Fachbereich 4: Informatik Arbeitsgruppe Aktives Sehen

Erweiterung der Spielegraphik von Cam2Dance durch den Einsatz von Shadern und komplexen Modellen

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Kathrin-Jennifer Kunze

Betreuer:	Dipl.-Inf. Detlev Droege, Institut für Computervisualistik, Fachbereich Informatik
Erstgutachter:	Dipl.-Inf. Detlev Droege, Institut für Computervisualistik, Fachbereich Informatik
Zweitgutachter:	Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik

Koblenz, im Oktober 2006

Erklärung

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

Die Richtlinien der Arbeitsgruppe für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. ja ☐ nein ☐

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. ja ☐ nein ☐

Koblenz, den ..... Unterschrift

Inhaltsverzeichnis

Verzeichnis der Bilder

Kapitel 1

Einleitung

Das Projekt „Cam^2Dance“ wurde 2005 von der Universität Koblenz ins Leben gerufen. Ziel war es ein Tanzspiel zu entwickeln, welches mit alternativen Tanzspielen (siehe Kapitel 2), die auf dem Markt vertreten waren, konkurrieren konnte. Benutzerinteraktionen sollten dabei ausschließlich mittels Stereokameras aufgenommen und anschließend verarbeitet werden. Dieser Herausforderung haben sich fünf Studenten gestellt, die ihre Diplom- bzw. Studienarbeiten der erfolgreichen Entwicklung dieses Projektes widmeten. Gregory Orchard war, im Umfang seiner Diplomarbeit, für die gesamte Spielearchitektur zuständig. Fabian Graf nahm sich in seiner Diplomarbeit dem Problem des Trackings an. Alle Angelegenheiten, die das Audio Systems betrafen, wurden, im Rahmen seiner Studienarbeit, von Daniel Brehme übernommen. Seine Aufgabe war es das Tanzspiel durch Musik aufzuwerten sowie einen Algorithmus zu schreiben, der die Pfeile abhängig vom jeweiligen Musikstück beat-genau setzt. Die Modellierung und Animation des virtuellen Tänzers, der dem Benutzer des Spiels das korrekte Schrittmuster vortanzen sollte, übernahm Frédéric Jochum in seiner Studienarbeit. Ioannis Mihailidis war als studentische Hilfskraft für die Leitung des Projektes zuständig.

Das Projekt bot über die bereits verteilten grundlegend zu verrichtenden Arbeiten noch weitere Aufgaben, die für Studienarbeiten Inhalte boten. Nach Gesprächen mit der Projektgruppe, welche Erweiterungen möglich und sinnvoll wären, habe ich mich für eine

Studienarbeit im Rahmen des Cam^2Dance Projektes entschieden. Die resultierende Aufgabe wird im Folgenden beschrieben.

1.1 Aufgabenstellung

Bislang sind drei relativ einfach modellierte Szenen in dem Tanzspiel vorhanden, in denen sich der virtuelle Tänzer bewegen kann. Um für Abwechslung und Vielfalt zu sorgen, sollen im Rahmen dieser Studienarbeit weitere Szenen in Maya (siehe Kapitel 3.1) modelliert und mittels eines Importers für Maya Modelle in das Tanzspiel geladen werden. Der Umweg, der bislang über die Modellierung in 3DStudioMax, dem Import in Milkshape3D und das anschließende Laden der Szene in das Spiel verläuft, soll auf diese Weise umgangen werden.

Um die Graphik des Spiels optisch aufzuwerten und das Potenzial der verfügbaren Graphikhardware nutzen zu können, sollen darüber hinaus Vertex und Fragment Shader geschrieben werden. Es werden Schatten mittels des Shadow Maps Algorithmus (Kapitel 3.6) implementiert um eine realitätsnähere Wirkung von Szenen zu erzielen sowie ein Toon-Shader (Kapitel 3.5.1), der für eine entgegengesetzte, realitätsfremde Wirkung in anderen Szenen sorgen soll.

1.2 Aufbau der Arbeit

Im Anschluss der Einleitung, in Kapitel 2, wird auf alternative Tanzspiele, die auf dem Markt zu erwerben sind, eingegangen. Dabei wird sowohl das allgemeine Spielprinzip vorgestellt, als auch Erweiterungen, um die sich die Spiele im Laufe der Zeit entwickelt haben, beschrieben.

Kapitel 3 dient vorwiegend der Vermittlung von Grundlagenwissen, welches für das Verstehen dieser Arbeit erforderlich ist. Am Anfang des Kapitels wird Maya kurz vorgestellt. Anschließend wird auf OpenGL sowie die OpenGL Pipeline eingegangen, verwendete Beleuchtungsmodelle werden erläutert und ein Einblick in Non-photorealistic Rendering ge-

geben. Darüber hinaus wird die Funktionsweise des Shadow Maps Algorithmus beschrieben sowie die Shading Language CG (C for Graphics) von NVIDIA kurz vorgestellt.

Der praktische Teil der Arbeit ist in Kapitel 4 beschrieben. Zu Beginn wird auf die Modellierung der Szenen in Maya und das anschließende Einbinden der Modelle in OpenGL eingegangen. Anschließend wird auf die Implementierung des Toon-Shaders und des Shadow Maps-Shaders eingegangen.

In Kapitel 5 werden die Ergebnisse der Arbeit präsentiert. Darüber hinaus werden Verbesserungsmöglichkeiten aufgezeigt.

Kapitel 2

Alternative Tanzspiele auf dem Markt

Im Jahre 1998 war es die Firma Konami, die die Ära der Tanzspiele mit dem Spiel „Dance Dance Revolution“ eröffnet hat. Seitdem kommen regelmäßig neue Tanzspiele und Variationen bereits bestehender auf den Markt. Ursprünglich wurde das Spiel, wie Abbildung 2.2[1] zeigt, für Spielhallen in Japan entwickelt. Mittlerweile gibt es allerdings auch Versionen für Dreamcast, PlayStation 1 und 2, GameCube, Xbox sowie für Windows und Mac.

Bild 2.1: Tanzmatte von 2-TECH für den PC

Bild 2.2: Zwei DDR Tanzmatten

[1] http://de.wikipedia.org/wiki/Dance_Dance_Revolution

Das Prinzip von Dance Dance Revolution (DDR) ist relativ einfach. Auf dem Bildschirm werden im Takt eines Liedes Pfeile angezeigt, die vom unteren zum oberen Bildschirmrand entlang laufen. Während dieses Vorgangs hat der Spieler Zeit sich das Muster anzusehen. Sobald sich die Pfeile über eine vorgegebene Position am oberen Bildschirmrand (siehe Abb. 2.3[2]) bewegen, liegt es am Spieler, der während des Spiels auf einer Tanzmatte steht, das entsprechende Feld auf der Tanzmatte (siehe Abb. 2.1[3]) zu betreten.

Bild 2.3: Spielbildschirm DDR

Sobald der Spieler ein Tanzfeld betritt, wird die Eingabe von Sensoren aufgenommen und weitergeleitet. Auf diese Weise kann überprüft werden, ob der Spieler nach dem vorgegebenen Pfeilmuster tanzt. Spielern von DDR wird auch die Möglichkeit geboten zu zweit gegeneinander anzutreten. Auch hier kommt es darauf an, nach dem vorgegebenen Pfeilmuster zu tanzen.

Stepmania ist vom Prinzip her ähnlich wie Dance Dance Revolution. Ursprünglich sollte es DDR als kostenloses[4] Open Source Spiel simulieren. Mittlerweile wurde es um weitere Aspekte erweitert. So können beispielsweise eigene Musikstücke hinzugefügt werden, für

[2] http://www.demonews.de/galerie/index.php?spieleid=1113
[3] http://www.2-tech.de/index.php?menu=2&pid=93
[4] http://www.softonic.de/file.phtml?&id_file=33151&action=view&view=downloads

die man allerdings manuell die Tanzschritte eingeben muss, nach denen im Spiel getanzt werden kann.

Über die bereits erwähnten Funktionalitäten hinaus, verfügt „Dance Factory“ über eine weitere: jedes Lied einer Musik-CD kann vom Programm eingelesen werden. Es werden automatisch, dem Rhythmus angepassten, Schrittfolgen generiert, nach denen im Spiel getanzt werden muss. Negativ zu beurteilen ist der Aspekt, dass nicht alle Musikstücke korrekt eingelesen werden und somit die Tanzschritte nicht im Takt der Musik gesetzt werden. Darüber hinaus sind nur sehr wenige der automatisch erzeugten Schritte annähernd anspruchsvoll.

Verfolgt man die Entwicklung der Spiele über die Jahre hinweg, und vergleicht die gebotenen Funktionalitäten, so kann man feststellen, dass die Tanzspiele des öfteren mit neuen Features werben, um sich von den anderen Spielen hervorzuheben. Dieses Absetzen ist mittlerweile nötig, da vielzählige Variationen von Tanzspielen käuflich zu erwerben sind. Der Druck, ein besseres Spiel als andere Hersteller bereits veröffentlicht haben zu entwickeln, kann, wie das Beispiel von „Dance Factory“ zeigt, dazu führen das eigentliche Ziel aus den Augen zu verlieren. Primäres Ziel sollte sein, dass die angebotenen Features volle Funktionalität besitzen und somit auch von den Benutzern in vollem Umfang akzeptiert werden. Eine neue, nur mäßig funktionierende Entwicklung wird selten von ambitionierten Spielern akzeptiert.

Kapitel 3

Grundlagen

3.1 Maya

Maya (von Alias entwickelt) gilt als eine leistungsstarke, flexible und umfangreiche 3D-Graphik-Software. Sowohl die Modellierung als auch die Animation von 3D Objekten ist durch die Möglichkeit der Personalisierung des Arbeitsbereichs an die individuellen Bedürfnisse des Benutzers sehr ergonomisch und erlaubt somit ein effizientes Arbeiten. Es wird die Möglichkeit geboten Objekte mit Hilfe von Polygonen, NURBS, Subdivision Surfaces oder Bézier-Splines zu erstellen. Für das Rendering stehen zwei interne Renderer (Maya Software, Maya Hardware) sowie zwei eigentlich eigenständige, eingebettete Renderer (MentalRay, Vector) zur Verfügung[1].

Für die vorliegende Arbeit wurde Maya ausschließlich verwendet um die Objekte einer Szene zu modellieren. Animationen wurden dabei komplett ausgespart. Da die Möglichkeit besteht Texturen in Maya hinzuzufügen, wurden die Objekte in Maya mit Texturen versehen und anschließend mittels eines geeigneten Exporters exportiert.

[1] vgl. http://de.wikipedia.org/wiki/Maya_%28Software%29

3.2 Open Graphics Library

„OpenGL ist eine Spezifikation für ein plattform- und programmiersprachenunabhängiges API zur Entwicklung von 3D-Computergrafik.[2]“ Um die Plattformunabhängigkeit zu gewährleisten, werden keine plattformspezifischen Aktionen von OpenGL unterstützt. Dies bedeutet beispielsweise, dass keine Aktionen auf Fenstern möglich sind, sowie keine Möglichkeit geboten wird Benutzereingaben auszulesen. Zur Unterstützung können Bibliotheken wie z. B. GLUT eingesetzt werden. Aus dem eben genannten Grund bietet OpenGL auch keine Funktion um Bilddateien zu laden. Ein externer Image Loader, wie er z. B. von SDL-Image bereitgestellt wird, kann für diese Aufgabe verwendet werden. Erst dann kann OpenGL die geladene Bilddatei als Textur nutzen.

Objekte, die eine gewisse Komplexität nicht übersteigen, können direkt mittels OpenGL Funktionen erzeugt werden. Für komplexe Objekte oder aufwendige Szenen empfiehlt es sich die Modelle mit einem 3D-Modellierungstool wie z. B. Maya zu erstellen und anschließend mittels eines Loaders in OpenGL zu laden. Nebel, Transparenzen sowie weitere einfache Effekte können mit Hilfe von OpenGL Funktionen erzeugt werden. Für weitere spezielle Effekte können Shader geschrieben werden, die einen Teil der OpenGL Pipeline ersetzen. Die mehrstufige Pipeline wird von den einzelnen Vertices der Objekte durchlaufen und im Laufe dieser Verarbeitung in Fragmente konvertiert, die in den Framebuffer geschrieben werden. Bis zu dieser Stufe steht es dem Programmierer frei die Vertices bzw. Fragmente zu manipulieren. Nachdem die Fragmente als letzten Schritt in Pixel konvertiert wurden, werden diese letztendlich als 2D-Bild ausgegeben. Weitere Einzelheiten der Verarbeitung werden im folgenden Kapitel 3.3 erläutert.

3.3 OpenGL Pipeline

Zu Beginn befinden sich die Vertexkoordinaten in einem Koordinatensystem relativ zum Objekt zu dem diese jeweils gehören. Die Standard OpenGL Pipeline (siehe Abb. 3.1[3]) erhält diese Eckpunktkoordinaten sowie Farben, Normalen und eventuell Texturkoordinaten

[2] vgl. http://de.wikipedia.org/wiki/Opengl

[3] vgl. Trappe, Rodja (2003). Computergraphik 1 Stoffsammlung. Universität Koblenz.

