Hardware-beschleunigte Textur-Extraktion für ein fotorealistisches prädiktives Display

Tim Burkert

Lehrstuhl für Realzeit-Computersysteme

Hardware-beschleunigte Textur-Extraktion für ein fotorealistisches prädiktives Display

Tim Burkert

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. (Univ. Tokio) M. Buss
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. G. Färber
	2. Hon.-Prof. Dr.-Ing. G. Hirzinger

Die Dissertation wurde am 15.09.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 12.01.2006 angenommen.

Danksagung

Diese Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit-Computersysteme der Technischen Universität München. Teile der Arbeit wurden von der *Deutschen Forschungsgemeinschaft* (DFG) im Rahmen des Sonderforschungsbereichs 453 "Wirklichkeitsnahe Telepräsenz und Teleaktion" gefördert.

An erster Stelle möchte ich meinem Doktorvater Prof. Färber für das Ermöglichen dieser Arbeit und seine stetige Unterstützung danken. Aber auch dafür, dass er viel Verständnis gezeigt hat und immer auch ein offenes Ohr für persönliche Dinge hatte und hat. Herrn Prof. Hirzinger danke ich für seine spontane Bereitschaft, trotz seines vollen Terminkalenders als Zweitgutachter zu fungieren.

Ich danke allen Kollegen vom RCS für das angenehme Arbeitsklima und die interessanten Diskussionen. Meine Zeit am Lehrstuhl möchte ich nicht missen. Besonders danke ich den Kollegen der (ehemaligen) RoVi-Gruppe und ganz speziell meinen Kollegen im Projekt, Jan Leupold und Georg Passig. Ich habe die Zusammenarbeit trotz nervenaufreibender Begutachtungen als sehr angenehm und konstruktiv empfunden. In diesem Zusammenhang soll auch Norbert Stöffler genannt werden, der das Projekt mit ins Leben gerufen hat und uns in der Anfangsphase beratend und auch mit helfender Hand unterstützt hat.

Nicht vergessen möchte ich auch alle meine Diplomanden, die für einige Teile dieser Arbeit wichtige Beiträge geliefert haben. Wenn Ihr mal wieder jemanden braucht, der mit seinem Rotstift pedantisch alles annörgelt: meldet Euch! Danke auch an alle Korrekturleser für die hilfreichen Anmerkungen.

Ein sehr großer Dank geht natürlich an meine Eltern, für den Glauben an mich, ihre Unterstützung und überhaupt alles. Ein extragroßer Dank geht schließlich an meine Lebensgefährtin Christiane Homberg, die immer für mich da war und ist. Dein Verständnis und Deine Geduld, gerade in der "heissen Phase", haben mir vieles erleichtert. Danke!

München, im August 2005

Für meinen Vater

Inhaltsverzeichnis

Verzeichnis der verwendeten Symbole

$a, b, c \ldots$	Skalare
$\mathbf{a}, \mathbf{b}, \mathbf{c} \ldots$	Vektoren
$\mathbf{A}, \mathbf{B}, \mathbf{C} \ldots$	Matrizen
$\mathbf{P}, \mathbf{p}$	dreidimensionaler Szenenpunkt und zugehöriger Bildpunkt
${}^{W}\mathbf{P}$	Punkt im Koordinatensystem W
$\mathcal{BB}_b$	Bounding Box um einen Bildbereich
$\mathcal{BB}_p$	Bounding Box um ein Polygon
$\lceil x \rceil_2$	kleinste Zweierpotenz $\geq x$
$\underline{C}^p$	Farbwert eines Texels einer perspektivischen Textur
α^p	Alphawert eines Texels einer perspektivischen Textur
$\underline{C}^n$	Farbwert eines Texels einer normalisierten Textur
α^n	Alphawert eines Texels einer normalisierten Textur
$\underline{C}^f$	Farbwert eines Fragments für die Darstellung
α^f	Alphawert eines Fragments für die Darstellung
$\mathcal{M}_t$	geometrisches Modell zum Zeitpunkt t

AGP	Accelerated Graphics Port
API	Application Programming Interface
AR	Augmented Reality
ARB	Architecture Review Board
CPU	Central Processing Unit
DMA	Direct Memory Access
GPU	Graphics Processing Unit
ISSE	Internet Streaming SIMD Extension
IBR	Image Based Rendering
KI	Künstliche Intelligenz
LRU	Least-Recently Used
MMX	Multi Media Extension
PCI	Peripheral Component Interconnect
PCIe	PCI Express
RGB	Rot, Grün, Blau
RGBA	Rot, Grün, Blau, Alpha

RPU	Ray Processing Unit
S3TC	S3 Texture Compression
SIMD	Single Instruction, Multiple Data
SSD	Sum of Squared Differences
UDP	User Datagram Protocol
VGA	Video Graphics Array
VR	Virtual Reality

Zusammenfassung

Das Ziel von Telepräsenzsystemen ist es, dem Benutzer bei der Fernsteuerung eines entfernten Roboters den Eindruck zu vermitteln, er wäre selbst an der Stelle des Roboters, d. h. er soll die Umgebung der Roboters sehen, hören und auch fühlen können. Ein Problem sind dabei Übertragungszeiten, die Verzögerungen zwischen gegebenen Steuerkommandos und daraus resultierendem Feedback verursachen. Für die visuelle Wahrnehmung, bei der Latenzen ab etwa 300 ms als störend empfunden werden, sind eine Lösung *prädiktive Displays*: es wird vorhergesagt, wie die Umgebung des Roboters als Folge der Kommandos aussehen wird.

Die vorliegende Arbeit beschreibt neuartige Verfahren für die Realisierung eines *fotorealistischen* prädiktiven Displays. Die Basis bilden Texturen, die auf ein mit Polygonen beschriebenes benutzerseitig lokales 3D-Modell abgebildet werden. Kern der vorliegenden Arbeit ist die dazu erforderliche Extraktion der Texturen aus den vom Roboter aufgenommenen Kamerabildern. Dies beinhaltet unter anderem die Detektion von Verdeckungen in der Szene und eine zweckmäßige Verwaltung der Texturen. Bildartefakte, die durch Ungenauigkeiten bei der Datenerfassung entstehen, werden durch das Verwerfen unsicherer Farbinformationen unterdrückt. Da die Kamerabilder nur Bildinformationen für den jeweils sichtbaren Teil der Szene enthalten, können durch Änderung der Betrachterposition Bereiche der Szene dargestellt werden, für die keine Bildinformationen vorhanden sind. Es werden zwei Methoden zum Füllen solcher Gebiete mit dem Ziel eines stimmigen Gesamtbildes vorgestellt. Der Fokus liegt dabei immer auf einer für den Benutzer subjektiv harmonischen Darstellung der Szene. Um die Texturen ständig aktuell zu halten, müssen sie möglichst schnell extrahiert und mit älteren Texturen fusioniert werden. Heutige 3D-Grafikkarten bieten im Vergleich zu CPUs für bestimmte parallelisierbare Aufgaben eine sehr viel höhere Rechenleistung. Alle Methoden dieser Arbeit wurden daher mit dem Ziel entwickelt, möglichst große Teile der Verarbeitung der enormen Datenmengen direkt auf der Grafikkarte ablaufen zu lassen, wobei die damit verbundenen Einschränkungen berücksichtigt werden müssen. Da die Textur-Extraktion auf derselben Grafikkarte abläuft, die ständig für eine flüssige Darstellung der Szene benötigt wird, sind zusätzlich Strategien für Scheduling und das Teilen der Ressourcen erforderlich.

Die erfolgreiche Umsetzung des Systems wird anhand von drei verschiedenen Anwendungs-Szenarios gezeigt. Die gemessenen Ausführungszeiten belegen die erreichte Geschwindigkeit.

1 Einleitung

Eine grundlegende Frage ist in der Robotik stets von Bedeutung: welche Operationen führt der Roboter selbständig durch und was wird vom Menschen gesteuert? Für viele Aufgaben in der Industrie haben sich autonome Roboter als nützlich erwiesen. Gerade immer wiederkehrende Aufgaben können von diesen zuverlässig und (fast) ohne Ermüdungserscheinungen bewältigt werden. Für komplexere Aufgaben muss der Roboter jedoch zunehmend über seine Sensorik Informationen aufnehmen, die jeweilige Situation beurteilen und schließlich Entscheidungen treffen. Bei der Einschätzung von Situationen ist der Mensch der Maschine zumindest aktuell in der Regel noch deutlich überlegen. In den letzten Jahren ist daher in vielen Gebieten als gegensätzlicher Ansatz zur autonomen Robotik ein großes Interesse an *Telepräsenz* zu beobachten.

Telepräsenz wird erreicht, wenn es einem menschlichen Operator durch technische Mittel ermöglicht wird, mit seinem subjektiven Empfinden in einer anderen, entfernten oder nicht zugänglichen Remote-Umgebung präsent zu sein [94]. Dies beinhaltet idealerweise alle Modalitäten der menschlichen Wahrnehmung, wobei vor allem die visuelle, die akustische und die haptische (Taktilität und Kinästhetik) Wahrnehmung hervorzuheben sind. KI-Forscher Marvin Minsky beschrieb die Telepräsenz bereits 1980 als "Fernsteuerung von Robotern über immersive Schnittstellen" [70]. Dies beinhaltet aber nach heute oft geläufigem Wortgebrauch bereits die *Teleaktion*: der Benutzer soll sich nicht nur passiv subjektiv in der Remote-Umgebung befinden, er soll dort auch aktiv eingreifen können. Während in der Science-Fiction-Literatur vergleichbare Szenarien zur Integration einer Person in eine virtuelle Umgebung noch rein fiktiv als "Holodeck" [84] und "Cyberspace" [34] beschrieben werden, rücken zumindest Teile davon inzwischen im wahrsten Sinne des Wortes in greifbare Nähe.

Die Einsatzmöglichkeiten für Telepräsenzsysteme sind vielfältig. Naheliegende Anwendungen sind natürlich alle Systeme, bei denen aufgrund der Entfernung eine räumliche Trennung zwischen Operator und Roboter (Teleoperator) vorhanden ist. Gerade in der Raumfahrt müssen dabei gewaltige Entfernungen überbrückt werden, wie aktuell anhand verschiedener Mars-Missionen deutlich wird. Aber auch zur Verringerung der Gefahr für den Menschen lassen sich ferngesteuerte Roboter sinnvoll einsetzen. Das Beispiel einer Bombenentschärfung macht wohl deutlich, wie sinnvoll es dabei oft ist, dem Operator dabei den Eindruck der Telepräsenz zu vermitteln: nur wenn der Benutzer die Umgebung des Roboters intuitiv erfassen kann lassen sich falsche Bewegungen mit verheerender Folge vermeiden. Bei vielen Anwendungen spielt auch die Skalierung zwischen Operator und Teleoperator eine wichtige Rolle. So ist es für eine intuitive und feinfühlige Steuerung häufig von Vorteil bei einem minimal-invasiven robotischen Eingriff Bewegungen der

Hände des Chirurgen auf wesentlich kleinere Bewegungen der Instrumente abzubilden und dem Chirugen damit zu suggerieren er wäre selbst in verkleinerter Form an der Stelle des Roboters.

1.1 Motivation

Innerhalb des Sonderforschungsbereichs 453 *Wirklichkeitsnahe Telepräsenz und Teleaktion* [93] werden viele Aspekte der Telepräsenz in Teilprojekten aus unterschiedlichen Fachgebieten erforscht. Wie der Titel bereits verrät, ist dabei die Wirklichkeitsnähe von Telepräsenzsystemen von zentraler Bedeutung. Die vorliegende Arbeit entstand im Rahmen des Teilprojekts "Übertragungszeitkompensation durch Szenenprädiktion", das sich mit einem Aspekt der visuellen Telepräsenz beschäftigt.

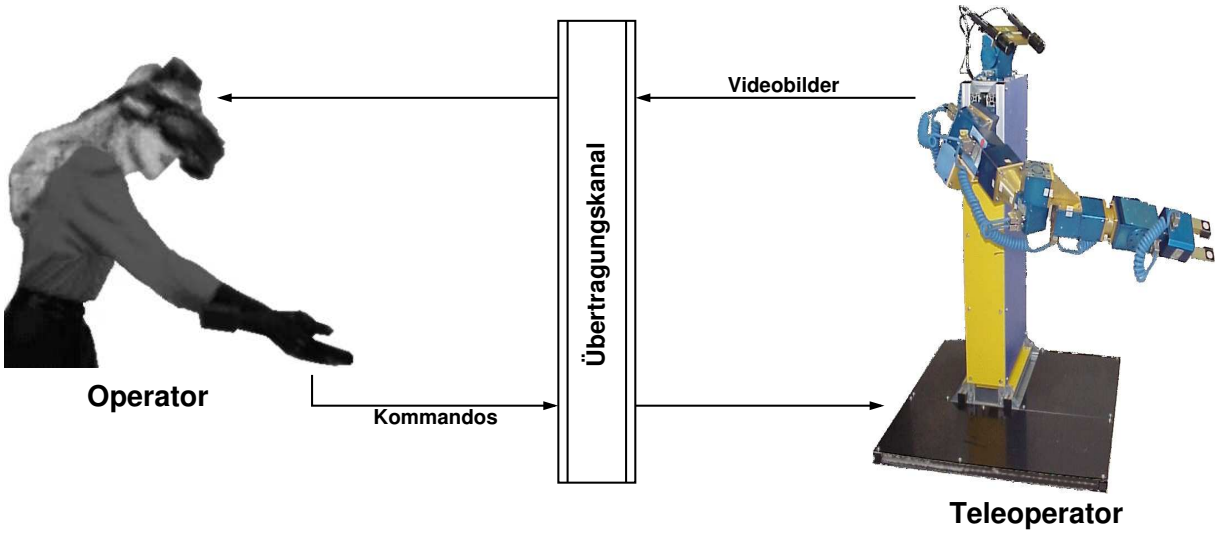


Abbildung 1.1: Telepräsenz-Szenario

Abbildung 1.1 zeigt ein System zur Realisierung einer typischen Telepräsenzaufgabe. Ein menschlicher Operator kontrolliert dabei einen enfernten robotischen Manipulator. Aktionen des Benutzers werden mittels Sensoren, wie beispielsweise Positions-Trackern an seinem Kopf und seinen Händen, gemessen. Diese Kommandos werden über den Übertragungskanal zur Remote-Umgebung geschickt und dort vom Roboter umgesetzt. Handbewegungen führen so zu Bewegungen des Roboterarms, Bewegungen des Kopfes verursachen entsprechende Bewegungen des Kamerakopfes, die beispielsweise über eine Pan-Tilt-Einheit umgesetzt werden können. Die Kameras des Roboters nehmen Bilder der Szene auf, die anschließend über den Übertragungskanal zurück zum Benutzer geschickt werden. Für viele Anwendungen ist es als visuelles Feedback ausreichend, dem Benutzer diese Bilder beispielsweise über ein Head-Mounted-Display (HMD) zu präsentieren.

Probleme entstehen jedoch mit zunehmender Übertragungszeit zwischen Operator und Teleoperator. Der Benutzer sieht die Folgen seiner Handlungen nicht mehr unmittelbar,

sondern nimmt sie erst mit entsprechender Verzögerung wahr. Dies wird vom Operator als störend empfunden und senkt seine Leistungsfähigkeit. Größere Verzögerungen führen schließlich zu einer "Move and Wait" Strategie, d. h. der Benutzer wartet nach seinen Aktionen zunächst das visuelle Ergebnis ab, bevor er neue Aktionen ausführt. Ähnliche Probleme entstehen durch eine begrenzte Bandbreite des Übertragungskanals. Ist diese zu gering, so können nicht mehr genug Bilder für eine flüssige Darstellung der Szene übertragen werden.

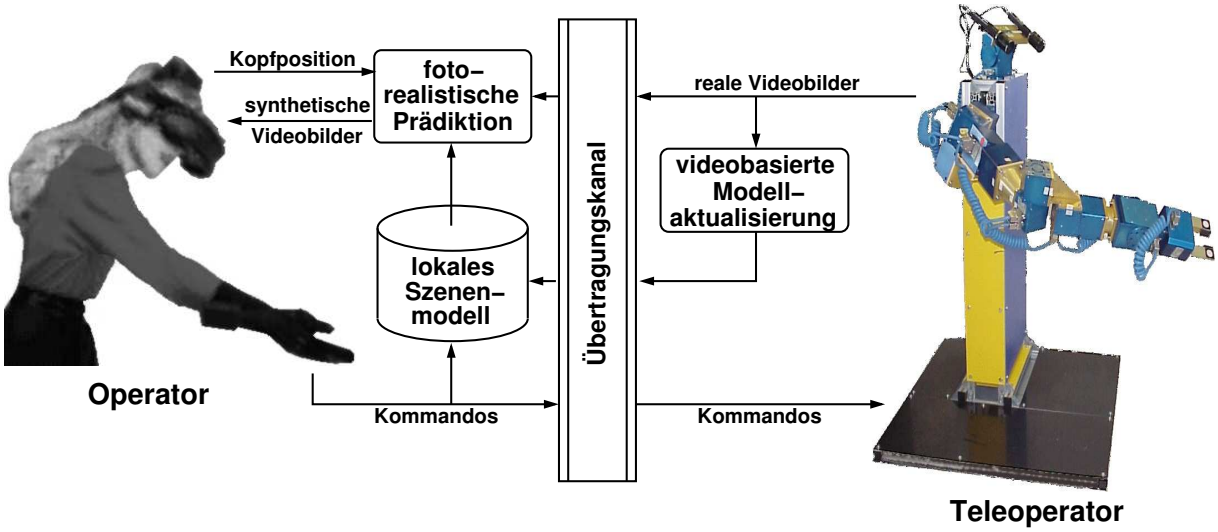


Abbildung 1.2: Prädiktives Display

Eine Lösung dafür wird mit dem in Abbildung 1.2 dargestellten *prädiktiven Display* angestrebt: es wird vorhergesagt, wie die Umgebung des Roboters als Folge der Aktionen des Benutzers aussehen wird. Anstatt der realen Bilder der Roboterkameras sieht der Benutzer dazu eine *Virtuelle Realität*, die dem Roboter und seiner Umgebung nachempfunden ist. Im genannten Projekt soll dazu eine polygonale Beschreibung der Szene über Bildverarbeitung aufgebaut und ständig aktualisiert werden. Die Szene lässt sich dabei in zwei Bereiche unterteilen. Die Geometrie von Objekten, mit denen interagiert wird, kann in der Regel vorab bestimmt werden. Ihre Aktualisierung in der Szenenbeschreibung beschränkt sich daher auf das Tracking ihrer Positionen. Auch die geometrische Beschreibung des Manipulators ist im Voraus bekannt. Über die restliche Szene sind vorab keine Informationen vorhanden. Dieser Bereich muss über Stereoverfahren und anschließende Triangulierung aufgebaut werden. Mit diesem Teil der Szene wird nicht interagiert, er ist lediglich für die Orientierung des Benutzers sowie für ein überzeugendes Empfinden von Telepräsenz erforderlich. Dieser Teil des Projektes wird in [78] beschrieben.

Die Modellbeschreibung der Szene wird auf Benutzerseite lokal gehalten. Aktionen des Benutzers kontrollieren den Greifpunkt sowohl des realen als auch des simulierten Roboters. Bei bekannter inverser Kinematik des Roboters sind die Positionen, die die einzelnen Gelenke als Folge der Kommandos einnehmen werden, bekannt und können direkt in das Modell eingetragen werden. Weitere Prädiktionen für Objekte der Szene sind durch Kollisionserkennung und/oder physikalische Simulation (z. B. eine rollende

Kugel) sowie die Mitführung von als gegriffen detektierten Objekten möglich. Die resultierende prädizierte Szene kann dem Benutzer nun anstatt der Kamerabilder als 3D-Computergrafik präsentiert werden. Durch das Schließen der Regelschleife auf der Benutzerseite sieht der Operator das Ergebnis seiner Aktionen somit ohne spürbare Verzögerung.

1.2 Zielsetzung und Gliederung der Arbeit

Ein Ziel wirklichkeitsnaher Telepräsenz ist eine hohe *Immersion* ("Eintauchen" des Benutzers in die Remote-Umgebung). Dazu ist es nicht ausreichend dem Benutzer ein geometrisches Modell zu zeigen, das als solches erkennbar ist. Statt dessen muss die Darstellung der virtuellen Realität möglichst *fotorealistisch* erfolgen. Ein gängiges Mittel in der Computergrafik zur Steigerung der Realitätsnähe sind *Texturen*. Damit werden in der Regel Bilder bezeichnet, die die Oberflächen der Polygone repräsentieren und bei der Darstellung der Szene auf diese projiziert werden. Meistens sind die Texturen bereits vorab vorhanden und müssen nur noch passend abgebildet werden. In dieser Arbeit wird für die Steigerung der Immersion des prädiktiven Displays ein neuartiger Ansatz zur automatischen Gewinnung von Texturen aus den (verzögert eintreffenden) Kamerabildern verfolgt, um sie anschließend bei der Darstellung des Modells zu verwenden. Ein weiterer Vorteil neben der dadurch gesteigerten Wirklichkeitsnähe ist, dass diese Bildinformationen dem Betrachter durch ihre perspektivisch korrekte Abbildung zusätzliche Hinweise über die Entfernung von Objekten geben ("depth cues"), wodurch die räumliche Wahrnehmung der Szene durch den Operator gefördert wird.

Um dem Benutzer ständig eine Ansicht der Szene aus beliebigem Blickwinkel zu erlauben, müssen bei der Darstellung ergänzend Bildinformationen aus älteren Aufnahmen verwendet werden, da das aktuelle Kamerabild immer nur einen Ausschnitt der Szene zeigt. Die Speicherung vollständiger Kamerabilder und ihre Verwendung für die Texturierung ist daher für diese Anwendung nicht ausreichend, weil so eine stetige Zunahme an Farbinformationen durch Fusion mit älteren Aufnahmen nicht erreicht werden kann. Aus diesem Grund erfolgt die Verwaltung der Texturen statt dessen pro Polygon. Die Gewinnung dieser Texturen aus den Kamerabildern und ihre Weiterverarbeitung ist Kern dieser Arbeit. Wichtig ist in diesem Kontext eine *schnelle Verarbeitung* der Texturen. Aktuelle 3D-Grafikkarten bieten im Vergleich zu CPUs für bestimmte parallelisierbare Operationen eine sehr hohe Rechenleistung. Bei Blick auf die Entwicklung der letzten Jahre (der Leistungszuwachs bei Grafikprozessoren hat mit einem jährlichen Faktor von etwa 2,4 sogar das Moore'sche Gesetz übertroffen) ist anzunehmen, dass sich dieser Trend weiter verstärken wird. Um die Verarbeitung der enormen Datenmengen bei der Textur-Extraktion zu beschleunigen, wurden daher alle Algorithmen mit dem Ziel entwickelt, möglichst große Teile direkt auf der Grafikkarte ablaufen zu lassen, wobei die damit verbundenen Einschränkungen berücksichtigt werden mussten.

Kapitel 2 beschreibt die Grundlagen fotorealistischer Computergrafik mit Fokus auf den für diese Arbeit interessanten Aspekten. Aufgrund der engen Verknüpfung vieler

dieser Aspekte mit der Grafik-API werden teilweise auch deren spezifische Eigenheiten betrachtet.

In **Kapitel 3** folgt eine Betrachtung einiger verwandter Arbeiten aus dem Gebiet prädiktiver Displays und der Verwendung von Kamerabildern zur Erzeugung fotorealistischer Ansichten. Auch einige Arbeiten zur Nutzung von Grafik-Hardware für Aufgaben außerhalb der Grafik-Domäne werden hier vorgestellt.

Die eingesetzten Methoden zur Extraktion der Texturen aus den Kamerabildern werden in **Kapitel 4** erläutert. Dabei erfolgt zunächst die Gewinnung der perspektivisch verzerrten Texturen aus den Kamerabildern und ihre Transformation in eine Form, die eine optimale Ausnutzung des Texturspeichers und eine vereinheitlichte Weiterverarbeitung erlaubt. Die Texturen enthalten durch Verdeckungen in der Szene teilweise Bildinformationen, die eigentlich zu anderen Polygonen gehören. Daher erfolgt eine automatische Detektion und Behandlung dieser Verdeckungen unter Ausnutzung der Grafik-Hardware. Die zuvor genannte Fusion mit älteren Texturen der Polygone füllt unbekannte Bereiche soweit möglich mit Farbinformationen. Die Beschreibung der Ursachen verschiedener Darstellungsfehler an den Polygongrenzen und Maßnahmen zu ihrer Vermeidung schließen das Kapitel ab.

Auf zwei zusätzliche Schritte zur Verarbeitung der Texturen geht **Kapitel 5** ein. Ungenauigkeiten bei der Erfassung der Daten, wie beispielsweise bei der erforderlichen Bestimmung der Kameraposition, haben zur Folge, dass Bereiche der Kamerabilder teilweise falschen Polygonen zugeordnet werden. Die Folge sind störende Artefakte bei der anschließenden Darstellung. Während in vergleichbaren Anwendungen solche Artefakte in der Regel von Hand selektiert und verworfen werden, werden hier verschiedene neue Algorithmen zum automatischen Minimieren solcher Artefakte vorgestellt. Weiterhin gibt es auch nach der Verarbeitung vieler Kamerabilder fast immer noch Bereiche in den Texturen, die in keinem dieser Bilder sichtbar waren. Um dem Benutzer trotzdem ein stimmiges Bild präsentieren zu können, werden zwei verschiedene neuartige Verfahren zum automatischen Füllen solcher "Löcher" beschrieben.

Es ist meistens nicht ausreichend, die Texturen in einer Initialisierungsphase zu gewinnen und anschließend für die Darstellung zu verwenden. Statt dessen sollen, wie in **Kapitel 6** beschrieben, die Texturen auf Basis neu empfangener Kamerabilder ständig aktualisiert werden, um Veränderungen in der Szene sichtbar zu machen. Gleichzeitig muss dem Benutzer für eine intuitive und effektive Interaktion mit der Remote-Umgebung ständig eine flüssige, prädizierte Ansicht gezeigt werden. Daher wird eine Lösung des Ressourcen-Konflikts zwischen Textur-Extraktion und Darstellung, die beide die CPU und vor allem auch die Grafikkarte für sich beanspruchen, mittels Parallelisierung bei gemeinsamer Texturverwaltung vorgestellt. Das dabei entwickelte Scheduling-Konzept erlaubt die Gewinnung der Texturen parallel zur mit nahezu konstanter Framerate ablaufenden Darstellung. Das Kapitel schließt mit der Beschreibung von Maßnahmen ab, die für einige zeitkritische Operationen zur Erhaltung der Framerate der Darstellung getroffen werden mussten.

In **Kapitel 7** folgen einige zusätzliche Betrachtungen im Kontext der Arbeit. Dabei wird ein Server beschrieben, der es erlaubt, die Textur-Extraktion asynchron zum Empfang der Kamerabilder ablaufen zu lassen. Weiterhin werden hier einige Messergebnisse zu

Ausführungszeiten des Systems präsentiert. Die Betrachtung von drei unterschiedlichen Applikationen, in denen das System erfolgreich getestet wurde, bei Fokussierung auf jeweils interessante Aspekte, runden das Kapitel ab.

Das abschließende **Kapitel 8** fasst die wichtigsten Ergebnisse zusammen.

2 Grundlagen der 3D-Grafik

Die Grundlage des prädiktiven Displays bildet ein Flächenmodell aus Polygonen mit Texturen. Mit dieser Beschreibungsform (siehe auch [27, 30]) ist eine echtzeitfähige Darstellung der Szene sogar auf Standard-PC-Hardware mit einer Consumer-Grafikkarte möglich [71]. Alle Beschreibungen und Erklärungen, die im weiteren Verlauf dieser Arbeit über 3D-Computergrafik gemacht werden, beziehen sich daher auf diese Modellbeschreibung.

Bei der Umsetzung dieser Arbeit kam die Grafik-API *OpenGL* zum Einsatz. Diese wurde ursprünglich von Silicon Graphics (SGI) als *IrisGL* entworfen und wird seit 1992 als OpenGL vom *Architecture Review Board* (ARB) weiterentwickelt, dem verschiedene Hard- und Software-Firmen angehören. OpenGL ist als *Statemachine* realisiert, d. h. alle Einstellungen werden in Form von Zustandsvariablen (State Variables) gespeichert. Solche Zustände umfassen z. B. die aktuelle Zeichenfarbe oder die aktuelle Projektionsmatrix. Ein gesetzter Zustand bleibt gültig, bis er erneut verändert wird. Eine Auflistung aller Zustandsvariablen ist in [92] enthalten.

Die Implementierung der beschriebenen Arbeit erfolgte unter dem Betriebssystem Linux, wobei für die Anbindung von OpenGL an Linux die API *GLX* [48, 103] genutzt wurde. Diese stellt verschiedene betriebssystemspezifische Funktionen, wie die Bereitstellung eines Fensters mit gewünschter Konfiguration, zur Verfügung.

