Hybride Schattengenerierung mittels Shadow Maps und Shadow Volumes

Studienarbeit
im Fach Informatik
angefertigt am Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

vorgelegt von
Thomas Möck

Betreuer: Frank Firsching
Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Bearbeitungszeitraum: 01. Juni 2005 bis 01. Februar 2006

Kurzfassung

Gegenstand der hier vorgestellten Arbeit ist ein Verfahren zur Berechnung und Darstellung von Schatten in einer virtuellen Szene in Echtzeit. Es basiert auf den beiden gebräuchlichen Methoden „Shadow Mapping“ und „Shadow Volumes“, wobei es die Nachteile beider Algorithmen minimiert. Die erarbeitete Lösung „Hybrid Shadows“ ist somit genauso generell einsetzbar wie Shadow Maps, jedoch werden die Aliasing-Artefakte, die das größte Problem von Shadow Maps darstellen, minimiert.

Zu dieser Arbeit wurde ein Programm erstellt, welches das vorgestellte Verfahren „Hybrid Shadows“ implementiert.

Schlagwörter: Schatten, Hybrid Shadows, Shadow Maps, Shadow Volumes

Abstract

Subject matter of this work is a method to calculate and display shadows in a virtual scene in real-time. It is based on the two common algorithms “Shadow Mapping” and “Shadow Volumes” whereas it minimizes the drawbacks of these techniques. The developed solution “Hybrid Shadows” is all-purpose like Shadow Maps but the aliasing artefacts are minimized, which are the biggest problems of Shadow Maps.

There has also been implemented a program for this work which accomplishes the introduced method “Hybrid Shadows”.

Keywords: Shadows, Hybrid Shadows, Shadow Maps, Shadow Volumes

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

FBO Framebuffer-Objekt bzw. Framebuffer Object

VBO Vertexbuffer-Objekt bzw. Vertexbuffer Object

VTF Vertex Texture Fetch

Vorwort

Das hier vorgestellte Hybrid Shadow Verfahren wurde von mir am Lehrstuhl für Grafische Datenverarbeitung an der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt.

Motivation und Idee zu dieser Arbeit entstanden durch das Paper von McCool [McC00], welches aus dem Jahr 2000 stammt. Es beschreibt ein Verfahren, das eine Untersuchung und die dafür nötige Weiterentwicklung mit heutigen Mitteln lohnt, denn trotz vieler Verbesserungen gibt es nach wie vor kein perfektes Verfahren zur Schattenberechnung und -darstellung. Daher besteht Bedarf für Forschung und neue Ideen.

1 Einleitung

1.1 Was sind eigentlich Schatten und wofür brauchen wir sie?

Schatten sind in der Realität allgegenwärtig. Immer wenn Licht vorhanden ist, gibt es auch Schatten. Doch was sind Schatten genau?

In dieser Arbeit wird „Schatten“ wie folgt definiert: Ein Schatten ist der Effekt der auftritt, wenn ein Objekt Licht davon abhält, auf ein anderes Objekt zu treffen. The Free Dictionary [DictSha] nennt Schatten „An area that is not or is only partially irradiated or illuminated because of the interception of radiation by an opaque object between the area and the source of radiation“ und „The rough image cast by an object blocking rays of illumination”. Schatten ist also das durch die Lichtquelle und den Gegenstand erzeugte Projektionsbild.

Nun könnte man denken, Schatten wären relativ nutzlos. Wer will schon Schatten wenn er ein Licht anmacht? Die Computergrafik böte ja durchaus die Möglichkeit, Schatten zu deaktivieren bzw. erst gar nicht zu aktivieren. Warum sollte man also diesen Effekt erzielen wollen, wo er doch sogar noch zusätzliche Rechenleistung in Anspruch nimmt? Tatsache ist, dass Schatten dazu dienen, eine Szene realistischer zu gestalten, sowie das räumliche Vorstellungsvermögen zu verbessern. Da jede dreidimensionale Szene i.d.R. auf zwei Dimensionen abgebildet werden muss (z.B. auf den Computerbildschirm), helfen gerade Schatten enorm dabei, die genaue Lage eines Objektes zu erkennen. Nicht zuletzt werden Schatten von Künstlern (sei es nun beim Film, bei der Fotografie oder in der Computergrafik) gerne verwendet, um z.B. Schockeffekte hervorzurufen oder einer Szene eine bestimmte Stimmung zu verleihen.

1.2 Bisherige Schattenberechnungsmethoden in der Computergrafik

Es gibt mittlerweile eine Reihe von Möglichkeiten, Schatten in Echtzeit – Computergrafik zu simulieren und darzustellen. Die wichtigsten beiden sind Shadow Maps [Wil78] und Shadow Volumes [Cro77]. Beide Verfahren haben in den letzten Jahren viele Modifikationen und Optimierungen erfahren, wodurch einerseits Qualität und Geschwindigkeit sehr verbessert werden konnten, andererseits einige Probleme behoben wurden, beispielsweise das Renderproblem bei Shadow Volumes, wenn sich die Kamera im Schatten befindet [Eve02].

1.2.1 Shadow Volumes

Der Shadow Volume Algorithmus wurde 1977 von Frank Crow [Cro77] erfunden. Bei diesem geometriebasierten Verfahren werden die Schattenvolumina, welche durch die

Lichtquelle und die Objekte der Szene bestimmt werden, berechnet und dann alles, was sich im Volumen befindet, dunkler gezeichnet als das, was außerhalb ist. Um diese Volumina zu bestimmen, muss vorerst die Silhouette jedes Objekts – von der Lichtquelle aus betrachtet – bestimmt werden, um danach das zugehörige Schattenvolumen durch die Erweiterung der Silhouette ins Unendliche zu erhalten. Einer der gravierendsten Nachteile des Verfahrens besteht darin, dass für eine effiziente Berechnung dieser Silhouetten Verbindungsinformationen der Polygone zueinander vorhanden sein müssen. Des Weiteren ist die Effizienz des Algorithmusses abhängig von der Komplexität der Szene, da die Silhouette, die später das Schattenvolumen formt, für jedes Objekt einzeln berechnet werden muss.

1.2.2 Shadow Maps

Shadow Mapping wurde von Lance Williams im Jahre 1978 entwickelt. Im Gegensatz zu den geometriebasierten Shadow Volumes sind Shadow Maps bildbasiert. Das bedeutet, es ist kein weiteres Wissen über die Geometrie der Szene vonnöten, wie es bei den Shadow Volumes der Fall ist. Auch sucht man hier die restlichen erwähnten Nachteile der Shadow Volumes vergeblich. Jedoch hat der durch Shadow Maps resultierende Schatten durch seine bildbasierte Natur eine begrenzte Auflösung, wodurch sehr leicht Aliasing-Artefakte, also Treppeneffekte im Schatten entstehen können.

1.3 Hybrid Shadows

Das in dieser Arbeit vorgestellte Verfahren – Hybrid Shadows – ist sowohl geometrie- als auch pixel- und bildbasiert, kann allerdings viele der erwähnten Nachteile der anderen Verfahren eliminieren oder zumindest minimieren.

Die Effizienz wird nicht abhängig von der Komplexität der Szene sein, wodurch dieser Algorithmus seine Vorteile gerade bei komplexeren Szenen sehr gut ausspielen kann. Einige Probleme der Shadow Volumes werden verschwinden, weil es nur noch ein Schattenvolumen geben wird und nicht mehrere, wie beim Shadow Volume Algorithmus. Außerdem werden nicht mehr Informationen über die Szene benötigt als bei Shadow Maps, d.h. es werden z.B. keine Verbindungsinformationen der Polygone untereinander nötig sein. Die Aliasing-Artefakte des Shadow Mapping Verfahrens werden weitestgehend minimiert werden können, wozu zwei Optimierungsverfahren für Hybrid Shadows vorgestellt werden.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in sieben Kapitel. Nach der Einleitung in diesem Kapitel befassen sich die folgenden Teile detaillierter mit den bisher bekannten Schattenverfahren Shadow Volumes und Shadow Maps. Kapitel 4 stellt das entwickelte Hybrid Shadow – Verfahren vor, Kapitel 5 taucht tiefer in dessen Implementierungsdetails ein. Als Abschluss werden Geschwindigkeitsmessungen durchgeführt und die Praxisrelevanz un-

tersucht. Die Arbeit endet schließlich mit der Zusammenfassung und Bewertung der Ergebnisse.

2 Shadow Volumes

Wie schon erwähnt, ist das Shadow Volume Verfahren von Crow [Cro77] ein geometriebasiertes Verfahren, welches zur effizienten Berechnung des Schattenvolumens Informationen über den Zusammenhang der einzelnen Polygone in einem Objekt braucht. Nur durch das Wissen, welches Polygon Nachbar von welchem anderen Polygon ist, kann die Silhouette jedes Objekts – von der Lichtquelle aus betrachtet – bestimmt und somit das Schattenvolumen berechnet werden. Mit den heutigen Grafikkarten wird zum Testen der abzudunkelnden Stellen – die Pixel, die im Schattenvolumen liegen – für gewöhnlich der Stencilpuffer verwendet, der auf per-pixel Ebene arbeitet. Ein Nachteil, der sich dadurch ergibt, ist die begrenzte Füllrate des Stencilpuffers, weshalb bei vielen zu rendernden Schattenvolumenpolygonen die Effizienz des Algorithmus leiden kann. Doch wie funktioniert das Verfahren eigentlich genau?

Bei der Shadow Volume Technik wird zuerst die Szene von der Kamera aus in den Tiefenpuffer gezeichnet und gleichzeitig in den Farbpuffer, wobei nur der ambiente und emittierte Anteil des Lichts verwendet wird. Danach muss berechnet werden, welches Fragment (Pixel) sich im Schatten befindet bzw. welches nicht. Mit diesem Wissen kann als nächstes jedes Pixel außerhalb des Schattens noch mal gezeichnet werden, diesmal allerdings mit dem kompletten Licht aktiviert.

Nachfolgend der Pseudo-Code für Shadow Volumes:

```
function renderWithShadowVolume {
   for all rasterized fragments do {
      draw fragment with ambient and emissive lighting
      update the Z-buffer
   }

   computeFragmentsInShadow()

   for all rasterized fragments do {
      if not inShadow(fragment) then {
         draw fragment with diffuse and specular lighting
      }
   }
}
```

Die Schlüsselfunktion im Pseudo-Code ist `computeFragmentsInShadow()`, die entscheiden muss, welche Pixel sich im Schatten befinden und welche nicht. Nachdem die Szene mit ambientem und emittierendem Licht gezeichnet wurde, muss nun die Schattensilhouette für jedes Objekt berechnet werden. Dies geschieht dadurch, dass für jedes Polyon dessen Normale berechnet und mit den Normalen der Nachbarpolygone verglichen wird. Zeigt eine Normale zur Lichtquelle und die andere davon weg, ist die Grenze der beiden Polygone, also die Kante dazwischen, eine Grenze der Silhouette – von der Lichtquelle aus betrachtet. Aus dieser Silhouette wird das Schattenvo-

lumen für jedes Objekt konstruiert, indem es vom Objekt aus von der Lichtquelle weg extrudiert wird. Dadurch entsteht für jedes Objekt der Szene ein Volumen. Nun kann der Stencilpuffer verwendet werden, um entweder Z-Pass oder Z-Fail zu implementieren, zwei mögliche Algorithmen, die dazu dienen, den Schatten letztendlich zu visualisieren. Bei beiden Möglichkeiten wird zunächst das Rendern in den Farbpuffer und in den Tiefenpuffer deaktiviert, da man das Schattenvolumen nicht sichtbar machen will, sondern damit nur ermittelt, welcher Pixel sich nun im Schatten befindet.

Bei Z-Pass wird folgendermaßen vorgegangen:

```
function computeFragmentsInShadow(z-pass) {
  for all shadow casting objects do {
    compute potential silhouette edges (PSE) of the polygonal model
    compute shadow volume polygons from light source(s) and PSE
  }

  for all front facing polygons of the shadow volume do {
    if Z-buffer test passes then {
      increment stencil buffer value
    }
  }

  for all back facing polygons of the shadow volume do {
    if Z-buffer test passes then {
      decrement stencil buffer value
    }
  }
}
```

Bei Z-Pass werden also zunächst alle Front-Faces des Schattenvolumens gezeichnet. Falls ein Pixel den Tiefenpuffer-Test besteht, wenn sich also nichts mehr davor befindet, wird der Stencilpuffer um eins erhöht. Das Gleiche passiert danach mit den Back-Faces, wobei hier im Falle eines Bestehens des Tiefenpuffer-Tests vom Stencilwert eins abgezogen wird. Um zu entscheiden ob sich ein bestimmtes Pixel im Schatten befindet, reicht es nun aus, den entsprechenden Wert im Stencilpuffer zu betrachten. Falls der Wert nach obiger Funktion größer als Null ist, liegt das Pixel im Schatten, ansonsten liegt es im Licht und es muss noch mal mit diffusem und spekularem Licht gezeichnet werden (siehe Pseudo-Code).

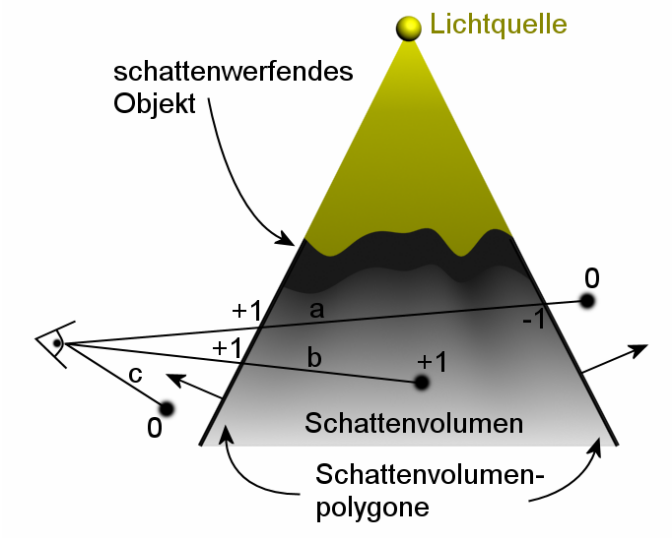

Abbildung 1: Der Z-Pass Algorithmus

Z-Pass funktioniert sehr gut, solange sich die Kamera nicht im Schattenvolumen befindet. Befindet sich die Kamera im Schatten, dreht sich der Effekt jedoch um, d.h. an der Stelle, an der Schatten sein sollte, ist keiner und wo kein Schatten sein sollte, ist Schatten. Eine der elegantesten Lösungen zu diesem Problem stellt Z-Fail [Eve02] dar. Die Idee ist, nicht von vorne nach hinten zu zählen sondern von der Unendlichkeit nach vorne. Der Wert des Stencilpuffers wird verändert, wenn der Tiefenpuffer-Test fehlschlägt, d.h. wenn ein zu rendernder Pixel von etwas verdeckt würde, weil seine Entfernung zur Kamera zu groß ist:

```
function computeFragmentsInShadow(z-fail) {
  for all shadow casting objects do {
    computer potential silhouette edges (PSE) of the polygonal model
    compute shadow volume polygons from light source(s) and PSE
  }

  for all front facing polygons of the shadow volume do {
    if Z-buffer test fails then {
      decrement stencil buffer value
    }
  }

  for all back facing polygons of the shadow volume do {
    if Z-buffer test fails then {
      increment stencil buffer value
    }
  }
}
```

Mit Z-Fail funktioniert die Schattenberechnung bzw. -visualisierung nun in jedem Fall richtig. Ein paar Probleme gibt es aber dennoch. Bei Z-Fail reicht es nicht aus, nur die seitlichen Ränder des Schattenvolumens zu zeichnen. Das Schattenvolumen sollte vielmehr geschlossen sein, d.h. es muss sowohl an der der Lichtquelle abgewandten

als auch an der der Lichtquelle zugewandten Seite Polygone geben, die das Volumen abschließen („capping“). Ansonsten kann es leicht zu Fehlern im resultierenden Schatten kommen. Bei der der Lichtquelle zugewandten Seite werden für gewöhnlich die entsprechenden Polygone des schattenwerfenden Objektes in das Schattenvolumen hinein übernommen.

Während es bei Z-Pass Probleme geben kann, wenn Teile des Schattenvolumens von der Near-Plane weggeclippt werden, gibt es bei Z-Fail Probleme, falls Teile von der Far-Plane geclippt werden. Um dies zu vermeiden werden bei Z-Pass diese Polygone, falls sie sich vor der Near-Plane befinden, auf die Near-Plane projiziert [Kil01]. Für das Problem mit der Far-Plane bei Z-Fail gibt es auch eine elegante Lösung: Damit Polygone des Schattenvolumens nicht von der Far-Plane weggeclippt werden und keine fehlerhaften Schatten resultieren, kann diese auf die Unendlichkeit projiziert werden [Eve02].

Zusammenfassend ist zu den Shadow Volumes zu sagen, dass sie qualitativ sehr gute Ergebnisse liefern. Allerdings gibt es einige Bedingungen, die die Szene erfüllen muss, damit es funktioniert: Durch die Geometriebasierung des Verfahrens ist es nötig, die Szene mit Informationen vorliegen zu haben, die besagen, wie die Polygone jedes Objekts miteinander verbunden sind. Ansonsten kann die Silhouette jedes Objekts nicht effizient berechnet werden. Eine weitere Einschränkung besteht darin, dass es zur Ermittlung der Silhouette und des Schattenvolumens nötig ist, dass die Objekte in mannigfaltiger Geometrie („manifold geometry“) vorliegen. D.h. es müssen folgende Bedingungen erfüllt sein, damit der Shadow Volume Algorithmus funktioniert:

- Jede Kante gehört zu zwei Flächen
- Jeder Vertex ist umgeben von einer Sequenz von Kanten und Flächen
- Flächen schneiden sich gegenseitig nur in gemeinsamen Kanten und Vertices
- Material gibt es nur auf einer Seite einer Fläche

Zwar gibt es einige Arbeiten, die sich diesem Problem der mannigfaltigen Geometrie annehmen, generell ist es jedoch noch nicht gelöst.

