UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Entwicklung einer Beispielapplikation zur Demonstration verschiedener nicht-photorealistischer Renderingverfahren

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Lena Kohl

Betreuer: Dipl.-Inform. Matthias Biedermann
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Oktober 2006

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

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Moderne Grafikhardware bietet heutzutage Möglichkeiten, nahezu photorealistische Bilder in Echtzeit zu generieren. Oftmals werden jedoch, aus verschiedenen Gründen, andere Darstellungsformen benötigt. Das sogenannte Nicht-Photorealistische Rendering erlaubt es verschiedene Mal- und Zeichentechniken zu simulieren, um realistische Grafiken unrealistisch darzustellen. Diese Grafiken bieten ein hohes Abstraktionspotential und sind dadurch im Stande wichtige Details in den Vordergrund und unwichtige Dinge in den Hintergrund zu stellen. Nicht-Photorealistisches Rendering wird zur Illustration sowohl für Computerspiele, als auch für technische oder medizinische Anwendungen verwendet, siehe Abbildung 1.

Abbildung 1: *Nicht-Photorealistische Illustrationen im Computerspiel (links)[NIN05] und in der Medizin (rechts)[WIM]*

1.2 Ziel

Das Ziel dieser Studienarbeit ist die Entwicklung einer Beispielapplikation verschiedener nicht-photorealistischer Renderingverfahren, um anschließend eine Auswertung hinsichtlich ihrer "künstlerischen"Verwertbarkeit zu erstellen.

1.3 Gliederung der Arbeit

Im zweiten Kapitel dieser Arbeit werden Dinge über das Nicht-Photorealistische Rendering erläutert, und mit Photorealistischem Rendering verglichen.

Das dritte Kapitel befasst sich mit der Programmierung auf der Grafikhardware, den verschiedenen Shadern (Vertex und Fragment Shader) und der Programmiersprache CG von NVIDIA.
Das vierte Kapitel beschreibt die Implementierung der Nicht-Photorealistischen Renderer, die im Laufe dieser Studienarbeit erstellt wurden.
Im fünften Kapitel werden die visuellen Ergebnisse der Arbeit vorgestellt und mit anderen Ergebnissen ähnlicher Projekte verglichen.
Im sechsten Kapitel wird ein Fazit erstellt und ein Ausblick auf weitere Möglichkeiten gegeben.

2 Nicht-Photorealistisches Rendering

2.1 Photorealistisches Rendering

Die Computergrafik beschäftigt sich seid über 35 Jahren hauptsächlich mit der Generierung von Bildern und Szenen die der Realität gleichen sollen und bei einem direktem Vergleich mit einem Foto nicht bzw. nur schwer unterschieden werden können.
Diese sogenannte photorealistische Computergrafik arbeitet hauptsächlich mit der Simulation von Licht [MUE05], da jedes Bild welches wir in unserer Umgebung aufnehmen, das Ergebnis von Licht ist, welches von der Lichtquelle emittiert wird, an Oberflächen der Umgebung reflektiert und letztendlich auf die Kamera, bzw. das Auge trifft. Zu diesem Zweck wurden verschiedene Techniken entwickelt um Licht so realitätsnah wie möglich zu simulieren. Mit Hilfe des sogenannten Raytracing und Radiosity wurden bemerkenswerte Ziele erreicht.
Doch Photorealismus läßt keinen Platz für Phantasie und Kreativität.

2.2 Nicht-Photrealistisches Rendering

Im Gegensatz zum Photorealistischen Rendering, versucht man beim Nicht-Photorealistischem-Rendering (NPR) Objekte so erscheinen zu lassen, dass sie nicht dem physikalisch korrektem Abbild eines Modells entsprechen. Das NPR ist im Gegensatz zum Photorealistischen Rendering ein noch Recht junges Forschungsgebiet der Computergrafik, jedoch wurde in den letzten Jahren auch in diesem Bereich viel erreicht.
Während photorealistische Bilder oft zur Darstellung von Architektur- und Industrie-Design benutzt werden, sind illustrierte, also nicht-photorealistische Bilder meistens effektiver. Diese können Informationen besser vermitteln und die Aufmerksamkeit auf relevante Details lenken, indem klare Linien und vereinfachte Umrisse verwendet, oder verdeckte Regionen freigestellt werden.
Der Vorteil von Illustrationen gegenüber Fotografien kann in vielen praktischen Kontexten nachvollzogen werden. So wird zum Beispiel in vielen medizinischen Texten zusätzlich zu Fotos ein handgezeichnetes Bild verwendet, da diese verdeckte oder kleine Strukturen besser beschreiben. Auch bei Gebrauchsanweisungen oder Bauanleitungen werden häufig Illustrationen den Fotos vorgezogen, auf Grund ihrer Klarheit.
Durch die Möglichkeiten mit dem Computer diese Illustrationen in 3D darzustellen, ermöglicht man dem Betrachter, durch Rotationen, Objekte von allen Seiten zu begutachten, ohne das man mehrere Bilder produzieren muß. Zusätzlich zur Nützlichkeit der nicht-photorealistischen Bilder, spricht auch der ästhetische Aspekt für sie. Mit Hilfe von NPR können mit dem Computer Bilder produziert werden, die an Ölgemälde von Künstler

wie van Gogh oder Cezanne erinnern, so wie in Abbildung 2.
Nicht-Photorealistisches Rendering umfasst jegliche Darstellungsform, die

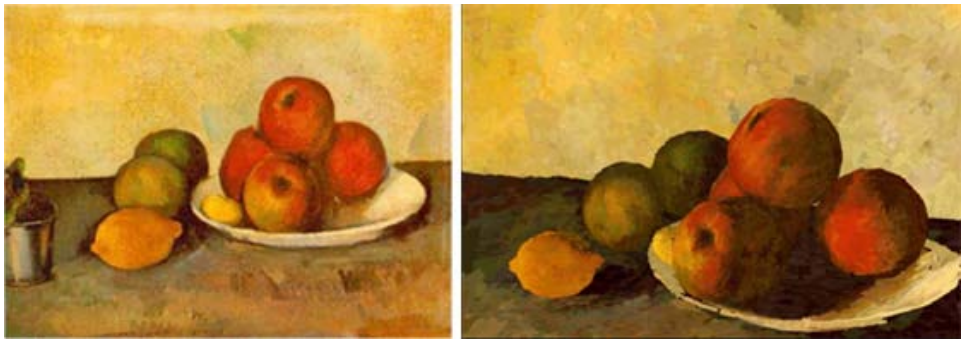

Abbildung 2: *links: Stillleben von Cezanne [CEZ], rechts: NPR-Replikation von Teece [TEE98]*

von der realen Wiedergabe abweicht. Nicht nur Painterly Rendering und Pen and Ink Shader, die man als erstes mit nicht-photorealistischen Bildern in Zusammenhang bringt, sondern, wie man in Abbildung 3 sieht, auch Geometrien mit nicht traditionellen Perspektiven, oder gar mehreren Perspektiven, wie zum Beispiel kubistische Bilder, Bewegungslinien in Bildern, oder auch einfach nur Silhouetten von Gegenständen gehören zum Gebiet des NPR.