In den folgenden Abschnitten werden einige für diese Arbeit relevante Elemente der Computergrafik erläutert. Da die Umsetzung verschiedener Algorithmen auf der Grafikkarte ein Kern dieser Arbeit ist, werden ausgewählte Elemente genauer betrachtet. Teilweise wird dabei im Folgenden auch auf spezielle Funktionalitäten der OpenGL-API Bezug genommen. Auch die für diese Arbeit relevanten GLX-Funktionen werden in den folgenden Unterkapiteln kurz vorgestellt. Auf die Beschreibung von Aspekten, die für diese Arbeit nur eine untergeordnete Bedeutung haben, wie beispielsweise die Beleuchtungsrechnung, wird verzichtet.

2.1 Perspektivische Projektion

Die Darstellung einer 3D-Szene mittels Computergrafik entspricht der Projektion der in der Szene enthaltenen Objekte auf eine Bildebene. Abbildung 2.1 illustriert dies für eine perspektivische Abbildung und zeigt dabei zusätzlich die in OpenGL und auch in dieser Arbeit verwendeten Koordinatensysteme. Die Beziehungen der Koordinatensysteme zueinander werden im folgenden Kapitel erläutert.

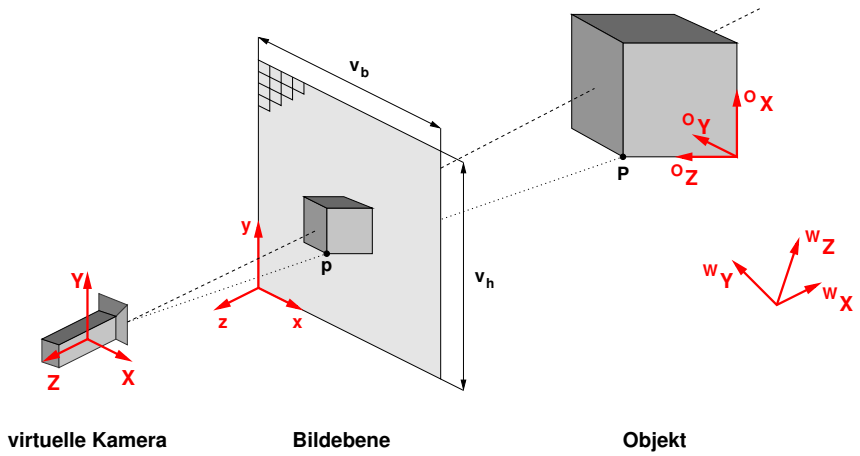


Abbildung 2.1: Projektion eines Objektes auf die Bildebene

Die Projektion wird (bei perspektivischer Abbildung) durch das Modell einer Lochkamera beschrieben. Die Analogie der für die Darstellung mittels 3D-Grafik eingeführten *virtuellen Kamera* zu einer realen Kamera ist dabei für die vorliegende Arbeit von entscheidender Bedeutung (s. Kap. 4.1).

2.2 Grafik-Pipeline

Die meisten Umsetzungen von 3D-Grafik auf einem Computer lassen sich mit dem *Pipeline-Modell* darstellen. Dieses beschreibt die genannte Umwandlung einer dreidimensionalen Modellbeschreibung in ein zweidimensionales Computerbild. Es existieren verschiedene Beschreibungen und Implementierungen dieser Pipeline, die sich jedoch zum Großteil auf eine ähnliche Struktur zurückführen lassen. Der gesamte Ablauf, bei dem diese Eingangsdaten in ein zweidimensionales Bild umgewandelt werden, wird im allgemeinen als *Rendern* (Wiedergeben) bezeichnet. Abbildung 2.2 stellt eine zur Zeit übliche und für diese Arbeit verwendete Grafik-Pipeline vereinfacht dar. Zugleich ist in der Illustration eine momentan gängige Verteilung der Komponenten auf CPU und *GPU (Graphics Processing Unit)* zu sehen. Im Folgenden sollen kurz die typischen Elemente dieser Pipeline erläutert werden.

Modell: Ein geometrisches Modell besteht üblicherweise aus mehreren Objekten (z. B. Tisch, Stühle, ...). Jedes Objekt wird in einem eigenen Koordinatensystem in 3D-Koordinaten $({}^{O}X, {}^{O}Y, {}^{O}Z)$ beschrieben und die Position dieses Objektkoordinatensystems in der virtuellen Welt festgelegt. Bei Bewegungen und Veränderungen in den Modelldaten muss dieser Datensatz aktualisiert werden.

Modelltransformationen: Die einzelnen Teile des Modells werden in ein gemeinsames Koordinatensystem transformiert. In diesem in Weltkoordinaten $({}^{W}X, {}^{W}Y, {}^{W}Z)$ beschriebenen Koordinatensystem koexistieren alle Objekte und Lichtquellen sowie die virtuelle Kamera.

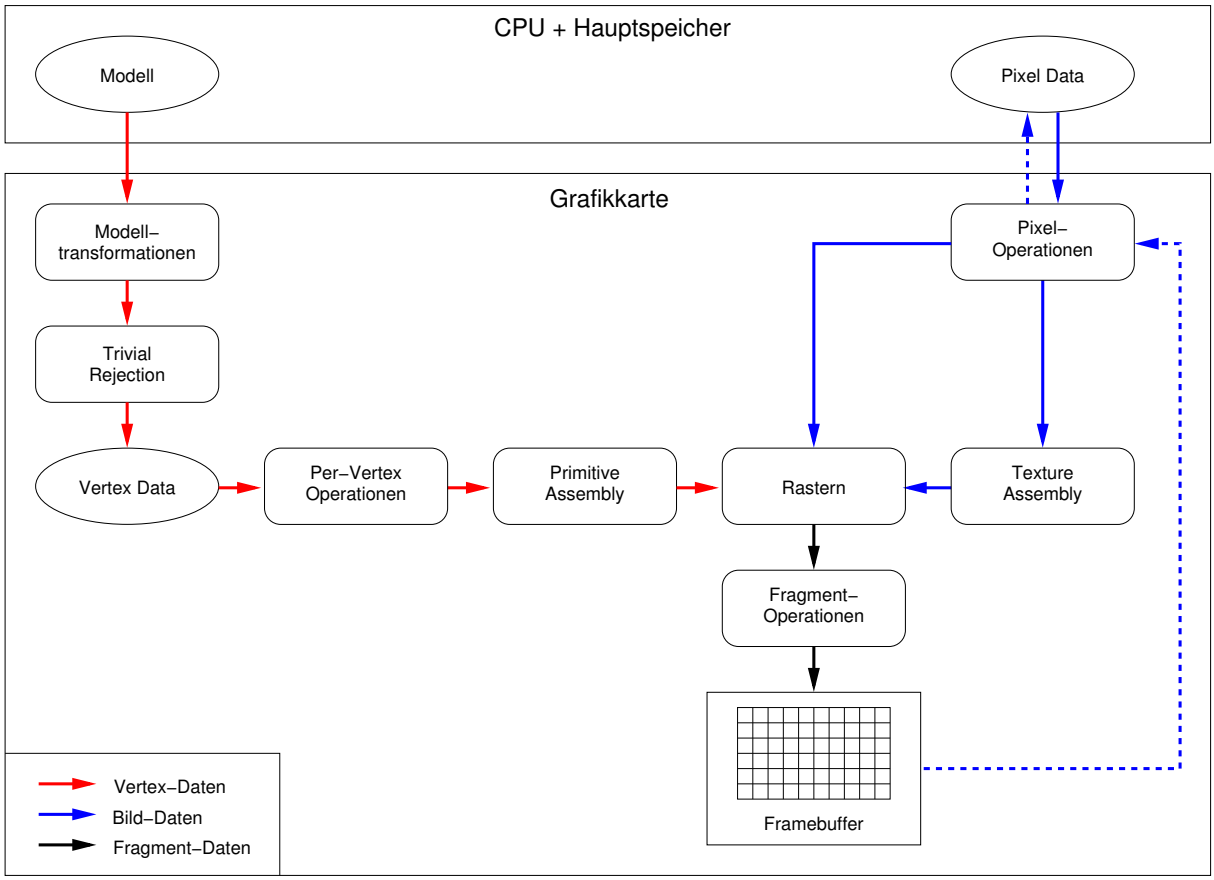


Abbildung 2.2: Grafik-Pipeline

Trivial-Rejection: Es werden alle geometrischen Primitiva verworfen, die vollständig im Halbraum hinter der virtuellen Kamera liegen. Geometrische Primitiva können Punkte, Linien oder Polygone sein und bilden die Objekte.

Vertex Data: Alle nach der Trivial-Rejection noch vorhandenen geometrischen Primitiva mit ihren *Vertices* (Eckpunkte) werden als *Vertex Data* an die nächste Stufe der Pipeline übergeben.

Per-Vertex Operationen: Die Per-Vertex Operationen transformieren unter anderem die Koordinaten der Vertices von Weltkoordinaten in die Koordinaten (X, Y, Z) der virtuellen Kamera.

Primitive Assembly: In dieser Stufe findet das *Clipping* und *Backface-Removal* statt. Beim Clipping wird die Szene auf das dreidimensionale Sichtvolumen begrenzt. Das Sichtvolumen bezeichnet den sichtbaren Bereich einer Szene. Teile der Szene, die außerhalb dieses Volumens liegen, müssen nicht gezeichnet werden. Bei perspektivischer Abbildung ist das Sichtvolumen ein Pyramidenstumpf (*Viewing Frustum*), bei orthographischer Abbildung ein Quader. Die *Clipping Planes* beschränken das Sichtvolumen auf eine minimale (*Near Clipping Plane*), und eine maximale (*Far Clipping Plane*) Tiefe. Die Form des Pyramidenstumpfes einer perspektivischen Abbildung hängt von der Vorgabe des *Field*

of Views (Sichtbereich) ab. Das Backface-Removal wird auch als *Backface Culling* bezeichnet[1]. Es eliminiert alle Polygone, deren Vorderseite nicht in Richtung der virtuellen Kamera zeigt.

Rastern: Die Objekte werden mit den Parametern der virtuellen Kamera auf ein zweidimensionales Bild abgebildet. Das durch die Projektion entstehende Bild wird in einzelne, diskrete Elemente, die *Fragmente*, unterteilt. Diese Fragmente besitzen jeweils eine eindeutige Position (x, y) in Bildschirmkoordinaten, die der Position des zugehörigen Pixels am Bildschirm entspricht. Zu einem Fragment gehören außer der Position und dem Farbvektor $\underline{C}_f = (R, G, B)^T$ noch ein Tiefenwert z_f, der sich aus der Projektion berechnet, und in der Regel ein *Alphawert* α_f. Der Alphawert wird meist als ein Maß für die Transparenz eines Fragments angesehen, kann jedoch von der Anwendung auch anders interpretiert werden. Kapitel 2.3 beschreibt die Raster-Stufe der Pipeline genauer.

Fragment-Operationen: Hier erfolgen die letzten Manipulationen der einzelnen Fragmente, bevor diese im *Framebuffer* als Pixel abgelegt werden. Diese Operationen werden in Kapitel 2.4 genauer erläutert.

Es existiert die Möglichkeit 2D-Bilder (*Pixel Data*) aus dem Hauptspeicher in den Speicher der Grafikkarte oder direkt in die Rasterstufe der Grafik-Pipeline zu laden, um daraus z.B. Texturen zu generieren. Mit den **Pixel Operationen** können diese 2D-Bilder während des Transfers aus dem Hauptspeicher manipuliert werden (z. B. Spiegeln des Bildes oder Skalierung der Farbwerte). Die **Texture Assembly** liefert der Rasterstufe dabei die Texturinformationen. Mit diesen Informationen kann die Rasterstufe die Farbvektoren $\underline{C}_f$ für jedes zu texturierende Fragment bestimmen. Texturierung wird in Kapitel 2.5 ausführlich behandelt.

In Abbildung 2.2 ist auch ein Datentransfer vom Framebuffer zurück zur Stufe der Pixel Operationen angedeutet. Über diesen Weg können gerenderte Szenen, oder Teile von diesen, beispielsweise als Texturen abgelegt werden. Diese können dann in einem folgenden Durchlauf wiederum zum Zeichnen verwendet werden. Diese Art von Algorithmen wird als *Multipass-Rendering* bezeichnet. In dieser Arbeit wird diese Methode für viele Zwecke eingesetzt. Wann immer Teile einer gerenderten Szene als Textur verwendet werden, kommt dieses Verfahren zum Einsatz.

2.3 Rastern

Ein *Pixel* (abgeleitet aus "picture element") bezeichnet einen Punkt mit dem Farbvektor $\underline{C}_p$ an der Stelle (x, y) auf dem Bildschirm. Die Position (x, y) und der Farbvektor $\underline{C}_p$, der aus $\underline{C}_f$ entsteht, sind im Framebuffer abgespeichert. Der Framebuffer ist dabei meist in *Frontbuffer* und *Backbuffer* unterteilt. Der Frontbuffer bezeichnet den Speicherbereich, in dem der aktuelle, am Bildschirm sichtbare *Frame* liegt. Bei *Double Buffering* wird im Backbuffer das zweidimensionale Bild für den nächsten Frame gerastert. Anschließend

[1] Das Backface Culling kann auch bereits bei der Trivial-Rejection durchgeführt werden

wird die Aufgabe von Front- und Backbuffer getauscht, d. h. der Frontbuffer wird zum Backbuffer und umgekehrt. Auf diese Weise wird der Bildaufbau vor dem Betrachter verborgen.

Für die Tiefenwerte z_f der Fragmente ist im Framebuffer ein eigener Speicherbereich reserviert, der *Z-Buffer* (auch *Depth-Buffer*). In ihm wird üblicherweise für jedes Fragment die (projizierte) Entfernung zwischen virtueller Kamera und nächstgelegenem Objekt gespeichert. Auf die Berechnung des Tiefenwertes wird unter anderem in den folgenden Abschnitten näher eingegangen.

2.3.1 Rastern von Linien

Abgebildete Linien haben eine Standardbreite von einem Fragment bzw. einem Pixel auf dem Bildschirm. Dies ist unabhängig von der Entfernung der Linie zur virtuellen Kamera in der Szene [88]. Das Rastern der Linien wird mit dem *Bresenham Algorithmus* realisiert [6]. Für jedes Fragment f, mit der Fragmentmitte an den Bildschirmkoordinaten (x_f, y_f) wird eine diamantenförmige Region mit Höhe und Breite gleich eins definiert:

$$R_f = \{(x,y) || x - x_f| + |y - y_f| < \frac{1}{2}\} \tag{2.1}$$

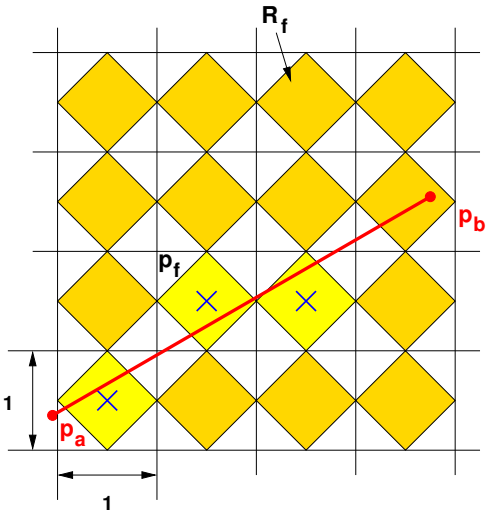


Abbildung 2.3: Diamond Exit Rule

Zu einer Linie, die von p_a nach p_b verläuft, gehören alle Fragmente deren Bereich R_f von der Linie verlassen wird (Abb. 2.3). Durch die Bedingung, dass die Linien den Bereich R_f nicht nur schneiden sondern auch verlassen müssen, wird verhindert, dass die Anfangs- und Endpunkte zusammenhängender Linien mehrfach gerastert werden. Deshalb wird diese Variante des Bresenham Algorithmus als *Diamond Exit Rule* bezeichnet. Der Bresenham Algorithmus gilt nur für Linien mit einer Steigung im Intervall $[-1, 1]$. Für Linien mit größeren Steigungen werden die Koordinaten vertauscht.

Die Berechnung der Z-Buffer-Werte $z_{f_{line}}$ eines Fragments $p_f = (x_f, y_f)$ für eine Linie erfolgt mit folgender Näherung ([88]):

$$z_{f_{line}} = (1-t)z_a + tz_b \tag{2.2}$$

mit z_a und z_b als z-Werte der Endpunkte $p_a = (x_a, y_a)$ und $p_b = (x_b, y_b)$ der Linien und t als Verhältnis

$$t = \frac{(\mathbf{p}_f - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\| \mathbf{p}_b - \mathbf{p}_a \|^2} \tag{2.3}$$

2.3.2 Linienbreite

Es können auch Linien gerastert werden, die breiter als ein Pixel sind. Die Breite b der Linie wird in Fragmenten angegeben. Die Methode, wie diese Breite b erreicht wird, hängt von der Steigung der Linie ab. Liegt sie im Intervall [-1,1], dann werden zusätzlich zu den durch die Diamond Exit Rule bestimmten Fragmenten die in positiver und negativer y-Richtung angrenzenden Fragmente bis zur Breite b gezeichnet. Liegt sie außerhalb dieses Intervalls, dann wird die Linie in x-Richtung ausgedehnt. Die maximale Breite einer Linie wird durch die Hardware vorgegeben.

2.3.3 Rastern von Polygonen

Beim Rastern von Polygonen wird die gesamte Polygonfläche gerastert. Die Vorschrift mit der Polygone gerastert werden wird als *Point-Sampling* bezeichnet. Die Eckpunkte des Polygons werden dazu auf ihre zweidimensionalen Bildschirmkoordinaten projiziert. Alle Fragmente, deren Mittelpunkt innerhalb der Begrenzung des Polygons liegt, werden gerastert. Teilen sich zwei Polygone eine gemeinsame Kante und liegt ein Fragmentmittelpunkt auf dieser Kante, dann wird es per Definition nur einem Polygon zugeordnet.

Der Z-Buffer-Wert $z_{f_{poly}}$ eines Fragments f eines Polygons wird mit folgender Näherung berechnet [88]:

$$z_{f_{poly}} = az_1 + bz_2 + cz_3 \tag{2.4}$$

mit a, b und c als baryzentrische Koordinaten (s. Kap. 2.5.4).

2.4 Fragment Operationen

Die Fragment Operationen sind eine Gruppe von Operationen zur Bearbeitung der Fragmente nach dem Rastern. Sie können nicht auf ein einzelnes Fragment an einer bestimmten Position (x_f, y_f) im Framebuffer angewendet werden. Die Anwendung erfolgt statt dessen immer auf alle Fragmente, die durch das Rastern aus einem geometrischen Primitivum

hervorgegangen sind. Alle Operationen sind optional. Abbildung 2.4 gibt einen Überblick über die möglichen Operationen und ihre Reihenfolge. Es kommt dabei eine Reihe von zusätzlichen Puffern neben dem Color-Buffer zum Einsatz, die auch als *Ancillary Buffer* bezeichnet werden.

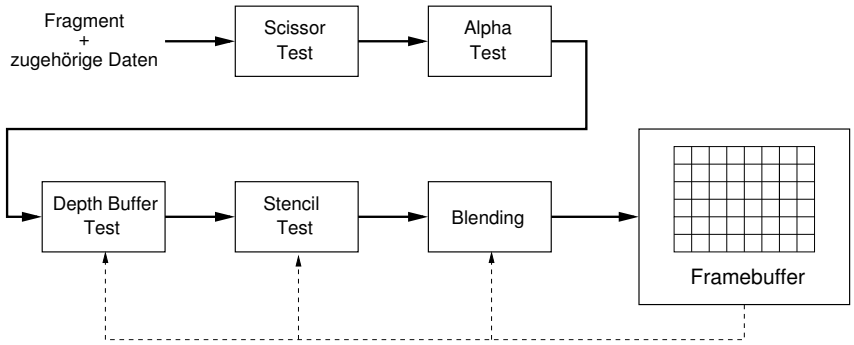


Abbildung 2.4: Fragment Operationen

2.4.1 Scissor-Test

Für den Scissor-Test wird ein Rechteck in Bildschirmkoordinaten (x, y) angegeben. Ein Fragment wird nur dann an die nächste Fragment Operation übergeben, wenn es sich innerhalb dieses Rechtecks befindet. Diese Operation wird vor allem für so genannte Bild-in-Bild Verfahren verwendet (z. B. Rückspiegel bei Fahrsimulationen).

2.4.2 Alpha-Test

Der Alpha-Test vergleicht den Alphawert α_f des aktuellen Fragments mit einem für alle Fragmente gleichen Schwellwert r. Das Fragment wird nur im Framebuffer abgelegt, falls die Vergleichsbedingung erfüllt wird. Meist entspricht $\alpha = 0$ einem vollständig transparenten und $\alpha = 1$ einem opaken Fragment. Die möglichen Werte im Intervall $\{\alpha \mid 0 \leq \alpha \leq 1\}$ beschreiben dann zunehmend undurchsichtige Fragmente (s. Kap. 2.4.5). Im Folgenden wird der Alpha-Test mit (α *op* r ?) bezeichnet. Als Vergleichsoperatoren *op* stehen die üblichen Operatoren $(<, >, =, \leq, \geq)$ zur Verfügung. Eine Anwendung für den Alpha-Test ist die Modellierung komplexer Formen. Ein Polygon wird dabei mit einer teilweise transparenten Textur gezeichnet. Diese Textur kann z. B. ein zweidimensionales Bild eines Baumes sein, bei dem alle Bereiche außerhalb des Baumes und zwischen den Blättern und Ästen einen Alphawert $\alpha = 0$ haben. Nach dem Rendern ist dann an diesen Stellen der Hintergrund der Szene sichtbar.

2.4.3 Stencil-Test

Für den Stencil-Test steht ein eigener zusätzlicher Bereich im Framebuffer zur Verfügung, der als *Stencil-Buffer* bezeichnet wird. Ein Einsatzgebiet des Stencil-Test ist die Maskierung bestimmter Bereiche, ähnlich wie beim Scissor-Test. Die Form dieses Bereiches wird nicht angegeben, sondern hängt von den in den Stencil-Buffer gerasterten Primitiva ab. In [69] ist z. B. eine Methode beschrieben, die den Stencil-Test zur Darstellung von Reflexionen nutzt.

2.4.4 Depth-Buffer-Test

Der Depth-Buffer-Test oder auch Z-Buffer-Test ermöglicht den Vergleich des Tiefenwertes z_f des aktuellen Fragments mit dem Tiefenwert des Z-Buffers. Der in [97] beschriebene Z-Buffer-Algorithmus erlaubt es z. B. komplexe Szenen zu rendern ohne die Reihenfolge der zu zeichnenden Objekte so festlegen zu müssen, dass verdeckte Objekte als erste gezeichnet werden. Der Z-Buffer-Algorithmus läuft pro Fragment in folgenden Schritten ab:

1. *Berechne z_f an der Stelle (x_f, y_f)*
2. *Ist z_f kleiner als der im Tiefenbuffer gespeicherte Wert, dann überschreibe den Tiefenbuffer und den Framebuffer an der Stelle (x_f, y_f).*

Der Tiefentest kann auch mit anderen Vergleichs-Operationen durchgeführt werden.

2.4.5 Blending

Blending ist die Mischung von Farb- und Alphawerten des aktuellen Fragments mit den Werten des im Framebuffer gespeicherten Fragments. Dabei fungiert meist der Alphawert als Parameter für das Blending. Er dient oft als Maß für das Verhältnis der Anteile, mit denen die Farbwerte gemischt werden. Dadurch kann z. B. eine zum Teil transparente Oberfläche simuliert werden.

2.5 Texture Mapping

Die am häufigsten eingesetzte Methode zur Darstellung von zusätzlichen Oberflächeninformationen bei geometrischen Flächenmodellen ist die Verwendung von *Texturen* (s. Abb. 2.5). In den Texturen können z. B. Farbinformationen, Reliefkarten oder Normalenvektoren gespeichert werden. Die in dieser Arbeit verwendeten diskreten (im Gegensatz zu prozeduralen) 2D-Texturen enthalten immer Farbinformationen. Für jedes *Texel* (aus *Tex*ture

und *el*ement in Anlehnung an Pixel) wird ein Farbvektor $\underline{C}$ und ein Alphawert α gespeichert. Die Projektion einer Textur auf ein Polygon wird als Texture Mapping (mapping, engl.: Abbildung) bezeichnet.

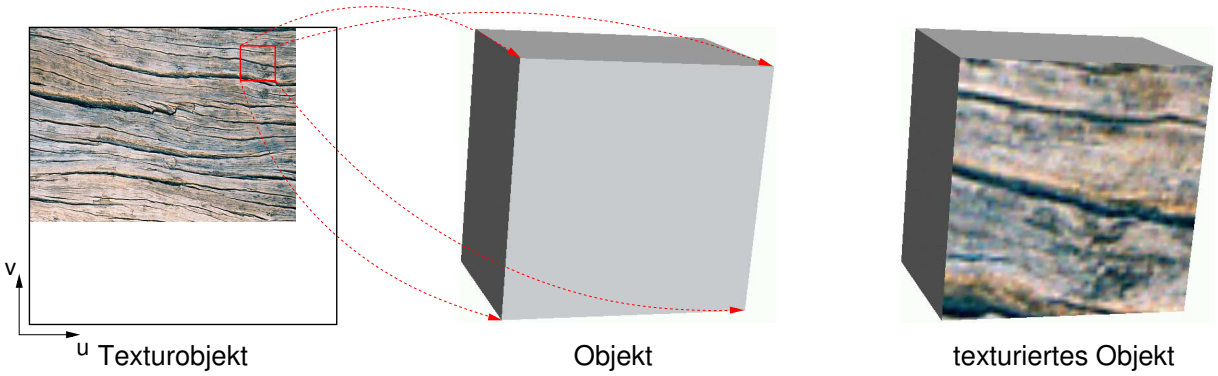


Abbildung 2.5: Texturierung eines Objektes

2.5.1 Texturobjekte

Eine Textur wird in einem rechteckigen Texturobjekt gespeichert. Per Definition sind Breite und Höhe eines Texturobjekts immer eine Zweierpotenz[2)] und werden hier mit

$$t_x = 2^m \tag{2.5}$$
$$t_y = 2^n \tag{2.6}$$

angegeben. Das Texturobjekt wird bei zur Zeit handelsüblichen PC-Grafikkarten solange im lokalen Grafikspeicher gehalten, bis es gelöscht wird oder der Speicher voll ist und das Objekt deshalb in den Hauptspeicher ausgelagert wird.

Ein Texturobjekt muss nicht vollständig von der eigentlichen Textur ausgefüllt werden (Abb. 2.5, links), sondern es können auch kleinere Texturen darin gespeichert werden. Ein Texturobjekt kann direkt aus im Hauptspeicher liegenden Bilddaten erzeugt werden. Dazu müssen die Bilddaten jedoch mit den angesprochenen Seitenlängen in Größe einer Zweierpotenz vorliegen. Eine andere Möglichkeit ein Texturobjekt zu erzeugen ist das Kopieren eines rechteckigen Bereichs aus dem Framebuffer, wobei dieser Bereich ebenfalls die Seitenlängen in Größe einer Zweierpotenz besitzen muss. In ein einmal angelegtes Texturobjekt kann, sofern die Größe ausreicht, jeder rechteckige Bereich des Framebuffers kopiert werden. Neue Texturobjekte werden in dieser Arbeit immer durch Kopieren von Bereichen aus dem Framebuffer angelegt.

[2)] Auf sehr aktuellen Grafikkarten ist diese Beschränkung aufgehoben. Für Texturen in Größen einer Zweierpotenz ist jedoch aktuell meist ein optimierter Pfad in der Pipeline vorhanden. Aus Gründen der Abwärtskompatibilität wird die Beschränkung in dieser Arbeit berücksichtigt

2.5.2 Textur-Komprimierung

Neben der Reduzierung von Auflösung oder Farbtiefe kann der für die Texturen benötigte Speicher auf der Grafikkarte auch durch Komprimierung verringert werden. Dabei wird häufig ein verlustbehaftetes Verfahren verwendet, das ursprünglich vom Grafikkartenhersteller S3 entwickelt wurde und daher den Namen *S3TC* (S3 Texture Compression, auch *DXTC*) trägt [86]. Dieses Verfahren steht durch die OpenGL-Extension `EXT_texture_compression_s3tc` (siehe [55, 75]) zur Verfügung und wird auf vielen Grafikkarten von der Hardware unterstützt.

Dabei werden bei der für diese Arbeit interessanten Kodierung jeweils für einen 4×4 Texelblock zwei repräsentative Farben mit 16 Bit Auflösung gebildet. Aus diesen werden (während der Dekodierung) zwei weitere Farben durch Interpolation im RGB-Farbraum gebildet. Jedem der 16 Texel des Blocks wird eine der vier Farben zugeordnet. Über 2 Bit Indizes wird dann bei der Dekodierung der jeweils passende Farbwert selektiert. Insgesamt werden damit für jeden Texelblock 64 Bit an Speicher benötigt: jeweils 16 Bit für die Farbwerte der beiden repräsentativ ausgewählten Farben, sowie jeweils 2 Bit für die Indexwerte der 16 Texel. Alphawerte werden, sofern vorhanden, unabhängig von den Farbwerten kodiert, wobei für jeden Texel-Block weitere 64 Bit benötigt werden. Die sich daraus ergebenden Komprimierungsfaktoren im Vergleich zu einer Speicherung im RGB(A)-Modus mit jeweils 8 Bit pro Texel und Farbkanal sind in Tabelle 2.1 dargestellt.

