UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Augmented Reality und Non-Photorealistic Rendering

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Martin Schumann

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**

2 Grundlagen **2**
2.1 Non-Photorealistic Rendering 2
2.2 Augmented Reality 3
2.2.1 ARToolKit 4
2.3 OpenGL Shading Language 8
2.3.1 Aufbau 9
2.3.2 Shader Management 11

3 Implementierung **15**
3.1 Grafik 16
3.1.1 Toonshading 16
3.1.2 Gooch-Shading 18
3.1.3 Silhouetten 22
3.2 Videobild 24
3.2.1 Kantendetektion 26
3.2.2 Vereinfachung 28
3.2.3 Farbanpassung 29
3.3 Steuerung 31

4 Ergebnisse **32**

5 Fazit und Ausblick **39**

Literatur **41**

1 Einleitung

Die Entwicklung in der Computergrafik ist seit jeher stark von dem Ziel geprägt, möglichst photorealistische Bilder in Echtzeit zu erzeugen. Dieser Trend wird durch die rasanten Fortschritte der Leistungsfähigkeit im Bereich moderner Grafikhardware weiter vorangetrieben. In Anwendungen der Erweiterten Realität (Augmented Reality, AR) werden von Kameras erfasste Videobilder mit virtuellen Objekten überlagert. Gerade hier sind die Bestrebungen groß, die eingeblendete Grafik so echt wie möglich erscheinen zu lassen, um größtmögliche Immersion zu erreichen. Es werden komplexe Lichtberechnungen durchgeführt, in welche die reale Beleuchtungssituation mit einbezogen wird, damit sich das Virtuelle dem Realen annähert. Oftmals ist eine nicht-photorealistische Darstellung jedoch aus verschiedenen Gründen erstrebenswerter.

Als Inspiration dienen [FIS1], [FIS2] und [HAL05], die den umgekehrten Weg zum Photorealismus in der AR beschreiten. Sie stellen Verfahren vor, welche die reale Umgebung durch Stilisierung an die virtuelle Welt angleichen und zugleich eine Darstellung in Echtzeit ermöglichen.

Abbildung 1: aus [FIS2]

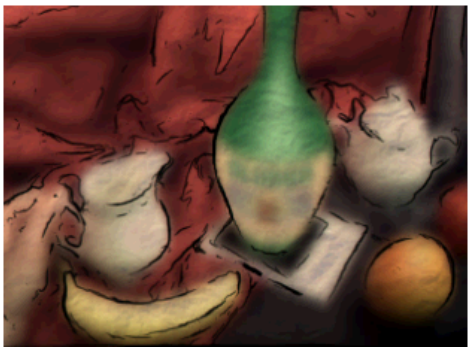

Abbildung 2: aus [HAL05]

Ziel dieser Studienarbeit ist es, eine vorhandene video-see-through Augmented Reality Visualisierung (ARToolKit) anhand von Shaderprogrammierung mit der OpenGL Shading Language um nicht-photorealistische Renderingverfahren zu ergänzen. Dabei sollen nicht nur die virtuellen Objekte mit diesen Verfahren dargestellt, sondern auch die realen vom AR-System gelieferten Bilder analog stilisiert werden, sodass die Unterscheidbarkeit zwischen Realität und Virtualität abnimmt.

In Kapitel 2 werden die dazu notwendigen Grundlagen und die verwendeten Techniken erläutert. Kapitel 3 geht auf die konkrete Umsetzung der Ziele in der Implementierung ein. In Kapitel 4 wird das Ergebnis der Arbeit besprochen. Zum Abschluss werden Vorschläge zur Verbesserung und Erweiterung des Systems erörtert.

2 Grundlagen

2.1 Non-Photorealistic Rendering

Unter Non-Photorealistic Rendering (NPR) versteht man das Darstellen von Bildern, deren Elemente unter Verwendung von bestimmten Stilmitteln in den Bereichen Form, Farbe, Struktur, Schattierung und Licht verändert werden, sodass sie von der wahrnehmbaren Wirklichkeit abweichen. NPR stellt dabei jedoch nicht das Gegenteil von photorealistischer Computergrafik dar, sondern gilt als Erweiterung der Darstellungsmöglichkeiten, genauer als generalisierende Form der Darstellung (nach Deussen und Masuch, 2001).

Non-Photorealistic Rendering wird vor allem eingesetzt um die Kommunikation mit Bildern zu vereinfachen. Photorealistische Darstellungen sind zu detailreich und oft zu komplex, um schnelle Verarbeitung zu gewährleisten. Neben der langen Verarbeitungszeit sind auch die Fehlerrate beim Erkennen und der subjektive sowie interkulturelle Interpretationsspielraum problematisch. Um die Verständlichkeit und damit die Informationsvermittlung zu optimieren, wird mit NPR versucht, die bestmögliche Darstellung für den jeweiligen Anwendungszweck eines Bildes zu finden, was besonders im technischen und medizinischen Bereich von Vorteil ist.

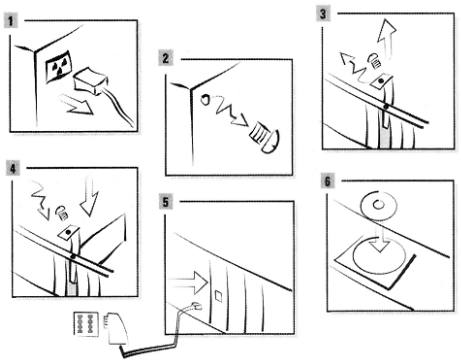

Abbildung 3: Installationsanleitung ELSA AG 1999, aus [NPR]

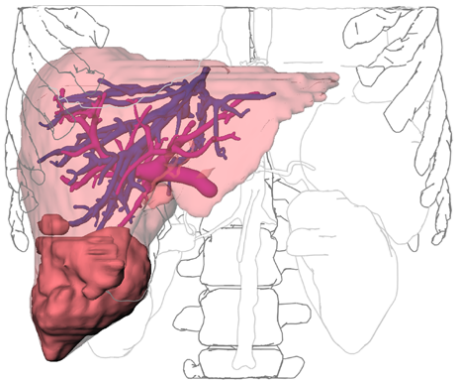

Abbildung 4: Medizinische Daten in NPR Darstellung [MED]

Dies geschieht mit Hilfe von Strukturierung und Abstraktion der Informationen. Durch das Reduzieren von Details (Level of Detail) oder Betonung von Merkmalen durch Stilisierung wird ein Fokus gesetzt, der es ermöglicht, sich auf das Wesentliche zu konzentrieren. So können textuelle Beschreibungen besser durch piktographische Bilderreihen ergänzt oder gar ganz ersetzt werden. Daher wurden die Begriffe „comprehensible rendering“ und „illustrative rendering“ geprägt.

Neben der effizienten Informationsvermittlung ist auch die durch NPR

verringerte Datenmenge und damit verbundene Rechenlast ein wichtiger Aspekt. Mobile Systeme und Augmented Reality Anwendungen können von nicht photorealistischen Darstellungen durch höhere Echtzeitfähigkeit profitieren.

Ein weiteres Gebiet des NPR sind die gestalterisch-künstlerischen Anwendungen. Während das Ziel der photorealistischen Computergrafik möglichst geringe Unterscheidbarkeit vom Foto ist, wird hier größtmögliche Ähnlichkeit zum handproduzierten Werk angestrebt. Dabei werden Kunsttechniken und Stile sowie künstlerische Medien simuliert und auf Bilder angewandt. Neben automatisierten Systemen zur Stilisierung gegebener Darstellungen gibt es auch unterstützende Systeme, welche es dem Benutzer ermöglichen, mit simulierten künstlerischen Materialien kreativ tätig zu werden. Zu diesem Bereich gehört auch das bekannte Cartoon-Shading, dass sich durch typische Farbabstufungen und Silhouettendarstellung auszeichnet.

Abbildung 5: Cartoon-Style [DMA]

2.2 Augmented Reality

Als Augmented Reality (Erweiterte Realität) bezeichnet man die Ergänzung der wahrgenommenen Wirklichkeit durch zusätzliche Informationen. Die allgemeine Definition bezieht sich auf alle menschlichen Sinne. Hier soll jedoch nur die Darstellung visueller Informationen von Bedeutung sein. Augmented Reality in der Computergrafik realisiert dies durch rechnergestützte virtuelle Einblendungen grafischer Objekte in von Kameras aufgenommene Echtzeit-Videoströme (video-see-through) oder über spezielle AR-Brillen, die eine direkte Einblendung in das Gesichtsfeld des Betrachters zulassen (optical-see-through). In der vorliegenden Arbeit kommt die erstgenannte Methode zum Einsatz.

Neben der echtzeitfähigen Interaktion mit den eingeblendeten Objekten und entsprechendem haptischen Feedback liegt der Aufgabenschwerpunkt von Augmented Reality Systemen in der Berechnung von Lage und Orientierung der bildgebenden Kamera im Raum. Sie stellt den Sichtpunkt

des Betrachters dar und ist daher Voraussetzung für das korrekte Rendern der Objekte. Ermöglicht wird das Ermitteln der benötigten Informationen über die Kamera durch das Verfahren des sogenannten Trackings, dem Finden und Verfolgen von vordefinierten Merkmalen (Marker) im Bild.[1]

Das AR System liest dazu von einer angeschlossenen Web- oder Videokamera alle aufgenommenen Bilder (Frames) aus und durchsucht sie nach bekannten Markern. Wird ein solcher Marker im aktuellen Frame gefunden, führt das System Berechnungen durch, welche die Position und Orientierung der Kamera relativ zum Marker ergeben. Das virtuelle Objekt kann nun aus Sicht der Kamera gerendert und anhand der gewonnenen 3D-Koordinaten durch Rotation und Translation an der Position des Markers eingeblendet werden.

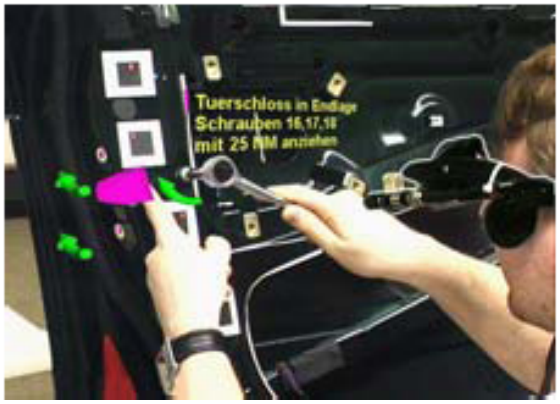

Abbildung 6: AR Montageeinsatz [VRAR]

Anwendung findet die AR Technologie besonders im medizinischen und technischen Bereich. Sie wird vorwiegend in den Phasen der Entwicklung und des Designs sowie zur Montage- und Wartungsunterstützung oder für Lern- und Trainingssituationen eingesetzt. Aber auch mobile Informationssysteme und AR-Spiele eröffnen in jüngster Zeit weitere Forschungsfelder [VRAR][WIKI].