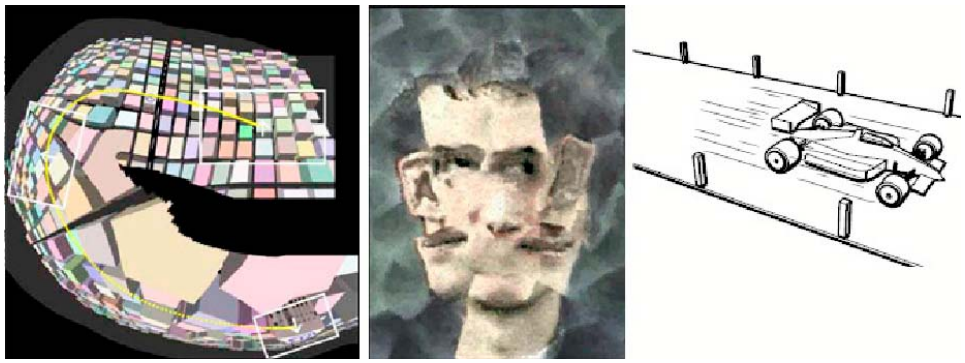

Abbildung 3: *links: Helikopterszene mit verzerrter Perspektive [WOO97], Mitte: kubistisches Bild, mit mehreren Perspektiven [COL02], rechts: Geschwindigkeitslinien in unbewegtem Bild [MAS99]*

3 GPU-Programmierung

Seit einigen Jahren ist die Entwicklung der Grafikhardware soweit vorangeschritten, dass sie vollständig programmierbar sind. Das sogenannte GPGPU[1] verwendet den Grafikprozessor für Berechnungen, für die er ursprünglich nicht entwickelt wurde. Die Entwicklung der programmierbaren Pipeline durch Vertex- und Fragment-Shader und die große Geschwindigkeit der GPU[2] ermöglichen es, Algorithmen, die zuvor von der CPU[3] bearbeitet wurden, nun auf der GPU auszuführen.
Während CPUs für mehrere Anwendungen, wie zum Beispiel dem Laufen des Betriebssystems und Anwendungsprogrammen zuständig sind, ist die GPU nur auf die Darstellung spezialisiert. Durch diese Spezialisierung ist es in Grafikaufgaben viel schneller als die CPU, die mehrere Zwecke hat. Neue GPUs können mehrere Milliarden Fragmente und einige Millionen Vertices pro Sekunde rasterisieren (zum Beispiel kann die NVIDIA GeForce 6800 Peak-Werte von 60 GFlops[4] erreichen, während ein Pentium4 12 GFlops erreichen kann [WIK]), und sie werden sogar noch immer schneller. Dies übersteigt bei Weitem die Zahl, die die CPU vollbringen würde. Durch diese Geschwindigkeit ist es nun möglich viele Grafikanwendungen in Echtzeit zu erzeugen und interaktiv zu gestalten.

3.1 Grafik-Pipeline

Die Grafik-Pipeline beschreibt beim (Echtzeit-)Rendering den Weg von der vektoriellen, mathematischen Beschreibung der 3D-Szene, bis zum gerenderten Bild in 2D auf dem Monitor.

3.1.1 Fixed-Function-Pipeline

Eine Pipeline ist eine Sequenz von verschiedenen Stufen, die parallel oder in einer festen Reihenfolge arbeitet. Jede Stufe erhält Eingabedaten und sendet ihre Ausgabe zu der nächsten Stufe. Die Grafik-Pipeline (auch Rendering-Pipeline genannt) arbeitet auf eine Menge von Vertices, geometrischen Primitiven und Fragmenten. Jedes Vertex beinhaltet eine Reihe an Informationen, wie seine Position, seine Farbe, seine Texturkoordinaten und einen Normalenvektor. Während des Rendering-Vorgangs durchlaufen die Objekte die Grafik-Pipeline, diese berechnet die zur Darstellung benötigten Umrechnungsschritte, so dass die 3D-Modelle auf dem Bildschirm dargestellt werden können. Die konventionelle Rendering-Pipeline hatte feststehende Abläufe, und arbeitete nur über die Hardware; daraus

[1] General Purpose Computation on Graphics Processing Units
[2] Graphics Processing Unit
[3] Central Processing Unit
[4] Flops = floating-point operation per second, GFlops = 10^9 Flops

resultiert auch der Name Fixed-Function-Pipeline[FER03].

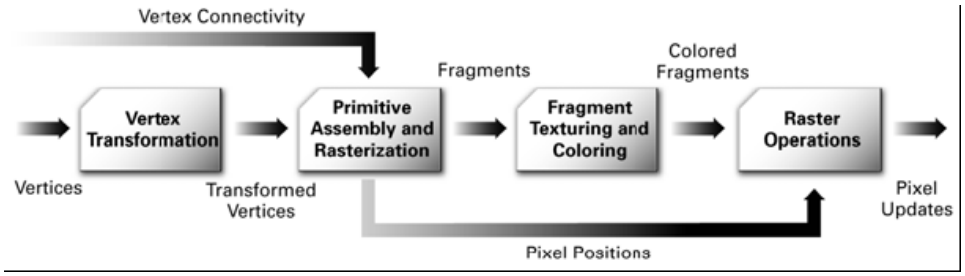

Abbildung 4: *Fixed-Function-Pipeline[FER03]*

Der erste Schritt der Fixed-Function-Pipeline ist die Vertex Transformation, siehe Abbildung 4. Hier werden mehrere mathematische Operationen auf jedes Vertex angewandt. Die Objektkoordinaten des Vertex werden umgerechnet in Clipkoordinaten damit die Rasterisierung, das Generieren von Texturkoordinaten und die Beleuchtung berechnet werden kann.
Die transformierten Vertices werden in der nächsten Stufe, der Primitve-Assembly-Stufe, zu geometrischen Primitiven, wie Dreiecke, Linien, oder Punkte, zusammengefasst. Außerdem müssen diese Primitive gegen eventuell gesetzte Clipping-Planes und das View-Frustum geclippt werden. Zusätzlich werden beim Backface Culling die Polygone verworfen, die dem Betrachter abgewandt sind. Die Polygone die dies überlebt haben, gelangen zur nächsten Stufe der Pipeline, der Rasterisierung. Hier werden die geometrischen Primitiven in die Position der Fragmente im 2D-Bild umgerechnet. Zusätzlich wird die Farbe und der Tiefenwert (Depth-Wert) für jedes Fragment durch Interpolation bestimmt.
Die nächste Stufe in der Pipeline berechnet die endgültige Farbe eines Fragments, indem die interpolierte Farbe des Vertex mit einer, eventuell angelegten, Textur kombiniert wird.
Die letzte Stufe die das Fragment zu durchlaufen hat, bevor es im Frame-Buffer aktualisiert wird, umfasst einige Rasteroperationen. Hier wird der Scissor-Test, der Alpha-Test, der Stencil-Test und der Depth-Test durchgeführt. Besteht ein Fragment einen dieser Tests nicht, dann wird es verworfen und nicht aktualisiert. Ansonsten wird noch eine Blending- und Dithering-Operation durchgeführt, und das Fragment wird an den Frame-Buffer übergeben.

3.1.2 Programmable-Pipeline

Seid einigen Jahren besteht die Möglichkeit einige Stufen der Fixed-Function-Pipeline durch programmierbare Abschnitte, den Vertex- und den Fragment-

Prozessor[FER03], zu ersetzten. Dadurch kann die Funktionalität der Rendering-Pipeline noch um neue Funktionen erweitert werden, und da alle Berechnungen direkt auf der GPU stattfinden, wird hierdurch viel Zeit gespart.

