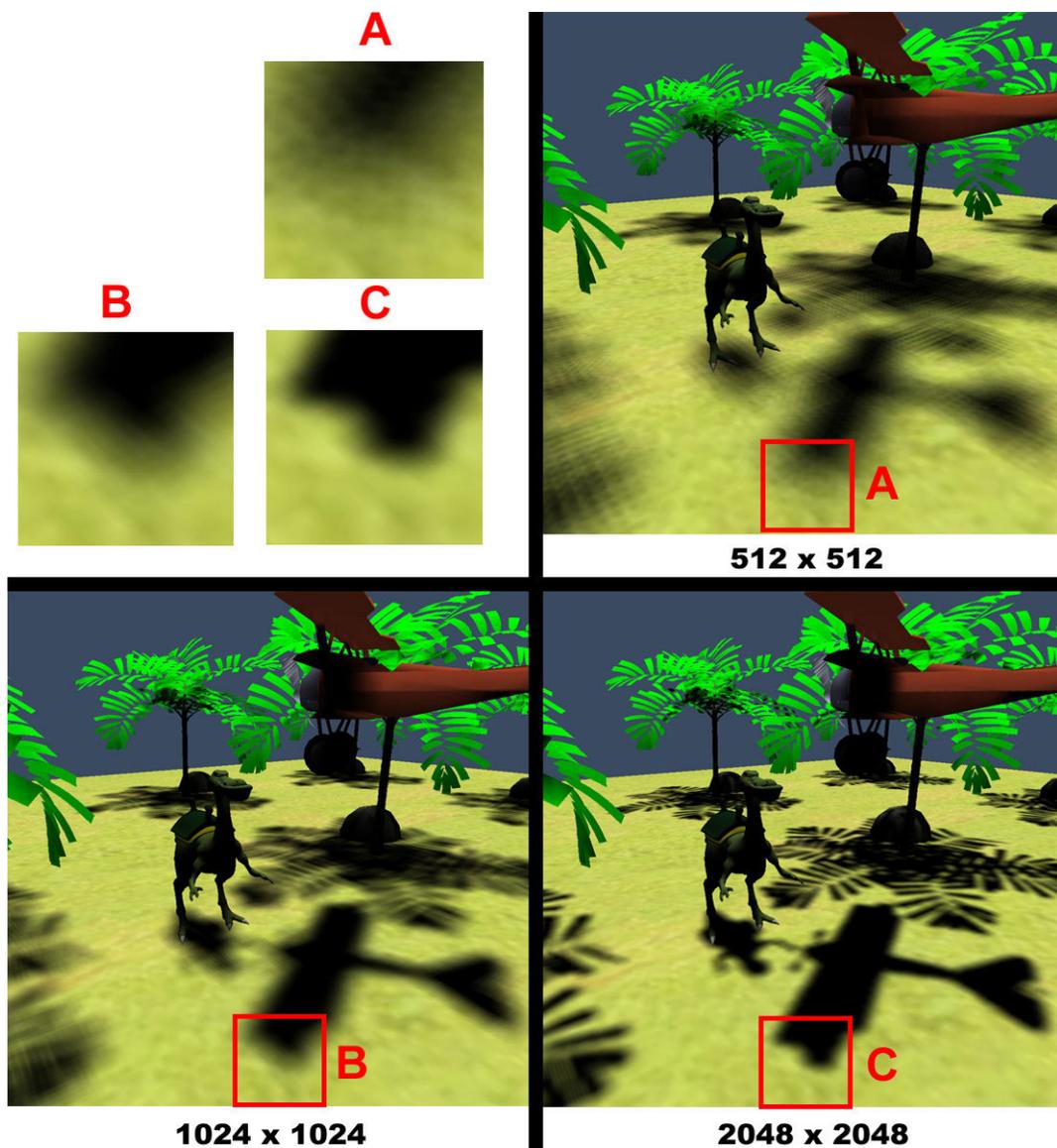


Bild 62 – PCF-Soft-Shadows mit aktiviertem Hardware-PCF-Filter

Man sieht, dass die Qualität an den Schattenkanten deutlich besser ist. Bei einer Auflösung von 512^2 Pixel der Shadow Map reicht der Hardware-PCF-Filter nicht mehr aus, um die Schattenqualität auf ein gutes Maß zu bringen. Aus diesem Grund ist es wichtig, für die Darstellung von Soft-Shadows mit dem PCF-Verfahren, die richtige Relation zwischen Shadow-Map-Auflösung und Suchregion zu wählen.

Als nächstes betrachten wir das Perspektivische-PCF-Shadow-Mapping. Alle hier gezeigten Screenshots zu diesem Verfahren stammen aus einer OpenGL-Anwendung, da die Implementierung dieses Verfahrens in OpenSG noch Problemen hervorrief.

