UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Non-Photorealistic Rendering mit Hilfe der GPU

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Stefan Müller

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im März 2007

Erklärung

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Ziel 1
1.3 Übersicht 2

2 Grundlagen **3**
2.1 Non-Photorealistic Rendering 3
2.1.1 Cool-to-Warm Shading 3
2.1.2 Toon Shading 5
2.1.3 Silhouetten 6
2.2 Programmierung der Graphics Processing Unit (GPU) 8
2.2.1 Fixed-Function-Pipeline 8
2.2.2 Programmierbare Graphik-Pipeline 9
2.2.3 NVIDIA Cg 10
2.2.4 NVIDIA CgFX 11
2.3 Microsoft Direct3D und DXUT 12

3 Implementierung **13**
3.1 Klassendiagramm 13
3.2 Shading-Modelle 15
3.2.1 Cool-to-Warm Shading 16
3.2.2 Toon Shading 17
3.2.3 Phong Shading 19
3.3 Silhouettenalgorithmen 19
3.3.1 Silhouetten durch Manipulation der Geometrie . . . 21
3.3.2 Bildbasierte Erkennung von Silhouetten 21
3.4 Graphische Benutzerschnittstelle 24
3.5 Maussteuerung 25

4 Ergebnisse **27**
4.1 Bilder 27
4.2 Performanz 32

5 Fazit und Ausblick **35**

Abbildungsverzeichnis

1 Vektoren des Phong-Beleuchtungsmodells 3
2 Vergleich von (a) Phong Shading und (b) Cool-to-Warm Shading ohne und (c) mit Silhouetten [7] 4
3 1D-Textur und der Graph der von ihr kodierten Funktion . . 5
4 Umriss durch Verschiebung von Eckpunkt-Normalen 7
5 Renderpasses des bildbasierten Silhouettenalgorithmus . . . 8
6 Die Fixed-Function-Graphik-Pipeline [4] 9
7 Die programmierbare Graphik-Pipeline [4] 10
8 Cg und CgFX im Kontext einer 3D-Anwendung [4] 11
9 UML-Klassendiagramm 13
10 Zustandsautomat 25
11 Vergleich von Toon Shading ohne und mit Weichzeichnung 26
12 Die graphische Benutzerschnittstelle 27
13 3D-Modell einer Schraube. (a) und (d) Phong Shading. (b) und (e) Cool-to-Warm Shading ohne Silhouette. (c) und (f) Cool-to-Warm Shading mit bildbasierter Silhouette. 28
14 Zeichnung von Bart Simpson als Vorlage für ein 3D-Modell 29
15 „Bart Simpson“-Modell im Vergleich 30
16 „Chamäleon“-Modell im Vergleich 30
17 Bildbasierte Silhouette des „Killeroo“-Modells 31
18 Silhouettenalgorithmen im Vergleich an Modell „Dwarf“ . . 32
19 Darstellungsgeschwindigkeit der Shader 33

1 Einleitung

1.1 Motivation

Die rasante Entwicklung von Graphikhardware wird von dem Wunsch vorangetrieben virtuelle Szenarien realistischer erscheinen zu lassen. Praktisch jeder neue PC der heute verkauft wird ist mit einer programmierbaren Graphikkarte ausgerüstet, die in der Lage ist sämtliche graphikrelevanten Berechnungen wesentlich schneller auszuführen als die CPU[1]. Dem entlasteten Hauptprozessor bleiben damit mehr Ressourcen für andere Operationen, was wiederum rechenintensivere Anwendungen ermöglicht.

Das Ziel der photorealistischen Computergraphik vor Augen, wird oft übersehen, dass eine Darstellung, die nicht die Realität widerspiegelt, auch Vorteile bringen kann. Künstlerische Bilder besitzen eine eigene Ästhetik und sind meist besser geeignet uns in besondere Stimmungen zu versetzen als eine Photographie. Bilder, die von der Wirklichkeit abstrahieren und in denen wichtige Details hervorgehoben werden, beispielsweise durch einen Umriss, können vom menschlichen Gehirn schneller verarbeitet werden als eine komplexe Nachbildung der Realität. Die realistische Beleuchtung eines Modells ist nicht zwangsläufig die beste Art dem Betrachter die Form des Objekts unmittelbar bewusst zu machen.

1.2 Ziel

Im Rahmen dieser Studienarbeit sollen die Möglichkeiten programmierbarer Graphikhardware genutzt werden um Verfahren zu implementieren, die dem Non-Photorealistic Rendering zugeordnet sind. Es wird ein Programm vorgestellt, das in der Lage ist 3D-Modelle zu laden und diese mit unterschiedlichen Shading-Modellen in Echtzeit darzustellen. Über eine graphische Benutzerschnittstelle wird es ermöglicht das Modell und die Art der Darstellung jederzeit zu ändern und somit Vergleiche anzustellen. Der Benutzer erhält zudem die Möglichkeit sämtliche variablen Parameter der implementierten Verfahren zu regulieren.

Die Implementierung des vorzustellenden Programms erfolgte mit C++ unter Microsoft Windows. Mit Hilfe der Shader-Hochsprache Cg von NVIDIA wurde die Graphikhardware programmiert. Als Graphikbibliothek kam Direct3D zum Einsatz, das Teil der Programmierschnittstelle DirectX von Microsoft ist. Das DirectX Utility Toolkit (DXUT) diente zur Realisierung der Benutzerschnittstelle. Eine umfangreiche HTML-Dokumentation des Quellcodes wurde mit dem Programm Doxygen erstellt.

[1]Central Processing Unit

1.3 Übersicht

Zunächst soll das Thema Non-Photorealistic Rendering vorgestellt werden, bevor auf die Theorie der implementierten Verfahren eingegangen wird. Im Vergleich zur klassischen Graphik-Pipeline wird anschließend auf die Pipeline-Stufen eingegangen, die sich bei moderner Graphikhardware programmieren lassen. Mit Cg wird eine Shader-Hochsprache präsentiert, die zur Programmierung von Graphikkarten eingesetzt wird. Danach wird die Graphikbibliothek Direct3D und das Framework DXUT vorgestellt.

Vom softwaretechnischen Entwurf ausgehend, wird die Implementierung der einzelnen Verfahren des Non-Photorealistic Rendering dargestellt. Anschließend wird die Planung und Realisierung der Benutzerschnittstelle erläutert.

Die erzielten Ergebnisse werden anhand von Bildschirmphotos aufgezeigt und es wird kurz auf die Darstellungsgeschwindigkeit eingegangen.

Abschließend sollen sinnvolle Erweiterungen des Programms und interessante Verfahren, die nicht implementiert wurden, erläutert werden.

2 Grundlagen

2.1 Non-Photorealistic Rendering

Non-Photorealistic Rendering (NPR) bezeichnet eine Disziplin der Computergraphik, die sich mit der Erzeugung von meist künstlerischen Bildern beschäftigt, die nicht photorealistisch aussehen. Diese bieten gegenüber Bildern, welche die Realität wiedergeben, einige Vorteile.

Nick Halper [8] untersucht unter dem Schlagwort „Supportive Presentation“ beispielsweise die Vorteile, die sich durch den Einsatz von NPR in Computerspielen für die Wahrnehmung des Spielers ergeben.

In den letzten Jahren wurden wiederholt Techniken die dem NPR zuzuordnen sind, gezielt eingesetzt, um durch optische Innovationen die Verkaufszahlen von Computer- und Videospielen zu steigern. Allerdings fanden sich darunter auch einige Titel, die vom Non-Photorealistic Rendering profitierten, indem sie ihre eigene Ästhetik schufen.

Gooch et al. [7] sehen in der Verwendung von Non-Photorealistic Rendering zudem Vorteile für die Visualisierung im medizinischen und industriellen Bereich.

Im Folgenden werden drei Aspekte des Non-Photorealistic Rendering genauer beleuchtet: Cool-to-Warm Shading, Toon Shading und die Darstellung von Silhouetten.

2.1.1 Cool-to-Warm Shading

Der Mensch erkennt die Form eines Objektes daran, wie es beleuchtet wird. In Abhängigkeit von der Position der Lichtquelle zum Objekt gibt es helle und dunkle Stellen und, je nach Materialeigenschaft, Glanzlichter in Bereichen in denen sich die Lichtquelle spiegelt. Diese Eigenschaften werden in der Computergraphik von dem Phong-Beleuchtungsmodell simuliert. Die Farbe c eines Punktes der Objektoberfläche berechnet sich danach aus der Oberflächennormalen $\vec{n}$, dem Lichtvektor $\vec{l}$, und dem Einheitsvektor $\vec{h}$, der zwischen $\vec{l}$ und dem Sichtstrahl liegt [14]. Abbildung 1 zeigt die Vektoren

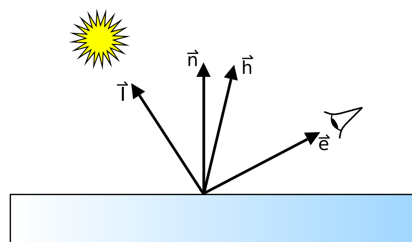
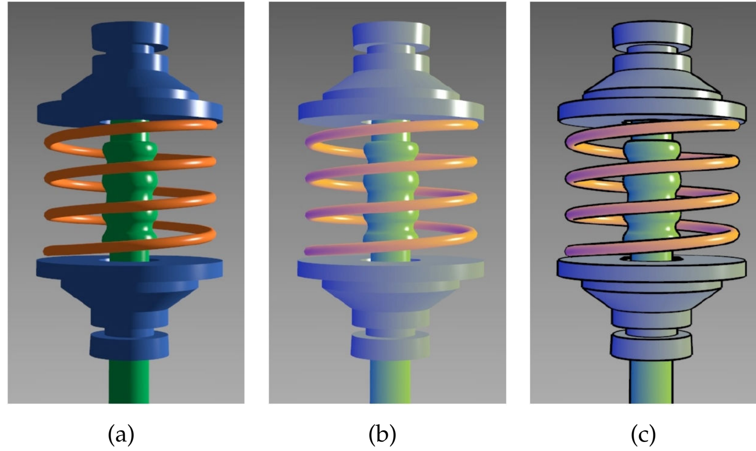


Abbildung 1: Vektoren des Phong-Beleuchtungsmodells



(a) (b) (c)

Abbildung 2: Vergleich von (a) Phong Shading und (b) Cool-to-Warm Shading ohne und (c) mit Silhouetten [7]

des Phong-Beleuchtungsmodells. Die Größe des Glanzlichts wird von dem Exponenten p bestimmt. Mit der diffusen Farbe c_r, der ambienten Farbe c_a, der Lichtintensität c_l und der Farbe der Glanzlichter c_p sind zusätzlich vier RGB-Tripel Teil der Gleichung:

$$c = c_r(c_a + c_l \max(0, \vec{n} \cdot \vec{l})) + c_l c_p(\vec{h} \cdot \vec{n})^p \tag{1}$$

Gooch et al. [7] stellen dagegen mit dem sogenannten Cool-to-Warm Shading ein Beleuchtungsmodell vor, das sich an dem Stil technischer Illustrationen, wie man sie beispielsweise in Betriebsanleitungen findet, orientiert. An Stelle von Helligkeitsunterschieden zwischen Flächen, deren Normalen in entgegengesetzte Richtungen zeigen, wird hier ein Übergang zwischen kalten und warmen Farben eingesetzt. Auf diese Weise geht deutlich weniger Information in Bereichen verloren, die ansonsten besonders dunkel oder hell wären. Vor allem in Kombination mit schwarzen Linien und weißen Highlights, die sich deutlich von dem niedrigen Dynamikbereich des Cool-to-Warm Shading abheben, erhält der Betrachter einen besseren Eindruck von der Form des dargestellten Objekts.

Einen Vergleich des klassischen Phong-Beleuchtungsmodells mit dem Cool-to-Warm-Beleuchtungsmodell von Gooch et al. zeigt Abbildung 2.