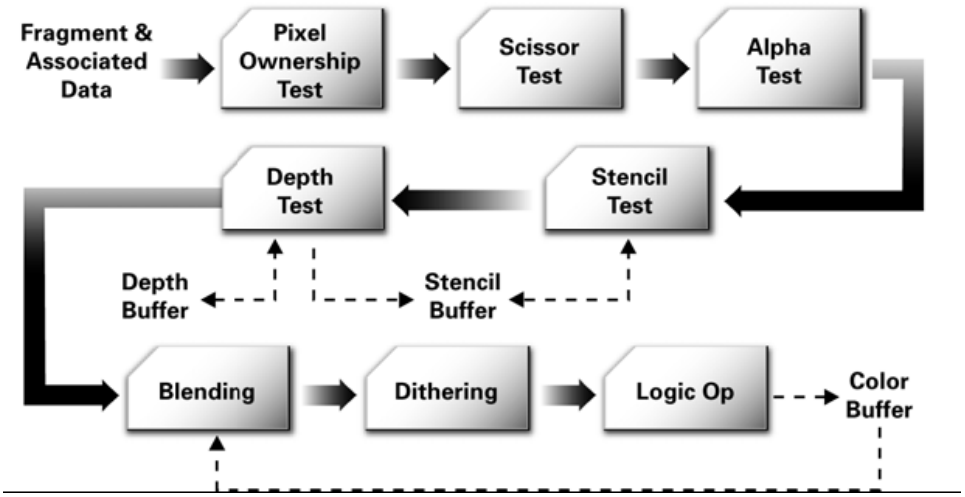

Abbildung 5: *Programmable-Pipeline[FER03]*

Der Vertex-Prozessor ersetzt die Vertex-Transformation (Abbildung 5), in dem er zuerst die Attribute jedes Vertices (Position, Texturkoordinaten, Farbe) lädt, und anschließend die Instruktionen des Vertex-Programms abarbeitet. Der Vertex-Prozessor ist meist auf eine begrenzte Anzahl von mathematischen Operationen beschränkt, wie zum Beispiel Addition, Multiplikation, Minimum, Maximum, Skalarprodukt und Kreuzprodukt.
Neuere Grafikkarten haben mehrere Vertex-Prozessoren, so dass verschiedene Aufgaben parallel nebeneinander herlaufen können.
Der Fragment-Prozessor ersetzt die Stufe der Texturierung der Fragmente. Im Fragment-Programm kann sowohl die Beleuchtung, als auch die Texturierung des Fragments berechnet werden, so dass letztendlich der endgültige Farbwert des Fragments bestimmt wird.

3.2 NVIDIA Cg

Durch die Entwicklung der programmierbaren Grafik-Pipeline wurde es erforderlich eine Programmierhochsprache zu entwickeln, die die neuen Möglichkeiten ausnutzen kann. NVIDIA hat im Jahr 2001 Cg[5] entwickelt. Der Name Cg weist auf die Nähe zur Programmiersprache C hin, auf dessen Syntax und Semantik Cg basiert. Dadurch ist es übersichtlicher zu lesen

[5] C for Graphics

und die Entwicklung von Shadern ist leichter, als in Assembler.
Cg Programme beliefern Programme der GPUs, aber sie benötigen Unterstützung von Anwendungen, um ein Bild zu rendern. Um Cg Programme mit einer Applikation zu verbinden muß man es für das entsprechende Profil kompilieren[FER03]. Dadurch ist das Cg Programm dann kompatibel für die 3D Schnittstelle, die von der Anwendung und der Hardware benutzt wird.
Cg ist weitgehend plattformunabhängig und kann sowohl OpenGL als auch DirectX als Schnittstelle verwenden.
Zusätzlich muß das Cg Programm noch mit dem Anwendungsprogramm verlinkt werden, so dass benötigte Parameter übergeben werden und das Programm ausgeführt werden kann.
Die Vertex- und Fragment-Programme basieren auf dem Stream-Prinzip, dass heißt, dass das Programm für jedes Vertex bzw. jedes Fragment aus dem jeweiligen Stream aufgerufen wird.
Neben Cg gibt es noch weitere Shader Sprachen, wie HLSL[6] und die OpenGL Shading Language, jedoch sind diese, im Gegensatz zu Cg, nicht unabhängig von der benutzten API.

3.2.1 Die Syntax von Cg

Der Sprachaufbau von Cg orientiert sich an der Programmiersprache C. Zusätzlich wurden einige Strukturen eingebaut, um den speziellen Fähigkeiten der GPU gerecht zu werden. Um Daten an ein Cg-Programm zu schicken, hat man zwei Möglichkeiten. Wenn sie für alle Vertices gleich sind benutzt man die sogenannten Uniform Inputs, wenn sie variieren die Varying Inputs. Einheitliche Eingabedaten werden in der Parameterliste der Funktion als **uniform** gekennzeichnet, variierende Daten erhalten ein zusätzliches Schlüsselwort, die sogenannten Binding Semantics.
Die meisten aus C bekannten Kontrollstrukturen, wie Schleifen und if-/else-Verzweigungen, findet man auch in Cg. Auch eigene Funktionen können definiert werden. Die elementaren Datentypen in Cg sind Gleitkommazahlen, Vektoren und Matrizen.
Die Rückgabewerte von Vertex- und Fragment-Programmen müssen erneut mit Hilfe von Binding Semantics gekennzeichnet werden. Die Ausgabe vom Vertex-Programm dient dabei als Eingabe des Fragment-Programms, so dass hier bei beiden die gleiche Struktur verwendet werden muß, um sie gemeinsam zu verwenden. Fragment-Programme erzeugen immer die Farbe des Ergebnis-Pixels und optional noch dessen Depth-Wert.

[6] High Level Shading Language

3.2.2 Profil

Da programmierbare Grafikkarten sich in ihren Fähigkeiten erheblich voneinander unterscheiden, muß bei Cg ein Profil gewählt werden, welches diese Unterschiede kontrolliert.
Das gewünschte Profil wird dem Compiler als Argument angegeben. Wenn das Programm im angefordertem Profil nicht kompilierbar ist, erhält man eine Fehlermeldung.

3.2.3 Cg Runtime Library

Um Cg Programme mit der Grafik-API zu verbinden und somit die Grafik-Pipeline anzusprechen, benötigt man die Cg Runtime Library (Laufzeit-Bibliothek). Diese arbeitet in zwei Schritten. Zuerst wird das Cg Programm mit dem gewählten Profil kompiliert, danach wird es mit der Applikation verknüpft, so dass diese später die Eingabedaten schicken kann.