	Bits pro 4×4 Texel-Block	
	RGB-Modus	**RGBA-Modus**
unkomprimiert	$16 \times 3 \times 8 = 384$	$16 \times 4 \times 8 = 512$
mit S3TC-Komprimierung	64	128
Kompressionsfaktor	6 : 1	4 : 1

Tabelle 2.1: Kompressionsfaktoren bei der Texturkomprimierung mit S3TC

Bezogen auf den RGBA-Modus benötigt eine komprimierte Textur somit nur ein Viertel des Speicherplatzes einer unkomprimierten Textur derselben Größe. Durch Anwendung der Textur-Komprimierung können daher bei einer Textur mit 1024×1024 Texeln 3 Megabyte an Grafikspeicher eingespart werden.

S3TC ist ein *asymmetrisches Kompressionsverfahren*. Während die Dekomprimierung sehr schnell ist, kann die Komprimierung sehr lange dauern.

2.5.3 Texturkoordinaten

Für die Texturierung eines Polygons werden den Eckpunkten als zusätzliche Parameter *Texturkoordinaten* übergeben. Diese selektieren den Bereich innerhalb des Texturobjekts, der auf das Polygon abgebildet werden soll (s. Abb. 2.5). Beim Rendern werden die Koordinaten dann für jedes Fragment interpoliert und die entsprechenden Texel aus der

Textur gelesen. Wie in Abbildung 2.6 zu sehen, werden die Koordinaten auf unterschiedliche Weise angegeben. Ein Element (i, j) bezeichnet ein einzelnes Texel. Mit (u, v) werden die Texturkoordinaten eines Punktes angegeben. (s, t) bezeichnet schließlich die auf die Größe des Texturobjekts normierten Koordinaten:

$$s = \frac{u}{t_x} \tag{2.7}$$

$$t = \frac{v}{t_y} \tag{2.8}$$

Texturkoordinaten können auch außerhalb des Texturobjekts liegen. Für die Auswahl der Texel in solchen Fällen können verschiedene Strategien, wie eine periodische Wiederholung der Textur (*Repeat*) oder Begrenzung des Wertebereichs von s und t auf $[0, 1]$ (*Clamp*), vorgegeben werden.

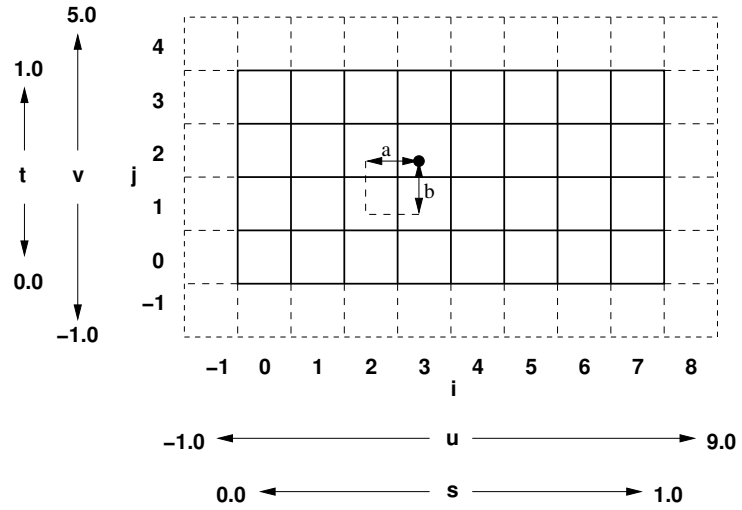


Abbildung 2.6: Texturkoordinaten

2.5.4 Filterverfahren

Die Projektion für das Texture Mapping erfordert eine Abbildung von Texeln auf einzelne Fragmente eines texturierten Polygons. Dabei wird von *Magnification* gesprochen, wenn ein Texel auf mehrere Pixel abgebildet wird. *Minification* bezeichnet die Abbildung mehrere Texel auf ein Pixel. Um den Rechenaufwand für die Magnification und Minification in Grenzen zu halten, werden die resultierenden Farbwerte nicht exakt berechnet, sondern mit Filterverfahren angenähert. Bei Magnification erfolgt die Filterung immer während des Renderns. Für Minification gibt es zusätzlich die Möglichkeit, eine Vorfilterung durchzuführen. Das bekannteste Verfahren für die Vorfilterung ist das *Mip-Mapping* [102], bei dem zusätzlich zur Textur in voller Auflösung auch Varianten in verschiedenen niedrigeren Auflösungen gespeichert werden. Für die Filterung während des Renderns stehen für Minification und Magnification die selben zwei Verfahren zur Verfügung. Diese sind das

Nearest Neighbor Verfahren und die *bilineare, gewichtete Interpolation* aus den nächsten vier Nachbarn. Diese Verfahren werden im Folgenden näher erläutert. In Abbildung 2.7 ist ein exemplarisches texturiertes Objekt mit Magnification für beide Verfahren dargestellt.

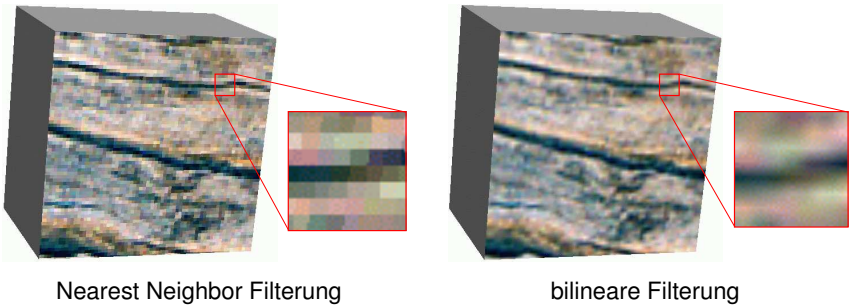


Abbildung 2.7: links: texturierter Quader mit Nearest Neighbor Magnification; rechts: texturierter Quader mit bilinearer Magnification

Bei dem Verfahren des **Nearest Neighbor** werden jeweils der Farbvektor $\underline{C}_{ij}$ und der Alphawert α_{ij} eines einzelnen Texels als neuer Farbwert für das zu texturierende Fragment übernommen. Dabei wird das Texel verwendet, dessen Texelkoordinaten den geringsten Abstand zu den Texturkoordinaten des Fragments besitzen. Für die Bestimmung des geringsten Abstands zwischen den Texturkoordinaten des Fragments und den Koordinaten des gesuchten Texels wird die *Manhatten-Distanz* berechnet.

Um ein Fragment f texturieren zu können, muss die Textur auf das Polygon projiziert werden. Statt der Abbildung der Texel auf die Pixel

$$F_{map} : (u_f, v_f) \rightarrow (x_f, y_f) \tag{2.9}$$

wird üblicherweise die durch die Rasterung praktikablere inverse Abbildung

$$F_{map}^{-1} : (x_f, y_f) \rightarrow (u_f, v_f) \tag{2.10}$$

berechnet ([26]).

Zur Bestimmung der Texturkoordinaten eines Fragments innerhalb des Polygons werden drei Zahlen a, b und c definiert, die alle im Intervall $[0, 1]$ liegen und für die $a + b + c = 1$ gilt. Sie werden als *baryzentrische Koordinaten* bezeichnet. Jeder Punkt $\mathbf{p}$ innerhalb eines Dreiecks $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ lässt sich mithilfe dieser Koordinaten wie folgt angeben (s. Abb. 2.8):

$$\mathbf{p} \quad = \quad a\mathbf{p}_1 + b\mathbf{p}_2 + c\mathbf{p}_3 \tag{2.11}$$

Die Werte für a, b und c können über die Verhältnisse

$$a \quad = \quad \frac{A(\mathbf{p}\mathbf{p}_2\mathbf{p}_3)}{A(\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3)}, \tag{2.12}$$

$$b \quad = \quad \frac{A(\mathbf{p}\mathbf{p}_1\mathbf{p}_3)}{A(\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3)} \tag{2.13}$$

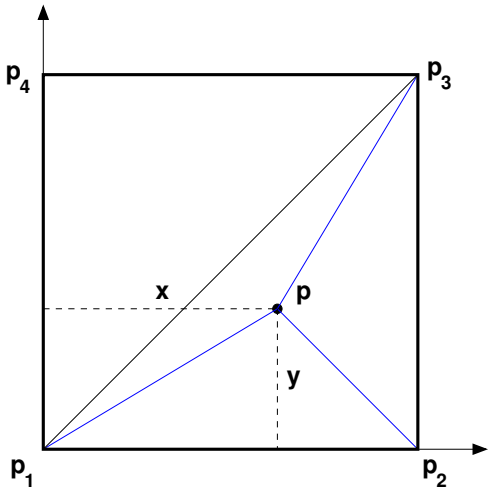


Abbildung 2.8: Rechteck mit Teildreiecken zur Berechnung der Texturkoordinaten

und

$$c = \frac{A(\mathbf{p}\mathbf{p}_1\mathbf{p}_2)}{A(\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3)} \tag{2.14}$$

berechnet werden. Dabei bezeichnet $A(\mathbf{lmn})$ die Fläche eines Dreiecks. Die Texturkoordinaten (u_f, v_f) für ein Fragment lassen sich nun wie folgt bestimmen. Es seien (u_1, v_1) die Texturkoordinaten von $\mathbf{p}_1$, (u_2, v_2) von $\mathbf{p}_2$ und (u_3, v_3) von $\mathbf{p}_3$. Dann ist

$$u_f = \frac{au_1/w_1 + bu_2/w_2 + cu_3/w_3}{aq_1/w_1 + bq_2/w_2 + cq_3/w_3} \tag{2.15}$$

und

$$v_f = \frac{av_1/w_1 + bv_2/w_2 + cv_3/w_3}{aq_1/w_1 + bq_2/w_2 + cq_3/w_3}. \tag{2.16}$$

q_1, q_2 und q_3 sind dabei die perspektivischen Korrekturfaktoren der Textur (siehe auch Kap. 4.1.2). w_1, w_2 und w_3 sind die homogenen Komponenten der Clip Koordinaten (Koordinaten nach Anwendung der Projektionsmatrix aber vor der perspektivischen Division) der projizierten Eckpunkte[3)]. Diese Berechnung gilt nur für Dreiecke. Daher müssen alle Polygone vor dem Berechnen der Texturkoordinaten in Dreiecke aufgespalten werden (Abb. 2.8).

Für das Nearest Neighbor Verfahren lassen sich mit diesen Texturkoordinaten (u_f, v_f) die Texelkoordinaten (i_f, j_f) des gesuchten Texels wie folgt bestimmen [88]:

$$i_f = \begin{cases} \lfloor u_f \rfloor, & s < 1 \\ 2^m - 1, & s = 1 \end{cases} \tag{2.17}$$

[3)] $w_i = 1 + \frac{z_i}{z_0}$, mit $i = 1, 2, 3$ und z_i als Tiefenwert eines Eckpunktes und z_0 als Tiefenwert des Fluchtpunktes

und

$$j_f = \begin{cases} \lfloor v_f \rfloor, & t < 1 \\ 2^n - 1, & t = 1 \end{cases}, \tag{2.18}$$

Dem Fragment wird nun der Farbvektor $\underline{C}_{ij}$ und der Alphawert α_{ij} des Texels mit den Koordinaten (i_f, j_f) zugeordnet.

Die Texturkoordinaten für ein Fragment werden bei der **gewichteten, bilinearen Interpolation** wie beim Nearest Neighbor Verfahren berechnet. Zur Bestimmung des Farbwertes $\underline{C}_f$ und Alphawertes α_f eines Fragments werden dann die vier Texel in der unmittelbaren Umgebung der Texturkoordinaten des Fragments verwendet. Dazu werden zunächst die Texelkoordinaten dieser vier Texel bestimmt:

$$i_0 = \lfloor u_f - 1/2 \rfloor, \tag{2.19}$$
$$j_0 = \lfloor v_f - 1/2 \rfloor, \tag{2.20}$$
$$i_1 = i_0 + 1, \tag{2.21}$$
$$j_1 = j_0 + 1. \tag{2.22}$$

Nun werden die Anteile für die Gewichtung der Nachbarn bestimmt:

$$\beta = \mathrm{frac}(u_f(x,y) - \frac{1}{2}), \tag{2.23}$$
$$\gamma = \mathrm{frac}(v_f(x,y) - \frac{1}{2}), \tag{2.24}$$

mit $\mathrm{frac}(x) = x - \lfloor x \rfloor$.

Damit berechnet sich der Farbvektor des Fragments durch

$$\underline{C}_f = (1-\beta)(1-\gamma)\underline{C}_{i_0 j_0} + \beta(1-\gamma)\underline{C}_{i_1 j_0} + (1-\beta)\gamma\underline{C}_{i_0 j_1} + \beta\gamma\underline{C}_{i_1 j_1} \tag{2.25}$$

und der Alphawert durch

$$\alpha_f = (1-\beta)(1-\gamma)\alpha_{i_0 j_0} + \beta(1-\gamma)\alpha_{i_1 j_0} + (1-\beta)\gamma\alpha_{i_0 j_1} + \beta\gamma\alpha_{i_1 j_1}. \tag{2.26}$$

Ein Beispiel eines texturierten Objekts bei bilinearer, gewichteter Interpolation ist in Abbildung 2.7 rechts zu sehen.

2.5.5 Texturmatrix

Die für einen Vertex angegebenen Texturkoordinaten werden vor dem Texture Mapping noch mit der *Texturmatrix* multipliziert. Die Texturmatrix ist als Identitätsmatrix initialisiert. Durch die Skalierung der Matrix ist es möglich, den Bereich der Texturkoordinaten

so festzulegen, dass die Texturkoordinaten Texel-exakt für die Eckpunkte eines Polygons angegeben werden können. Die Texturmatrix muss dazu wie folgt skaliert werden:

$$\mathbf{M}_{texel} = \begin{bmatrix} \frac{1}{2^m} \\ \frac{1}{2^n} \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2^m} & 0 & 0 & 0 \\ 0 & \frac{1}{2^n} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.27}$$

Daraus folgt für die Textur-Koordinaten innerhalb des Texturobjekts:

$$\{(s_{texel}, t_{texel}) \mid 0 \leq s_{texel} \leq 2^m,\ 0 \leq t_{texel} \leq 2^n\} \tag{2.28}$$

2.6 Rendering Contexts

Mit *Rendering Context* wird eine Instanz einer OpenGL-Statemachine bezeichnet. Dadurch ist es beispielsweise möglich, zwei Fenster mit voneinander unabhängigen OpenGL-Darstellungen zu realisieren, indem den Fenstern unterschiedliche Rendering Contexts zugeordnet werden. Zwischen Rendering Contexts kann gewechselt werden, ohne dass Zustandsänderungen des einen den anderen betreffen.

Unter Linux erfolgt die Erzeugung und Verwaltung von Rendering Contexts mittels GLX. Im Folgenden werden die dabei wesentlichen Aspekte kurz vorgestellt, für eine detaillierte Beschreibung sei auf die GLX-Spezifikation [103] verwiesen.

2.6.1 Einrichtung eines Rendering Contexts

Für die Einrichtung eines Rendering Contexts existieren folgende Möglichkeiten zur Konfiguration:

- **Farbmodus:** Rendering Contexts unterstützen hinsichtlich des Farbmodus entweder den *RGBA-Modus* oder den *Color-Index-Modus*. Beim RGBA-Modus wird die Zeichenfarbe anhand der Rot-, Grün- und Blauwerte spezifiziert, hinzu kommt der Alphawert. Beim Color-Index-Modus erfolgt die Farbauswahl anhand einer Tabelle (Color Map). In dieser Arbeit wird ausschließlich der RGBA-Modus verwendet.
- **Direct Rendering:** Beim Erzeugen eines Rendering Contexts muss spezifiziert werden, ob dieser grundsätzlich nur Indirect Rendering nutzen soll, oder ob Direct Rendering verwendet werden soll, falls dies vom Grafik-System unterstützt wird. Direct Rendering ermöglicht den direkten Zugriff auf die Grafikkarte durch den Treiber unter Umgehung des X-Protokolls.
- **Gemeinsame Nutzung von Display-Listen und Texturobjekten:** Display-Listen enthalten Folgen von OpenGL-Befehlen. Mit ihnen kann der Overhead durch Funktionsaufrufe reduziert werden. Sie können von verschiedenen Rendering Contexts gemeinsam genutzt werden. Auch Texturobjekte (vgl. Kap. 2.5.1) können von unterschiedlichen Rendering Contexts gemeinsam genutzt werden.

Bei der Einrichtung eines Rendering Contexts muss außerdem die Konfiguration der *GLX-Drawables* angegeben werden, die später in Verbindung mit dem Rendering Context verwendet werden sollen, da das Drawable und der Rendering Context miteinander kompatibel sein müssen. Dies ist der Fall, wenn beide denselben Farbmodus (d. h. RGBA oder Color-Index) unterstützen, wenn die Konfiguration hinsichtlich der Tiefe des Color-Buffers übereinstimmt und wenn sich der Rendering Context und das Drawable auf denselben X-Screen beziehen. Ein Drawable kann dabei ein Fenster (Window), eine Pixmap oder ein Pixel Buffer sein (s. Kap. 2.7).

2.6.2 Aktivieren eines Rendering Contexts

Um einen Rendering Context zu verwenden, muss er zum *aktuellen Rendering Context* (*current context*) des betreffenden Threads erklärt werden (`glXMakeCurrent`). Ein Thread kann zu jedem Zeitpunkt maximal einen aktuellen Rendering Context besitzen. Ein Rendering Context kann dabei nicht in mehreren Threads gleichzeitig der aktuelle Rendering Context sein.

2.7 Off-Screen Rendering

Neben dem Rendern zur Darstellung auf dem Bildschirm ist es möglich, in nicht sichtbare Bereiche des Framebuffers zu zeichnen. Dies wird als *Off-Screen Rendering* im Gegensatz zum *On-Screen Rendering* für das Rendern in sichtbare Bereiche des Framebuffers bezeichnet. Zwar ist auch der Backbuffer nicht am Bildschirm sichtbar, da er aber kein eigenständiger Framebuffer, sondern vielmehr Bestandteil des sichtbaren Framebuffers ist, fällt er nicht unter die Bezeichnung *Off-Screen Buffer*. Bei der Verwendung von Off-Screen Rendering ist es erforderlich, für den sichtbaren Framebuffer (*On-Screen Buffer*) und für den Off-Screen Buffer jeweils unterschiedliche Rendering Contexts zu benutzen (s. Kap. 2.6).

Unter Linux stehen seit GLX-Version 1.3 zwei Möglichkeiten des Off-Screen Renderings zur Verfügung: *Pixmaps* und *Pixel Buffer* (letzterer häufig auch kurz als "PBuffer" bezeichnet). In ihnen können auf die gleiche Weise wie in einem normalen Framebuffer Rendering-Operationen durchgeführt werden – mit dem Unterschied, dass die in einem Off-Screen Buffer erzeugten Bilddaten nicht direkt am Bildschirm dargestellt werden, sondern stattdessen anderweitig genutzt werden können. Die Generierung von Texturen zur Laufzeit, das so genannte *Dynamic Texturing*, ist eine der typischen Anwendungsmöglichkeiten des Off-Screen Renderings.

Ein Off-Screen Buffer zeichnet sich durch folgende Eigenschaften aus, die ihn von einem sichtbaren Framebuffer unterscheiden:

- **Maximale Größe unabhängig von der Bildschirmauflösung:** Ein Off-Screen Buffer ist nicht wie ein On-Screen Buffer auf die Größe der Bildschirmauflösung

beschränkt, sondern kann diese überschreiten, sofern genügend freier Grafikspeicher vorhanden ist.

- **Unabhängigkeit von der Konfiguration des sichtbaren Framebuffers:** Die Konfiguration des Off-Screen Buffers kann sich von der des sichtbaren Framebuffers unterscheiden. Einstellungen wie Farbtiefe, Anzahl der Color-Buffer (Single- oder Double-Buffering), etc. können somit unabhängig vom Modus des sichtbaren Framebuffers festgelegt werden.
- **Kein Pixel-Ownership-Test:** Bei Rendering-Operationen in einem sichtbaren Framebuffer wird stets geprüft, ob die jeweiligen Pixel am Bildschirm sichtbar sind (*Pixel-Ownership-Test*). Falls dies nicht der Fall ist, beispielsweise wenn das zugehörige Fenster verdeckt oder minimiert ist, werden die betreffenden Pixel in der Regel nicht verändert, d. h. das Rendering bleibt ohne Effekt. Bei einem Off-Screen Buffer existiert diese Einschränkung nicht.

In dieser Arbeit wird beim Off-Screen Rendering ausschließlich Gebrauch von Pixel Buffern gemacht, da diese gegenüber Pixmaps noch folgende, für die Geschwindigkeit entscheidende, Vorteile bieten:

- Pixel Buffer ermöglichen **hardware-beschleunigtes Rendering**, d. h. Rendering-Vorgänge in einem Pixel Buffer können von spezieller Grafikbeschleuniger-Hardware profitieren. Je nach Implementierung ist dies bei Pixmaps zum Teil nicht der Fall.
- Unter Linux unterstützen Pixel Buffer außerdem **Direct Rendering** (s. Kap. 2.6.1). Dies beschleunigt zusätzlich die Zeichenoperationen.

Im Folgenden wird daher die Einrichtung und Konfiguration eines Pixel Buffers, sowie seine Anwendung für Off-Screen Rendering näher betrachtet.

2.7.1 Einrichtung und Konfiguration eines Pixel Buffers

Die Einrichtung eines Pixel Buffers erfordert folgende Schritte:

1. **Auswahl einer Konfiguration**

   Die Konfiguration eines Pixel Buffers hinsichtlich Typ und Größe der Color- und Ancillary Buffer erfolgt über den so genannten `FBConfig`-Mechanismus ("FB" für "Framebuffer"), der von GLX ab der Version 1.3 angeboten wird. Dabei handelt es sich um eine komfortable Möglichkeit, nach bestimmten vorzugebenden Kriterien eine Konfiguration für den Pixel Buffer (oder auch für andere Drawables) zu ermitteln.

   Der Befehl `glXChooseFBConfig` liefert eine Liste aller vom Grafiksystem angebotenen Konfigurationen, die den Vorgaben entsprechen. Eine komplette Liste möglicher Vorgaben ist in [103] enthalten. Die für diese Arbeit relevanten Attribute sind:

   - der zu verwendende **Farbmodus**, d. h. RGBA oder Color-Index,

- die minimale **Tiefe des Color-Buffers**, d. h. die Mindestanzahl der Bits, die für die Speicherung der Farbwerte (und gegebenenfalls Alphawerte) pro Pixel verwendet werden sollen,
- der Typ des Bufferings, d. h. ob **Single-** oder **Double-Buffering** verwendet werden soll,
- die minimale **Größe des Depth-Buffers**, d. h. die Mindestanzahl der für die Speicherung der Tiefenwerte zu verwendenden Bits pro Pixel,
- eine untere Schranke für die maximal mögliche **Breite und Höhe des Pixel Buffers**, die in dieser Konfiguration festgelegt ist.

2. **Erzeugen eines Pixel Buffers**

   Basierend auf der gewählten Konfiguration des Framebuffers kann ein Pixel Buffer erzeugt werden (`glXCreatePbuffer`). Hierbei ist die Angabe folgende Parameter erforderlich:

   - **Größe des Pixel Buffers:** Die Größe (d. h. Höhe und Breite in Pixeln) ist zum einen durch den zur Verfügung stehenden Grafik-Speicher beschränkt, zum anderen sind meistens auch implementierungsabhängige Maximalgrößen vorgegeben. Falls die Angaben diese Grenzen überschreiten, kann die Erzeugung eines Pixel Buffers scheitern.
   - **Preserved oder Unpreserved Contents:** Diese Option legt fest, wie verfahren werden soll, falls während des Programmablaufs der Grafikspeicher zur Neige geht und dadurch ein Ressourcen-Konflikt eintritt. Falls *Preserved Contents* gewählt wurde, werden in einem solchen Fall Teile des Pixel Buffers vom Grafikspeicher in den Hauptspeicher ausgelagert. Falls dagegen die Option *Unpreserved Contents* gewählt wurde, können Teile des Pixel Buffers vom Grafik-Treiber überschrieben werden und somit verloren gehen. Sowohl bei Auslagerung als auch bei Überschreiben wird ein entsprechendes X-Event ausgelöst.

3. **Erzeugen eines Rendering Contexts**

   Als letzter Schritt muss ein Rendering Context erzeugt werden, dessen Konfiguration mit der des Pixel Buffers kompatibel ist (s. Kap. 2.6).

2.7.2 Verwenden eines Pixel Buffers

Damit ein Pixel Buffer für Off-Screen Rendering verwendet werden kann, muss der zugehörige Rendering Context zum aktuellen Context erklärt und der Pixel Buffer als zu verwendendes Drawable zugeordnet werden (siehe auch Kapitel 2.6). Alle daraufhin ausgeführten OpenGL-Befehle beziehen sich ab diesem Zeitpunkt auf den Rendering Context des Pixel Buffers.

Eine Möglichkeit, Bilddaten vom Pixel Buffer in den sichtbaren Framebuffer zu übertragen, besteht darin, diese aus dem Pixel Buffer in den Hauptspeicher zu kopieren (`glReadPixels`) und anschließend in den sichtbaren Framebuffer zu schreiben (`glDrawPixels`) oder als Textur zu laden. Dieser zweifache Transfer der Daten zwischen Grafikspeicher und Hauptspeicher ist jedoch ineffizient.

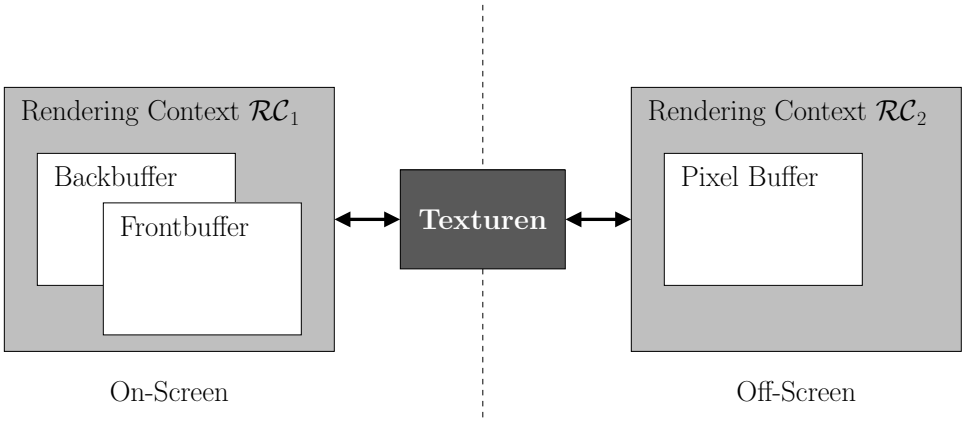


Abbildung 2.9: On-Screen- und Off-Screen Buffer mit gemeinsam genutzten Texturen

Statt dessen gibt es als *Schnittstelle* zwischen Pixel Buffer und sichtbarem Framebuffer die von den jeweiligen Rendering Contexts gemeinsam genutzten Texturobjekte. So können zum Beispiel Bildbereiche des Pixel Buffers in Texturen abgespeichert werden, welche anschließend auch im sichtbaren Framebuffer verwendet werden können. Abbildung 2.9 stellt diesen Sachverhalt schematisch dar. Diese Methode ist deutlich effizienter, da hier kein zeitaufwändiger Umweg über den Hauptspeicher erforderlich ist. In Kapitel 6.2 wird darauf noch ausführlich eingegangen.

2.7.3 Löschen eines Pixel Buffers

Wird ein Pixel Buffer nicht mehr benötigt, so kann er gelöscht werden (`glXDestroyPbuffer`), um den durch ihn belegten Grafikspeicher wieder freizugeben. Dass dies sinnvoll ist, wird bei Betrachtung des Speicherbedarfs eines Pixel Buffers deutlich.

Bestandteile des Pixel Buffers	**Speicherbedarf (Bit pro Pixel)**
RGBA-Colorbuffer, Front	$4 \times 8 = 32$
RGBA-Colorbuffer, Back	$4 \times 8 = 32$
Depth-Buffer	24
Stencil-Buffer	8
gesamt	96

Tabelle 2.2: Typischer Speicherbedarf eines Pixel Buffers

Ein Pixel Buffer in einer für viele Anwendungen typischen Konfiguration hat beispielsweise, wie in Tabelle 2.2 dargestellt, einen Speicherbedarf von 96 Bit pro Pixel. Im Falle

eines 1024×1024 Pixel großen Pixel Buffers dieser Konfiguration werden somit insgesamt 12 Megabyte benötigt.