Das Skalarprodukt von Lichtvektor $\vec{l}$ und Normale $\vec{n}$ wird beim Cool-to-Warm Shading eingesetzt, um zwischen einer kalten Farbe c_{cool} und einer warmen Farbe c_{warm} zu interpolieren:

$$c = \left(\frac{1 + \vec{l} \cdot \vec{n}}{2}\right) c_{cool} + 1 - \left(\frac{1 + \vec{l} \cdot \vec{n}}{2}\right) c_{warm} \tag{2}$$

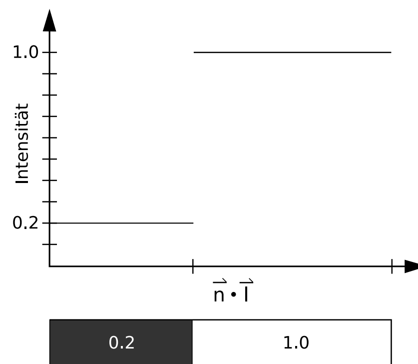


Abbildung 3: 1D-Textur und der Graph der von ihr kodierten Funktion

Die beiden Farben c_{cool} und c_{warm} berechnen sich aus einem Blau $c_{blue} = (0, 0, b)$ mit $b \in [0, 1]$ und einem Gelb $c_{yellow} = (y, y, 0)$ mit $y \in [0, 1]$ und der mit α, beziehungsweise β, gewichteten diffusen Objektfarbe c_r:

$$\begin{aligned} c_{cool} &= c_{blue} + \alpha c_r \\ c_{warm} &= c_{yellow} + \beta c_r \end{aligned} \tag{3}$$

2.1.2 Toon Shading

Mit Toon Shading, das auch unter dem Namen Cel Shading bekannt ist, wird versucht dreidimensionale Objekte in dem Stil eines Comics oder Zeichentrickfilms darzustellen. Toon Shading ist in der Lage einer computergenerierten Szene eine völlig andere Stimmung zu geben und hat vor allem im westlichen Kulturkreis teilweise starke Konnotationen von Kindheit [1].

Der Effekt des Toon Shading wird erreicht, indem für die Darstellung einer Oberfläche nur sehr wenige Farben eingesetzt werden. Man kann die Farben, die zum Einsatz kommen, beispielsweise auf eine Farbe für helle Bereiche, eine andere für dunkle Bereiche und eine dritte für Highlights, in denen sich die Lichtquelle spiegelt, beschränken.

Mit einer lokal konstanten Funktion wird dazu jedes der beiden Skalarprodukte aus Gleichung 1 auf einen von zwei Helligkeitsbereichen abgebildet. Liegt $\vec{n} \cdot \vec{l}$ unterhalb eines Schwellwertes, wird der Bereich als dunkel klassifiziert, liegt das Skalarprodukt oberhalb, gilt der Bereich als hell. Eine zweite lokal konstante Funktion für $\vec{h} \cdot \vec{n}$ liefert einen Schwellwert für die Sichtbarkeit von Highlights.

Eine solche Funktion kann durch den Zugriff auf eine eindimensionale Textur, die Grauwerte speichert, implementiert werden [4]. Das Skalarprodukt $\vec{n} \cdot \vec{l}$ dient dabei als Texturkoordinate und wird bei dem Zugriff auf die Textur auf einen Farbwert abgebildet. Abbildung 3 zeigt eine eindimensionale, monochrome Textur und den Graph der entsprechenden Funktion.

2.1.3 Silhouetten

Abbildung 2 hat bereits gezeigt, dass das Cool-To-Warm Shading, im Gegensatz zum Phong Shading, ohne die zusätzliche Darstellung von Silhouetten[2] nur wenig Sinn macht. Dies liegt daran, dass alle Flächen in etwa die selbe Helligkeit besitzen. Auch das Toon Shading ist nur in Verbindung mit Silhouetten nützlich, da es dem Betrachter durch die wenigen Farbunterschiede ansonsten schwer fallen würde, die Form, die dem dargestellten Objekt zu Grunde liegt, zu erkennen.

Die Stellen, an denen bei einem dreidimensionalen Datensatz Silhouetten erkannt werden sollten, lassen sich grob in drei Kategorien einteilen:

- Beim *Umriss* des Objekts zeigt ein Polygon zum Betrachter hin und ein benachbartes von ihm weg. Ein Spezialfall sind Polygone, die keine Nachbarn besitzen, beispielsweise der Umriss eines Dreiecks.
- Bei *harten Kanten* unterscheiden sich die Normalen von zwei benachbarten Polygonen deutlich.
- Bei *Materialkanten* besitzen zwei benachbarte Polygone unterschiedliche Materialeigenschaften.

Es gibt eine Vielzahl von Algorithmen um unterschiedliche Arten von Silhouetten zu erkennen. Einen Überblick bietet etwa das Buch "Real-Time Rendering" von Akenine-Möller und Haines [1]. Im Rahmen dieser Studienarbeit wurden zwei Algorithmen implementiert, die im Folgenden kurz vorgestellt werden.

Eine denkbar einfache Methode den Umriss darzustellen besteht darin jedes Objekt zweimal auszugeben. Bei einem der beiden Durchläufe wird jedoch jeder Eckpunkt um einen bestimmten Wert entlang seiner Normalen verschoben. Lässt man diese Version des Objekts nun in schwarz und nur im Hintergrund ausgeben, so erhält man den Umriss. Dieses Verfahren lässt sich mittels Vertex-Shader extrem effizient mit moderner Graphikhardware umsetzen und fand bereits Verwendung in einem Computerspiel [1]. Zu Problemen kommt es allerdings bei Polygonen die keine Nachbarn besitzen. Dazu kann man sich beispielsweise ein einfaches Dreieck vorstellen. Da ein Dreieck nur eine Normale besitzt, würden alle Eckpunkte mit der beschriebenen Methode entlang der selben Richtung verschoben werden. Sie müssten aber voneinander weg verschoben werden damit ein größeres Dreieck und somit ein Umriss entsteht. Ein anderes Problem sind Lücken im Umriss, die beispielsweise an Stellen auftreten, an denen die

[2] Der Begriff *Silhouette* wird im Verlauf dieser Arbeit für den Umriss und Linien, die Kanten darstellen, verwendet und ist nicht mit dem Schattenriß, einem Umriss mit schwarzer Fläche, gleichzusetzen.

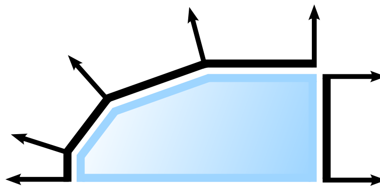


Abbildung 4: Umriss durch Verschiebung von Eckpunkt-Normalen

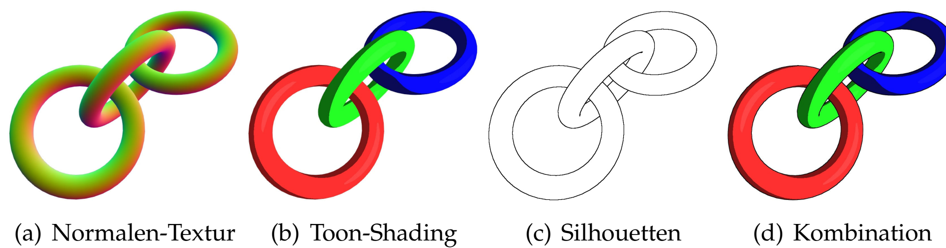
Eckpunkt-Normalen von zwei benachbarten Flächen orthogonal zueinander stehen. Abbildung 4 verdeutlicht diesen Fall.

Ein bildbasiertes Verfahren, das jede Art von Umriss und zusätzlich harte Kanten findet, wird an verschiedenen Stellen besprochen [1, 2, 9, 12]. Hierbei werden zunächst alle Normalen und die Tiefenwerte aus Sicht der Kamera in eine Textur gerendert. Im Rot-, Grün- und Blaukanal der Textur wird dafür $\vec{n_x}$, $\vec{n_y}$ und $\vec{n_z}$ gespeichert und im Alpha-Kanal die Tiefe jedes Bildpunktes. Diese Textur kann nun zur Erkennung von Silhouetten eingesetzt werden, da jedes Texel, das sich auffällig von seinen Nachbarn unterscheidet, auf eine Diskontinuität der Normalen oder der Tiefenwerte, und somit auf eine harte Kante oder einen Umriss schließen lässt. Zur Erkennung auffälliger Übergänge kann man die Textur beispielsweise mit dem Sobel-Operator falten. Das Ergebnis der Kantendetektion wird in einer neuen Textur gespeichert, welche im letzten Schritt auf ein bildschirmfüllendes Rechteck gelegt wird. Für jedes Pixel wird nun gestestet, ob es sich bei dem entsprechenden Texel um eine Silhouette handelt. Ist dies der Fall, wird das Pixel schwarz ausgegeben. Ansonsten wird zur Ermittlung der Farbe auf eine zuvor erzeugte, bildschirmfüllende Textur zugegriffen, die das herkömmlich geshadete Bild enthält.

Zusammengefasst kommt es bei dieser Methode der bildbasierten Silhouettenerkennung insgesamt zu vier Renderdurchläufen:

1. Normalen und Tiefenwerte in eine Textur rendern
2. Geshadetes Bild in eine Textur rendern
3. Sobel-Filter auf die in 1. erzeugte Textur anwenden und Ergebnis in neue Textur rendern
4. Für die gesamte Bildschirmauflösung die in 3. erzeugte Textur durchlaufen und testen, ob Pixel schwarz ausgegeben werden soll oder in der Farbe, welche die in 2. erzeugte Textur an dieser Stelle enthält.

Die Ergebnisbilder der vier Schritte des Verfahrens zeigt Abbildung 5.



(a) Normalen-Textur (b) Toon-Shading (c) Silhouetten (d) Kombination

Abbildung 5: Renderpasses des bildbasierten Silhouettenalgorithmus

Dieses Verfahren erkennt Silhouetten recht zuverlässig, ohne Anforderungen an die Geometrie zu stellen. Einer der wenigen Fälle in denen die vorgestellte Methode versagt, tritt bei minimalen Tiefenunterschieden von zwei Flächen auf, die sehr ähnliche Normalen besitzen. Dazu kann man sich beispielsweise ein Blatt Papier vorstellen, das auf einem Tisch liegt [1]. Ein anderes Problem besteht darin, dass die Silhouetten im Verhältnis zum Rest des Objekts immer stärker zunehmen, je weiter sich die Kamera vom Objekt entfernt. Das liegt daran, dass sich bei einem komplexen Objekt, das mit nur wenigen Pixel dargestellt wird, beinahe alle kodierten Normalen signifikant voneinander unterscheiden. Eine flüssige Darstellung beim Einsatz der vorgestellten Methode ist erst durch aktuelle Graphikhardware möglich geworden. Die Dauer der Silhouettenerkennung ist zwar völlig unabhängig von der Anzahl der dargestellten Polygone, sie steigt jedoch mit der Auflösung des Bildschirms, da sich in gleichem Maße die Größe der Textur ändert, auf die der Sobel-Filter angewandt wird.

2.2 Programmierung der Graphics Processing Unit (GPU)

2.2.1 Fixed-Function-Pipeline

Die Fixed-Function-Pipeline beschreibt die Verarbeitungsschritte, die nötig sind um ein dreidimensionales Objekt, das in Form von Daten gegeben ist, auf dem Bildschirm darzustellen. Diese Schritte waren bis zur Entwicklung der programmierbaren Graphikkarten in ihrer Funktionalität festgelegt. Abbildung 6 zeigt die Einheiten der Pipeline in ihrer Abfolge.

Die Daten, die in die Pipeline gehen, stammen üblicherweise von einem Programm das OpenGL oder Direct3D als Programmierschnittstelle zur Darstellung von dreidimensionaler Graphik nutzt. Zunächst werden alle Eckpunkte und Normalen des darzustellenden Objekts in das Kamerakoordinatensystem transformiert. Danach können die Eckpunkte bereits beleuchtet werden. In der nächsten Stufe werden alle Eckpunkte zu geometrischen Primitiven zusammengefasst. Diese werden am View Frustum geclippt und eventuell verworfen, falls sie sich auf der Rückseite des Objekts befinden und daher nicht gesehen werden können (*Backface Culling*).

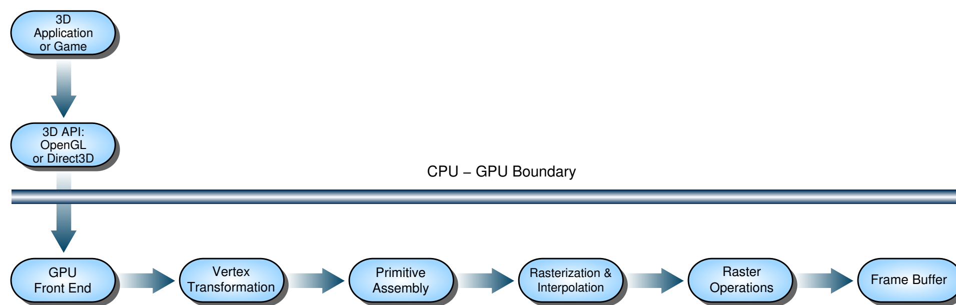


Abbildung 6: Die Fixed-Function-Graphik-Pipeline [4]

Bei der Rasterisierung wird die Menge an Bildelementen ermittelt, die jedes Polygon zu seiner Darstellung benötigt. An dieser Stelle spricht man noch von *Fragments* anstatt von Pixel [4]. Jedes Fragment wird mit Werten über die Tiefe, Farbe und Texturkoordinaten verknüpft, die aus den transformierten Eckpunkten des Polygons interpoliert werden. Bevor das Fragment zum Frame-Buffer geschickt und somit als Pixel dargestellt werden kann, muss es noch eine Reihe von Tests, wie den Alpha-Test, den Stencil-Test und den Tiefentest, durchlaufen.