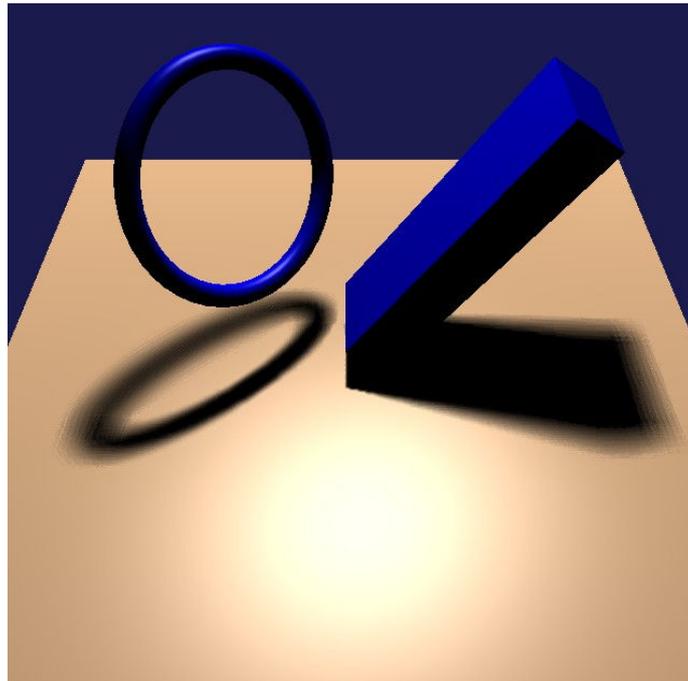


Bild 63 – Soft-Shadows des Perspektivischen-PCF-Shadow-Mappings

Wie man auf den ersten Blick erkennt, erzeugt dieses Verfahren die realistischsten Soft-Shadows aller in dieser Arbeit vorgestellten Verfahren. Der Schattenrand verschwimmt mit gestiegenem Abstand vom Schattenverursacher zum Schattenempfänger immer mehr. In dem oben gezeigten Bild habe ich einen relativ kleinen Wert für die Größe der Lichtquelle gewählt. Im nächsten Bild habe ich diesen Wert erhöht, um eine größere Lichtquelle zu simulieren.

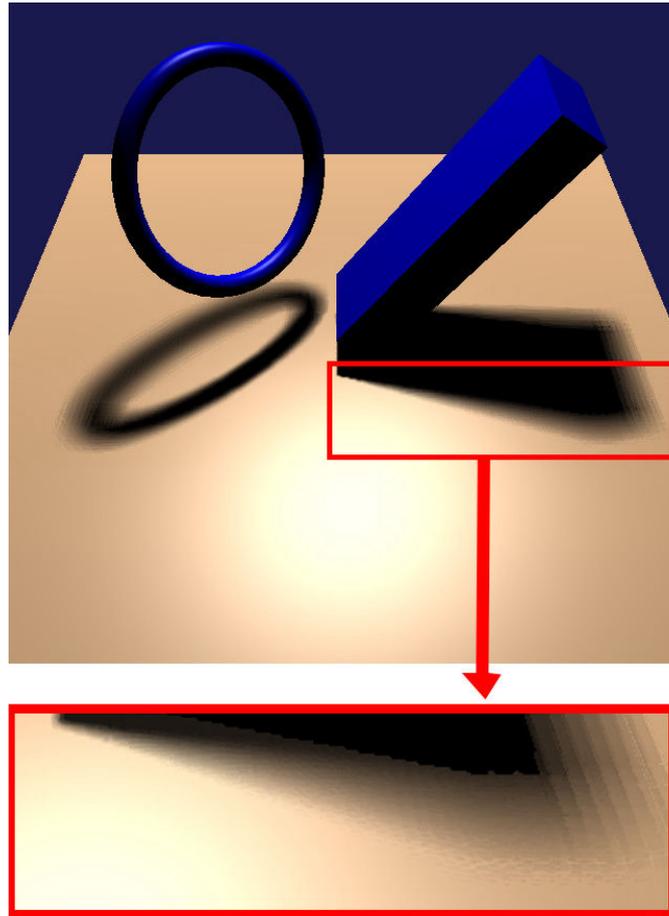


Bild 64 – Stark abgegrenzte Farbverläufe sind erkennbar

Die Schatten verschwimmen jetzt weiter als bei einer kleineren Lichtquelle. Allerdings erkennt man in der Vergrößerung, dass dieselben Probleme mit der Schattenqualität auftreten wie beim PCF-Shadow-Mapping. Zu deren Verbesserung kann man die gleichen Maßnahmen ergreifen wie beim PCF-Shadow-Mapping. So steigert z.B. ein aktivierter Hardware-PCF-Filter die Qualität.

Im Folgenden möchte ich aber eine andere Möglichkeit zur Qualitätsverbesserung vorstellen. Es handelt sich hierbei um die in Kapitel 7.3 vorgestellte Filtermethode. Bild 65 zeigt das Bild, dass ohne die beschriebene Modifikation zur Vermeidung von leuchtenden Objekten erstellt wurde, um das Problem zu verdeutlichen.