2.8 Programmierbare Grafik-Hardware

Die Funktionalität von OpenGL (und auch anderen 3D-Grafik-APIs) hat in den letzten Jahren immer mehr zugenommen. Dies beschränkte sich zuächst auf die Erweiterung der Funktionalität der klassischen *Fixed Function Pipeline* mit der Kontrolle über zusätzliche Zustandsvariablen. Dabei ist hier vor allem die in OpenGL 1.3 mit *Multi Texturing* eingeführte Möglichkeit erwähnenswert, für ein Polygon mehrere Texturen zu verwenden und (zum damaligem Zeitpunkt in beschränktem Maße) zu kombinieren. Zunehmende Konfigurierbarkeit hat die Möglichkeiten der Render-Algorithmen gesteigert. Für diese Arbeit relevant sind hier vor allem die in Kapitel 5.2.3 beschriebenen *Register Combiners*.

Ab dem Jahr 2001 wurden die ersten Grafikkarten zum Teil programmierbar [58]. Dabei wird die Pipeline-Stufe mit den Per-Vertex Operationen optional durch *Vertex Shader* ersetzt: kleine Programme, die pro Vertex dessen Parameter (z. B. Position und Normalenvektor) manipulieren. Seit etwa 2002/2003 sind auch die Fragment-Operationen der Grafik-Pipeline optional programmierbar. In diesen für diese Arbeit interessanten *Fragment Shaders* wird seit 2004 teilweise auch eine dynamische Flusskontrolle unterstützt.

Die Programmierung erfolgte zunächst über hersteller-abhängige Erweiterungen [64], später entstand eine hersteller-unabhängige assembler-ähnliche Sprache für OpenGL (`GL_ARB_vertex_program, GL_ARB_fragment_program`). Inzwischen kommen größtenteils Hochsprachen zum Einsatz: *HLSL* (*High-Level Shader Language*, Microsoft)[24], C_g (*C for graphics*, NVIDIA)[60] und *GLSL* (*OpenGL Shading Language*)[47]. Alle drei besitzen eine C-ähnliche Syntax mit ein paar C++-Erweiterungen. Neben anderen Änderungen zu C sind besonders die eingebaute Unterstützung für Vektortypen und grafik-orientierte Funktionen, wie Kreuzprodukte und Matrix-Multiplikationen, hervorzuheben. In dieser Arbeit kam GLSL zum Einsatz. Diese Shader-Sprache wurde vom ARB in die Spezifikation zu OpenGL 2.0 aufgenommen.

Ein Fragment Shader berechnet jeweils für ein einzelnes Fragment dessen Farbe $\underline{C}_f$ und Alphawert α_f, sowie optional auch Tiefe z_f. Als Eingangsdaten dienen vor allem die Ergebnisse der Rasterung, wie die interpolierten Farbwerte der Vertices und Texturkoordinaten. Zusätzlich können benutzerdefinierte Daten an den Shader übergeben werden. Innerhalb eines Shaders kann weiterhin auf viele Zustandsvariablen des aktuellen Rendering Contexts lesend zugegriffen werden ("uniforms"). Interessant für diese Arbeit sind besonders die Zugriffe auf Texturen. Dabei sind auch mehrfache Zugriffe auf mehrere Texturen möglich. Das Ergebnis eines Texturzugriffs kann auch als Texturkoordinaten für einen weiteren Zugriff verwendet werden ("dependent texture reads"). Zu beachten ist, dass (um Parallelisierung auf der GPU zu ermöglichen) innerhalb eines Shaders keine Zugriffe auf andere Fragmente möglich sind. Eine genaue Beschreibung der Syntax von GLSL

und das Zusammenspiel mit OpenGL ist in [47] und [85] zu finden. Die Schnittstellen eines Fragment Shaders sind in Abbildung 2.10 skizziert.

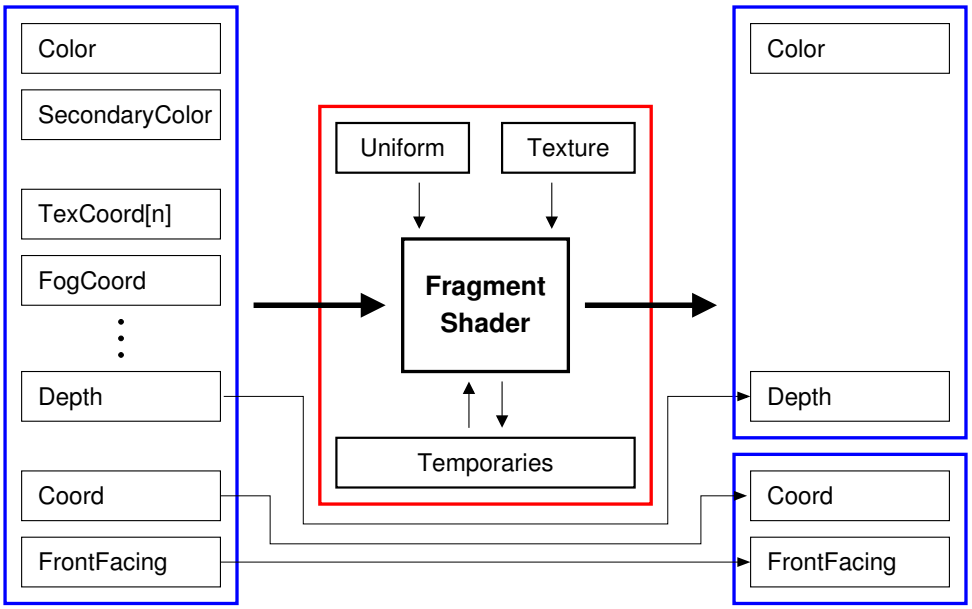


Abbildung 2.10: Schnittstellen eines Fragment Shaders

Ein Überblick über die historische Entwicklung programmierbarer Grafik-Hardware ist in [28] zu finden.

3 Verwandte Arbeiten

3.1 Prädiktive Displays

Der Einfluss von Latenzenzzeiten im Übertragungskanal der Kontrollschleife von Telepräsenz- und Teleaktionssystemen ist ein bereits seit langem bekanntes Problem [29]. Lane et al. zeigten in einer Fallstudie, dass Zeitverzögerungen von mehr als einer Sekunde die Performanz des Benutzers deutlich reduzieren [52]. Aber bereits geringere Verzögerungen von etwa 300 ms führen zu einer "Move and Wait"-Strategie [25], d. h. der Operator wartet nach einer Aktion zunächst das visuelle Ergebnis ab, bevor er ein neues Kommando gibt.

Neben anderen Ansätzen, wie beispielsweise halbautonomen Teleoperatoren [96], sind *prädiktive Displays* (s. Kap. 1.1) ein vielversprechender Ansatz zur Kompensation von Übertragungszeiten [4]. Es wurden verschiedene Varianten von prädiktiven Displays vorgeschlagen, die beispielsweise Kamerabildern Computergrafik überlagern (*Augmented Reality*) [49, 99]. Bei anderen Ansätzen wurden die Kamerabilder vollständig durch synthetische Ansichten ersetzt (*Virtual Reality*) [39].

In [59, 61, 62] wurde versucht die Übertragungszeit zu kompensieren, wenn Betrachter und Rendering Engine (Grafik-Hardware) einer 3D-Darstellung durch ein Netzwerk getrennt sind. Die Erzeugung vollständig gerenderter Bilder auf der Remote-Seite ist dabei seltener als in Framerate erforderlich. Sobald diese Ansichten vorhanden sind, werden sie inklusive Tiefenkarte zur Betrachter-Seite übertragen. Die Darstellung in Framerate erfolgt dann durch die Berechnung von Zwischenbildern dieser Ansichten mittels Warping. Dabei wird der Z-Buffer der Bilder genutzt, um Verzerrungen durch Tiefenwerte der Bildpunkte (Disparitäten) zu korrigieren. Die Benutzerposition bestimmt dann die Betrachterposition aus der die gewarpten Bilder lokal beim Benutzer generiert werden. Um Löcher durch Aufdeckungen zu minimieren, wird die resultierende Ansicht aus mehreren Referenzbildern erzeugt. Interaktionen mit der Remote-Umgebung sind mit diesem Ansatz natürlich nicht möglich. Eine weitere Anwendung des Systems ist die Erhöhung der Framerate bei der Darstellung komplexer Szenen.

Ein zweistufiges prädiktives Display wird in [42] vorgestellt. Dabei handelt es sich um einen bildbasierten Ansatz mit unkalibrierten Kameras und vorab unbekanntem Manipulator-Setup. Die Prädiktion erfolgt mittels Zuordnung von Motor-Kommandos zu erhaltenen Kamerabildern. Bei wenigen Trainings-Daten wird zunächst eine den Kamerbildern überlagerte Drahtgitter-Darstellung angezeigt. Sobald die Schätzung des Modells genau genug ist, wird die prädizierte Ansicht aus den gespeicherten Kamerabildern generiert.

Das in [19] vorgestellte fotorealistische prädiktive Display soll etwas genauer vorgestellt werden. Mit Structure-From-Motion wird dabei vorab mit einer unkalibrierten Kamera ein Modell aufgebaut. Basierend auf ein paar markanten Vierecken wird mit auf 3D eweitertem SSD-Tracking (Sum of Squared Differences) die Roboterposition verfolgt. Während das Tracking nur mit ausgewählten Vierecken durchgeführt wird, erfolgt das Rendern mit der etwas feiner triangulierten Szene. Dabei werden die mit Structure-From-Motion ermittelten geometrischen Feinheiten durch die Verwendung von blickwinkel-abhängigen Texturen berücksichtigt (vgl. Kap. 4.4.2). Die Differenzen zwischen erwartetem Bild und resultierendem Bild bei der Initialisierung liefern die Differenzen, d. h. Verschiebungen in den Texturen. Das gewichtete Blending der aus den Kamerabildern erhaltenen Texturen bei der Darstellung ist mithilfe von Register Combiners realisiert. Eine Aktualisierung der Texturen im laufenden Betrieb erfolgt in der genannten Arbeit nicht.

3.2 Fotorealistische Computergrafik auf Basis von Kamerabildern

Die Verwendung von Kamerabildern für die Darstellung fotorealistischer Szenen ist für viele Aufgaben sinnvoll. In diesem Kapitel sollen einige der dabei häufig verwendeten Methoden kurz vorgestellt werden. Weiterhin werden einige Anwendungen, die eine mit dieser Arbeit vergleichbare Zielsetzung haben, anhand von ausgewählten Projekten beschrieben.

Das Erzeugen fotorealistischer Ansichten auf Basis von Kamerabildern wird häufig als *Image-Based Rendering (IBR)* bezeichnet. 1993 beschrieben Chen und Williams die Interpolation zwischen Kamerabildern mithilfe des optischen Flusses [17]. Diese Methode erzeugt jedoch nur für nahe beisammen liegende Referenzilder gute Ergebnisse. Seitz und Dyer interpolierten ebenfalls zwischen zwei Bildern, beschränkten sich aber dabei auf Positionen auf der Linie zwischen den beiden optischen Zentren [90]. Eine einfache Variante des IBR ist die Verwendung von Panoramabildern, wie es in *Quicktime VR* verwendet wird [16]. Je nach Blickrichtung wird ein entsprechender Ausschnitt des Bildes gewonnen, passend verzerrt und dargestellt. Diese Methode ist jedoch auf diskrete Betrachterpositionen beschränkt, lediglich die Rotation kann jeweils variiert werden.
Viele Verfahren sind auf das in [68] vorgestellte *Plenoptic Modeling* zurückzuführen. Die plenoptische Funktion ist ein mathematisches Modell zur vollständigen Beschreibung der Beleuchtung einer Szene mit Sichtstrahlen. Ein Sample der plenoptischen Funktion ist somit eine sphärische Karte an einem bestimmten Betrachtungspunkt. Vorteilhaft bei dieser Repräsentation ist die Unabhängigkeit der Komplexitität von der Szene (z. B. Anzahl der Polygone) und dass keine geometrischen Informationen für eine Darstellung erforderlich sind. Ein großer Nachteil dieser Beschreibungsform sind jedoch die großen Datenmengen, die die Darstellung dynamischer Szenen erschweren. *Light Field Rendering* [57] und *Lumigraph* [35] basieren auf einer Reduktion der plenoptischen Funktion auf 4 Dimensionen, indem nicht mehr berücksichtigt wird, wo sich auf dem jeweiligen Sichtstrahl die Betrachterposition befindet. Während Light Fields dabei ohne jegliche geometrische Informationen auskommen, basiert der Lumigraph auf einer groben geometrischen Beschrei-

bung. Beide haben den Nachteil, dass Objekte nur von außerhalb einer konvexen Hülle bzw. Szenen nur von innerhalb einer solchen betrachtet werden können. Als interessanter Aspekt sei hier noch erwähnt, dass beim Lumigraphen die durch die diskrete Anzahl an Samples im Datensatz verbleibenden "Löcher" ähnlich der in Kapitel 5.2.2 beschriebenen Methode gefüllt werden, jedoch ohne Hardware-Beschleunigung. Methoden wie *Relief Textures* verwenden zusätzlich zu den Farbwerten der Pixel noch Tiefenwerte [74]. Aus ähnlichen Positionen können dann mittels Warping neue Ansichten erzeugt werden.
Die genannten Ansätze sind für das in dieser Arbeit beschriebene Display jedoch nicht verwendbar, da für die Prädiktion ein geometrisches Modell benötigt wird. Ein weiteres Argument für die Verwendung einer polygon-basierten Oberflächenbeschreibung ist die Optimierung aktueller Grafikkarten auf die Verarbeitung von Dreiecken. Eine Variante des IBR zur Darstellung eines einzelnen Objektes mithilfe von darum verteilten Kameras sind *Visual Hulls* [63]. Diese ähneln einer konvexen Hülle um das Objekt und werden durch Schnitt der aus den Silhouetten des Objekts in den Kameraaufnahmen entstehenden Konen gebildet. Die Bestimmung erfolgt in der genannten Arbeit unter Berücksichtigung der Epipolargeometrie im 2D-Bildraum. Für die Darstellung dienen die Referenzbilder der Kameras als Texturen, wobei blickwinkel-abhängige Texturierung verwendet wird. Ein klarer Nachteil dieses Ansatzes ist seine Beschränkung auf ein einzelnes Objekt.

Ein mittlerweile sehr bekanntes Projekt ist die von Kanade et al. entwickelte *Virtualized Reality*^{TM} [44, 45]. Dieses Projekt hat sich zum Ziel gemacht Betrachter einer realen dynamischen Szene nicht auf die Wahrnehmung der Bilder einzelner Kameras einzuschränken, sondern sie statt dessen beliebige Betrachterpositionen einnehmen zu lassen. Somit findet hier zwar keine zeitliche Prädiktion statt, die Problemstellung ist aber nicht weit entfernt von der prädiktiver Displays, wenn sie als "räumliche Prädiktion" angesehen wird. Dazu ist die Szene von einer großen Anzahl kalibrierter und synchronisierter Kameras umgeben. Die Szene wird zunächst für jeden Zeitpunkt mittels Multibaseline-Stereorekonstruktion als Tiefenkarte pro Kamera erfasst. Diese Tiefenkarten werden anschließend in einem globalen 3D-Modell, bestehend aus Voxel- und Oberflächenrepräsentationen, fusioniert. Zu einem späteren Zeitpunkt kann die dynamische Szene dann als Wiederholung aus beliebigen Blickwinkeln betrachtet werden. Die Kamerabilder werden dabei für eine blickwinkel-abhängige Texturierung der Darstellung verwendet. Zu diesem Zweck erfolgt eine Überlagerung der Kamerabilder bei jeweiliger Gewichtung entsprechend des Betrachtungswinkels. Vor der Darstellung ist auch noch eine Manipulation der Szene möglich. Eine schritthaltende Verarbeitung ist hier aufgrund der zeitlich versetzten Wiedergabe nicht erforderlich. Beim Superbowl kam diese Technik bereits mithilfe von über 30 Kameras zum Einsatz. Dabei kontrollierte ein Benutzer Pan-Tilt-Einheit und Zoom einer der Kameras ("master camera"), die anderen Kameras folgten der gleichen fokussierten Stelle auf dem Feld. In Wiederholungen konnten dann Szenen des Spiels aus beliebigen Blickwinkeln gezeigt werden, z. B. aus der Sicht des Balls.

Auch andere Forschungsgruppen beschäftigen sich mit *3D-Video* bzw. *3D-TV*, beschränken sich jedoch in der Regel auf kleinere Szenen. In [14] werden Texturen extrahiert, um 3D-Videos menschlicher Darsteller zu erstellen. Während die Darstellung in Echtzeit erfolgt, findet die Rekonstruktion auf Basis von mehreren statischen kalibrier-

ten Kameras offline statt. Die Person wird duch ein Dreiecksnetz mit Skelett-Kinematik beschrieben, das aus 16 Körperteilen besteht. In einer Initialisierungsphase werden die personen-abhängigen Skalierungen der Körperteile und ihre Startpositionen ermittelt. Durch Hintergrund-Subtraktion wird später pro Bild die Silhouette der Person gewonnen. Die Parameter der Körperposition werden dann so bestimmt, dass die Überlappung der projizierten Modell-Silhouette mit der gemessenen maximal ist, wobei die Überdeckung mithife des Stencil-Buffers auf der Grafikkarte ermittelt wird (XOR-Verknüpfung). Für jeden Zeitpunkt werden dann Texturen durch gewichtete Überblendung aller Kamerabilder erzeugt. Da das Modell nur eine Approximation ist, können Bereiche des Hintergrunds fälschlicherweise auf die Texturen der Person abgebildet werden (vgl. Kap. 5.1.1). Als Lösung werden hier zunächst die äußersten Pixel der Silhouette entfernt, anschließend werden ihnen die Werte der am nächsten liegenden Vordergrund-Pixel zugewiesen. Die Sichtbarkeitsberechnung in jedem Kamerabild erfolgt pro Vertex. Um Artefakte durch Fehler im Modell zu reduzieren, wird pro Vertex zusätzlich von einer Reihe leicht versetzter Positionen aus die Sichtbarkeit getestet. Der Vertex wird nur als sichtbar markiert, wenn er in allen diesen Ansichten sichtbar ist.

Eine den 3D-Videos ähnliche Anwendung sind *Video-Konferenzen* mit freier Auswahl des Blickwinkels. Dadurch soll der Eindruck vermittelt werden, dass sich alle Teilnehmer tatsächlich im selben Raum befinden. Unter anderem kann damit auch das Problem von Videokonferenzen gelöst werden, sich gegenseitig nicht in die Augen sehen zu können, wenn sich die Kameras jeweils über den Bildschirmen befinden. In [18] wurde zunächst ein System mit einer statischen Szene realisiert. Das polygonale Modell wurde mit einem 3D-Laserscanner und anschließender Triangulation aufgebaut. Danach wurde anstelle des Scanners eine kalibrierte Kamera montiert, die mehrere Bilder bei unterschiedlichen Azimuthwinkeln aufnahm. Diese Prozedur wurde an insgesamt vier Positionen bei einer jeweiligen Dauer von etwa 30 Minuten durchgeführt. Für die Darstellung wurden die Kamerabilder als projektive Texturen auf die Dreiecke gelegt. Blickwinkel-abhängige Faktoren in den Texturen wurden aufgrund der relativ diffusen Szene nicht berücksichtigt. Bei Verwendung je einer SGI InfiniteReality2 pro Auge wurden 30 Hz Framerate erreicht. Die Anzeige erfolgte dabei über ein 3D-Display von etwa 1,3 mal 1 Meter.
In späteren Arbeiten sollten dann auch dynamische Szenen präsentiert werden, wobei hier zehn oder mehr Kameraeinheiten mit jeweils drei Kameras zum Einsatz kamen [46, 101]. Im Gegensatz zu den bisher beschriebenen Anwendungen ist hier neben flüssiger Darstellung auch eine schritthaltende Rekonstruktion zwingend erforderlich, um Latenzen zu minimieren. Mittels korrelations-basiertem Stereo aus jeweils zwei monochromatischen Kameras werden dabei Tiefenkarten erstellt, wobei auch Tests auf einem aus 3.000 Alpha-Prozessoren gebildeten Cluster durchgeführt wurden. Die jeweils dritte Kamera nimmt Farbbilder auf und ordnet daraus den einzelnen 3D-Punkten Farbwerte zu. Die Tiefenkarten mit Farbwerten werden über UDP zu den Rendering-Systemen der anderen Konferenzteilnehmer übertragen, wo die Tiefenkarten unter Verwendung des Z-Buffers als Punktewolken dargestellt werden. Ein Problem bei der Darstellung sind die großen Datenmengen: bei VGA-Auflösung ergeben sich mit zehn Kameraeinheiten etwa 75 Millionen Punkte/Sekunde. Daher konnten auch mit einem aus drei PCs gebildeten Cluster nur etwa

2 Millionen Punkte/Sekunde bei 30 Hz dargestellt werden. Die Anzeige erfolgte asynchron zur Aktualisierung der Tiefenkarten.

Ein ähnliches Ziel verfolgt das Projekt *blue-c* der ETH Zürich [36]. Dort wurde ein immersives Display, ähnlich einer CAVE^{TM} [21], aus drei Projektionsflächen aufgebaut, wobei hier jedoch zusätzlich Bildaufnahmen erfolgen müssen, um eine Kollaboration zwischen mehreren solcher "Portale" zu ermöglichen [95]. Die Stereo-Darstellungsflächen können dazu zwischen lichtundurchlässig und transparent umgeschaltet werden. Die Transparent-Schaltung erfolgt zu festen Zeitpunkten zwischen der Darstellung für linkes und rechtes Auge, d. h. die Shutterbrille des Benutzers ist vollständig geschlossen. Dabei nehmen 16 hinter den Projektionsflächen befindliche kalibrierte VGA-Kameras synchronisiert Bilder bei aktiver Beleuchtung auf. Darauf basierend wird eine 3D-Repräsentation des Benutzers als Punkte-Darstellung in Echtzeit berechnet [105]. Mittels Hintergrund-Subtraktion und anschließender Extraktion der Silhouette kommt eine Variante der Image-Based Visual Hulls zum Einsatz [63]. Um die räumlich-zeitlichen Kohärenzen aufeinander folgender Kamerabilder von mehreren Kameras auszunutzen, wird hier ein differentielles Aktualisierungs-Schema für die Punkt-Darstellung verwendet. Dabei werden aus den Referenzbildern Farb- und/oder Geometrieänderungen ermittelt und nur diese in einem gemeinsamen 3D-Video-Stream übertragen, wodurch etwa 75% der Datenmenge eingespart wird. Zur Beschleunigung der Verarbeitung werden außerdem nur diejenigen Kamerabilder verarbeitet, die der Betrachterposition am nächsten liegen. Kontrollinformationen werden zwischen den Portalen über CORBA verschickt, die Streaming-Daten über UDP. Die Latenz des kompletten Systems liegt zwischen drei und fünf Frames bei einer Rekonstruktions-Rate von fünf bis neun Frames/Sekunde.
Eine Nutzung des Systems für virtuelle Verkaufsräume mit getrennten Portalen für Käufer und Verkäufer wurde mit dem *IN:SHOP*-Konzept vorgestellt [54]. Wie beim IBR ist ein Nachteil der Darstellung von Punktewolken, neben den oft sehr großen Datenmengen, dass sie, im Gegensatz zu Polygondarstellungen, auf aktuellen Grafikkarten nicht optimal von der Hardware unterstützt werden. Bei polygonaler Beschreibung kann die geometrische Komplexität, falls erforderlich, reduziert werden, während gleichzeitig Farbinformationen in Form von Texturen erhalten bleiben.

Debevec et al. kombinieren IBR mit geometrie-basiertem Rendering für die Darstellung architektonischer Szenen auf Basis von ein paar Kameraaufnahmen [22, 23]. Das polygonale Modell wird vom Benutzer interaktiv, unter anderem durch Zuordnung von Kanten in Kamerabildern zu Modellkanten, erstellt. Ein Fokus dieser Arbeiten liegt auf einer blickwinkel-abhängigen Texturierung von Polygonen (s. Kap. 4.4.2). Zuvor wird jedes Polygon in kleinere Polygone unterteilt, die in den Kamerabildern immer jeweils vollständig oder überhaupt nicht sichbar sind. Dazu wird unter anderem die Szene unter Verwendung des Z-Buffers und mit Kodierung der IDs der Polygone in ihren Farben aus der jeweiligen Kamerasicht gezeichnet. Eine Abtastung des resultierenden Bildes ergibt, an welchen Stellen nicht das erwartete Polygon zu sehen ist (andere ID) und somit eine Unterteilung notwendig ist. Für Polygone, die in keinem Kamerabild sichtbar waren, sind keine Farbinformationen vorhanden. Um dies zu verbergen, werden sie mit Gouraud-Shading gefüllt, indem den Vertices eine Art Mittelwert der angrenzenden Polygone zugeordnet

werden. Geometrische Details werden schließlich noch automatisch durch "model-based-stereo" bestimmt. Ein zweites Bild (offset image) wird dazu auf das Modell aus der Sicht des Referenzbilds (key image) projiziert (warped offset image) und mit diesem korreliert. Die resultierenden Tiefenkarten können mit der Methode aus [17] beim Rendern der texturierten Polygone verwendet werden.

Mit der bereits genannten *Augmented Reality* wird im visuellen Kontext die Erweiterung der realen Sicht um virtuelle Komponenten bezeichnet. Da somit auch dort das Zusammenspiel von Realität und VR von Bedeutung ist, werden in diesem Kontext viele zum beschriebenen Projekt ähnliche Themenstellungen bearbeitet. Im von 1999 bis 2003 vom BMBF geförderten Leitprojekt *ARVIKA* wurden "Augmented-Reality-Technologien zur Unterstützung von Arbeitsprozessen [...] erforscht und realisiert" [2]. Das auf den Ergebnissen von ARVIKA aufbauende BMBF-Projekt *ARTESAS* (Advanced Augmented Reality Technologies for Industrial Service Applications) hat die Erforschung und Erprobung von Augmented Reality Basistechnologien für den Einsatz im industriellen Service-Umfeld zum Ziel [1].

3.3 Ausführung von Algorithmen auf GPUs

Die Ausführung von Algorithmen auf der GPU ist ein wichtiger Bestandteil dieser Arbeit. Daher sollen im Folgenden einige Projekte vorgestellt werden, die ebenfalls die Grafik-Hardware zur Beschleunigung nutzen, wobei jeweils interessante Aspekte hervorgehoben werden.

Bereits bevor die Grafikprozessoren programmierbare Pipeline-Stufen enthielten, gab es Ansätze sie für Anwendungen außerhalb der Computergrafik-Domäne zu verwenden. So wurde beispielsweise in [40] die Grafik-Hardware für die Approximation von verallgemeinerten *Voronoi-Diagrammen* genutzt. Dazu werden die Distanzfunktionen der Primitive auf Polygone abgebildet und diese mit jeweiliger ID als Farbe gezeichnet. Ein Punkt-Primitivum wird beispielsweise als aus Polygonen genäherter Kegel bei Betrachtung von oben gezeichnet. Die durch die Rasterung diskreten Distanz-Vergleiche zwischen den Primitiva werden dabei mithilfe des Z-Buffers durchgeführt. Die Farbe jedes Rasterpunkt gibt anschließend an, in welcher Voronoi-Region er sich befindet, die Tiefe entspricht der Distanz zum nächsten Primitivum. Als Beispiel wurde unter anderem eine schnelle Bewegungsplanung zur Hindernisvermeidung in statischen und dynamischen Umgebungen unter Verwendung einer Potentialfeld-Methode vorgestellt [41].

Durch die Entwicklung der Grafikprozessoren von Pipelines mit konfigurierbarer aber fester Funktionalität hin zu programmierbaren Floating-Point-Pipelines entstand auch die Möglichkeit komplexere Algorithmen auf die Grafik-Hardware auszulagern. Dabei wird die hochparallele Verarbeitung der typischerweise als SIMD-Architekturen (Single Instruction, Multiple Data) realisierten GPUs ausgenutzt. Eine beliebige Interpretation der Ergebnisse von Shadern ermöglicht dabei vielfache Anwendungsmöglichkeiten in unterschiedlichen Gebieten. Unter *GPGPU* (*General-Purpose computation on GPUs*) [33] sind

Projekte zusammengefasst, die sich mit der Nutzung von GPUs für andere Anwendungen als Computergrafik beschäftigen. Dabei wird die hohe Rechenleistung der GPUs genutzt, indem sie als Coprozessoren fungieren.