2.2.2 Programmierbare Graphik-Pipeline

Moderne Graphikprozessoren erlauben es dem Entwickler, die Funktionalität der Graphik-Pipeline zu einem Teil selbst zu programmieren. Dadurch ist es heute möglich viele Effekte in Echtzeit darzustellen, die vorher nur durch die teils stundenlange Rechenzeit von Offline-Renderern möglich gewesen sind. Da sich die Aufgaben der Graphics Processing Unit auf die Graphik-Pipeline beschränken, sind moderne Graphikkarten mittlerweile in der Lage, bestimmte Operationen um ein Vielfaches schneller auszuführen als heutige CPUs. Dies macht die Programmierung der Graphikprozessoren auch für andere rechenintensive Anwendungen ohne direkten Bezug zur Computergraphik interessant. Die beiden Abschnitte der Graphik-Pipeline, deren Funktionalität derzeit durch selbst geschriebene Programme ersetzt werden kann, zeigt Abbildung 7.

Der programmierbare Vertex-Prozessor ersetzt die Transformation der Eckpunkte in der Fixed-Function-Pipeline. Zunächst lädt er die Attribute jedes einzelnen Eckpunktes, wie beispielsweise die Position, die Normale oder die Farbe. Die Instruktionen des programmierten Vertex-Shaders werden anschließend auf den Attributen jedes Eckpunkts ausgeführt. Für eine perspektivisch korrekte Darstellung müssen alle Eckpunkte im Vertex-Shader transformiert werden. Es ist nicht möglich, mit dem programmierbaren Vertex-Prozessor neue Eckpunkte zu erzeugen, oder bestimmte Eckpunkte zu verwerfen.

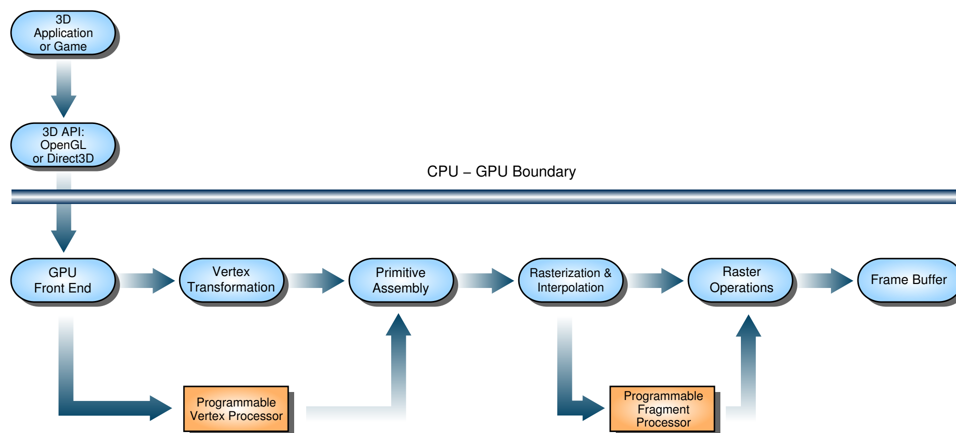


Abbildung 7: Die programmierbare Graphik-Pipeline [4]

Der Fragment-Prozessor ersetzt den Teil der Fixed-Function-Pipeline, der die endgültige Farbe jedes Fragments bestimmt. Es wird dadurch ermöglicht erst an dieser Stelle der Pipeline zu beleuchten. Dies führt zu wesentlich besseren Ergebnissen als eine Beleuchtung der transformierten Eckpunkte, da sich sonst die Farbe der Fragments nur aus einer Interpolation zwischen den Eckpunkten ergeben würde. Durch den Zugriff auf Texturen, die zuvor als Render-Target dienten, erlauben Fragment-Shader auch die Programmierung von Algorithmen der Bildverarbeitung und damit die Anwendung diverser Filter auf das zu rendernde Bild.

2.2.3 NVIDIA Cg

Cg ist eine von NVIDIA, in Zusammenarbeit mit Microsoft, entwickelte Hochsprache zur Programmierung der Graphikkarte. Der Name steht für "C for Graphics" und deutet bereits an, dass Cg eine C-ähnliche Syntax besitzt.

Vor der Entwicklung von Hochsprachen wie Cg, der Microsoft High-Level Shading Language (HLSL), oder der OpenGL Shading Language war eine direkte Graphikkartenprogrammierung nur mit Hilfe von Assembler-Sprachen möglich. Gegenüber solchen Assembler-Sprachen bieten Hochsprachen Vorteile, welche die Wartbarkeit, die Wiederverwendbarkeit und die Portabilität des Quellcodes betreffen. Der Cg-Compiler übernimmt darüber hinaus fehleranfällige Aufgaben, wie die Allokation von Speicher, und nimmt Optimierungen vor, die in einer höheren Effizienz resultieren.

Mit OpenGL und Direct3D unterstützt Cg die zwei gängigsten Programmierschnittstellen zur 3D-Graphikprogrammierung. Cg besitzt dazu die beiden Bibliotheken cgGL und cgD3D, deren Aufgabe es ist den Cg-Code in eine Form zu übersetzen, die von der eingesetzten Programmierschnittstelle verstanden wird. Diese schickt die Anweisungen anschließend

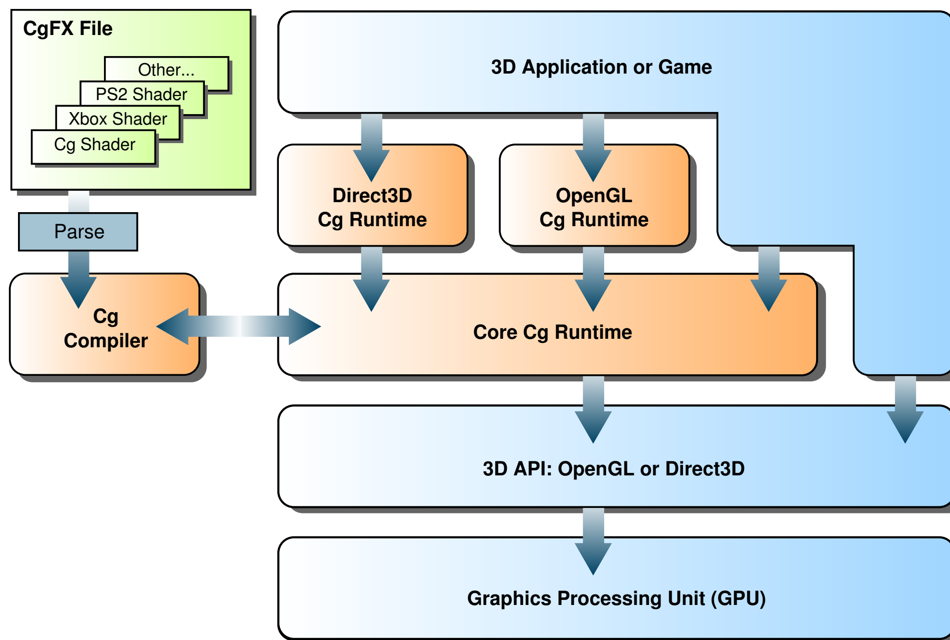


Abbildung 8: Cg und CgFX im Kontext einer 3D-Anwendung [4]

an die GPU. Die Cg-Runtime-Bibliothek bietet die Möglichkeit den Shader-Code erst zur Laufzeit dynamisch zu kompilieren, wodurch sich das Programm optimal den Hardware-Gegebenheiten der Zielplattform anpassen kann. Abbildung 8 zeigt wie die Bestandteile von Cg zwischen der 3D-Anwendung und der Programmierschnittstelle einzuordnen sind.

Durch die enge Zusammenarbeit von NVIDIA und Microsoft sind Cg und die High-Level Shading Language im Grunde identisch, bis auf den Unterschied, dass Cg auch mit OpenGL und unter Linux oder dem Mac OS X eingesetzt werden kann [4].

2.2.4 NVIDIA CgFX

CgFX ist ein Dateiformat zur Spezifikation komplexer Shader.[3] In einer CgFX-Datei lassen sich unterschiedliche Vertex- und Fragment-Shader mit der Syntax von Cg zu Techniken (*techniques*) zusammenfassen. Dies erlaubt beispielsweise die optimale Unterstützung mehrerer Generationen von Graphikhardware, indem zur Laufzeit ermittelt wird, welche Technik eingesetzt werden kann. Darüber hinaus ist CgFX in der Lage Shader zu definieren, die aus mehreren Renderdurchläufen (*passes*) bestehen. Für jeden dieser Durchläufe lassen sich Eigenschaften wie die Texturfilterung

[3] Das CgFX Format ist identisch zum Microsoft DirectX 9.0 Effect Format (.fx 2.0).

oder die Richtung des Backface Culling festlegen.

Dadurch, dass CgFX alle Informationen eines Shading-Effekts in einer Datei vereint, lassen sich CgFX-Shader leicht zwischen verschiedenen Programmen portieren. Anwendungen für *Digital Content Creation* (DCC), wie beispielsweise Maya von Autodesk, sind mit Hilfe von Plug-Ins in der Lage CgFX-Shader zu laden und anzuwenden. Über die gewohnten Schnittstellen ist der Benutzer des Programms dann in der Lage sämtliche Parameter des Shaders direkt zu ändern.

2.3 Microsoft Direct3D und DXUT

Direct3D ist eine Komponente der Programmierschnittstelle DirectX, die seit 1995 von Microsoft für das Betriebssystem Windows weiterentwickelt wird. DirectX 9 besteht darüber hinaus aus den Komponenten DirectInput zur Unterstützung von Eingabegeräten und DirectSound für die Entwicklung von Audio-Anwendungen. Neben der Entwicklung von 3D-Graphik-Anwendungen unter Windows, kommt DirectX auch bei der Programmierung von Videospielen für Microsofts Xbox 360 zum Einsatz [5].[4]

Das DirectX Utility Toolkit (DXUT) unterstützt die Entwicklung von Direct3D-Anwendungen [11]. Das Framework vereinfacht die Erzeugung von Fenstern und bietet die Möglichkeit unkompliziert zwischen Vollbild- und Fenster-Modus zu wechseln. Darüber hinaus stehen dem Entwickler zusätzliche Klassen zur Verfügung, beispielsweise für unterschiedliche Kameraarten, und es gibt die Möglichkeit einer präzisen Zeitmessung. Besonders interessant ist eine Sammlung von Elementen zur Erzeugung eines *Graphical User Interface* (GUI). Über Callback-Funktionen bietet das DXUT dem Entwickler die Möglichkeit auf verschiedene Events, wie das Beenden des Programms, einen Tastendruck des Benutzers oder das Anklicken eines GUI-Elements, unterschiedlich zu reagieren.

[4] Im Gegensatz zur PC-Version unterstützt die DirectX-Variante der Xbox 360 nicht mehr die Fixed-Function-Pipeline, sondern setzt vollständig auf die programmierbare Graphik-Pipeline.

3 Implementierung

3.1 Klassendiagramm

Abbildung 9 zeigt ein UML-Klassendiagramm zu dem Programm, das im Rahmen dieser Studienarbeit implementiert wurde. Attribute und Methoden wurden zur Übersichtlichkeit ausgelassen. Im Folgenden wird auf die Architektur der Anwendung eingegangen, indem die Aufgaben der einzelnen Klassen kurz vorgestellt werden.