4 Implementierung

Als Umgebung der implementierten Shader wurde mit Hilfe von OpenGL und GLUT ein Objekt geladen und dargestellt. Bei Benutzung der linken Maustaste hat man die Möglichkeit das Objekt zu rotieren. Diese Rotation wurde mit Hilfe von Polarkoordinaten[CG1/2] realisiert.

```
theta -= 0.01 * yVariation; // Bewegung in y-Richtung :
                            // theta aendern

if (theta < 0.01)
theta = 0.01;           // theta muss > 0 sein
else if (theta > PI - 0.01)
theta = PI - 0.01; // theta muss < PI sein

phi += 0.01 * xVariation; // Bewegung in x-Richtung :
                          // phi aendern
if (phi < 0)
phi += 2*PI; // falls < 0 : wieder bei 2PI anfangen
else if (phi > 2*PI)
phi -= 2*PI; // falls > 2PI : wieder bei 0 anfangen

float eyepos[3];
eyepos[0] = r * sin(theta) * cos(phi);
eyepos[1] =  r * cos(theta);
eyepos[2] = r * sin(theta) * sin(phi);
```

θ ist mit π, ϕ mit 2π initialisiert. Durch die Bewegung der Maus in x- und y-Richtung wird θ und ϕ verändert, jedoch dürfen sie nicht außerhalb ihres Wertebereichs liegen. Dieser ist für θ zwischen 0 und π und für ϕ zwischen 0 und 2π. Die neu berechneten Werte werden der Blickposition der Kamera zugewiesen.
Drückt man die rechte Maustaste, wird das Objekt entlang der Z-Achse transliert.

```
r += 0.05 * yVariation;
```

Um die Beleuchtungsrichtung zu verändern, kann man die mittlere Maustaste bzw. das Rädchen drücken. Die Rotation, die die Lichtquelle vornimmt, wird ebenfalls mit Polarkoordinaten erzeugt.
Dies sind die Grundfertigkeiten des Programms, um das Objekt navigieren zu können. Die weiteren Fähigkeiten sind mit Shadern implementiert und der eigentliche Hauptbestandteil dieser Studienarbeit. Um zwischen den verschiedenen Darstellungsformen umschalten zu können, wurde mit Hilfe einer switch-Anweisung eine Tastatureingabe implementiert:

- b - einfache Beleuchtung
- t - Toon Shader
- p - Pen and Ink Shader mit handgezeichneter Crosshatching Textur
- o - Pen and Ink Shader mit computergenerierter Crosshatching Textur
- i - Pen and Ink Shader mit handgezeichneter Stippling Textur
- u - Pen and Ink Shader mit computergenerierter Bricks Textur
- w - Wireframe Darstellung
- f - Texturierte Darstellung

4.1 Basic Light

Zu Beginn wird das Objekt realistisch beleuchtet (Abbildung 6). Die äußere Erscheinung eines Objektes ergibt sich aus ihren Materialeigenschaften und der Beleuchtung. Ein Beleuchtungsmodell beschreibt, wie Licht mit dem Objekt interagiert, basierend auf die Charakteristiken des Materials und des Lichts. Das am häufigsten verwendete Beleuchtungsmodell ist das Phong-Beleuchtungsmodell, welches für die Fixed-Function-Pipeline konzipiert wurde. Basic Light ist eine Erweiterung des Phong-Modells, schon auf Grund der Tatsache, dass es mit Hilfe eines Vertex-Shaders implemetiert ist. Im Phong-Modell wird die Farbe der Objektoberfläche berechnet, indem man die Summe der emissiven, ambienten, diffusen und spekularen Beleuchtungsanteile berechnet. Jeder dieser Anteile beruht auf den Eigenschaften

Abbildung 6: *realistisch beleuchteter Teapot*

der Oberfläche (wie z.B. Materialfarbe) und der Lichtquelle (wie z.B. Lichtfarbe und Position). Der emissive Anteil stellt den Teil des Lichtes dar, der von der Oberfläche abgestrahlt wird und ist somit unabhängig von der Lichtquelle. Der RGB-Wert, der den emissiven Anteil bezeichnet, gibt die Farbe des emittierenden Lichtes an. Wenn man ein Objekt in einem völlig abgedunkelten Raum sieht, dann hat das gesamte Objekt, genau diese Farbe.
Der ambiente Anteil stammt von dem Licht, das von überall aus der Szene kommt, es hat keine bestimmte Richtung und ist deshalb auch von der Lichtquelle unabhängig. Der RGB-Wert, der den ambienten Anteil bezeichnet, hängt vom Reflexionsgrad des Materials und der Farbe des einfallenden ambienten Lichts ab. Wenn ein Objekt nur durch den ambienten Anteil beleuchtet werden würde, dann hätte es, genau wie beim emissiven Anteil, genau eine Farbe. Der diffuse Anteil stammt von gerichtetem Licht, welches von der Oberfläche gleichmäßig in alle Richtungen reflektiert wird. Dieser Anteil ist abhängig von der Position der Lichtquelle.
Der spekulare Anteil stellt den Teil des Lichts dar, der von der Oberfläche gestreut wird. Anders als die anderen Werte, ist dieser abhängig von der Position des Betrachters. Wenn die Kamera keine reflektierenden Strahlen empfängt, wird der Betrachter keine spekularen Highlights auf der Oberfläche sehen können. Der spekulare Anteil wird nicht nur durch die spekularen Farbeigenschaften der Lichtquelle und des Materials beeinflusst,

sondern zusätzlich noch wie glänzend das Material ist. Glänzendere Materialien haben kleinere Glanzpunkte, wohingegen weniger glänzende Materialien ausgebreitetere Highlights aufweisen.
Die Kombination dieser Anteile ergibt die endgültige Beleuchtung:

$$surfaceColor = emissive + ambient + diffuse + specular$$

wobei

$$emissive = K_e$$

- K_e die Farbe des emissiven Materials ist,

$$ambient = K_a \times globalAmbient$$

und

- K_a der Reflexionsgrad des Materials, und
- $globalAmbient$ die Farbe des eintretenden, ambienten Lichts ist,

$$diffuse = K_d \times lightColor \times max(N \cdot L, 0)$$

und

- K_d die diffuse Farbe des Materials,
- $lightColor$ die Farbe des eintretenden diffusen Lichts,
- N die normalisierte Oberflächennormale,
- L der normalisierte Vektor Richtung Lichtquelle und
- P der betrachtete Punkt auf der Oberfläche ist,

$$specular = K_s \times lightColor \times facing \times (max(N \cdot H, 0))^{specular}$$

- K_s ist die spekulare Farbe des Materials,
- $lightColor$ ist die Farbe des eintretenden spekularen Lichts,
- N die normalisierte Oberflächennormale,
- V der normalisierte Vektor Richtung Kamera,
- L der normalisierte Vektor Richtung Lichtquelle,
- H der normalisierte Vektor der genau zwischen V und L liegt,

- P der betrachtete Punkt auf der Oberfläche und
- $facing$ ist 1 wenn $N \cdot L$ größer als 0 ist, und ansonsten 0.

Der Exponent $specular$ des Skalarprodukts von N und H gewährleistet, dass das spekulare Highlight schnell kleiner wird, wenn H und V näher zusammenrücken.
Hinzu kommt, dass der spekulare Anteil Null wird, wenn der diffuse Anteil Null ist, da $N \cdot L$ dann negativ ist. Dadurch wird gewährleistet, dass es keine Highlights gibt, wenn die Oberfläche dem Licht abgewandt ist.

```
//Emmisiven Anteil berechnen
float3 emissive = Ke;

//Ambienten Anteil berechnen
float3 ambient = Ka * globalAmbient;

//Diffusen Anteil berechnen
float3 L = normalize (lightPosition - P);
float diffuseLight = max(dot(N,L), 0);
float3 diffuse = Kd * lightColor * diffuseLight;

//Spekularen Anteil berechnen
float3 V = normalize(eyePosition - P);
float3 H = normalize(L+V);
float specularLight = pow(max(dot(N,H),0), shininess);
if (diffuseLight <= 0) specularLight = 0;
float3 specular = Ks * lightColor * specularLight;

color.xyz = emissive + ambient + diffuse + specular;
color.w = 1;
```