Mit *Sh* wurde eine vollständig in C++ eingebettete *Meta-Programmiersprache für moderne GPUs* entwickelt, die vor allem für grafische und numerische Anwendungen gedacht ist [66]. Statt einer Trennung von Anwendungs- und Shader-Code ist hier das Ziel ein gemeinsames Programmiermodell für CPUs und GPUs. Dadurch sollen einfachere Implementierungen, direktere Interaktion, beispielsweise beim Allozieren von Texturen, sowie Generierung und Manipulation von Shadern zur Laufzeit ermöglicht werden. Durch die Einbindung in C++ können dort verwendete Konstrukte, wie beispielsweise für den Kontrollfluss, auch genutzt werden, um ähnliche Fähigkeiten für die Shader zur Verfügung zu stellen.
Die in [67] beschriebene *Shader Algebra* verwendet Sh für die Modularisierung von Shadern. Zu diesem Zweck können Shader als Objekte mit zwei Arten von Operatoren verknüpft werden, die durch Überladen in C++ eingebunden sind: "Connections" verbinden die Ausgänge eines Shaders mit den Eingängen eines weiteren, "Combinations" bilden durch Konkatenation der Ein- und Ausgänge aus zwei Shadern einen einzelnen. Erweiterungen der Shader Algebra um Streams ermöglichen ein Modell für die allgemeine Verarbeitung von Streams. Texturen fungieren in diesem Zusammenhang beispielsweise als Arrays, beim Speicher-Management werden automatisch Off-Screen Buffer genutzt.
Mit *Brook for GPUs* [7] wurde 2004 ebenfalls eine Sprache für die Verarbeitung von Streams vorgestellt. Diese erweitert C um einfache Konstrukte für parallele Daten bei Verwendung der GPU als Streaming-Coprozessor. Sie arbeitet als Präprozessor, der Brook-Programme auf C++/C_g Implementierungen abbildet. Fragment Shader fungieren dabei als Kernels zur Verarbeitung der Streaming-Daten. Werden bei der Anzahl der Ausgabewerte des Shaders die Beschränkungen der Hardware überschritten, so wird der Kernel auf mehrere Durchläufe aufgeteilt ("pass-splitting"). Die Ausführung eines Kernels erfolgt, wie bei vielen GPGPU-Anwendungen, durch Zeichnen eines Rechtecks im Off-Screen Buffer. Bei Messungen wurden bis zu sieben mal schnellere Ausführungszeiten im Vergleich zu einer reinen Umsetzung auf der CPU erreicht. Diese Arbeit enthält auch eine, wenn auch eher theoretische, mathematische Betrachtung, ab wann sich die Verarbeitung eines Kernels auf der GPU unter Berücksichtigung der Übertragungszeit der Daten zwischen Hauptspeicher und GPU lohnt. Eine Untersuchung zum Einfluss der Transferzeit bei Verwendung der GPU ist auch in [43] zu finden.

Thompson et al. erstellten eine *Analyse* zum Einsatz von Grafik-Hardware für General-Purpose Computing [100]. Als konkrete Anwendung verwendeten sie Vertex Shader zur Beschleunigung von beispielsweise Matrix-Multiplikationen. Bei der Verarbeitung großer Vektoren erreichten sie durch die Parallelisierung von CPU und GPU Beschleunigungen bis zum Faktor 16,7.

Konkrete Anwendungen von GPUs außerhalb der Computergrafik-Domäne beinhalten beispielsweise Ansätze für die *lineare Algebra*. In [51] wurde dazu ein Framework vorgestellt, das die Implementierung allgemeiner numerischer Techniken zum Lösen von Differentialgleichungen auf der GPU vorstellt. Die dafür nötigen Matrizen werden dabei in

Form von Texturen verwaltet, wobei die Elemente so eingetragen werden, dass die in numerischen Simulationen typischerweise auftretenden Bandmatrizen effizient verarbeitet werden können. Die arithmetischen Operationen werden dann in Fragment Shadern ausgeführt. Für Operationen, die aus allen Elementen eines Vektors einen einzigen Skalar berechnen (z. B. Maximumsuche), wird eine mit dem Aufbau von Mip-Maps vergleichbare Methode verwendet. Die Beschleunigung der Algorithmen im Vergleich zur Software-Lösung betrug in Messungen je nach getesteter Operation zwischen Faktor 3 und 15. Unter der Annahme iterativer Algorithmen mit vielen Durchläufen wurde dabei der Datentransfer zwischen Hauptspeicher und GPU nicht berücksichtigt.
In [50] wurden als Beispiel für eine numerische Simulation die Ergebnisse dieser Arbeit auf die Simulation von Strömungen von Fluiden, basierend auf einem Partikelsystem, angewandt. Da in dieser Anwendung das Ergebnis direkt für die Visualisierung genutzt wurde und es daher nicht zwischen Hauptspeicher und GPU übertragen werden musste, konnte sogar eine Beschleunigung um den Faktor 30 im Vergleich zur Berechnung auf der CPU erreicht werden.
Ein Beispiel für die Realisierung einer *physikalischen Simulation* auf der GPU ist die in [37] beschriebene Simulation der Dynamik von Wolken. Diese werden mit partiellen Differentialgleichungen modelliert, die die Strömungsbewegung, thermodynamische Prozesse und weitere physikalische Prozesse beschreiben. Die 3D-Daten der Wolken werden dabei aus Gründen der Effizienz für die Simulation in Schichten zerlegt und innerhalb einer 2D-Textur gespeichert ("flat 3D textures"). Die einzelnen Schichten werden dann über eine zusätzliche Textur angesprochen, die die Offsets enthält. Zusätzlich zur Dynamik wird die Interaktion der Wolken mit einfallendem Licht simuliert, wobei auch Lichtstreuung und Selbstschattierung berücksichtigt werden. Für die Darstellung wird dabei eine 3D-Textur verwendet. Die rechenintensive Simulation innerhalb der in C_g geschriebenen Fragment Shader wird auf mehrere Frames verteilt, um eine interaktive Visualisierung unter Einsatz von "Impostors" (Repräsentation komplexer Objekte als texturierte Rechtecke) zu ermöglichen. Dadurch konnte eine Darstellung mit bis zu 80 Frames/s bei 1-5 Aktualisierungen der Simulation pro Sekunde erreicht werden.

Naheliegend ist die Umsetzung von *Bildverarbeitungs-Techniken* auf GPUs, da die Bildverarbeitung auch häufig im Vergleich zur Computergrafik als inverses Problem angesehen wird [56]. Computergrafik wird dazu als *Bildsynthese* betrachtet, da sie aus einer mathematischen Beschreibung einer Szene ein 2D-Array aus Zahlen erstellt: ein Bild. Bildverarbeitung widerum lässt sich als *Bildanalyse* beschreiben, da sie ein 2D-Bild in eine mathematische Beschreibung umwandelt. So wurde beispielsweise gezeigt, dass der Prozess der Bildregistrierung mithilfe algebraischer projektiver Geometrie isomorph zum Prozess der perspektivischen Projektion einer Textur auf ein Polygon ist [32].
In [20] wurde der Einsatz von GPUs für Aufgaben der Bildverarbeitung untersucht. Dabei wurden hauptsächlich einfache Beispiele, wie Mittelwertfilterung und Farbraum-Konvertierung, untersucht. Die Beispiele wurden dazu nach verschiedenen Aspekten, wie der Menge an Vektor-Operationen und der Anzahl an Speicherzugriffen, klassifiziert. Hier wurden bis zu 10-mal schnellere Verarbeitungszeiten im Vergleich zur Ausführung auf der CPU gemessen.
Auch in [31, 32] werden mathematische Operationen der Bildverarbeitung auf Grafik-

Hardware abgebildet. Als Beispiel wird ein realzeit-fähiger Motion-Tracker mit OpenGL und C_g realisert. Die Anwendung ist *Mediated Reality*, eine Variante der AR für Wearable Computing: ein Benutzer sieht seine Umgebung durch eine Art HMD, das ihm eine veränderte Ansicht der Szene präsentiert, die er eigentlich sehen würde (z. B. Entfernung von Werbung). Für die Realisierung wurden unter anderem auf der GPU Tiefpassfilterung sowie räumliche und zeitliche Ableitung der Farbwerte der Fragmente durchgeführt. Gegenüber der CPU wurde hier ein Faktor von 3,5 gemessen. Inzwischen gibt es Bestrebungen Bibliotheken für die Ausführung von Bildverarbeitungs-Algorithmen auf GPUs anzubieten [76]. Neben einfachen Operatoren, wie Kantendetektoren, kann dort auch auf komplexere Algorithmen, wie Tracking, zugegriffen werden.

Als Beispiel für Anwendungen, die zwar im Computergrafik-Kontext zu sehen sind, jedoch nicht auf Rendering basieren, sollen hier noch zwei Projekte genannt werden, die komplette *Ray Tracer* (und damit ein globales Beleuchtungsmodell anstatt eines lokalen wie beim Rendering) [81, 82] oder zumindest Teile davon [15] auf der GPU umsetzen. Speziell die Abbildung der erforderlichen Berechnung der Schnittpunkte der Strahlen mit den Dreiecken auf mehrere Render-Durchläufe wird in beiden Arbeiten behandelt. In Konkurrenz zu diesen Lösungen steht der gerade vorgestellte tatsächlich für Ray Tracing entwickelte Prozessor (*Ray Processing Unit, RPU*) “SaarCOR” [104].

4 Gewinnung von Texturen aus Kamerabildern

Bei dem in dieser Arbeit beschriebenen prädiktiven Display bilden die Texturen neben dem Modell die Schnittstelle zwischen Realität und dargestellter, virtueller Realität. Die **Darstellung** der Szene ist ein einfacher Rendering-Algorithmus unter Verwendung der Texturen. Dabei werden die gewonnenen Texturen auf die entsprechenden Polygone gelegt und die Szene mit den passenden Abbildungseigenschaften der *virtuellen Kamera* gerendert (Abb. 4.1, rechts). Die intrinsischen Parameter C_d der virtuellen Kamera müssen bei Einsatz eines HMDs dessen Eigenschaften möglichst gut reproduzieren. Diese Kalibrierung ist erforderlich, um dem Benutzer eine Zuordnung zwischen seinen Aktionen und den dadurch für ihn generierten Bildern zu ermöglichen. Gesehene und gefühlte Distanzen müssen dazu möglichst gut korrelieren. Dieses Problem ist aus der Augmented Reality bekannt, wo die Anforderungen an die Kalibrierung eines HMDs noch wesentlich höher sind, da Realität und virtuelle Realität zu einer gemeinsamen Darstellung verschmelzen müssen. Die Position der virtuellen Kamera $\mathbf{P}_d$ ergibt sich typischerweise über einen Positions-Tracker aus der Kopfposition des Benutzers, andere Interfaces sind je nach Anwendung sinnvoll. Beim prädiktiven Display entspricht das Modell $\mathcal{M}_d$ der (zu erwartenden) Remote-Umgebung, *nachdem* die aktuellen Kommandos des Benutzers umgesetzt wurden. Dazu werden aus Aktionen des Benutzers resultierende Änderungen der Szene direkt in dieses Modell übernommen. Dies können explizite Änderungen, wie eine Bewegung der Roboterarms oder auch implizite Änderungen, wie das Mitführen eines gegriffenen Objekts, sein. $\mathcal{M}_d$ enthält somit die eigentliche Prädiktion.

Die **Gewinnung der Texturen** aus den Kamerabildern (Abb. 4.1, links) ist eine der Kernaufgaben des prädiktiven Displays. Die dazu entwickelten Methoden werden in diesem und dem nächsten Kapitel vorgestellt.

Kurze Ausführungszeiten sind dabei für eine schnelle Aktualisierung der Texturen auf Basis neuer Kamerabilder zwingend erforderlich. Die Texturen müssen ständig aktualisiert werden, um Änderungen in der Remote-Szene zu übernehmen. Mit steigender Dauer der Texturgewinnung nimmt die zeitliche Differenz Δt zwischen Aufnahme des Kamerabildes und der auf den resultierenden Texturen basierenden Darstellung zu. Die dadurch sinkende Aktualität der verwendeten Texturen vermindert die Qualität der Prädiktion. Die zumindest theoretisch bestenfalls erreichbare Untergrenze der Aktualität der Texturen ist die unvermeidliche durch die Übertragung des Kamerabilds verursachte Zeitdifferenz. Ein weiterer Grund für die Notwendigkeit kurzer Ausführungszeiten ist, dass die Textur-

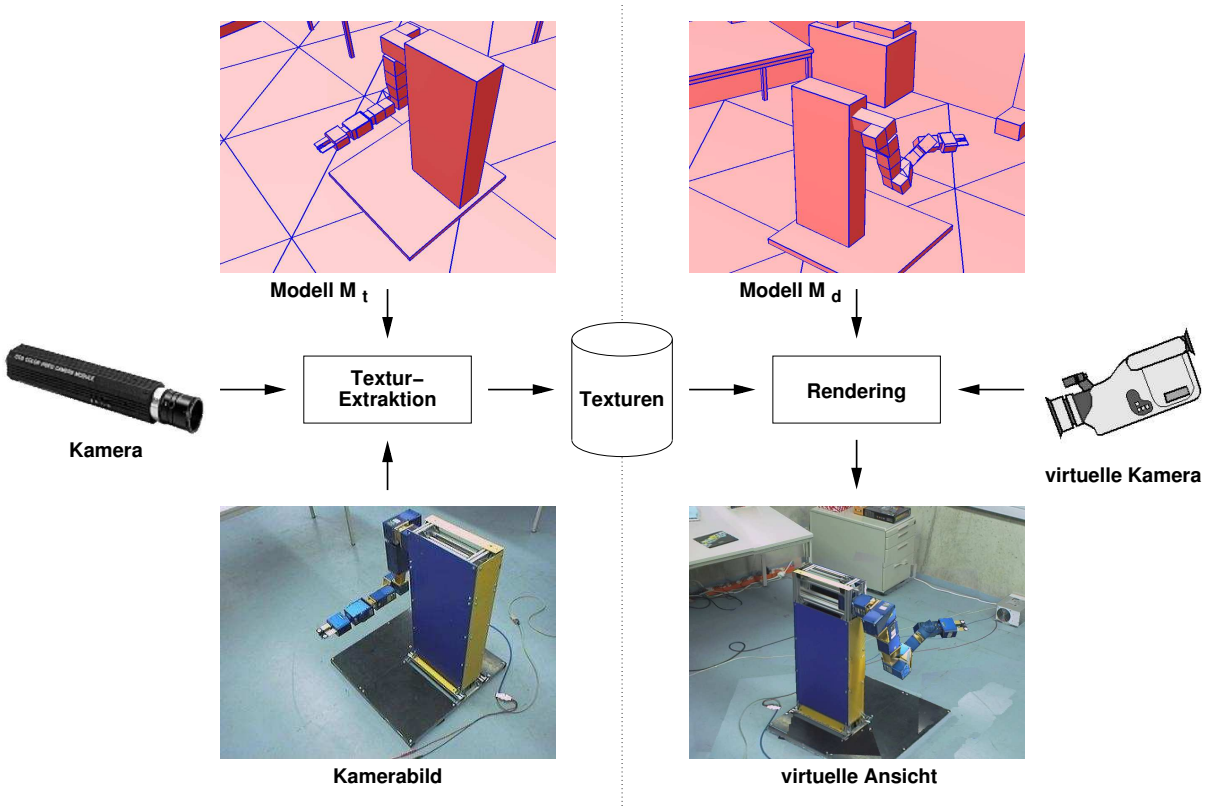


Abbildung 4.1: Textur-Extraktion und Darstellung

Aktualisierung Rechenzeit auf CPU und GPU (s. Kap. 6) belegt. Da diese auch für die Darstellung benötigt wird, muss sie gering gehalten werden.

Die Extraktion und Verarbeitung der Texturen beinhaltet hoch-parallelisierbare Berechnungen. Im Gegensatz zur CPU ist die GPU für solche Operationen optimiert. Selbst unter Ausnutzung von Architektur-Erweiterungen wie MMX, ISSE oder 3Dnow! kann mit der CPU kein vergleichbarer Durchsatz erzielt werden. Weiterhin führt die Verarbeitung der Texturen auf der CPU zu starkem Transfer von Bilddaten über den Grafikport bzw. -bus (PCI, AGP, bei neueren Systemen auch PCI Express). Dieser wird dann selbst bei aktuellen Anbindungen (PCI Express, 16x: theoretisch bis zu 4 GB/s) schnell zum Flaschenhals. Beim Transfer zwischen GPU und dem auf der Grafikkarte lokalen schnellen Speicher werden hingegen aktuell Bandbreiten von z. B. 38,4 GB/s (NVidia GeForce 6800 Ultra) erreicht. Aus diesen Gründen wurden alle Algorithmen so entwickelt, dass ein möglichst großer Teil der Berechnungen direkt auf der Grafik-Hardware durchgeführt wird.

Ein Nachteil der Ausführung der Algorithmen auf der GPU ist die Beschränkung der möglichen Operationen auf diejenigen, die von der Grafik-API zur Verfügung gestellt werden. Die folgenden Kapitel zeigen, wie die erforderlichen Operationen trotz der durch OpenGL vorgegebenen Einschränkungen realisiert werden können.

4.1 Extraktion der perspektivischen Texturen

In einem ersten Schritt werden die perspektivischen Texturen der einzelnen Polygone aus dem Kamerabild gewonnen. Diese Texturen enthalten die durch die Abbildungseigenschaften der Kamera entstehende perspektivische Verzerrung. Für ihre Extraktion müssen folgende Daten bekannt sein:

- das entzerrte (Kissenverzerrung, ...) Kamerabild I_t,
- das Szenenmodell $\mathcal{M}_t$ zum Zeitpunkt der Aufnahme,
- die intrinsischen Kameraparameter C_t: Bildgröße (d_x, d_y), Hauptpunkt (c_x, c_y), Pixelgröße (s_x, s_y) und Brennweite f für das entzerrte Bild sowie
- die extrinsischen Kameraparameter, d. h. Kameraposition und -orientierung $\mathbf{P}_t$ zum Zeitpunkt der Aufnahme.

Zunächst wird das entzerrte Kamerabild in einen der Color-Buffer auf der Grafikkarte kopiert. Hier wird vorerst von einer Extraktion der Texturen im sichtbaren Framebuffer ausgegangen. In den Kapiteln 6.2.3 und 6.4.2 wird auf diesen Kopiervorgang genauer eingegangen. Dies ist der einzige Moment in dem Bilddaten zwischen Hauptspeicher und Grafikspeicher übertragen werden.

Für den weiteren Verlauf wird die Projektion der realen Kamera für die entzerrten Bilder benötigt. Mit ihr werden die Abbildungseigenschaften der Kamera auf der Grafik-Hardware nachgebildet. Dies kann mit der Projektionsmatrix

$$\mathbf{M}_C = \begin{bmatrix} \frac{2}{x_1-x_0} & 0 & \frac{x_1+x_0}{x_1-x_0} & 0 \\ 0 & \frac{2}{y_0-y_1} & \frac{-y_1-y_0}{-y_1+y_0} & 0 \\ 0 & 0 & \frac{-(c_{far}+c_{near})}{c_{far}-c_{near}} & \frac{-2c_{far}c_{near}}{c_{far}-c_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.1}$$

mit den normierten Pixelkoordinaten der Bildeckpunkte

$$x_0 = \frac{-c_x s_x}{f} \tag{4.2}$$

$$x_1 = \frac{(d_x - c_x) s_x}{f} \tag{4.3}$$

$$y_0 = \frac{(d_y - c_y) s_y}{f} \tag{4.4}$$

$$y_1 = \frac{-c_y s_y}{f} \tag{4.5}$$

$$\tag{4.6}$$

erreicht werden. c_{near} und c_{far} geben dabei die Distanz von Near- bzw. Far-Clipping-Plane an und können entsprechend der Skalierung des Szenarios gewählt werden.

Sofern nicht explizit anders genannt, kommt im Folgenden bei allen Render-Durchläufen für die Abbildung als Projektionsmatrix $\mathbf{M}_C$ zum Einsatz.

4.1.1 Zuordnung von Bildbereichen zu Polygonen

Für jedes Polygon wird nun der mit ihm korrespondierende Bereich im Kamerabild ermittelt. Dazu muss die Abbildung der Eckpunkte des Polygons auf die Bildebene bestimmt werden (s. Abb. 4.2, links).

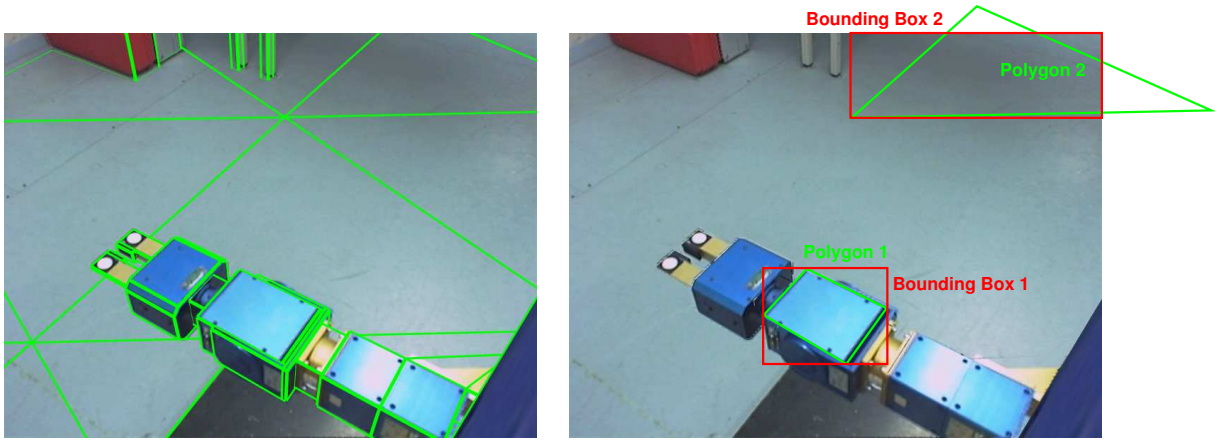


Abbildung 4.2: links: Kamerabild mit überlagertem Modell, rechts: Bounding Boxes zweier Polygone

Zunächst erfolgt dazu die Transformation der Weltkoordinaten der einzelnen N Eckpunkte ${}^{W}\mathbf{P}_i$ des Polygons über die homogene Matrix ${}^{C}\mathbf{M}_W$ in die Darstellung in Kamerakoordinaten $\mathbf{P}_i$:

$$\mathbf{P}_i = {}^{C}\mathbf{M}_W \, {}^{W}\mathbf{P}_i \tag{4.7}$$

Die Abbildung auf die Bildebene wird dann über

$$\mathbf{p}_i = \mathbf{M}_C \mathbf{P}_i \tag{4.8}$$

bestimmt.

Aus allen Eckpunkten des Polygons wird anschließend die Bounding Box $\mathcal{BB}_p$ um das Polygon ermittelt (Abb. 4.2, rechts, Polygon 1). Durch die projektive Abbildung können je nach Kameraposition einzelne Polygone sehr groß erscheinen. Für Polygone, die zum Teil außerhalb des Sichtvolumens liegen, muss daher zusätzlich noch Clipping erfolgen (Polygon 2), um zu vermeiden, dass die perspektivische Textur zu groß wird. Die Bounding

Box wird dazu auf den im Kamerabild sichtbaren Bereich beschnitten. Ihre endgültigen Eckpunkte berechnen sich damit aus

$$x_{BB_{min}} = max\{0, min\{x_i\}\} \tag{4.9}$$
$$x_{BB_{max}} = min\{d_x, max\{x_i\}\} \tag{4.10}$$
$$y_{BB_{min}} = max\{0, min\{y_i\}\} \tag{4.11}$$
$$y_{BB_{max}} = min\{d_y, max\{y_i\}\} \tag{4.12}$$

Der Bereich innerhalb dieser Bounding Box wird nun aus dem Framebuffer in ein Texturobjekt kopiert. Dieses enthält alle im Kamerabild enthaltenen Bildinformationen für das Polygon.

4.1.2 Bestimmung der Texturkoordinaten

Die Auflösung der perspektivischen Textur ergibt sich aus dem in $\mathcal{BB}_p$ enthaltenen Bereich und beträgt somit

$$t_x^p = x_{BB_{max}} - x_{BB_{min}} \tag{4.13}$$
$$t_y^p = y_{BB_{max}} - y_{BB_{min}} \tag{4.14}$$

$(x_{BB_{min}}, y_{BB_{min}})$ gibt den Offset der Bounding Box relativ zum Bildursprung an. Damit können die normalisierten homogenen Texturkoordinaten (s, t, r, q) eines Vertex (X, Y, Z) der perspektivischen Textur über

$$s = s_o + s_i \tag{4.15}$$
$$t = t_o + t_i \tag{4.16}$$
$$r = 0 \tag{4.17}$$
$$q = \frac{z}{f} \tag{4.18}$$

mit

$$s_o = \frac{c_x - x_{BB_{min}}}{t_x^p} \tag{4.19}$$
$$t_o = \frac{c_y - y_{BB_{min}}}{t_y^p} \tag{4.20}$$
$$s_i = \frac{x}{t_x^p s_x} \tag{4.21}$$
$$t_i = \frac{y}{t_y^p s_y} \tag{4.22}$$

berechnet werden. (s_o, t_o) ist der Offset der Texturkoordinaten durch die Lage von $\mathcal{BB}_p$. Mit (s_i, t_i) wird die korrekte Koordinate innerhalb der Bounding Box selektiert. r ist nur für 3D-Texturen relevant und hat bei 2D-Texturen immer den Wert 0. q enthält die aus der Kameraprojektion resultierende perspektivische Verzerrung und führt damit zu einer korrekten Interpolation der Texturkoordinaten innerhalb der Textur während des Renderns (s. Gleichungen 2.15 und 2.16). In [89] wird der mathematische Hintergrund von q und die Bedeutung für perspektivische Texturen genauer beschrieben. Eine gerenderte Szene ohne perspektivische Korrektur ist in Abbildung 4.3 zu sehen. Besonders an der Wand sind Verzerrungen innerhalb der Polygone durch die lineare Interpolation zu erkennen. Abbildung 4.3 zeigt die gleiche Szene bei Berücksichtigung der in der Textur enthaltenen Perspektive.

Abbildung 4.3: links: Verzerrungen durch in den Texturen enthaltene Perspektive, rechts: mit perspektivischer Korrektur

Die Texturkoordinaten sind auch für Punkte außerhalb des Sichtvolumens gültig. Dadurch können die Texturkoordinaten als Tupel gemeinsam mit den jeweiligen Eckpunkt-Koordinaten abgespeichert und später verwendet werden, auch wenn die Projektion des Eckpunktes außerhalb des Kamerabildes liegt. Die Berechnung der Texturkoordinaten von auf das Sichtvolumen geclippten Vertices ist daher nicht erforderlich.

4.2 Detektion von Verdeckungen

4.2.1 Motivation

Bei der zuvor beschriebenen Methode wurde immer der komplette Bereich innerhalb der Projektion eines Polygons als Textur abgespeichert und später zum Zeichnen verwendet. Es entstehen jedoch Artefakte, wenn Teile des Polygons eigentlich von anderen Teilen des Modells verdeckt werden. In Abbildung 4.4 ist beispielhaft ein Bild zu sehen, das durch Rendern unter Verwendung von Texturen entstand, die auf die im vorangegenen Kapitel beschriebene Weise extrahiert wurden. Teile des Kamerabildes, die eigentlich

zu einem *verdeckenden* Polygon gehören, werden auch in die Textur des *verdeckten* Polygons aufgenommen. Beim Zeichnen neuer synthetischer Ansichten erscheinen dann Teile des verdeckenden Polygons zusätzlich in der Textur des weiter entfernten Polygons. Im Beispielbild ist dadurch z. B. der Roboterarm zweimal zu sehen. Einmal ist er tatsächlich modelliert, zum Zweiten ist er in der Textur des Bodens enthalten.

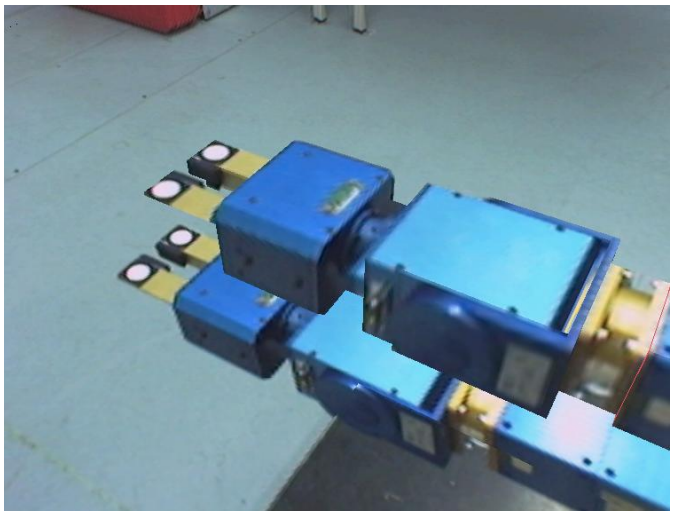


Abbildung 4.4: Synthetische Szene nach Textur-Gewinnung ohne Verdeckungsrechnung

Zur Vermeidung solcher Artefakte müssen Bereiche in den Texturen, die eigentlich zu anderen Polygonen gehören, als ungültig markiert werden. Dies wurde durch eine Kodierung im Alphakanal der Texturen realisiert. Gültige Texel werden dazu mit dem Wert α_v^p (valid) kodiert, ungültige Texel erhalten den Wert α_i^p (invalid). Die genauen Werte werden in Abschnitt 4.5 erläutert. Abbildung 4.5 zeigt rechts die Textur eines Polygons mit Ausmaskierung (rot) der ungültigen Texel. Die Detektion dieser Verdeckungen auf der Grafikkarte ist Inhalt des folgenden Abschnitts.