Sind die genannten Bedingungen erfüllt, gibt es ein paar Nachteile, was die Effizienz anbelangt: Da die Silhouette aus jedem Objekt berechnet werden muss, hat die Komplexität der Szene großen Einfluss auf die Effizienz des Verfahrens. Denn je mehr und komplexere Objekte, desto mehr und komplexere Silhouetten gilt es zu berechnen. Außerdem kann bei komplexeren Szenen mit vielen Objekten leicht die Füllrate des Stencilpuffers zum Problem werden, da Shadow Volumes heutzutage i.d.R. mit dem Stencilpuffer der Grafikkarte arbeiten und es für jedes Objekt der Szene ein Schattenvolumen gibt.

3 Shadow Maps

Das Shadow Map Verfahren von Williams [Wil78] ist ein Schattenalgorithmus der generell einsetzbar, leicht zu realisieren und effizient ist. Bei Shadow Maps sind nicht, wie bei Shadow Volumes, Informationen über die Geometrie der Objekte nötig, denn das Verfahren arbeitet bildbasiert.

Zuerst wird ein Bild mit den Tiefenwerten der Szene von der Lichtquelle aus gemacht (Depth Map oder Shadow Map genannt). Dadurch, dass der Z-Buffer (Tiefenpuffer) benutzt wird, erhält die Depth Map nur die Tiefenwerte der – von der Lichtquelle aus betrachtet – vordersten Pixel. Nun wird in einem zweiten Pass die Szene von der Kamera aus gerendert, wobei mit einem Shader die aktuelle Tiefe in das Lichtkoordinatensystem transformiert wird und mit der korrespondierenden Tiefe in der Shadow Map verglichen wird. Ist die Tiefe in der Shadow Map kleiner als die transformierte von der Kamera aus, befindet sich der Pixel im Schatten, ansonsten nicht. Nun kann im Shader die Farbe des Pixels bestimmt werden, wie im folgenden Pseudo-Code ersichtlich.

```
intensity = ambient + emitting +
            (zReceiver <= zOccluder) * (diffuse + specular)
```

Falls der Pixel im Schatten liegt, wird nur ambientes und emittierendes Licht verwendet, ansonsten gehen auch die restlichen Anteile in die Berechnung mit ein. `zReceiver` bezeichnet hier den Z-Wert des zweiten Rendering-Passes, `zOccluder` den vom ersten Rendering-Pass, der sich in der Shadow Map befindet. Ist `zReceiver` kleiner oder gleich `zOccluder`, ergibt `(zReceiver <= zOccluder)` 1, ansonsten 0.

Abbildung 2 schildert anschaulich das Prinzip von Shadow Mapping.

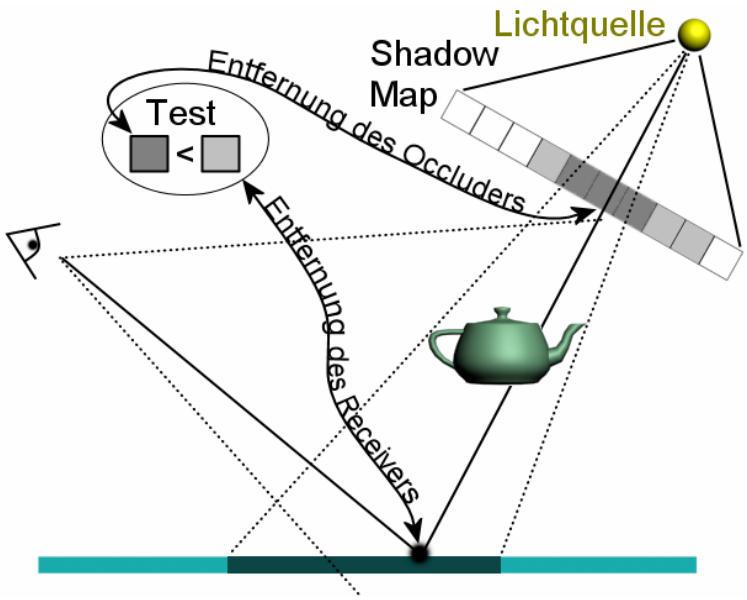

Abbildung 2: Das Prinzip von Shadow Mapping, schematisch dargestellt

Nachfolgend der Pseudo-Code für Shadow Maps:

```
function renderShadowMap {
   render depth buffer from light's point of view to shadow map
   render scene from the camera's point of view
   for all rasterized fragments do {
      determine fragment's xyz position relative to the light
      (transform each fragment's xyz into light's coordinate system)
      A = shadow map(x, y)
      B = z-value of fragment's position in light's coordinate system
      if A < B then {
         fragment is shadowed
      } sonst {
         fragment is lit
      }
   }
}
```

Dadurch, dass der Algorithmus vollständig im Bildkoordinatenraum stattfindet, ergeben sich die oben genannten Vorteile. Diese Tatsache führt jedoch gleichzeitig zu seinem größten Manko: Bedingt durch die endliche Auflösung der Shadow Map ergeben sich leicht Aliasing-Artefakte im Schatten, die zwar durch eine höhere Auflösung der Depth Map reduziert werden, jedoch niemals generell verschwinden. Denn immer wenn ein Pixel der Shadow Map auf mehr als einen Pixel im Framebuffer abgebildet wird, treten diese Pixelartefakte auf, was besonders leicht dann passieren kann, wenn die Kamera sehr nahe an den Schatten kommt und der Schatten sehr langgezogen ist, vgl. Abbildung 3.

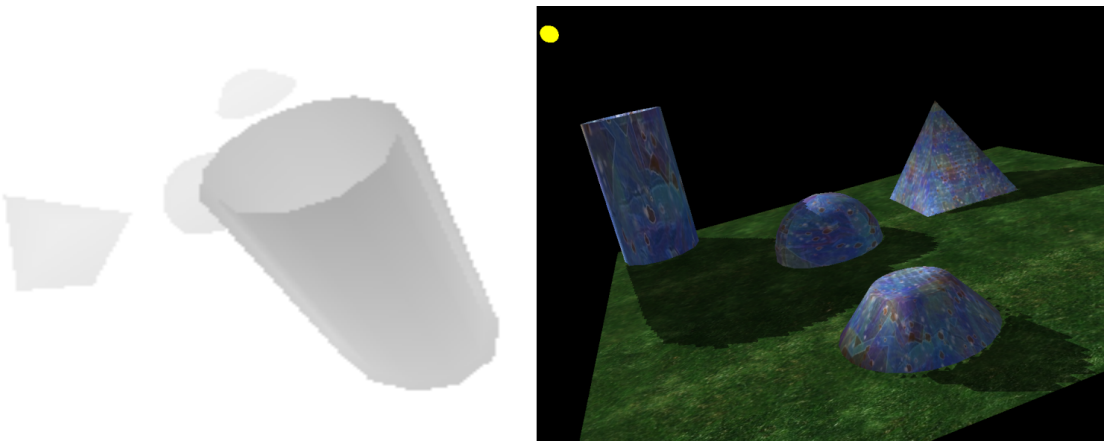

Abbildung 3: Die Depth Map und Aliasing-Artefakte von Shadow Mapping

Ein weiteres Problem ist ein möglicher fehlerhafter Schattenwurf eines Objektes auf sich selbst. Falls ein Pixel im Licht ist, wären im Idealfall die Variablen A und B des obigen Pseudo-Codes identisch. In Wirklichkeit gibt es aber immer Fehler durch die Quantisierung des Tiefenpuffers, wodurch es vorkommen kann, dass A kleiner als B ist, obwohl sich der Pixel eigentlich im Licht befinden müsste. Dadurch werfen Objekte

Schatten auf sich selbst, was natürlich unerwünscht ist. Umgangen werden kann dies durch einen Offset in den Tiefenwerten der Shadow Map, wobei man darauf achten muss, dass man ihn nicht zu groß wählt, da ansonsten der Schatten zu spät, also zu weit von der Lichtquelle entfernt beginnt. Wie groß dieser Wert ist, muss somit im Einzelfall entschieden und eventuell auch an die jeweilige Szene angepasst werden.

4 Hybrid Shadows - Überblick

Im Jahr 2000 wurde von Michael D. McCool das Paper „Shadow Volume Reconstruction from Depth-Maps“ [McC00] veröffentlicht, eine Methode, die Shadow Maps verwendet, um das Schattenvolumen zu konstruieren und Shadow Volumes, um daraus den Schatten zu rendern. Anstatt die Silhouette eines Objektes mit einem Kreuzprodukt für jedes Polygon und Vergleichen mit den Nachbarpolygonen zu bestimmen, wurde diese mit Hilfe von Mustererkennungstechniken anhand einer Depth Map von der Lichtquelle aus bestimmt. Dadurch wurde das Schattenvolumen konstruiert und im weiteren Verlauf der Schatten mittels Shadow Volumes gezeichnet. Bei diesem Hybridverfahren wurden die Nachteile von Shadow Maps und Shadow Volumes weitgehend aufgehoben. Jedoch musste damals, um dies zu erreichen, noch sehr viel mit der CPU gearbeitet werden, was einiges an Performance kostete.

4.1 Motivation

Mit heutigen Mitteln ist natürlich weit mehr möglich. Nicht nur stehen mittlerweile Vertex- und Pixelshader in der Version 3.0 zur Verfügung, aktuelle Grafikkarten unterstützen außerdem viele neue OpenGL-Erweiterungen, womit es letztendlich möglich ist, das Hybrid Shadow Verfahren zu 100% auf die Grafikkarte abzubilden. In dem hier vorgestellten Verfahren wird nicht mehr mit Mustererkennungstechniken gearbeitet, sondern vielmehr die Depth Map direkt als Punktquelle für das Schattenvolumen aufgefasst und weiterverwendet. Dies führt dazu, dass es nur noch ein Schattenvolumen für alle Objekte in der Szene gibt, nicht, wie bei Shadow Volumes, ein Schattenvolumen pro Objekt. Folglich fällt der Nachteil der Shadow Volumes, die begrenzte Füllrate des Stencilpuffers, weg, da mit dem Stencilpuffer bei weitem nicht mehr soviel gezählt werden muss. Auch ist es weder nötig, die Szene mit Informationen über die Lage der Polygone zueinander vorliegen zu haben, noch, dass die Objekte der Szene die Bedingungen der mannigfaltigen Geometrie erfüllen; das Schattenvolumen kann von jeder beliebigen Szene durch die Depth Map (bild-basiert) bestimmt werden. Und da die Silhouette des Schattenvolumens nicht mehr gebraucht wird, muss sie auch nicht zeitraubend mittels des Kreuzproduktes und durch Vergleiche mit benachbarten Polygonen berechnet werden.

Der Nachteil, der von Shadow Maps bleibt, sind die Aliasing-Artefakte, die von der Depth Map herrühren und sich ins Volumen übertragen. Zwar sind diese an Stellen, an denen das Schattenvolumen durch Zufall richtig trianguliert ist, verschwunden oder stark vermindert: An allen anderen sind sie nichtsdestotrotz unvermindert stark, da das Volumen ja in der Auflösung der Depth Map quantisiert ist. In dieser Arbeit werden zwei Lösungen zur Vermeidung bzw. Verminderung dieser Artefakte vorgestellt: Zum einen die richtige Triangulierung des Schattenvolumens an allen Stellen durch Topologieveränderung („Perfect Triangulation“), zum anderen ein Unschärfealgorithmus für

das Volumen, welcher die Z-Werte aneinander annähert („Depth Blurring“). Beide Verfahren finden vollständig auf der Grafikkarte statt.

4.2 Generelles Verfahren

Um das Schattenvolumen mit dem Hybrid Verfahren zu erhalten, werden die Tiefenwerte der Szene zuerst von der Lichtquelle aus in eine Textur (Depth Map) gerendert. Diese Depth Map wird danach direkt als Punktquelle für die Vertices des Schattenvolumens interpretiert. Die Depth Map speichert dazu mit Hilfe eines Fragmentshaders sowohl die Tiefe als auch die XY-Position jedes Pixels in normalisierten Koordinaten der Lichtquelle als floating-point Zahlen in den drei Farbwerten der Textur: Rot, grün und blau. Diese Werte werden beim Zeichnen des Schattenvolumens in Weltkoordinaten transformiert und direkt als Positionen der Vertices des Volumens aufgefasst. Die Schattenvolumenpolygone bestehen somit aus genau so vielen Punkten wie die Textur des vorherigen Schrittes Pixel aufweist und die Positionen dieser Punkte enthält die Textur in ihren RGB-Werten. Der grobe Ablauf des Verfahrens ist in Abbildung 4 dargestellt.

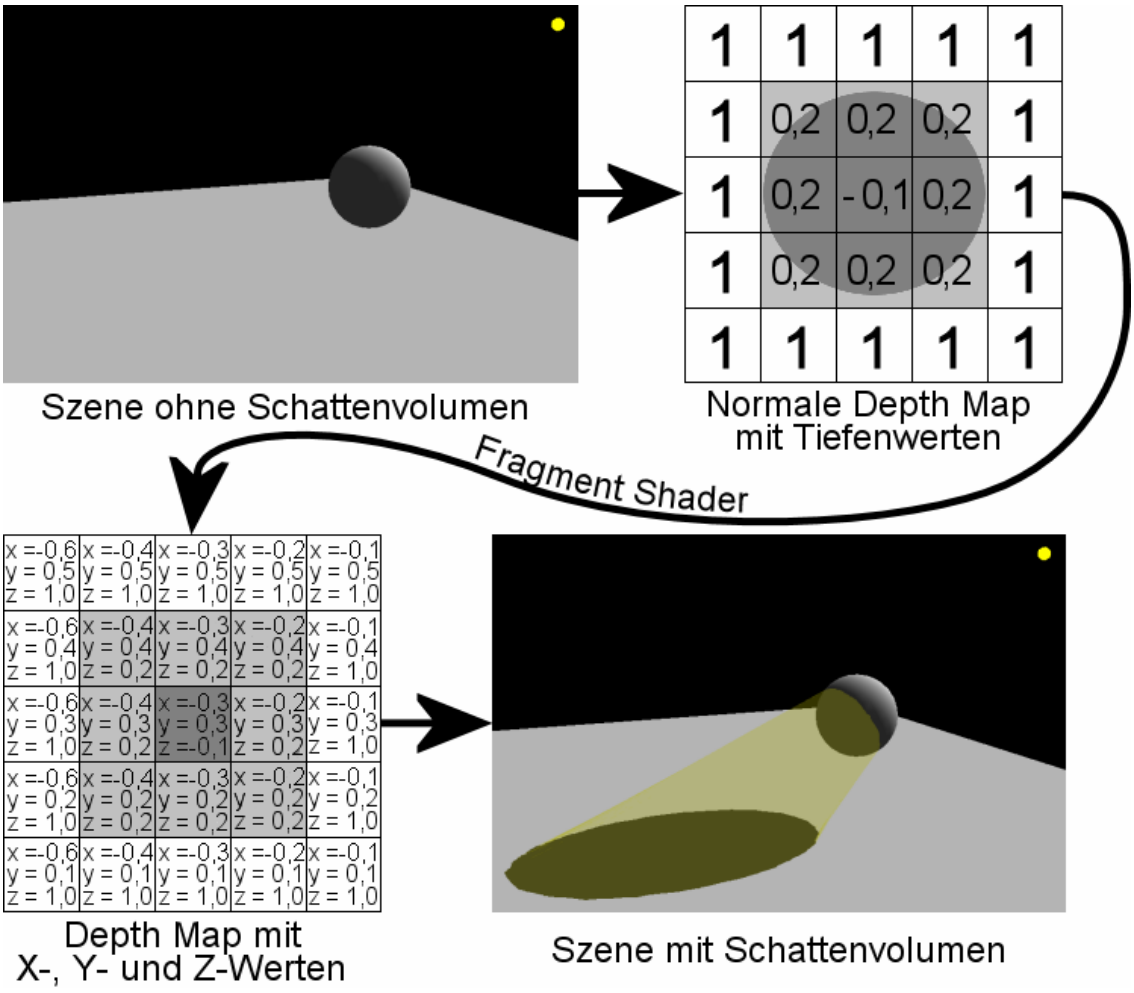

Abbildung 4: Generelles Vorgehen beim Hybrid Verfahren

Nun gibt es zwei Möglichkeiten, die Positionen der Vertices der Schattenvolumenpolygone auf die Werte in der Textur zu setzen:

Die erste ist, die Werte aus der Textur in ein Vertexbuffer-Objekt zu kopieren und dieses als Datenquelle für die Positionen der Vertices herzunehmen. Das Kopieren geschieht durch die Verwendung eines Pixelbuffer-Objekts (PBO) und sollte sehr schnell funktionieren, da die Daten die Grafikkarte laut Dokumentation dabei nicht verlassen sollten.

Die zweite Möglichkeit besteht darin, zuerst eine schachbrettartige Fläche aus darin regelmäßig angeordneten Vertices zu zeichnen (wobei die Fläche aus sovielen Vertices besteht, wie die Textur des vorherigen Schrittes Pixel aufweist) und während des Zeichnens einen Vertexshader zu verwenden, der direkt die Positionswerte der Vertices in der Textur nachschlägt und diese entsprechend setzt. Diese Technik nennt sich „Vertex Texture Fetch“, kurz „VTF“.