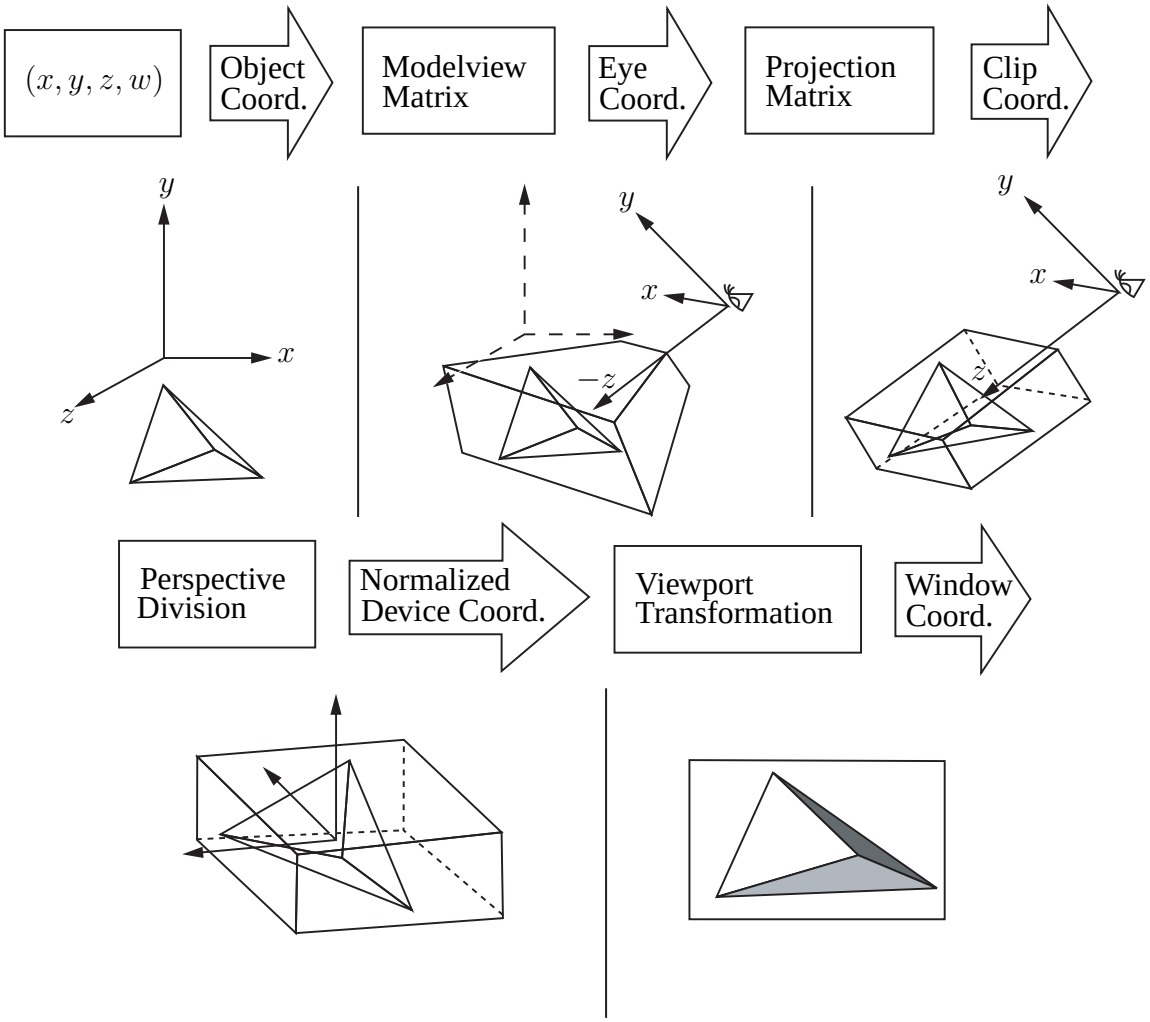


Bild 3.1: OpenGL Pipeline

für jeden Vertex als Input geliefert. Anschließend erfolgt eine Multiplikation der Vertices mit der Modelview-Matrix (2), welche alle affinen sowie die View-Transformation beinhaltet. Nach dem Vorgang liegen die Vertices im Eye-Space und die ursprünglichen Objektkoordinaten sowie -normalen liegen relativ zum Augpunkt (Eye-Coordinates) in einem Rechtssystem vor. Die per-Vertex Beleuchtung erfolgt in diesem Raum. Eine Alternative zur per-Vertex Beleuchtung stellt die per-Fragment Beleuchtung dar, welche während der Rasterisierung stattfindet. Mittels OpenGL selbst kann allerdings nur per-Vertex beleuch-

tet werden. Soll eine per-Fragment Beleuchtung stattfinden, müssen entsprechende Shader implementiert werden. Auf Vor- und Nachteile soll in dieser Arbeit allerdings nicht eingegangen werden.

Mittels der Projectionmatrix (3) erfolgt eine Drehung ins Linkssystem, sowie eine optionale perspektivische und obligatorische orthographische Projektion. An dieser Stelle befinden sich die Vertices nun im Clip-Space. In diesem homogenen Raum findet das Clipping der Objekte am View-Frustum statt. Die Homogenisierung der Koordinaten findet bei der anschließenden Perspektivischen Division (4) statt. Im letzten Schritt (5) erfolgt die Viewport-Transformation sowie die Interpolation von Farben und Texturkoordinaten. Jedes Fragment erhält in diesem Schritt seinen Farbwert, welcher während der Rasterisierung in den jeweiligen Fensterkoordinaten auf dem Bildschirm ausgegeben wird.

Jeder Rendervorgang wird von dieser Pipeline automatisch übernommen. Dem Benutzer steht es allerdings frei, einige oder auch alle Teile der Pipeline durch selbst geschriebene Shader zu ersetzen. Für gewöhnlich führt der Vertex-Shader die Umwandlung der Object Coordinates in Clip Coordinates aus und der Fragment-Shaders die Umwandlung von Normalized Device Coordinates in Window Coordinates.

3.4 Beleuchtungsmodelle

3.4.1 OpenGL Beleuchtungsmodell

OpenGL stellt ein einfaches, allerdings physikalisch unkorrektes, Beleuchtungsmodell[4] bereit, was für die meisten Szenen, sofern keine photorealistische Qualität gefordert ist, allerdings ausreichend ist:

$$L = M_e + M_a \cdot L_a + \sum_{i=1}^{\#Lq} (M_a \cdot L_{a,i} + M_d \cdot \cos\varphi_i \cdot L_{d,i} + M_s \cdot \cos^n \psi_i \cdot L_{s,i})$$

Die emissive Materialfarbe M_e wird dabei als Basishelligkeit von Objekten verwendet. Da sowohl bei der Implementierung des Toon Shaders sowie des Shadow Maps Shaders komplett auf den emissiven Teil verzichtet wird, soll darauf nicht weiter eingegangen werden.

[4] vgl. Quelle: [MÖ4a]

Die ambiente Beleuchtung $M_a \cdot L_a$ (ambiente Materialfarbe bzw. Lichtfarbe) simuliert gestreutes Umgebungslicht und ist, als einziger Teil, völlig unabhängig von der Geometrie. Die Anzahl der Lichtquellen in beiden Implementierungen ist auf 1 festgesetzt. Es kann demnach eine vereinfachte Formel verwendet werden:

$$L = M_a \cdot L_{a,i} + M_d \cdot \cos\varphi_i \cdot L_{d,i} + M_s \cdot \cos^n \psi_i \cdot L_{s,i}$$

Diese vereinfachte Formel, beschreibt das Beleuchtungsmodell nach Phong, welches im Toon Shader (siehe Kapitel 4.3) und Shadow Maps Shader (siehe Kapitel 4.4) implementiert wurde. Auf die Berechnung des diffusen sowie spiegelnden Lichtanteils wird detailliert in Kapitel 3.4.2 eingegangen.

Um die Farbe pro Eckpunkt zu berechnen wird, nach der Formel, die ambiente Materialfarbe mit der ambienten Lichtfarbe multipliziert. Diesem Produkt wird jeweils der diffuse sowie spiegelnde Term (für die Berechnung siehe Kapitel 3.4.2) aufaddiert. Dieses Modell ermöglicht es, was in der Praxis selten Anwendung findet, jeweils eine unterschiedliche Farbe für diffuses, spiegelndes sowie ambientes Licht zu wählen.

3.4.2 Phong'sches Beleuchtungsmodell

Diffuser Term Eine vollständig diffuse Oberfläche hat die Eigenschaft, einfallendes Licht gleichmäßig in alle Richtungen zu reflektieren. Dementsprechend ist die Helligkeit der Oberfläche, wie in der Abbildung 3.2 zu sehen ist, unabhängig vom Viewpoint des Betrachters.

Zur Berechnung des diffusen Lichtanteils wird der Light-Vector L, der zwischen der Position der Lichtquelle und der Eckpunktkoordinate P aufgespannt wird, normalisiert und anschließend das Skalarprodukt mit der normalisierten Oberflächennormale N berechnet.

$$dot_{NL} = |N| \circ |L| = \cos(\varphi)$$

Falls das Licht senkrecht auf den Punkt P fällt, also $N = L$ ist, ist die Beleuchtung an diesem Punkt maximal und der $\cos(\varphi) = 1$. Stehen dagegen beide Vektoren, N und L, senkrecht aufeinander, ist der $\cos(\varphi) = 0$. Der Punkt P ist in dem Fall also unbeleuchtet.

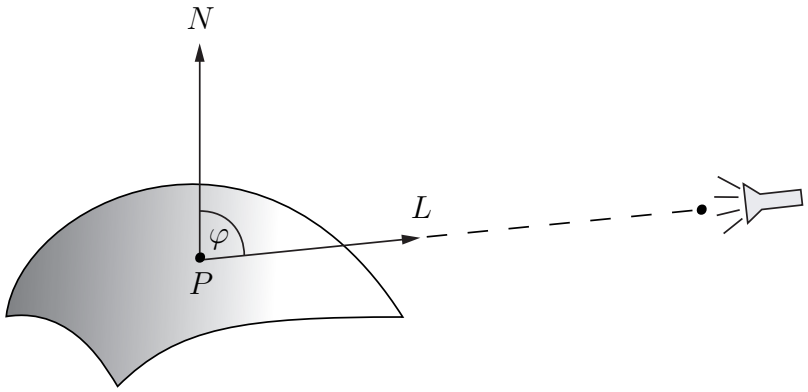


Bild 3.2: Diffuses Beleuchtungsmodell

Das Produkt der diffusen Materialfarbe M_d, der diffusen Lichtfarbe L_d sowie von dot_{NL} ergeben die diffuse Farbe L_d für den jeweiligen Eckpunkt P.

$$\textit{Diffus} = M_d \cdot L_d \cdot dot_{NL}$$

Spiegelnder Term Der spiegelnde Term ist etwas aufwendiger zu berechnen als der diffuse Term, da dieser zusätzlich noch abhängig ist von der Blickrichtung des Betrachters V, der Richtung des reflektierten Lichtstrahls R (siehe Abb. 3.3) sowie einer manuell zu bestimmenden Glanzzahl n. Auf das Verhalten der Glanzzahl wird im späteren Verlauf eingegangen. Der Viewvektor V wird aufgespannt zwischen dem Punkt P und der Position der Kamera. Der Reflexionsvektor R gibt die Richtung an, in die das einfallende Licht bei einer perfekten Spiegelung hin reflektiert wird. Die Berechnung des Reflexionsvektors R erfolgt über[5]:

$$dot_{NL} = |N| \circ |L| = \cos(\varphi)$$

$$R = 2 \cdot dot_{NL} \cdot N - L$$

Das Highlight, das bei der Berechnung des spiegelnden Beleuchtungsanteils eine wichtige Rolle spielt, ist umso heller, je direkter der Betrachter in die Spiegelung des Lichtstrahls schaut. Das bedeutet, dass das maximale Licht in die Blickrichtung reflektiert wird. Somit

[5] vgl. http://www.delphi3d.net/articles/viewarticle.php?article=phong.htm

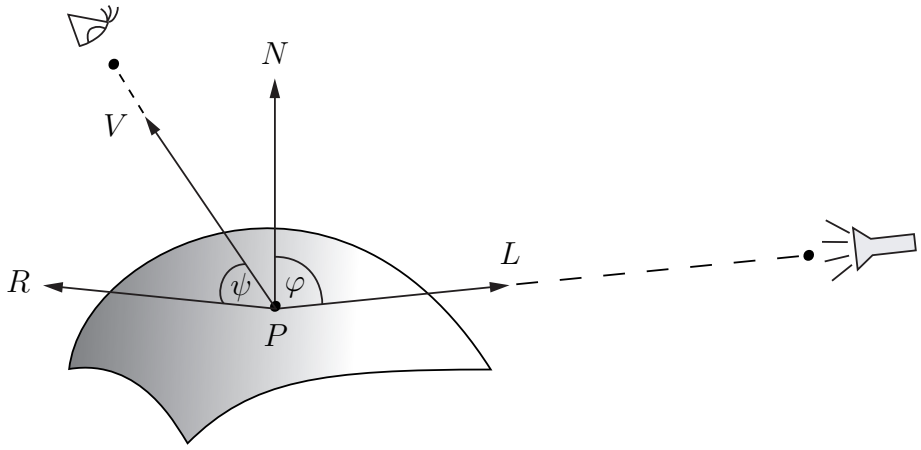


Bild 3.3: Spiegelndes Beleuchtungsmodell

wäre in dem Fall $R = V$. Je kleiner der Winkel zwischen R und V, desto stärker sind die spiegelnden Reflexionen, und somit auch das Highlight, sichtbar. Auch hier erfolgt wird das Skalarprodukt berechnet, welches für die weitere Berechnung relevant ist.

$$dot_{RV} = |R| \circ |V| = \cos(\psi)$$

Je nachdem wie groß die Glanzzahl n gewählt wird, die angibt wie glänzend (shiny) die Oberfläche sein soll, verändert sich die Größe des Highlights. Je größer die Glanzzahl gewählt wird, desto kleiner ist letztendlich das Highlight. Abbildung 3.4 illustriert diese Eigenschaften anhand eines Funktionsgraphens.