Im oben aufgeführten Vertex Programm gibt es für die Berechnung des emissiven Anteils nicht viel zu tun, dieser wird einfach von der OpenGL Applikation übergeben. Die Berechnung des ambienten Anteils wird mit Hilfe einer Vektormultiplikation der Materialfarbe und der ambienten Lichtfarbe durchgeführt. Diese Werte werden ebenfalls übergeben.
Um den Anteil der Farbe, den das diffuse Licht beisteuert, zu erhalten, muß zuerst noch der Vektor vom Vertex zur Lichtquelle L berechnet werden. Da nur die Richtung interessant ist, wird er direkt normalisiert. Mit diesem Vektor kann nun durch ein Skalarprodukt die diffuse Farbe des Lichts berechnet werden; wobei Oberflächen die der Lichtquelle abgewandt sind ein negatives Ergebnis liefern würden, so dass dieser Fall noch abgefangen wird. Für die spekulare Berechnung benötigt man noch den Vektor vom

Vertex zur Kamera, den sogenannten Viewvektor V, und den Vektor der mittig zwischen L und V liegt, H. Danach wird, genau wie bei dem diffusen Anteil, das Skalarprodukt genommen und mit Hilfe der **pow**-Funktion von Cg, $shininess$ als Exponent verwendet. Zuletzt werden zur Berechnung der endgültigen Vertex-Farbe, die einzelnen Vektoren zusammengerechnet.
Da das Vertex Programm die Beleuchtungsberechnung bereits vornimmt, hat das Fragment Programm nur die Aufgabe, die interpolierte Farbe an den Framebuffer zu übergeben.

4.2 Toon Shader

Toon Shading ist eine Rendering Technik die Objekte mit konstanten, scharf von einander abgegrenzten Farben shadet, so dass diese erscheinen, als wären sie aus einem Cartoon und keine realen Objekte.
Der Toon Shader reduziert die für die reale Beleuchtung notwendigen Lichtbestandteile, auf drei wichtige Hauptkomponenten:
Für die diffuse Beleuchtung benötigt man nur noch zwei Farben, eine für helle Bereiche und eine für dunkle Bereiche. Jeder Pixel hat ursprünglich eine Farbe die zwischen 0.0 und 1.0 liegt. Die Werte variieren von vollständig beleuchtet (1.0) zu vollständig unbeleuchtet (0.0). Um diese Werte aufzuteilen, so dass nur noch zwei Farben auftauchen, wählt man einen Schwellwert zwischen 0 und 1. Die Werte die unterhalb dieses Wertes liegen, gelten als unbeleuchtet und werden somit mit der dunklen Farbe eingefärbt. Die Werte überhalb des Schwellwertes werden mit der hellen Farbe eingefärbt. Diese *Treppenfunktion* erzeugt somit ein zweifarbiges Objekt.
Der Toon Shader verwendet hierfür eine eindimensionale Textur, die in Abbildung 7 angezeigt wird. Der diffuse Teil des Toon Shaders benutzt das

Abbildung 7: *1D Textur für diffuse Beleuchtung des Toon Shaders*

Skalarprodukt des Normalenvektors N und des Vektors Richtung Lichtquelle L um zu bestimmen, ob der Bereich hell oder dunkel ist.

```
//berechne diffuse Beleuchtung
float3 N = normalize (normal);
float3 L = normalize (lightPosition - position.xyz);
diffuseLight = max (dot (N, L), 0);
```

Die spekularen Highlights werden mit einer einzigen Farbe gefärbt. Um zu berechnen, wann genau der Fall eintritt, dass ein Highlight gesetzt werden muß, geht man wie bei der diffusen Beleuchtung vor. Man verwendet eine Treppenfunktion, doch an Stelle der Entscheidung welche Farbe gesetzt

wird, entscheidet hier die Treppenfunktion, ob ein Highlight gesetzt wird, oder nicht. Die Textur, Abbildung 8, ist wiederum eindimensional, jedoch nur einfarbig.

Abbildung 8: *1D Textur für spekulare Highlights des Toon Shaders*

```
//berechne spekulare Beleuchtung
float3 V = normalize (eyePosition - position.xyz);
float3 H = normalize (L + V);
specularLight = pow (max (dot (N, H), 0), shininess);
if (diffuseLight <= 0)
specularLight = 0;
```

Zum Schluss erhält das Objekt noch Silhouetten. Dafür muß man Pixel finden die das Objekt umranden und diese mit einer gewählten Umrandungsfarbe einfärben.
Um diese Pixel zu finden muß man das Skalarprodukt von N und des Vektors Richtung Kamera V nehmen. Dadurch erhält man die Information, wieviel der Oberfläche vom momentanen Blickpunkt sichtbar ist. Wenn das Ergebnis des Skalarprodukts positiv ist, ist der Punkt in einem sichtbaren und wenn es negativ ist, in einem verdeckten Bereich. Punkte für die das Skalarprodukt nahe an der Null sind, repräsentieren Übergänge zwischen den sichtbaren und den verdeckten Bereichen. Diese Punkte sind die gesuchten Silhouetten.

```
//Kantendetektion
edge = max (dot (N, V), 0);
```

Die dazugehörige eindimensionale Textur ist in Abbildung 9 dargestellt. Diese Berechnungen werden im Vertex Programm aufgeführt.

Abbildung 9: *1D Textur für Silhouetten des Toon Shaders*

Das Fragment Programm des Toon Shaders Berechnet die Farbe des Vertex.

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

```
color = edge * (Kdif * diffuseLight
             + Kspec * specularLight);
```

4.3 Pen and Ink Shader

Beim Pen and Ink-Shading geht es um die Simulation von Bleistift- bzw. Tusche-Zeichnungen. Solche Illustrationen haben nur zwei Farben zur Verfügung (Schwarz und Weiß), so dass es schwieriger ist, unterschiedliche Farben und Helligkeiten darzustellen. Dies wird mit Hilfe verschiedener Kombinationen von Strichen gemacht, siehe Abbildung 10. Die häufigst verwendete Methode ist das sogenannte Cross Hatching, bei der annähernd parallele Linien verwendet werden. Durch Überlagerung mit ebenfalls parallel liegenden Querlinien in unterschiedlichen Richtungen, können verschiedene Helligkeiten erreicht werden. Eine weitere Methode ist, das Stippling. Hier wird mit kleinen Punkten, die pro Helligkeit eine andere Dichte haben, gearbeitet.

Der Pen and Ink Shader arbeitet ähnlich wie der Toonshader. Er führt die

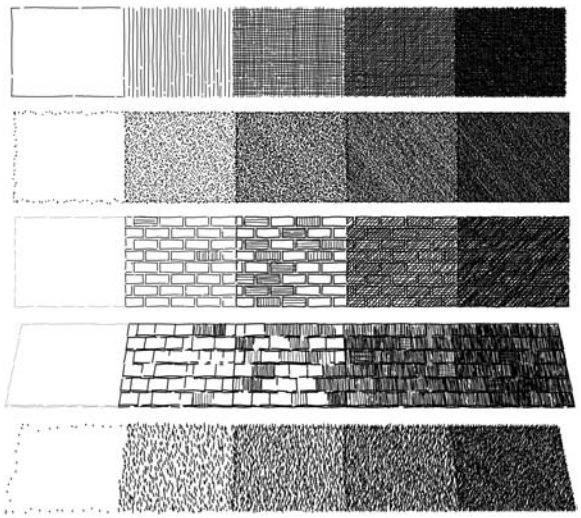

Abbildung 10: *Verschiedene Texturen nach [WIN94], von oben nach unten "Cross-Hatching", SStippling", "Bricks", SShingles", "Grass", die für Pen and Ink Rendering geeignet sind*

gleichen Berechnungen durch, um Helligkeiten zu bestimmen und Silhouetten zu zeichnen. Jedoch arbeitet er mit einer zweidimensionalen Textur, die in fünf Bereiche aufgeteilt ist (Abbildung 11), an Stelle von drei eindimensionalen Texturen.