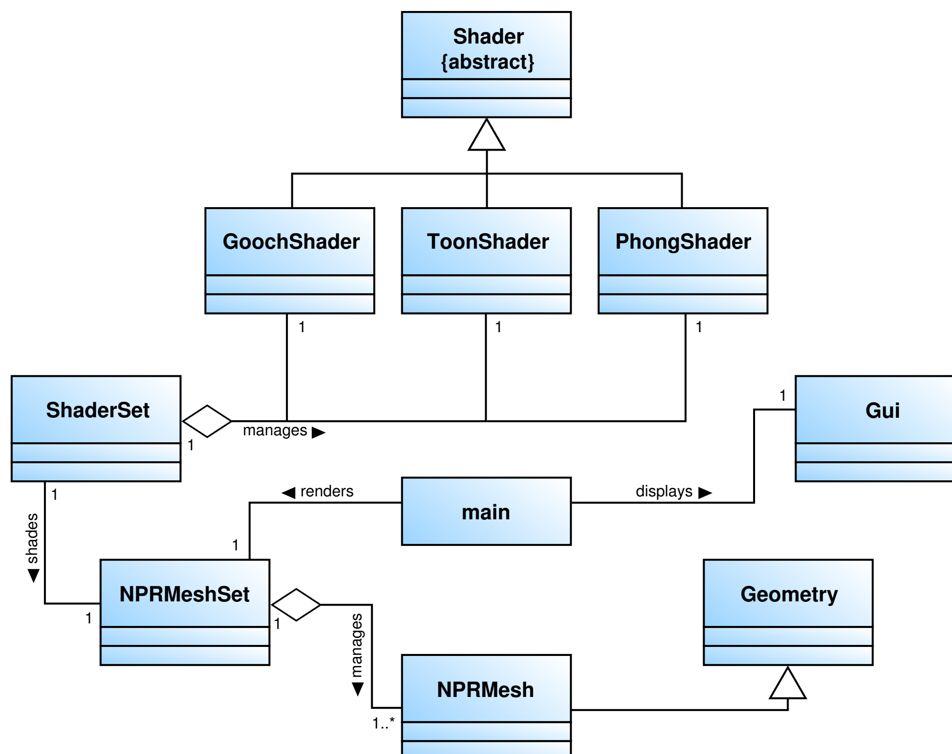


Abbildung 9: UML-Klassendiagramm

Geometry

Jede Instanz der Klasse *Geometry* speichert ein 3D-Modell. Die Klasse beinhaltet eine Methode zum Laden von Modellen aus dem X-Dateiformat (.x) von Microsoft. Da Direct3D für diese Aufgabe bereits Methoden bereitstellt, gestaltet sich dies recht unkompliziert. Beim Laden eines Modells werden zusätzlich seine Materialeigenschaften ausgelesen und von *Geometry* verwaltet. Das X-Format unterteilt jedes 3D-Objekt in Untermengen mit dem gleichen Material. Indem man beim Rendervorgang in einer Schleife

über alle Untermengen eines Modells läuft und diese nacheinander ausgibt, wird das teuere Umschalten des Materials minimiert. Texturen, die in der X-Datei Untermengen der Geometrie zugeordnet sind, werden gleich nach dem Modell geladen und von *Geometry* verwaltet. Die Direct3D-Methode `D3DXCreateTextureFromFile()` lädt Texturen in den gängigsten Bildformaten. Die DirectX-Dokumentation empfiehlt jedoch Texturen in Microsofts DirectDraw-Surface-Dateiformat (.dds) abzuspeichern [11].

NPRMesh

Die Klasse *NPRMesh* ist eine Spezialisierung von *Geometry*. Sie beinhaltet zusätzliche Funktionalität für alle Modelle, die mit einem Shading dargestellt werden. Die einzige Instanz von *Geometry*, die kein *NPRMesh* ist, verwaltet das bildschirmfüllende Rechteck, das für den bildbasierten Silhouettenalgorithmus benötigt wird (siehe dazu Abschnitt 2.1.3 und Abschnitt 3.3.2).

NPRMeshSet

Alle Instanzen von *NPRMesh* werden von der Klasse *NPRMeshSet* verwaltet. Hier werden sämtliche 3D-Modelle geladen und das darzustellende Modell als aktiv markiert. Mit Methoden, die *NPRMeshSet* bereitstellt, kann das aktive Modell gewechselt werden. Um diesen Wechsel attraktiv zu gestalten, wird das vorherige Modell schrittweise verkleinert und das neue Modell anschließend bis zu seiner angedachten Größe hochskaliert. Diese Animation dauert etwa zwei Sekunden. Damit der Übergang zwischen den Modellen flüssig abläuft, werden alle Modelle gleich zum Start des Programms in den Arbeitsspeicher geladen.

Um dem Programm ein zusätzliches 3D-Modell hinzuzufügen, muss in *NPRMeshSet* lediglich eine neue Instanz von *NPRMesh* erzeugt werden.

Shader

Die Klasse *Shader* enthält Attribute und Methoden um Materialeigenschaften und Transformationsmatrizen an die CgFX-Shader zu übertragen. Von *Shader* selbst existieren keine Instanzen — lediglich von den drei Spezialisierungen *GoochShader*, *ToonShader* und *PhongShader*. Jede dieser drei Klassen verwaltet einen CgFX-Shader und enthält spezielle Methoden und Attribute für das jeweilige Shading.

Um das Programm mit einem zusätzlichen Shading auszustatten, muss eine neue Klasse von *Shader* abgeleitet werden.

ShaderSet

Jeweils eine Instanz jeder von *Shader* abgeleiteten Klasse wird von *ShaderSet* verwaltet. Zu jedem Zeitpunkt ist eines dieser drei Objekte als aktiv markiert. *ShaderSet* stellt Methoden zur Verfügung um das aktive Shading zu wechseln und die Silhouetten an- und abzuschalten. Das Shading bleibt beim Wechseln des Modells dasselbe. Hat der Benutzer sich beispielsweise für Toon Shading mit bildbasierter Silhouette entschieden und alle Einstellungen vorgenommen, bekommt er beim Durchschalten sämtliche verfügbaren Modelle mit diesem Shading angezeigt.

Gui

Die Klasse *Gui* verwaltet sämtliche Elemente der Benutzerschnittstelle. Die Instanz von *Gui* gibt Informationen darüber, ob eine Schaltfläche aktiviert ist und auf welchem Wert ein Schieberegler steht.

Utility

Die abstrakte Hilfsklasse *Utility* wird von mehreren Klassen verwendet und wurde für eine bessere Übersicht in Abbildung 9 ausgelassen. *Utility* verwaltet den Namen des Programms und die aktuelle Auflösung des Fensters. Sie enthält darüber hinaus zwei Methoden, die ermitteln, ob Cg einen Fehler meldet.

main

Bei *main* handelt es sich um keine Klasse, sondern den Startpunkt des Programms. Die Methode `WinMain()` erzeugt das Fenster der Anwendung, setzt alle Callback-Funktionen des DXUT und gibt anschließend die Kontrolle an das DXUT, welches sich um die Behandlung von Nachrichten kümmert und die Render-Schleife aufruft.

3.2 Shading-Modelle

Jedes Shading-Modell wurde in einer eigenen CgFX-Datei implementiert. Diese Datei ruft die nötigen Vertex- und Fragment-Shader, die alle in separaten Dateien gespeichert sind, mit den entsprechenden Parametern auf.

Da alle drei implementierten Shading-Modelle für eine bessere Ausgabequalität lediglich im Fragment-Shader beleuchten, ist der Vertex-Shader identisch. Listing 1 zeigt den Inhalt des Vertex-Shaders, der für das Cool-to-Warm Shading, das Toon Shading und das Phong Shading zum Einsatz kommt. Die Beleuchtung im Fragment-Shader findet im Kamerakoordinatensystem statt, sodass die Position und die Normale jedes Eckpunktes für das Shading mit der Modelview-Matrix (`ModelView`), beziehungs-

Listing 1 Cg-Vertex-Shader für alle drei Shading-Modelle

```
void eachShader_vertex(
    in  float4 position : POSITION,
    in  float4 normal   : NORMAL,

    out float4 outPosition    : POSITION,
    out float3 objectPosition : TEXCOORD0,
    out float3 objectNormal   : TEXCOORD1,

    uniform float4x4 ModelViewProj,
    uniform float3x3 ModelView,
    uniform float3x3 ModelViewIT )
{
    objectPosition = mul(ModelView, position.xyz);
    objectNormal   = normalize(mul(ModelViewIT, normal.xyz));
    outPosition    = mul(ModelViewProj, position);
}
```

weise ihrer transponierten Inversen (`ModelViewIT`), multipliziert werden müssen. Dadurch, dass Position und Normale als Texturkoordinaten gespeichert werden, kann der Fragment-Shader auf diese Werte zugreifen. Für eine perspektivisch korrekte Ausgabe muss jeder Eckpunkt im Vertex-Shader zudem mit einer Matrix multipliziert werden, welche auch die Projektionsmatrix enthält (`ModelViewProj`).

Jedes Modell wird im Folgenden von einer einzigen Punktlichtquelle beleuchtet, da Cool-to-Warm Shading und Toon Shading so am Besten zur Geltung kommen. Die diffuse Farbe, die Farbe der Higlights und der Phong-Exponent — in den folgenden Listings `c_r`, `c_p` und `p` genannt — werden im Programm aus der X-Datei ausgelesen und können somit für jedes 3D-Modell separat festgelegt werden.

3.2.1 Cool-to-Warm Shading

Wie das in Abschnitt 2.1.1 erläuterte Cool-to-Warm Shading implementiert wurde, zeigt Listing 2. Da im Kamerakoordinatensystem beleuchtet wird — die Kamera also entlang der Z-Achse blickt — kann der Sichtstrahl mit (0, 0, -1) initialisiert werden. Die lineare Interpolation zwischen kalter und warmer Farbe wird mit der Funktion `lerp()` aus der Cg-Bibliothek realisiert. Die Implementierung des Cool-to-Warm Shadings wurde zusätzlich um Highlights erweitert. Dazu wird, genau wie beim Phong Shading aus Gleichung 1, das Skalarprodukt von $\vec{h}$ und $\vec{n}$ mit dem Exponenten p aufaddiert. Damit alle Glanzlichter beim Cool-to-Warm Shading weiß bleiben,

Listing 2 Cg-Fragment-Shader für Cool-to-Warm Shading

```
void goochShader_fragment(
    in  float3 position : POSITION,
    in  float3 normal   : NORMAL,

    out float4 outColor : COLOR,

    uniform float3 lightPosition,
    uniform float4 c_r,
    uniform float  p,
    uniform float3 c_blue,
    uniform float3 c_yellow,
    uniform float  alpha,
    uniform float  beta )
{
    float3 N = normal;
    float3 L = normalize(lightPosition - position);
    float3 V = float3(0.0, 0.0, -1.0);
    float3 H = normalize(L + V);

    float3 c_cool = c_blue + alpha * c_r.rgb;
    float3 c_warm = c_yellow + beta * c_r.rgb;

    float3 gooch = lerp(c_cool, c_warm, (1 + dot(N, L)) * 0.5);

    float specularLight = pow(max(dot(H, N), 0.0), p);

    outColor.rgb = gooch + specularLight;
    outColor.a = 1.0;
}
```

wird dieser Teil jedoch nicht mit der spekularen Farbe `c_p` multipliziert.

3.2.2 Toon Shading

Listing 3 zeigt den Fragment-Shader, der für das Toon Shading zum Einsatz kommt. Wie in Abschnitt 2.1.2 beschrieben, wird die harte Abstufung der Farben durch den Zugriff auf zwei eindimensionale Texturen realisiert.

Damit die Texturen im Fragment-Shader verfügbar sind, müssen sie im C++-Code mit Direct3D geladen und der benutzerdefinierten Semantik der CgFX-Datei für Toon Shading zugewiesen werden. Listing 4 zeigt dieses Vorgehen am Beispiel der 1D-Textur für das diffuse Shading. Fehlerbehandlungen für den Fall, dass die Textur nicht gefunden wurde, oder

Listing 3 Cg-Fragment-Shader für Toon Shading

```
void toonShader_fragment(
    in  float3 position : POSITION,
    in  float3 normal   : NORMAL,

    out float4 outColor : COLOR,

    uniform float3 lightPosition,
    uniform float4 c_r,
    uniform float4 c_p,
    uniform float  p,
    uniform sampler1D diffuseRamp,
    uniform sampler1D specularRamp )
{
    float3 N = normal;
    float3 L = normalize(lightPosition - position);
    float3 V = float3(0.0, 0.0, -1.0);
    float3 H = normalize(L + V);

    float diffuseLight = max(dot(N, L), 0.0);
    float specularLight = pow(max(dot(H, N), 0.0), p);

    if(diffuseLight <= 0.0)
        specularLight = 0.0;

    diffuseLight  = tex1D(diffuseRamp, diffuseLight).x;
    specularLight = tex1D(specularRamp, specularLight).x;

    outColor = c_r * diffuseLight + c_p * specularLight;
}
```

`diffuseRamp` in der CgFX-Datei nicht definiert ist, wurden zur Übersichtlichkeit im Listing ausgelassen. Die Member-Variable `mCgEffect` wurde vorher bereits für die CgFX-Datei des Toon Shadings initialisiert.

Um dem Benutzer des Programms mehr Kontrolle über die Farbübergänge beim Toon Shading zu überlassen, wurde der Zugriff auf die beiden Texturen letztlich durch einen variablen Schwellwert ersetzt, der von der Klasse *Gui* verwaltet wird. Damit können die Farbübergänge und die Sichtbarkeit von Highlights über zwei Schieberegler der graphischen Benutzerschnittstelle angepasst werden.