Zur finalen Berechnung des spiegelnden Lichtanteils wird die spiegelnde Materialfarbe M_s multipliziert mit der spiegelnden Lichtfarbe L_s sowie dem zuvor berechneten Wert dot_{RV}, welcher mit der Glanzzahl n potenziert wird.

$$\textit{Spiegelnd} = M_s \cdot L_s \cdot (dot_{RV})^n$$

3.5 Non-photorealistic Rendering (NPR)

„Als nicht-photorealistisch bezeichnen wir Bilder, deren Elemente zwar realistische Aspekte der Abbildung beinhalten, deren Darstellungsweise jedoch aufgrund der Verwendung

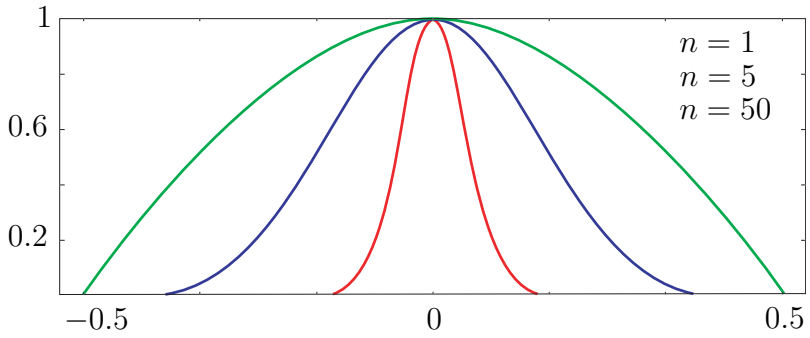


Bild 3.4: Funktionsgraph $f(x) = \cos^n(x)$.

bestimmter Stilmittel in den Bereichen Form, Farbe, Struktur, Schattierung, Licht oder Schattenwurf von der wahrnehmbaren Wirklichkeit abweichen.“ Prof. Dr. Maic Masuch von der Universität Magdeburg hat die Aspekte, die NPR im Eigentlichen ausmachen, in diesem Satz zusammengefasst. Die Abweichung von der Realität in nur einigen wenigen Punkten spielt dabei eine entscheidende Rolle. Um diese Abweichung erzeugen zu können, werden Shader programmiert. Durch die Shader entstehen dann beispielsweise geometrisch korrekte Bilder, die wie Aquarelle, Ölgemälde oder Strichzeichnungen aussehen. Es kann aber auch, anstatt der klassischen OpenGL Beleuchtung, eine nichtphotorealistische Beleuchtung implementiert und verwendet werden, wie es etwa beim Toon-Shader der Fall ist. Auch Schatten, die nicht der Realität entsprechen, sind denkbar und ebenfalls durch Shader realisierbar.

Oftmals wird dem non-photorealistic Rendering allerdings das photorealistische Rendering vorgezogen, um eine wirklichkeitsgetreue Darstellung von Objekten zu erzielen. Auch in diesem Fall sind es Shader, die programmiert werden müssen. So können etwa Shadow Maps implementiert werden um eine realistische Darstellung von Schatten zu erhalten oder aber ein physikalisch korrektes Beleuchtungsmodell realisiert werden, um eine realistische Beleuchtung der Objekte zu erzielen. Es lassen sich unzählige Shader programmieren, die zu einer erhöhten realitätsnahen Darstellung von Oberflächen führen. Beispielsweise ist eine reflektierende animierte Wasseroberfläche, oder eine raue metallene Struktur denkbar.

Da die Graphik des Tanzspiels allerdings nicht auf photorealistische Qualität zielt, wird in dieser Arbeit nicht auf weitere photorealistische Aspekte eingegangen.

3.5.1 Cartoon Look

Wie der Name schon sagt, soll erzielt werden, dass Objekte aussehen wie Zeichnungen in einem klassischen Comic Heft. Die Figuren haben (meist) nur einen Farbton für Haut, Haare und Kleidung. Die Schattierung dieser erfolgt allerdings nicht im Verlauf sondern stufenweise. Für ein blaues Shirt werden dann z. B. ein helles und ein dunkles Blau verwendet. Darüber hinaus sind die Konturen der Figuren mit einem schwarzen Stift gezeichnet. Um einen Effekt im Cartoon-Look zu erhalten, gilt es also mehrere Punkte zu implementieren:

- eine diffuse Beleuchtung, die durch sehr weniger Werte (2 bis 5) repräsentiert wird
- spiegelnde Highlights, die von einer hellen Farbe mit hoher Intensität repräsentiert werden
- eine schwarze Umrandung aller Objekte

Um eine Beleuchtung zu erzeugen, die ihre Helligkeit nicht kontinuierlich sondern in Stufen ändert, kann eine Stufenfunktion verwendet werden. Die Beleuchtungswerte werden mit Hilfe dieser Stufenfunktion (die als Look-up Tabelle in einer 1D Textur vorliegt) diskretisiert. Auf diese Weise erhält man den gewünschten Effekt. Auf die detaillierte Berechnung der diffusen sowie spiegelnden Beleuchtung wurde bereits ausreichend in 3.4.2 eingegangen. Die Erzeugung der Kontur wird im Implementationsteil 4.3 beschrieben.

3.6 Schatten

Von Natur aus sind wir es gewohnt Schatten in unserer Umgebung zu sehen. Aus dieser Gewohnheit heraus scheinen daher Szenen, die keine Schatten beinhalten, unrealistisch. Weiterhin wird es dem Betrachter erschwert die räumliche Anordnung einer Szene korrekt

und ohne größere Schwierigkeiten zu erfassen[6]. Um diesen Punkten gerecht zu werden, sind bereits einfache Schatten wie precomputed Shadows[7] ausreichend. Hat man es allerdings mit einer dynamischen Szene zu tun, in der sich Objekte verändern oder bewegen, müssen aufwendigere Algorithmen zur Anwendung kommen. Die beiden Algorithmen, die primär zum Einsatz kommen, Shadow Maps und Shadow Volumes, unterscheiden sich stark in ihrer Implementierung, erzielen allerdings beide sehr gute Ergebnisse. Es gibt aber dennoch Gründe, die für Shadow Maps sprechen:

Shadow Volumes müssen sobald sich ein Shadow Caster oder die Lichtquelle bewegt neu berechnet werden (d. h. bei animierten Modellen potentiell jedes Frame). Die benötigte Hilfsgeometrie belegt abhängig von der Größe des Modells zusätzlichen Speicher. Bei naiver Implementierung sind Shadow Volumes recht langsam, da sie einen hohen Fillrate Bedarf haben. D. h. große Teile des Bildschirms müssen mehrfach pro Frame überzeichnet werden.

Ein weiterer Vorteil von Shadow Maps gegenüber Shadow Volumes ist, dass es einfacher ist aus den pixeligen harten Schattenkanten einen weichen unscharfen Rand zu erzeugen als es bei dem harten Schattenrand, der Shadow Volumes der Fall ist. Shadow Maps haben sich weiterhin in Outdoor Szenen (Spiele wie Battlefield 2, FarCry) bewährt, während Shadow Volumes eher in Indoor Szenen (Spiele wie Quake 4, Doom 3) Verwendung gefunden haben. Da ich für die Szenen im „Cam^2Dance“ Spiel zwei Outdoor Szenen vorgesehen habe, ist dies ein weiterer positiver Aspekt der für den Shadow Maps Algorithmus spricht.

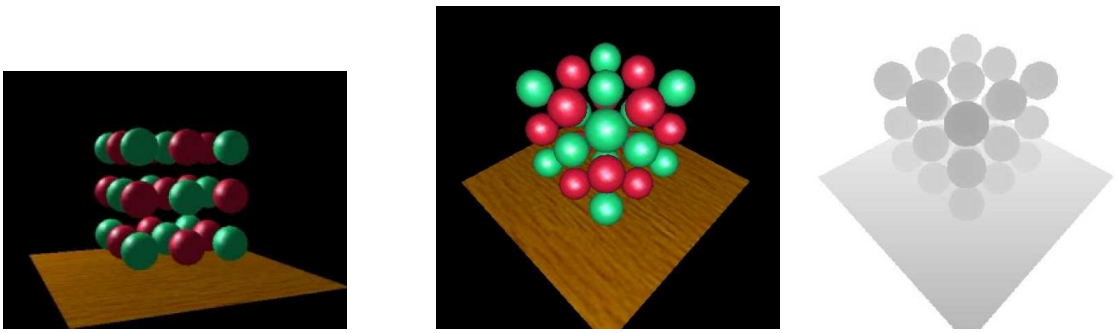
3.6.1 Shadow Mapping

Shadow Mapping ist ein Schatten-Algorithmus, der aus zwei Schritten besteht. Der erste Schritt betrifft das Erzeugen der Shadow Map und der zweite beinhaltet den Vergleich der Tiefenwerte sowie das Rendern des Bildes. Im Folgenden wird auf beide Schritte im Detail eingegangen. Der vorgestellte Algorithmus bezieht sich dabei auf Spotlight Lichtquellen.

[6] vgl. Quelle: [Sch03]

[7] Precomputed Shadows sind vorberechnete Schatten, die in Lightmaps gespeichert werden. In der eigentlichen Szene werden sie nur noch als Textur dargestellt. Die Schatten sind statisch, die aber den Vorteil haben, dass keine Berechnung im eigentlichen Programm stattfinden muss.

Erzeugen der Shadow Map Zu Beginn wird die Position und Blickrichtung des Betrachters $P_B(x, y, z)$ (siehe Abb. 3.5 (a)) auf die Position und Richtung der Spotlight Lichtquelle $P_L(x, y, z)$ (siehe Abb. 3.5 (b)) gesetzt. Anschließend werden die Objekte, die Schatten werfen sollen, direkt in den Framebuffer gerendert. Die Tiefenwerte an Position (x, y) werden daraufhin aus dem Framebuffer in einer Textur $T_{SM}(s, t)$, der Shadow Map (siehe Abb. 3.5 (c)), gespeichert, um im zweiten Schritt wieder ausgelesen und zum Vergleich herangezogen werden zu können. Alternativ kann auch direkt in eine Textur gerendert werden. Punkte die von der Lichtquelle aus gesehen, größere (also weiter hinten liegende) Tiefenwerte haben, als die die in der Textur gespeichert sind, liegen im Schatten.

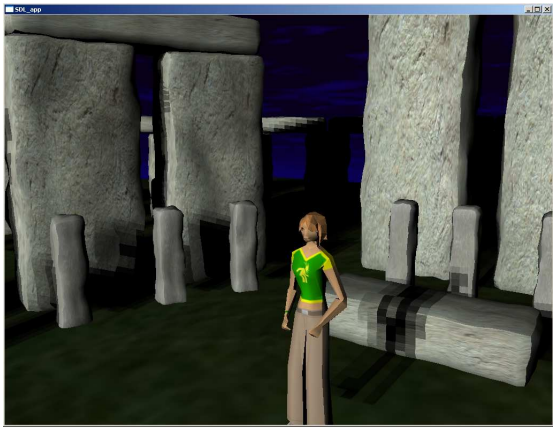


(a) Eyeview $P_B(x, y, z)$ (b) Lightview $P_L(x, y, z)$ (c) Shadow Map

Bild 3.5: Erzeugen der Shadow Map. Quelle: [Kil00]

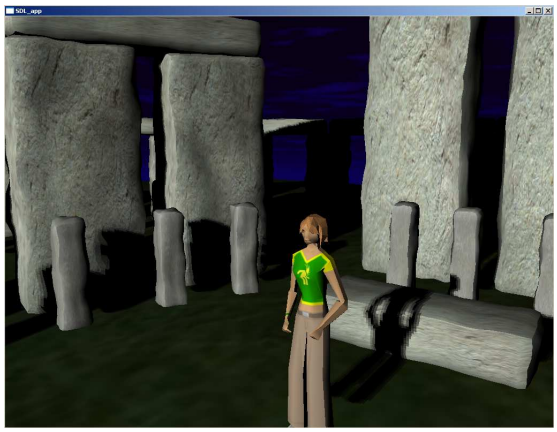
Die Qualität der Schatten, wie präzise diese sind und wie stark Aliasing Effekte zum Ausdruck kommen, hängt in hohem Maße von der Auflösung der Shadow Map Textur ab. Für große, komplexe Szenen empfiehlt es sich daher eine angemessen große Textur zu verwenden, um ein ansprechendes Ergebnis zu erzielen. Abbildung 3.6 zeigt die Ergebnisse der Schatten bei Verwendung unterschiedlich großer Shadow Maps.