Die benötigten normalisierten Vektoren, der Lichtvektor L und der Normalenvektor N werden berechnet und mit ihnen der Anteil des diffusen Lichts bestimmt. Danach wird der normalisierten Viewvektoren V und der

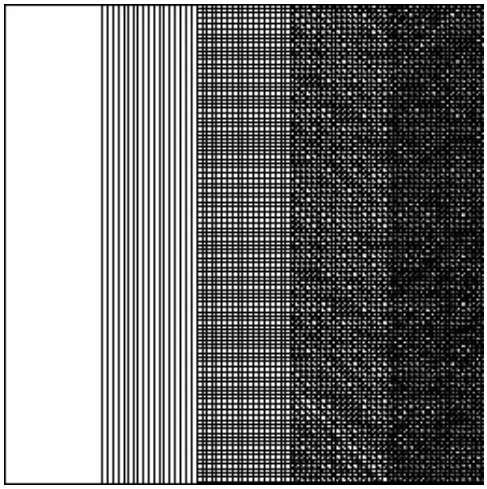

Abbildung 11: *Cross Hatching Textur des Pen And Ink Shaders*

Vektor der mittig zwischen L und V liegt, H, berechnet. Dann wird genauso wie bei der realistischen Beleuchtung, der spekulare Anteil des Lichts berechnet.
Nachdem sowohl die diffuse Beleuchtung, als auch die spekulare Beleuchtung berechnet wurde folgt die eigentlich wichtige Berechnung des Pen and Ink Shaders; die Aufteilung der Texturbereiche, auf die verschiedenen Helligkeiten des Objekts. Dies wird mit Hilfe von if-Abfragen im Vertex-Programm kontrolliert.

```
if( edge<0.2 ){
open.x = pen.x * 0.2 + 0.8; // Wertebereich von 0.8-1.0
}
else if (edge >= 0.2 && edge < 0.4){
open.x = pen.x * 0.2 + 0.6; //Wertebereich von 0.6-0.8
}
else if (edge >= 0.4 && edge < 0.6){
open.x = pen.x * 0.2 + 0.4; //Wertebereich von 0.4-0.6
}
else{
open.x = pen.x * 0.2 + 0.2; //Wertebereich von 0.2-0.4
}
```

Um die Highlights zu erzeugen, wird eine extra Abfrage gemacht.

```
if(specularLight>0.2 ){
open.x = 0.1;
}
```

Hierdurch wird gewährleistet, dass sich die Highlights bei jeder Aktualisierung der Blickposition, also des Viewvektors V und somit auch des $specularLight$, welches ja mit dessen Hilfe berechnet wird, neu berechnen. Da bisher nur der x-Wert der Textur benutzt wurde, wird zuletzt noch der y-Wert bestimmt.

```
open.y = pen.y*0.4 + 0.05;
```

Das Fragment-Programm berechnet die endgültige Farbe und gibt dies dann an den Frame Buffer weiter.

4.3.1 Verschiedene Pen and Ink Variationen

Zusätzlich, zu dem oben bereits erläutertem Cross Hatching gibt es noch weitere Variationen des Pen and Ink Shaders.
Anstelle der Linien-Textur des Cross-Hatchings, wird beim Stippling mit

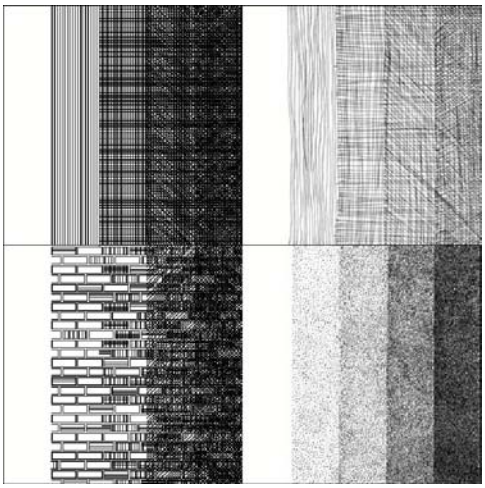

Abbildung 12: *Verschiedene Texturen des Pen And Ink Shaders, oben links: mit dem Computer generierte Cross Hatching Textur, oben rechts: von Hand gezeichnete Cross Hatching Textur, unten links: mit dem Computer generierte Bricks Textur, unten rechts: von Hand gezeichnete Stippling Textur*

Punkten gearbeitet und bei Bricks mit gezeichneten Backsteinen. Da diese Variationen grundsätzlich nur eine andere Textur verwenden, die in Abbildung 12 aufgezeigt werden, aber ansonsten genauso arbeitet wie der Cross-Hatching Pen and Ink Shader, muß auf die Implementierung nicht weiter eingegangen werden.

5 Ergebnisse

Der hier implementierte Toonshader liefert gute Ergebnisse. In Abbildung 13 sieht man, dass das Objekt insgesamt nur drei Farben hat, die diffuse Grundfarbe des Objekts in beleuchteten und unbeleuchteten Bereichen und das Highlight. Auch die geforderten Silhouetten sind vorhanden.
Das Ergebnis ist vergleichbar mit den Ergebnissen (siehe Abbildung 14)