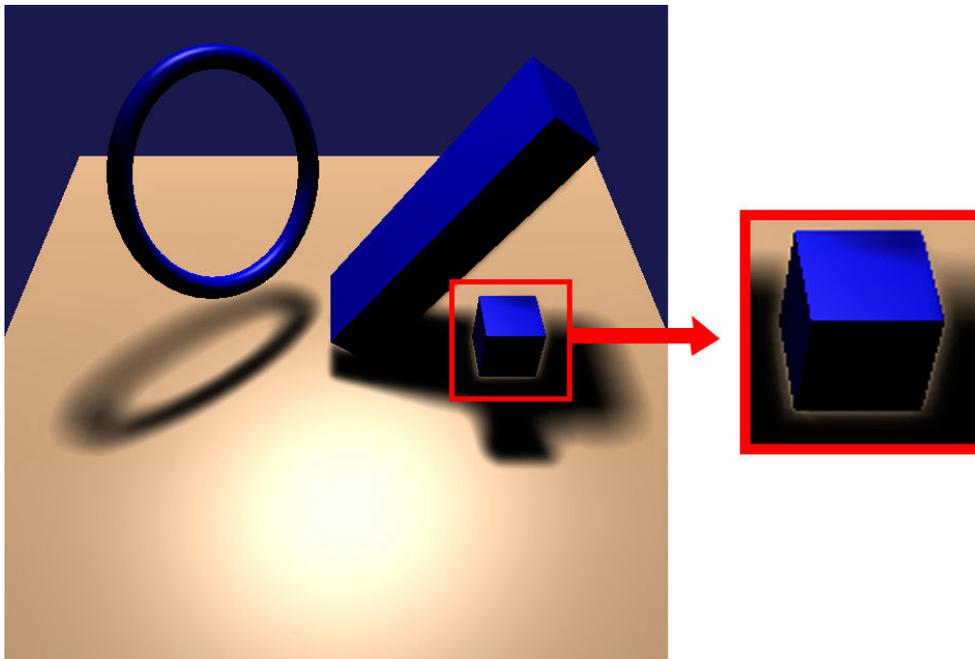


Bild 65 – Schatten um den Würfel verschwinden

Man kann einen hellen Kranz um den Würfel erkennen, der einen Teil des schattierten Bereichs verdeckt. Der Grund hierfür wurde im Kapitel 7.3 beschrieben. Bild 66 zeigt das Ergebnis nach der notwendigen Modifikation zur Behebung des Leuchteffekts.

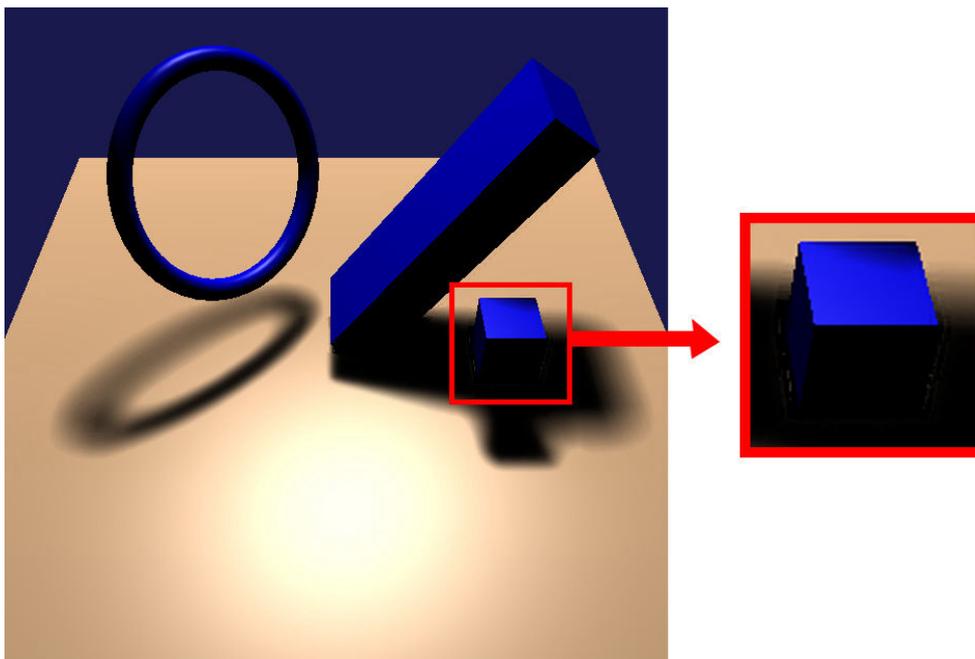


Bild 66 – Ergebnis nach der Korrektur

Hier ist der Leuchteffekt verschwunden. Die beschriebenen Unregelmäßigkeiten, die trotz der Verbesserung auftreten, sind so gering, dass sie auf der Vergrößerung kaum zu erkennen sind. Das Ergebnis ist nahezu perfekt. Zum Vergleich zeige ich noch einmal das ungefilterte Ergebnis neben dem gefilterten.

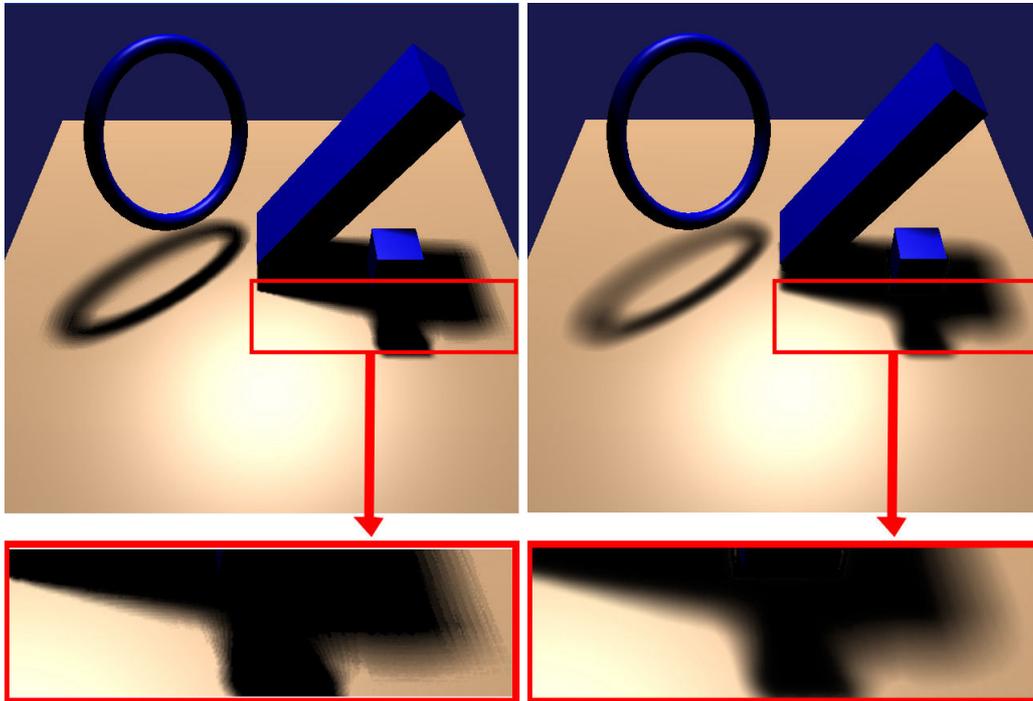


Bild 67 – Farbverläufe ohne Filter (links) und mit Gauß-Filter (rechts)