2.2.1 ARToolKit

Als Grundlage für diese Arbeit soll das ARToolKit [ART] dienen. Es handelt sich dabei um eine Software-Bibliothek in C/C++ die als Grundlage zur Erstellung von Augmented Reality Anwendungen mit markerbasiertem Tracking dient. Besondere Vorteile, die für den Einsatz des ARToolKit sprechen, sind

[1] Es handelt sich dabei um optisches Tracking. Für Informationen zu weiteren hier nicht relevanten Arten des Trackings wird auf Literatur über AR und [VRAR] verwiesen.

- Verfügbarkeit als OpenSource unter der GNU General Public License
- große Plattformkompatibilität
- auf OpenGL und GLUT basierendes Rendering
- gute Erlernbarkeit durch Beispielapplikationen
- Einsetzbarkeit preiswerter Webcams
- einfache Kamerakalibrierung
- Echtzeit-Markererkennung
- Verwendung günstiger ausdruckbarer Marker
- Erkennung neuer selbstdefinierter Marker nach Training möglich
- optionale Unterstützung von VRML Modellen

Als Voraussetzungen zur Arbeit mit dem ARToolKit muss eine C/C++ kompatible Entwicklungsumgebung vorhanden sein, da die mitgelieferten AR Bibliotheken plattformspezifisch zu kompilieren sind. Ferner werden vorhandene Installationen der Grafikbibliothek OpenGL[2], des OpenGL Utility Toolkits GLUT, einer aktuellen DirectX Runtime[3] (nur für Windows) und eine plattformabhängige Video Library zur Kommunikation mit der Videoquelle benötigt. Sollen VRML 3D-Modelle dargestellt werden, ist optional die OpenVRML[4] Bibliothek einzubinden. Die genaue Prozedur der Einrichtung ist in der Begleitdokumentation [ART] beschrieben.

Die wichtigsten Funktionen für die einfache Erstellung einer AR Anwendung werden von der API des ARToolKit zur Verfügung gestellt. Dazu sind lediglich die zur Anwendungsentwicklung benötigten Header-Dateien des ARToolKit in das Projekt einzubinden.

ar.h - Routinen zur Initialisierung, Bildverarbeitung, Markererkennung und Berechnung der 3D Kameratransformation

param.h - Laden, Speichern und Modifizieren der Kameraparameter

video.h - Plattformunabhängiger Zugriff und Steuerung des Video Inputs

gsub.h - OpenGL / GLUT basierte Display-Funktionen für das Rendering

gsub_lite.h - Alternativ zu gsub.h für Unabhängigkeit von GLUT

arvrml.h - Optional zum Laden und Rendern von VRML Modellen

[2]http://www.opengl.org
[3]http://www.microsoft.com
[4]http://www.openvrml.org

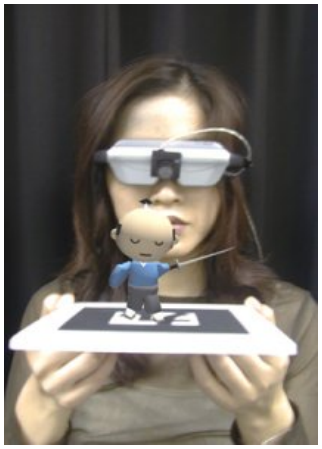

Abbildung 7: ARToolKit Beispielanwendung [ART]

Nach dem Einbinden der oben genannten Header und der dazugehörigen Libraries, kann ein AR Framework mit dem in Tabelle 1 beschriebenen Standardablauf erstellt werden. Er stellt die einfachste Form einer funktionierenden AR Anwendung dar und ist so auch in den vom ARToolKit mitgelieferten Beispielanwendungen zu finden.

Initialisierung	Öffnen der Videoquelle Einlesen der Kameraparameter und zu erkennenden Marker	init
Main Loop für jedes Frame	Auslesen eines Frames der Videoquelle	arVideoGetImage
	Ausgabe des Video Hintergrundes	argDispImage
	Finden des Markers im aktuellen Videoframe und Mustererkennung	arDetectMarker
	Berechnung der Kamera- transformation relativ zum er- kannten Marker	arGetTransMat
	Rendern der virtuellen Objekte an Position des Markers	draw
Shutdown	Schließen des Videoquelle	cleanup

Tabelle 1: ARToolKit Funktionen

Auf Implementationsdetails wird in Kapitel 3 genauer eingegangen. Zum allgemeinen Verständnis soll hier jedoch kurz der interne Verarbeitungsprozess[5] des ARToolKit beschrieben werden:

[5]Für Details siehe Literatur und Vorlesungen zu Bildverarbeitung und Rechnersehen

Das von der Videoquelle bezogene Bild liegt zunächst in durch die Kameraoptik verzerrter Form vor (Image Distortion). Es muss zuerst über die bereits gespeicherten oder die per Kamerakalibrierung ermittelten intrinsischen Kameraparameter entzerrt und in ideale Bildschirmkoordinaten gebracht werden. Danach folgt die Umwandlung in ein schwarz-weißes Binärbild, welches dann mit Hilfe von Schwellwertverfahren und Bildverarbeitungsalgorithmen zur Eckendetektion auf die quadratischen schwarzen Strukturen der AR Marker geprüft wird. Das Muster im Inneren des Markers wird in einem Pattern-Matching Verfahren mit gespeicherten Mustern verglichen. So kann der Marker eindeutig identifiziert und ihm ein zu renderndes Objekt fest zugewiesen werden. Dieses Vorgehen ermöglicht den simultanen Einsatz mehrerer Marker.

Anhand eines gefundenen Markers werden Position und Orientierung der Kamera bezüglich des Markers berechnet. Dazu kommen Algorithmen aus der Disziplin des Rechnersehens zum Einsatz. Sie leiten über die im aktuellen Frame gefundenen Merkmale und durch Vergleich mit dem vorhergehenden Frame die extrinsischen Kameraparameter her, welche die Lage der Kamera in 3D-Weltkoordinaten (Nullpunkt im Marker) angeben. Im letzten Schritt erfolgt die Transformation vom Weltkoordinatensystem in das Kamerakoordinatensystem (Nullpunkt in der Kamera). Damit ist die Lage des Markers in 3D-Koordinaten von der Kamera aus gesehen bekannt. Sie werden als Transformationsmatrix an OpenGL übergeben und dienen der korrekten Translation und Rotation der gerenderten Objekte.

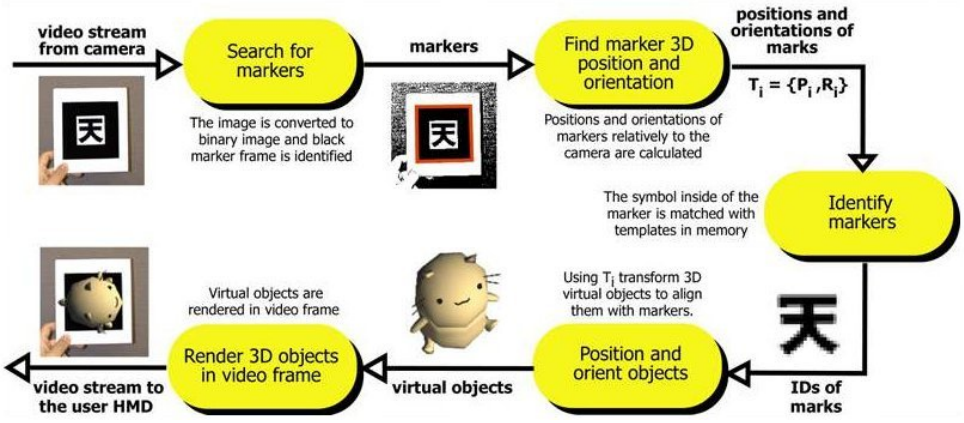

Abbildung 8: ARToolKit Funktionsweise [ART]

Die genaue Funktion der Kamerakalibrierung und Bildentzerrung sowie das Trainieren neuer Marker und Verwendung multipler Marker wird an dieser Stelle nicht behandelt, ist aber in der Dokumentation des ARToolKits [ART] beschrieben.

2.3 OpenGL Shading Language

Die Entwicklung immer schneller werdender Grafikhardware hat dazu geführt, dass Grafikprozessoren (GPUs) längst die Leistungsfähigkeit von modernen Hauptprozessoren (CPUs) überstiegen haben. Mit der Einführung von Shader-Hochsprachen wurde eine Möglichkeit für Entwickler geschaffen, GPUs zu programmieren und so ihre Leistung für komplexe Berechnungen zu nutzen.

Da die Video- und Grafikausgabe einer Augmented Reality Anwendung in Echtzeit anzustreben ist, stellte sich vor Beginn der eigentlichen Implementierung die Frage, welcher Weg für die Realisierung des Shadings und der Verarbeitung des Videostreams am effizientesten sein würde. Ein Vergleich von [FIS1] und [FIS2] zeigt, dass diese Berechnungen auf der CPU erheblich geringere Frameraten lieferten als der Einsatz der GPU unter der OpenGL Shading Language. Aufgrund der Abgestimmtheit von OpenGL und OpenGL Shading Language aufeinander, war zudem eine größtmögliche Kompatibilität und Plattformunabhängigkeit zu erwarten. Daher erschien es sinnvoll, ebenfalls diese Shading Language zu verwenden.

Die OpenGL Shading Language (auch GLSL oder glslang genannt) ist eine Programmiersprache in Anlehnung an die C-Syntax mit Erweiterungen um C++ Funktionen [GLSL1]. Mit ihr können Shader-Programme geschrieben werden, welche sich auf dem Grafikprozessor ausführen lassen. Sie ersetzen teilweise die Funktionalität der fest vordefinierten OpenGL-Grafikpipeline (fixed function pipeline). Dies eröffnet dem Programmierer eine wesentlich größere Vielfalt an Darstellungsmöglichkeiten gegenüber der bloßen Verwendung der eingeschränkten OpenGL Befehle.