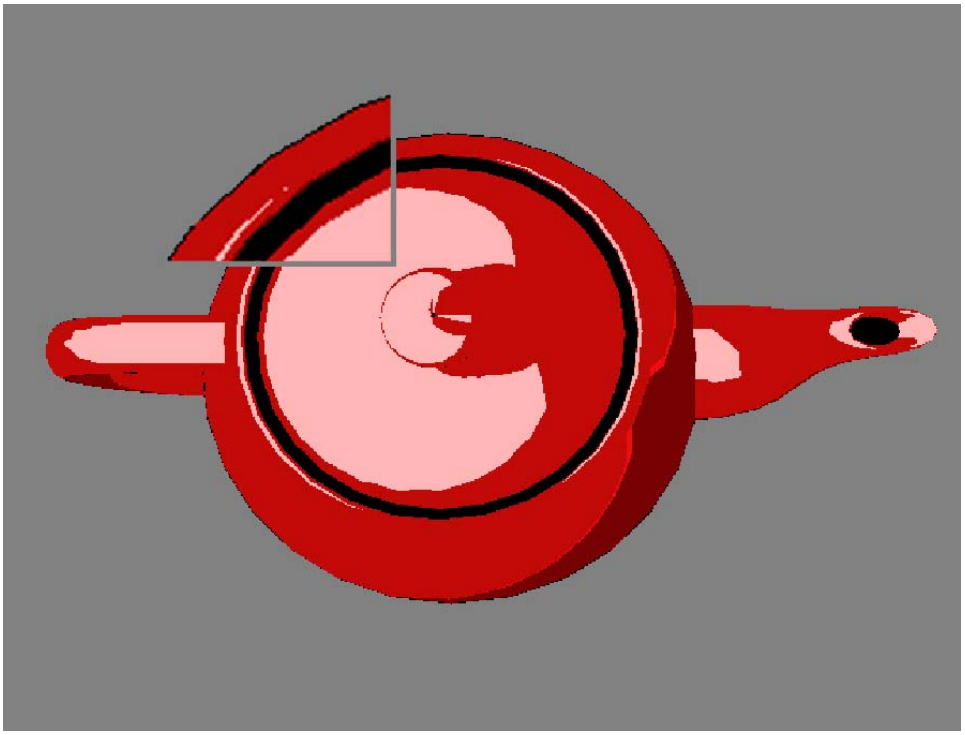

Abbildung 13: *Objekt mit Toon Shader*

des Toonshaders nach [LAK00].
Um das Ergebnis richtig bewerten zu können, wurde einem Comiczeichner beauftragt nach Vorlage des realistisch gerenderten Teapots aus Abbildung 6 ein Cartoon zu erstellen. Ihm wurden keine weiteren Anweisungen gegeben, so dass man an Hand des Ergebnisses sehen kann, wo die Unterschiede zwischen dem vom Computer erzeugten und dem von Hand gemalten Cartoon sind. Da auf Anweisungen Verzichtet wurden, ist das Bild 15 in schwarz-weiß erstellt worden. Doch trotz allem erkennt man den Unterschied zwischen den computergenerierten und den gezeichneten Versionen. Das vom Computer berechnete Ergebnis hat keine Fehler, weder in der Form des Teapots, noch in den Positionen der Lichtflecke. Doch genau diese kleinen Fehler, wie die nicht ganz runden Rundungen und den

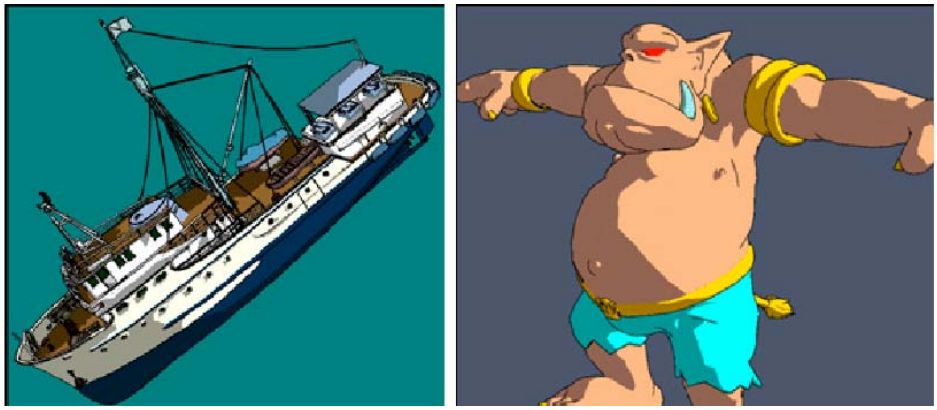

Abbildung 14: *links: Schiffmodell, rechts: Olaf in Cartoon Shading, nach [LAK00]*

Teilweise verschobenen Perspektiven, bringen Charakter in ein Cartoon.
Der klare Vorteil des Programms liegt jedoch darin, dass das Modell drei-

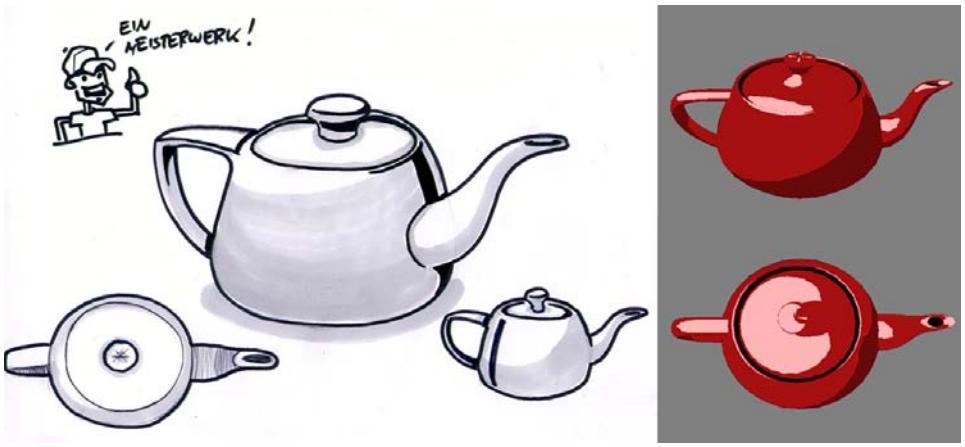

Abbildung 15: *vom Comiczeichener erstellte Bilder des Teapots mit Vergleichsbildern des Programms*

dimensional ist und navigiert werden kann. Der Comiczeichner müsste, um eine neue Perspektive oder andere Lichtverhältnisse zu erzeugen, ein neues Bild malen.
Für den Pen and Ink Shader wurden vier Texturen erstellt.
Die Cross Hatching Methoden liefern relativ gute Ergebnisse. Jedoch tauchen hier einige Artefakte auf, die mit der Auflösung der Textur zusammenhängen. Dies fällt sowohl bei der computergenerierten, als auch bei der von Hand gezeichneten Textur auf, siehe Abbildung 16. Bei der Stippling und der Bricks Methode sind die Ergebnisse ähnlich zu denen der Cross

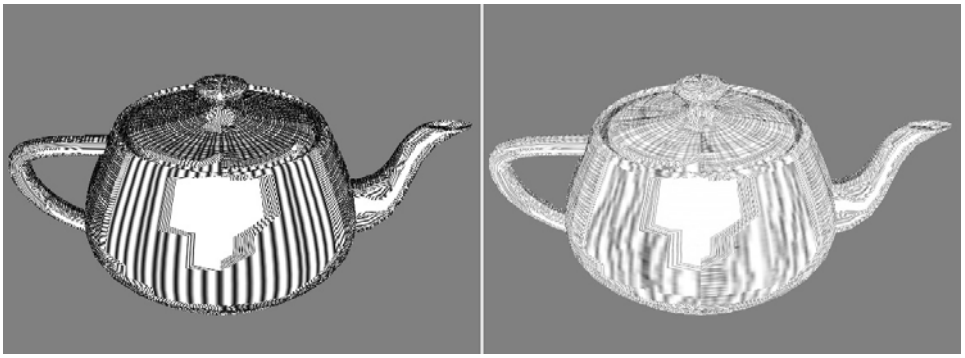

Abbildung 16: *Teapot mit Cross Hatching Pen and Ink Shader, links: mit handgezeichneter Textur, rechts: computergenerierte Textur*

Hatching Methode, vergleiche mit Abbildung 17. Zu vergleichen sind die-

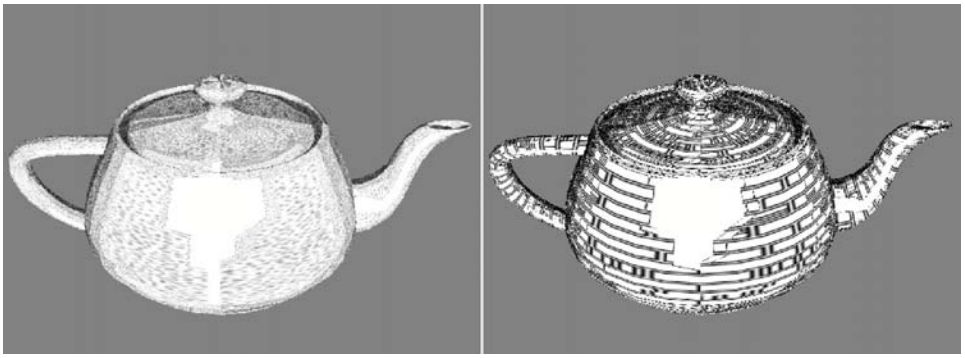

Abbildung 17: *links: Teapot mit Stippling Pen and Ink Shader, rechts: Teapot mit Bricks Pen and Ink Shader*

se Methoden mit den Ergebnissen nach[WIN94], wobei in Abbildung 18 verschiedene Texturen gleichzeitig verwendet wurden, was bei dem Pen and Ink Shader dieser Studienarbeit nicht möglich ist.