Im Rahmen dieser Arbeit wurden beide Möglichkeiten implementiert und getestet, obwohl der Verdacht bestand, dass Vertex Texture Fetch gegenüber der ersten langsamer wäre, da für jeden Vertex die Position einzeln in der Textur nachgeschlagen werden muss. Im Gegensatz zur Methode mit dem Pixelbuffer-Objekt, bei der die gesamten Daten auf einmal kopiert werden. Tatsächlich wurde allerdings festgestellt, dass das Kopieren mit dem PBO bei AGP-Hardware viel langsamer arbeitet als VTF, woraus der Schluss gezogen werden muss, dass das Kopieren mit dem PBO doch die Grafikkarte verlässt. Diese Einsicht wurde in der Tat noch verstärkt, als entdeckt wurde, dass die Geschwindigkeitsdifferenz der beiden Möglichkeiten mit PCI-Express Hardware kleiner ausfällt. Doch dazu später mehr.

4.3 Perfect Triangulation

Die erste Frage, die sich nun aufdrängt, ist natürlich die, was das Berechnen eines Volumens, welches in der Auflösung der Depth Map quantisiert ist, gegenüber dem herkömmlichen Shadow Mapping für einen Vorteil bringt. Denn der resultierende Schatten weist nun natürlich immer noch Aliasing-Artefakte auf, die vom Shadow Mapping bekannt sind. Tatsache ist allerdings, dass die Treppeneffekte im Schatten schon ohne weitere Optimierung des Volumens vermindert sind, da das von der Depth Map extrahierte Schattenvolumen an durchschnittlich der Hälfte der Quads richtig herum trianguliert ist.

Durch die zufällige richtige Triangulierung wird im Idealfall aus einer Ecke im Schatten eine gerade Kante. Auch ohne die Anwendung einer Optimierung verbessert sich die Qualität des Schattens allein durch die Verwendung von Hybrid Shadows bei durchschnittlich 25% der Fälle. Wie in Abbildung 5 zu sehen ist, verschwinden links oben und rechts unten im Schatten einer Kugel die Ecken des Schattens.

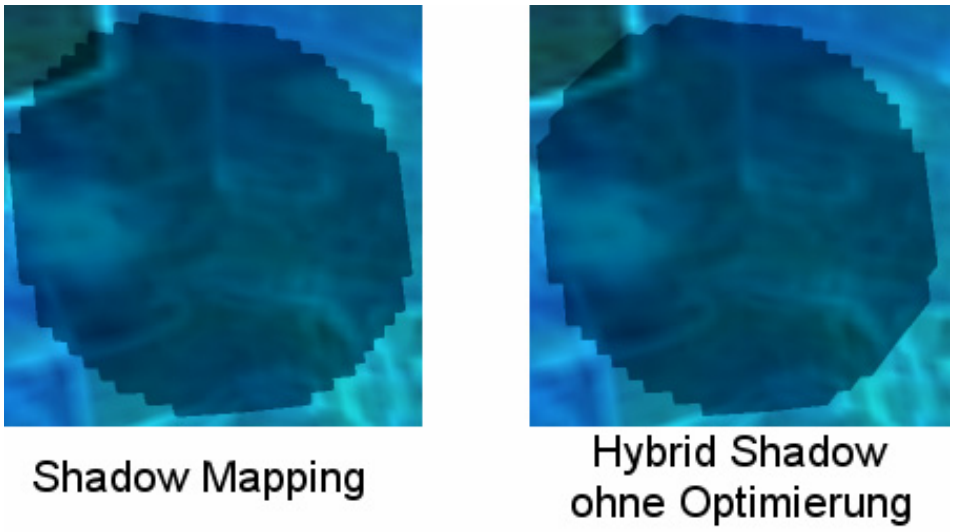

Abbildung 5: Schatten, erzeugt mit Shadow Mapping bzw. mit Hybrid Shadow ohne Optimierung

Nun liegt es nahe, auch die restlichen falsch herum triangulierten Quads in den Schattenvolumenpolygonen zu erkennen, dort die Triangulierung zu verändern und somit vermeidbare Aliasing-Artefakte im Schatten zu eliminieren. Diese Technik wird in dieser Arbeit mit „Perfect Triangulation“ benannt.

Werden nun auch die Fälle erkannt, die noch falsch herum trianguliert sind und entsprechend deren Triangulierung umgekehrt, bleiben keine scharfen Ecken mehr übrig. Alle Quads sind nun so trianguliert, dass der resultierende Schatten die bestmögliche Qualität erreicht (ohne dabei die Position der Vertices der Schattenvolumenpolygone zu verändern). Zwar sind immer noch Artefakte zu sehen und man kann bei genauem Hinsehen auch noch Pixel erkennen. Trotzdem ist die Qualität weit besser, als bei einfachen Shadow Maps.

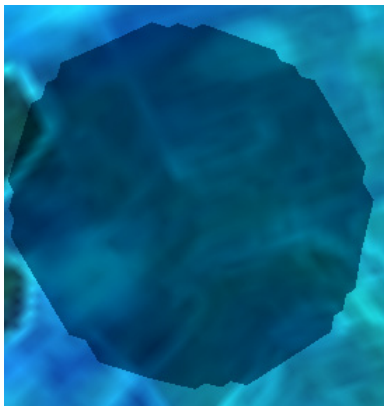

Abbildung 6: Schatten erzeugt durch Hybrid Shadow mit Perfect Triangulation

Doch warum ändert die Veränderung der Triangulierung der Quads des Schattenvolumens eigentlich etwas am resultierenden Schatten?

Betrachten wir ein vereinfachtes Schattenvolumen als Drahtgittermodell, wie in Abbildung 7 gezeigt. Angenommen der Schatten würde, hervorgerufen durch das skizzierte Volumen, entlang der blauen Linie laufen. Offensichtlich ergibt sich aus diesem Volumen eine scharfe Ecke im Schatten, was genau dieselben Aliasing-Artefakte hervorruft, die vom Shadow Map Verfahren bekannt sind (leicht zu erkennen in Abbildung 8).

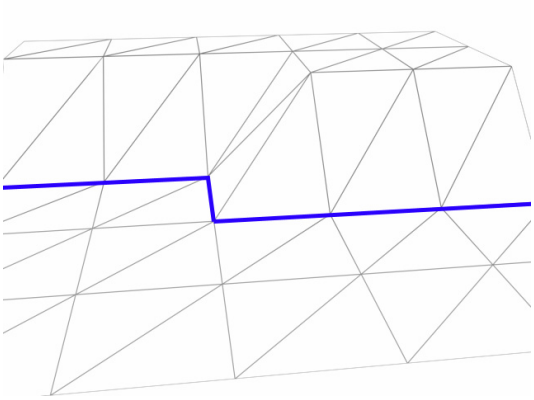

Abbildung 7: Vereinfachtes Schattenvolumen, mit resultierendem Aliasing-Artefakt

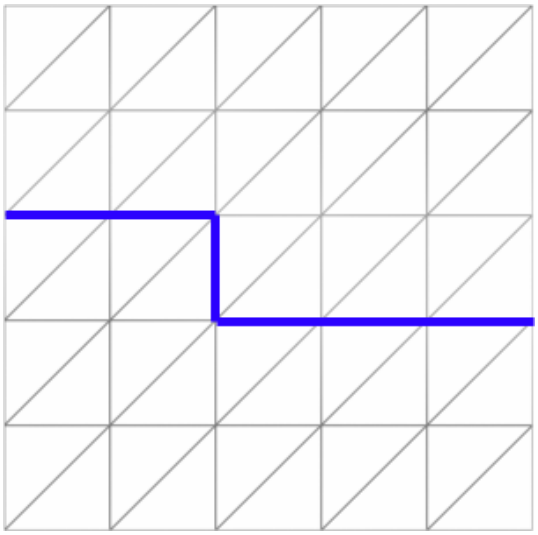

Abbildung 8: Vereinfachtes Schattenvolumen, mit resultierendem Aliasing-Artefakt von oben

Abhilfe schafft die bessere Triangulierung des Schattenvolumens. Der Clou ist: Es müssen dafür keinerlei Positionen der Vertices geändert werden, allein die Reihenfolge der Triangulierung ermöglicht den gewünschten Effekt. An der Stelle, an der die scharfe Ecke auftritt, wird das Quad einfach in umgekehrter Richtung trianguliert, vgl. Abbildung 9. Im Ergebnis (Abbildung 10) ist die störende scharfe Ecke nunmehr verschwunden.

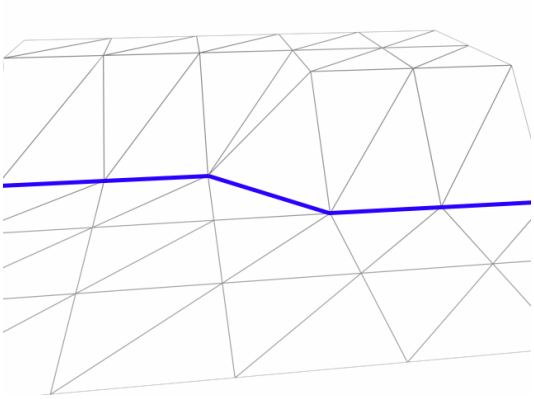

Abbildung 9: Vereinfachtes Schattenvolumen, Perfect Triangulation

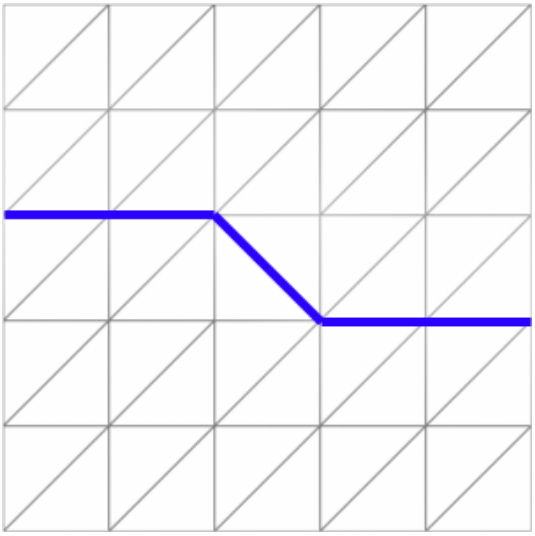

Abbildung 10: Vereinfachtes Schattenvolumen, Perfect Triangulation

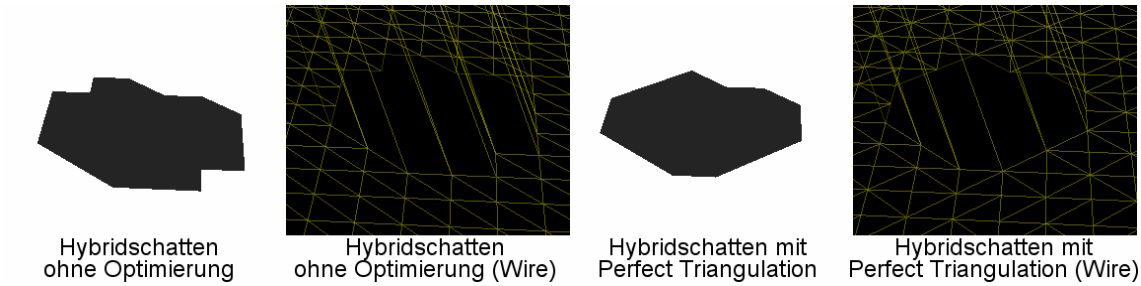

Abbildung 11: Perfect Triangulation in der Praxis

Bleibt die Frage, wie erkannt wird, welches Quad richtig und welches falsch herum trianguliert ist, um die falsche Triangulierung umzudrehen und den Fehler überhaupt beheben zu können. Die erarbeitete Lösung sieht folgendermaßen aus: Betrachtet wird immer ein einzelnes Quad und dessen vier Vertices, siehe Abbildung 12. Standardmä-

ßig läuft die Diagonale des Quads von Punkt B zu Punkt D, es besteht also aus den beiden Dreiecken ABD und BCD. Nun werden vier verschiedene Ebenengleichungen aufgestellt, jeweils mit dreien von den vier Vertices, und der Abstand der jeweiligen Ebene zu dem Vertex berechnet, der nicht in die Ebenengleichung mit eingegangen ist. Falls entweder der Abstand von B zur Ebene (dB) oder der von D zur Ebene (dD) die größten Abstände sind, wird die Triangulierung umgekehrt. Ansonsten wird sie beibehalten. Die Kante eines Quads sollte also niemals den Punkt enthalten, dessen Abstand zur Ebene der restlichen Punkte am größten ist.

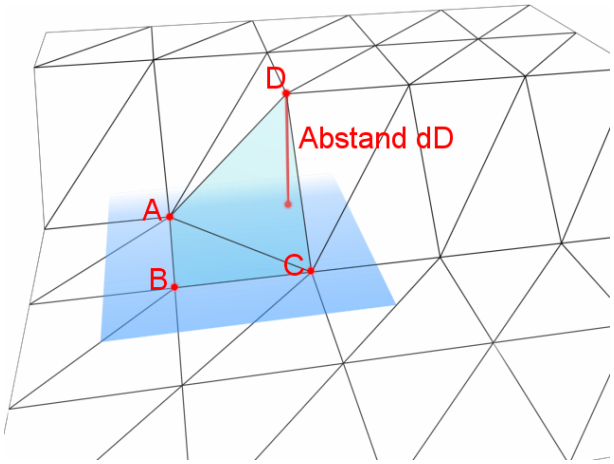

Abbildung 12: Vereinfachtes Schattenvolumen, Ermittlung der richtigen Triangulierung

Nachfolgend der Pseudo-Code des Perfect Triangulation Algorithmusses:

```
float function computeDistance(vertex P,vertex A,vertex B,vertex C) {
   // calculates the distance of vertex P to plane ABC
   vertex normal = crossProduct(vector(A, B), vector(A, C))
   return abs(scalarProduct(vector(A, P), normalize(normal)))
}

function triangulateQuadRight(vertex A, vertex B,
   vertex C, vertex D) {
   // flips the quad if necessary
   float distanceA = computeDistance(A, B, C, D)
   float distanceB = computeDistance(B, A, C, D)
   float distanceC = computeDistance(C, A, B, D)
   float distanceD = computeDistance(D, A, B, C)
   if (distanceB > distanceC and distanceB > distanceA) or
       (distanceD > distanceC and distanceD > distanceA) then {
     flipTriangulation()
   }
}
```

4.4 Depth Blurring

Im Rahmen der vorliegenden Arbeit wurde noch ein zweites Optimierungsverfahren für Hybrid Shadows entwickelt und untersucht: „Depth Blurring“. Der Ansatz ist etwas anders als bei der oben vorgestellten Perfect Triangulation. Während dort die Positionen der Vertices im Schattenvolumen unverändert blieben und nur die Topologie geändert wurde, wird bei Depth Blurring die Position der Vertices verändert. Im Prinzip ähnelt diese Methode einem Blur-Effekt. Die Tiefe der Vertices des Schattenvolumens – von der Lichtquelle aus betrachtet – wird mit denen der jeweiligen Nachbarvertices vermischt und somit angeglichen. Die scharfen Ecken verschwinden, da die Quantisierung des Schattenvolumens in der Z-Richtung buchstäblich aufgehoben wird.

Beim Shadow Mapping gibt es eine Optimierung, die ähnlich arbeitet: Percentage Closest Filtering. Bei dieser Praktik wird im Fragmentshader die Intensität eines Pixels davon mitbestimmt, ob die Nachbarpixel des aktuellen Fragments im Schatten liegen oder nicht. Dann wird eine Interpolation durchgeführt, um zu bestimmen, wie hell der Pixel letztendlich wird. Im Ergebnis führt dies dazu, dass der (vormals pixelige) Schatten verschwommen dargestellt wird, wodurch die Aliasing-Artefakte abgeschwächt werden. Leider ist der Schatten mit Percentage Closest Filtering auch an Stellen unscharf, an denen es im wirklichen Leben eigentlich scharfe Schatten geben sollte. Die Pixelartefakte sind zwar vermindert, jedoch trotzdem noch vorhanden, besonders wenn sich die Kamera nahe am Schatten befindet, da es trotz allem ein bildbasiertes Verfahren bleibt. Mit der Hybrid Shadow Technik erhält man jedoch ein Volumen, von dem die Positionen der Vertices prinzipiell beliebig verändert werden können. Dies machen wir uns bei Depth Blurring zunutze.

Das Ergebnis ist erstaunlich. Abbildung 13 zeigt den Schatten einer Kugel, erzeugt durch Shadow Mapping bzw. durch Hybrid Shadow, optimiert mit Depth Blurring. Beide Beispiele verwenden dieselbe Größe für die Depth Map.