Vergleich der Tiefenwerte Im zweiten Schritt wird die Szene vom Augpunkt des Betrachters $P_B(x, y, z)$ aus gerendert. Für jeden gezeichneten Pixel wird nun der Tiefenwert bestimmt, den dieser Pixel hätte, wenn er von der Lichtquelle aus betrachtet würde. Dieser Tiefenwert $T_{L,Pix}(x, y)$ des Pixels, aus Sicht der Lichtquelle, wird anschließend mit



(a) 256 x 256 Pixel

(b) 512 x 512 Pixel



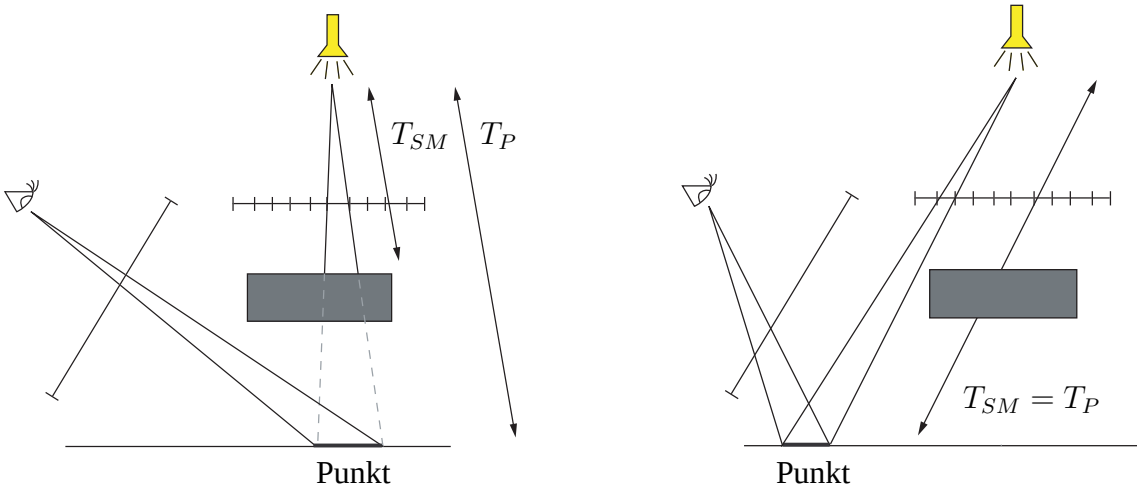
(c) 1024 x 1024 Pixel

Bild 3.6: Wahl der Shadow Map-Größe

dem korrespondierenden[8] gespeicherten Tiefenwert $T_{SM}(s,t) = z$ in der Shadow Map Textur verglichen. Durch den Vergleich ergibt sich ob ein Punkt beleuchtet wird oder nicht. Abbildung 3.7 verdeutlicht den Vergleich anhand eines Beispiels. Für den Fall, dass $T_{SM}(s,t) < T_{L,Pix}(x,y)$ ist, ist der Tiefenwert $T_{SM}(s,t)$ in der Shadow Map geringer als der Abstand des getesteten Pixels $T_{L,Pix}(x,y)$ zur Lichtquelle. Dementsprechend muss sich ein weiteres Objekt zwischen dem getesteten Punkt und der Lichtquelle befinden, was dazu führt, dass der Punkt im Schatten liegt. Andernfalls, wenn $T_{L,Pix}(x,y) = T_{SM}(s,t)$,

[8] Auf die Abbildung $T_{L,Pix}(x,y)$ –> $T_{SM}(s,t)$ wird in 4.4 genauer eingegangen

ist der Tiefenwert $T_{SM}(s,t)$, der in der Textur gespeichert ist, ungefähr gleich dem Tiefenwert des Pixels aus Sicht der Lichtquelle $T_{L,Pix}(x,y)$. Somit ist dieses Pixel von der Lichtquelle aus sichtbar und wird beleuchtet.

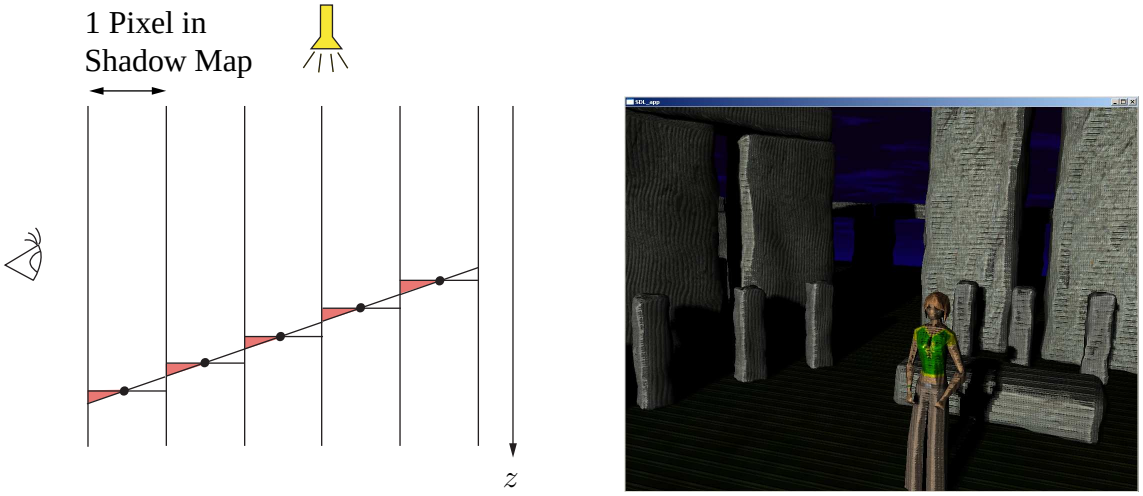


(a) $T_{SM}(s,t) < T_{L,Pix}(x,y)$ (b) $T_{L,Pix}(x,y) = T_{SM}(s,t)$

Bild 3.7: Vergleich der Tiefenwerte. Nach Quelle: [MÖ4b]

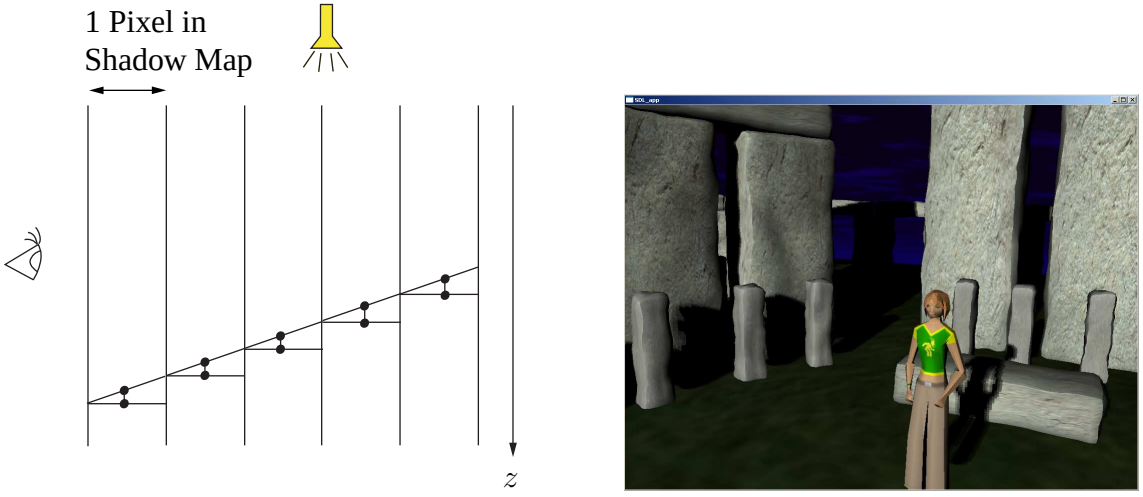
Da die Oberfläche eines Objektes nur pixelgenau abgetastet wird, kann es zu einer fehlerhaften Selbstverdeckung kommen. Abbildung 3.8 zeigt, wie sich dieser Fehler auf das Rendern der Szene auswirkt.

Durch Korrektur der Tiefenwerte $T(x,y) = z$ der Shadow Map durch einen geeigneten Offset-Wert, kann dieses Problem in der Praxis umgangen werden (vgl. Abb. 3.9). Der Wert ist szenenspezifisch und sollte daher durch Testen bestimmt werden und bewirkt, dass die z-Werte näher zur Lichtquelle bewegt werden. Bei Wahl eines geeigneten Offsets muss allerdings darauf geachtet werden, dass dieser nicht zu groß oder zu klein gewählt wird, da andernfalls die Objekte nicht korrekt beleuchtet werden.



(a) (b)

Bild 3.8: Fehlerhafte Selbstverdeckung



(a) (b)

Bild 3.9: Korrekte Wahl des Offset Wertes

3.7 C for Graphics (Cg)

Cg ist eine von NVIDIA, in enger Zusammenarbeit mit Microsoft, begründete Programmiersprache für Grafikprozessoren, welche der Syntax und der Struktur von C sehr nahe

kommt[9]. Die Zusammenarbeit mit Microsoft war erforderlich um Cg DirectX kompatibel zu gestalten. Die Kompatibilität zu OpenGL konnte ohne weitere Hilfe erreicht werden.

Diese High Level Grafiksprache soll es den Entwicklern erleichtern, visuell aufwendige und komplexe Effekte einfach und dennoch effizient zu implementieren. Kleine Recheneinheiten auf den Graphikchips sorgen für Unterstützung von selbst geschriebenen Shadern[10]. Mit den sogenannten Vertex Shadern wird in der Regel die Geometrie vom World bzw. Object Space in den Clip Space transformiert. Desweiteren können hier per-Vertex Berechnungen durchgeführt werden (beispielsweise für die per-Vertex Beleuchtung oder eine Veränderung der Vertex Positionen wie sie z. B. für Wasserwellen gebraucht werden könnte).

Texturkoordinaten, Vertexkoordinaten sowie Farben pro Eckpunkte werden auf dem Weg zum Fragment-Shader über die gesamte zu zeichnende Fläche hinweg interpoliert, bevor die endgültige Farbe pro Fragment in den Framebuffer geschrieben wird, um letztendlich auf dem Bildschirm dargestellt zu werden. Im Fragment Shader selbst können beispielsweise per-Fragment Berechnungen durchgeführt werden.

Neben Cg gibt es noch weitere Shader Sprachen wie die High Level Shader Language (HLSL), welche Bestandteil von DirectX ist, sowie die GL Shader Language (GLSL), welche fester Bestandteil von OpenGL 2.0 ist, die allerdings einen Nachteil beinhalten. Wenn Programme implementiert werden sollen, die sowohl DirectX als auch OpenGL kompatibel sein sollen, ist der Programmierer gezwungen entweder zwei Shader zu schreiben (jeweils für DirectX und OpenGL) oder aber Cg zu verwenden. Diverse Probleme lassen sich allerdings auch ohne Cg lösen, so dass Cg nicht immer erste Wahl sein muss. Unter der Voraussetzung lassen sich dann allerdings nicht alle HLSL bzw. GLSL spezifischen Features ausnutzen.

[9] vgl. http://www.nvidia.de/object/cg_de.html und http://de.wikipedia.org/wiki/C_for_graphics
[10] vgl. http://de.wikipedia.org/wiki/Shader