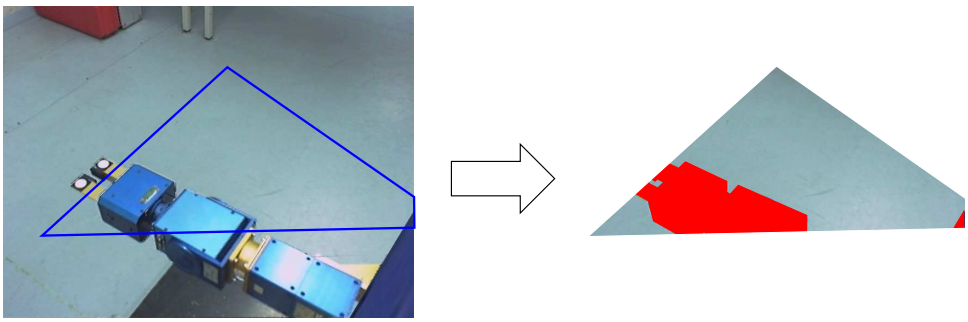


Abbildung 4.5: Ausmaskierung ungültiger Texel

4.2.2 Z-Buffer Verfahren

Bei der Lösung kommt der Z-Buffer der 3D-Hardware zum Einsatz. Die folgenden Schritte werden durchlaufen nachdem, wie bereits zuvor beschrieben, das Kamerabild in den Color-Buffer des Framebuffers geladen wurde. Der Alphawert jedes Pixels des Framebuffers wird mit α_v^p initialisiert, d. h. jedes Pixel wird zunächst als gültig markiert. Der Color-Buffer wird anschließend schreibgeschützt, um die Farbinformationen der Texturen bei den nachfolgenden Operationen zu erhalten. Die komplette Szene wird nun aus der Sicht von $\mathbf{P}_t$ aus gezeichnet. Dadurch wird für jedes Fragment die Entfernung zum nächstgelegenen Polygon in der Szene bestimmt und im Tiefenpuffer gespeichert. Für die nächsten Schritte wird der Tiefenpuffer dann ebenfalls schreibgeschützt.

Anschließend wird jedes Polygon separat in einem eigenen Durchlauf erneut gezeichnet. Dieses Mal wird während des Zeichnens die Tiefe jedes Fragments des Polygons mit der bereits im Framebuffer gespeicherten Tiefe des Pixels verglichen. Ist die gespeicherte Tiefe kleiner als die des Fragments, so bedeutet dies eine Verdeckung des Polygons an dieser Stelle. Daher wird der Alphawert dieses Pixels auf α_i^p gesetzt. Abbildung 4.6 zeigt dies für ein Polygon der in Bild 4.2 dargestellten Szene. Nachdem das Polygon verarbeitet wurde wird, wie zuvor beschrieben, der Inhalt der Bounding Box um dieses als perspektivische Textur gespeichert. Diesmal wird jedoch zusätzlich zu den Farbinformationen der Alphakanal des Framebuffers mit in den Alphakanal der Textur kopiert. Nach Rücksetzen aller Alphawerte im Framebuffer auf α_v^p wird das nächste Polygon verarbeitet. In Abbildung 4.7 ist der beschriebene Ablauf zur besseren Übersicht als Diagramm dargestellt.

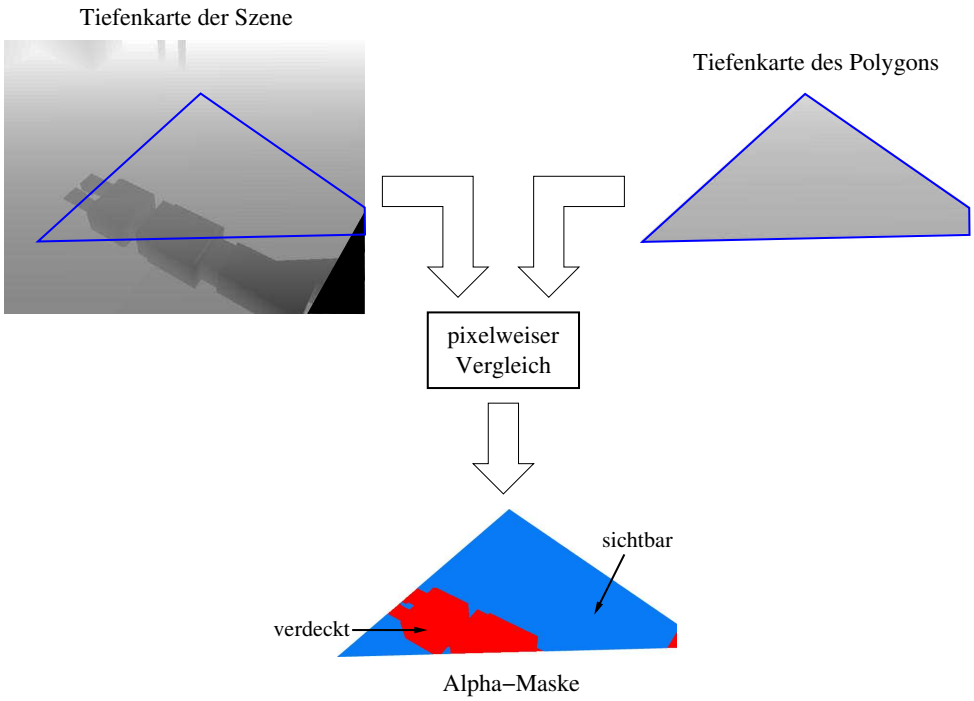


Abbildung 4.6: Generierung der Alphamaske durch Tiefenvergleich

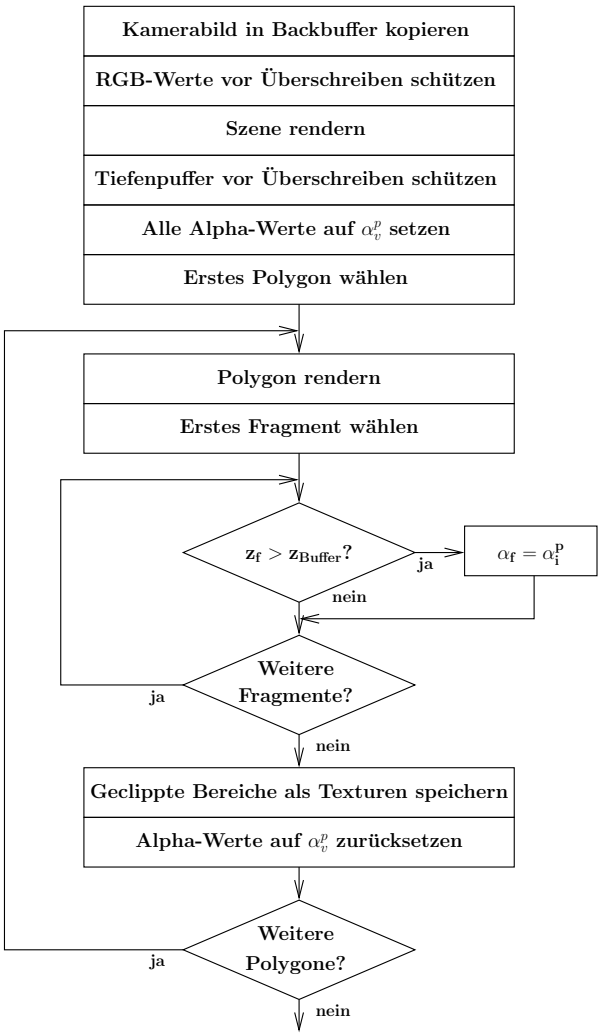


Abbildung 4.7: Detektion von Verdeckungen

4.2.3 Ergebnisse

In Abbildung 4.8 ist die gleiche gerenderte Szene wie in Bild 4.4 zu sehen, wobei diesmal jedoch bei der Texturgewinnung die Verdeckungsrechnung durchgeführt wurde. Durch die Verwendung eines Alpha-Tests beim Zeichnen ist zu sehen, welche Bereiche in den Texturen ausmaskiert wurden (rosa). Es ist zu erkennen, dass die fälschlicherweise in die Texturen des Bodens aufgenommenen Farbinformationen des Arms nahezu vollständig entfernt wurden. Die verbleibenden Artefakte werden in Kapitel 5.1 behandelt.

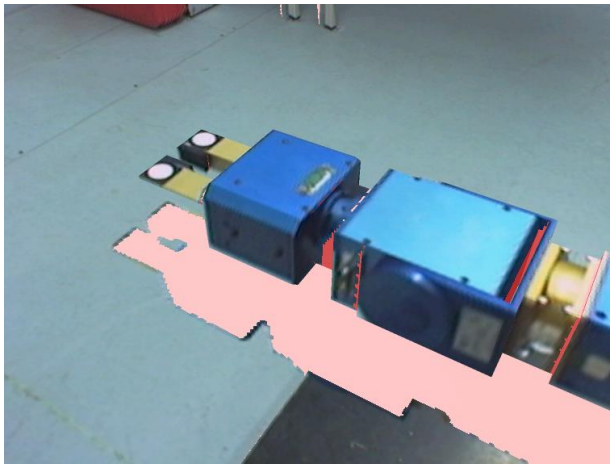


Abbildung 4.8: Gerenderte Szene nach Detektion und Ausmaskierung von Verdeckungen

4.3 Normalisierung und Skalierung

4.3.1 Motivation

Die Speicherung und Verwaltung der gewonnenen perspektivischen Texturen führt zu zwei Problemen:

1. Aufeinanderfolgende Kamerabilder werden normalerweise nicht von der gleichen Kameraposition aus aufgenommen. Es ist unklar welche Perspektive dann bei der Fusion zweier Texturen eines Polygons verwendet werden soll (s. Kap. 4.4).

2. Die Position der Kamera relativ zum Polygon bestimmt durch die perspektivische Abbildung auch die Größe der Textur (s. Abb. 4.9). Dies kann zu sehr großen Texturen selbst bei kleinen Polygonen führen, wenn sich die Kamera zum Zeitpunkt der Aufnahme nahe an diesen befindet. Diese hohen Auflösungen verursachen unnötig große Datenmengen. Dies führt zu zusätzlichem Zeitaufwand bei der weiteren Verarbeitung der Texturen, sowie zu einer zusätzlichen Dauer beim Zeichnen. Kritisch ist jedoch vor allem, dass dadurch eine Auslagerung der Texturen in den Hauptspeicher verursacht werden kann. Auf Grund der durch die verschiedenen Perspektiven bedingten wechselnden Auflösungen der Texturen kann dabei auch nicht a priori eine Abschätzung des benötigten Texturspeichers erfolgen.

Aus diesen Gründen müssen die Texturen in anderer Form verwaltet werden.

4.3.2 Transformation der perspektivischen Textur

Als Lösung werden die perspektivischen Texturen nur temporär gespeichert. Jede perspektivische Textur wird in eine *Orthogonal-Textur* transformiert. Dazu wird sie mit einer

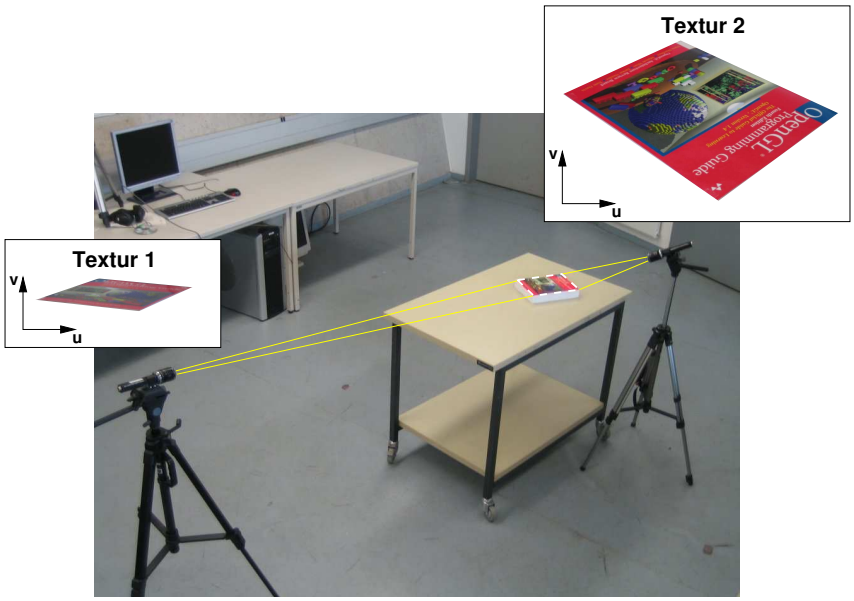


Abbildung 4.9: Abbildung der Textur eines Polygons aus unterschiedlichen Ansichten

Betrachterposition senkrecht auf das Polygon auf dieses projiziert. Bei der Projektion erfolgt die perspektivische Korrektur innerhalb der Textur wie in Kapitel 4.1 beschrieben. Abbildung 4.10 illustriert diese Projektion.

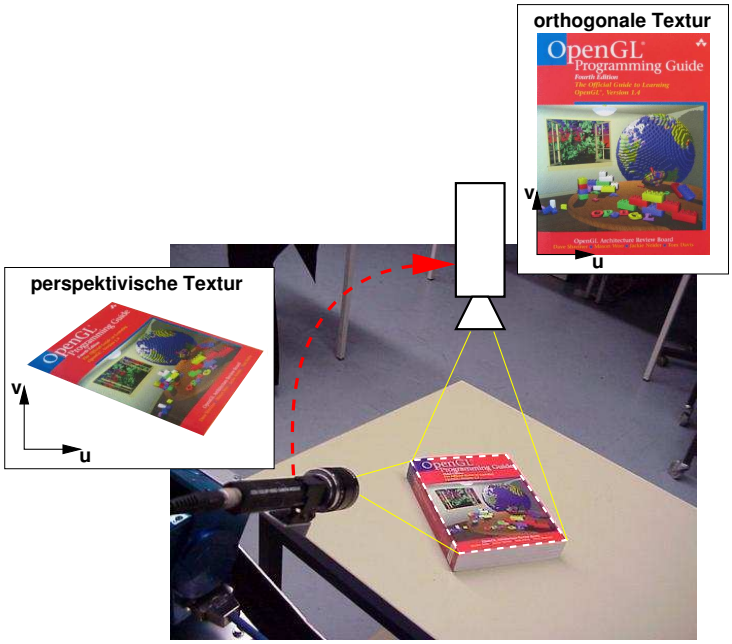


Abbildung 4.10: Normalisierung einer perspektivischen Textur

Für die Abbildung wird eine orthographische Projektion verwendet:

$$\mathbf{M}_t = \begin{bmatrix} \frac{2}{v_b} & 0 & 0 & 1 \\ 0 & \frac{2}{v_h} & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.23}$$

Hier geben v_b und v_h die Breite und Höhe des Viewports (Bereich in den gezeichnet wird) an. Bei dieser linearen Abbildung werden die Längenverhältnisse beibehalten. Sie ermöglicht die Angabe von Positionen nachfolgender Render-Operationen in Pixeln des Viewports, wodurch eine exakte Zuordnung von Texeln auf Pixel vereinfacht wird. Daher wird diese Abbildung für alle weiteren Verarbeitungsschritte anstelle von $\mathbf{M}_C$ verwendet.

Der Normalenvektor $\mathbf{n}_{pol}$ des Polygons gibt die Blickrichtung an. Ein beliebiger aber im Modell fester Eckpunkt wird als Ursprung des Polygon-Koordinatensystems gewählt. Die im Uhrzeigersinn angrenzende Kante wird als y-Achse festgelegt und bestimmt somit die Orientierung des Polygons. Die Basisvektoren des Polygon-Koordinatensystems sind damit

$$\mathbf{x}_{pol} = \mathbf{y}_{pol} \times \mathbf{n}_{pol} \tag{4.24}$$

$$\mathbf{y}_{pol} = {}^{W}\mathbf{P}_1 - {}^{W}\mathbf{P}_0 \tag{4.25}$$

$$\mathbf{z}_{pol} = \mathbf{n}_{pol} \tag{4.26}$$

Ist der Winkel zur anderen angrenzende Kante größer als 90 Grad, so werden Teile des Polygons außerhalb des Viewports abgebildet. Daher erfolgt für diesen Fall eine Verschiebung um den Offset

$$y_{offset} = \frac{2y_{min}}{v_h f_y}. \tag{4.27}$$

y_{min} ist dabei der kleinste y-Wert der abgebildeten Eckpunkte des Polygons. f_y bezeichnet den Skalierungsfaktor der Bounding Box des Polygons in y-Richtung auf die nächstgrößere Zweierpotenz. Ein Offset ist nur für den beschriebenen Fall erforderlich, da alle Polygone auf Grund der Vorgaben durch OpenGL konvex sein müssen.

Die resultierenden orthogonalen Ansichten der Polygone werden nun als Texturen gespeichert, die perspektivischen Texturen können verworfen werden. Da innerhalb der normalisierten Texturen keine perspektivische Verzerrung mehr vorhanden ist, ist eine weitere Betrachtung von q nicht mehr erforderlich ($q = 1$). Die Texturkoordinaten können daher auf (s, t) reduziert werden.

Durch die beschriebenen Methoden ist die Form der resultierenden rückprojizierten Textur eines Polygons invariant gegenüber der Position von der aus das Kamerabild aufgenommen wurde. Die durch die Abtastung bei der zusätzlichen Filterung entstehenden Artefakte sind minimal.

4.3.3 Skalierung der normalisierten Textur

Die Größe der normalisierten Textur ist noch frei wählbar und kann über eine zusätzliche Skalierung von $\mathbf{M}_t$ reguliert werden. Um die Auflösung in Relation zur jeweiligen Polygongröße zu setzen, wird sie mit

$$s_n = min\{r_{max}, \lceil d(x_{max} - x_{min}) \rceil_2\} \quad (4.28)$$
$$t_n = min\{r_{max}, \lceil d(y_{max} - y_{min}) \rceil_2\} \quad (4.29)$$

berechnet.

d gibt dabei das Verhältnis von Texturauflösung zu Polygongröße in Texel pro Meter an. Dieser Wert beschreibt den unvermeidbaren Kompromiss zwischen Qualität und Speicherbedarf und ist sinnvollerweise abhängig vom Szenario. Während in einem makroskopischen Szenario ein Wert von etwa 500-1.000 Texel/Meter typischerweise zu sehr guten Ergebnissen führt, sind für Szenarien mit kleinerer Skalierung, wie beispielsweise in der Mikromontage, wesentlich größere Verhältnisse erforderlich. Durch die Skalierung des Ergebnisses auf die nächste Zweierpotenz wird das Texturobjekt maximal ausgenutzt.

r_{max} bezeichnet eine vorgegebene Maximalauflösung der Texturen. Durch diese wird ein Speicherüberlauf vermieden, der ansonsten durch die aus großen Polygonen resultierenden hochauflösenden Texturen entstehen würde. Jede Textur die in s- oder t-Richtung größer als dieser Grenzwert ist wird in die entsprechende Richtung auf die Maximalgröße skaliert.

Abbildung 4.11 zeigt eine Szene bei unterschiedlichen Vorgaben für d und r_{max}. Zu kleine Werte (z. B. $d = 50$ Texel/Meter oder $r_{max} = 128$ Texel) führen hier zu einer deutlichen Abnahme der Bildqualität durch Unschärfe aufgrund der niedrigen Auflösung. Ein guter Kompromiss zwischen Bildqualität und benötigtem Texturspeicher wird in diesem Beispiel bei $d = 500$ Texel/Meter und $r_{max} = 256$ Texel erreicht. Größere Werte ergeben hier keine weitere sichtbare Verbesserung des Bildes.

4.4 Fusion

4.4.1 Motivation

In den vorherigen Kapiteln wurde lediglich die Gewinnung von Texturen aus einem einzelnen Kamerabild betrachtet. Es ist jedoch für das prädiktive Display erforderlich, ständig neue Texturen aus vom Teleoperator empfangenen Bildern zu extrahieren. Durch die Steuerung der Kamera durch den Benutzer sind dabei immer wieder neue Bereiche der Szene sichtbar. Dadurch können mit der Zeit mehr und mehr Polygone der Szene mit Farbinformation versehen werden. Es kommen dabei jedoch nicht immer nur neue Polygone in das Blickfeld der Kamera, sondern es sind auch fast immer Polygone sichtbar, für die bereits zuvor Texturen gewonnen wurden. Die neuen Farbinformationen müssen daher mit diesen älteren Texturen fusioniert werden.

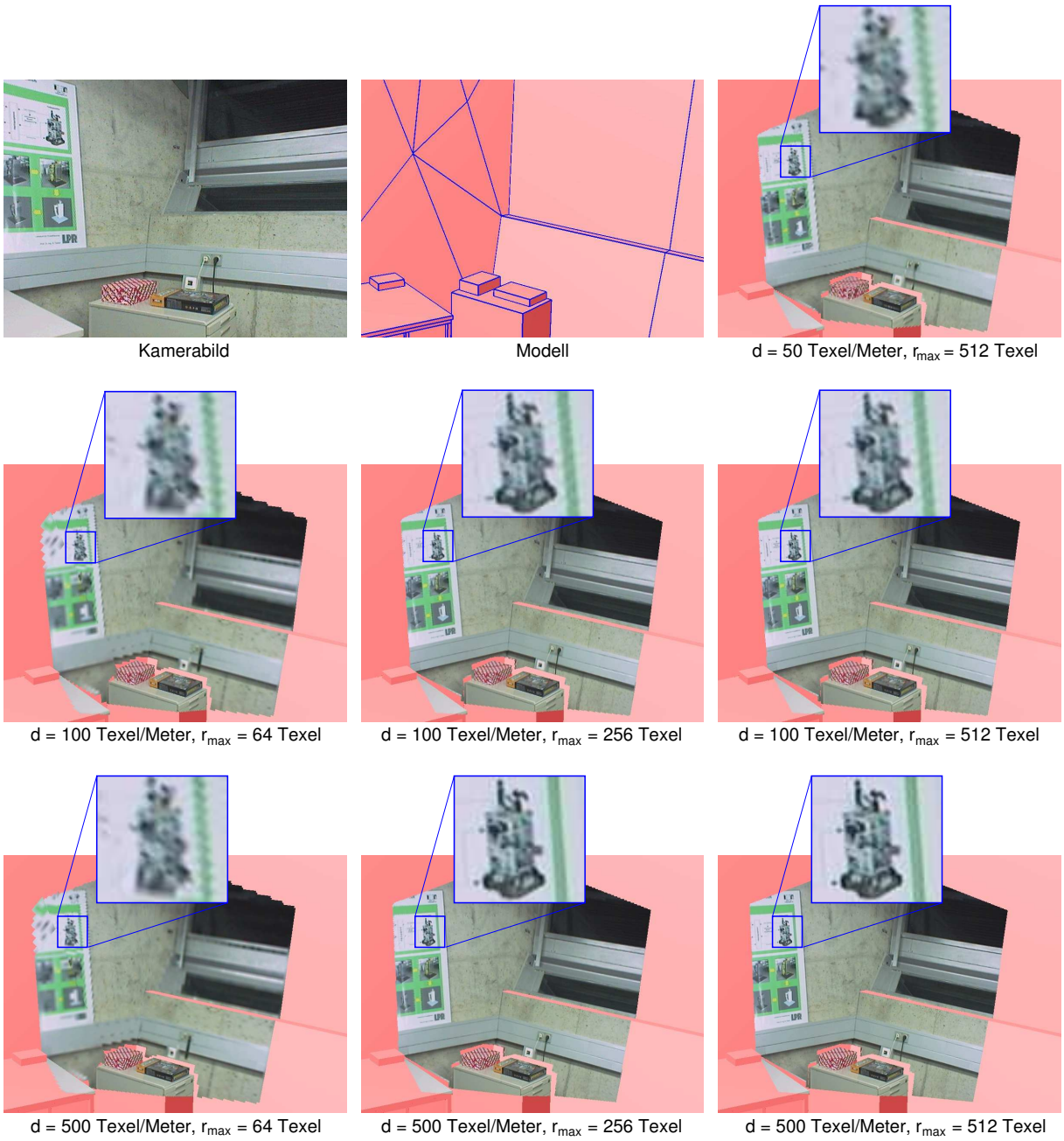


Abbildung 4.11: Szene mit unterschiedlichen Skalierungs-Parametern für die Texturen

4.4.2 Priorisierung aktueller Bildinformationen

Aus unterschiedlichen Ansichten eines Polygons sind oft verschiedene Gebiete von diesem sichtbar. Bereiche in den Texturen, für die bisher keine Farbinformation vorhanden ist, sollen soweit wie möglich durch die Fusion gefüllt werden. Daher wird bei Texeln, für die nur entweder der Farbwert der alten Textur T_a oder derjenige der neuen Textur T_n als gültig markiert ist, dieser in die fusionierte Textur T_f übernommen.

Für Texel, die sowohl in T_a als auch in T_n sichtbar sind, ist eine Mittelwertbildung denkbar. Für manche Anwendungen mit Gewinnung von Texturen ist dies auch sinnvoll. Da bei dem prädiktiven Display die Farbinformation jedoch immer möglichst aktuell sein soll, wird hier der Farbwert von T_n übernommen. Damit werden Beleuchtungsänderungen der Szene schnell in die Texturen aufgenommen. Weiterhin enthalten fast alle Oberflächen nicht nur ambiente und diffuse Anteile, sondern ihr Erscheinungsbild ist auch von der Position des Betrachters abhängig. Glanzlichter sind ein bekanntes Beispiel für solche Effekte. Da die Betrachterposition auch die Kamera steuert, wird durch Verwendung der aktuellsten Farbinformationen sichergestellt, dass die Differenz zwischen Betrachterposition $\mathbf{P}_d$ in der virtuellen Szene und Kameraposition $\mathbf{P}_t$ zum Zeitpunkt der Bildaufnahme für die Textur gering ist. Ein weiterer Vorteil der Verwendung der aktuellsten Farbinformationen ist die Verschleierung von Fehlern in der Modellbildung. Da das Modell immer nur eine Approximation sein kann, sind die als eben modellierten Polygone in Wirklichkeit nicht vollständig planar. Die Gewinnung einer Textur aus einem Bild, das von einer Kameraposition aus aufgenommen wurde, kann dann zu Verzerrungen bei der Betrachtung dieser Textur aus anderen Blickrichtungen führen. In Abbildung 4.12 ist dies anhand eines sehr schlecht modellierten Szenarios zu sehen. Die Gegenstände auf dem Tisch sind nicht modelliert, sondern in der Textur der Tischplatte enthalten. Im rechten Bild sind starke Verzerrungen in der Textur durch die große Abweichung zwischen $\mathbf{P}_t$ und $\mathbf{P}_d$ zu erkennen. Durch Verwendung der aktuellsten Farbinformationen in den Texturen wird auch die Minimierung dieser Verzerrungen sichergestellt.

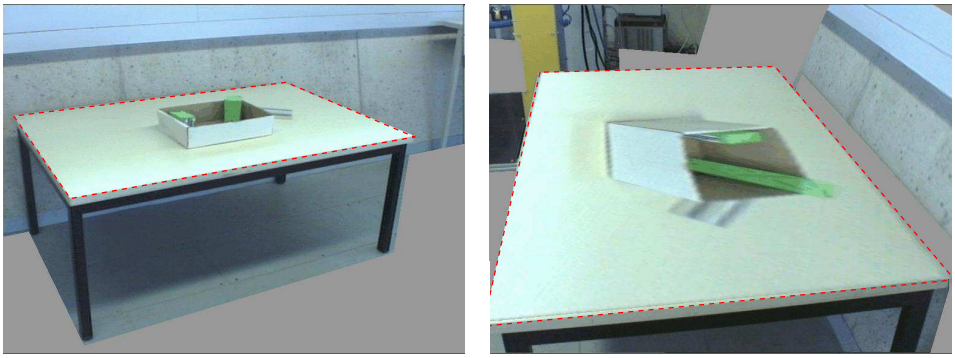


Abbildung 4.12: Verzerrung nicht modellierter Bereiche in einer Textur bei Änderung des Blickwinkels

Texel die in keiner der beiden Texturen als gültig markiert waren erhalten auch in T_f den Alphawert α_i^n, weil dieser Teil des Polygons in noch keinem Kamerabild sichtbar war.

Abbildung 4.13 verdeutlicht den Vorgang der Fusion, die die beschriebenen Eigenschaften aufweist.

Wird zu einem späteren Zeitpunkt erneut ein Kamerabild empfangen, so wird die vorher fusionierte Textur als alte Textur für die neue Fusion mit T_n^{i+1} verwendet: $T_a^{i+1} = T_f^i$ (gestrichelter Pfeil in Abbildung 4.13). Daher enthalten die Texturen mit der Zeit immer mehr gültige Texel.