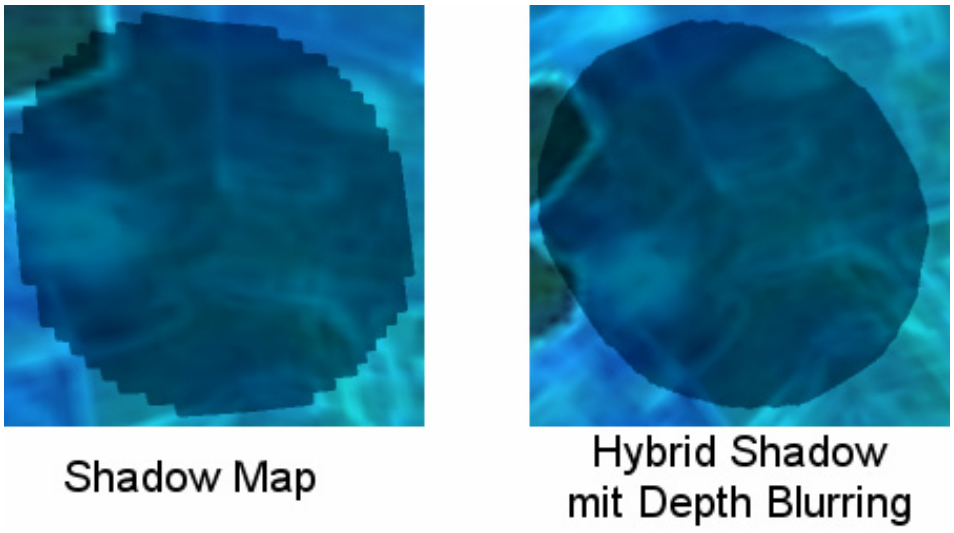

Abbildung 13: Vergleich eines herkömmlichen Schattens mit Shadow Mapping und eines Schattens erzeugt mit Hybrid Shadow optimiert mit Depth Blurring

5 Hybrid Shadows – Implementierungsdetails

5.1 Die Rendering Pipeline heutiger Grafikkarten

Eine dreidimensionale Szene in der Computergrafik besteht aus Punkten („Vertices“), Linien und Flächen, die aus Dreiecken („Triangles“) aufgebaut sind und „Polygone“ genannt werden. Alle diese Objekte haben Koordinaten, die ihre Position im Raum beschreiben. Diese Koordinaten müssen einige Modifikationsschritte durchlaufen, damit sie letztendlich entsprechend der Absicht des Entwicklers auf zwei Dimensionen (z.B. dem Bildschirm) abgebildet werden können. Die Rendering Pipeline einer Grafikkarte führt diese Schritte nacheinander mit jedem Element, welches je nach Stufe in der Rendering Pipeline entweder ein Vertex oder ein Pixel ist, durch, wobei mehrere gleichzeitig in verschiedenen Stufen bearbeitet werden können, wie auch ein Skilift mehrere Skifahrer gleichzeitig den Berg hoch befördert. Abbildung 14 zeigt die groben funktionalen Stufen einer Rendering Pipeline heutiger Grafikkarten.

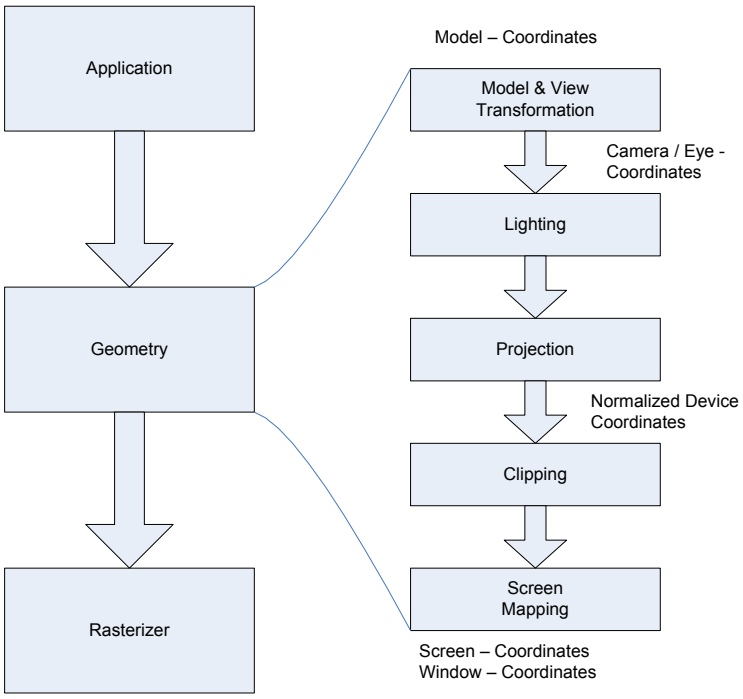

Abbildung 14: Die Graphics Rendering Pipeline

Die „Application“, also die Anwendung berechnet bzw. bestimmt die Objekte und deren Koordinaten, sowie Transformationen, die auf ihnen angewandt werden sollen. Am Ende dieser Stufe gibt die Anwendung die Daten an die Geometriestufe der Grafikkarte weiter.

Die Geometrie arbeitet mit Polygonen und Vertices. Sie erhält die Objekte der Szene in Model-Koordinaten, d.h. jedes Objekt befindet sich im geometrischen Nullpunkt und muss zuerst mit Hilfe der Model Transformation in Weltkoordinaten transformiert werden. Nach diesem Schritt befindet sich jedes Objekt am gewünschten Ort in der gewünschten Rotation und Skalierung. Allerdings würde der Betrachter die Welt nun von ihrem geometrischen Nullpunkt aus betrachten, weshalb noch eine View Transformation vonnöten ist, die alle Objekte so transformiert, dass sich die Kamera – also das Auge des Betrachters – im Nullpunkt befindet und entsprechend ihrer Blickrichtung rotiert ist. Das Ergebnis sind die Kamerakoordinaten oder auch Koordinaten im „Eye Space“ genannt. Sowohl die Model- als auch die View-Transformation geschehen mit jeweils einer 4x4-Matrix-Multiplikation. Nun wird die Beleuchtung jedes Vertex durch Materialeigenschaften, Texturen und Eigenschaften der Lichtquelle(n) berechnet. Daraufhin erfolgt die Berechnung der Projektion in einen Einheitswürfel. Sie funktioniert wieder mit einer 4x4-Matrix und ist anschaulich die Transformation von drei zu zwei Dimensionen. Das Ergebnis sind „Normalized Device Coordinates“, also Koordinaten die i.d.R. im Wertebereich von -1 bis 1 in jeder Dimension reichen. Alle Objekte außerhalb des Einheitswürfels werden verworfen, da sie von der Kamera aus nicht sichtbar sein werden. Als nächstes werden alle Objekte, die den Einheitswürfel schneiden, sich also weder vollständig außerhalb noch vollständig innerhalb des Würfels befinden, geclippt. D.h. der Anteil jedes Objekts, der außerhalb des Würfels ist, wird verworfen und dafür werden neue Vertices generiert, um ein Objekt zu erhalten, welches sich vollständig im Würfel befindet und dennoch genauso aussieht, wie das gesamte Objekt aussähe, mit Ausnahme des Teils, der sich außerhalb des Würfels befindet. Danach kommt das Screen Mapping, welches die XY-Koordinaten der Vertices so verändert, dass der Einheitswürfel auf das Fenster abgebildet wird, in dem das Bild dargestellt werden soll. Das Ergebnis sind „Screen Coordinates“ also Bildschirmkoordinaten. Nimmt man die Z-Koordinaten dazu, erhält man die „Window Coordinates“, also die Fensterkoordinaten. Am Ende der Geometrie-Stufe werden die berechneten Daten der Rasterisierung weitergegeben.

Die Rasterisierung konvertiert alle Objekte zu Pixeln. Dabei werden sowohl eventuelle Texturen berücksichtigt, wie auch die Z-Reihenfolge der Objekte (Stichwort: Z-Buffer Algorithmus).

5.2 Generelle Implementierungsinformationen und OpenGL-Extensions

Um Methoden zur Implementierung von Hybrid Shadows zu entwickeln, zu testen und zu veranschaulichen, wurde ein Programm in C++ geschrieben. Als Grafik-API wurde OpenGL verwendet, für die Vertex- und Pixelshader die OpenGL Shadersprache GLSL, für die grafische Benutzerschnittstelle die Bibliothek GLUI, für die OpenGL-Extensions GLEW und für das Laden von Texturen die Bildbibliothek DevIL. Für einige elementare Mathematikfunktionen wurden teilweise C++ Klassen von Paul Baker

[Bak02] verwendet. Zum Einlesen von Wavefront OBJ-Szenendateien wurde ein einfacher Parser geschrieben, der die dem Programm in der Kommandozeile übergebene Szenendatei beim Starten einliest.

Gearbeitet wurde mit einigen OpenGL-Extensions, wovon die wichtigsten kurz vorgestellt werden.

5.2.1 Framebuffer Objects

Um zuerst eine Depth Map zu rendern und diese danach zum Zeichnen des eigentlichen Bildes weiter zu verwenden, ist Offscreen-Rendering nötig. Dies gilt sowohl für die Implementierung von Shadow Maps, wie auch für Hybrid Shadows. Bisher wurde dies gern mit den sehr unübersichtlichen pbuffers realisiert. EXT_framebuffer_object, kurz FBO, ist eine relativ neue OpenGL-Erweiterung, die die gesamte pbuffer-Funktionalität ersetzt. Sie stellt eine Schnittstelle zur Verfügung, um auf andere Rendering-Ziele zu zeichnen als die Buffer, die OpenGL vom Fenster-System zur Verfügung gestellt werden. Insbesondere sind das Offscreen-Render-Buffers, wie z.B. Texturen oder sog. Renderbuffers, allgemein werden diese als „Attachments“ bezeichnet. Diese Rendering-Ziele können leicht nach Belieben zur Laufzeit ein- oder ausgehängt werden, es muss nur darauf geachtet werden, dass das Framebuffer-Objekt bei der Benutzung „framebuffer-complete“ ist. Dazu müssen die eingehängten Attachments bestimmte Bedingungen erfüllen, auf die hier nicht näher eingegangen wird. Nachzulesen ist dies alles in der Spezifikation von EXT_framebuffer_object [FBO05].

5.2.2 Vertexbuffer Objects

Im Programm gibt es einige Daten, auf die in jedem Frame – evtl. auch mehrmals pro Frame – zugegriffen werden muss. Dies betrifft z.B. das Vertexindex-Array für den Triangle-Strip des Schattenvolumens, oder die Positionen der Vertices des Schattenvolumens, bevor sie mittels Vertex Texture Fetch verändert werden. Die OpenGL-Erweiterung ARB_vertex_buffer_object [VBO03], kurz VBO, stellt eine Möglichkeit zur Verfügung, Daten direkt auf der Grafikkarte in einem high-performance Speicher abzulegen und davon zu lesen. Durch Vertexbuffer-Objekte wird somit kein konventioneller Speicherzugriff auf den Arbeitsspeicher benötigt, da alles direkt auf der Grafikkarte passiert.

5.2.3 Vertex Programm, Shader Modell 3.0

Zur Verwendung der Depth Map als Positionsquelle für die Vertices des Schattenvolumens gibt es die Möglichkeit, die Depth Map auszulesen und in ein Vertexbuffer-Objekt zu kopieren, dessen Inhalt dann direkt als Renderingdaten verwendet wird. Dieses Kopieren nimmt jedoch u.U. enorm viel Zeit in Anspruch, was die Performance des gesamten Hybrid Shadow Algorithmusses drastisch reduzieren kann. Zur Lösung dieses Problems wird Vertex Texture Fetch eingesetzt, d.h. die Depth Map wird beim Zeichnen des Schattenvolumens als Textur gebunden und die Positionen der Vertices

werden direkt mit einem Vertexshader, der auf diese Textur zugreift, gesetzt. Dafür wird das Shader Modell 3.0 benötigt, welches einen Texture-Lookup im Vertexshader ermöglicht. Die Extension dafür ist NV_vertex_program3 [VP304].

5.2.4 Depth-Clamping

Für das in dieser Arbeit vorgestellte Verfahren ist es nötig, dass auf die Far-Plane gerendert werden kann. Die OpenGL-Erweiterung NV_depth_clamp [DCL03] bewirkt, dass jeder Vertex, der von der Near- oder Far-Plane geclippt würde, direkt auf der Near- bzw. Farplane gezeichnet wird.

5.2.5 Doppelseitiger Stencilpuffer

Nach dem Extrahieren des Schattenvolumens aus der Depth Map ist der Hybrid Shadow Algorithmus ähnlich dem von Stencil Shadow Volumes [Cro77]. Entschieden, ob ein Fragment innerhalb oder außerhalb des Schattens liegt, wird somit auch bei Hybrid Shadow Volumes durch den Stencilpuffer. Würde der herkömmliche (einseitige) Stencilpuffer verwendet, müssten für den Z-Fail (wie auch für Z-Pass) Algorithmus einmal alle Front-Faces und danach alle Back-Faces des Schattenvolumens gezeichnet werden, um mit dem Stencilpuffer zu zählen. Genau für solche Anwendungen wurde die OpenGL-Extension EXT_stencil_two_side [Ste03] entworfen. Ein doppelseitiger Stencilpuffer, der in einem Durchgang Front- und Back-Faces genau so behandelt, wie der Programmier es verlangt. Da nur noch ein Rendering-Pass nötig ist, verbessert dies natürlich die Performance des Algorithmusses.

5.2.6 Primitive Restart

Um z.B. mehrere Triangle-Strips zu rendern, wäre ein Array und ein Rendering-Aufruf pro Strip nötig. Um dies zu vermeiden, gibt es die OpenGL-Erweiterung NV_primitive_restart [PrR02], welche durch eine definierbare Indexnummer erkennt, wann eine neue Primitive (z.B. ein Triangle-Strip) beginnt. Der Overhead dieser Extension wird als extrem klein angegeben.

5.3 Die Hybrid Shadow Implementierung

Bei der für diese Arbeit angefertigten Implementierung gibt es verschiedene Möglichkeiten, zwischen denen der Benutzer wählen kann. Zum einen kann er den Schattentyp auswählen:

- Kein Schatten: Es wird kein Schatten berechnet und die Szene ganz ohne Schatten gezeichnet.
- Shadow Map: Der Schatten wird nach dem Shadow Map Verfahren berechnet und gezeichnet.

- Hybrid Shadows: Das entwickelte Hybrid Shadow Verfahren kommt zur Anwendung und der Schatten wird mit dem Stencil Shadow Volume Algorithmus Z-Fail gezeichnet.

Falls Hybrid Shadow aktiviert ist, stehen folgende Optionen zur Verfügung:

- Keine Optimierung: Das Hybrid Shadow Verfahren wird zwar angewandt, es wird jedoch nicht weiter ausgenutzt, dass nun ein Volumen zur Verfügung steht, welches noch optimiert werden könnte. Es kommt kein weiterer Optimierungsschritt zur Anwendung.
- Depth Blurring: Nach der Berechnung des Schattenvolumens wird der Depth Blurring Algorithmus angewandt und danach der Schatten aus den modifizierten Daten heraus gezeichnet.
- Perfect Triangulation: Anstelle des Depth Blurring Algorithmus wird das Schattenvolumen in einem zusätzlichen Optimierungsschritt an allen Stellen richtig triangulliert, so dass möglichst wenig Ecken im resultierenden Schatten auftreten.

Als weitere aktivierbare Funktionen gibt es noch die „Extras“:

- Show Depth Map: Zeigt die Depth Map nach einem etwaigen Optimierungsalgorithmus an.
- Draw Volume: Zeichnet zusätzlich das Schattenvolumen und macht es somit sichtbar. Nicht möglich, falls als Schattentyp Shadow Map ausgewählt ist.
- Draw Scene: Zeichnet die Szene.
- Show Wireframe: Zeichnet nur das Drahtgittermodell der Szene und des Volumens, falls diese aktiviert sind.
- Vertical Sync: Vertical Sync beschränkt die Framerate auf die Frequenz des Monitors, zur Messung der Geschwindigkeit sollte es ausgeschaltet sein.

Mit dem Button „Reload Shaders“ werden alle verwendeten Shaderprogramme neu geladen, um mit ihnen experimentieren zu können, ohne das Programm neu starten zu müssen.

Auf die Implementierung von Shadow Mapping wird hier nicht näher eingegangen, da diese weder besonderen Neuigkeitswert aufweist, noch zur Forschungsarbeit dieser Studienarbeit gehört.

Die Beschreibung der Implementierung wird der Übersicht halber in drei Teile aufgeteilt: Hybrid Shadow ohne Optimierung, Hybrid Shadow mit Depth Blurring und Hybrid Shadow mit Perfect Triangulation. Vorerst wird nur auf die Implementierung ohne Vertex Texture Fetch eingegangen, die Beschreibung von VTF folgt im Anschluss.

5.3.1 Hybrid Shadow ohne Optimierung

Wie schon erwähnt, ist selbst bei der Anwendung von Hybrid Shadow ohne weiteren Optimierungsschritt eine qualitative Verbesserung des Schattens gegenüber Shadow Mapping zu erwarten und auch zu beobachten. Doch wie wird bei der Implementierung vorgegangen?

Als erstes ist es nötig, die Indices für das Zeichnen der Schattenvolumenpolygone zu berechnen und zu speichern, damit beim Zeichnen des Volumens in jedem Frame darauf zugegriffen werden kann. Die Indices weisen OpenGL an, in welcher Reihenfolge welche Vertices gezeichnet werden sollen und werden dazu nach ihrer Berechnung in ein Vertexbuffer-Objekt geschrieben, auf das sehr schnell zugegriffen werden kann, da dies ein Speicher direkt auf der Grafikkarte ist. Beim Hybrid Shadow Verfahren ohne Optimierung werden aus Performancegründen Triangle-Strips gerendert, was sich auf die Indices auswirkt. Nachfolgend der Pseudo-Code für die Berechnung der Indices ohne Optimierung (bei der Depth Blurring Optimierung wird genauso vorgegangen):

```
vector shadowVolumeIndex
for (int y = 0; y < shadow_map_height - 1; y++) {
   for (int x = 0; x < shadow_map_width; x++) {
      shadowVolumeIndex.add(x + (y + 1) * shadow_map_width)
      shadowVolumeIndex.add(x + y * shadow_map_width)
   }
   shadowVolumeIndex.add((GLuint)-1)
}
```

Am Ende jedes Triangle-Strips wird ein Index mit der größtmöglichen Integer-Zahl eingetragen. Dies wird dadurch bewirkt, dass -1 zu einem unsigned integer gecastet wird. Diese Zahl dient später zur Identifikation des Endes eines Strips und des Beginns eines neuen mit der OpenGL-Erweiterung „Primitive Restart“.