Die OpenGL Shading Language ist seit OpenGL Version 2.0 ein fester Teil der OpenGL-Spezifikation mit eigenem Befehlssatz. Sie kann jedoch auch bereits ab OpenGL 1.5 als Extension verwendet werden. Dabei handelt es sich um Funktionen die zwar bereits vom Architectural Review Board (ARB)[6] verifiziert, aber noch nicht endgültig in die OpenGL Spezifikationen aufgenommen wurden. Wird großer Wert auf Abwärtskompatibilität gelegt, muss die GLSL als Extension

- GL_ARB_vertex_shader
- GL_ARB_fragment_shader

verwendet werden. Das Einbinden der vom ARB herausgegebenen Extension Header glext.h, glxext.h und wglext.h macht die Extensions im Programm verfügbar. Ihre Syntax ist von den normalen OpenGL Befehlen

[6] Vereinigung von Grafikkartenherstellern. Beschließt die OpenGL Spezifikationen.

durch den Namenszusatz ARB oder EXT zu unterscheiden. Eine Gegenüberstellung der Syntax von OpenGL 2.0 und der Befehle unter Verwendung der ARB Extensions ist in [GLSL2] zu finden. Im Folgenden wird nur auf die OpenGL 2.0 Syntax eingegangen.

2.3.1 Aufbau

In diesem Abschnitt werden nur die wichtigsten Grundlagen für die Benutzung von GLSL Shadern vorgestellt. Weiterführende Erklärungen bieten [GLSL1] und [GLSL2].

Der sonst starre Weg der Geometrie durch die Grafikpipeline (auch Rendering Pipeline genannt) bis zur Bildschirmausgabe bietet zwei Ansatzpunkte zur Shaderprogrammierung: Die programmierbaren Einheiten des Vertex Prozessors und des Fragment Prozessors. Ihre Funktion kann bei Bedarf durch entsprechende Vertex oder Fragment Shader ersetzt werden. Die Vertex-Einheit ist zuständig für alle Operationen die auf der Grundlage der einzelnen Geometrie-Eckpunkte ausgeführt werden können.

- Transformation des Vertex
- Transformation und Normalisierung der Normalen
- Generierung und Transformation der Texturkoordinaten
- Beleuchtung und Färbung des Vertex

Ein Vertexshader wird pro Vertex aufgerufen, das die Pipeline durchläuft. Er erhält dazu Informationen über die Position des Vertex, seine Farbe, die Normale, Texturkoordinaten und hat Zugriff auf weitere OpenGL States sowie benutzerdefinierte Variablen. Die OpenGL States werden dabei mit dem Präfix `gl` adressiert (etwa `gl_Vertex`, `gl_Normal`, `gl_Color`). Vertices, Normalen und Texturkoordinaten werden mit den entsprechenden OpenGL Matrizen transformiert und an den Fragmentshader weitergeleitet. Dabei findet eine Interpolation der Werte zwischen den einzelnen Vertices statt. Für jeden Shader der GLSL muss genau eine main-Methode definiert sein. In der einfachsten Form gibt ein Vertex Shader nur den transformierten Vertex zurück.

Code 1 Standard Vertexshader

```
void main(void)
{
  gl_Position = gl_ModelViewMatrix * gl_Vertex;
  alternativ: gl_Position = ftransform();
}
```

Die Fragment-Einheit ist zuständig für diejenigen Verarbeitungsschritte, die nach der Interpolation der Vertexdaten und der Rasterisierung in Fragmente ausgeführt werden.

- Berechnung der Pixelfarbe
- Texturzugriff und Texturierung
- Nebeleffekte
- Beleuchtung per Pixel

Der Fragmentshader (auch Pixelshader) wird für jedes zu rendernde Pixel aufgerufen und bestimmt die Farbe desselben, bevor er die Farbinformation über das Pixel in den Framebuffer schreibt. Er hat nur Zugriff auf das aktuelle Pixel. In der einfachsten Form schreibt ein Fragment Shader nur die Pixelfarbe in den Framebuffer oder verwirft ihn.

Code 2 Standard Fragmentshader

```
void main(void)
{
  gl_FragColor = gl_Color;
  oder: discard();
}
```

Für die benutzergesteuerte Kommunikation zwischen Anwendung und Shadern sowie zwischen Vertex- und Fragmentshader stehen folgende Daten-Qualifier zur Verfügung:

attribute - Globale Daten von der OpenGL Anwendung an den Vertexshader, die sich pro Vertex ändern und nur gelesen werden können

uniform - Globale Daten von der OpenGL Anwendung an Vertex- oder Fragmentshader, die relativ selten geändert werden (Szenenkonstant)

varying - Daten die vom Vertexshader geschrieben werden und als interpolierte Daten an den Fragmentshader übergeben werden

const - Deklaration konstanter Daten

Weitere Datentypen der GLSL sind:

- int, float, bool
- vec{2,3,4} - ein Vektor aus float Werten
- ivec{2,3,4} - ein Vektor aus integer Werten

- bvec{2,3,4} - ein boolscher Vektor
- mat{2,3,4} - Matrizen
- sampler{1D,2D,3D} - zur Aufnahme von 1D, 2D oder 3D Texturen
- samplerCube - zur Aufnahme von Cube Maps
- sampler{1D,2D}Shadow - zur Aufnahme von Shadow Maps

Innerhalb eines Shaders können neben der obligatorischen main-Methode auch eigene Funktionen definiert werden. Die Ein- und Ausgabe von Werten erfolgt im Funktionskopf mit den Definitionen

- in - für Eingabeparameter
- out - für Ausgabeparameter. Rückgabe ebenfalls mit return möglich
- inout - für Parameter die gelesen und geschrieben werden können

Strukturierte Daten werden nach dem Schema

Code 3 Datenstruktur in der GLSL

```
struct lightvector
{
  vec3 direction;
  vec3 color;
}
```

erzeugt. Zugriff erfolgt über den entsprechenden Index. Für vordefinierte Strukturen sind die Zugriffe festgelegt auf

- x,y,z,w bei Vektoren
- r,g,b,a bei Farben
- s,t,p,q bei Texturen

Die GLSL bietet des Weiteren eine Fülle mathematischer Built-in Funktionen, sowie Arrays, Schleifen und Verzweigungen (außer switch) wie in C.

2.3.2 Shader Management

Um die Shader der OpenGL Shading Language in eine für die Grafikhardware verständliche Maschinencode-Form zu bringen, müssen Vertex und Fragment Shader einzeln kompiliert und dann zu einem Programm verlinkt werden. Der gesamte Vorgang wird vom OpenGL Treiber übernommen.

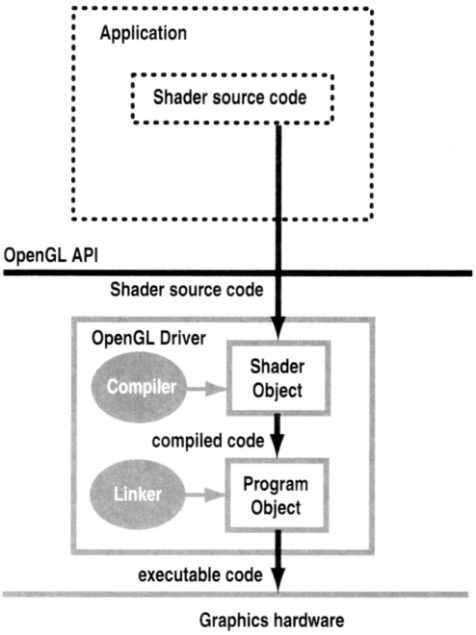

Abbildung 9: Shader Einbindungsprozess, aus [GLSL1]

Das Einbinden der Shader geschieht während der Initialisierung des aufrufenden Programms. Zuerst werden für Vertex und Fragment Shader mit `glCreateShader` je ein Shaderobjekt vom Typ `GLuint` definiert. Sie dienen als Container-Objekte für den Shader-Quellcode, der mit der Funktion `glShaderSource` an die Shaderobjekte gebunden wird. Dies geschieht durch Übergabe des Quellcodes in Form eines Pointers auf einen String oder ein Array von Strings. Sollen die Shader in Quelltextdateien vorliegen (Dateiendungen beliebig, hier .vert für den Vertex Shader und .frag für den Fragment Shader), muss eine externe Funktion zum Einlesen von Textdateien hinzugezogen werden. Anschließend werden die eingelesenen Shader zur Laufzeit mit `glCompileShader` kompiliert.

Die kompilierten Shadermodule werden dann zu einem lauffähigen Programm gebunden. Dazu wird zunächst mit `glCreateProgram` ein Programmobjekt erstellt, dass die Shaderobjekte aufnimmt. Die Shader werden über `glAttachShader` dem Programmobjekt hinzugefügt und das Programm abschließend unter Aufruf von `glLinkProgram` gelinkt.

Der Shader ist jetzt einsatzbereit und wird mit `glUseProgram` und dem Programmnamen einfach vor dem Geometrieaufruf des zu rendernden Objekts aktiviert. Um die ursprüngliche OpenGL Renderpipeline zu verwenden, wird der Shader mit `glUseProgram(0)` wieder deaktiviert. Für den Fall, dass benutzerdefinierte Variablen vom Tpy `uniform` an den Shader übergeben werden sollen, muss zuerst die Position der zu setzen-

Code 4 Laden und Kompilieren der Shader

```
GLuint vertex, fragment, program;

vertex = glCreateShader(GL_VERTEX_SHADER);
fragment = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(vertex, 1, &vertex_source, NULL);
glShaderSource(fragment, 1, &fragment_source, NULL);

glCompileShader(vertex);
glCompileShader(fragment);
```

Code 5 Binden und Linken eines Shaderprogramms

```
program = glCreateProgram();

glAttachShader(program, vertex);
glAttachShader(program, fragment);

glLinkProgram(program);
```

den Variable im Shaderprogramm mit `glGetUniformLocation` gesucht werden. Der dabei angegebene Variablenname muss exakt dem im Shader verwendeten Variablennamen entsprechen. Danach kann der gefundenen Speicherposition der Variable ein Wert zugewiesen werden. Je nach Variablentyp können Einzelwerte oder auch Vektoren mit der Funktion `glUniform{1,2,3,4}fv` übergeben werden. Dabei steht f im Funktionsaufruf für float-Werte, i für integer und v für zu übergebende Arrays. Die Zahl davor gibt die Anzahl der zu übergebenden Werte an. Analoges gilt für die Übergabe von Matrizen mit `glUniformMatrix{2,3,4}fv`. Für Variablen des Typs `attribute` existieren entsprechende Übergabefunktionen (siehe [GLSL1], [GLSL2]).