Kapitel 4

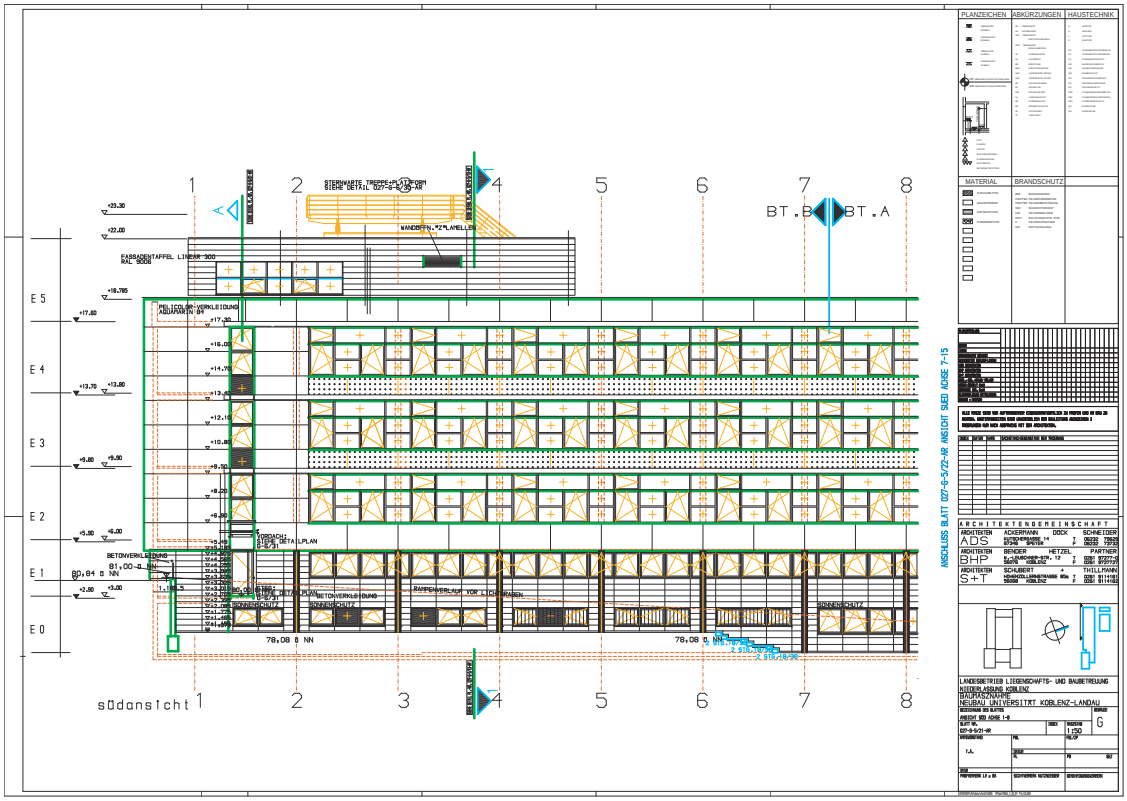
Umsetzung

4.1 Modellierung der Szenen in Maya

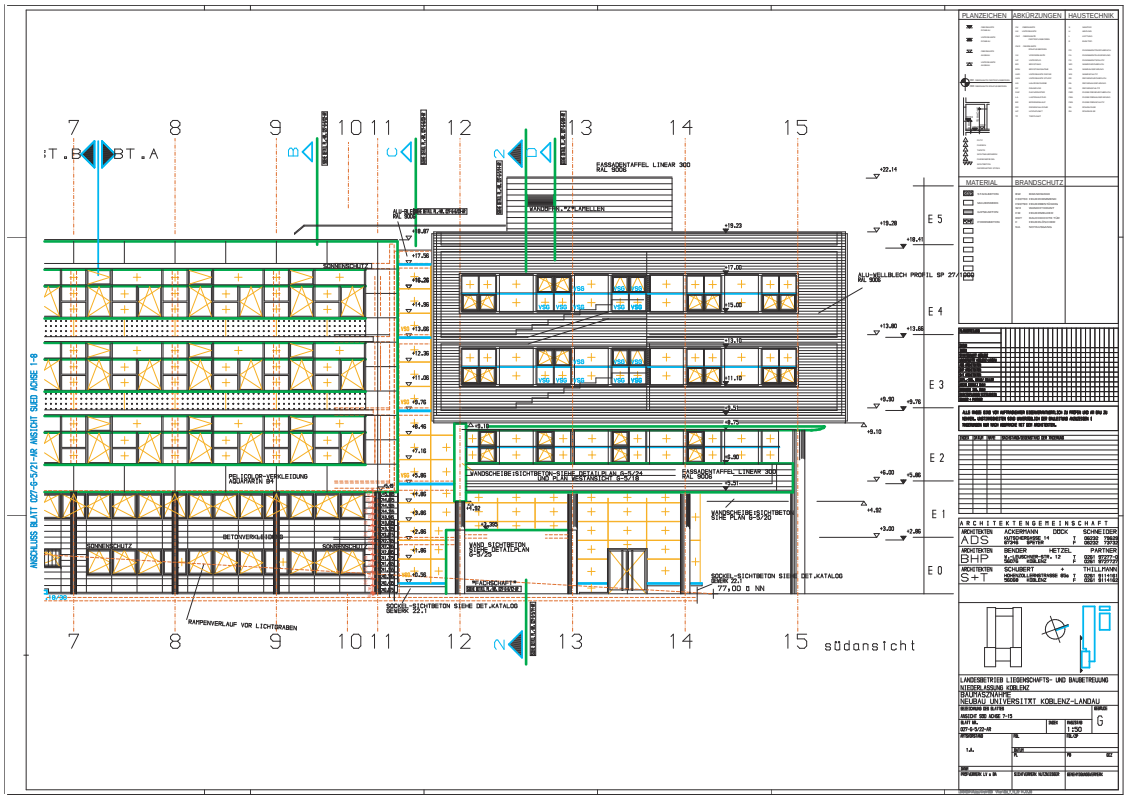
Da bereits relativ einfache Szenen für das Tanzspiel vorhanden waren, sollten mit dieser Arbeit weitere, allerdings komplexere Szenen, hinzugefügt werden, damit die Effekte, die durch die Shader anschließend implementiert werden sollten, besser zum Ausdruck kommen. Bei der Modellierung wurde ausschließlich auf Polygon Objekte zurückgegriffen, da sowohl die Campus- als auch die Stonehenge- Szene keine runden Formen aufweisen sollten, die beispielsweise aufwendigere Nurbsflächen rechtfertigen würden.

Campus der Universität Koblenz Um die einzelnen Gebäude des Campus in korrekten Verhältnissen modellieren zu können, war es nötig geeignete Pläne aller Gebäude zur Verfügung zu haben. Somit mussten zu Beginn der Arbeit die Seitenansichten der Gebäude vom Staatsbauamt Koblenz organisiert werden. Abbildung 4.1 zeigt einen Plan des G-Gebäudes.

Da die Vorlagen der großen, komplexen Gebäude des Campus aus zwei Teilen pro Ansicht bestanden, mussten diese anschließend manuell zusammengefügt werden. Daraufhin galt es die Vektorgraphiken in Pixelgraphiken umzuwandeln, damit Nord-, Ost-, Süd- und West- Ansichten jeweils als Image Plane für jedes Gebäude in eine einzelne Szene in Ma-



(a) Abschnitt 1 - 8



(b) Abschnitt 7 - 15

Bild 4.1: Südansicht G-Gebäude

ya importiert werden konnten. Über einen Alphawert, mit dem die Transparenz der Image Planes gesteuert werden kann, können Ebenen, die gerade nicht benötigt werden, ausgeblendet werden. Abbildung 4.2 zeigt das Wireframe Modell von der Bibliothek und dem C-Gebäude mit zwei der Image Planes. Auf die vorgestellte Art und Weise entstanden nach und nach die einzelnen Gebäude der Universität Koblenz von allen Seiten. Aufgrund des Aufwandes und der Übersichtlichkeit, wurden die Gebäude allerdings nur bis auf einen vorher festgelegten Detailgrad modelliert. Die Ergebnisse der Modellierung können in Kapitel 5 betrachtet werden.

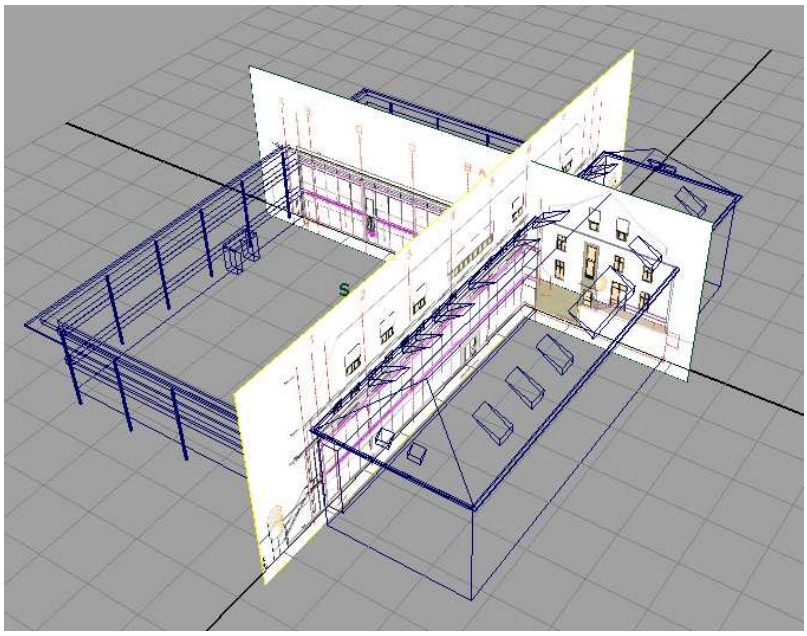


Bild 4.2: Image Plane der Gebäude C und der Bibliothek

Stonehenge Das Stonehenge Modell ist auf ähnliche Weise modelliert worden, wobei hier nur eine Aufsicht (siehe Abb. 4.3[1]), die als Image Plane in das Projekt importiert wurde, als Vorlage diente. Die Höhenverhältnisse wurden aus Bildern heutiger Zeit, in denen beispielsweise Touristen neben den antiken Steinen posierten, abgeschätzt und sind

[1] http://www.fromoldbooks.org/OldEngland/pages/0002-Stonehenge-Restored-plan/0002-Stonehenge-Restored-plan-q85-1165x1066.jpg

demnach lediglich annähernd korrekt. Die Grundform der Steine wurde aus einfachen Polygon Cubes erzeugt. Anschließend wurde jeder einzelne Stein, mittels eines „Rock Generator“ Skriptes, welches die planaren Oberflächenstrukturen aufbricht, erzeugt. Die verwendeten Texturen wurden aus einer Quelle für kostenlose Texturen aus dem Internet bezogen.

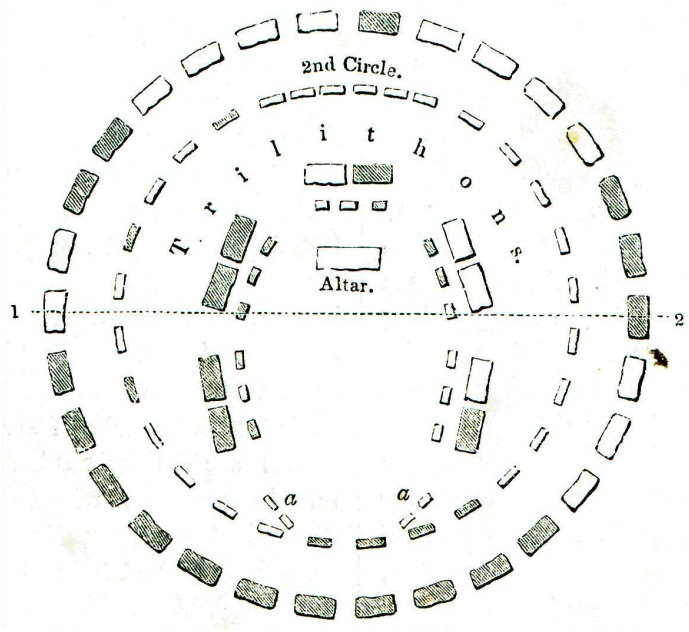


Bild 4.3: Vorlage für die Stonehenge Szene

4.2 Einbindung der Modelle in OpenGL

Bislang wurden die Modelle, die im Tanz-Projekt „Cam^2Dance“ eingebunden sind, in 3DStudioMax erstellt und gespeichert, und anschließend in Milkshape3D importiert. Die gespeicherten Milkshape Objekte wurden dann mittels eines Milkshape Loaders in das Tanzprojekt geladen.

Um den Umweg über Milkshape3D zu vermeiden, sollten die Modelle möglichst direkt von einem 3D-Modellierungstool in OpenGL importiert werden können. Dadurch dass

Milkshape im Vergleich zu Maya oder 3DStudioMax nur eingeschränkte Funktionen zum modellieren bietet und somit nicht den vollen Funktionsumfang von Maya oder 3DStudioMax bieten kann, kam dieses Programm zum Modellieren der Szenen nicht in Frage. Da Maya als leistungsstarkes, umfangreiches 3D-Modellierungstool gilt und ich bereits Erfahrungen im Umgang mit diesem Tool sammeln konnte, wurden die Szenen für das Tanz-Projekt in Maya modelliert. Zeitgleich zu dieser Studienarbeit wurde von einem weiteren Studenten der Computervisualistik ein Maya-Exporter sowie ein Maya-Importer geschrieben, die einen Teil seiner Studienarbeit darstellen. An dieser Stelle möchte ich nicht weiter auf die Funktion des Exporters bzw. Importers eingehen, möchte allerdings auf die Studienarbeit von Mirko Geissler[2] verweisen.

4.3 Toon-Shader

Wie in Kapitel 3.5.1 erläutert gilt es drei Komponenten zu implementieren, um einen Cartoon-Effekt zu erhalten. Zunächst soll auf die beiden Aspekte, die die Beleuchtung betreffen eingegangen werden. Zum einen war eine diffuse Beleuchtung, die durch sehr wenige Werte (2 bis 5) repräsentiert wird, gefordert, sowie spiegelnde Highlights, die von einer hellen Farbe mit hoher Intensität repräsentiert werden.

4.3.1 Vertex und Fragment Shader

Da die klassische Beleuchtung im Eye-Space stattfindet (siehe Kapitel 3.3) beinhaltet der erste Schritt, der im Vertex Shader ausgeführt wird, eine Umwandlung der Eckpunktnormalen und zur Berechnung benötigten Vektoren, wie dem Lightvector L und dem Viewvector V, in Eye-Space Koordinaten. Die Normalen werden dabei mit der inverstransponierten Modelview Matrix transformiert.

```
oNormal = normalize(mul(
            glstate.matrix.invtrans.modelview[0],
```

[2]Geissler, Mirko (2006). Entwicklung einer 3D Anwendung mit erweiterter optischer und haptischer Unterstützung. Universität Koblenz.

```
            float4(iNormal.xyz,0 ) ).xyz);
oViewVec = normalize(eyePosition - mul(
            glstate.matrix.modelview[0],
            iPosition).xyz);
oLightVec = normalize(glstate.light[0].position - mul(
            glstate.matrix.modelview[0],
            iPosition).xyz);
```

Um die Beleuchtung ansprechender zu gestalten, findet die Beleuchtung, und damit alle weiteren Schritte, im Fragment Shader statt. Bei einer per-Vertex Beleuchtung werden nur die Eckpunkte beleuchtet und die Farbwerte über die Fläche hinweg interpoliert. Bei einer per-Fragment Beleuchtung werden die Eckpunktnormalen über die Fläche hinweg interpoliert, was auch begründet warum diese an das Fragment Programm weitergereicht werden müssen, und anschließend die Beleuchtung für jedes Fragment berechnet wird.