6 Fazit und Ausblick

In dieser Studienarbeit wurden drei verschiedene Shader implementiert. Der erste zeigt das Objekt bei realistischer Beleuchtung, die beiden anderen sind Nicht-Photorealistische Shader.
Der Toonshader liefert zufrieden stellende Ergebnisse. Im Vergleich zu dem in Auftrag gegebenen Bild, erkennt man jedoch, welche Vorzüge von Hand erstellte Cartoons haben. Man könnte in Zukunft versuchen, solche Unregelmäßigkeiten in der Form des Objektes automatisch durch Zufallsgene-

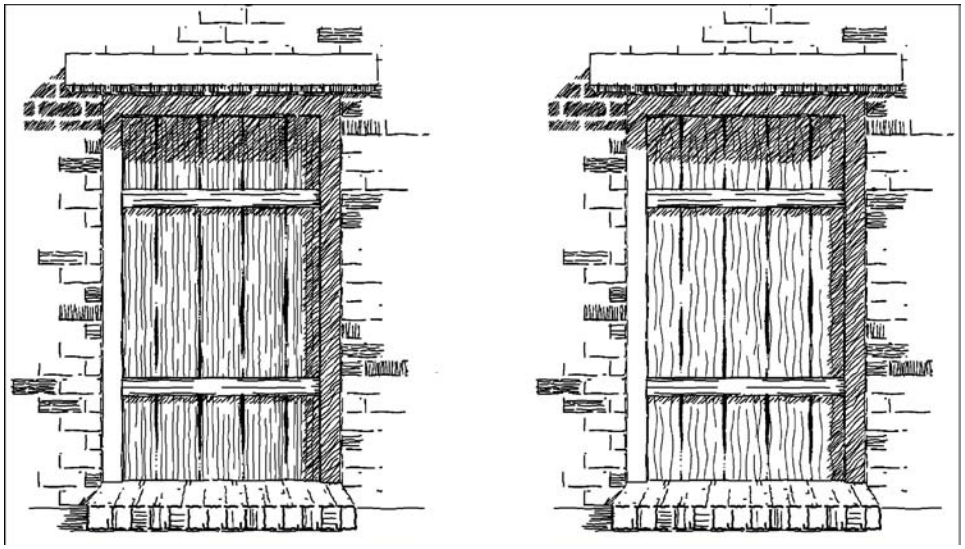

Abbildung 18: *Variation verschiedener Pen and Ink Shader nach [WIN94]*

ratoren zu erzeugen.
Da der Pen and Ink Shader leicht mit verschiedenen Texturen neue Ergebnisse erzeugt, wurde hier sowohl das Cross Hatching, als auch das Stippling Verfahren getestet. Die Texturen könnten automatisch generiert werden, wodurch eine größere Regelmäßigkeit gewährleistet wäre, als bei den hier genutzten, von Hand gezeichneten.
Zusätzlich wäre es für die Qualität der Ergebnisse vorteilhaft, wenn man verschiedene Texturen auf ein Objekt anwenden könnte, so dass wie in Abbildung 22 stimmigere Ergebnisse entstehen.
Für die Zukunft können weitere Nicht-Photorealistische Shader, wie zum Beispiel ein Painterly Renderer oder ein kubistischer Shader, dem System hinzugefügt werden.
Zusätzlich sollte ein VRML Loader oder ein anderer Loader für 3D Modelle implementiert beziehungsweise dem System hinzugefügt werden, so dass es einfacher wird, verschiedene Modelle zu laden.

Literatur

[NIN05] Nintendo, "The Legend of Zelda", zu finden unter http://zelda.nintendo.de/deDE/?mId=3

[WIM] "Netzhaut", zu finden unter http://de.wikipedia.org/wiki/Netzhaut, Oktober 2006

[CEZ] Cezanne, SStillleben mit Äpfeln", zu finden unter http://www.ee.ryerson.ca/ elf/powerbook/07-journal.html

[TEE98] D. Teece, "3D Painting for Non-Photorealistic Renderingïn SIGGRAPH 98: Conference Abstracts and Applications, ACM SIGGRAPH, 1998

[FER03] R. Fernando, M. J. Kilgard, "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics", Addison Wesley, 2003

[GOO01] Amy Gooch, Bruce Gooch, "Non-photorealistic Rendering", University of Utah, School of Computing, AK Peters, Ltd., 2001

[GOO99] Bruce Gooch, P.-P. J. Sloan, Amy Gooch, Peter Shirley, R. Riesenfeld, Ïnteractive Techical Illustration", Symposium on Interactive 3D Graphics, 1999

[GOO98] Amy Gooch, Bruce Gooch, Peter Shirley, E. Cohen, Ä Non-photorealistic Lighting Model for Automatic Technical Illustration", in SIGGRAPH '98 Conference Proceedings, ACM SIGGRAPH, 1998

[WIN94] G.Winkenbach, Daniel H.Salesin, "Computer-Generated Pen-and-Ink Illustration", In SIGGRAPH '94 Conference Proceedings, ACM SIGGRAPH, 1994

[LAK00] A. Lake, C. Marshall, M. Harris, M. Blacksteib, SStylized Rendering Techniques For Scalable Real-Time 3D Animation", ...

[MUE05] Prof. Dr. S. Müller, Materialien der Vorlesung "Photorealistisches Computergraphik", http://www.uni-koblenz.de/FB4/Institutes/ICV/AGMueller/Teaching/SS05/PCG/Materialien, 2005

[SIG99] D. Salesin, S. Schofield, A. Hertzmann, P. Litwinowicz, A. Gooch, C. Curtis, B. Gooch, "Non-Photorealistic Renderingïn SIGGRAPH 99 Course 17, 1999

[WOO97] Daniel N. Wood, Adam Finkelstein, John F.Hughes, Craig E. Thayer, David H. Salesin, "Multiperspective Panoramas for Cel Animationïn Proceedings of SIGGRAPH 97, in Computergraphics Proceedings, Annual Conference Series, 1997

[COL02] J.P. Collomosse, P.M. Hall, "Cubist Style Rendering from Photographs", 2002

[MAS99] Maic Masuch, Stefan Schlechtweg, Ronny Schulz, SSpeedlines - Depicting Motion in Motionless Picturesïn SIGGRAPH 99 Conference Abstracts and Applications, S.277, ACM SIGGRAPH, 1999

[WIK] "GPGPU", zu finden unter http://de.wikipedia.org/wiki/GPGPU, Juli 2006

[GRU06] Timo Grubing, "Teapot", von der Verfasserin in Auftrag gegebene Zeichnung des Teapots, 2006