Code 6 Aufruf eines geladenen Shaders mit Variablenübergabe

```
glUseProgram(program);
GLint location = glGetUniformLocation(program, "Variable");
glUniform1f(location, value);
//Geometrieaufruf
glUseProgram(0);
```

Durch den Einsatz eines bereits vorhandenen Shader Management Frameworks [CHR04], das als Wrapper für die eigentlichen OpenGL Aufrufe dient, kann der ganze Ablauf der Shaderinitialisierung und Anwendung stark vereinfacht werden. Dies ist besonders bei der Nutzung vieler einzelner einzubindender Shader von Vorteil. Weitere positive Aspekte sind das integrierte Einlesen der Quellcodedateien, die einfachere Variablenübergabe und bereits vorhandene Fehlerbehandlung.

Nach dem Einbinden der Header-Datei des Frameworks (aGLSL.h) wird genau ein Objekt vom Typ `aShaderManager` erstellt, welches das Laden der Shader Quelldateien und den gesamten Prozess des Kompilierens und Linkens mit einem einzelnen Aufruf von `loadFromFile` übernimmt. Für jeden einzusetzenden Shader existiert ein Objekt vom Typ `aShaderObject`, dem das Shader Manager Objekt den fertigen Shader zuweist. Das Aktivieren, Deaktivieren und Übergeben von Variablen geschieht über einen Zeigerzugriff auf das Shaderobjekt. Die Variablenübergabe wird dabei mit `send` aufgerufen, gefolgt von `Uniform` oder `Attribute` und der Anzahl der zu übergebenden Werte, sowie dem Wertetyp.

Code 7 Laden und Aufruf von Shadern mit Shadermanagement-Klasse

```
aShaderManager manager;
aShaderObject* shader;
shader = manager.loadfromFile("vertex.vert","fragment.frag");

shader->begin();
shader->sendUniform1f("Variable", value);
//Geometrieaufruf
shader->end();
```

3 Implementierung

Vor Beginn der eigentlichen Implementierung müssen zunächst entsprechende Einstiegspunkte in das bereits beschriebene AR Framework gefunden werden, die es erlauben, das Videobild für eigene Verarbeitung zugänglich zu machen, sowie Einfluss auf den Renderprozess zu nehmen. Die Struktur der zu erstellenden Anwendung soll wie folgt aussehen.

Code 8 Ablauf der Anwendung

```
Initialisierung von GLUT und ARToolKit
Laden der Shader
Laden von Objektmodellen
Vorbereiten der notwendigen Texturen

Für jedes Frame
{
  Videocapture
  Schreiben des Videobildes in den Framebuffer

  Ausführen von Filtern auf dem Videobild
  {
    Kopieren des Framebuffers in Textur
    Aufruf von Videoshadern auf bildschirmfüllendem Polygon
  }

  Markerdetektion
  Berechnung der Transformation

  Falls Marker gefunden
  {
    Aufruf der Grafikshader auf Geometrie
    Erzeugen der Silhouette auf Geometrie
  }

  Ausführen von Filtern auf Videobild + Objekten
  {
    Kopieren des Framebuffers in Textur
    Aufruf von Videoshadern auf bildschirmfüllendem Polygon
  }
}
```

Die einzelnen Schritte des Ablaufs sollen dabei so modularisiert werden, dass sie iterativ angewandt werden können, einfach auszutauschen oder

zu erweitern sind. Dies wird erreicht durch die Definition getrennter Funktionen für die Geometrie- und Videoshader, die an den entsprechenden Stellen der Display-Schleife aufgerufen werden. Der Aufuf von mehreren beliebigen Videofiltern soll möglich sein und verschiedene Filterkombinationen zulassen. Zwischen den Geometrieshadern soll ebenso während der Laufzeit gewechselt werden können, wie zwischen den darzustellenden Objekten selbst.

3.1 Grafik

Da die OpenGL Befehle zur Erzeugung der einzublendenden Grafik nach dem Erkennen des Markers und der Berechnung der Transformation gesammelt durch die Methode `draw()` des ARToolKit aufgerufen werden, ist es einfach möglich, an dieser Stelle Shader einzusetzen. Für das Shading der Grafik sollen exemplarisch Gooch- und Toonshader verwendet werden. Die zu erzeugende Geometrie wird nach der Aktivierung des gewünschten Shaders durch die Methode `geometry()` ausgewählt, welche alle in der Anwendung verfügbaren Aufrufe von Geometrieobjekten enthält.

3.1.1 Toonshading

Das einfachste Shading im Non-Photorealistic Rendering beruht auf Reduzierung der Farbabstufungen eines Objekts, wobei jegliche Farbverläufe eliminiert werden und nur flächige Farbbereiche mit harten Übergängen dargestellt werden [GLSL2]. Üblicherweise wird eine Farbe ausgewählt und je nach Lichtintensität in wenigen Tönen zwischen Schwarz und Weiß dargestellt. Dieses Verfahren wird auch Cel-Shading genannt, da es den Eindruck eines zweidimensionalen Comics vermittelt.

In der konkreten Umsetzung werden die Shader dazu verwendet, eine entsprechende per Fragment Beleuchung durchzuführen. Die zur Entscheidung über die anzuwendende Farbstufe nötige Intensität `I` der Beleuchtung wird durch den Winkel zwischen Lichtvektor `L` und Normale `N` des zu beleuchtenden Punktes berechnet. Der Cosinus des Winkels zwischen zwei Vektoren lässt sich als Skalarprodukt darstellen ($I = L \bullet N$) und entspricht der diffusen Lambert-Beleuchtung. Je flacher das Licht auf den Punkt trifft, desto dunkler wird er erscheinen. Senkrechter Lichteinfall erzeugt die größte Helligkeit. Der Vertexshader liest dazu die Position der Standardlichtquelle von OpenGL aus und berechnet den Lichtvektor vom transformierten Eckpunkt zur Lichtquelle. Zusätzlich stellt er die transformierte Eckpunktnormale bereit. Lichtvektor und Normale werden für das aktuelle Fragment interpoliert und an den Fragmentshader weitergegeben.

Dieser bestimmt nach der Normierung beider Werte ihr Skalarprodukt und fängt negative Ergebnisse ab. Danach wird den Intensitäten ein Farbton zugewiesen. Ein Skalarprodukt nahe 1 zeigt eine hohe Intensität an.

Solch ein Punkt wird mit einer Farbe nahe Weiß geshadet, um den Eindruck eines spekularen Highlights zu erzeugen. Intensitäten nahe 0 werden mit sehr dunklen Tönen gezeichnet. Für Intensitätswerte dazwischen können je nach gewünschter Anzahl an Farbabstufungen beliebige Schwellwerte gesetzt werden.

Oft wird der Farbton auch mit Hilfe einer 1D-Textur bestimmt (Color-Ramp-Map), die alle Farbabstufungen enthält welche je nach errechnetem Intensitätswert ausgelesen werden. Da unnötige Texturzugriffe vermieden werden sollen und es wünschenswert ist, das Shading von Benutzerseite zur Laufzeit ändern zu können, wird der Farbwert direkt von der Anwendung an den Fragmentshader übergeben.

Abbildung 10: Toonshading

Code 9 Vertexshader zum Toonshading

```
//Varying variables for passing to Fragment shader
varying vec3 normal, lightVec;

void main()
{
  //Transformation of vertex and normal
  vec3 vertex = vec3(gl_ModelViewMatrix * gl_Vertex);
  normal = gl_NormalMatrix * gl_Normal;

  //Computation of lightvector
  lightVec = vec3(gl_LightSource[0].position) - vertex;

  gl_Position = ftransform();
}
```

Code 10 Fragmentshader zum Toonshading

```
//Uniform variables passed by application
uniform vec4 color1, color2;

//Varying variables passed by Vertex shader
varying vec3 normal, lightVec;

void main()
{
  float intensity;
  vec3 n;
  vec4 color;

  //Light intensity by dot product of normal and lightvector
  n = normalize(normal);
  intensity = max(dot(normalize(lightVec),n),0.0);

  //Decision of colorstep to use
  if (intensity > 0.98)
    color = vec4(0.8,0.8,0.8,1.0);
  else if (intensity > 0.5)
    color = color1;
  else if (intensity > 0.25)
    color = color2;
  else
    color = vec4(0.0,0.0,0.0,1.0);

  gl_FragColor = color;
}
```

3.1.2 Gooch-Shading

Bei technischen Illustrationen ist eine Reduzierung von überflüssigen Details und Betonung wichtiger Merkmale für das Verständnis oft von großem Vorteil. Standard-Beleuchtungsmodelle sind ungeeignet für diese Art der Darstellung, da sie keine einheitliche Stilisierung zulassen. In gering oder nicht beleuchteten Bereichen nimmt die Erkennbarkeit stark ab oder Details verschwinden ganz.

Gooch et al. haben ein Beleuchtungsmodell entwickelt, welches diese Probleme ausräumt [GLSL1]. Vollkommen schwarze und weiße Bereiche kommen beim Shading des Objekts nicht vor, um bessere Unterscheidbarkeit von schwarzen Silhouettenkanten und weißen Highlights von der

Objektoberfläche zu gewährleisten. Auch auf Schatten und Reflektionen wird verzichtet. Stattdessen werden eine warme und eine kalte Farbe definiert, um Form und Krümmungsverlauf der Oberfläche zu kennzeichnen. Sie werden gewichtet auf die Grundfarbe des Objekts gerechnet. Flächen mit kaltem Farbton stehen dabei für vom Licht abgewandte Bereiche, diejenigen mit warmem Farbton für dem Licht zugewandte Bereiche. Dieses Beleuchtungsmodell wird daher auch Cool-to-Warm Shading genannt. Die so gewonnenen Kalt- und Warmschattierungen werden dann mit Hilfe des diffusen Reflektionsterms $L \bullet N$ über das Objekt interpoliert. Die Gooch-Beleuchtungsformel lautet

$$k_{cool} = k_{blue} + \alpha * k_{diffuse}$$

$$k_{warm} = k_{yellow} + \beta * k_{diffuse}$$

$$k_{final} = \left(\frac{1 + N \bullet L}{2}\right) * k_{cool} + \left(1 - \frac{1 + N \bullet L}{2}\right) * k_{warm}$$

mit
k_{blue} - kalter Farbton
k_{yellow} - warmer Farbton
$k_{diffuse}$ - Grundfarbe des Objekts
α - Gewicht mit dem Grundfarbe und Kaltton verrechnet werden
β - Gewicht mit dem Grundfarbe und Warmton verrechnet werden

Nach dieser Berechnung wird ein einzelnes Highlight hinzugefügt. Es lässt sich gemäß spekularem Term des Phong-Beleuchtungsmodells aus Skalarprodukt von reflektiertem Lichtstrahl `R` und Sichtvektor `V` sowie der Glanzzahl `n` berechnen $(R \bullet V)^n$. Je höher die Übereinstimmung von reflektiertem Lichtstrahl und Sichtvektor, desto größer ist der Effekt des Highlights.