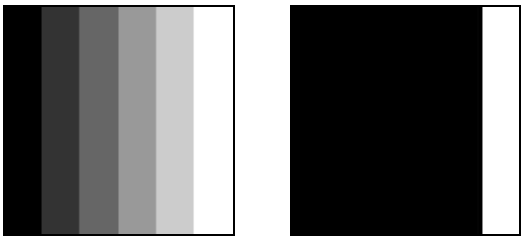
In Kapitel 3.4.2 wird das Beleuchtungsmodell nach Phong beschrieben, welches implementiert werden soll. Dabei wird erst der diffuse und anschließend der spiegelnde Lichtanteil berechnet. Damit Oberflächen, deren Normale vom Licht wegzeigt, dessen komponentenweise Multiplikation also negativ ist, nicht negativ beleuchtet werden, wird die Funktion `saturate` verwendet, die die Werte auf den Bereich `[0,1]` clampt. Bei der Berechnung des spiegelnden Anteils wird zusätzlich noch geprüft, ob der diffuse Anteil des Lichtes `= 0` ist. Ist dies der Fall, wird auch der spiegelnde Anteil auf `0` gesetzt.

```
float ndotl = dot(iNormal, iLightVec);
float diffuseLight = saturate(ndotl);
float3 R = normalize(2 * ndotl * iNormal - iLightVec);
float specularLight = pow(
        saturate(dot(R, iView)), shininess);
if (diffuseLight <= 0) specularLight = 0;
```

Im nächsten Schritt finden die zwei Textursampler Verwendung, die im Fragment Shader zu Beginn deklariert wurden.

```
uniform sampler1D diffuseRamp
uniform sampler1D specularRamp
```

Die 1D Texturen (siehe Abb. 4.4) fungieren als Look-up Tabellen, in denen der Helligkeitswert pro Pixel ausgelesen wird. Um Probleme zu vermeiden, die durch Texturen mit einer Auflösung ungleich einer zweier Potenz auftreten können, wurde eine Auflösung 2^x gewählt.



(a) diffus (b) spiegelnd

Bild 4.4: Look-up Texturen

Mittels der Funktion `tex1D(<Texturbezeichner>, <Texturkoordinaten>)` wird die Textur gesamplet. Der Rückgabewert der tex1D-Funktion wird zur diffusen und spiegelnden Beleuchtung der Objekte verwendet. Der bis dahin berechnete diffuse bzw. spiegelnde Wert wird somit mit dem korrespondierenden Wert von der Stufenfunktion ersetzt. Mit den neu bestimmten Werten wird letztendlich die finale Farbe des Fragments berechnet.

```
float diff = tex1D(diffuseRamp, diffuseLight).x;
float spec = tex1D(specularRamp, specularLight).x;
oColor = float4((Kd * diff + Ks * spec), 1.0);
```

Durch das Toon-Shading wird das Objekt nun nicht mehr kontinuierlich beleuchtet, sondern diskret. Auch die Highlights sind klar abgegrenzt und finden keinen weichen Übergang.

4.3.2 Hauptprogramm

Der letzte Punkt, der für einen Cartoon-Effekt relevant ist, betrifft das Umranden der Objekte. Die Objekte erhalten ihre schwarze Kontur an der Stelle im Programm, an welcher diese auch gezeichnet werden. Bei deaktiviertem Toon-Shading wird nur die Rückseite der Objekte im Wireframe Modus bei einer erhöhter Linienstärke gezeichnet. Dies bewirkt, dass die stark gezeichneten Linien quasi „überstehen“, was dann zu der gewünschten Kontur führt. Um zu erreichen, dass die gezeichneten Vertices schwarz sind, kommt ein simpler Fragment Shader zum Einsatz, der alle eingehenden Vertices schwarz zeichnet.

```
useToon::getInstance()->enableFragment();
useToon::getInstance()->bindFragmentBlack();
glLineWidth(5);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
< Modell zeichnen >
```

Anschließend wird das Toon-Shading aktiviert und die Objekte (Front- und Backfaces) werden im klassischen Fill-Modus gezeichnet. Damit allerdings nur das betreffende Modell den Toon-Look erhält, werden die Shader im Anschluss wieder deaktiviert. Teile einer Szene, die nicht mit dem Toon Shader manipuliert werden sollen, wie beispielsweise eine Skybox, sollten zu Beginn der `draw()` Methode gerendert werden, da der Shader erst im Anschluss aktiviert wird.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glDisable(GL_CULL_FACE);
glMatrixMode(GL_MODELVIEW);
glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
useToon::getInstance()->enableVertex();
useToon::getInstance()->bindVertexToon();
useToon::getInstance()->bindFragmentToon();
```

```
< Modell zeichnen >
useToon::getInstance()->disableVertex();
useToon::getInstance()->disableFragment();
```

4.4 Shadow Mapping Shader

Die Funktionsweise des Shadow Mapping Algorithmus wurde bereits im Kapitel 3.6 erläutert. Im Folgenden wird näher auf die Implementation und Lösung der Probleme, die mit Shadow Maps einhergehen, eingegangen.

4.4.1 Hauptprogramm

Zu Beginn wird die Position des Betrachters $P_B(x, y, z)$ in die Position der Lichtquelle $P_L(x, y, z)$ gesetzt. Die View Matrix wird gespeichert, da diese im Shader benötigt wird. Anschließend werden die Objekte der Szene, die Schatten werfen sollen, gezeichnet.

Nach dem Zeichnen der Objekte wird die Textur gebunden und die Tiefenwerte in die Shadow Map kopiert. Da die View Matrix der Kamera ebenfalls im Shader benötigt wird, wird auch diese gespeichert.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(light_pos[0], light_pos[1], light_pos[2],
          light_dir[0], light_dir[1], light_dir[2],
          0, 1, 0);
glGetFloatv(GL_MODELVIEW_MATRIX, (GLfloat*)lightViewMat);
< Objekte zeichnen >
glBindTexture(GL_TEXTURE_2D, sMTexture);
glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
                 0, 0, shadowmap_xres, shadowmap_yres, 0);
```

```
glGetFloatv(GL_MODELVIEW_MATRIX, (GLfloat*)cameraViewMat);
```

Als letzten Schritt wird der Shadow Mapping Shader aktiviert sowie die Uniforms für die Shader gesetzt. Die Szene wird erneut gezeichnet und die Shader im Anschluss wieder deaktiviert. Das Deaktivieren ist notwendig, falls Teile eine Szene vom Shading ausgeschlossen werden sollen. Diese werden dann zu Beginn der `draw()` Methode gezeichnet. Somit muss sichergestellt sein, dass bei Eintritt in diese Funktion kein Shader aktiv ist.

```
useShadow::getInstance()->enableFragment();
useShadow::getInstance()->enableVertex();
useShadow::getInstance()->bindFragmentShadow();
useShadow::getInstance()->bindVertexShadow();
useShadow::getInstance()->setShadowUniforms();
< Szene zeichnen >
useShadow::getInstance()->disableFragment();
useShadow::getInstance()->disableVertex();
```

4.4.2 Vertex und Fragment Shader

Im Vertex Shader finden vorbereitende Berechnungen statt. Die Vertices, die im World Space vorliegen, werden mit der View Matrix des Lichtes transformiert, so dass diese anschließend in Clip Coordinates vorliegen. Darüber hinaus werden die Vertices auch mit der View Matrix der Kamera transformiert und ebenfalls in Clip Coordinates transformiert. Anschließend wird noch der Lightvector berechnet und die Normale transformiert. Die View Matrix des Lichtes und der Kamera wurden dem Vertex Shader zuvor als Uniforms übergeben.

```
float4 worldPos = mul(
         glstate.matrix.modelview[0], iPosition);
oTexCoord = mul(glstate.matrix.projection,
                mul(lightViewMatrix, worldPos));
oPosition = mul(glstate.matrix.projection,
```

```
                mul(cameraViewMatrix, worldPos));
oNormal = normalize(mul(
            glstate.matrix.invtrans.modelview[0],
            float4(iNormal.xyz,0)).xyz);
oLightVec = normalize(glstate.light[0].position -
            worldPos.xyz);
```

Die eigentliche Berechnung des Schattens erfolg im Fragment Shader. Über die Funktion „lookup“ wird ermittelt ob ein Pixel im Schatten liegt oder nicht. Die lookup Funktion berechnet für jedes vom Betrachter aus sichtbare Pixel eine korrespondierende Koordinate aus Sicht der Lichtquelle. Zuerst werden die Texturkoordinaten von $(-1..1)$ auf $(1..0)$ gemappt und durch die perspektivische Division normalisiert. Der Tiefenwert z wird noch durch ein angemessenen Offset korrigiert, um das Problem der fehlerhaften Selbstverschattung zu beheben. Anschließend kann der zugehörige Wert $T_{SM}(s,t)$ in der Shadow Map ausgelesen und mit dem Wert $T_{L,Pix}(x,y)$, aus Sicht der Lichtquelle, verglichen werden.

```
float shadow = lookup(shadowMapTexture, iTexCoord).x{

float2 tc = (iTexCoord.xy / iTexCoord.w) * 0.5 + 0.5;
float test = ((iTexCoord.z / iTexCoord.w) *
               0.5 + 0.5) * 0.99985;
float shadow;
float zwert = tex2D(shadowMapTexture, tc).x;
if (test < zwert) shadow = 1;
else shadow = 0;
return shadow;
}
```

Bei einem shadow von 0, liegt der getestete Pixel im Schatten, andernfalls wird dieser beleuchtet. Bevor die Farbe für das entsprechende Fragment ausgegeben werden kann, muss das Skalarprodukt von `n` und `l` berechnet werden. Mit `saturate` wird erreicht, dass Rückseiten von Objekten, also Flächen deren Normalen vom Licht wegzeigen, nicht beleuchtet werden. Auf die Berechnung der Beleuchtung wurde bereits in Kapitel 3.4 eingegangen. In diesem Fall muss noch der Faktor `shadow` hinzugefügt werden, der für eine schwarze Ausgabefarbe sorgt, wenn das Fragment im Schatten liegt. Andernfalls findet eine reguläre diffuse Beleuchtung statt.

```
float ndotl = saturate(dot(iLightVec, iNormal));
oColor = float4(tex2D(diffuseTexture, iRealTC).xyz *
                shadow * ndotl * lightColor +
                globalAmbient, 1);
```

Die Bestimmung des Schatten erfolgt hier auf einfache Weise und führt letztlich zu harten Schattenkanten. Die Aliasing Effekte entstehen dadurch, dass die Texturkoordinate nur selten auf genau ein Pixel in der Shadow Map trifft. Es können dabei 2-4 Pixel getroffen werden. Ohne weitere Filterverfahren würde dem Pixel der Wert des oberen linken Pixels der Shadow Map zugewiesen, was zu einem Fehler führt, der sich im Aliasing bemerkbar macht. Bereits durch ein einfaches 3x3 Box Filtering kann dieser Effekt abgeschwächt werden. Im folgenden Kapitel 4.4.3 werden einige Filter vorgestellt.

4.4.3 Aliasing Effekte an den Schattenkanten

Ein Problem des Shadow Mapping Algorithmus ist das der pixeligen und harten Schattenkanten. Bislang wird für jedes Pixel entschieden ob es im Schatten liegt oder nicht. Für den Fall, dass es im Schatten liegt, wird dem Pixel der Farbwert `(0, 0, 0)` zugewiesen. Um den Aliasing Effekt zu reduzieren, muss die Nachbarschaft in die Bestimmung mit einbezogen werden. Darüber hinaus soll anhand dessen ermittelt werden, welchen Grauwert der Pixel zugewiesen bekommt, falls sich ein Teil der Nachbarschaft ebenfalls im Schatten befindet.

Bilineare Interpolation Bei der bilinearen Interpolation werden die 4 (maximal) möglichen Pixel der Shadow Map getestet, die bei der Berechnung getroffen wurden. Wie in Abbildung 4.5[3] zu sehen ist, sind die Nachkommastellen wichtig um die finale Farbe des Pixels zu bestimmen. Dementsprechend muss die lookup Funktion, die ohne Filterung verwendet wurde, abgeändert werden.