Die Filterung hebt die Schattenqualität deutlich an, die Übergänge sehen nahezu perfekt aus. Bild 68 zeigt zum Abschluss den Unterschied zwischen dem uniformen PCF-Shadow-Mapping und dem Perspektivischen-PCF-Shadow-Mapping.

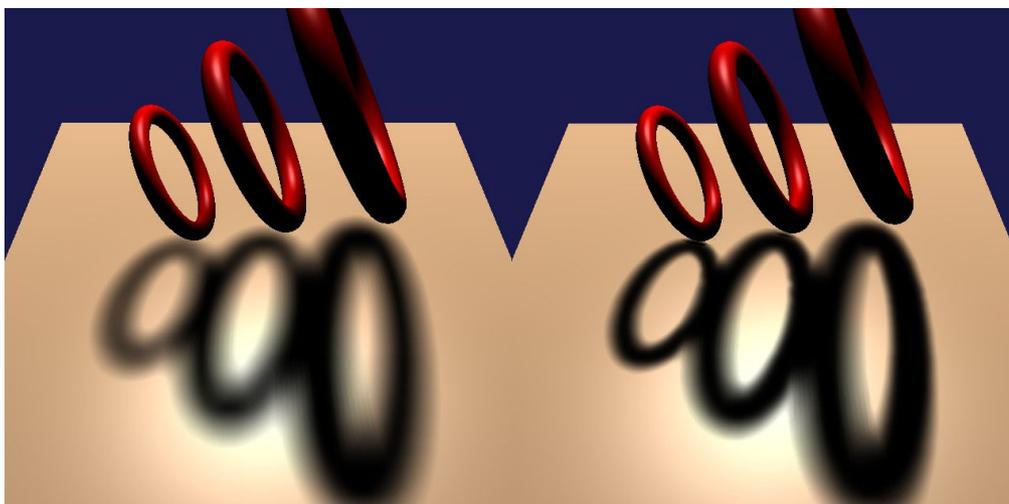


Bild 68 – Links: Uniformes PCF-Shadow-Mapping, Rechts: Perspektivisches-PCF-Shadow-Mapping

10.2.1 Diskussion

Eine Szene, in der Objekte weiche Schatten werfen, sieht deutlich realistischer aus als eine, in der nur harte Schatten erzeugt werden. Es wurde deutlich, dass der Übergang von harten zu weichen Schatten die Realitätsnähe einer Szene auf das nächste Level anhebt, egal ob es sich dabei um uniforme oder perspektivisch korrekte Soft-Shadows handelt.

Um diese Art von Schatten in Echtzeit darstellen zu können, wurde von mir erläutert, wie man die durchgeführten Berechnungen verringert, um eine akzeptable Leistung zu erreichen. Leider ließ dadurch die Schattenqualität ebenfalls nach.

Für die Aufwertung der erreichten Ergebnisse habe ich zwei Methoden vorgestellt. Die erste Methode bezog sich auf die Verwendung des Hardware-PCF-Filters. Dieser arbeitet sehr schnell, kann aber die Qualität nur begrenzt anheben. Die zweite Möglichkeit war das Filtern der Shadow Factor Map. Diese Methode ist etwas rechenintensiver, die Qualitätssteigerung fällt dabei aber deutlich höher aus.

Durch die genannten Recheneinsparungen laufen alle vorgestellten Verfahren in Echtzeit. Im Vergleich zum Standard-Shadow-Mapping sank die Framerate je nach gewählter Suchreichweite und Shadow-Map-Auflösung auf 25 bis 40%. Für den in den Beispielen gezeigten reduzierten 8x8-PCF-Filter mit aktiviertem Hardware-PCF-Filter und einer Fester- sowie Shadow-Map-Auflösung von 1024² Pixel, wurden 24 FPS im Vergleich zu 77 FPS beim Standard-Shadow-Mapping erreicht.

Die Testszene für das Perspektivische-PCF-Shadow-Mapping lief bei einer Fenster- sowie Shadow-Map-Auflösung von 512² Pixel und aktiviertem Hardware-PCF-Filter mit 20 FPS. Durch die Aktivierung des Gauß-Filters sank diese auf 14 FPS.

Im Vergleich zu den Hard-Shadow-Verfahren liegen die Frameraten hier teilweise deutlich unter den dort erreichten. Im Gegenzug wird die Szene durch die Darstellung von weichen Schatten deutlich authentischer. Dazu muss gesagt werden, dass die gezeigten Verfahren noch große Optimierungsreserven bieten. So kann beispielsweise ein effektiveres Filterverfahren oder die Reduzierung der benötigten Rendering-Passes die Framerate deutlich erhöhen.

11 Fazit

Ziel dieser Arbeit war die Implementation eines Shadow-Viewports in das Szenegraphensystem OpenSG. Dieses sollte dieser die Möglichkeit bieten, aus mehreren Schattenverfahren zu wählen, um anhand der erzielten Ergebnisse einen Vergleich der unterschiedlichen Verfahren durchführen zu können. Der Viewport selbst sollte Modular erweiterbar sein. Diese Punkte sind in dieser Arbeit umgesetzt worden.