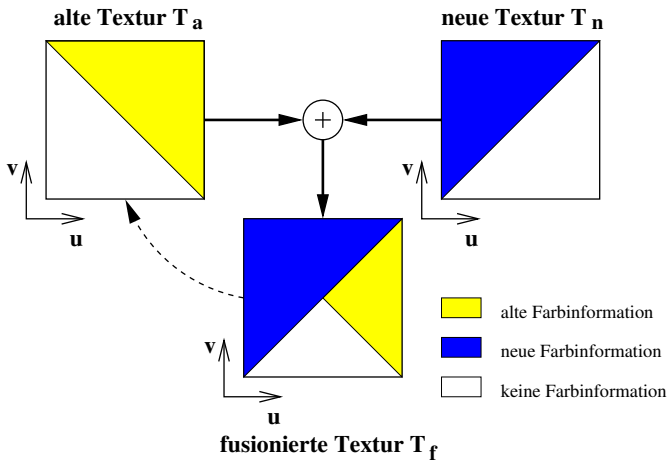


Abbildung 4.13: Fusion neuer und alter Texturinformationen

Um das Problem der Blickwinkelabhängigkeit der Oberflächen zu umgehen, wurde auch *View-Dependent Texture Mapping*, basierend auf [22, 23], implementiert und getestet. Dabei werden für jedes Polygon mehrere Texturen gespeichert, die aus unterschiedlichen Blickrichtungen gewonnen wurden. Bei der Darstellung werden dann die Texturen, die aus dem ähnlichsten Blickwinkel zum aktuellen Betrachterwinkel gewonnen wurden, überlagert. Auf Grund des hohen Speicherbedarf für die Verwaltung vieler Texturen pro Polygon, die für brauchbare Ergebnisse erforderlich sind, wurde dieser Ansatz dann aber nicht weiter verfolgt.

Realisierung auf der Grafikkarte

Für die Realisierung auf der Grafikkarte wird zunächst das Polygon mit der bereits gespeicherten Textur T_a aus orthogonaler Ansicht gezeichnet. Während des Zeichnens erfolgt ein Alpha-Test, sodass nur gültige Pixel gezeichnet werden. Anschließend wird das Polygon mit der, wie bereits beschrieben, aus dem neuen Kamerabild gewonnenen und normalisierten Textur T_n in einem zweiten Rendering Pass erneut in der gleichen Ansicht gezeichnet, wobei wiederum ein Alpha-Test durchgeführt wird. Die resultierende Textur T_f enthält die fusionierten Texel in der zuvor beschriebenen gewünschten Form. Sie wird nun statt T_a gespeichert, T_n kann verworfen werden.

Abbildung 4.14 zeigt die Fusion am Beispiel dreier aus aufeinanderfolgenden Kamerabildern extrahierter Texturen eines Polygons.

4.5 Darstellungsfehler an den Polygongrenzen

Die in Abbildung 4.15 gezeigten Szenen zeigen Darstellungsfehler bei Verwendung der mit den zuvor beschriebenen Methoden gewonnenen Texturen. Die hier betrachteten Fehler

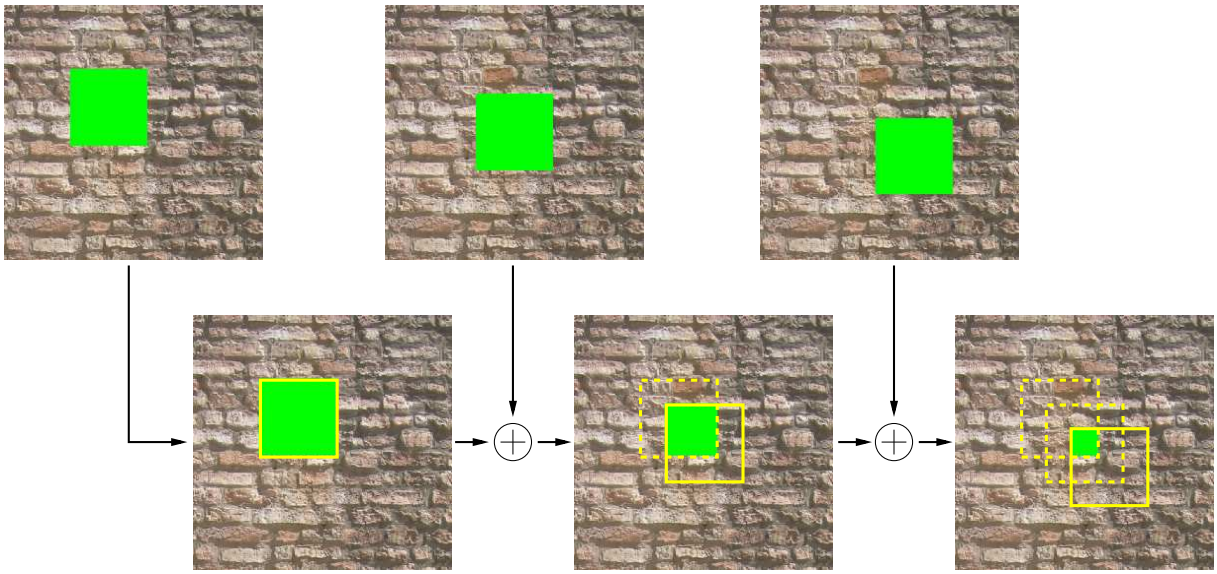


Abbildung 4.14: Fusion der Texturen aus drei aufeinanderfolgenden Kamerabildern

treten entlang der Polygonkanten auf und verringern die Realitätsnähe des dargestellten Bildes erheblich. Im linken Bild sind sie an den Polygongrenzen texturierter Bereiche zu sehen. Im rechten Bild entstehen die Fehler ebenfalls an Polygongrenzen, hier jedoch in Bereichen wo eigentlich überhaupt keine Farbinformation vorhanden ist. Das in Kapitel 5.2 beschriebene Verfahren zum Füllen von Löchern in den Texturen verstärkt diesen störenden Effekt sogar noch. Im Folgenden sollen die Ursachen für diese Darstellungsfehler diskutiert und Lösungsansätze für ihre Vermeidung vorgestellt werden.

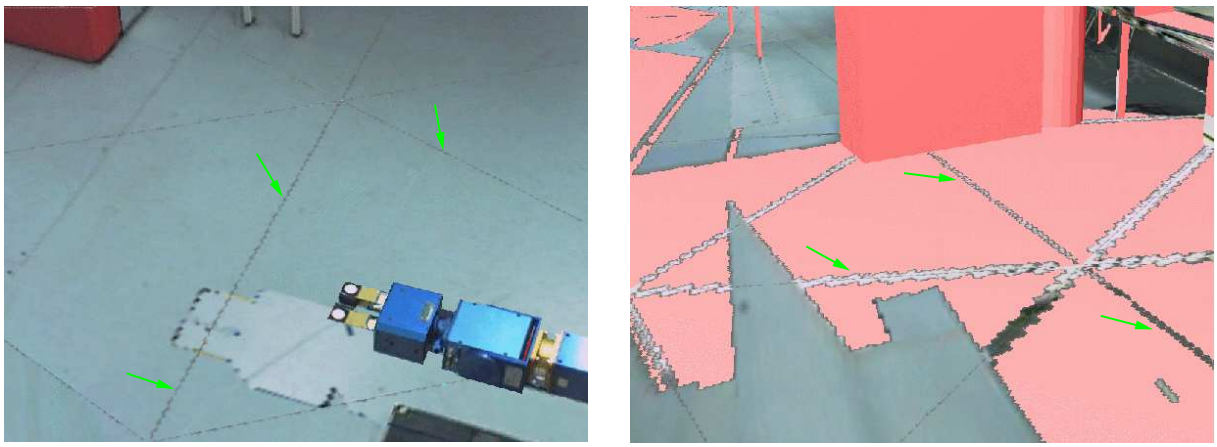


Abbildung 4.15: Darstellungsfehler an Polygongrenzen: links: in sichtbarem Bereich, rechts: in verdecktem Bereich

4.5.1 Kodierung mit Alphawerten

Für die weiteren Betrachtungen ist es sinnvoll zunächst die in den Texturobjekten enthaltenen Alphawerte zu klassifizieren und genauer zu analysieren.

Perspektivische Textur

Ein Texturobjekt, in dem eine perspektivische Textur gespeichert ist, kann in vier Bereiche unterteilt werden (Abb. 4.16, links):

Sichtbare Bereiche in der Textur (blau): Alle sichtbaren Texel sind mit Alphawerten α_v^p markiert. Sie werden beim Ablauf der Verdeckungsrechnung auf den Wert 1 gesetzt.

Verdeckte Bereiche in der Textur (rot): Alle verdeckten Texel sind mit Alphawerten α_i^p markiert. Sie werden beim Ablauf der Verdeckungsrechnung auf den Wert 0 gesetzt.

Bounding Box (grün): Alle Texel der Bounding Box, die außerhalb der Textur liegen, sind mit Alphawerten α_{BB}^p markiert. Der genaue Wert bei der Initialisierung wird in Kapitel 5.1.2 hergeleitet. Er wird zunächst als 1 angenommen, damit, wie später erläutert, bei der Normalisierung eine Interpolation der Farbwerte über die Polygongrenzen hinaus erfolgen kann.

Texturobjekt (gelb): Alle Texel des Texturobjekts außerhalb der Bounding Box sind mit Alphawerten α_O^p markiert. Sie werden beim Initialisieren des Texturobjekts auf den Wert $\alpha_O^p = \alpha_{BB}^p$ gesetzt.

Mit α_T^p werden im Folgenden alle Texel der perspektivischen Textur bezeichnet, d. h. sie können den Wert α_v^p oder α_i^p enthalten.

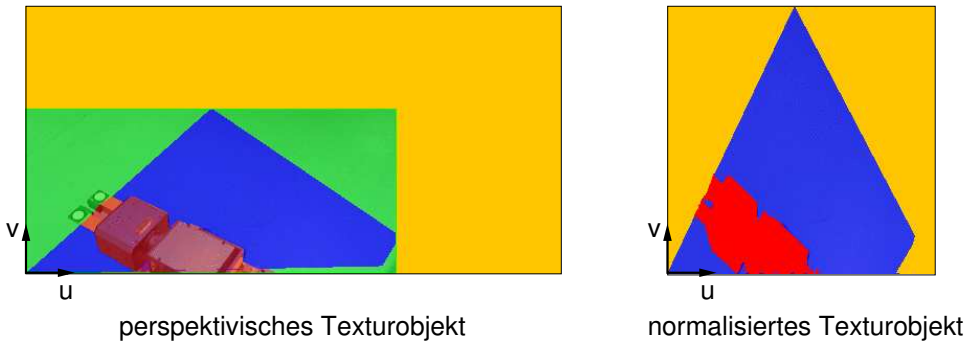


Abbildung 4.16: Bereiche in Texturobjekten: blau: sichtbarer Bereich, rot: verdeckter Bereich, grün: Bounding Box, gelb: restliches Texturobjekt

Normalisierte Textur

Ein Texturobjekt, in dem eine normalisierte Textur gespeichert ist, teilt sich in drei Bereiche auf. Ein Unterscheidung zwischen Bounding Box und Texturobjekt ist hier nicht

erforderlich, da die Bounding Box bei der Normalisierung so skaliert wird, dass sie das Texturobjekt einer normalisierten Textur vollständig ausfüllt. Die drei Bereiche sind somit (Abb. 4.16, rechts):

Sichtbare Bereiche in der Textur (blau): Alle sichtbaren Texel sind mit Alphawerten α_v^n markiert. Sie liegen im Wertebereich $]0;1]$.

Verdeckte Bereiche in der Textur (rot): Alle verdeckten Texel sind mit Alphawerten $\alpha_i^n = 0$ markiert.

Texturobjekt (gelb): Alle anderen Texel des Texturobjekts außerhalb der Textur werden zunächst mit Alphawerten α_O^n markiert, da sie keine gültigen Farbwerte enthalten.

Mit α_T^n werden im Folgenden alle Texel der normalisierten Textur bezeichnet, d. h. sie können α_v^n oder α_i^n enthalten.

4.5.2 Ursachen der Darstellungsfehler

Die in Abbildung 4.15 gezeigten Darstellungsfehler an den Grenzen der Texturen entstehen durch die in Kapitel 2.5.4 beschriebene bilineare Interpolation während des Texture Mappings, die sowohl für die Texturierung der dreidimensionalen Szene als auch für den Prozess der Normalisierung erfolgt (Abb. 4.17).

Normalisierung der Texturen

Durch die für die Normalisierung erforderliche Projektion wird die ursprüngliche Form der Texturen verändert. Die Farbvektoren $\underline{C}^n$ und Alphawerte α^n der Fragmente für die normalisierte Textur ergeben sich aus den Farbvektoren $\underline{C}^p$ und Alphawerten α^p der perspektivischen Textur.

Im Folgenden werden die Farbvektoren $\underline{C}_T^n$ und Alphawerte α_T^n der Texel betrachtet, die in der normalisierten Textur an der *Grenze zwischen Texturobjekt und Textur* zum liegen kommen. Die obere Hälfte von Abbildung 4.17 illustriert die Beschreibung.

Farbvektoren $\underline{C}_T^n$: Die resultierenden Farbvektoren $\underline{C}_T^n$ entlang der Grenze der normalisierten Textur werden mit der bilinearen Interpolation zwischen Farbvektoren der perspektivischen Textur $\underline{C}_T^p$ und der Bounding Box $\underline{C}_{BB}^p$ berechnet. Sind die Texel in der perspektivischen Textur mit α_v^p als sichtbar markiert, dann entstehen durch die Interpolation keine sichtbaren Fehler, da die betroffenen Farbvektoren $\underline{C}_{BB}^p$ der Bounding Box gültige Nachbartexel des Kamerabilds sind. Es findet damit eine bilineare Interpolation über die Texturgrenzen hinweg statt. Sind die Texel jedoch entlang der Grenze mit α_i^p als nicht sichtbar markiert, so werden Farbvektoren $\underline{C}_T^n$ für die normalisierte Textur berechnet, die keine gültigen Werte enthalten, da weder die Farbvektoren $\underline{C}_T^p$ noch die Farbvektoren $\underline{C}_{BB}^p$ gültige Werte für das Texel der Textur enthalten. Die so ermittelten Farbvektoren $\underline{C}_T^n$ sind die Basis für die Farbvektoren aus denen die Darstellungsfehler in den verdeckten Bereichen bestehen.

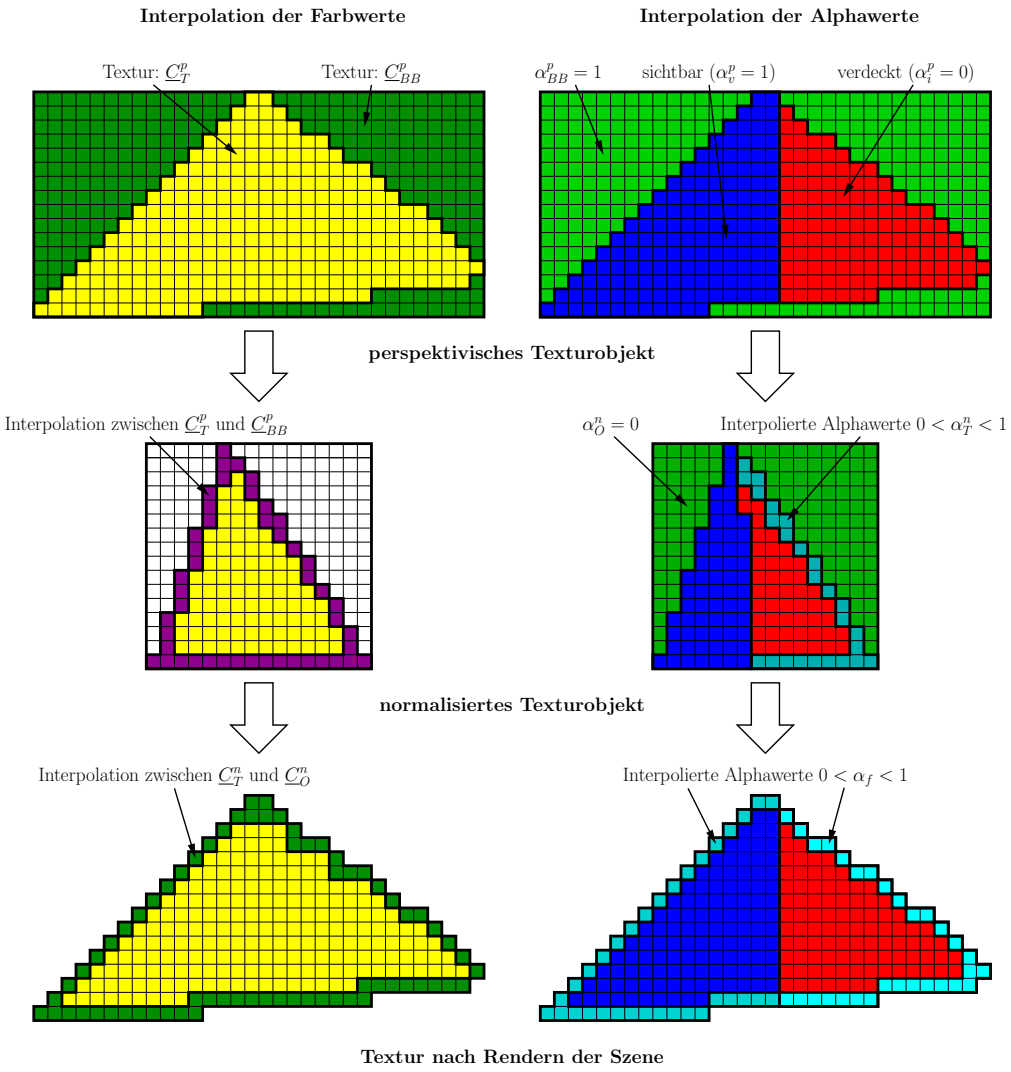


Abbildung 4.17: Interpolation der Farbwerte und Alphawerte bei der Normalisierung und beim Rendern einer Textur

Alphawerte α_T^n: Die bilineare Interpolation zwischen den Alphawerten α_{BB}^p der Bounding Box und α_T^p der perspektivischen Textur ist nur dann problematisch, wenn ein mit α_i^p als verdeckt markierter Bereich entlang der Grenze zur Bounding Box verläuft. Die dabei berechneten Alphawerte α_T^n liegen, in Abhängigkeit von der Gewichtung, irgendwo im Intervall $]0, 1[$. Dass diese Texel daher nicht eindeutig zugeordnet werden können, ist einer der Gründe für die Entstehung der Darstellungsfehler in den verdeckten Bereichen.

Texture Mapping beim Rendern der Szene

Analog werden nun die interpolierten Farbvektoren $\underline{C}^f$ und Alphawerte α^f eines Fragments entlang der Grenze des für die Darstellung gerenderten Polygons betrachtet (Abb. 4.17, unten).

Farbvektoren $\underline{C}^f$: Die Farbvektoren $\underline{C}_O^n$ außerhalb der normalisierten Textur, die mit den Farbvektoren $\underline{C}_T^n$ der normalisierten Textur interpoliert werden, enthalten keine sinnvollen Werte. Sie sind auf die Werte gesetzt, die vor der Normalisierung im Texturobjekt gespeichert waren. Durch die Interpolation entstehen die Farbvektoren der in Abbildung 4.15 links dargestellten Fehler entlang der Polygongrenzen.

Alphawerte α^f: Die bilineare Interpolation zwischen den Alphawerten α_v^n von gültigen Bereichen der normalisierten Textur und Alphawerten des normalisierten Texturobjekts außerhalb der Textur α_O^n ergibt Alphawerte α^f für die Fragmente im Intervall $]0, 1[$. Bei der Normalisierung entstehen jedoch wie gezeigt an den Grenzen der ungültigen Bereiche ebenfalls Werte in diesem Intervall durch die Interpolation zwischen α_i^p und α_{BB}^p. Es lässt sich daher anhand der resultierenden Alphawerte nicht mehr feststellen, ob die beteiligten α_T^p der perspektivischen Textur die Werte α_v^p oder α_i^p enthielten, d. h. ob sie gültig waren. Der Schwellwert zur Unterscheidung zwischen gültigen und ungültigen Texeln bei der Darstellung kann somit nur einen Kompromiss zwischen starker Ausprägung der Darstellungsfehler in den gültigen Bereichen oder in den ungültigen Bereichen darstellen. Eine vollständige Unterdrückung beider Arten von Fehler ist ohne zusätzliche Maßnahmen nicht möglich.

4.5.3 Vermeidung der Darstellungsfehler

Die Ursache der Darstellungsfehler ist somit die Verwendung von Interpolationsfiltern. Zwar könnten mit einer Nearest-Neighbor Filterung diese Fehler vermieden werden, die dadurch entstehenden Blockartefakte sind jedoch ebenfalls nicht akzeptabel. Die Markierung mit Alphawerten α_O^p in den Texturobjekten der perspektivischen Textur bei der Textur-Generierung und die Normalisierung muss daher so erfolgen, dass folgende Eigenschaften erfüllt sind:

- Für ein texturiertes Fragment an einer Polygongrenze soll ein Farbvektor $\underline{C}^f$ berechnet werden, der demjenigen entspricht, den das Fragment hätte, wenn es nicht an einer Polygongrenze läge. Die Interpolation kann damit über die Texturgrenzen hinweg erfolgen.
- Texel, die in verdeckten Bereichen der perspektivischen Textur liegen, müssen sich von allen anderen Texeln unterscheiden lassen. Beim Texture Mapping für die Darstellung der Szene müssen die entstehenden Alphawerte der nicht texturierten Fragmente $\alpha^f = 0$ sein.
- Texel, die ursprünglich in sichtbaren Bereichen der perspektivischen Textur liegen, müssen so markiert werden, dass sie sich von solchen unterscheiden lassen, die als ungültig markiert waren, d.h die Alphawerte der texturierten Fragmente bei der Darstellung müssen $\alpha_f > 0$ sein.
- Die Texel des Texturobjekts einer perspektivischen Textur müssen so mit Alphawerten α_O^p markiert werden, dass zwischen verdeckten Texeln, sichtbaren Texeln und

Texeln, die außerhalb der Textur liegen, unterschieden werden kann. Diese Bedingung ist ein Vorgriff auf Kapitel 5.1 und ist notwendig, um die in diesem Kapitel beschriebenen Artefakte unterdrücken zu können.

Transformation der Bounding Box

Die Bounding Box enthält neben der eigentlichen Textur des Polygons auch Texel, die zu (im Kamerabild) benachbarten Polygonen gehören. Mit den Farbvektoren $\underline{C}^p_{BB}$ dieser Texel können die Farbvektoren $\underline{C}^f$ für texturierte Fragmente wie gefordert über die Texturgrenzen hinweg interpoliert werden. Dazu müssen die Farbvektoren $\underline{C}^f_{BB}$ der Bounding Box jedoch auch in dem normalisierten Texturobjekt zur Verfügung stehen. Aus diesem Grund wird für die Normalisierung nicht nur das Polygon gezeichnet, sondern die komplette Bounding Box $\mathcal{BB}_p$ des Polygons.

Allerdings sind von der Bounding Box zunächst nur die *2D-Bildschirmkoordinaten* (x_i, y_i) ihrer Eckpunkte bekannt. Zur Normalisierung werden aber die *3D-Kamerakoordinaten* (X_i, Y_i, Z_i) dieser Eckpunkte benötigt. Normalerweise lassen sich Bildschirmkoordinaten (x, y) mithilfe des Z-Buffer-Wertes z und als Funktion der Rückprojektion in Kamerakoordinaten umrechnen. Da jedoch die Bounding Box in der Szene nicht tatsächlich existiert, sind für ihre Eckpunkte keine Tiefenwerte bekannt. Die 3D-Lage dieser Eckpunkte muss daher explizit berechnet werden.

Gesucht werden somit die Eckpunkte des Vierecks in Kamerakoordinaten, das durch die Projektion in die Bounding Box übergeht. Dieses Viereck liegt in der selben Ebene wie das Polygon und wird mit der gleichen Projektion abgebildet. Die Gleichung der Polygonebene

$$a_{BB}X + b_{BB}Y + c_{BB}Z + d_{BB} = 0 \tag{4.30}$$

lässt sich mit drei Eckpunkten (X_1, Y_1, Z_1),(X_2, Y_2, Z_2) und (X_3, Y_3, Z_3) des Polygons aufstellen:

$$\begin{aligned}
a_{BB} &= Y_1(Z_2 - Z_3) + Y_2(Z_3 - Z_1) + Y_3(Z_1 - Z_2) && (4.31)\\
b_{BB} &= Z_1(X_2 - X_3) + Z_2(X_3 - X_1) + Z_3(X_1 - X_2) && (4.32)\\
c_{BB} &= X_1(Y_2 - Y_3) + X_2(Y_3 - Y_1) + X_3(Y_1 - Y_2) && (4.33)\\
d_{BB} &= -X_1(Y_2Z_3 - Y_3Z_2) - X_2(Y_3Z_1 - Y_1Z_3) && (4.34)\\
&\quad -X_3(Y_1Z_2 - Y_2Z_1).
\end{aligned}$$

Der Projektionsstrahl ist bestimmt durch

$$\begin{aligned}
x_n &= (x - c_x)\frac{s_x}{f} && (4.35)\\
y_n &= (y - c_y)\frac{s_y}{f} && (4.36)
\end{aligned}$$

mit

- x_n, y_n : normierte Koordinaten

- x, y : Bildschirmkoordinaten

Der Schnittpunkt mit der Polygonebene ergibt sich dann aus

$$Z = -\frac{d_{BB}}{a_{BB}x_n + b_{BB}y_n + c_{BB}} . \tag{4.37}$$

Damit lassen sich schließlich auch die beiden anderen Komponenten

$$X = x_n Z \tag{4.38}$$
$$Y = y_n Z \tag{4.39}$$

der Kamerakoordinaten berechnen.

Setzt man nun in Gleichung 4.35 und 4.36 für (x, y) die bekannten Bildschirmkoordinaten $\{(x_{BB_1}, y_{BB_1}); \ldots; (x_{BB_4}, y_{BB_4})\}$ der vier Eckpunkte der Bounding Box ein, so ergeben sich die für die Normalisierung der Bounding Box benötigten Kamerakoordinaten.

Bereiche außerhalb der Textur

Die Alphawerte α_T^p, α_{BB}^p und α_O^p sollen so gesetzt werden, dass die oben beschriebenen Eigenschaften erfüllt werden. Die Werte für α_T^p werden dazu wie in Kapitel 4.5.1 beschrieben gesetzt. Für die in Kapitel 5.1 beschriebene Unterdrückung der z. B. durch Modellierungsfehler entstehenden Artefakte hat sich ein Alphawert $\alpha_{BB}^p = \frac{31}{32}$ als sinnvoll erwiesen, wie in Kapitel 5.1.2 zu sehen sein wird. Eine neues Texturobjekt für eine perspektivische Textur wird deshalb mit einem Alphawert für alle Texel $\alpha_{BB}^p = \frac{31}{32}$ angelegt (Abb. 4.18, grün gefärbter Bereich). Eine Unterscheidung zwischen α_{BB}^p und α_O^p ist nicht erforderlich, d. h. $\alpha_O^p = \alpha_{BB}^p$ (Abb. 4.18, grün gefärbter Bereich).