Zusätzlich zu den Indices ist ein Framebuffer-Objekt nötig, um Offscreen-Rendering zu ermöglichen, in diesem Fall das Rendern in eine Textur (die Shadow- oder auch Depth Map genannt). Das Ziel ist es, die Koordinaten der Szene in normalisierten Koordinaten der Lichtquelle in die Textur zu rendern und darauf später wieder zuzugreifen. Da mittels eines Pixelshaders die XYZ-Positionen der Vertices des Schattenvolumens festgelegt werden, ist eine Farbtextur nötig. Die XYZ-Positionen werden also auf die Rot-, Grün- und Blauwerte der Farbtextur gespeichert. Um einen Tiefentest durchführen zu können, wird zusätzlich ein Depth-Attachment benötigt. Das FBO bekommt also die beiden Attachments: Farb- und Tiefentextur wie in Abbildung 15 dargestellt.

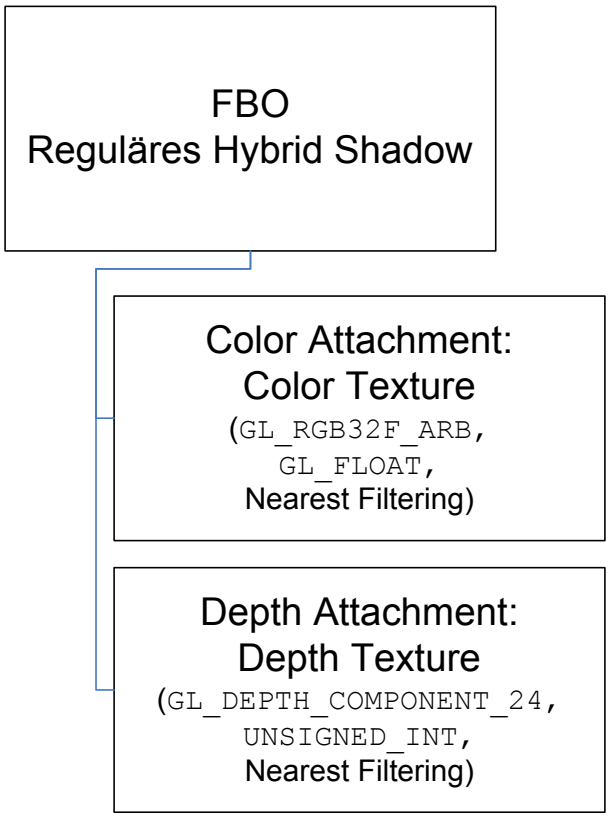

Abbildung 15: Framebuffer-Objekt für Hybrid Shadow ohne Optimierung

Nachdem diese Vorbereitungen (Indices und FBO) einmalig zum Programmstart durchgeführt wurden, beginnt das Zeichnen eines Frames. Nun wird das Framebuffer-Objekt gebunden, wodurch OpenGL angewiesen wird, in das FBO zu rendern, statt in den Framebuffer. Es beginnt der erste Rendering-Pass. Ähnlich wie beim Shadow Mapping Verfahren wird die Szene von der Lichtquelle aus betrachtet in das gebundene FBO gerendert. Gezeichnet werden nur die Back-Faces, da es ausreicht, wenn das Schattenvolumen von den Backfaces der Objekte zur Far-Plane reicht. Um zum gewünschten Ergebnis zu kommen wird aber vor dem eigentlichen Rendering der Szene in die Depth Map ein Quad über die gesamte Fläche gezeichnet und mit einem Fragmentshader erreicht, dass die RG-Werte jedes Pixels die XY-Werte relativ zur Position des Pixels im Wertebereich [-1; 1] sind und der Blau-Wert auf 1 gesetzt wird, was einer Tiefe von unendlich entspricht. Dies ist wichtig, da die Szene möglicherweise nicht das gesamte Bild ausfüllt und in diesem Fall ist das Schattenvolumen an den Stellen, an denen kein Objekt der Szene zu sehen ist, unendlich weit entfernt – also auf der Far-Plane. Darauf folgend werden mit dem aktivierten „Shader für Hybrid Points“ nun also die Objekte der Szene in die Depth Map gezeichnet.

Nachfolgend der Pseudo-Code des Shaders für Hybrid Points:

```
void main() {
  // no change at the border of the depth texture
  if pixel_is_at_border_of_frame then discard pixel
```

```
   fragment.r = (2 * fragment.x) / (width - 1) - 1
   fragment.g = (2 * fragment.y) / (height - 1) - 1
   fragment.b = 2 * Fragment.z - 1
}
```

Nun ist zu beachten, dass der Rand des Schattenvolumens unangetastet in der Unendlichkeit bleibt. Das sichert die Abgeschlossenheit des Volumens, ohne welches es Fehler in den Schatten geben kann, falls sich z.B. ein Objekt nur teilweise im View-Frustum der Lichtquelle befindet. Falls das aktuelle Fragment nicht am Rand des Bildes ist, wird dessen Position mittels diesen Shaders auf die Farbe des Fragments und den Wertebereich [-1; 1] abgebildet. Dies ist möglich, da eine Floating-Point Textur im FBO eingehängt ist.

Nach dem ersten Rendering-Pass muss nun das Kopieren der Daten von der Tiefentextur in das Vertexbuffer-Objekt vorbereitet werden. Dies geschieht durch das Binden des Buffers als Pixelpack-Buffer, danach kann vom Color-Attachment in das VBO kopiert werden. Nun wird der Buffer als Array-Buffer gebunden und das VBO mit den Indices als Elementarray-Buffer. Der Vertex-Pointer wird auf den Anfang des Vertexbuffers gesetzt, in dem nun die Vertexdaten stehen:

```
glBindBuffer(GL_PIXEL_PACK_BUFFER_EXT, ShadowVolumeDataPointer);
glReadPixels(0, 0, width, height, GL_RGB, GL_FLOAT, BUFFER_OFFSET(0));
glBindBuffer(GL_ARRAY_BUFFER_ARB, ShadowVolumeDataPointer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER_ARB, ShadowVolumeIndexPointer);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
```

Nachdem sowohl der Buffer mit den Positionen der Schattenvolumenvertices gebunden ist, als auch der mit den benötigten Indices, kann nun der zweite Rendering-Pass beginnen. Das Framebuffer-Objekt wird deaktiviert, somit also der Framebuffer aktiviert. Zuerst wird die Szene nur mit ambientem und aussendendem Licht von der Kamera aus gezeichnet, wodurch man praktischerweise auch gleich die Tiefe der Objekte erhält. Danach kommt der schon beschriebene Z-Fail Algorithmus mit dem doppelseitigen Stencilpuffer zur Anwendung. Dazu wird OpenGL vor dem Zeichnen des Schattenvolumens angewiesen, bei einem fehlschlagenden Tiefentest der Backfaces den Stencilpuffer zu erhöhen und bei einem fehlschlagenden Tiefentest der Frontfaces diesen zu erniedrigen. Nun kann das Volumen gezeichnet werden.

Da das Volumen im Licht-Space vorliegt, muss zuerst eine Transformation in den Kamera-Space erfolgen. Dies geschieht, indem die Projektionsmatrix der Lichtquelle mit der Modelviewmatrix des Lichts multipliziert und anschließend invertiert wird. Das Ergebnis wird mit der aktuellen Modelview-Matrix multipliziert.

```
lightToViewMatrix = Invert(lightProjectionMatrix * lightViewMatrix)
glMultMatrixf(lightToViewMatrix)
```

Nun ist darauf zu achten, dass das Schattenvolumen abgeschlossen ist. Im Besonderen betrifft dies das sog. Far-Cap, d.h. es muss eine der Lichtquelle abgewandte Fläche auf der Far-Plane das Volumen abschließen. Momentan würde das Volumen aussehen wie in Abbildung 16 skizziert. Wo – von der Lichtquelle aus betrachtet – keine

Objekte vorhanden sind, befinden sich die Polygone des Schattenvolumens in der Unendlichkeit (Far-Plane), obwohl dort eigentlich gar keine gewollt sind, da dies fehlerhaften Schatten produziert.

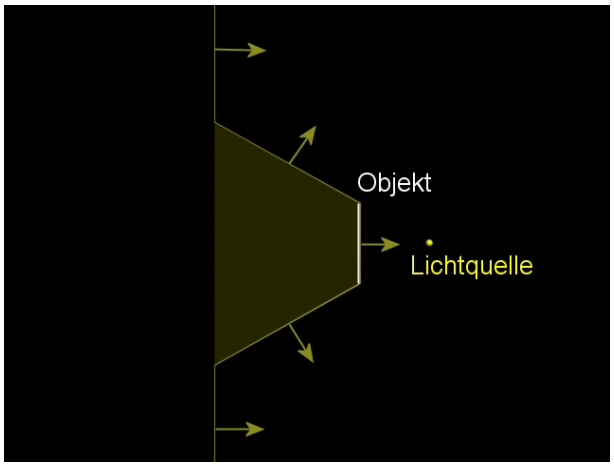

Abbildung 16: Falsches Schattenvolumen ohne Far-Caps

Erwünscht ist ein Schattenvolumen, welches an dessen lichtabgewandten Seite eine Fläche aufweist, deren Normalen relativ zum Volumen nach außen zeigen, wie in Abbildung 17 dargestellt. Falls kein weiteres Objekt vorhanden ist, soll es sonst auch keine weiteren Polygone des Schattenvolumens mehr auf der Far-Plane geben. Die Polygone des Volumens, die sich in Abbildung 16 auf der Far-Plane befinden, sollten also eliminiert und stattdessen ein korrektes Far-Cap gezeichnet werden. Nur so kann Z-Fail richtig funktionieren.

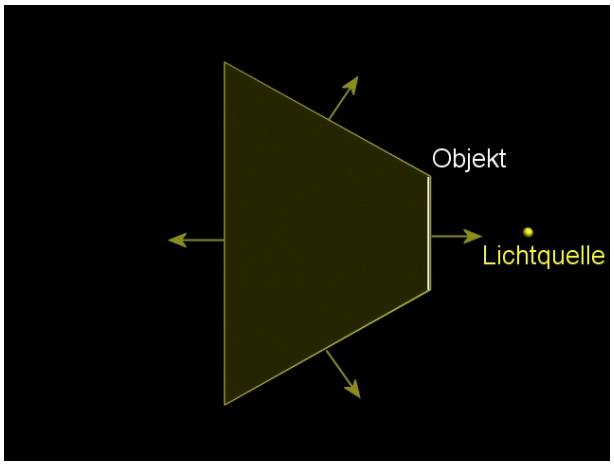

Abbildung 17: Richtiges Schattenvolumen mit Far-Caps

Da es bei Hybrid Shadows ein einziges großes Volumen gibt, das über alle Objekte gelegt wird und im Gegensatz zu normalen Shadow Volumes keinerlei Informationen über die Szene – und somit auch nicht über die Schattenvolumenpolygone – vorliegen,

ist dies allerdings nicht möglich. Zwar könnte man die fehlerhaften Schattenvolumenpolygone auf der Far-Plane vermeiden, indem man sie z.B. wegclippt, jedoch wäre es nicht effizient möglich, das korrekte Far-Cap des Volumens zu berechnen. Um dennoch ein Far-Cap zu erhalten, welches einen korrekten Schatten liefert, wird beim Rendern des Schattenvolumens zusätzlich ein dem Licht abgewandtes Quad auf die Far-Plane gezeichnet, wie in Abbildung 18 dargestellt. Dies hebt den Effekt der „falschen“ Polygone auf und erzeugt genau an den gewünschten Stellen die Abgeschlossenheit des Schattenvolumens.

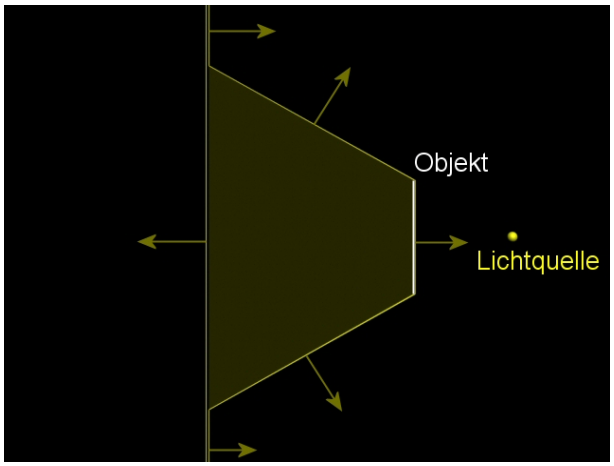

Abbildung 18: Lösung des Far-Cap Problems

Konkret bedeutet dies, dass vor dem Zeichnen des eigentlichen Volumens das in Abbildung 18 skizzierte Quad auf die Far-Plane mittels der Nutzung der OpenGL-Extension `GL_DEPTH_CLAMP` über die gesamte Fläche gezeichnet wird. Dabei muss auch hier wieder beachtet werden, dass der Rand des Volumens – von der Lichtquelle aus betrachtet – in seiner Wirkung nicht aufgehoben wird, da dies sonst einen eigentlich nicht existierenden Schatten am Rand des View Frustums der Lichtquelle erzeugen würde.

Nun kommt das eigentliche Zeichnen des Volumens, welches relativ unspektakulär ist. Zuvor wird das Rendern in den Farb- und Tiefenpuffer deaktiviert. Das eigentliche Zeichnen erfolgt anschließend durch `glDrawElements(…)`, wobei als Indices der Anfang des aktiven Vertexbuffer-Index-Objekts (Element-Array) angegeben wird. Der Datapointer wurde ja schon vorher entsprechend gesetzt.

Nachdem nun der Stencilpuffer die richtigen Werte enthält, kommt der letzte Rendering-Pass zur Anwendung. An allen Stellen, an denen der Stencilwert Null ist, wird die Szene mit vollem Licht gezeichnet, da sich genau dort kein Schatten befindet.

Das Ergebnis der eben beschriebenen Vorgehensweise ist, wie in Kapitel 4.3 beschrieben, ein Schatten wie man ihn beim Shadow Mapping erhält mit dem Unterschied, dass hier einige Stellen (durchschnittlich 25%) durch Zufall richtig trianguliert

sind und dadurch schöner aussehen. Die scharfen Ecken an diesen Stellen verschwinden, wie schon in Abbildung 5 gezeigt wurde.

5.3.2 Hybrid Shadow mit Depth Blurring

Der große Vorteil von Hybrid Shadow ist die Tatsache, dass man ein Volumen erhält, welches zu Gunsten der Schattenqualität in einem Optimierungsschritt verändert werden kann. Dies wurde bisher nicht ausgenutzt. Eine mögliche Optimierungsmethode ist Depth Blurring. Sie mischt die Tiefe eines jeden Pixels in der Depth Map mit denen der jeweiligen Nachbarpixel. Als Folge sind die von der Shadow Mapping Methode bekannten Aliasing-Artefakte im Schatten nahezu ausgeschlossen, wobei dies natürlich trotzdem immer noch von der Auflösung der Depth Map abhängt.

Zu Beginn kommt das Framebuffer-Objekt von Depth Blurring ohne ein Color-Attachment aus, da im Gegensatz zu Hybrid Shadow ohne Optimierung noch ein Optimierungsschritt folgen wird, bei dem die XYZ-Positionen der Vertices für die Schattenvolumenpolygone in die – dann einzuhängende – Farbtextur geschrieben werden. Als Eingabe für den Optimierungsschritt sind jedoch nur die Tiefenwerte, von der Lichtquelle aus betrachtet, relevant. Für das Depth-Attachment wird hier eine Tiefentextur genommen, die linear gefiltert ist. Das hat den Vorteil, dass im Optimierungsschritt zum Mischen der Tiefen nicht unbedingt nur ganzzahlige Pixelentfernungen vom aktuellen Pixel genommen werden können, sondern z.B. auch die Tiefenwerte der Pixel gemischt werden können, die 1.5 Pixel weit vom aktuellen Pixel entfernt sind. Dies lieferte nach einigen Tests auch in der Tat die besseren Ergebnisse als einfach nur die Tiefenwerte zu mischen, die einen Pixel entfernt sind. Somit erhält das FBO von Depth Blurring für den ersten Rendering Pass also nur ein Depth-Attachment wie in Abbildung 19 ersichtlich.