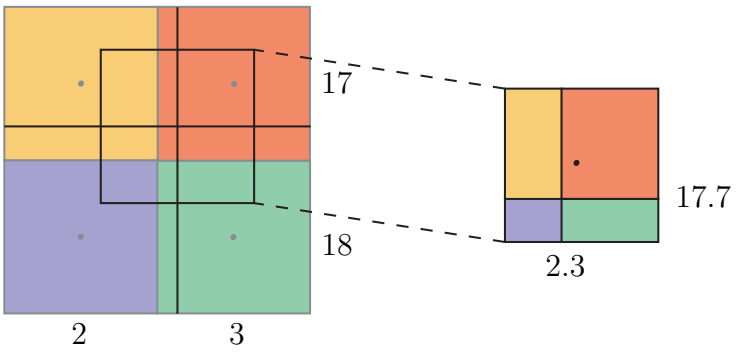


Bild 4.5: Bilineare Interpolation

Die Berechnung der Texturkoordinaten und des Tiefenwertes findet wie oben erwähnt statt. Darüber hinaus werden die Nachkommawerte der Texturkoordinate (über die Funktion `frac()`) bestimmt und gespeichert. Um die Werte von `fracs` auf die der Koordinaten `tc` anzupassen, erfolgt eine Division von 1024 (Größe der Shadow Map), um ein Mapping auf Werte zwischen 0 und 1 zu gewährleisten. Der gesuchte Pixel ist um den Wert von `fracs` verschoben und kann durch Subtraktion dessen bestimmt werden. Anschließend wird für jedes der 4 Pixel geprüft ob dieses im Schatten liegt oder nicht.

```
float2 fracs = float2(frac(tc.x * 1024), frac(tc.y * 1024));
tc.x = tc.x - fracs.x / 1024;
tc.y = tc.y - fracs.y / 1024;

float zwert = tex2D(shadowMapTexture, tc).x;
if (test < zwert) vals[0] = 1;
else vals[0] = 0;
```

[3] vgl. http://www.3dcenter.de/artikel/grafikfilter/index4.php

```
zwert = tex2D(shadowMapTexture, tc +
              float2(onePix, 0)).x;
if (test < zwert) vals[1] = 1;
else vals[1] = 0;

zwert = tex2D(shadowMapTexture, tc +
              float2(0, onePix)).x;
if (test < zwert) vals[2] = 1;
else vals[2] = 0;

zwert = tex2D(shadowMapTexture, tc +
              float2(onePix, onePix)).x;
if (test < zwert) vals[3] = 1;
else vals[3] = 0;
```

Die berechneten Werte für `i = (0..3)` von `vals[i]` werden nun bilinear interpoliert, um einen korrekten Grauwert abhängig von der Umgebung zu bestimmen.

```
float shadow = (vals[0] * (1 - fracs.x) +
                vals[1] * fracs.x) * (1 - fracs.y) +
                fracs.y * (vals[2] * (1 - fracs.x) +
                vals[3] * fracs.x);
```

Die Kanten des Schatten werden bei Nutzung dieser lookup Funktion nicht mehr schwarz dargestellt, sondern in Graustufen, womit der Aliasing Effekt abgeschwächt wird. Für bessere Ergebnisse sollte eine größere Nachbarschaft betrachtet werden. Eine Alternative zum bilinearen filtering stellt das Nutzen einer Gauß Maske dar.

Gauß Filter Der Gauß Filter ist ein so genannter Tiefpassfilter, der bei Anwendung harte Kanten glättet. Für den vorliegenden Fall bedeutet dies, dass die Schattenkanten weniger hart dargestellt werden. Es wird für jeden Pixel die 8er Nachbarschaft, sowie das

betrachtete Pixel selbst, mit der Gauß Maske gewichtet. Diese Maske besteht aus einer 3x3 Matrix, welche nach der Gaußschen Glockenkurve konstruiert worden ist.

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

Die gewichteten Pixel werden addiert und es wird anschließend der Durchschnitt gebildet. Das zentrale Pixel wird durch diesen berechneten Wert ersetzt, und die Kanten somit geglättet.
Für die Anwendung im Shader bedeutet dies, dass die Tiefenwerte der 8er Umgebung aus der Shadow Map Textur ausgelesen werden und diese dann mit dem Tiefenwert des zu testenden Pixel verglichen werden. Für den Fall, dass der Tiefenwert in der Textur größer ist als der Wert des zu testenden Pixels, wird der entsprechende Wert der Gaußmaske für die vorliegende Position einer Zählervariable aufaddiert.
Für den Fall, dass alle Pixel der Umgebung im Schatten liegen ist `count = 0` und somit erhält auch der Schatten den Wert `0`. Andernfalls berechnet sich die Farbe des Pixel aus dem Durchschnitt. Der Schatten ist damit an dieser konkreten Position nicht immer schwarz, sondern kann in einer beliebigen Abstufung eines Grautons vorliegen.

```
zwert = tex2D(shadowMapTexture, tc +
              float2(-onePix, -onePix)).x;
if (test < zwert) count+=1;
zwert = tex2D(shadowMapTexture, tc +
              float2(0, -onePix)).x;
if (test < zwert) count+=2;
zwert = tex2D(shadowMapTexture, tc +
              float2(onePix, -onePix)).x;
if (test < zwert) count+=1;
```

```
zwert = tex2D(shadowMapTexture, tc +
              float2(-onePix, 0)).x;
if (test < zwert) count+=2;
zwertCenter = tex2D(shadowMapTexture, tc);
if (test < zwert) count+=4;
zwert = tex2D(shadowMapTexture, tc +
              float2(onePix, 0)).x;
if (test < zwert) count+=2;



zwert = tex2D(shadowMapTexture, tc +
              float2(-onePix, onePix)).x;
if (test < zwert) count+=1;
zwert = tex2D(shadowMapTexture, tc +
              float2(0, onePix)).x;
if (test < zwert) count+=2;
zwert = tex2D(shadowMapTexture, tc +
              float2(onePix, onePix)).x;
if (test < zwert) count+=1;

 float shadow;
if (count == 0.0) shadow = 0.0;
else shadow = count / 16.0;
```

Um die Effekte der einzelnen Filter besser beurteilen zu können, ist der gleiche Ausschnitt der Stonehenge Szene in Abbildung 4.6 dargestellt. Der Aliasing Effekt kommt dabei im ersten Teilbild klar zum Ausdruck und es wird deutlich, warum eine Verminderung dieses Effektes zwingend notwendig ist.

(a) Kein Filter

(b) Bilineare Interpolation

(c) Gauß Filter

Bild 4.6: Aliasing an den Schattenkanten

Kapitel 5

Ergebnisse, Ausblick und Fazit

Im ersten Teil der Arbeit sind die Modelle der Universität Koblenz sowie von Stonehenge entstanden. Die folgenden zwei Bilder (siehe Abb. 5.1 und 5.2), die mittels der Renderer „Maya Software“ bzw. „Maya Hardware“ gerendert wurden, zeigen die modellierten Szenen in vollem Umfang.

Um die Objekte einer jeden Szene in das Projekt „Cam^2Dance“ zu importieren, mussten die Objekte, wie in Kapitel 4.2 beschrieben, zuerst mit Hilfe eines Maya Exporters exportiert werden, bevor diese geladen werden konnten. Dies führte vor allem bei der Stonehenge Szene mit über 60.000 Vertices zu Problemen, da die Größe der Datei beim Vorgang des Exports fast verfünffacht wurde (2.4mb –> 11.3mb). Möglicherweise sind Schwächen des Exporters die Ursache, der eventuell mit einer großen Anzahl an Vertices (noch) nicht effizient genug umgehen kann, was allerdings kein endgültiger Aspekt ist, da sich der Exporter sowie Importer zu der Zeit des Einsatzes noch in der Entwicklung befanden. Die Größe der Uni Szene mit über 8.000 Vertices dagegen wurde beim Export um ein vielfaches verkleinert (8.37mb –> 1.09mb).

Eine Möglichkeit die Größe der Datei zu minimieren wäre zum einen eine komplett neue Modellierung, die mit weniger Vertices auskommt, oder aber eine alternative Möglichkeit um die in Maya erstellten Objekte in das „Cam^2Dance“ Projekt zu laden. Möglicherweise tritt das Problem in der finalen Version des Exporters auch nicht mehr auf.

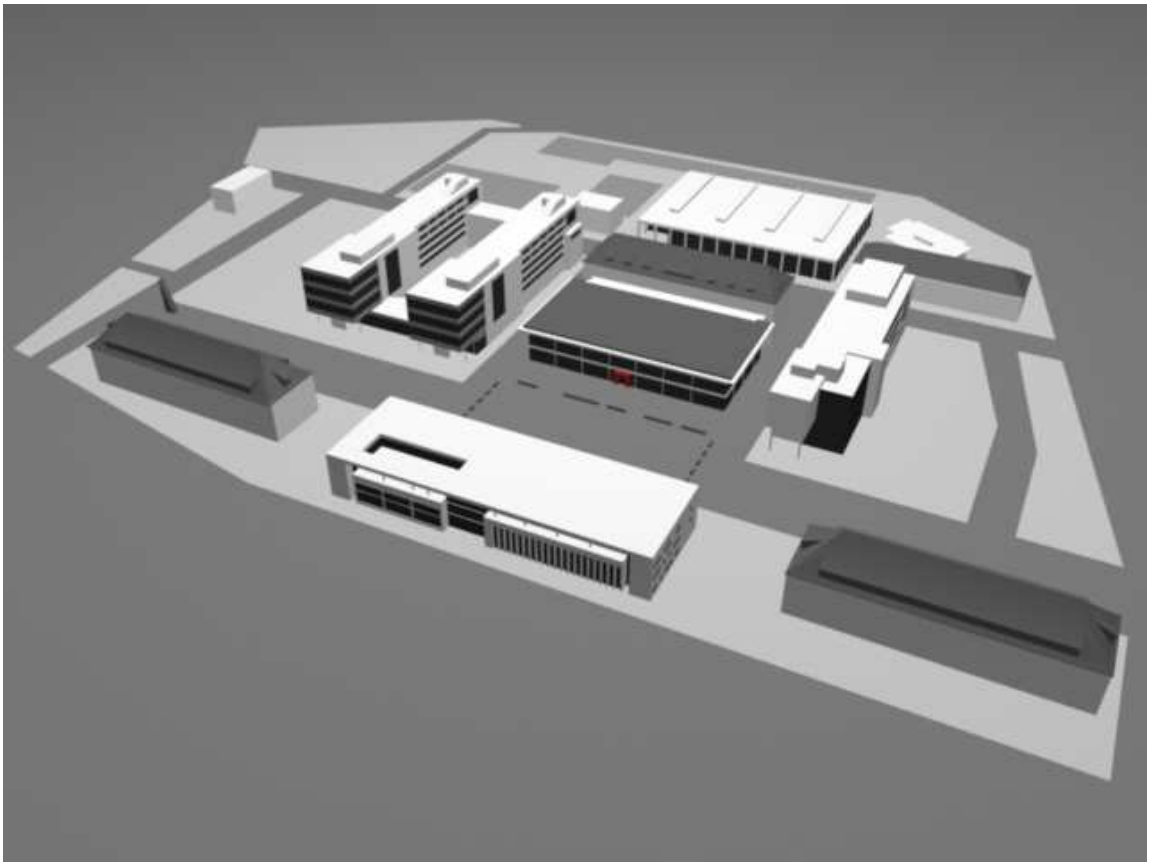


Bild 5.1: 3D Modell des Campus (gerendert in Maya)

Bild 5.2: 3D Modell von Stonehenge (gerendert in Maya)

Im zweiten Teil der Arbeit, sollten nun die Shader implementiert werden. Bevor allerdings mit der Implementierung begonnen werden konnte, galt es zuerst Linux, als zweites Betriebssystem neben Windows, zu installieren, da das Tanzprojekt bislang nur unter Linux lauffähig ist und das Testen somit auch unter dem Betriebssystem stattfinden musste. Das Nachinstallieren der fehlenden, aber benötigten, Bibliotheken (siehe Anhang A) stellte, für mich als Windows-User, eine erste Herausforderung dar. Darüber hinaus ergaben sich Probleme mit der genutzten Graphikkarte, da diese aus Altersgründen, keine Programmierung von Fragment Shadern unterstützte. Erst nachdem diese Probleme gelöst werden konnten, konnte mit der eigentlichen Programmierung begonnen werden.

Nach einer erforderlichen Einarbeitungszeit in OpenCG und Programmierung von ersten Testprogrammen entstanden nach Lösung diverser Fehler die beiden Shader. Die ursprüngliche Idee der komplexen Modelle konnte allerdings nicht in vollem Umfang verwirklicht werden. Geplant war es zum einen das Tanzspiel durch aufwendigere Szenen interessanter und vielseitiger zu gestalten, was mit den modellierten Szenen erreicht worden ist. Das weitere Ziel, die Effekte, die durch die Shader implementiert werden sollten durch die komplexen Szenen besser zum Ausdruck kommen zu lassen, wird mit dem Toon Effekt in der Campus Szene nur mäßig erreicht, da vorwiegend große eckige Flächen vorhanden sind. Bei runden Objekten würde der Effekt (siehe Abb. 5.7), was mir im Laufe der Implementierung klar geworden ist, deutlicher sichtbar sein. Das gerenderte Modell des Campus kann in Abbildung 5.3 mit der toon-shaded Szene im Tanz-Projekt 5.4 verglichen werden.