Listing 4 Laden von Texturen mit Direct3D

```
void ToonShader::loadDiffuseRampTexture( IDirect3DDevice9* d3dDev,
    const WCHAR* diffRampFile )
{
    IDirect3DTexture9* diffRamp;

    D3DXCreateTextureFromFileW( d3dDev, diffRampFile, &diffRamp )

    CGparameter param =
        cgGetNamedEffectParameter( mCgEffect, "diffuseRamp" );

    cgD3D9SetTextureParameter( param, diffRamp );
    cgSetSamplerState( param );
}
```

Listing 5 Ausschnitt des Cg-Fragment-Shaders für Phong Shading

```
    float diffuseLight = max(dot(N, L), 0);
    float specularLight = pow(max(dot(H, N), 0), p);

    outColor = c_r * diffuseLight + c_p * specularLight;
```

3.2.3 Phong Shading

Das Phong Shading wurde implementiert, um die Shading-Methoden des Non-Photorealistic Rendering mit einem klassischen Beleuchtungsmodell vergleichen zu können. Listing 5 zeigt einen Ausschnitt der Cg-Datei für Phong Shading. Der Shader gleicht dem aus Listing 3, mit der Ausnahme, dass die beiden Texturzugriffe fehlen.

3.3 Silhouettenalgorithmen

Die Möglichkeit jederzeit zwischen den beiden implementierten Verfahren zum Zeichnen von Silhouetten und einer Darstellung ohne Silhouetten zu wechseln, wurde über verschiedene Techniken in jeder CgFX-Datei realisiert.

Listing 6 zeigt an einem Ausschnitt der Datei für Toon Shading, wie die Technik (`technique`) für den einfachen Silhouettenalgorithmus, auf den im folgenden Abschnitt eingegangen wird, aussieht. Zunächst werden alle nötigen Vertex- und Fragment-Shader mit `#include` eingebunden. An-

Listing 6 Auschnitt der Datei toonShader.cgfx

```
#include ≪eachShader_vertex.cg≫
#include ≪toonShader_fragment.cg≫
#include ≪silhouette_vertex.cg≫
#include ≪silhouette_fragment.cg≫

float4x4 mvp : ModelViewProjectionMatrix;
float3x3 mv  : ModelViewMatrix;
float3x3 mvit: ModelViewInverseTransposeMatrix;

float3 lightPos : lightPosition;
float4 diffuse  : diffuse;
float4 specular : specular;
float  shininess: specularPower;

float silhSize: silhouetteSize;

float diffThreshold: diffuseThreshold;
float specThreshold: specularThreshold;

technique simpleSilhouette
{
  pass p0
  {
    ZWriteEnable = false;
    VertexShader = compile vs_2_0 silhouette_vertex( mvp, silhSize );
    PixelShader  = compile ps_2_0 silhouette_fragment();
  }
  pass p1
  {
    ZWriteEnable = true;
    VertexShader = compile vs_2_0 eachShader_vertex( mvp, mv, mvit );
    PixelShader  = compile ps_2_0 toon_fragment( lightPos, diffuse,
        specular, shininess, diffThreshold, specThreshold );
  }
}
```

schließend werden alle Variablen, die im Shader zum Einsatz kommen, mit einer benutzerdefinierten Semantik deklariert und mit den entsprechenden Werten aus dem C++-Code initialisiert. Die Matrizen zur Transformation werden dabei von der Kamera-Klasse des DXUT geliefert. Die Materialparameter werden von der Klasse *Geometry* verwaltet. Die Variablen, die sich auf das Shading und die Silhouette beziehen, werden über die Benutzerschnittstelle eingestellt und aus einer Instanz der Klasse *Gui* ausgelesen.

Welche Technik des CgFX-Shaders aktiv ist, hängt davon ab, welche Auswahl über die Benutzerschnittstelle getroffen wurde. Wurde beispielsweise ein Toon Shading komplett ohne Silhouetten gewählt, dann kommt eine Technik mit nur einem Renderdurchlauf zum Einsatz, die mit `pass p1` aus Listing 6 identisch ist.

3.3.1 Silhouetten durch Manipulation der Geometrie

Eine besonders einfache Möglichkeit Silhouetten zu rendern wurde in Abschnitt 2.1.3 vorgestellt. In einem zusätzlichen Renderdurchlauf wird dabei jedes Modell leicht vergrößert und in schwarz dargestellt.

Listing 6 zeigt, dass Vertex- und Fragment-Shader für die Silhouetten im ersten Renderpass der Technik aufgerufen werden. Damit die Flächen des Modells, die zum Betrachter hin verschoben werden, nicht das Shading des zweiten Renderdurchlaufs überdecken, dürfen die Silhouetten-Shader nicht in den Depth Buffer schreiben. Dies lässt sich in der CgFX-Datei mit `ZWriteEnable = false` festlegen. Alternativ dazu könnte man nur die rückseitigen Eckpunkte des Modells entlang ihrer Normalen verschieben.

Der Vertex-Shader, welcher alle Eckpunkte entlang ihrer Normalen verschiebt, wird in Listing 7 gezeigt. Die Variable `displacement` bestimmt, wie stark die Verschiebung und damit die Breite der Silhouette ist. Der Wert liegt im Programm zwischen 0 und 1 und lässt sich über die graphische Benutzerschnittstelle ändern. Da meistens eine Silhouette erwünscht ist, die in allen Bereichen des Modells gleich breit ist, müsste bei großen Tiefenunterschieden der Wert der Verschiebung zusätzlich von der Entfernung des Eckpunktes zur Kamera abhängig gemacht werden. Andernfalls wird durch die perspektivische Verzerrung die Silhouette der Teile schmaler, die weiter von der Kamera entfernt sind.

Der Vertex-Shader legt schwarz als Farbe für die Silhouetten fest. Im zugehörigen Fragment-Shader muss lediglich noch die ausgehende Farbe den Wert der eingehenden Farbe zugewiesen bekommen.

3.3.2 Bildbasierte Erkennung von Silhouetten

In Abschnitt 2.1.3 wurde auch ein bildbasiertes Verfahren zur Erkennung von Silhouetten vorgestellt. An den Stellen, an denen hierbei ein Filter Unstetigkeiten der Normalen oder der Tiefenwerte erkennt, sollen Silhouetten

Listing 7 Cg-Vertex-Shader für einfache Silhouetten

```
void silhouette_vertex(
    in float4 position : POSITION,
    in float4 normal   : NORMAL,

    out float4 outPosition : POSITION,
    out float4 outColor    : COLOR,

    uniform float4x4 ModelViewProj,
    uniform float    displacement )
{
    float4 direction = float4(normalize(normal.xyz), 0.0);
    float4 newPosition = position + displacement * direction;

    outPosition = mul(ModelViewProj, newPosition);
    outColor    = float4(0.0, 0.0, 0.0, 1.0);
}
```

dargestellt werden.

In der Callback-Methode des DXUT, die für das Rendern zuständig ist, wird das aktuelle Render-Target zunächst in einer Variablen hinterlegt. Anschließend kann mit der Direct3D-Methode `SetRenderTarget()` eine Textur als neues Render-Target bestimmt werden.

Im ersten Schritt wird eine Technik des CgFX-Shaders aktiviert, die Normale und Tiefe jedes Fragments der ausgehenden Farbe zuweist. Daraufhin wird die gesamte Szene zum ersten Mal gerendert. Das Ergebnis ist eine Textur, die später zur Detektion der Silhouetten eingesetzt werden kann.

Als nächstes wird mit Direct3D eine neue Textur als Render-Target festgelegt und die Szene mit dem aktiven Shading gerendert. Die entsprechende Technik des CgFX-Shaders unterscheidet sich nicht von einem Shading ohne Silhouetten. Es kommt also der Vertex-Shader aus Listing 1 zum Einsatz und einer der Fragment-Shader aus Listing 2, 3 oder 5.

Im dritten Schritt wird erneut eine Textur als Render-Target bestimmt. Da nun ein Sobel-Filter auf die Textur des ersten Renderdurchlaufs angewandt werden soll, wird diese dem CgFX-Shader als Parameter übergeben. Der Vertex-Shader des dritten Renderdurchlaufs transformiert ein Rechteck so, dass es den kompletten Bildschirm ausfüllt, und legt die Texturkoordinaten fest, sodass die Textur mit den Normalen und Tiefenwerten auf dem Rechteck liegt. Listing 8 zeigt den Fragment-Shader, der auf dieser Textur anschließend mit einem Sobel-Filter Kanten detektiert. Die Methode `sobel()` greift mit der Cg-Funktion `tex2D()` auf das aktuelle Texel und

Listing 8 Cg-Fragment-Shader für das Zeichnen eines Silhouettenbildes

```
void edgeDetection_fragment(
    in float2 texCoord : TEXCOORD0,

    out float4 outColor : COLOR,

    uniform sampler2D normalAndDepthMap,
    uniform float     hOffset,
    uniform float     vOffset,
    uniform float     sobelThreshold )
{
    if( sobel(normalAndDepthMap, texCoord, hOffset, vOffset)
        > sobelThreshold )
        outColor.rgba = float4(0.0, 0.0, 0.0, 1.0);
    else
        outColor.rgba = float4(1.0, 1.0, 1.0, 1.0);
}
```

alle innerhalb des Offsets liegenden Nachbarn zu und liefert als Ergebnis die Summe der vier Farbkanäle nach der Sobel-Operation. Mit dieser Summe wird anschließend anhand eines Schwellwerts, der über die Benutzerschnittstelle reguliert werden kann, überprüft, ob es sich bei einem Fragment um den Teil einer Silhouette handelt. Alle als Silhouette klassifizierten Fragments werden schwarz ausgegeben, alle restlichen weiß. Damit die Entfernung zu den benachbarten Texel auch tatsächlich mit den entsprechenden Pixel übereinstimmt, müssen `hOffset` und `vOffset` abhängig von der aktuellen Auflösung berechnet werden. Die Klasse *Utility* verwaltet dazu die Größe des Fensters. Mit den Callback-Methoden des DXUT lässt sich die aktuelle Auflösung des Fensters zum Start des Programms und nach jeder Änderung durch den Benutzer, sofort feststellen.

Im letzten Schritt wird das zuvor gespeicherte, ursprüngliche Render-Target wieder aktiv und eine Kombination aus Silhouetten- und Shading-Textur wird auf einem bildschirmfüllenden Rechteck dargestellt. Listing 9 zeigt den Fragment-Shader, der die Silhouetten mit dem Shading verbindet. Ob es sich bei einem Texel der Silhouetten-Textur um eine Silhouette handelt, wird mit der Funktion `all()` aus der Cg-Bibliothek ermittelt. Für jedes Texel der Silhouetten-Textur, das nicht schwarz ist, wird die Textur, in der das Shading gespeichert wurde, an der entsprechenden Stelle ausgelesen. In dem implementierten Programm realisiert der Fragment-Shader zusätzlich die Option eines Gauss-Filters, der auf die Shading-Textur angewandt wird, sowie eine Ausgabe von weißen Pixel anstelle des Shadings.

Listing 9 Cg-Fragment-Shader für die Kombination von Silhouetten und Shading

```
void combination_fragment(
    in float2 texCoord : TEXCOORD0,

    out float4 outColor : COLOR,

    uniform sampler2D silhouetteMap,
    uniform sampler2D shadingMap )
{
    if( all(tex2D(silhouetteMap, texCoord).rgb) == false )
        outColor = float4(0.0, 0.0, 0.0, 1.0);
    else
        outColor = tex2D(shadingMap, texCoord);
}
```

3.4 Graphische Benutzerschnittstelle

Abbildung 10 zeigt an einem Zustandsautomaten die Auswirkungen aller Elemente der graphischen Benutzerschnittstelle des Programms. Alle Zustände sind in insgesamt sechs Regionen unterteilt, wobei in jeder Region stets ein Zustand aktiv ist. Die schwarzen Kreise verweisen auf die Startzustände. Alle Transitionen werden durch ein Ereignis ausgelöst und müssen teilweise eine Bedingung erfüllen, die in eckigen Klammern angegeben ist. Manche Zustände besitzen ein Eintrittsverhalten (*entry*), ein internes Verhalten (*do*), oder ein Austrittsverhalten (*exit*).

Es kann zu jedem Zeitpunkt zwischen allen drei Arten von Shading gewechselt werden. Die variablen Parameter eines Shadings können jedoch nur verändert werden, wenn das entsprechende Shading aktiv ist.

Ein Wechsel vom Fenster-Modus zum Vollbild-Modus und ein Anzeigen oder Verbergen der Hilfe ist jederzeit über die GUI, beziehungsweise die Taste F1, möglich.

Das darzustellende 3D-Modell kann nur zyklisch gewechselt werden. Um dies zu verdeutlichen und trotzdem Übersichtlichkeit zu bewahren, enthält der Zustandsautomat lediglich drei Modelle.