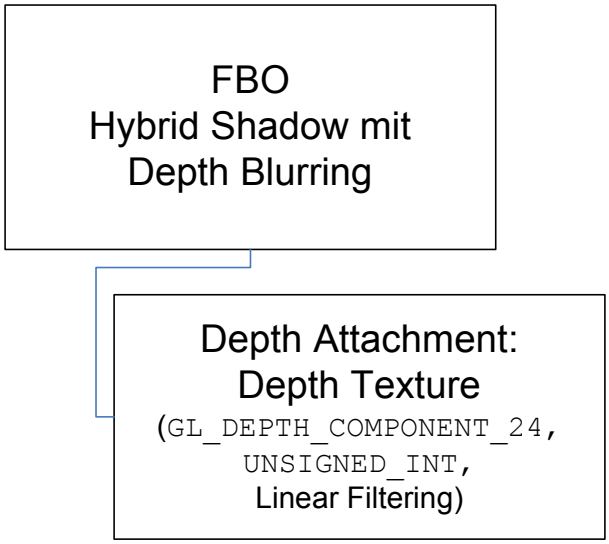

Abbildung 19: Framebuffer-Objekt für Hybrid Shadowing mit Depth Blurring

Der erste Rendering-Pass läuft mit Depth Blurring genauso ab, wie der ohne Optimierungen, wobei hier allerdings nicht die Backfaces sondern die Frontfaces gerendert werden und somit als Schattenvolumenberechnung fungieren. Dies hat den Grund, dass bei Depth Blurring die Schatten gerne etwas später „anfangen“, d.h. sie sind etwas weiter von der Lichtquelle entfernt, als sie sein sollten. Verwendet man nun die Frontfaces zur Berechnung, wird dieses Problem etwas abgeschwächt, da nun der Schatten ohne Optimierung bereits an den Frontfaces beginnen würde und nicht erst an den Backfaces. Außerdem wird, im Gegensatz zum Verfahren ohne Optimierungen, weder ein Quad in die Unendlichkeit gezeichnet, noch der „Shader für Hybrid Points“ verwendet. Als Eingabe für den Optimierungsschritt sind nur die Tiefenwerte der Frontfaces nötig.

Für den Optimierungsschritt wird die Depth Map vom Framebuffer-Objekt ausgehängt und ein Color-Attachment eingehängt. Dann wird die Depth Map unter Benutzung des Fragmentshaders zum Depth Blurring auf die gesamte Fläche gezeichnet. Im Prinzip erhält im Shader der Blauwert (welcher die Tiefe repräsentiert) des aktuellen Pixels den Durchschnitt der Tiefen seiner Nachbarpixel in der Depth Map und ihm selbst, wie aus dem nachfolgenden Pseudo-Code des Shaders für Depth Blurring ersichtlich, wobei als „Nachbar“, wie schon erwähnt, die eineinhalbfache Pixelentfernung sehr gute Erbgebnisse liefert:

```
Float function depthOfAPixel () {
   float n1 = getDepthOfNeighbor(left)
   float n2 = getDepthOfNeighbor(right)
   float n3 = getDepthOfNeighbor(top)
   float n4 = getDepthOfNeighbor(bottom)
   float n5 = getDepthOfNeighbor(topleft)
   float n6 = getDepthOfNeighbor(topright)
   float n7 = getDepthOfNeighbor(bottomleft)
   float n8 = getDepthOfNeighbor(bottomright)
   float currentDepth = getDepthOfCurrentPixel()
   return (n1 + n2 + n3 + n4 + n5 + n6 + n7 + n8 + currentDepth) / 9.0
}
```

Die Werte Rot und Grün erhalten, wie schon bei Hybrid Shadow ohne Optimierungen, die X- und Y-Position des Pixels in normalisierten Koordinaten von der Lichtquelle aus.

Der Rest verläuft genauso wie bei der Version ohne Optimierungen. Nachdem die Positionsdaten in das VBO kopiert wurden, wird der Data-Buffer und der Index-Buffer gebunden, sowie der Vertex-Pointer auf den Beginn des Data-Buffers gesetzt. Nachdem das FBO deaktiviert wurde, wird der Z-Fail Algorithmus in Gang gesetzt.

Die Ergebnisse dieses Verfahrens sind bereits in Abbildung 13 dargestellt. Selbst mit einer Depth Map Auflösung, die bei Shadow Mapping erhebliche Pixel-Artefakte produziert, sind bei Hybrid Shadow mit Depth Blurring kaum Treppeneffekte zu erkennen. Zwar ist dies abhängig von der Szene und der Konstellation von Licht und Objekt, in den meisten Fällen sind jedoch erhebliche Verbesserungen in der Qualität des Schattens zu erkennen.

Ein Nachteil besteht allerdings darin, dass der Schatten von Depth Blurring, wie in Abbildung 20 dargestellt, u. U. „zu früh“ anfängt.

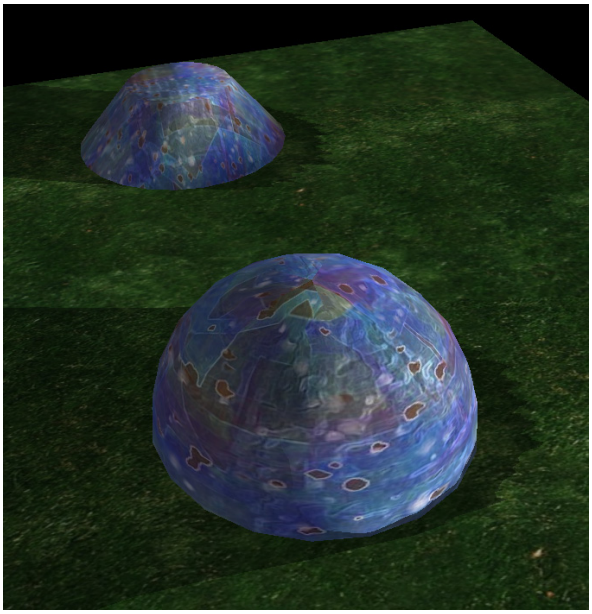

Abbildung 20: Problem von Depth Blurring: Schatten fängt zu früh an (Resultat)

Dies geschieht deshalb, weil der Tiefenwert in der Depth Map von einem Polygon (im obigen Beispiel eines der Halbkugel) mit dem Tiefenwert eines anderen Polygons (im Beispiel eines des Bodens) vermischt wird. Das Resultat ist ein Tiefenwert, der zu klein ist und den Schatten somit fälschlicherweise näher an die Lichtquelle zieht. Das Problem wird umso stärker je größer die Differenz der Tiefenwerte zweier Nachbarpixel in der Depth Map ist. Im obigen Beispiel „sieht“ die Lichtquelle den Boden aus einem extrem flachen Winkel, weswegen sich der korrespondierende Tiefenwert sehr von dem seines Nachbarpixels (welcher aus der Halbkugel resultiert) unterscheidet. Abbildung 21 zeigt dies aus schematischer Sicht. Es wurde zwar versucht, den optimierten / geblurrten Tiefenwert nur zu setzen, falls er weiter von der Lichtquelle entfernt ist als der ursprüngliche, dadurch verschlechtert sich die Qualität des resultierenden Schattens aber erheblich.

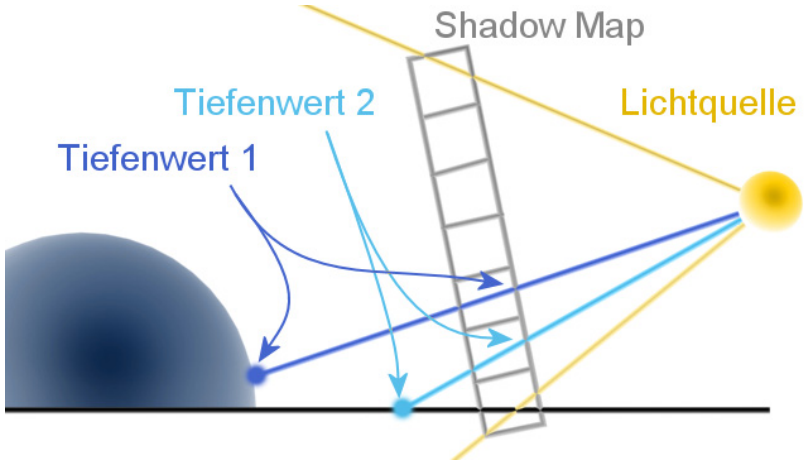

Abbildung 21: Problem von Depth Blurring: Schatten fängt zu früh an (Skizze)

Das Problem lässt sich vermindern, wenn im Depth Blurring Shader ein Offset zum berechneten Tiefenwert hinzugezählt wird. Dadurch kann es allerdings vorkommen, dass der Schatten von der Lichtquelle aus betrachtet zu spät anfängt, was das Problem also auch nicht generell löst. Die Alternative ist, die Auflösung der Depth Map zu erhöhen, ganz vermeidbar ist der Fehler allerdings nie.

Ein weiteres Phänomen – ähnlichen Ursprungs – betrifft hauptsächlich dünne oder kleine Objekte. Abbildung 22 zeigt einen Baum mit sehr dünnem Stamm und sein Schattenvolumen. Da der Stamm in der relativ niedrig aufgelösten Depth Map höchstens ein paar Pixel in der Breite misst und diese mit ihren Nachbarpixeln (deren Tiefenwerte in diesem Fall um einiges größer sind) vermischt werden, kommt es leicht dazu, dass das Ergebnis ein Volumen ist, welches zu spät beginnt – von der Lichtquelle aus betrachtet. Dies hat im Bild keinen Effekt und auch sonst gibt es kaum denkbare Fälle, in denen dieses Phänomen Einfluss auf den resultierenden Schatten nimmt. Denn wäre ein Objekt im dargestellten Bild im Bereich zwischen dem Beginn des Volumens und dem Baumstamm, würden die Schattenvolumenpolygone, sofern das schattenempfangende Objekt größer ist als das schattenwerfende, ohnehin früher anfangen – um genau zu sein, immer mindestens 1/9 der Strecke vom schattenempfangenden zum schattenwerfendem Objekt vor dem schattenempfangenden Objekt (immer von der Lichtquelle aus betrachtet). In dieser Konstellation wird der Punkt eines Schattenvolumenpolygons nämlich mindestens zu 1/9 von der Tiefe des schattenwerfenden Objekts bestimmt und maximal 8/9 von der des schattenempfangenden. Ausser, der Extremfall tritt ein und auch das schattenempfangende Objekt ist klein genug, dass weniger als 8/9 dessen Tiefe mit in die Blurring-Formel eingeht und der Rest von einem noch weiter entfernten Objekt. Da aber der in Abbildung 22 skizzierte Fall ohnehin schon voraussetzt, dass das schattenwerfende Objekt klein ist, ist es extrem unwahrscheinlich, dass sich ein schattenempfangendes noch kleineres Objekt genau an der „falschen“ Stelle befindet und dies trotzdem so groß ist, dass der Betrachter einen fehlenden Schatten auf diesem Objekt überhaupt bemerken würde. Sollte dies den-

noch passieren und nicht akzeptabel sein, ist die einzige Lösung dieses Problems eine Erhöhung der Auflösung der Depth Map.

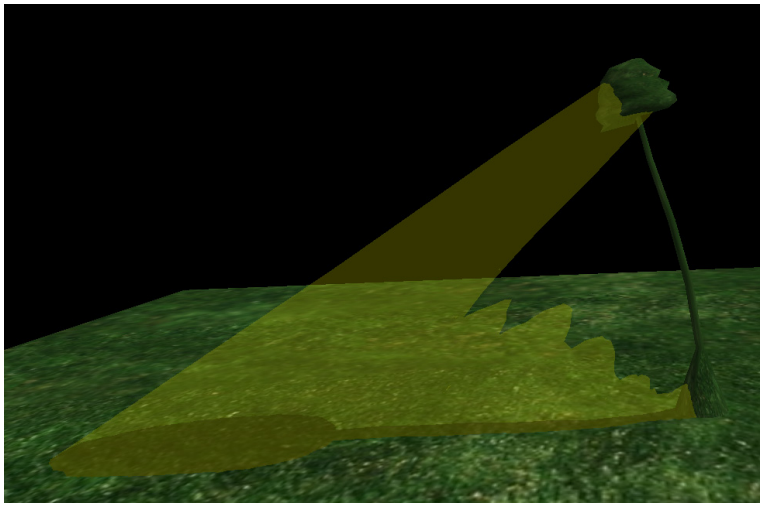

Abbildung 22: Problem von Depth Blurring: Schatten fängt zu spät an

5.3.3 Hybrid Shadow mit Perfect Triangulation

Perfect Triangulation bezeichnet ein weiteres Optimierungsverfahren, welches im Rahmen dieser Arbeit für Hybrid Shadows entwickelt wurde. Die Idee ist, die Topologie der Triangulierung der Schattenvolumenpolygone zu verändern. Durch die daraus resultierende neue Anordnung der Polygone verschwinden einige scharfe Ecken bzw. werden weicher. Da dies aus Performancegründen natürlich auf der GPU stattfinden muss und eine Topologieänderung auf der Grafikkarte von Haus aus nicht möglich ist, musste eine Möglichkeit gefunden werden, dies dennoch zu realisieren. Der Trick ist, die Textur mit den Positionen der Schattenvolumenpolygone auf eine nahezu viermal so große Textur zu rendern, so dass jeder Vertex – die am Rand ausgenommen – viermal vorhanden ist. Da nun kein Quad (Viereck) mehr seine Vertices teilen muss, reicht es aus, die Vertices eines Quads um 90° zu „rotieren“, also die Position eines Vertex jeweils den im Uhrzeigersinn nächsten Vertex weiterzugeben, um eine andere Triangulierung dieses Quads zu erreichen. Alle anderen Quads bleiben dadurch unberührt.

Um dies zu erreichen, wird der erste Rendering-Pass genauso durchgeführt wie bei Hybrid Shadow ohne Optimierungen. Auch das Framebuffer-Objekt wird genauso aufgebaut wie dort, wobei hier – wie schon beim Depth Blurring – kein Color-Attachment nötig ist, da der Optimierungsschritt vorerst nur die Tiefe von der Lichtquelle aus als Eingabe benötigt. Allerdings sind beim Optimierungsschritt einige Modifikationen am FBO nötig. Denn die Änderung der Triangulierung bestimmter Quads ist nicht möglich, wenn das Schattenvolumen mit Triangle-Strips gerendert wird. Da sich die Reihenfolge der Vertices beim Zeichnen ändern können soll, darf kein Vertex mehr zu mehreren Quads gehören. Somit muss aus der berechneten Depth Map eine Textur berechnet

werden, die fast doppelt so breit und doppelt so hoch wie die Depth Map ist und fast jeden Pixel der Depth Map zweimal enthält – siehe Abbildung 23. Um genau zu sein, muss jeder Pixel doppelt bzw. vierfach vorhanden sein bis auf den ersten und den letzten jeder Zeile und Spalte. Die neue Textur muss also folgende Bedingung erfüllen:

```
colorTexture_perfectTriangulation.width = 2 * depth_map.width - 2
colorTexture_perfectTriangulation.height = 2 * depth_map.height - 2
```

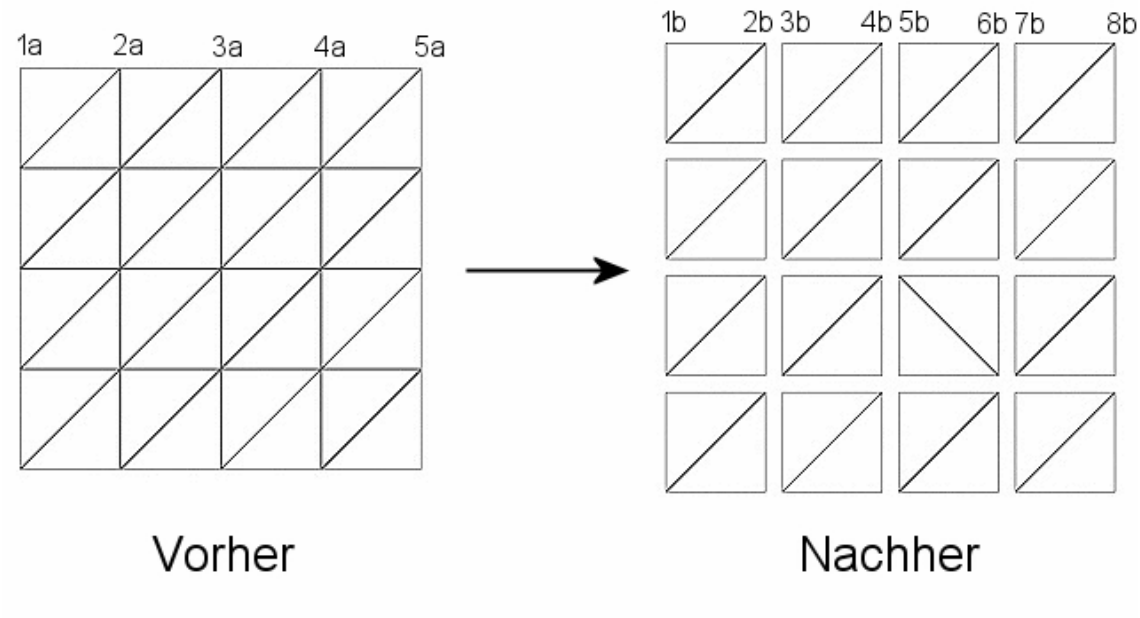

Abbildung 23: Die Speicherung der Vertices für das Schattenvolumen der Perfect Triangulation Optimierung

In Abbildung 23 wird z.B. Vertex 1a auf Vertex 1b abgebildet, Vertex 2a auf Vertex 2b und 3b, Vertex 3a auf Vertex 4b und 5b, usw.

In der Implementierung passen somit die bisher beim Verfahren ohne Optimierung und dem mit Shadow Blurring verwendeten Indices nicht mehr. Die Indices müssen entsprechend so gesetzt sein, dass sie das Rendern von einzelnen Vierecken ermöglichen, also jeweils immer zwei Triangles. Folgend der Pseudo-Code für die Berechnung der Indices für Hybrid Shadow mit Perfect Triangulation:

```
for (int y = 0; y < shadow_map_height * 2 - 2; y = y + 2) {
  for (int x = 0; x < shadow_map_width * 2 - 2; x = x + 2) {

    // first triangle in quad
    shadowVolumeIndex.add(x + (y + 1) * (shadow_map_width * 2 - 2))
    shadowVolumeIndex.add(x + y * (shadow_map_width * 2 - 2))
    shadowVolumeIndex.add(x + 1 + y * (shadow_map_width * 2 - 2))

    // second triangle in quad
    shadowVolumeIndex.add(x + 1 + y * (shadow_map_width * 2 - 2))
    shadowVolumeIndex.add(x + 1 + (y + 1) * (shadow_map_width * 2-2))
    shadowVolumeIndex.add(x + (y + 1) * (shadow_map_width * 2 - 2))
  }
}
```

Bei Hybrid Shadow mit Perfect Triangulation ist kein Primitive-Restart nötig, da hier die Primitiven Triangles sind, keine Strips, somit ist auch kein spezieller Index nötig, der signalisiert, dass die Primitive aufhört und eine neue beginnt.