Für die Erzeugung harter Schatten habe ich das Standard-Shadow-Mapping, das Perspective- und Lisp-Shadow-Mapping sowie das Trapez-Shadow-Mapping vorgestellt und in den Shadow-Viewport implementiert. Durch den Vergleich dieser Verfahren habe ich festgestellt, dass sowohl das Lisp-, als auch das Trapez-Shadow-Mapping die besten Ergebnisse erzielten und qualitativ dem Standard-Shadow-Mapping in nahezu jeder Situation überlegen sind. Das Perspective-Shadow-Mapping liefert im Gegensatz zu diesen beiden Verfahren nur in speziellen Situationen akzeptable Ergebnisse.

Das Trapez-Shadow-Mapping hat im direkten Vergleich zum Lisp-Shadow-Mapping einige Vorteile, weshalb ich es für das besten der hier gezeigten Verfahren für harte Schatten halte. Ein wesentlicher Vorteil ist dessen universelle Einsatzmöglichkeit. Durch den im Gegensatz zum Lisp-Shadow-Mapping verständlichen Focus-Region-Parameter lässt sich die Schattenqualität in nahezu jeder Situation optimal nach den Wünschen des Benutzers an die jeweiligen Gegebenheiten anpassen. Dieser Aspekt ist besonders für das gesetzte Ziel wichtig, den Shadow-Viewport so nutzer- und anwendungsfreundlich wie möglich zu machen. Zudem lässt sich die Qualität dieses Verfahrens durch die gezeigten Verbesserungen noch weiter erhöhen. Im Performance-Test konnte ich zeigen, dass alle perspektivischen Shadow-Mapping-Verfahren im Vergleich zum Standard-Shadow-Mapping mit nur minimalem Mehraufwand berechnet werden können.

Bei den Soft-Shadow-Verfahren habe ich gezeigt, dass weiche Schatten eine Szene deutlich realer wirken lassen. Die mit Abstand beste Schattenqualität lieferte dabei das Perspektivische-PCF-Shadow-Mapping. Da die Performance bei den Soft-Shadow-Verfahren teilweise deutlich unter der der Hard-Shadow-Verfahren liegt, muss für deren Darstellung genügend Rechenleistung vorhanden sein. Eine optimale Lösung für diese aufwendigen Berechnungen bietet das Clustering, wie es von OpenSG angeboten wird. Damit ist es möglich, die Rechenleistung mehrerer vernetzter Rechner zu bündeln und für die Berechnung einer Szene zu verwenden. Dadurch kann beispielsweise das hier gezeigte Perspektivische-PCF-Shadow-Mapping auch für hohe Auflösungen oder große Szenen mit guten Frameraten eingesetzt werden. Des Weiteren bieten die gezeigten Soft-Shadow-Verfahren noch hohes Optimierungspotential, und auch eine andere Filtermethode zur Qualitätsverbesserung kann die Framerate noch steigern.

Die Erkenntnisse aus dieser Arbeit lassen sich dahingehend zusammenfassen, dass heute genügend Möglichkeiten und Verfahren existieren, Schatten in guter Qualität und in Echtzeit in eine Szene einzubinden. Keine Anwendung, sei es im Bereich der Computerspiele oder anderswo, kann und braucht heute mit qualitativ schlechten Schatten auszukommen. Die Erfindung der perspektivischen Shadow Maps führte dazu, dass der bis dahin größte Nachteil der Shadow-Maps, das Aliasing, erheblich verringert wurde. Leider kann man hier nicht pauschal sagen, dass es ein Verfahren gibt, welches in jeder Situation

optimale Ergebnisse liefert. So hat jedes Verfahren seine eigenen Vor- und Nachteile, wie wir in Kapitel 10 gesehen haben. Aus diesem Grund muss vor der Wahl eines zu implementierenden Verfahrens genau analysiert werden, welches Verfahren für die Anwendung am besten geeignet ist.

Ein weiterer allgemeiner Vorteil des Shadow Mappings ist dessen Fähigkeit unabhängig von der Komplexität der Szene zu arbeiten. Aktuelle Anwendungen verfügen über immer höher aufgelöste Szenen mit großer Polygonanzahl. Je komplexer eine Szene wird, desto stärker fällt der Geschwindigkeitsunterschied zwischen dem Shadow Mapping und anderen Verfahren, wie z.B. den Shadow Volumes aus.

Abschließend möchte ich festhalten, dass sich im Bereich der Echtzeitschatten in den letzten Jahren, gerade im Bereich des Shadow Mappings, einiges getan hat. Das Shadow Mapping bietet heute viele Möglichkeiten, die unterschiedlichsten Schatten darzustellen. Es werden immer neue Verfahren vorgestellt, die wieder bessere Ergebnisse liefern, als das beste bis dahin bekannte Verfahren. Diese Tatsache verdeutlicht, wie wichtig es geworden ist, Szenen immer realistischer virtuell nachbilden zu können. Die Darstellung von realistischen Schatten ist und bleibt einer der wichtigsten Schritte in diese Richtung.