Es ist stets möglich zwischen den beiden implementierten Silhouettenalgorithmen und einer Darstellung ohne Silhouetten zu wechseln. Ist das bildbasierte Verfahren zur Silhouettenerkennung aktiv, dann können zwei zusätzliche Operationen auf den verwendeten Texturen zum Einsatz kommen. Zum einen ist es möglich jedes Pixel, das keiner Silhouette angehört weiß auszugeben. Das Shading wird somit unterdrückt und die Silhouetten kommen besser zur Geltung. Zum anderen kann ein Gauss-Filter auf

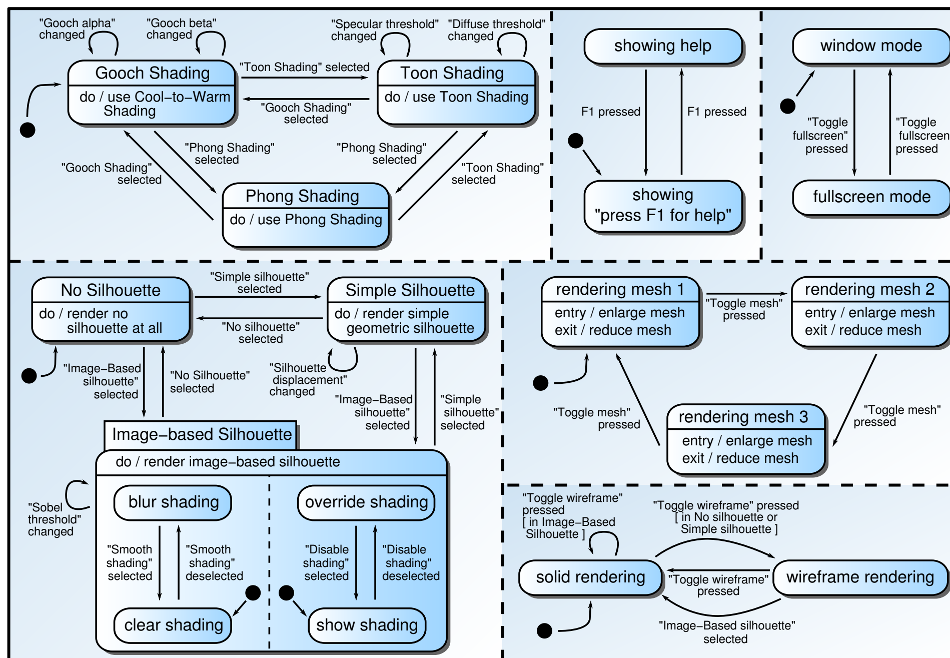


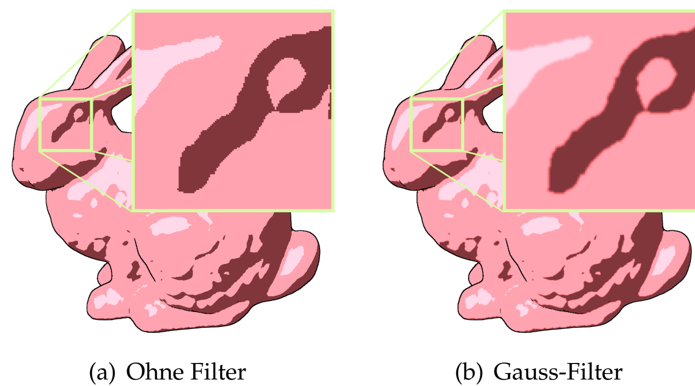
Abbildung 10: Zustandsautomat

alle Pixel, die keiner Silhouette entsprechen, angewandt werden. Dieses Weichzeichnen unterdrückt Aliasing-Effekte, die vor allem bei den harten Farbübergängen des Toon Shading auffallen. Abbildung 11 zeigt den Effekt des Gauss-Filters in vierfacher Vergrößerung.

Die Benutzerschnittstelle bietet zuletzt die Möglichkeit, sich jedes Modell im Wireframe-Modus anzeigen zu lassen. Dabei werden die Eckpunkte aller Polygone, aus denen ein Modell besteht, durch Linien miteinander verbunden. Ein solches Drahtgittermodell liefert dem Betrachter einen guten Eindruck darüber, aus wievielen Polygonen ein Modell tatsächlich aufgebaut ist. Da der bildbasierte Silhouettenalgorithmus mit Texturen arbeitet, die auf ein bildschirmfüllendes Rechteck gelegt werden, bekommt man in diesem Fall bei einem Umschalten in den Wireframe-Modus lediglich die zwei Dreiecke zu sehen, aus denen das Rechteck aufgebaut ist. Da diese Darstellung wenig sinnvoll ist, wurde verhindert, dass bildbasierte Silhouetten und Wireframe-Modus im Programm gleichzeitig aktiv sein können.

3.5 Maussteuerung

Das dargestellte Modell lässt sich mit Hilfe einer Maussteuerung, die mit der DXUT-Klasse `CModelViewerCamera` realisiert wurde, von allen Seiten betrachten. Gooch [6] beschreibt in ihrer Master-Arbeit über das Cool-



(a) Ohne Filter (b) Gauss-Filter

Abbildung 11: Vergleich von Toon Shading ohne und mit Weichzeichnung

to-Warm Shading auch das Problem der Interaktion mit dem Dargestellten. Um eine Betrachtung von allen Seiten zu ermöglichen, kann entweder das Modell selbst gedreht werden, oder die Kamera kann um das Modell herum bewegt werden. Eine Rotation des Modells kann den Betrachter verwirren, da sich das Shading aller Teile stets verändert. Bewegt man dagegen die Kamera, so geht insbesondere der Vorteil des Cool-to-Warm Shading verloren. Je nach Position der Kamera kommt es dann nicht mehr zu einem Übergang zwischen kalten und warmen Farben. Bei dem implementierten Programm wurde dem Benutzer durch die Belegung beider Maustasten selbst die Wahl überlassen, ob das Modell rotiert werden soll (linke Maustaste), oder, ob sich die Kamera um das Objekt bewegen soll (rechte Maustaste). Mit dem Mausrad kann darüber hinaus die Entfernung zwischen Kamera und Modell reguliert werden.

4 Ergebnisse

Im Rahmen dieser Studienarbeit habe ich mich erstmals intensiv mit Direct3D und Cg beschäftigt. Bei der Einarbeitung stellten sich die zahlreichen Beispielprogramme des DirectX SDK und des NVIDIA SDK als besonders hilfreich heraus. Darin sind zwar bislang noch keine Demos zum Thema Non-Photorealistic Rendering vorhanden, aber es ließ sich vieles zur Verwendung des DXUT und dem Einsatz von Cg, beziehungsweise der HighLevel Shading Language, in Kombination mit Direct3D als Anschauungsmaterial verwenden. Leider erwiesen sich besonders die Demo-Programme von Microsoft als unübersichtlich, deren teils 2000 Zeilen langer Quellcode oft nur in einer Datei untergebracht ist. In dieser Hinsicht wurde durch den softwaretechnischen Entwurf des hier vorgestellten Programms versucht einen anderen Weg einzuschlagen.

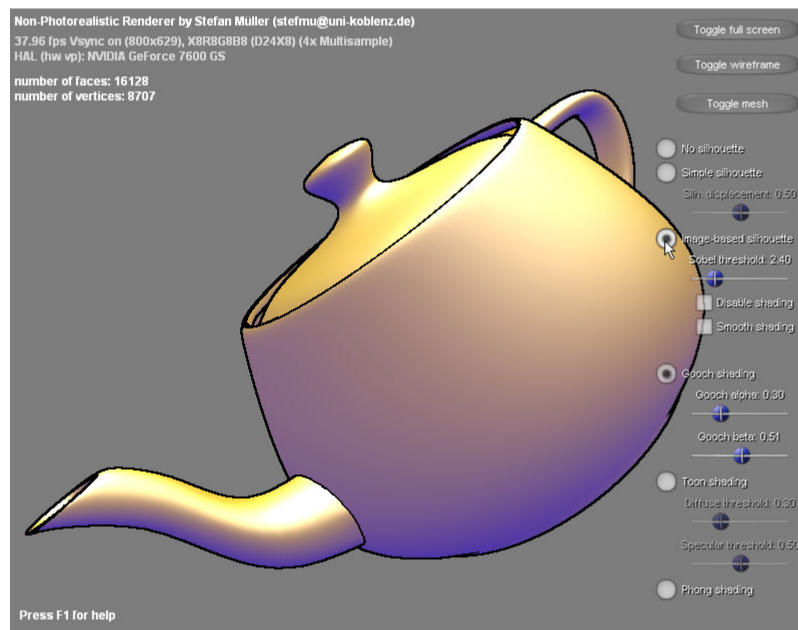


Abbildung 12: Die graphische Benutzerschnittstelle

4.1 Bilder

Abbildung 12 zeigt ein Bildschirmphoto des Programms, auf dem die mit dem DXUT implementierte GUI zu sehen ist.

Im oberen linken Bereich des Fensters sieht man allgemeine Informationen zum Programm. Neben dem Namen der Anwendung handelt es sich dabei um die aktuelle Darstellungsgeschwindigkeit in Frames pro Sekunde

(fps), die aktuelle Größe des Fensters in Pixel, die Farbtiefe und den Namen der Graphikkarte, die gerade zum Einsatz kommt. Direkt darunter werden die Anzahl der Polygone und der Eckpunkte des aktuell dargestellten Modells ausgegeben.

Auf Wunsch wird im unteren linken Bereich eine Hilfe zur Steuerung angezeigt. Dabei handelt es sich vor allem um die Belegung der Maustasten.

Auf der rechten Seite des Fensters befinden sich Schaltflächen um zu einer Anzeige im Vollbild-Modus zu wechseln, um eine Wireframe-Darstellung zu aktivieren und um ein anderes 3D-Modell anzeigen zu lassen. Darüber hinaus befinden sich hier alle GUI-Elemente, mit denen der eingesetzte Silhouettenalgorithmus und das Shading gewechselt werden kann. Alle Regler lassen sich entsprechend des Zustandsautomaten aus Abbildung 10 nur betätigen, wenn sich das Programm im jeweiligen Zustand befindet. Alle GUI-Elemente, die sich zum aktuellen Zeitpunkt nicht betätigen lassen, werden grau dargestellt.

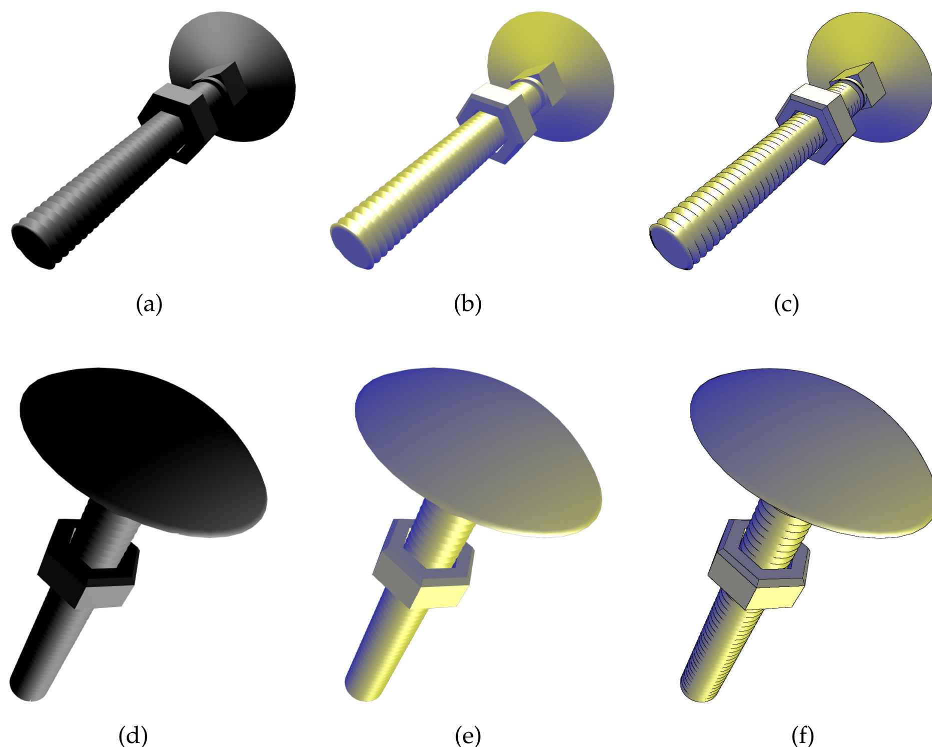
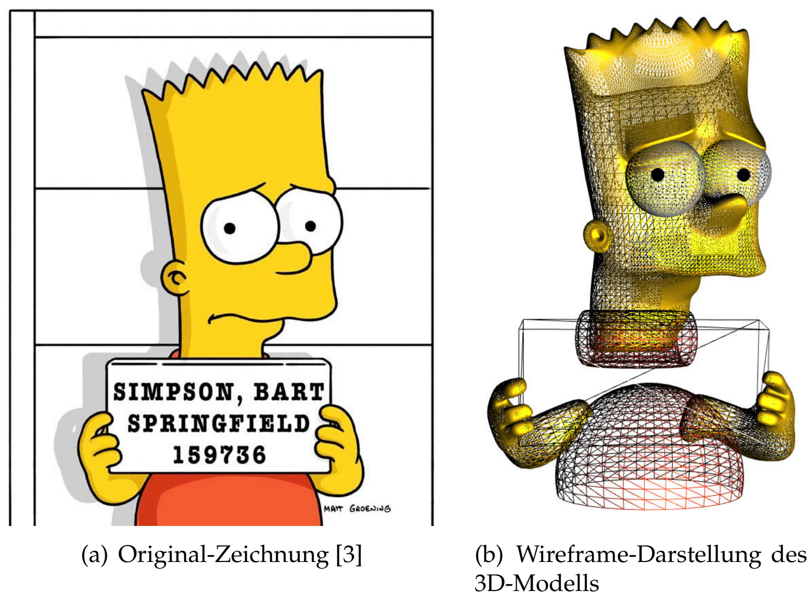


Abbildung 13: 3D-Modell einer Schraube. (a) und (d) Phong Shading. (b) und (e) Cool-to-Warm Shading ohne Silhouette. (c) und (f) Cool-to-Warm Shading mit bildbasierter Silhouette.