Des Weiteren muss vor der Anwendung der Optimierung – also nachdem in die Depth-Texture gerendert wurde – im FBO das zuvor verwendete Depth-Attachment ausgehängt und ein Color-Attachment eingehängt werden, welches die benötigte Größe hat.

Der Rest der Optimierung erfolgt im Optimierungsschritt, in welchem die alte Textur (Depth Map) mit dem aktivierten Fragmentshader für Perfect Triangulation auf die gesamte Fläche gezeichnet wird. Im Fragmentshader werden zunächst alle vier Vertices im aktuellen Quad errechnet. Im folgenden Code des Shaders erhält `oldCoordinates` die aktuelle Texturkoordinate der Depth Map. `curVPos` erhält Informationen darüber, wie der aktuelle Vertex (repräsentiert durch den aktuellen Pixel im Shader) im Quad liegt. Mit diesen Informationen können die Tiefenwerte der vier Vertices im Quad in der Depth Map nachgeschlagen werden.

```
// texture coordinates of the old (smaller) texture
vec2 oldCoordinates = vec2(ivec2((gl_FragCoord + 1.0) / 2.0));

// 4 cases for curVPos:
// 0,0: topLeft; 1,0: topRight; 0,1: bottomLeft; 1,1: bottomRight
vec2 curVPos;
curVPos = vec2(ivec2(gl_FragCoord.xy) % 2);

// computing the 4 Vertices in Quad
vec3 TL, TR, BL, BR;

vec2 curXY = (oldCoordinates - curVPos) / oldSize;
float curZ = texture2D(tex0, curXY).x;
TL = vec3(curXY, curZ);

curXY = (oldCoordinates - curVPos + vec2(1.0, 0.0)) / oldSize;
curZ = texture2D(tex0, curXY).x;
TR = vec3(curXY, curZ);
```

```
curXY = (oldCoordinates - curVPos + vec2(0.0, 1.0)) / oldSize;
curZ = texture2D(tex0, curXY).x;
BL = vec3(curXY, curZ);

curXY = (oldCoordinates - curVPos + vec2(1.0, 1.0)) / oldSize;
curZ = texture2D(tex0, curXY).x;
BR = vec3(curXY, curZ);
```

Das Ergebnis sind vier Vertices mit jeweils drei Koordinaten, x, y und z (die Tiefe): `TL`, `TR`, `BL` und `BR`.

Wie schon erläutert, ist das Kriterium, nach welchem die Triangulierung umgedreht werden muss, die Entfernung der vier Punkte im Quad zu der Ebene, die von den jeweils restlichen drei Punkten aufgespannt wird. Im Shader werden also alle vier Entfernungen berechnet und miteinander verglichen. Dies geschieht mit folgender Pseudo-Code-Funktion, die die Entfernung des Punktes P von der Ebene, die von den Punkten A, B und C aufgespannt wird, zurückgibt:

```
float computeDistance(vec3 P, vec3 A, vec3 B, vec3 C) {
  vec3 normal = cross(vectorFrom(A, B), vectorFrom(A, C))
  return abs(dot(vectorFrom(A, P), normalize(normal)))
}
```

Falls die Entfernung des rechten oberen Vertex im Quad größer ist als die des linken oberen und des rechten unteren oder falls die Entfernung des linken unteren größer ist als die des rechten unteren und des linken oberen, wird der Quad um 90 Grad gedreht, was die gewünschte Änderung der Topologie zur Folge hat.

```
if (((distanceTR > distanceTL) && (distanceTR > distanceBR)) ||
  ((distanceBL > distanceBR) && (distanceBL > distanceTL))) {

  oldCoordinates.x = oldCoordinates.x + float(int(!(bool(curVPos.x) ||
    bool(curVPos.y))) - int(bool(curVPos.x) && bool(curVPos.y)));

  oldCoordinates.y = oldCoordinates.y + float(int(curVPos.x) -
    int(curVPos.y));
}
```

In GLSL wird `true` zu 1 und `false` zu 0 gecastet und umgekehrt. Wie schon erwähnt, bezeichnet `curVPos` die Lage des aktuellen Vertex – repräsentiert durch den aktuellen Pixel im Shader – im Quad, wobei der x-Wert „0“ „links“ und „1“ „rechts“, sowie der y-Wert „0“ „oben“ und „1“ „unten“ bedeuten. Der obige Code wird ausgeführt, falls das Quad um 90° gedreht werden soll und arbeitet wie folgt: Falls durch den aktuellen Pixel der linke obere Vertex im Quad repräsentiert wird, wird zu dessen X-Koordinate eins hinzugezählt. Falls der rechte untere Vertex mit dem aktuellen Pixel verkörpert wird, wird von der X-Koordinate eins abgezogen. Der Y-Koordinate wird 1 hinzugezählt, falls sich das aktuelle Element rechts im Quad befindet und 1 abgezogen falls es unten im Quad ist. Diese beiden Zeilen erwirken also genau die erwünschte Rotation um 90°.

Nach dem Optimierungsschritt sind nun in der nahezu viermal so großen Farbtextur also die X-, Y- und Z-Werte der Quads für die Schattenvolumenpolygone in normali-

sierten Koordinaten der Lichtquelle gespeichert. Die Topologie ist optimiert, so dass im Schatten möglichst wenig Treppeneffekte auftreten werden.

Bis auf die Tatsache, dass nun beim weiteren Verlauf eine andere Textur als bei der Version ohne Optimierungen als Punktequelle verwendet wird und diese größer ist als die ursprüngliche Depth Map – was natürlich beim Kopieren der Werte mit `glReadPixels(…)` und bei mit der Verwendung des neuen Indexarrays für Perfect Triangulation beachtet werden muss, ist der Rest der Implementierung von Perfect Triangulation mit der bisher beschriebenen ohne Optimierung vergleichbar. Beim Rendern des Volumens muss natürlich mit `GL_TRIANGLES` statt mit `GL_TRIANGLE_STRIP` gerendert werden.

Der Vorteil der Perfect Triangulation Optimierung besteht darin, dass die Qualität des Schattens verbessert wird, ohne die Position der Vertices der Schattenvolumenpolygone ändern zu müssen. Geändert wird lediglich die Topologie der Vertices. Es ist also nicht möglich, dass durch eine zu geringe Auflösung der Depth Map der Schatten zu nahe an oder zu weit weg von der Lichtquelle ist, wie es bei Depth Blurring vorkommen kann. Der Schatten bleibt genauso akkurat wie der des normalen Shadow Mapping-Verfahrens. Er sieht nur besser aus.

Ein Nachteil dieser Optimierung ist allerdings, dass die Pixelartefakte, die durch die Depth Map hervorgerufen werden, im resultierenden Schatten bei weitem nicht so gut entfernt sind, wie es bei der Depth Blurring Methode der Fall ist. Des Weiteren benötigt Perfect Triangulation mehr Rechenleistung, da auf eine nahezu vierfach größere Textur gerendert werden muss und dementsprechend das Schattenvolumen nahezu viermal so viele Vertices besitzt wie ohne Verwendung dieser Optimierung.

5.3.4 Verwendung von Vertex Texture Fetch

Die Verwendung des Pixelbuffer-Objekts, um den Inhalt der Textur in das Vertexbuffer-Objekt zu kopieren, ist auf AGP-Hardware relativ langsam, da über den Bus kopiert wird. Bei PCI-Express funktioniert dieser Vorgang schneller, bei AGP gibt es jedoch erhebliche Effizienzeinbußen. Als Lösung dieses Problems wurde versucht, das Shader Modell 3.0, im Besonderen den damit möglichen Texture-Lookup in einem Vertexshader, auszunutzen. Für das Zeichnen des Schattenvolumens wird ein Vertexprogramm aktiviert, welches für den aktuell zu rendernden Vertex die Position aus der Textur liest und setzt. Dieses Verfahren arbeitet direkt auf der GPU, weshalb die Hoffnung bestand (welche auch bestätigt wurde), dass es einige Geschwindigkeitsvorteile bietet.

Zu beachten ist dabei, dass die wenigsten Texturformate von OpenGL für Vertex Texture Fetch hardwarebeschleunigt sind. Laut dem GPU Programming Guide 2.4.0 von NVIDIA [NVI2.4.0] gibt es nur zwei hardwarebeschleunigte VTF-Texturformate: `R32F` und `RGBA32F`. `RGB32F` wird laut dem Dokument „NVIDIA OpenGL Texture Formats“ [NVITex] intern als `RGBA32F` gespeichert und ist somit auch hardwarebeschleunigt,

was der Grund dafür ist, dass dieses Format in der Hybrid Shadow Implementierung verwendet wurde.

Wie schon beschrieben ersetzt der Vertex Texture Fetch in der Implementierung das Kopieren der Texturdaten mit `glReadPixels(…)`, d.h., dass dieser Abschnitt im Falle der Benutzung von VTF übergangen werden muss. Dafür gibt es einen Zusatz von Code beim Zeichnen des Volumens: Es müssen sowohl der Vertexshader aktiviert und dessen Parameter übergeben, als auch die Textur gebunden werden, die die Positionsdaten für die Vertices des Schattenvolumens enthält. Gerendert wird eine ebene Fläche mit so vielen Vertices wie die Depth Map Pixel enthält bzw. die optimierte größere Textur bei Perfect Triangulation. Die initialen Positionen der Vertices in dieser Fläche müssen in der Vorbereitung auf das Rendern, also zu Beginn der Applikation, berechnet und im Vertexbuffer-Objekt gespeichert werden. Hierzu wird einfach dasselbe VBO verwendet, welches auch bei der VTF-Alternative (das Kopieren über das PBO mit `glReadPixels(…)`) zum Einsatz kommt. Falls kein VTF benutzt wird, sind die initialen Positionen für VTF ohne Belang, da dann für jeden Frame die Positionen komplett von der Depth Map kopiert werden. Im Falle der Anwendung von Vertex Texture Fetch braucht der Vertexshader, der die Position für den jeweiligen Vertex aus der Textur holt, allerdings einen Anhaltspunkt, um zu wissen, welchen Pixel er aus der Depth Map nehmen soll. Dies geschieht durch lineare Texturkoordinaten. Damit diese richtig berechnet werden, muss das Schattenvolumen vor diesem Vorgang schon eine bestimmte Form haben. In der Vorbereitung auf VTF werden also die Vertex-Positionen des Schattenvolumens so gesetzt, dass es die Form eines ebenen Quadrates aufweist und die Vertices regelmäßig (schachbrettartig) darin aufgeteilt sind. Es werden sowohl ein Vertexbuffer-Objekt für Perfect Triangulation als auch eines für Depth Blurring bzw. ohne Optimierung gebraucht, da wie beschrieben, Perfect Triangulation mehr Vertices beansprucht. Falls VTF im Betrieb aus- und wieder angeschaltet wird, hat das entsprechende VBO im Verlauf der Berechnung eines Frames ohne VTF seine initialen Daten verloren, da diese mit den kopierten Daten aus der Depth Map ersetzt wurden. Folglich müssen bei der Aktivierung von VTF die beiden VBOs neu auf den Initialzustand gesetzt werden.

Weil die Vertices im VBO gleichmäßig schachbrettartig verteilt sind, kann durch die Position eines jeden Vertex die entsprechende Texturkoordinate des Pixels in der Depth Map bzw. in der optimierten Textur, die seine korrekte Position gespeichert hat, berechnet werden. Durch einen Texture-Lookup im Vertexshader wird diese Position dann für jeden Vertex nachgeschlagen und entsprechend gesetzt. Nachfolgend der Code für den Vertex Texture Fetch im Vertexshader:

```
void main(void) {
  // calculation of the texture coordinates
  // gl_Vertex.x is in [0; width - 1]
  // gl_Vertex.y is in [0; height - 1]
  vec2 texCoord = vec2(gl_Vertex.x / float(width - 1),
    gl_Vertex.y / float(height - 1));
```

```
  // texture-lookup (VTF)
  vec4 newPosition = texture2D(tex0, texCoord);

  newPosition.w = 1.0;

  // coordinate transformation
  gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix *
    newPosition;

  // only necessary to color the volume for its visualization:
  gl_FrontColor = gl_Color;
}
```

Der Vorteil der Verwendung von VTF liegt ganz klar in der Geschwindigkeit. Da beim Kopieren mit dem Pixelbuffer-Objekt entgegen der Spezifikation der Erweiterung offenbar über den Speicher kopiert wird, kann VTF die Performance bei AGP-Hardware stark erhöhen. Auch bei PCI-Express wurden Geschwindigkeitsgewinne festgestellt, wie zu erwarten allerdings nicht in dem Ausmaß wie bei AGP. Da sowohl mit dem Kopieren über das PBO als auch mit der VTF-Technik das Gleiche erzielt wird – nämlich das Benutzen der bereits berechneten und gespeicherten Positionen der Schattenvolumenpolygone in der Textur zum Rendern – kann zusammenfassend festgestellt werden, dass es eigentlich nur Vorteile bringt, VTF zu verwenden. Bis auf einen generellen Nachteil: Für VTF ist das Shader Modell 3.0 nötig. Die Technik ist also auf älteren Grafikkarten definitiv nicht realisierbar.

6 Geschwindigkeitsmessungen und Untersuchung der praktischen Anwendung

Der beste Algorithmus hat keinen Nutzen, wenn er zuviel Zeit für Berechnungen braucht. Zwar ist die Qualität des Hybridschattens gut, doch wie sieht es eigentlich mit der Effizienz des Hybrid Shadow Verfahrens aus?

Das Verfahren wurde auf einem Rechner mit einer AGP Grafikkarte GeForce 6800 GT sowie auf einem mit einer PCI-Express Grafikkarte GeForce 7800 GT getestet. Die Größe der Depth Map betrug vorerst in allen Tests (wie auch zuvor in den Beispielbildern) 256 x 256 Pixel. Zum Vergleich wurde ein einfaches Shadow Mapping Verfahren implementiert, welches allerdings mit drei Rendering-Passes ohne Shaderprogramme arbeitet.

Polygon-anzahl	Keine Schatten	Shadow Map	Hybrid Shadow VTF	Hybrid Shadow PBO	Hybrid Shadow Depth Blurring VTF	Hybrid Shadow Depth Blurring PBO	Hybrid Shadow Perfect Triangu-lation VTF	Hybrid Shadow Perfect Triangu-Lation PBO
500	845	555	144	119	139	113	65	36
2000	739	432	135	112	131	107	63	35
8000	472	232	110	93	105	89	58	33
32000	194	76	59	55	58	53	41	26
128000	60	22	21	20	20	19	18	14
200000	40	14	14	14	14	14	13	11
800000	11	4	4	4	4	4	4	4

Tabelle 1: Frameraten in Abhängigkeit des Schattenverfahrens und der Polygonanzahl bei einer AGP GeForce 6800 GT

Wie in Tabelle 1 und Abbildung 24 ersichtlich, ist der Hybrid Shadow Algorithmus bei kleineren Szenen (< 30.000 Polygone) gegenüber dem Shadow Mapping deutlich im Nachteil, wobei die Qualität der Schatten des Shadow Mapping mit gleicher Auflösung der Depth Map natürlich nicht an die der Hybrid Shadows heranreicht. Gerade bei komplexeren Szenen kann die Hybrid Shadow Technik allerdings ihre Vorteile ausspielen. Bei einer zu rendernden Polygonanzahl ab 32.000 ist der – immer mehr schrumpfende – Effizienzunterschied der beiden Verfahren durch die gesteigerte Qualität der Schatten beim Hybrid Shadowing auf jeden Fall zu rechtfertigen.

Besonders die Optimierung „Depth Blurring“ benötigt kaum mehr Ressourcen als Hybrid Shadow ohne Optimierungen, weshalb der praktischen Verwendung von Hybrid Shadows mit Depth Blurring ab einer dazustellenden Polygonanzahl von 32.000 kaum etwas im Wege stehen dürfte. Perfect Triangulation hinkt dem – die Effizienz betreffend

– leider etwas hinterher, was an der fast vierfach größeren Textur liegt, die bei dieser Optimierung zur Anwendung kommt.

Vertex Texture Fetch kann auf AGP Hardware die Geschwindigkeit der Hybrid Shadows extrem steigern. In der Messung liegt die Framerate bei bis zu 180% des Wertes ohne VTF. Besonders effektiv ist dies bei Perfect Triangulation, da dort die mit dem Pixelbuffer-Objekt zu kopierenden Daten nahezu viermal so groß sind wie sonst.