Der Shadow Maps Shader hingegen erzeugt im Zusammenhang mit der erzeugten Himmeltextur eine eindrucksvolle Stimmung, die gut zur Stonehenge Szene passt (siehe Abb. 5.6). Das gerenderte Maya Modell kann in Abbildung 5.5 betrachtet werden. Der Gauß Filter, der eingesetzt wurde um das Aliasing an den Schattenkanten zu reduzieren, führt durch reduzierte Härte der Schattenkanten zu einem weicheren, ästhetischeren Bild. Zur weiteren Verbesserung der Schatten, könnte ein weiterer Shader geschrieben werden, der die Schattenkanten weichzeichnet. Darüber hinaus würden Schatten, die nicht komplett schwarz sind, zu einem realistischeren Ergebnis führen, was allerdings ebenfalls einen weiteren Shader bzw. eine Veränderung des derzeit vorhandenen erfordert.

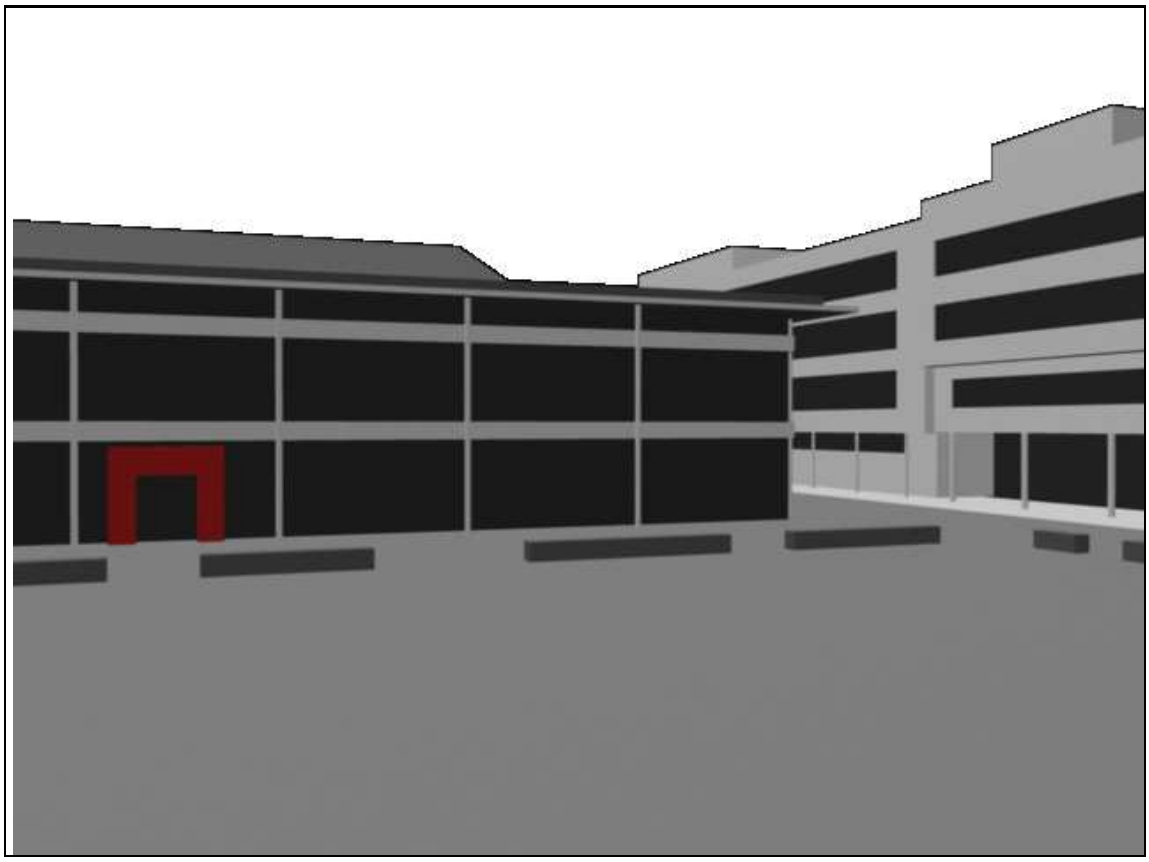


Bild 5.3: 3D Modell des Campus (gerendert in Maya)

Bild 5.4: Toon-Shaded Szene im Tanzprojekt

Bild 5.5: 3D Modell von Stonehenge (gerendert in Maya)

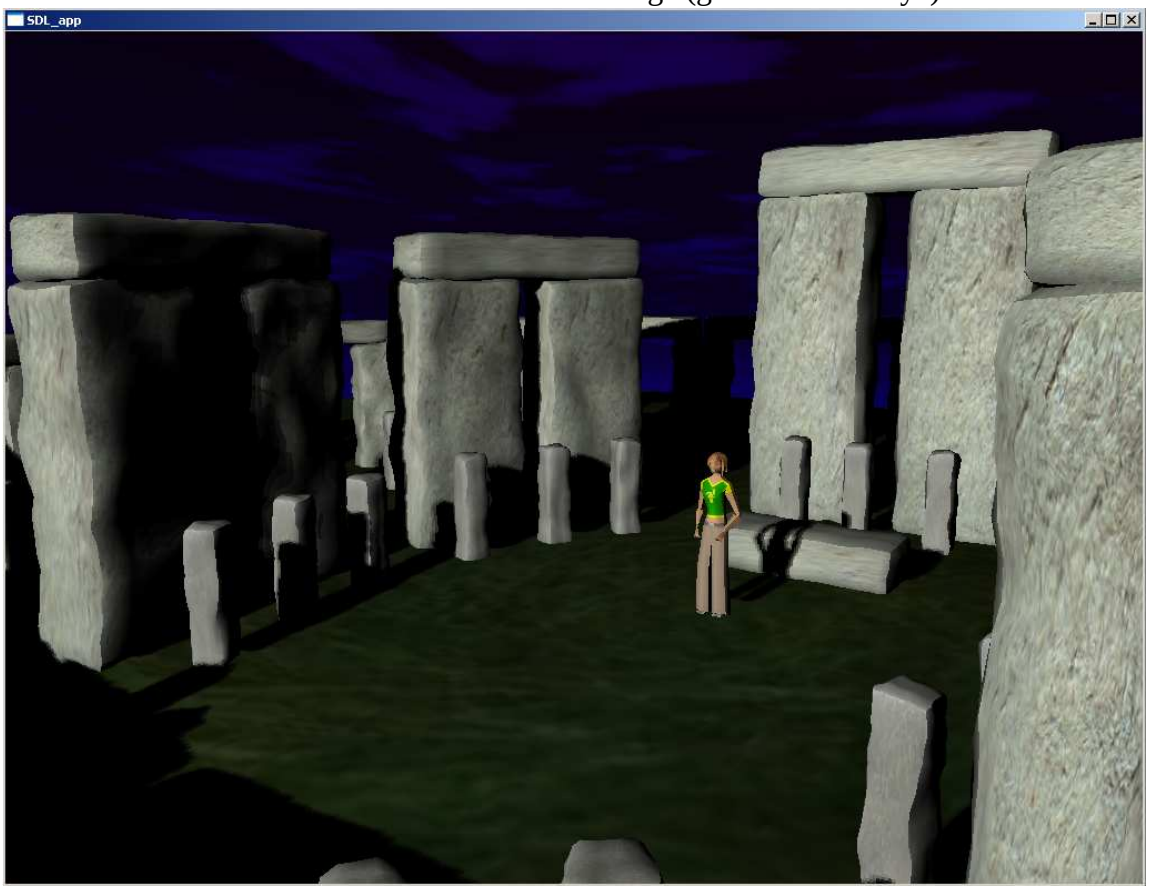
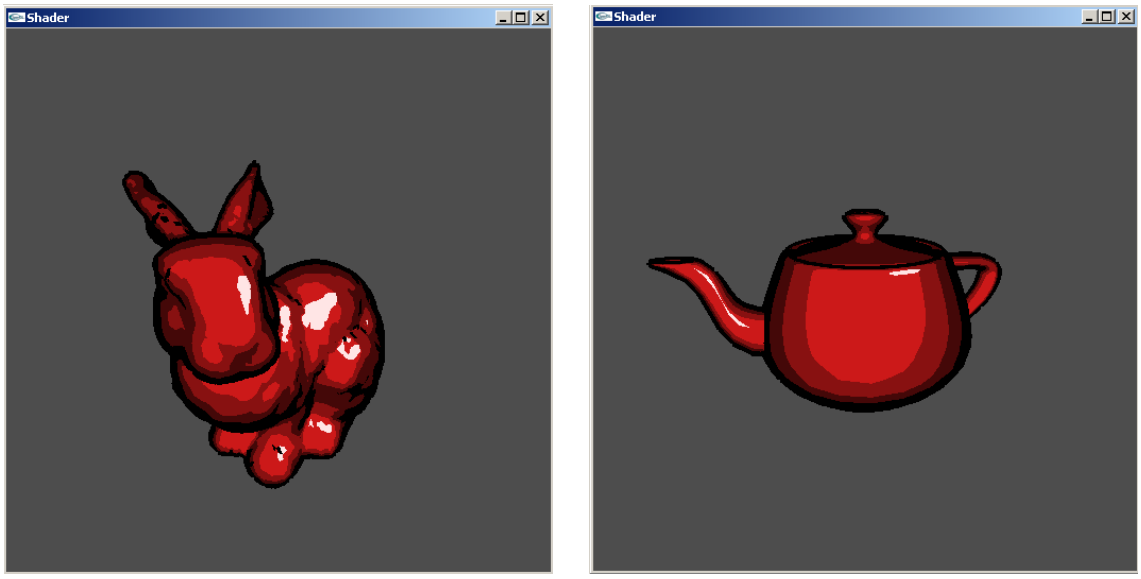


Bild 5.6: Shadow Maps-Shaded Szene im Tanzprojekt



(a) Stanford Bunny (b) Teapot

Bild 5.7: Weitere Toon-Shaded Objekte

Anhang A

Installation und Konfiguration

Da das Tanzspiel Cam2Dance unter Linux entwickelt wurde und darüber hinaus bereits existierende Bibliotheken wie ARToolKit, Lapack++ oder SDL Verwendung gefunden haben, galt es, bevor mit der eigentlichen Arbeit begonnen werden konnte, zum einen Linux als zweites Betriebssystem zu installieren und zum anderen alle nötigen Bibliotheken und Pakete nachzuinstallieren sowie nötige Konfigurationen vorzunehmen. Der folgende Abschnitt dient somit auch als Installationsanleitung für das Tanzprojekt.

A.1 Linux

Setzen der Umgebungsvariablen:

- `export kate .bashrc`
- `export ARTOOLKIT = /home/<name>/<Pfad zu Cam2Dance>/ARToolKit`
- `export VIRTANZDIR = /home/<name>/<Pfad zu Cam2Dance>/1.0`
- `export ARTKP = /home/<name>/<Pfad zu Cam2Dance>/ARToolKitPlus`

Linken:

- `sudo ln -s /home/<name>/<Pfad zu Cam2Dance> /ARToolKitPlus/lib/libARToolKitPlus.so.2 /usr/lib/libARToolKitPlus.so.2`
- `sudo ln -s /usr/local/lib/liblapackcpp.so.1 /usr/lib/liblapackcpp.so.1`

A.2 Pakete und Bibliotheken

Folgende Pakete / Bibliotheken müssen unter Linux zusätzlich installiert werden:

- `sdl, sdl-dev`
  http://www.libsdl.org/download-1.2.php
- `sdl_mixer, sdl_mixer-dev`
  http://www.libsdl.org/projects/SDL_mixer/
- `sdl_image, SDL_image-devel`
  http://www.libsdl.org/projects/SDL_image/
- `libraw1394, libraw1394-devel`
- `libdc1394, libdc1394-devel`
- `blas, blas-dev`
- `lapack++`
  http://sourceforge.net/projects/lapackpp
- `ARToolKit`
  http://sourceforge.net/projects/artoolkit

- `ARToolKitPlus`
  http://studierstube.icg.tu-graz.ac.at/handheld_ar/artoolkitplus.php

A.3 Projekt starten

- Wechseln in den Ordner `60_tools`
  - `qmake`
  - `make`
- Wechseln in den Ordner `30_prog/VTTrack2Dance/src`
  - `make`
- Wechseln in den Ordner `30_prog/TrackCamMusicMerge`
  - `qmake`
  - `make` -> erstellt „game“
- Spiel starten mit
  - `./game`, wenn Kameras angeschlossen sind
  - `./game solo`, wenn keine Kameras angeschlossen sind

Anhang B

Quellcode

Siehe CD.

Literaturverzeichnis

[Ebs05] R. Ebser. Die programmierbare hardware-pipeline und ihre verwendung für glaubhafte künstliche charaktere. Master's thesis, Fachhochschule Stuttgart, 2005.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[FK04] Randima Fernando and Mark J. Kilgard. *GPU Gems - Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004.

[Kil00] M. J. Kilgard. Shadow mapping with today's opengl hardware. 2000.

[MÖ4a] Stefan Müller. Skript zur vorlesung computergraphik i im sommersemester 2004, 2004.

[MÖ4b] Stefan Müller. Skript zur vorlesung computergraphik ii im wintersemester 2004, 2004.

[Nuy03] Tom Nuydens. Phong's model, 2003.

[Ros04] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2004.

[Sch03] Sebastian Schäfer. Ausarbeitung zum thema schatten. 2003.