Der Vertexshader hat die Aufgabe, nach der Transformation der Vertices und Normalen das positive Skalarprodukt zwischen Lichtvektor und Eckpunktnormale zu berechnen. Außerdem werden Sicht- und Reflektionsvektor hergeleitet, die für die Erzeugung des Highlights im Fragmentshader nötig sind. Zur Berechnung des reflektierten Lichtstrahls aus Normale und einfallendem Lichtstrahl stellt die GLSL eine eigene Funktion `reflect` zur Verfügung.

Der Fragmentshader übernimmt diese Werte vom Vertexshader und zusätzlich von der Anwendung die benötigten Farbwerte und Gewichte. Nach der Verrechnung der Farben mit Hilfe der Gewichte erfolgt ihre Interpolation anhand des Skalarprodukts zwischen Normale und Lichtvektor zur endgültigen Ausgabefarbe. Auf diese wird noch der Wert des Highlights aus Skalarprodukt zwischen Reflektions- und Sichtvektor addiert. Bei allen Schritten wird die richtige Skalierung auf Werte im Bereich [0,1] durch die Verwendung von `min` und `max` Funktionen sichergestellt.

Code 11 Vertexshader zum Gooch-Shading

```
//Varying variables for passing to Fragment shader
varying float NdotL;
varying vec3  ReflectVec, ViewVec;

void main()
{
  //Transformation of vertex and normal
  vec3 vertex    = vec3(gl_ModelViewMatrix * gl_Vertex);
  vec3 normal    = normalize(gl_NormalMatrix * gl_Normal);

  //Vectors and dot products for light calculation
  vec3 lightPos  = vec3(gl_LightSource[0].position);
  vec3 lightVec  = normalize(lightPos - vertex);
  ReflectVec     = normalize(reflect(-lightVec, normal));
  ViewVec        = normalize(-vertex);
  NdotL          = (dot(lightVec, normal) + 1.0) * 0.5;

  gl_Position    = ftransform();
}
```

Abbildung 11: Gooch-Shading

Code 12 Fragmentshader zum Gooch-Shading

```
//Uniform variables passed by application
uniform vec3  SurfaceColor;
uniform vec3  WarmColor;
uniform vec3  CoolColor;
uniform float DiffuseWarm;
uniform float DiffuseCool;

//Varying variables passed by Vertex shader
varying float NdotL;
varying vec3  ReflectVec, ViewVec;

void main()
{
  //Combination of cool, warm and surface color
  vec3 cool  = min(CoolColor + DiffuseCool * SurfaceColor,1.0);
  vec3 warm  = min(WarmColor + DiffuseWarm * SurfaceColor,1.0);

  //Interpolation by dot product of normal and lightvector
  vec3 final = mix(cool, warm, NdotL);

  //Specular highlight by reflected lightvector and viewvector
  vec3 nreflect  = normalize(ReflectVec);
  vec3 nview     = normalize(ViewVec);
  float specular = max(dot(nreflect, nview), 0.0);
  specular       = pow(specular, 32.0);

  gl_FragColor   = vec4(min(final + specular, 1.0), 1.0);
}
```

3.1.3 Silhouetten

Das Finden und Rendern von Objektsilhouetten kann auf verschiedene Arten geschehen. Neben Bildraumverfahren sind auch Verfahren auf dem Objekt- oder Geomerieraum möglich [HER99]. Letztere beruhen insbesondere auf Normalentests gegen den Blickvektor für jedes Polygon. Dabei wird entweder für polygonale Netze anhand der Oberflächennormalen getestet, ob eine Objektkante jeweils ein dem Betrachter zugewandtes und ein abgewandtes Polygon verbindet

$$(Normal_{front} \bullet ViewVector) * (Normal_{back} \bullet ViewVector) \leq 0$$

oder für Freiformflächen, ob die Normale eines Punktes i senkrecht zum Blickvektor steht

$$Normal_i \bullet (Vertex_i - CameraCenter) = 0.$$

Um eine ausreichende Genauigkeit der Silhouette zu erreichen, muss die Geometrie in entsprechend hoher Polygonzahl vorliegen, was den Berechnungsaufwand stark erhöht und zusätzliche Speicherstrukturen für die Testergebnisse benötigt. Da in dieser Arbeit jedoch die Echtzeitfähigkeit von großer Bedeutung ist, sind diese Ansätze ungeeignet.

Die Bildraumverfahren arbeiten auf dem Tiefen- und Normalenpuffer, indem ein Kantenfilter auf diesen ausgeführt wird. Unstetigkeiten in Tiefen- oder Normalenwert führen so zu inneren und äußeren Silhouettenlinien, die dann im Ergebnisbild kombiniert werden.

Die einfachste und effizienteste Methode der Silhouettenerzeugung ist das Two-Pass-Rendering nach Raskar und Cohen [RAS99]. Es kann durch OpenGL selbst realisiert werden. Dazu werden im ersten Schritt die Frontpolygone der Geometrie gefüllt gezeichnet. Danach wird der Tiefenpuffer so eingestellt, dass nur diejenigen Pixel gesetzt werden, die den Tiefenwert der Frontpolygone oder eine geringere Tiefe haben. Es folgt das Rendern der gefüllten Backpolygone in schwarz. Die Stärke der Silhouette kann dabei durch das Vergrößern der Überlappung von Backpolygonen über die Frontpolygone verändert werden. Dies geschieht durch Verschieben der Backpolygone in Richtung Betrachter oder mit dem Setzen eines Offsets, der auf den Tiefenwert gerechnet wird (`glPolygonOffset`). Eine Variation die hier zum Einsatz kommt, ist das Rendern der Backpolygone im Wireframe- statt im Füll-Modus. Die Silhouettenstärke wird dabei über das Bestimmen der Liniendicke (`glLineWidth`) beeinflusst.

Das Erzeugen der Geometriesilhouette erfolgt durch den Aufruf der Methode `silhouette()` im letzten Schritt der `draw`-Routine. Auch hier erfolgt die Auswahl der Geometrie über die Methode `geometry()`.

Code 13 Silhouettenalgorithmus

```
//Render front polygons
glPolygonMode(GL_FRONT,GL_FILL);
glDepthFunc(GL_LESS);
glCullFace(GL_BACK);
geometry();

void silhouette()
{
  //Render back polygons
  glLineWidth(width);
  glPolygonMode(GL_BACK, GL_LINE);
  glDepthFunc(GL_LEQUAL);
  glCullFace(GL_FRONT);
  glColor3f(0.0,0.0,0.0);
  geometry();
  glPolygonMode(GL_BACK, GL_FILL);
}
```

Abbildung 12: Objekt mit Silhouette

3.2 Videobild

Zum Angleichen des Hintergrundes an die Grafik müssen noch vor deren Einblendung im Ablauf des ARToolKit die gelieferten Videoframes mit Hilfe von Shadern bearbeitet und dann ausgegeben werden. Die naheliegendste Möglichkeit wäre, auf den Videostrom des ARToolKit direkt zuzugreifen, da dieser als Pointer auf die Videodaten vorliegt. Dies hätte jedoch eine Aufbereitung notwendig gemacht, weil er nicht im richtigen Format für OpenGL geliefert wird. Das ARToolKit schreibt das aktuelle Videobild mit einer eigenen Funktion aus dem Videostrom in den Framebuffer. Es ist daher sinnvoll, das bereits im Framebuffer befindliche Videoframe in eine Textur zu kopieren und danach an den Shader weiterzugeben. Zur Kopie des Framebufferinhalts in eine Textur stehen folgende OpenGL Befehle zur Verfügung:

gluBuild2DMipmaps - Erzeugen von Mipmaps verschiedener Größenordnungen. Für den Echtzeiteinsatz zu langsam.

glTexImage2D - Kopiert den Inhalt des Framebuffers Pixel für Pixel in den Texturspeicher

glTexSubImage2D - Kopiert einen Unterbereich des Framebuffers Pixel für Pixel in den Texturspeicher

glCopyTexImage2D - Kopiert den Inhalt des Framebuffers in einem Stück in den Texturspeicher

glCopyTexSubImage2D - Kopiert einen Unterbereich des Framebuffers in einem Stück in den Texturspeicher

Der zuletzt genannte Befehl hat den Vorteil, dass ein einmal bei der Initialisierung reservierter Bereich im Texturspeicher bei jedem Aufruf wiederverwendet wird und nicht für jede kopierte Textur ein neuer Speicherbereich angelegt wird. Er hat keinen Einfluss auf die Verarbeitungsgeschwindigkeit der Anwendung. Seine Ausführung wird in [FIS2] ebenfalls als performanzneutral beschrieben.

Es stellte sich heraus, dass die mit den Videobildern erzeugten rechteckigen Texturen zu einer extrem langsamen Verarbeitung der Shader führten. Da OpenGL Texturen mit der Größenrestriktion 2^n verlangt, wird die Verarbeitung rechteckiger Texturen durch den Shader in Software emuliert. Die Lösung des Problems lag in der Verwendung der standardmäßigen Texturdefinition GL_TEXTURE_2D. Wird diese durch die OpenGL Extension GL_TEXTURE_RECTANGLE_ARB [7] ersetzt, funktioniert die Verarbeitung einwandfrei. Alle Texturaufrufe innerhalb des Shaders müssen analog angepasst werden. Die GLSL Typdefinition `sampler2D` für Texturen

[7] http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_rectangle.txt

ist durch `sampler2Drect` zu ersetzten und der Texturzugriff erfolgt mit `texture2Drect` statt mit `texture2D`. Zu beachten ist, dass diese Extension nicht die in OpenGL üblichen Texturkoordinaten im Bereich [0,1] verwendet, sondern die Angabe absoluter Texturkoordinaten verlangt.