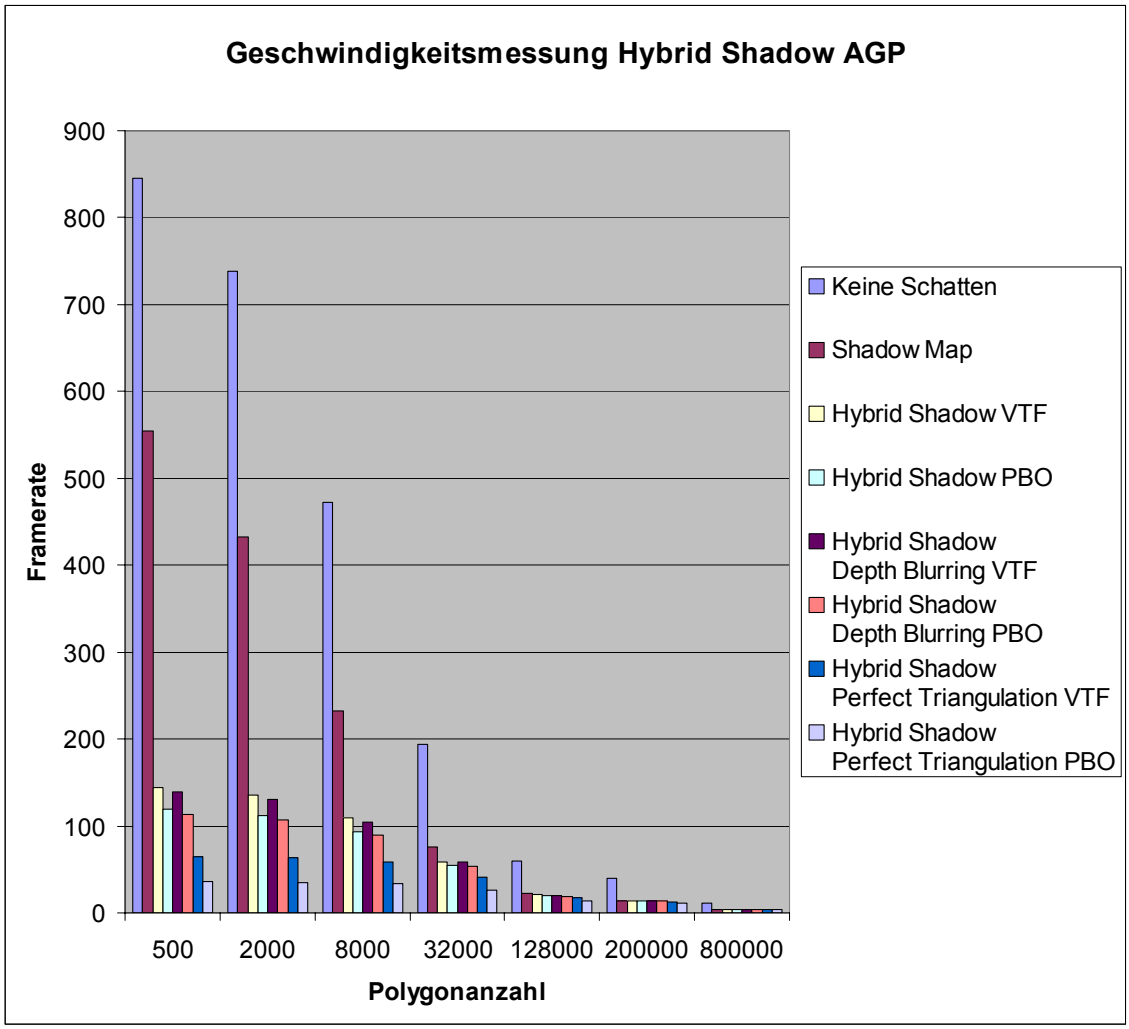

Abbildung 24: Frameraten in Abhängigkeit des Schattenverfahrens und der Polygonanzahl bei einer Depth Map Auflösung von 256x256 Pixeln

Bei PCI-Express Hardware schrumpft der Vorteil von Vertex Texture Fetch etwas, wie in Abbildung 25 ersichtlich. Der Grund dafür liegt auf der Hand: Die Daten, die beim Kopieren mit AGP über den langsamen AGP-Bus transportiert wurden, fließen hier mit PCI-Express-Geschwindigkeit.

Desweiteren fällt auf, dass Shadow Mapping gegenüber dem Hybrid Shadow Verfahren bei der schnelleren Hardware an Boden verliert. Während z.B. bei der AGP Grafikkarte 6800 GT von NVIDIA das Shadow Mapping bei 8000 Polygonen noch weit mehr

als doppelt soviele Frames liefert wie Hybrid Shadow mit Depth Blurring, sind bei der PCI-Express Grafikkarte 7800 GT mit 8000 Polygonen nur noch ca. eineinhalb mal soviele festzustellen.

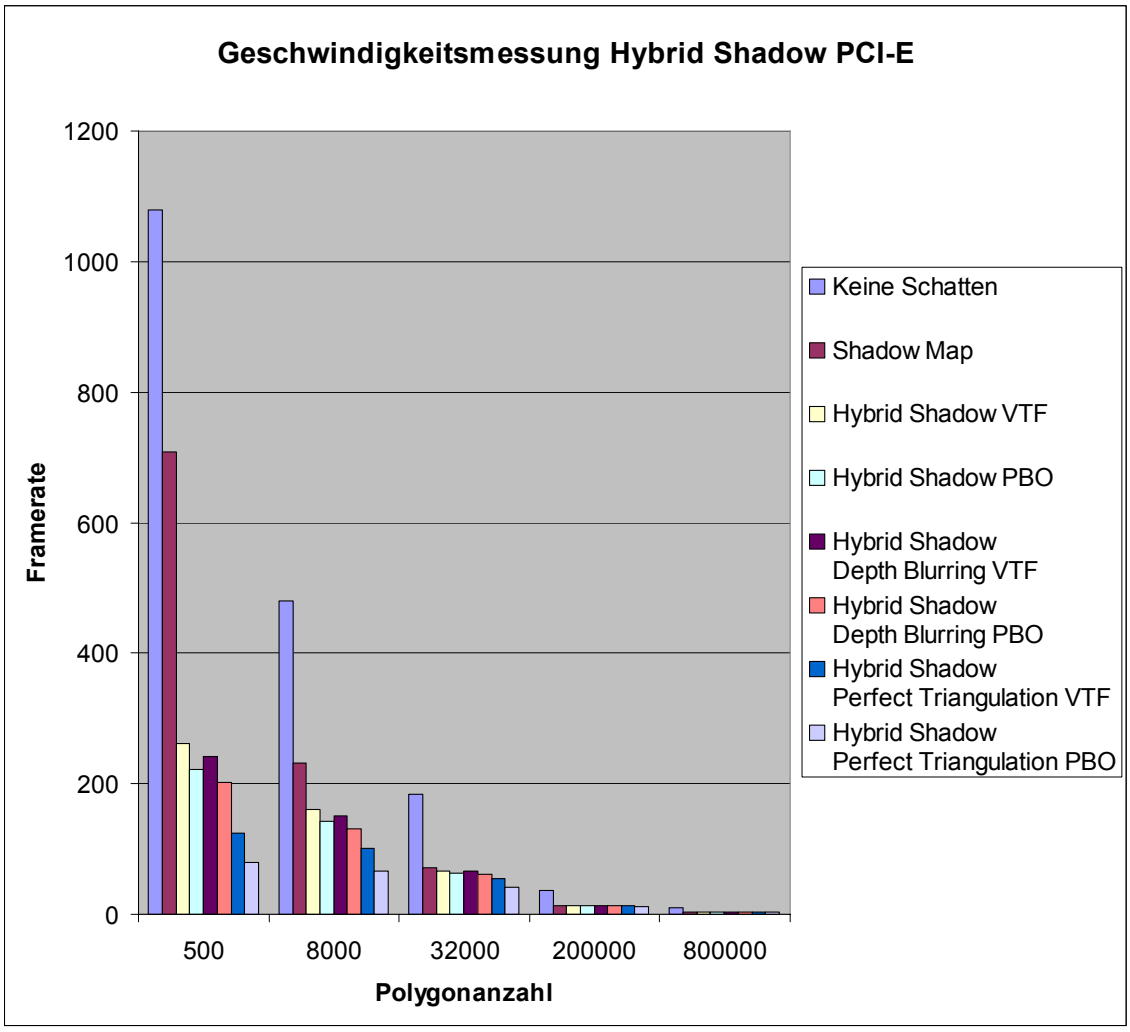

Abbildung 25: Frameraten in Abhängigkeit des Schattenverfahrens und der Polygonanzahl bei einer Depth Map Auflösung von 256x256 Pixeln und einer PCI-Express Grafikkarte GeForce 7800 GT

Nun wurde zusätzlich die Geschwindigkeit der Schattenberechnungsverfahren mit AGP Hardware und einer größeren Auflösung der verwendeten Depth Map gemessen, da diese Größe entscheidenden Einfluss auf die Qualität des Schattens hat. Wie in Abbildung 26 zu sehen fällt dieser Test leider eher negativ für Hybrid Shadows aus. Offensichtlich skalieren Hybridschatten mit der Erhöhung der Größe der Depth Map im Gegensatz zur Erhöhung der Komplexität der zu rendernden Polygone nicht so gut wie Shadow Maps.

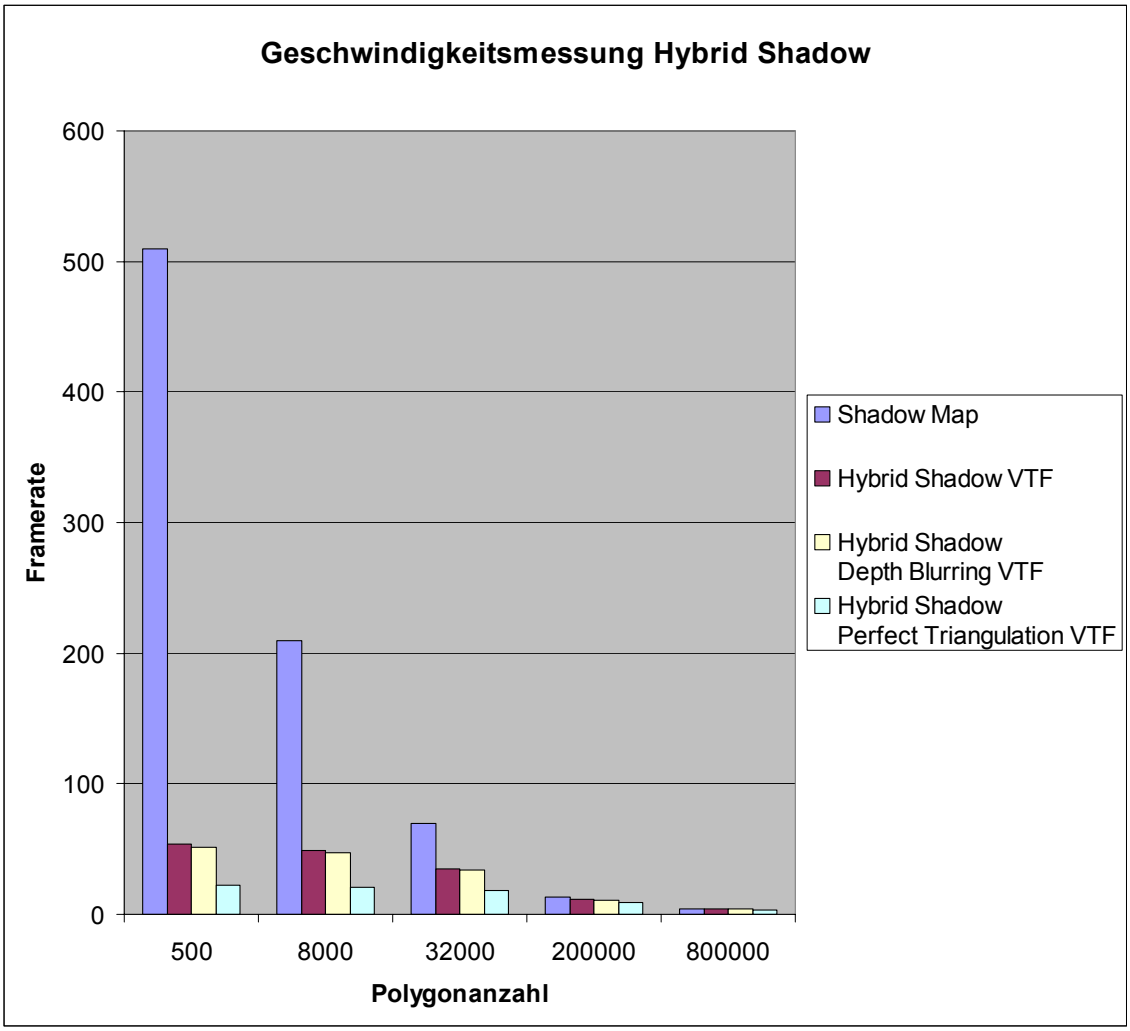

Abbildung 26: Frameraten in Abhängigkeit des Schattenverfahrens und der Polygonanzahl bei einer Depth Map Auflösung von 512x512 Pixeln

Zu Beachten bleibt dennoch, dass Hybridschatten mit einer Depth Map-Auflösung von 256x256 Pixeln – je nach Situation, Szene und Optimierungsverfahren – sogar genauso gut aussehen können wie Shadow Maps mit einer entsprechenden Auflösung von 2048x2048 Pixeln! Es bleibt also immer zwischen Performance und Qualität der Schattenberechnungs- und –darstellungsmethoden abzuwägen und zu vergleichen.

7 Zusammenfassung, Fazit und Ausblick

Wie Eingangs festgestellt wurde, gibt es eine Vielzahl von Möglichkeiten, Schatten in der Computergrafik zu berechnen und darzustellen. Jedes Verfahren hat gegenüber den anderen Vor- und Nachteile, ein perfektes Verfahren gibt es bislang nicht. So verhält es sich auch mit den hier vorgestellten Hybridschatten.

Erarbeitet wurde eine Möglichkeit, robuste Schatten in Echtzeit darstellen zu können. Mit der vorgestellten Technik werden sowohl die Nachteile von Shadow Volumes als auch die der Shadow Maps nahezu vollständig eliminiert. Hybrid Shadows sind generell einsetzbar und reduzieren die vom Shadow Mapping bekannten Aliasing-Artefakte, die von der Depth Map herrühren.

Wenn Hybridschatten mit herkömmlichen Shadow Maps verglichen werden, sollte immer bedacht werden, dass zur Erlangung der Qualität der Hybridschatten durch Shadow Maps i.d.R. eine weitaus höher aufgelöste Depth Map vonnöten ist, als bei Hybrid Shadow, wodurch je nach Verwendungszweck und Situation Hybridschatten den Vorzug bekommen können. Besonders bei Szenen, in denen für gewöhnlich wenig filigrane Objekte vorkommen, sind Hybridschatten evtl. eine sehr gute Wahl, da der Schatten zwar nicht detaillierter als bei einer gleich aufgelösten Shadow Map Methode ausfällt, jedoch um einiges besser aussieht.

Besonders bei einer darzustellenden Polyonzahl von mehr als 32.000 erweist sich die Hybrid Shadow Technik nach den Geschwindikeitsmessungen in dieser Arbeit mit heutigen Grafikkarten als eine praktikable Alternative zu Shadow Maps.

Es bleibt abzuwarten, wie sich die Geschwindigkeitsdifferenz der Verfahren in zukünftigen Grafikkartengenerationen ändert. Außerdem wäre es z.B. durchaus viel versprechend, ein Soft-Shadow-Verfahren für Hybridschatten zu entwickeln, um die Qualität nochmals zu steigern.

Literaturverzeichnis

[Wil78] Williams, L. (1978): Casting curved shadows on curved surfaces. In Computer Graphics (Proceedings of SIGGRAPH 78), vol. 12, 270–274.

[Cro77] Crow, F. C. 1977. Shadow algorithms for computer graphics. In Computer Graphics (Proceedings of SIGGRAPH 77), vol. 11, 242–248.

[Eve02] Everitt, C., and Kilgard, M. J. 2002. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. Veröffentlicht online auf developer.nvidia.com.

[McC00] McCool, M. D. 2000. Shadow volume reconstruction from depth maps. ACM Transactions on Graphics 19, 1 (January), 1–26. ISSN 0730-0301.

[NVI2.4.0] NVIDIA GPU Programming Guide, erhältlich online auf http://developer.nvidia.com/object/gpu_programming_guide.html

[NVITex] NVIDIA OpenGL Texture Formats, erhältlich online auf http://developer.nvidia.com/object/nv_ogl_texture_formats.html

[FBO05] EXT_framebuffer_object Erweiterung für OpenGL, Spezifikation erhältlich online auf http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt

[DictSha] Die Definition für „shadow“ in der freien Enzyklopädie „The Free Dictionary“ im Internet, erhältlich online auf http://www.thefreedictionary.com/shadow

[Kil01] Mark J. Kilgard, “Robust Stencil Volumes”, CEDEC 2001 Präsentation, Tokyo, 4. September, 2001

[VBO03] ARB_vertex_buffer_object Erweiterung für OpenGL, Spezifikation erhältlich online auf http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt

[VP304] NV_vertex_program3 Erweiterung für OpenGL, Spezifikation erhältlich online auf http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program3.txt

[DCL03] NV_depth_clamp Erweiterung für OpenGL, Spezifikation erhältlich online auf http://oss.sgi.com/projects/ogl-sample/registry/NV/depth_clamp.txt

[Ste03] EXT_stencil_two_side Erweiterung für OpenGL, Spezifikation erhältlich online auf http://oss.sgi.com/projects/ogl-sample/registry/EXT/stencil_two_side.txt

[PrR02] NV_primitive_restart Erweiterung für OpenGL, Spezifikation erhältlich online auf http://oss.sgi.com/projects/ogl-sample/registry/NV/primitive_restart.txt

[Bak02] Einige effizient implementierte Mathematikfunktionen, erhältlich online auf http://www.paulsprojects.net

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, daß die Arbeit veröffentlicht wird und daß in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 01.Februar 2006

(Thomas Möck)