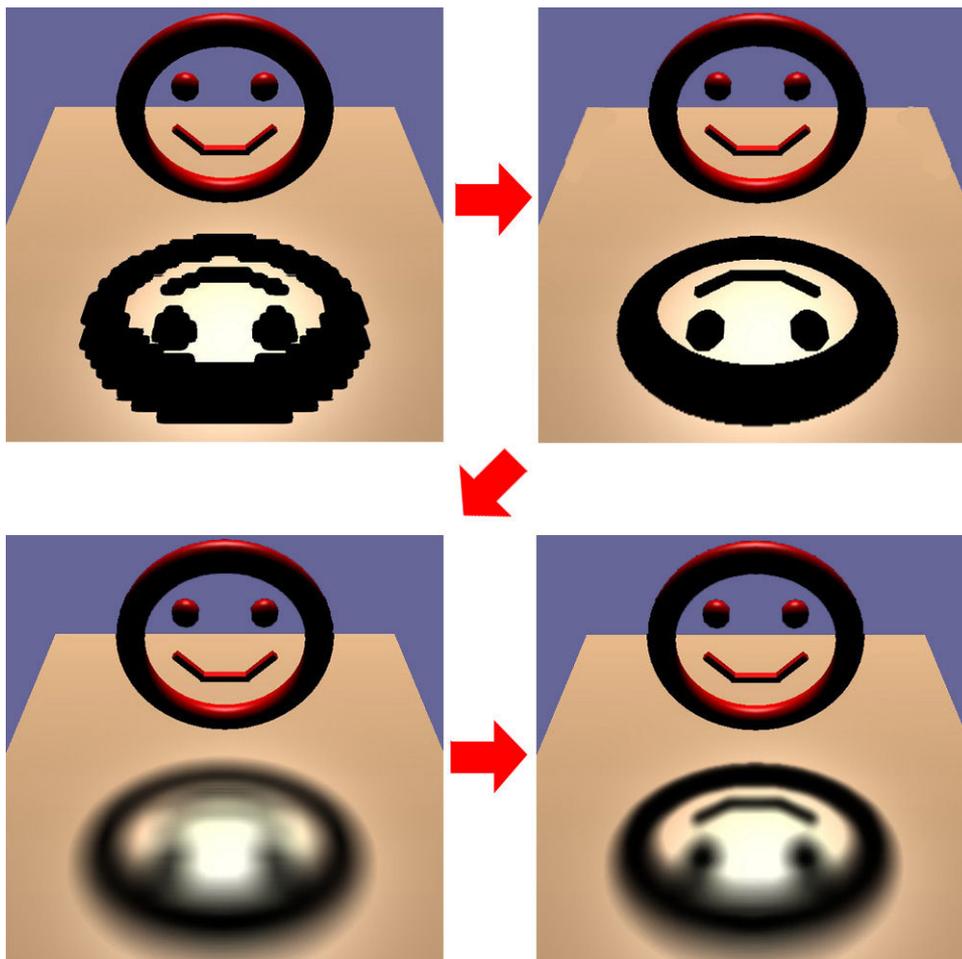


Bild 69 – Shadow Mapping hat viele Gesichter...

Anhang A GLSL Quellcode

PCF-Shadow-Mapping – Vertex Shader:

```
uniform float shadowBias;
uniform mat4 lightPM;
varying vec4 projCoord;

void main(void)
{
    vec4 realPos = gl_ModelViewMatrix * gl_Vertex;
    projCoord = lightPM * realPos;
    projCoord.z -= shadowBias;
    gl_Position = ftransform();
}
```

PCF-Shadow-Mapping – Fragment Shader:

```
uniform sampler2D shadowMap;
uniform float mapSize;
varying vec4 projCoord;
const mat4 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};

float PCF_Filter(vec4 projectiveBiased, float filterWidth, float numSamples)
{
    float stepSize = 2.0 * filterWidth / numSamples;
    projectiveBiased.x -= filterWidth;
    projectiveBiased.y -= filterWidth;
    float blockerCount = 0.0;

    for (float i=0.0; i<numSamples; i++)
    {
        for (float j=0.0; j<numSamples; j++)
        {
            projectiveBiased.x += (j*stepSize);
            projectiveBiased.y += (i*stepSize);

            blockerCount += shadow2D(shadowMap, projectiveBiased.xyz).x;

            projectiveBiased.x -= (j*stepSize);
            projectiveBiased.y -= (i*stepSize);
        }
    }

    float result = ((blockerCount)/(numSamples*numSamples));
    return result;
}

void main(void)
{
    vec4 projectiveBiased = vec4((projCoord.xyz / projCoord.q),1.0);
    projectiveBiased = bias * projectiveBiased;
    float shadowed = PCF_Filter(projectiveBiased, (1.0/mapSize)*2.0, 5);
    gl_FragColor = vec4(shadowed,0.0,0.0,1.0);
}
```

Perspective-PCF-Soft-Shadow – Vertex Shader:

```
uniform float shadowBias;
uniform mat4 lightPM;
varying vec4 projCoord;

void main(void)
{
    vec4 realPos = gl_ModelViewMatrix * gl_Vertex;
    projCoord = lightPM * realPos;
    projCoord.z -= shadowBias;
    gl_Position = ftransform();
}
```