Abbildung 13 zeigt das 3D-Modell einer Schlossschraube mit Mutter, das von dem implementierten Programm mit Phong Shading und Cool-to-Warm Shading gerendert wurde. Das Modell wurde von mir mit der Software Maya modelliert. Um den Datensatz als X-Datei zu speichern, wurde ein Maya-Plugin eingesetzt, das Teil des DirectX SDK ist. Die Bilder sollen zeigen, welche Vorteile das Cool-to-Warm Shading gegenüber dem Phong Shading besitzt. Da sich Gooch et al. [7] mit dem Cool-to-Warm Shading an der Darstellungsweise technischer Illustrationen orientieren, wurde die Schraube als Modell gewählt. Mit dem Gewinde, der kantigen Mutter und dem runden Kopf der Schraube, enthält das Modell einige typisch mechanische Formen. Vor allem die Form der Mutter lässt sich mit dem Cool-to-Warm Shading besser ausmachen als mit dem Phong Shading. Der zusätzliche Einsatz von Silhouetten hilft bei der Wahrnehmung der Form ganz besonders.

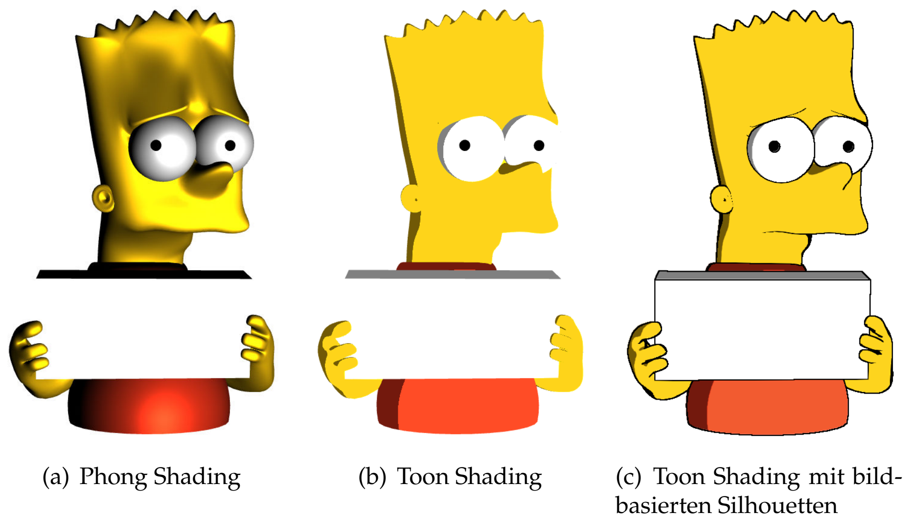


(a) Original-Zeichnung [3]

(b) Wireframe-Darstellung des 3D-Modells

Abbildung 14: Zeichnung von Bart Simpson als Vorlage für ein 3D-Modell

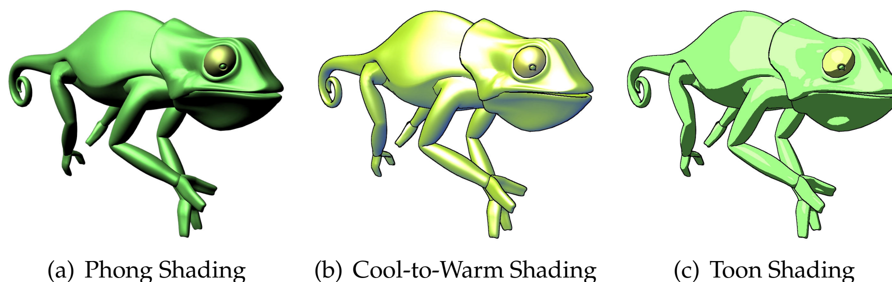
Um den Effekt des Toon Shadings zu verdeutlichen, habe ich mit Maya eine Zeichentrickfigur modelliert und mit dem implementierten Programm darstellen lassen. Abbildung 14(a) zeigt die Zeichnung von Bart Simpson, die als Vorbild für das 3D-Modell diente. Bei der Modellierung mit Maya wurden Subdivision Surfaces verwendet, die sich besonders gut für organische Formen eignen. Diese wurden vor dem Abspeichern im X-Dateiformat in Polygone konvertiert. Abbildung 14(b) zeigt das Drahtgittermodell der Figur, nachdem alle Subdivision Surfaces in Polygone umgewandelt worden sind. Man sieht auch, dass Bereiche der Figur, die auf dem Original-



(a) Phong Shading (b) Toon Shading (c) Toon Shading mit bildbasierten Silhouetten

Abbildung 15: „Bart Simpson“-Modell im Vergleich

bild nicht sichtbar waren, nicht modelliert wurden. Wie das gerenderte 3D-Modell mit Phong Shading und Toon Shading aussieht, zeigt Abbildung 15. Man sieht, dass die Figur mit Phong Shading deutlich plastischer wirkt, als mit Toon Shading. Das Toon Shading eignet sich dagegen besonders, um den Stil der Zeichentrickserie auf ein dreidimensionales Modell zu übertragen, das in Echtzeit dargestellt wird. Dass das Toon Shading nur in Kombination mit Silhouetten sinnvoll ist, zeigt Abbildung 15(b). Andernfalls sind Details im Gesicht, und somit auch der besorgte Ausdruck der Figur, aufgrund der gleichen Farben nicht auszumachen. Alle Unterschiede zwischen dem Original und der 3D-Darstellung aus Abbildung 15(c) lassen sich im Grunde auf eine falsche Positionierung der Lichtquelle, Unzulänglichkeiten des 3D-Modells oder perspektivische Fehler in der Zeichnung zurückführen.



(a) Phong Shading (b) Cool-to-Warm Shading (c) Toon Shading

Abbildung 16: „Chamäleon“-Modell im Vergleich

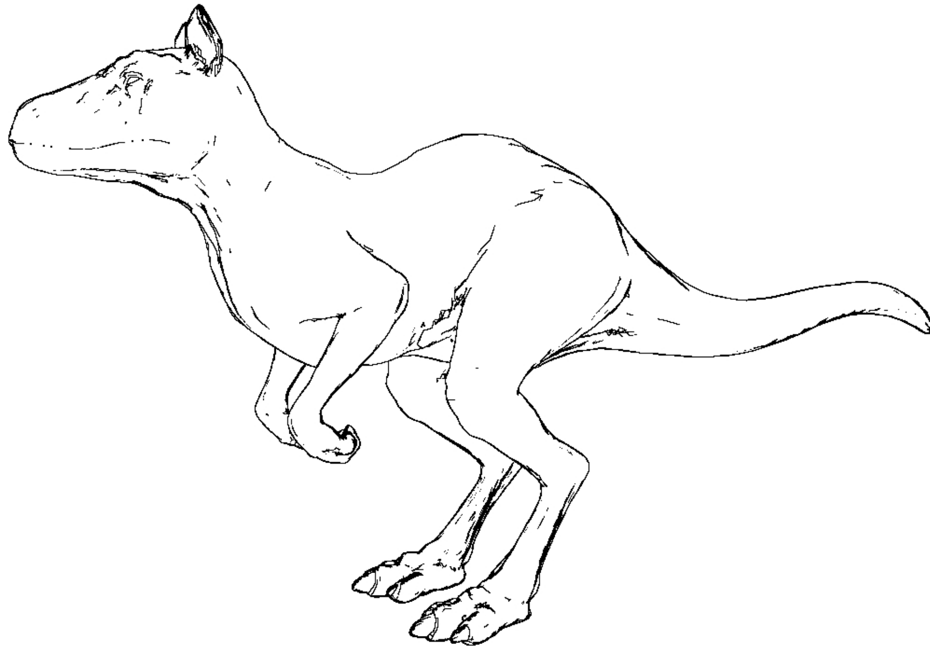
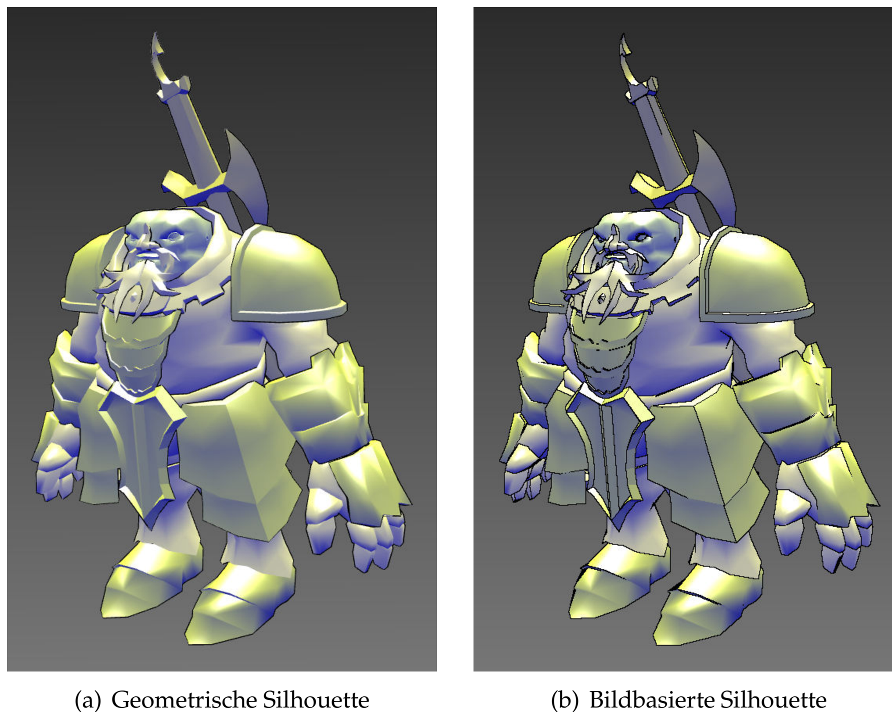


Abbildung 17: Bildbasierte Silhouette des „Killeroo"-Modells

Einen direkten Vergleich aller drei implementierten Shader liefert Abbildung 16. Cool-to-Warm Shading und Toon Shading werden typischerweise mit Silhouetten gezeigt.

Das Ergebnis des bildbasierten Silhouettenalgorithmus bei deaktiviertem Shading zeigt repräsentativ Abbildung 17. Diese Darstellungsform besitzt aufgrund ihrer Ähnlichkeit zu Zeichnungen eine eigene Ästhetik. Mit Kommentaren versehen könnte sie sich beispielsweise in der Anatomie als sinnvoll erweisen, insbesondere wenn der Betrachter durch die Verwendung von Farbe von wesentlichen Bildelementen abgelenkt würde, oder, wenn das Ausgabegerät nur eine eingeschränkt farbige Darstellung ermöglicht.