Um das gefilterte Videobild wieder auszugeben werden die entsprechenden Shader auf einem bildschirmfüllenden Polygon ausgeführt, welches mit `GL_QUADS` in `drawScreenPolygon()` definiert wird. Da dieses Polygon keiner Transformation bedarf, wird für alle Videofilter der gleiche Vertexshader verwendet. Er lässt die Standardfunktion der Grafikpipeline unverändert und gibt nur die notwendigen Texturkoordinaten an den Fragmentshader weiter:

Code 14 Vertexshader für die Videofilter

```
void main()
{
  gl_Position    = ftransform();
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Die Videofilter lesen den Inhalt der übergebenen Textur für jedes in den Framebuffer zu schreibende Fragment in einer 3x3 Umgebung um das aktuelle Pixel aus. Die so erhaltenen 9 Farbvektoren dienen der Weiterverarbeitung durch den eigentlichen Filter. Sämtliche Aufrufe innerhalb des jeweiligen Fragmentshaders erfolgen dabei linear, da die Verwendung von Schleifen in der GLSL sehr ineffizient ist.

Code 15 Auslesen der 3x3 Pixelumgebung

```
uniform sampler2DRect Textur;
void main(){
  vec2 texCoord = gl_TexCoord[0].xy;
  vec4 c  = texture2DRect(Textur, texCoord);
  vec4 bl = texture2DRect(Textur, texCoord + vec2(-1.0,-1.0));
  vec4 l  = texture2DRect(Textur, texCoord + vec2(-1.0, 0.0));
  vec4 tl = texture2DRect(Textur, texCoord + vec2(-1.0, 1.0));
  vec4 t  = texture2DRect(Textur, texCoord + vec2( 0.0, 1.0));
  vec4 tr = texture2DRect(Textur, texCoord + vec2( 1.0, 1.0));
  vec4 r  = texture2DRect(Textur, texCoord + vec2( 1.0, 0.0));
  vec4 br = texture2DRect(Textur, texCoord + vec2( 1.0,-1.0));
  vec4 b  = texture2DRect(Textur, texCoord + vec2( 0.0,-1.0));
  // Filter and output
}
```

3.2.1 Kantendetektion

Die erste wesentliche Aufgabe zur Stilisierung der Videobilder ist das Finden von Kanten im Bild. Es handelt sich dabei um Diskontinuitäten im Verlauf von Grau- oder Farbwerten im Bild [BV99]. Ein Faktor der die Kantendetektion erschwert ist die Störung durch Bildrauschen, das besonders bei der Verwendung von geringerwertigen Webcams auftritt. Dies hat starken Einfluss auf die Entscheidung, welche Methode zum Finden der Kanten im Bild eingesetzt wird.

Kanten sind dort im Bild zu finden, wo eine ausreichend große Änderung der benachbarten Pixelwerte vorliegt. Diese Änderungen (Gradienten) können über Differenzbildung (Ableitung erster Ordnung) in x- und y-Richtung für jedes Pixel ermittelt werden. Die Größe der Ableitungen, der Gradientenbetrag, wird mit einem vordefinierten Schwellwert verglichen. Ist die Differenz ausreichend groß, wird an dieser Stelle ein schwarzes Kantenpixel gesetzt.

$$Gradient_x = Pixel(x, y) - Pixel(x - 1, y)$$

$$Gradient_y = Pixel(x, y) - Pixel(x, y - 1)$$

$$Gradientenbetrag = |Gradient_x| + |Gradient_y|$$

Der Sobeloperator ist ein Kantendetektor, der auf einer 3x3 Filtermaske basiert, welche mit dem Ausgangsbild gefaltet wird (Konvolution). Er beinhaltet bereits eine binominale Glättung zur Rauschminderung des Bildes. Da er richtungsabhängig ist, ist es nötig, eine Ableitung in horizontaler und vertikaler Richtung zu kombinieren.

$$Sobel_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad Sobel_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Der Laplace-Operator hingegen gehört zu den Ableitungen zweiter Ordnung. Er ist rotationsinvariant und benötigt daher nur einen Filterkern. Seine erhöhte Empfindlichkeit gegen Bildstörungen hebt jedoch einzelne Störpixel stärker hervor als der Sobeloperator. Er ist daher ungeeignet für die Anwendung auf Videobilder.

$$Laplace_4 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad Laplace_8 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Der Kantenfilter schreibt die gefundenen Kanten wahlweise in das momentan im Framebuffer befindliche Bild, auf weißen Hintergrund oder auf eine Hintergrundtextur. Die Stärke der gefundenen Kanten kann über einen

zusätzlichen Filter beeinflusst werden. Dieser sucht in einer 3x3 Umgebung nach schwarzen Pixeln. Wird mindestens eines gefunden, so wird das Zentrumspixel ebenfalls auf schwarz gesetzt. Dieses Vorgehen beruht auf der Annahme, dass das natürliche Videobild keine absoluten Schwarzwerte (0,0,0) enthält, weil die CCD-Chip Technik der Kamera dies nicht zulässt. Schwarze Pixel im Bild können daher nur von den eingeblendeten Kanten herrühren und ein Verfälschen natürlicher Schwarzbereiche im Videobild wird vermieden. Die Kantendetektion wird über die Funktion `edgeFilter()` aufgerufen.

Code 16 Fragmentshader zur Kantendetektion

```
uniform int FilterType;

void main()
{
  vec4 sum1 = vec4(0.0,0.0,0.0,1.0);
  vec4 sum2 = vec4(0.0,0.0,0.0,1.0);

  //Reading texture values, see code 15
  //Sobel-Operation
  sum1 = 2.0 * (l - r) + tl + bl - tr - br;
  sum2 = 2.0 * (t - b) + tl + tr - bl - br;
  sum1 = abs(sum1) + abs(sum2);

  //Gray value of gradient sum to compare with treshold
  float avg = (0.3 * sum1.r + 0.59 * sum1.g + 0.11 * sum1.b);

  if ( avg >= 0.3 )
  {
    //If edge detected, draw black pixel
    gl_FragColor = vec4(0.0,0.0,0.0,1.0);

  }else{

    if (FilterType == 2)
      //If wanted, take white background
      gl_FragColor = vec4(1.0,1.0,1.0,1.0);
    else
      //Exiting shader if no edge detected
      discard;
  }
}
```

Code 17 Fragmentshader zur Verbreiterung der Kanten

```
void main()
{
  //Reading texture values, see code 15

  int sum = 0;

  //Searching for absolute black neighboring pixel
  if (l.r+l.g+l.b == 0.0) sum+=1;
  if (r.r+r.g+r.b == 0.0) sum+=1;
  if (t.r+t.g+t.b == 0.0) sum+=1;
  if (b.r+b.g+b.b == 0.0) sum+=1;
  if (bl.r+bl.g+bl.b == 0.0) sum+=1;
  if (tl.r+tl.g+tl.b == 0.0) sum+=1;
  if (br.r+br.g+br.b == 0.0) sum+=1;
  if (tr.r+tr.g+tr.b == 0.0) sum+=1;

  //If sourrounding black pixel found, set center pixel black
  if (sum > 1)
    gl_FragColor = vec4(0.0,0.0,0.0,1.0);
  else
    discard;
}
```

3.2.2 Vereinfachung

Da das Videobild bei der Kantendetektion noch stark verrauschte Kanten erzeugt, soll ein Glättungsfilter als Vorverarbeitungsschritt eingesetzt werden. Als erster Ansatz dient ein einfacher linearer Mittelwertfilter, der auf die 3x3 Umgebung des aktuellen Pixels angewandt wird. Neben der Rauschunterdrückung sorgt dieser Filter auch für eine Vereinfachung der Kantenverläufe. Iterative Anwendung auf das Videobild verstärkt den Effekt. Allerdings führt sein Einsatz auch zur kompletten Auslöschung einiger Kanten.

Um das Eliminieren von Kanten zu verhindern, soll alternativ ein nichtlinearer Filter angewendet werden. Der Symmetric-Nearest-Neighbor Filter [BV99] vergleicht die umliegenden 8 Pixel mit dem Zentrumspixel. Nur diejenigen 4 der in der Filtermaske gegenüberliegenden Pixel werden gemittelt, die die geringste Differenz zum Zentrumspixel aufweisen. So wird das Mitteln über Kanten hinweg unterdrückt. Die aufrufenden Funktionen sind `blurFilter()` und `SNNFilter()`.

Code 18 Fragmentshader des Mittelwertfilters

```
void main(){
  //Reading texture values, see code 15
  //Blurring by simple average
  gl_FragColor = (c + b + t + l + r + tl + tr + bl + br) / 9.0;
}
```

Code 19 Fragmentshader des SNN Filters

```
void main(){
  //Reading texture values, see code 15
  //Taking differences of neighbor pixels to center pixel
  vec3 sum = vec3(0.0,0.0,0.0);
  vec3 m = abs(c-bl);
  vec3 n = abs(c-l);
  vec3 o = abs(c-tl);
  vec3 p = abs(c-t);
  vec3 q = abs(c-tr);
  vec3 s = abs(c-r);
  vec3 u = abs(c-br);
  vec3 v = abs(c-b);

  //Sum up 4 closest pixel and output their average
  if(all(lessThan(n,s))) sum += l; else sum += r;
  if(all(lessThan(p,v))) sum += t; else sum += b;
  if(all(lessThan(m,q))) sum += bl; else sum += tr;
  if(all(lessThan(u,o))) sum += br; else sum += tl;
  sum = sum / 4.0;
  gl_FragColor = vec4(sum, 1.0);
}
```

Die Glättungsfilter können sowohl als Vorverarbeitungsschritt und auch zur Nachbearbeitung nach der Einblendung grafischer Objekte und Anwendung anderer Filter eingesetzt werden, um die Übergänge zwischen Videobild und Grafik zu glätten.

3.2.3 Farbanpassung

Die Färbung des Videobildes ist der letzte Schritt zur Angleichung des Videobildes an das Shading der virtuellen Objekte. Dabei werden im Wesentlichen die bereits verwendeten Shader der Grafik auch auf das Bild ange-

wendet. Es wird lediglich die zur Farbauswahl benötigte Berechung der Lichtintensität über das Skalarprodukt $N \bullet L$ durch die Grauwertintensität (Mittelung der Farbkanäle) der Pixel aus der Videobildtextur ersetzt. Der Aufruf erfolgt mit der Funktion `colorFilter()`.

Code 20 Fragmentshader zur Einfärbung des Videobildes