Perspective-PCF-Soft-Shadow – Fragment Shader:

```
uniform sampler2D shadowMap;
uniform float lightSize;
uniform float nearPlaneScale;
uniform float maxPenumbra;
varying vec4 projCoord;
bool foundBlocker = false;
const mat4 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};

float findBlocker(vec4 projectiveBiased, float searchWidth, float numSamples)
{
    float blockerSum = 0.0;
    float blockerCount = 0.0;
    float stepSize = 2.0 * searchWidth / numSamples;
    projectiveBiased.x -= searchWidth;
    projectiveBiased.y -= searchWidth;

    for (float i=0.0; i<numSamples; i++)
    {
        for (float j=0.0; j<numSamples; j++)
        {
            projectiveBiased.x += (j*stepSize);
            projectiveBiased.y += (i*stepSize);
            vec4 shadMapDepth = texture2D(shadowMap, projectiveBiased.xy).xyzw;

            if ((shadMapDepth.z) < (projectiveBiased.z))
            {
                blockerSum += shadMapDepth.z;
                blockerCount++;
            }

            projectiveBiased.x -= (j*stepSize);
            projectiveBiased.y -= (i*stepSize);
        }
    }

    if (blockerCount > 0) foundBlocker = true;
    float result;
    result = (blockerSum / blockerCount);
    return result;
}

float estimatePenumbra(vec4 projectiveBiased, float blocker)
{
    float receiver = projectiveBiased.z;
    float penumbra;

    if (blocker == 0) penumbra = 0;
    else penumbra = ((receiver - blocker) * lightSize) / blocker;
    return penumbra;
}
```

```

float PCF_Filter(vec4 projectiveBiased, float filterWidth, float numSamples)
{
    float blockerCount = 0.0;
    float stepSize = 2.0 * filterWidth / numSamples;
    projectiveBiased.x -= filterWidth;
    projectiveBiased.y -= filterWidth;

    for (float i=0.0; i<numSamples; i++)
    {
        for (float j=0.0; j<numSamples; j++)
        {
            projectiveBiased.x += (j*stepSize);
            projectiveBiased.y += (i*stepSize);
            vec4 shadMapDepth = texture2D(shadowMap, projectiveBiased.xy).xyzw;

            if ((shadMapDepth.z) < (projectiveBiased.z))
            {
                blockerCount++;
            }

            projectiveBiased.x -= (j*stepSize);
            projectiveBiased.y -= (i*stepSize);
        }
    }
    float result = 1.0 - (blockerCount/(numSamples*numSamples));
    return result;
}

void main(void)
{
    vec4 projectiveUnBiased = vec4((projCoord.xyz / projCoord.q),1.0);
    vec4 projectiveBiased = bias * projectiveUnBiased;
    float searchSamples = 6.0;
    float blocker = findBlocker(projectiveBiased, ((nearPlaneScale * lightSize) /
                                                    (projectiveBiased.z)),searchSamples);

    float shadowed = 1.0;

    if (foundBlocker)
    {
        float penumbra;
        penumbra = estimatePenumbra(projectiveBiased, blocker);
        float samples = 8.0;
        if (penumbra > maxPenumbra) penumbra = maxPenumbra;
        shadowed = PCF_Filter(projectiveBiased, penumbra, samples);
    }
    gl_FragColor = vec4(shadowed,0.0,0.0, 1.0);
}

```

Edge Detection – Vertex Shader

```

void main(void)
{
    gl_TexCoord[1] = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = ftransform();
}

```

Edge Detection – Fragment Shader

```
uniform sampler2D depthMap;
const mat4 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};

bool findEdge ()
{
    float stepSize = 1.0/512.0;
    gl_TexCoord[1].x -= stepSize;
    gl_TexCoord[1].y -= stepSize;
    float min = 1.0;
    float max = 0.0;

    for (float i=0.0; i<3.0; i++)
    {
        for (float j=0.0; j<3.0; j++)
        {
            gl_TexCoord[1].x += (j*stepSize);
            gl_TexCoord[1].y += (i*stepSize);
            float sample = texture2D(depthMap, gl_TexCoord[1].xy).z;

            if(sample > max) max = sample;
            if(sample < min) min = sample;

            gl_TexCoord[1].x -= (j*stepSize);
            gl_TexCoord[1].y -= (i*stepSize);
        }
    }

    bool result = false;
    float diff = max - min;
    if (diff > 0.005) result = true;
    return result;
}

void main(void)
{
    gl_TexCoord[1] = bias * gl_TexCoord[1];
    vec4 color = vec4(1.0,1.0,1.0,1.0);
    if (findEdge()) color = vec4(0.0,0.0,0.0,1.0);
    gl_FragColor = color;
}
```

Gauß-Filter – Vertex Shader

```
void main(void)
{
    gl_TexCoord[1] = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = ftransform();
}
```

Gauß-Filter – Fragment Shader

```
uniform sampler2D BlurrMap;
uniform float ShadowRes;
const mat4 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};

float GetGaussianDistribution( float x, float y, float rho )
{
    float g = 1.0 / sqrt( 2.0 * 3.141592654 * rho * rho );
    return (g * exp( -(x * x + y * y) / (2.0 * rho * rho) ));
}

void main(void)
{
    vec2 outp = vec2(0.0, 0.0);
    gl_TexCoord[1] = bias * gl_TexCoord[1];

    float fSampleWeights[16];
    vec2 vSampleOffsets[16];
    fSampleWeights[0] = 1.0 * GetGaussianDistribution( 0, 0, 2.0 );
    vSampleOffsets[0] = vec2( 0.0, 0.0 );
    float stepSize = 1.0/ShadowRes;
    vec2 vAccum = vec2(0.0,0.0);

    for( float i = 1.0; i < 15.0; i += 2.0 )
    {
        vSampleOffsets[i + 0] = vec2( i * stepSize, 0.0 );
        vSampleOffsets[i + 1] = vec2( -i * stepSize, 0.0 );
        fSampleWeights[i + 0] = 2.0 * GetGaussianDistribution( float(i + 0), 0.0, 3.0 );
        fSampleWeights[i + 1] = 2.0 * GetGaussianDistribution( float(i + 1), 0.0, 3.0 );
    }

    for(int i = 0; i < 15; i++)
    {
        vAccum += texture2DProj( BlurrMap, gl_TexCoord[1].xyw +
                               vec3(vSampleOffsets[i],0.0) ).xy * fSampleWeights[i];
    }

    for( float i = 1.0; i < 15.0; i += 2.0 )
    {
        vSampleOffsets[i + 0] = vec2( 0.0, i * stepSize );
        vSampleOffsets[i + 1] = vec2( 0.0, -i * stepSize );
        fSampleWeights[i + 0] = 2.0 * GetGaussianDistribution( 0.0, float(i + 0), 3.0 );
        fSampleWeights[i + 1] = 2.0 * GetGaussianDistribution( 0.0, float(i + 1), 3.0 );
    }

    for(int i = 0; i < 15; i++)
    {
        vAccum += texture2DProj( BlurrMap, gl_TexCoord[1].xyw +
                               vec3(vSampleOffsets[i],0.0) ).xy * fSampleWeights[i];
    }

    outp = vAccum/2.0;

    if(outp.x < 1.0 && outp.y < 1.0)
    {
        //Korrigierte Schatten werden ohne den Faktor oft zu hell
        outp.x = 1.1*(texture2DProj(BlurrMap, gl_TexCoord[1].xyw).x);
    }

    vec2 resultYZ = texture2DProj( BlurrMap, gl_TexCoord[1].xyw ).yz;
    gl_FragColor = vec4(outp.x,resultYZ,1.0);
}
```