Abbildung 18 zeigt die beiden implementierten Silhouettenalgorithmen im Vergleich. Das gezeigte 3D-Modell ist Teil der Beispielprogramme des DirectX SDK. Es wurde ein dunkler Hintergrund hinter das Cool-to-Warm Shading gelegt, um das Modell räumlicher wirken zu lassen [6]. Da die Geometrie jedes Modells im X-Dateiformat in Untermengen mit eigenen Materialeigenschaften unterteilt wird, zeichnet der geometriebasierte Silhouettenalgorithmus neben dem Umriss teilweise auch Materialkanten ein. Das liegt daran, dass jedes Untermodell im ersten Renderdurchlauf des Verfahrens separat vergrößert wird. Die Methode der bildbasierten Silhouetten erkennt zwar keine Materialkanten, liefert aber durch die Darstellung



(a) Geometrische Silhouette (b) Bildbasierte Silhouette

Abbildung 18: Silhouettenalgorithmen im Vergleich an Modell „Dwarf“

von harten Kanten insgesamt ein besseres Ergebnis. Ein Nachteil des bildbasierten Verfahrens ist allerdings, dass Anti-Aliasing ohne einen Workaround nicht mehr ohne weiteres möglich ist.

4.2 Performanz

Decaudin [2] hat 1996 die Kombination von Toon Shading und bildbasierten Silhouetten auf einer Silicon Graphics Workstation implementiert. Das Rendern eines einzelnen Bildes hat damals etwa sechs Minuten in Anspruch genommen.[5]

Unter der Verwendung moderner Graphikhardware, sollten im Rahmen dieser Studienarbeit Verfahren des Non-Photorealistic Rendering in Echtzeit implementiert werden. Auch wenn das Programm nicht gezielt auf Geschwindigkeit hin optimiert wurde, soll die Performanz der Shader im Folgenden untersucht werden. Dadurch wird es möglich die Echtzeitfähigkeit der einzelnen Verfahren zu vergleichen.

Alle Werte wurden an einem PC mit AMD Athlon 64 X2 3800+ Prozessor, 2 GB RAM und einer NVIDIA GeForce 7600 GS Graphikkarte (512

[5]Das Programm von Decaudin bot zusätzlich Shadow Maps. Auch diese lassen sich mit heutigen Graphikkarten in Echtzeit berechnen und darstellen.

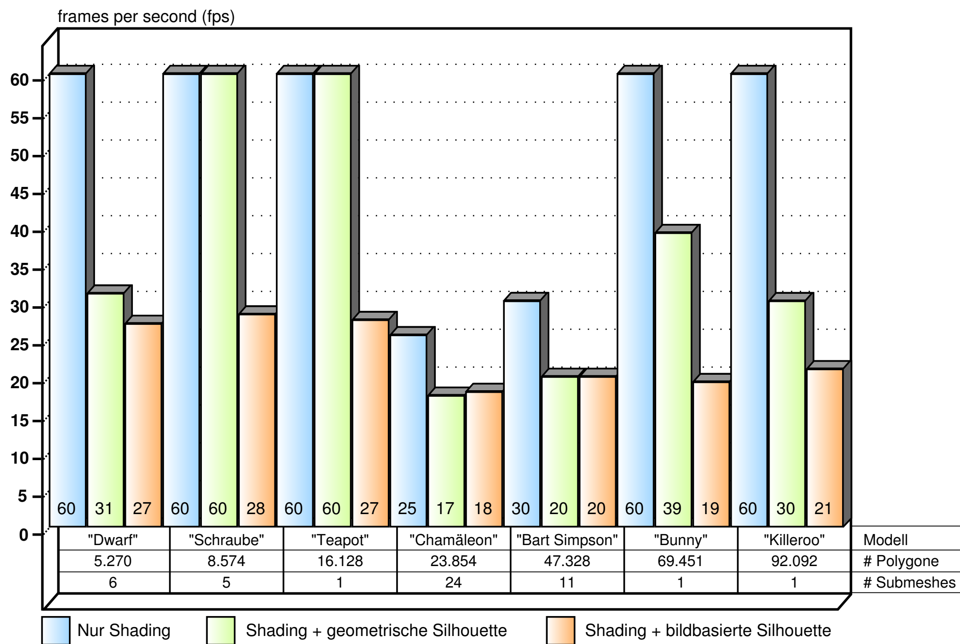


Abbildung 19: Darstellungsgeschwindigkeit der Shader

MB RAM) gemessen. Die Auflösung des Fensters betrug 1024×768 Pixel und über Direct3D wurde Anti-Aliasing durch vierfaches Multisampling für die gesamte Szene aktiviert.

Das Balkendiagramm in Abbildung 19 zeigt, wieviele Bilder pro Sekunde mit den verschiedenen Shadern dargestellt wurden. Alle Werte wurden über einen Zeitraum von 15 Sekunden gemittelt, in dem das 3D-Modell ununterbrochen mit der Maus bewegt wurde. Die Modelle sind nach aufsteigender Anzahl von Polygonen auf der X-Achse angeordnet.

Cool-to-Warm Shading, Toon Shading und Phong Shading wurden zusammengefasst, da sich in der Regel keine Unterschiede feststellen ließen. Die einzige Ausnahme bildet dabei das „Chamäleon"-Modell. Phong Shading ohne Silhouetten wurde hier mit 30 Frames pro Sekunde (fps), Toon Shading mit 25 fps und Cool-to-Warm Shading mit 20 fps dargestellt. Entsprechende Unterschiede traten auch mit aktivierten Silhouetten auf. Dies lässt sich nur auf die unterschiedlichen Instruktionen der einzelnen Fragment-Shader zurückführen, deren Ausführungsgeschwindigkeiten erst bei Geometrien mit einer größeren Anzahl von Untermengen divergieren.

Es fällt insgesamt auf, dass die Darstellungsgeschwindigkeit ebenso durch die Anzahl der Untermengen des Modells bestimmt wird, wie durch die Anzahl der Polygone, aus denen das Modell besteht. Dies erklärt sich dadurch, dass innerhalb der Render-Schleife alle Untermengen in einer For-Schleife durchlaufen werden. Bei jedem Durchlauf der Schleife werden

dabei neue Materialattribute an den aktiven Shader geschickt.

Der bildbasierte Silhouettenalgorithmus ist mit insgesamt vier Renderpasses und dem Filtern einer bildschirmfüllenden Textur besonders rechenintensiv. Allerdings hängt die Darstellungsgeschwindigkeit dieser Methode eher von der Auflösung des Fensters, als von der Anzahl der Polygone ab. Für das „Dwarf"-Modell konnten bei einer Auflösung von 250×250 Pixel 45 fps gemessen werden. Bei der vierfachen Auflösung von 500×500 Pixel waren es immerhin noch 40 fps, während bei 1000×1000 Pixel nur noch 20 fps erreicht wurden.

5 Fazit und Ausblick

In dieser Studienarbeit wurden Verfahren des Non-Photorealistic Rendering vorgestellt, die sich durch den Einsatz moderner Graphikhardware in Echtzeit darstellen lassen. Neben dem enormen Geschwindigkeitsvorteil bieten programmierbare Graphikkarten aber auch deutlich mehr Flexibilität. Ein aufwendiger Umweg, wie ihn beispielsweise Gooch et al. [7] beschreiben, um mit dem Phong-Beleuchtungsmodell der Fixed-Function-Pipeline das Cool-to-Warm Shading zu approximieren, ist mittlerweile unnötig geworden. Bei dem bildbasierten Silhouettenalgorithmus können die Werte der Normalen, durch den Einsatz von Shader-Sprachen, direkt den Farbkanälen zugewiesen werden. Ohne diese Funktionalität lässt sich ein Normalenbild nur über die Positionierung drei verschiedenfarbiger, achsenparalleler Lichtquellen auf jeweils einer der Koordinatenachsen des 3D-Raumes erzeugen.

Das Non-Photorealistic Rendering ist ein großes Forschungsfeld der Computergraphik und die hier vorgestellten Verfahren decken nur einen Bruchteil des Gebietes ab. Dementsprechend viele Erweiterungen des vorgestellten Programms sind denkbar und werden durch die modulare Architektur unterstützt.

Zu den interessantesten Shading-Algorithmen des Non-Photorealistic Rendering gehört sicherlich das von Praun et al. [13] beschriebene *Hatching*. Die Helligkeitsunterschiede des herkömmlichen Shadings werden hierbei durch unterschiedlich dichte Schraffuren ersetzt. Das Ergebnisbild wirkt dadurch, als wäre es von einem Künstler gezeichnet worden.

Beim *Charcoal Rendering* [10] handelt es sich um eine andere künstlerische Shading-Technik. Durch eine Kontrastverstärkung werden hierbei Kohlezeichnungen imitiert. Ebenso interessant wäre die Untersuchung von Verfahren, die unter dem Begriff *Painterly Rendering* zusammengefasst sind und Pinselzeichnungen nachahmen.

In der finalen Version des Programms werden nur die Modelle geladen, die zur Compile-Zeit im Quellcode angegeben wurden. Eine sinnvolle Erweiterung des Programms wäre daher die Möglichkeit, Modelle zur Laufzeit laden zu können. Dazu könnte die graphische Benutzerschnittstelle um ein Textfeld zur Eingabe der X-Datei erweitert werden.

Ein Vergleich zwischen dem implementierten, bildbasierten Silhouettenalgorithmus und einem Verfahren, das direkt auf der Geometrie operiert, wäre hinsichtlich Darstellungsqualität und Geschwindigkeit attraktiv. Vorraussetzung wäre allerdings eine Datenstruktur zur Verwaltung der Geometrie, auf die das vorgestellte Programm nicht ausgelegt ist. Unterschiedliche Stilarten für die Silhouetten — beispielsweise Bleistift oder Pinsel — wären zusätzlich eine sinnvolle Erweiterung des Programms.

Eine Untersuchung, wie die implementierten Verfahren auf eine ganze Szene angewandt zur Geltung kommen, steht zusätzlich aus. Für den

bildbasierten Silhouettenalgorithmus müssten hier Vorkehrungen getroffen werden, sodass bei Objekten im Hintergrund nicht die Silhouette über das Shading dominiert. Ein variabler Schwellwert, der sich der Entfernung zwischen Kamera und Objekt anpasst, könnte dazu einen ersten Ansatz darstellen.

Die Entwicklung der Graphikhardware geht weiterhin rasend schnell voran. Dennoch scheinen sich kaum neue Anwendungsgebiete zu erschließen, welche die Möglichkeiten der modernen Technik voll ausschöpfen. Photorealistische Computergraphik ist zwar unglaublich beeindruckend, aber nicht in allen Einsatzbereichen die optimale Lösung. In dieser Studienarbeit wurde gezeigt, dass sich programmierbare Graphikkarten auch auf andere Weise einsetzen lassen. Zum einen, um Kunstformen auf computergenerierte Szenen zu übertragen, und zum anderen, um uns die Wahrnehmung von Objekten in der Virtualität einfacher zu gestalten, als es uns in der Realität ermöglicht wird.

Literatur

[1] Tomas Akenine-Möller und Eric Haines. *Real-Time Rendering*. A K Peters, Ltd., 2002.

[2] Philippe Decaudin. Cartoon-Looking Rendering of 3D-Scenes. Technical Report 2919, INRIA, 1996.

[3] http://www.die-simpsons.de/pic-db/showpic.php?action=show&picid=3193&gid=&char=&order=&start=64&limit=5; Abruf: 21.03.2007.

[4] Randima Fernando und Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[5] Dag Frommhold. Spiele-Entwicklung für die Xbox 360. In */GameStar/dev - Magazin für Spiele-Entwicklung und Business-Development*, 4:18-20, November 2006.

[6] Amy Gooch. *Interactive Non-Photorealistic Technical Illustration*. Master's thesis, University of Utah, December 1998.

[7] Amy Gooch, Bruce Gooch, Peter Shirley and Elaine Cohen. A Non-Photorealistic Lighting Model For Automatic Technical Illustration. In *Computer Graphics Proceedings*, Annual Conference Series, pages 447-452. ACM SIGGRAPH, 1998.

[8] Nick Halper. *Supportive Presentation for Computer Games*. Doktorarbeit, Otto-von-Guericke-Universität, Magdeburg, Oktober 2003.

[9] Aaron Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In Stuart Green, editor, *ACM SIGGRAPH 1999 Course Notes*, ACM SIGGRAPH, ACM Press, 1999.

[10] Aditi Majumder and Meenakshisundaram Gopi. Hardware Accelerated Real Time Charcoal Rendering. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 59-66. ACM Press, 2002.

[11] Microsoft Corporation, *DirectX Documentation for C++*, 2006.

[12] Jason L. Mitchell, Chris Brennan and Drew Card. Real-Time Image-Space Outlining for Non-Photorealistic Rendering. SIGGRAPH, 2002.

[13] Emil Praun, Hugues Hoppe, Matthew Webb and Adam Finkelstein. Real-Time Hatching. In *Proceedings of SIGGRAPH 2001*, Computer Graphics, Annual Conference Series, pages 579-584, 2001.

[14] Peter Shirley. *Fundamentals of Computer Graphics*. A K Peters, Ltd., 2002.