```
uniform int FilterType;
//Color values passed by application
uniform vec4 color1, color2;

void main(){
  //Reading texture value
  vec2 texCoord = gl_TexCoord[0].xy;
  vec4 c  = texture2DRect(BaseImage, texCoord);
  vec4 color = vec4(0.0,0.0,0.0,1.0);
  //Taking grey value intensity
  float intensity = (c.r+c.g+c.b)/3.0;
  if (FilterType == 2){
    //Toon coloring
    if (intensity > 0.98)
      color = vec4(0.8,0.8,0.8,1.0);
    else if (intensity > 0.5)
      color = color1;
    else if (intensity > 0.25)
      color = color2;
    gl_FragColor = color;
  }else{
    //Gooch coloring
    vec3 SurfaceColor = vec3(0.75, 0.75, 0.75);
    vec3 WarmColor = vec3(0.6, 0.6, 0.0);
    vec3 CoolColor = vec3(0.0, 0.0, 0.6);
    vec3 cool    = min(CoolColor + 0.45 * SurfaceColor, 1.0);
    vec3 warm    = min(WarmColor + 0.45 * SurfaceColor, 1.0);
    vec3 final   = mix(cool, warm, intensity);
    gl_FragColor = vec4(final, 1.0);
  }
}
```

3.3 Steuerung

Pre-Mittelwertfilter einen Schritt erhöhen [a], zurücksetzen [y]

Post-Mittelwertfilter einen Schritt erhöhen [s], zurücksetzen [x]

SNN Filter einen Schritt erhöhen [d], zurücksetzen [c]

Objektsilhouette verstärken [f], zurücksetzen [v]

Kanten im Videobild verstärken [g], zurücksetzen [b]

Markerdetektion aktivieren [m]

Objektshader auswählen [h]

Kantendetektionsmodus wählen [j]

Kolorierung des Videobildes wählen [k]

Zu renderndes Objekt wählen [l]

Farbe für Toonshading wählen [o]

Framerate einblenden [i]

Drehen der Lichtquelle per Pfeiltasten

Drehen des Objekts auf dem Marker per Maus

Screenshot vom Anwendungsfenster [Alt+Print]

4 Ergebnisse

Die Ausführung erfolgte auf einem System mit der Konfiguration

- AMD AthlonXP 2400+
- ATI Radeon 9800 PRO 128MB
- ATI Catalyst Treiber 8.342.0.0
- DirectX 9.0c
- 1024 MB RAM
- WindowsXP Professional SP2
- OpenGL Version 2.0.6287
- OpenGL Shadding Language Version 1.10
- Logitech QuickCam Communicate STX Plus USB 2.0
- Logitech Treiber Version 10.0.0.1438

Ohne aktivierte Shader liefert das ARToolKit 14-15 Frames pro Sekunde bei einer Auflösung von 640x480 Pixeln und bis zu 28 Frames bei einer Auflösung von 320x240 Pixeln. Die verwendete Webcam sollte laut Spezifikationen zwar in der Lage sein bei 640x480er Auflösung 30 Frames zu erzeugen, jedoch ließ die Kombination aus verwendeten Treibern und ARToolKit diese Einstellung nicht zu. Es ist daher davon auszugehen, dass die gemessenen Werte für die 640x480er Auflösung mit einer höherwertigen Kamera deutlich übertroffen werden.

	Filterschritte	**320x240**	**640x480**
Ohne Filter		28.3	14.8
Mittelwertfilter	10	28.3	14.6
	20	28.3	14.6
	30	28.3	12.8
	40	28.3	9.3
SNN Filter	10	28.3	14.6
	20	28.3	9.8
	30	25.3	6.6
	40	19.0	5.8

Tabelle 2: Frameraten verschiedener Filter

Tabelle 2 listet die Frameraten für beide Auflösungen unter iterativer Verwendung der Videofilter auf. Wie zu erwarten war, führen komplexere Filteroperationen zu einer schnelleren Abnahme der Framerate, wobei diese unabhängig von der gewählten Auflösung im selben Verhältnis sinkt. Bei einem einfachen Filter jedoch bleibt die Framerate in geringer Auflösung stabil, während sie bei höherer Auflösung schneller abnimmt. Das Shading der virtuellen Objekte, Silhouetten- und Kantendetektion, sowie die farbliche Stilisierung des Videobildes führten nicht zu wahrnehmbaren Veränderungen der Framerate.

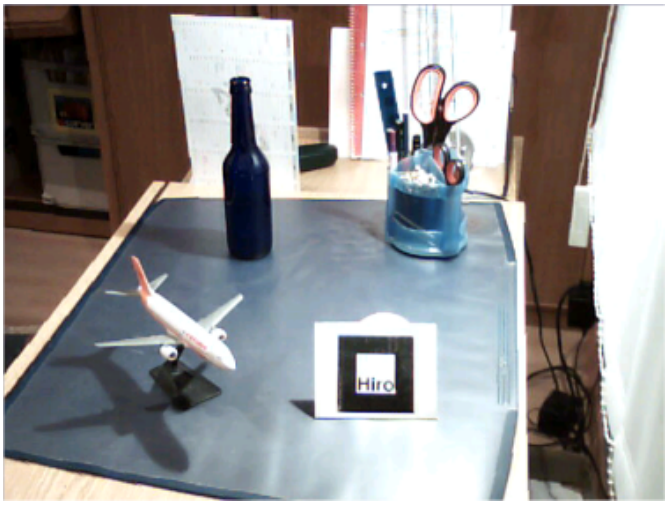

Abbildung 13: Original Testszene

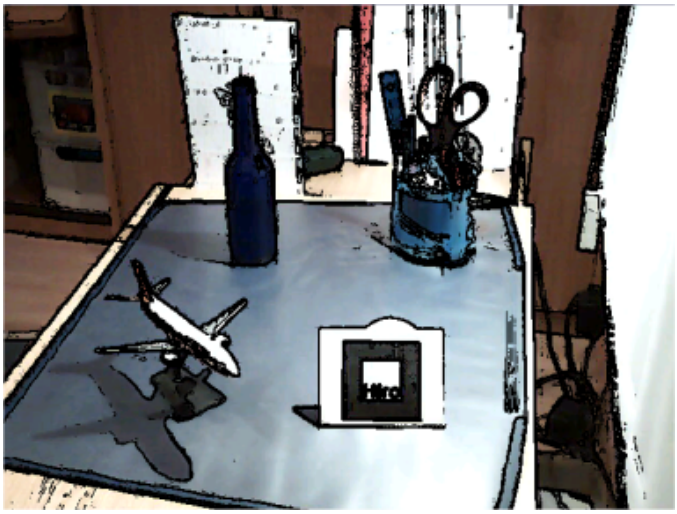

Abbildung 14: Kantendetektion ohne Vorverarbeitung

Abbildung 13 zeigt die originale Testszene ohne den Einsatz der Videoshader. Darunter wird eine einfache Kantendetektion mit dem Sobeloperator auf dem ungefilterten Videobild durchgeführt. In Abbildung 15 kommt der Mittelwertfilter iterativ 30 fach zur Anwendung. Zum Vergleich ist in Abbildung 16 die Wirkung des 30 fachen Symmetric Nearest Neighbor Filters zu sehen. Dieser bewirkt durch die Erzeugung gleichfarbiger Flächen einen künstlerischen Effekt, der an ein Ölgemälde erinnert. Abbildung 17 zeigt die Kantendetektion auf das mit dem Mittelwertfilter vorbearbeitete Hintergrundbild zur Reduzierung der Bildstörungen.

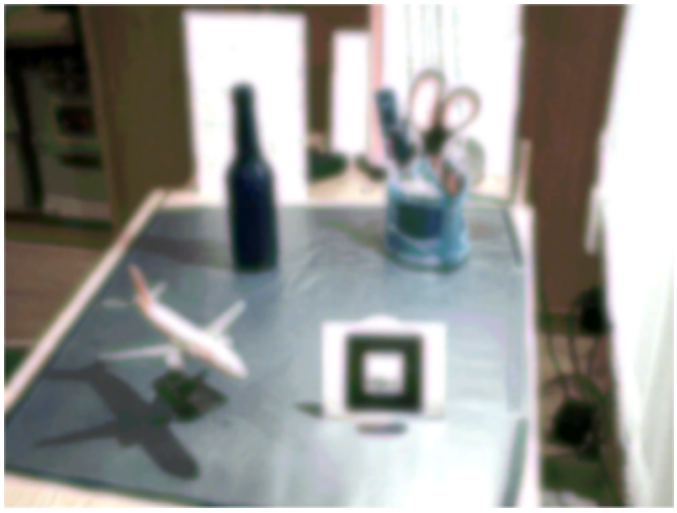

Abbildung 15: 30-facher Mittelwertfilter

Abbildung 16: 30-facher SNN Filter

Gut zu erkennen ist die kantenauslöschende Wirkung des Mittelwertfilters. Gleichzeitig werden die Kanten so verwaschen, dass die Breite ihrer Darstellung an Stärke zunimmt. Typischerweise erwiesen sich mehr als 20 iterative Filterschritte als nicht sinnvoll für Erkennbarkeit und Framerate. In Abbildung 18 wurde statt des Mittelwertfilters der Symmetric Nearest Neighbor Filter verwendet, um die Farben zu vereinfachen und dabei die Kanten zu erhalten. Aufgrund der durch die Filterung entstehenden Farbbereiche werden die Kanten allerdings unruhig und verzerrt, was sich durch kombiniertes Anwenden beider Filter ausgleichen lässt.

Abbildung 17: Kantendetektion mit Mittelwertfilter

Abbildung 18: Kantendetektion mit SNN Filter

Entscheidende Auswirkung auf die Qualität der gefundenen Kanten haben auch die Kameraeinstellungen des Webcam-Treibers. Insbesondere über die richtige Einstellung der Belichtungsempfindlichkeit ist eine gute Erkennungsrate für Kanten zu erzielen. Abbildung 19 zeigt die verwendete Testszene im Stil einer technischen Zeichnung nur mit den gefundenen Kanten im Videobild und den Silhouetten des virtuellen Objekts. In Abbildung 20 wurde die Szene mit einer Papiertextur hinterlegt, was einen künstlerischen Effekt wie etwa den einer Tuschezeichung simulieren kann.

Abbildung 19: Testszene ohne Hintergrund

Abbildung 20: Testszene mit Texturhintergrund

Die Abbildungen 21 bis 24 zeigen die Anwendung von Gooch- und Toonshading mit jeweils entsprechender Anpassung des Videobildes im Hintergrund an das eingeblendete virtuelle Objekt.

Die in [FIS1] und [HAL05] vorgeschlagenen Beschleunigungsverfahren kamen nicht zum Einsatz. Dies betrifft die Skalierung der Textur des Videobildes mit Hilfe einer Gauss-Pyramide vor der Anwendung der Videofilter und die Verwendung von Render-to-Texture Verfahren unter Nutzung von PBuffern. Daher ist eine weitere Leistungssteigerung unter Einbeziehung dieser Techniken zu erwarten.

Abbildung 21: Stilisierung mit Gooch-Shading

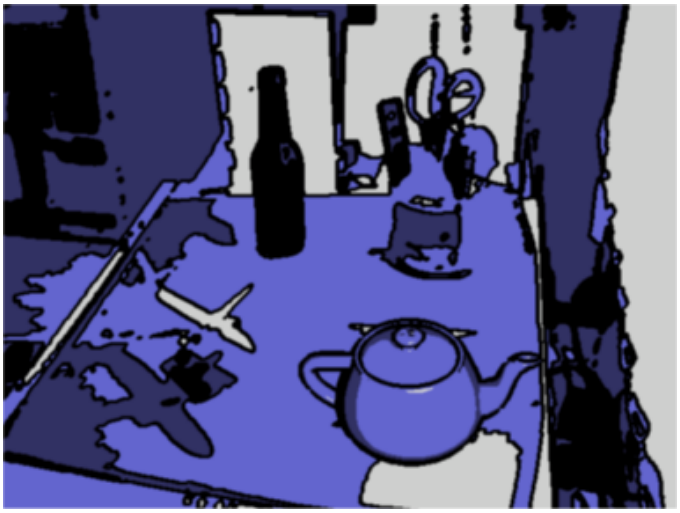

Abbildung 22: Stilisierung mit Toon-Shading

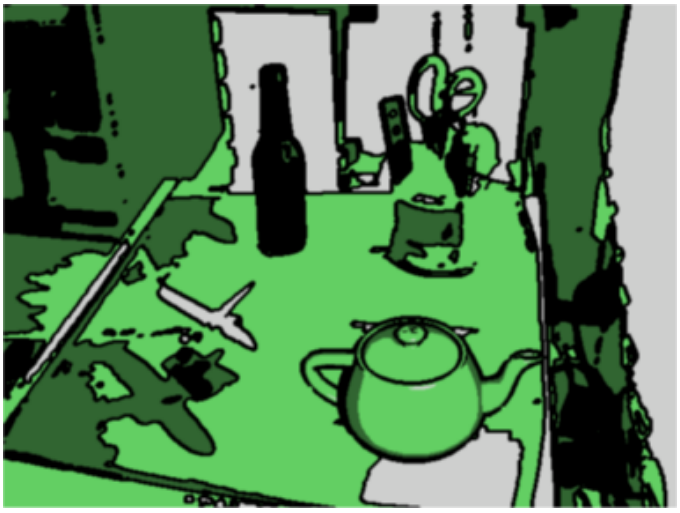

Abbildung 23: Stilisierung mit Toon-Shading

Abbildung 24: Stilisierung mit Toon-Shading

5 Fazit und Ausblick