Combine – Vertex Shader:

```
void main(void)
{
    gl_TexCoord[1] = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = ftransform();
}
```

Combine – Fragment Shader

```
uniform sampler2D colorMap;
uniform sampler2D shadowFactorMap;
const mat4 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};

void main(void)
{
    gl_TexCoord[1] = bias * gl_TexCoord[1];
    vec3 color = texture2DProj(colorMap, gl_TexCoord[1].xyw).xyz;
    color *= texture2DProj(shadowFactorMap, gl_TexCoord[1].xyw).x;
    gl_FragColor = vec4(color, 1.0);
}
```

Anhang B Quellenverzeichnis

- [Q01] „OpenSG Tutorial“
<http://www.opensg.org>
- [Q02] „Schatten“,
Marc Stamminger 2003
- [Q03] „Projizierte Schatten in OpenGL“, 2003
Stephan Drab
Fachhochschule Hagenberg
- [Q04] „Practical and Robust Shadow Volumes“, 2002
NVIDIA Corporation
- [Q05] “Shadow Mapping: Casting curved shadows on curved surfaces”,
Paul Baker
- [Q06] “Anti-aliasing and Continuity with Trapezoidal Shadow Maps”, 2004
Tobias Martin, Tiow-Seng Tan
National University of Singapore
- [Q07] “Generalized Trapezoidal Shadow Mapping for Infinite Directional Lighting”, 2004
Graham Aldridge
University of Canterbury New Zealand
- [Q08] “Shadow Mapping”, 2004
Nico Hempe
Universität Koblenz-Landau
- [Q09] “Fundamentals of Texture Mapping and Image Warping”,
Paul Heckbert's Master Thesis
- [Q10] “Practical Shadow Mapping”, 2002
Stefan Brabec, Thomas Annen, Hans-Peter Seidel
Max-Planck-Institut für Informatik Saarbrücken
- [Q11] “Perspective Shadow Maps”, 2002
Marc Stamminger, George Drettakis
- [Q12] “Adaptive Shadow Maps”, 2001
Randima Fernando, Sebastian Fernandez, Kavita Bala, Donald P. Greenberg
Cornell University
- [Q13] “Perspective Shadow Maps”, 2004
Gary King
NVIDIA Corporation
- [Q14] “GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time
Graphics”, 2004
Randima Fernando
NVIDIA Corporation
- [Q15] “Percentage-Closer-Soft-Shadows”, 2005
Randima Fernando
NVIDIA Corporation
- [Q16] “Image-Space Rendering Techniques”, 2002
Guennadi Riguer
Ati Corporation

- [Q17] "Soft-Edged Shadows", 2004
Anirudth.S Shastry
- [Q18] "Soft Shadows", 2004
NVIDIA Corporation
- [Q19] "Light Space Perspective Shadow Maps", 2004
Michael Wimmer, Daniel Scherzer, Werner Purgathofer
Technische Universität Wien
- [Q20] „Shadow Maps“, 2002
Timo Ahokas
Helsinki University of Technology
- [Q21] "Shadow Mapping with Today's OpenGL Hardware" 2002
GAME Developers Conference 2000
- [Q22] „A comparison of Shadow Algorithms“, 2005
Joen Sinholt
Technical University of Denmark
- [Q23] "Single Sample Soft Shadows using Depth Maps",
Stefan Brabec, Hans-Peter Seidel
Max-Planck-Institut für Informatik
- [Q24] „Hardware Accelerated Soft Shadows using Penumbra Quads“,
Jukka Arvo, Jan Westerholm
- [Q25] "GL2 Shading Language", 2003
Bill Licea-Kane, Ati Research Inc.
- [Q26] "The OpenGL Shading Language", 2004
John Kessenich, Dave Baldwin, Randi Rost
- [Q27] "GLSL Tutorial",
LightHouse 3D
- [Q28] "OpenGL 2.0 specification", 2004
Mark Segal, Kurt Akeley
- [B01] „World of Warcraft“,
Blizzard Entertainment
<http://www.blizzard.com/>
- [B02] "Doom3", "Quake 4",
id Software
<http://www.idsoftware.com/>
- [B03] Pixar
<http://www.pixar.com>
- [B04] "Warhammer 40k – Dawn of War",
THQ
<http://www.thq.com/>
- [B05] "Battlefield 2",
Electronic Arts
<http://www.eagames.com>
- [M01] 3D-Modelle

3D CAFE
<http://www.3dcafe.com>