Es ist gelungen, durch den Einsatz von Shadern eine Augmented Reality Anwendung zu schaffen, die eine aufeinander abgestimmte Stilisierung von realen Videobildern und überlagerter virtueller Grafik ermöglicht und so die Grenzen zwischen Realität und virtueller Welt verwischt. Durch Shader realisierte einfache Videofilter sind, wie gezeigt wurde, durchaus echtzeitfähig. Die Erfahrungen während der Implementierung deuten jedoch darauf hin, dass der Einsatz der OpenGL Shading Language - vor allem unter der verwendeten Grafikhardware - noch nicht vollkommen unproblematisch ist und die Möglichkeiten einschränkt.

Ein erster Ansatzpunkt für Verbesserungen ist der Geschwindigkeitsfaktor. Während das Shading der Grafik keine Auswirkung auf die Framerate hat, sollte die Filterleistung auf den Videobildern durch effizienteres Handhaben der Texturen innerhalb der Anwendung weiter erhöht werden können. Besonders die Anwendung von Texturskalierung vor der Filteranwendung und Render-To-Texture Verfahren sollten getestet werden. Ein Performanzproblem liegt in der im Moment noch nötigen Verwendung von Extensions für rechteckige Texturen und der schlechten Verarbeitung von Verzweigungen und Schleifen innerhalb der GLSL Shader. An dieser Stelle ist auf die Weiterentwicklung der OpenGL Treiber zu hoffen.

Für eine größere Vielfalt der Darstellungen können die in [HAL05] vorgeschlagenen Stilisierungen der Kantenlinien zur Simulation verschiedener Maltechniken implementiert werden. Besonders reizvoll für die Darstellung technischer Zeichnungen können an dieser Stelle Verfahren wie Hatching und Halftoning sein, wobei die Herausforderung in dem Finden einer passenden Angleichung des Videobildes liegt. Echtzeitfähige Lösungen zur Bildsegmentierung und Vereinfachung der Farbregionen im Videobild würden die Bildqualität gegenüber dem verwendeten einfachen Mittelwertfilter und SNN Filter erhöhen. Für einige dieser Verfahren wird funktionierendes Multitexturing unter OpenGL benötigt, dass in der derzeitigen Implementierung noch nicht unterstützt wird.

Da die stilisierten Bilder der vorgestellten AR Anwendung eine gewisse künstlerische Ästhetik ausstrahlen, ist eine Verwendung im kreativen Bereich naheliegend. Es bietet sich beispielsweise an, die Anwendung zu einem interaktiven Augmented Reality Comicbook zu erweitern. Der Benutzer könnte in einer Handlung mit anderen Personen und zusätzlichen virtuellen Objekten interagieren. Dabei werden nach Belieben Screenshots der Videoframes gemacht oder ein Selbstauslöser eingesetzt, der in definierten Zeitabständen Videocaptures durchführt und sie im Comic-Stil aneinander reiht.

Außerdem müsste eine grafische Benutzeroberfläche geschaffen werden, die es erlaubt, eine Auswahl aus comictypischen grafischen Elemen-

ten wie etwa Sprechblasen zu treffen und per Maus im Bild zu platzieren. Es sollte weiterhin eine Auswahl an virtuellen Objekten verschiedener Formate zur Verfügung stehen, welche momentan auf VRML beschränkt sind.

Aktuelle Filmproduktionen zielen bereits auf die Stilisierung von echten Filmaufnahmen durch den Cartoon-Look ab. Das dabei angewandte Rotoskopie[8] Verfahren ist entsprechend aufwändig, da alle Frames des Filmstreifens per Hand nachgezeichnet werden. Hier hat sich die computergestützte Farb- und Silhouettenstilisierung als vorteilhaft erwiesen. Abbildung 25 zeigt Filmbilder, die mit diesem Verfahren erstellt wurden.

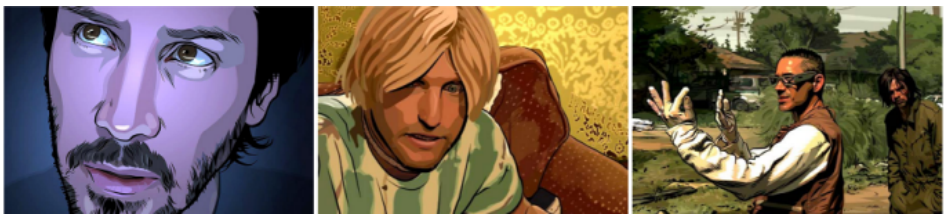

Abbildung 25: *A Scanner Darkly*, Warner Bros. Entertainment Inc 2006

Auch Spieleumgebungen bekannter Gamekonsolenhersteller setzen auf die AR-gestützte Technik, per Videokamera mit eingeblendeten Objekten interagieren zu können. In weiteren Anwendungsgebieten kann diese Art der nicht-photorealistischen Augmented Reality besonders für Lern- und Trainingssituationen von Nutzen sein, wenn erhöhte Konzentration und Anschaulichkeit gefordert sind. Es gibt bereits Ansätze, in denen die Verfälschung der Wirklichkeit zu therapeutischen Zwecken, etwa bei der Behandlung von Phobien oder Aufmerksamkeitsstörungen, eingesetzt wird. Wahrnehmungssteuerung durch Präsentieren oder Ausblenden von angstauslösenden oder ablenkenden Einflüssen ist mit Augmented Reality in der realen Umgebung glaubhafter umsetzbar, als in einer vollständigen Virtual Reality Installation.

[8] http://de.wikipedia.org/wiki/Rotoskopie

Literatur

[FIS1] **J. Fischer, D. Bartz, W. Straßer**. Stylized Augmented Reality for Improved Immersion. In *Proceedings of IEEE Virtual Reality* (VR 2005).

[FIS2] **J. Fischer, D. Bartz**. Real-time Cartoon-like Stylization of AR Video Streams on the GPU. Universität Tübingen, 2005.

[HAL05] **M. Haller, F. Landerl, M. Billinghurst**. A Loose and Sketchy Approach in a Mediated Reality Environment. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia* (GRAPHITE 2005).

[GLSL1] **R. J. Rost**. OpenGL Shading Language. Addison Wesley, 2004.

[GLSL2] **A. R. Fernandes**. GLSL Tutorial. `http://www.lighthouse3d.com/opengl/glsl/`

[ART] **H. Kato, M. Billinghurst**. ARToolKit 2.71.3. Human Interface Technology Laboratory (HITLab), University of Washington; HIT Lab NZ, University of Canterbury, New Zealand. `http://www.hitl.washington.edu/artoolkit/`

[CHR04] **M. Christen**. OGLSL C++ Framework 0.7.0 beta. Wrapper-Klassen, 2003. `http://www.clockworkcoders.com`

[VRAR] **S. Müller**. Virtuelle Realität und Augmented Reality. Vorlesung an der Universität Koblenz-Landau WS 2005/2006.

[BV99] **V. Rehrmann**. Skript zur Vorlesung Digitale Bildverarbeitung. Universität Koblenz-Landau WS 1999/2000.

[HER99] **A. Hertzmann**. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. Course on Non-Photorealistic Rendering (SIGGRAPH 1999).

[RAS99] **R. Raskar, M. Cohen**. Image Precision Silhouette Edges. Symposium on Interactive 3D Graphics 1999.

[NPR] **S. Schlechtweg, T. Strothotte**. Non-Photorealistic Computer Graphics - Modeling, Rendering and Animation. Morgan Kaufmann Publishers, 2002.

[MED] **C. Tietjen, T. Isenberg, B. Preim**. Combining Silhouettes, Surface, and Volume Rendering for Surgery Education and Planning. EUROGRAPHICS - IEEE VGTC Symposium on Visualization (2005).

[DMA] **J. Hagler**. Digital Media for Artists - E-Learning Plattform der Kunstuniversität Linz. Kurs *Nicht-fotorealistische Computergrafik /Cartoon Shading*. http://www.dma.ufg.ac.at/app/link/Grundlagen:3D-Grafik/module/11171

[WIKI] **Wikipedia**. Definition von Augmented Reality http://de.wikipedia.org/wiki/Augmented_reality