Abbildung 4.18: Bereiche im perspektivischen Texturobjekt bei erweiterter Verdeckungsrechnung (weiß) und einheitliche Behandlung der Bereiche außerhalb der Textur

Verdeckungsrechnung für die Bounding Box

Durch die Normalisierung der Bounding Box soll eine Interpolation der Farbvektoren über die Texturgrenzen hinweg erreicht werden. Für diese Interpolation werden, wie in

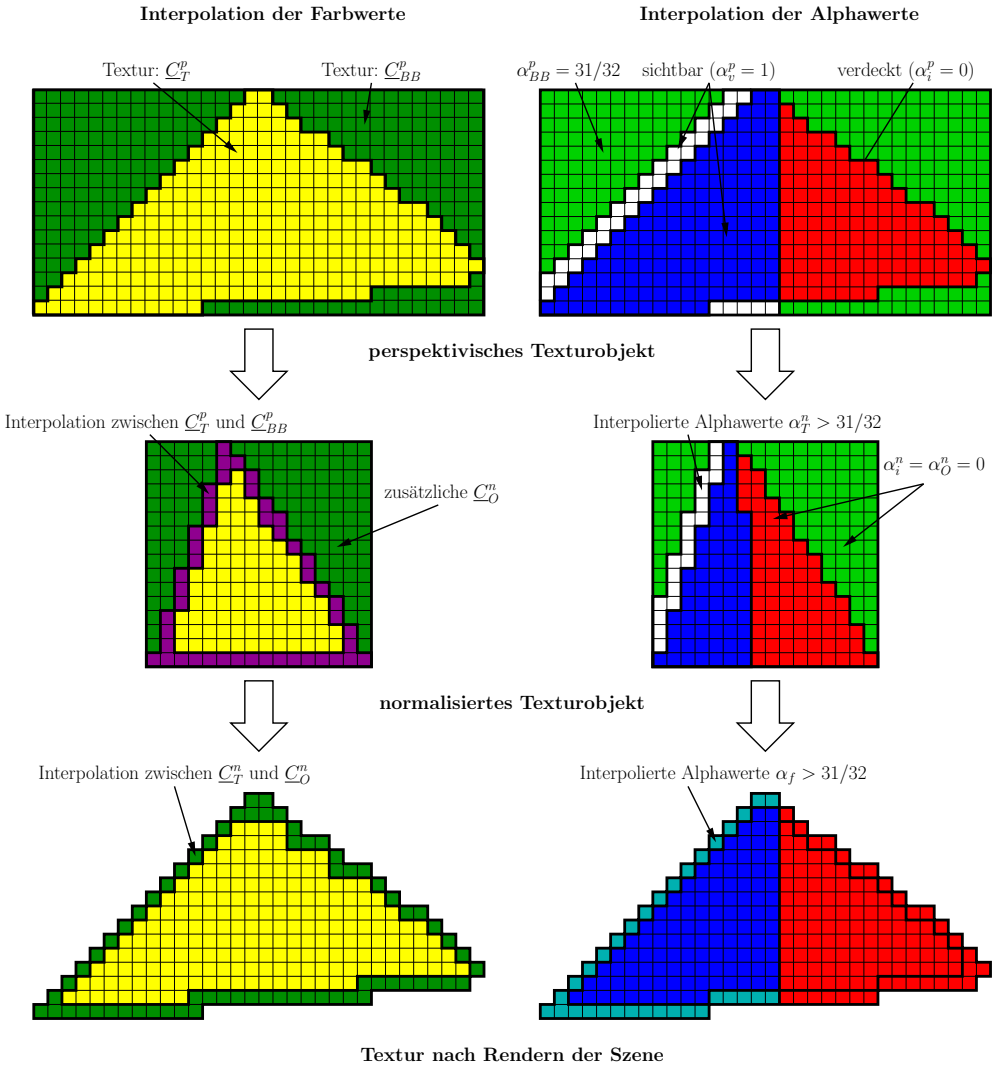


Abbildung 4.19: Interpolation der Farbwerte und Alphawerte bei Normalisierung und Rendern einer Textur ohne Darstellungsfehler

Kapitel 2.5.4 beschrieben, bis zu je zwei Texel in u- und v-Richtung benötigt. Daher ist es notwendig die Verdeckungsrechnung nicht nur für das Polygon, sondern auch für diesen zwei Texel breiten Bereich entlang der Texturgrenzen durchzuführen. Dies lässt sich realisieren, indem zusätzlich die Alphawerte einer Linie zwischen den Eckpunkten des Polygons beim Ablauf der Verdeckungsrechnung verwendet werden. Wie in Kapitel 2.3.2 beschrieben, wird die Breite einer Linie in Fragmenten angegeben. Die Ausdehnung erfolgt um die mit dem Bresenham Algorithmus ermittelten Fragmente in positive und negative Richtung. Mit einer Linienbreite von fünf ist daher sichergestellt, dass jeweils mindestens zwei (maximal drei) Fragmente bei der Verdeckungsrechnung außerhalb des Polygons berücksichtigt werden.

Problematisch ist die Berechnung der Z-Buffer-Werte der Fragmente dieser Linie durch OpenGL. Da sowohl Gleichung 2.2 als auch Gleichung 2.4 nur Näherungen sind, werden

für gleiche Fragmente u. U. leicht unterschiedliche Werte z_{xy_f} berechnet. Dieses Problem ähnelt dem *Z-Fighting*: bei Primitiva mit nahezu gleicher Tiefe kann es durch die begrenzte Genauigkeit passieren, dass teils das eine weiter vorne liegt und gezeichnet wird, teils das andere. Um dies zu verhindern, werden die Linien zwischen den Eckpunkten der Polygone mit einem Offset gerastert. Dieser Offset verändert die Werte $z_{xy_{line}}$ der Fragmente der gerasterten Linien gegenüber den Werten $z_{xy_{pol}}$ des Polygon nur minimal. Die Vorschriften $z_{xy_f} < z_{xy_{line}}$ zur Markierung von Verdeckungen und $z_{xy_f} \geq z_{xy_{line}}$ zur Markierung von sichbaren Fragmenten führen dann zu guten Ergebnissen. Der sich dadurch ergebende Bereich ist (stark vergrößert) in Abbildung 4.18 als weißer Bereich dargestellt.

In Abbildung 4.19 ist anhand des vorherigen Beispiels die Funktionsweise der neuen Verdeckungsrechnung und Normalisierung dargestellt. Für die Normalisierung der Textur gilt nun, dass alle Alphawerte $\alpha > 31/32$ sichtbar sind und gerendert werden müssen. Alle anderen Texel können verworfen werden.

Die beschriebenen Maßnahmen betreffen nur die Normalisierung der Textur. Eine Modifikation der Routine für die Darstellung ist daher nicht erforderlich. In den in Abbildung 4.20 gezeigten Szenen ist zu sehen, dass die getroffenen Maßnahmen zum gewünschten Ergebnis führen. Die im linken Bild noch zu sehenden Linien sind tatsächlich Teil des Bodenbelags. Die Artefakte mit der Silhouette des Roboterarms auf dem Boden haben eine andere Ursache und werden im folgenden Kapitel behandelt.

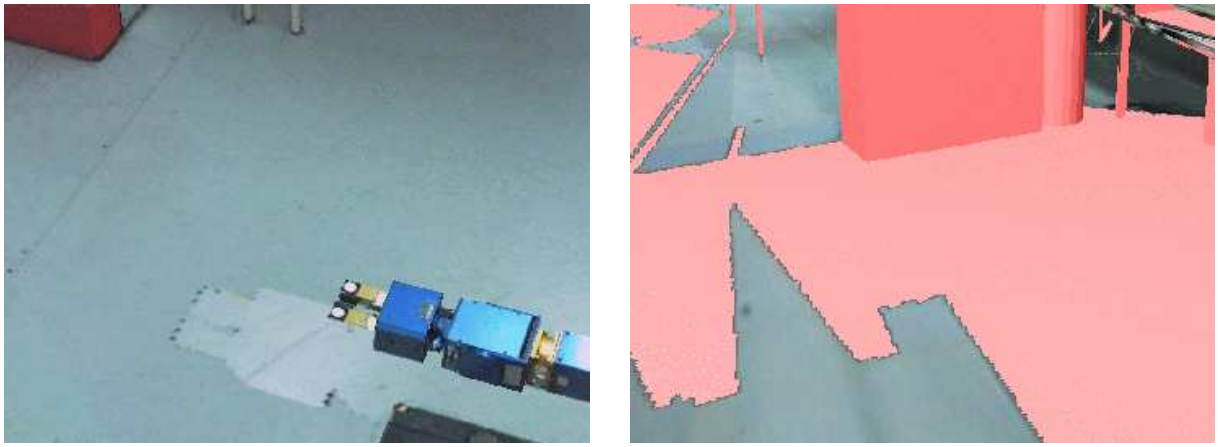


Abbildung 4.20: Gerenderte Szenen mit Vermeidung von Darstellungsfehlern an Polygongrenzen

5 Weiterverarbeitung der Texturen

5.1 Artefaktunterdrückung

5.1.1 Motivation

In den vorangegangenen Kapiteln wurde für einige Daten von perfekten Annahmen ausgegangen, die in der Praxis nicht erfüllt werden können. Statt dessen gibt es immer Ungenauigkeiten bei

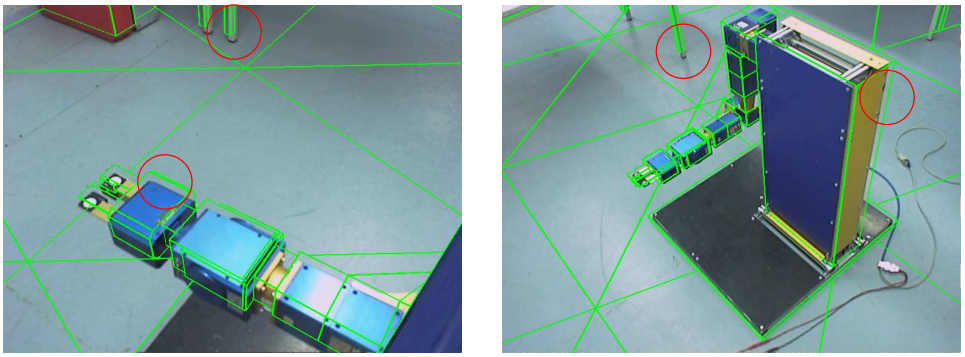
- der Registrierung der Kamera,
- der Kalibrierung der intrinsischen Kameraparameter C_t,
- der Entzerrung der Kamerabilder (Kissenverzerrung, ...) und
- der Modellierung der Szene: Die Beschreibung der Szene mit Polygonen kann immer nur eine Approximation sein.

Die Folge aller dieser Ungenauigkeiten ist die Zuordnung von Pixeln des Kamerabildes zu falschen Polygonen, wie in Abbildung 5.1 für die einzelnen Ursachen dargestellt wird. Bereits kleine Fehler in der geschätzten Rotation der Kamera können beispielsweise zur falschen Zuordnung von großen Bereichen im Kamerabild zu den Polygonen führen.

Dadurch entstehen Artefakte bei der späteren Darstellung der Szene unter Verwendung der gewonnenen Texturen. Derartige Artefakte können immer dann entstehen, wenn zwei Polygone, oder zumindest Teile von ihnen, *im Kamerabild* aneinander grenzen. Dabei lassen sich drei Fälle unterscheiden, die zu unterschiedlichen Artefakten führen:

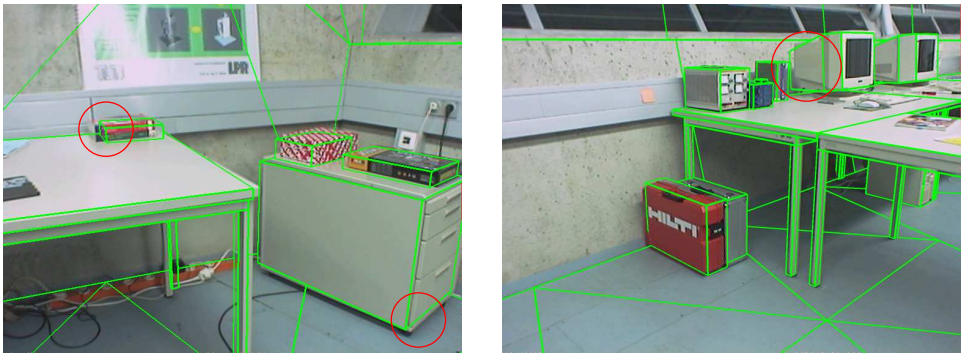
Fall 1: Gemeinsame Kante: Grenzen die beiden beteiligten Polygone auch in der 3D-Geometrie aneinander (Abb. 5.2, links), d. h. sie besitzen eine gemeinsame Kante, so liegen sie auch von jeder beliebigen Betrachterposition aus beisammen. In der Regel sind Artefakte hier kaum zu bemerken (Abb. 5.2, rechts). Erst bei relativ großen Ungenauigkeiten sind Fehler in der Darstellung zu erkennen. Durch die konstante Nähe der beiden Polygone entstehen bei einer Bewegung des Betrachters auch keine Divergenzen durch Aufspaltung eigentlich zusammenhängender Bereiche in der Darstellung.

Fall 2: Falsche Zuordnung zu *verdeckendem* Polygon: Im zweiten Fall besitzen die beiden Polygone keine gemeinsame Kante, sondern liegen nur auf Grund der Position der Kamera im Kamerabild direkt nebeneinander. Hier kann es passieren, dass



a) Registrierungsfehler (1^o Rotation)

b) ungenaue intrinsische Kameraparameter (2% Fehler in Brennweite)



c) nicht entzerrtes Bild (Kissenverzerrung mit $\kappa = -4000$)

d) ungenaue Modellierung

Abbildung 5.1: Ursachen für falsche Zuordnungen zu Polygonen

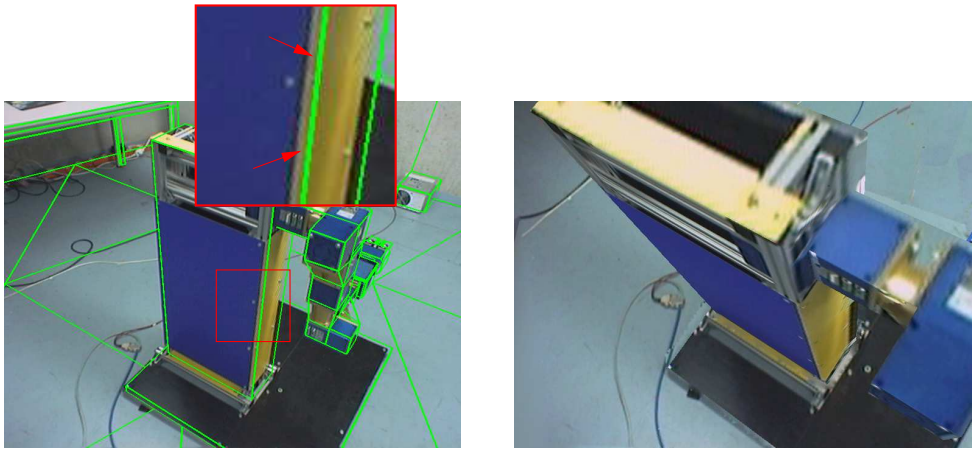


Abbildung 5.2: Falsche Zuordnung zu angrenzendem Polygon: links: Kamerabild mit Modell, rechts: texturierte Szene von anderer Betrachterposition aus

fälschlicherweise dem *verdeckenden* Polygon Bildbereiche zugeordnet werden, die eigentlich zum *verdeckten* Polygon gehören (Abb. 5.3, links). Die entstehenden Artefakte befinden sich naturgemäß immer an der Polygonkante. Dadurch werden sie vom Benutzer als wenig störend akzeptiert, da an diesen Stellen üblicherweise ohnehin ein Farbwechsel im

Bild vorhanden ist. Ist die Betrachterposition bei der Darstellung zusätzlich noch ähnlich der Kameraposition bei der Aufnahme, so sind die entstehenden Artefakte gerade bei einem Polygon im Hintergrund mit relativ homogenenem Farbverlauf gering (Abb. 5.3, rechts).

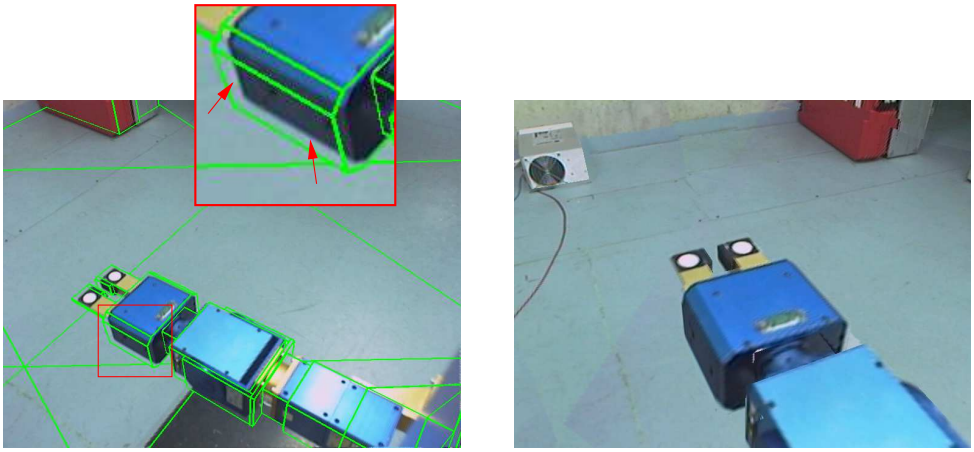


Abbildung 5.3: Falsche Zuordnung zu verdeckendem Polygon: links: Kamerabild mit Modell, rechts: texturierte Szene von anderer Betrachterposition aus

Fall 3: Falsche Zuordnung zu *verdecktem* Polygon: Dieser Fall entspricht Fall 2, nur dass hier Bildbereiche dem *verdeckten* Polygon zugeordnet werden, die eigentlich Teil des *verdeckenden* Polygons sind (Abb. 5.4, links). Diese Artefakte können nicht nur an den Rändern des verdeckten Polygons, sondern an irgend einer Stelle auftreten. Gerade bei relativ homogener Farbverteilung im Polygon können diese Artefakte daher sehr störend sein (Abb. 5.4, rechts). Bei einer Änderung der Betrachterposition wird hier ein Teil der Textur vom vorne liegenden Polygon “abgetrennt”.

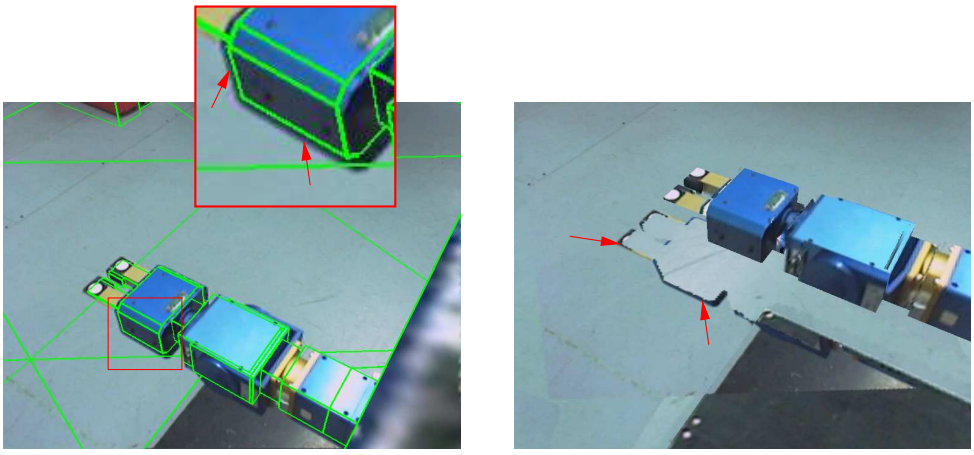


Abbildung 5.4: Falsche Zuordnung zu verdecktem Polygon: links: Kamerabild mit Modell, rechts: texturierte Szene von anderer Betrachterposition aus

Hauptsächlich die aus Fall 3 resultierenden Artefakte werden somit als störend empfunden. Daher wird im Folgenden betrachtet, wie die aus diesem Fall resultierenden Artefakte beseitigt oder zumindest reduziert werden können.

Es ist dabei jedoch nicht möglich eine sichere Aussage zu treffen, welche Zuordnung von Bildbereichen zu Polygonen falsch ist, d. h. es kann keine Interpretation der Farbinformation erfolgen. Sofern keine zusätzlichen Informationen über die Szene vorliegen, kann nicht ermittelt werden, ob ein Pixel zu einem anderen Polygon gehört oder vielleicht tatsächlich beispielsweise ein Farbwechsel im Polygon an dieser Stelle erfolgt. Statt dessen wurden Verfahren entwickelt, um Farbinformation zu verwerfen, bei der unsicher ist, ob sie dem richtigen Polygon zugeordnet wurde. Da die Gültigkeit der Texel in den Alphawerten kodiert ist, entspricht das Verwerfen dabei einer Manipulation der Alphamaske der Textur. Unsicher sind Farbinformationen immer an den Rändern der als verdeckt markierten Bereiche, da diese im Kamerabild unmittelbar an das verdeckende Polygon gegrenzt haben. Die Lösung ist daher eine Expansion dieser Bereiche. Der Algorithmus wird auf die perspektivischen Texturen angewandt (s. Kap. 7.2.1).

5.1.2 Modifikation der Alphamaske durch bilineare Interpolation

Zur Umsetzung auf der Grafik-Hardware wird die bilineare Interpolation von Texturen bei vergrößerter und verkleinerter Darstellung genutzt. Bei den erforderlichen Schritten wird nur der Alphakanal der bearbeiteten Textur verarbeitet, die enthaltene Farbinformation bleibt unverändert. Die Textur wird zunächst mit in beiden Dimensionen halbierter Größe gezeichnet, wobei für die Abbildung $\mathbf{M}_t$ verwendet wird. Dadurch entsteht jedes resultierende Fragment aus den Alphawerten von vier ursprünglichen Texeln. Da beim Zeichnen bilineare Interpolation zum Einsatz kommt, gilt für jedes bei dieser Minification entstehende Fragment

$$\alpha_{minified} \leq \frac{3}{4}, \tag{5.1}$$

falls mindestens eins der beteiligten Texel ungültig war. Die betroffenen Texel werden mit Hilfe eines entsprechenden Alphatests auf $\alpha = 0$ gesetzt. Die resultierende Maske wird in einem temporären Texturobjekt gespeichert. Diese Textur wird dann, erneut unter Verwendung von $\mathbf{M}_t$, wieder auf die ursprüngliche Auflösung vergrößert, wobei hier bilineare Interpolation bei Magnification erfolgt. Dadurch erfüllen alle Texel, zu deren Entstehung mindestens ein als verdeckt markiertes Texel der ursprünglichen Textur beteiligt war

$$\alpha_{magnified} \leq \frac{15}{16}. \tag{5.2}$$

Diese Texel werden erneut auf $\alpha = 0$ gesetzt. Abbildung 5.5 zeigt beispielhaft diese Prozedur. Es bleiben somit nur diejenigen Texel als gültig markiert zu deren Entstehung

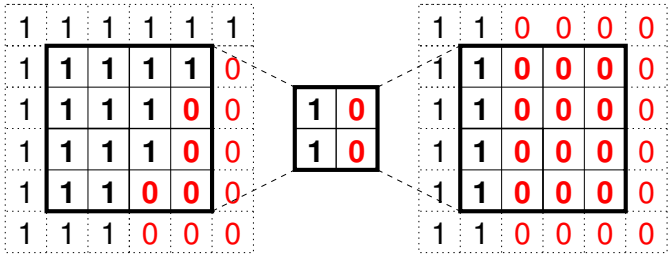


Abbildung 5.5: Ausdehnung ungültiger Bereiche in der Alphamaske

bei diesem Prozess kein ungültiges Texel beigetragen hat. Dies sind die Texel die sich nicht in unmittelbarer Nähe der ungültigen Bereiche befinden.

Hier wird deutlich, warum Bereiche des Texturobjekts außerhalb der perspektivischen Textur auf $\alpha^p_{BB} = \frac{31}{32}$ gesetzt werden. Dieser Wert erfüllt zwei Bedingungen, indem er in der Mitte zwischen zwei Werten liegt:

- $\alpha^p_{BB} > \frac{15}{16}$: Diese Bedingung muss erfüllt werden, damit bei der Ausdehnung der ungültigen Bereiche keine Ausdehnung von als ungültig markierten Texeln der Bounding Box in die Textur hinein erfolgt.
- $\alpha^p_{BB} < 1$: Diese Bedingung muss erfüllt werden, damit keine Artefakte an Polygongrenzen in ungültigen Bereichen entstehen (s. Kap. 4.5.2).

Die *maximale* Vergrößerung beträgt bei der beschriebenen Methode 1→16 Texel für ein einzelnes verdecktes Texel, wie in Abbildung 5.6 illustriert.

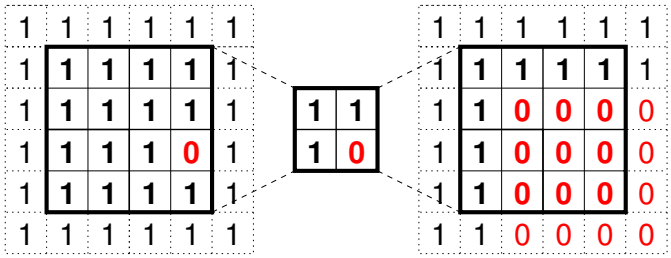


Abbildung 5.6: Maximale Ausdehnung für ein einzelnes ungültiges Texel

Sehr kleine Texturen werden von der Vergrößerung der als verdeckt markierten Bereiche ausgenommen. Sie können sonst durch das beschriebene Verfahren ihre Farbinformation vollständig verlieren.

5.1.3 Iterative Expansion ungültiger Bereiche

Um weitere Vergrößerungen zu erzielen, wird der beschriebene Algorithmus mehrfach durchlaufen. Bei jeder Iteration wird dabei als Eingabe das Resultat des vorherigen Durchlaufs verwendet (s. Abb. 5.7).

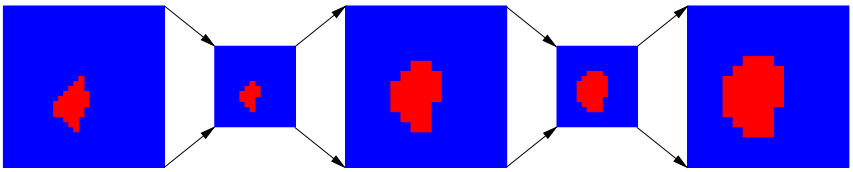


Abbildung 5.7: Alphamaske: Iterative Vergrößerung verdeckter Bereiche: alternierend

Alternativ können zunächst mehrfach Verkleinerungen der Maske erfolgen und das Ergebnis schließlich schrittweise auf die Originalgröße gebracht werden (s. Abb. 5.8).

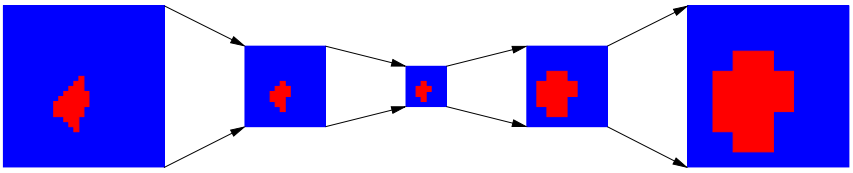


Abbildung 5.8: Alphamaske: Iterative Vergrößerung verdeckter Bereiche: fortschreitend

Die Anzahl der Iterationen stellt einen Kompromiss zwischen dem Verwerfen von möglichst viel falscher Farbinformation und der Erhaltung von möglichst viel korrekt zugeordneten Bildpunkten dar. Nach Durchlaufen aller Iterationen wird die resultierende Alphamaske schließlich in den Alphakanal der Textur kopiert.

Abbildung 5.9 zeigt eine Szene mit Textur-Extraktion bei leicht fehlerhafter Kameraregistrierung. Ohne Artefaktunterdrückung sind hier beispielsweise Teile des Roboterarms am Boden zu sehen. Eine einzelne Iteration des beschriebenen Algorithmus reduziert die Artefakte, entfernt sie aber noch nicht vollständig. Nach zwei Iterationen (hier alternierend) sind in diesem Beispiel keine Artefakte mehr vorhanden. Der grüne Pfeil im Bild markiert ein Polygon für das aufgrund seiner Größe keine Artefaktunterdrückung stattfindet.

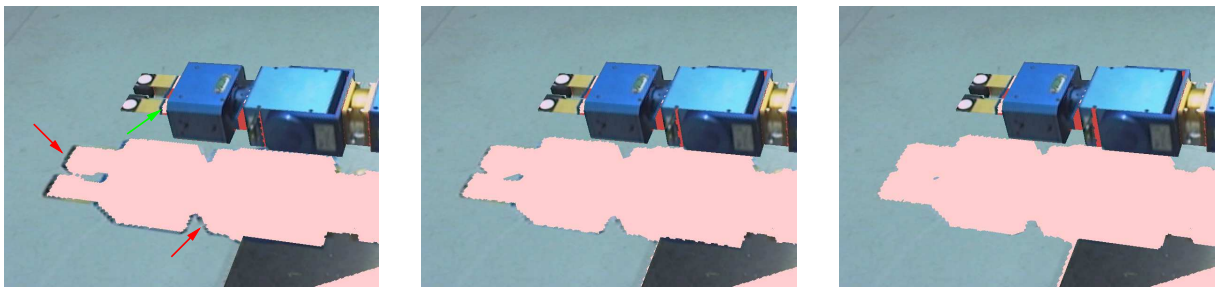


Abbildung 5.9: Artefaktunterdrückung bei Fehler in der Kameraregistrierung

5.1.4 Artefaktunterdrückung mit programmierbarer Hardware

Der Einsatz von programmierbarer Grafik-Hardware (Kap. 2.8) ermöglicht eine gezieltere Manipulation der Alphamaske zum Zweck der Ausdehnung der als ungültig markierten Bereiche. Eine solche Methode mit Verwendung eines Fragment Shaders soll hier vorgestellt werden. Das Verfahren ähnelt dem aus der Bildverarbeitung bekannten *Dilatations-Operator*, der gemeinsam mit der *Erosion* die Basis aller mathematischen morphologischen Operatoren bildet [91]. Die Dilatation einer Menge $X \subset \mathbb{R}^2$ mit einem strukturierenden Element B kann für ein binäres Bild durch

$$\delta_B(X) = X \oplus B = \{x \in \mathbb{R}^2 \,|\, X \cap B_x \neq \emptyset\} \tag{5.3}$$

mit

$$B_x = \{b + x \,|\, b \in B\}. \tag{5.4}$$

beschrieben werden.

Im Kontext dieser Arbeit ist vor allem eine Umsetzung mit Beschleunigung auf der Grafik-Hardware von Bedeutung. Für die Manipulationen der Alphamaske wird dazu jeweils die zu bearbeitende Textur einmal gezeichnet und dabei ein Shader für die Verarbeitung jedes Fragments durchlaufen. Da nur die Werte der Texel verändert werden, aber keine Verzerrungen innerhalb der Textur auftreten sollen, kommt für das Zeichnen erneut $\mathbf{M}_t$ zum Einsatz, um jeweils ein Texel auf ein Pixel abzubilden.

Dabei ist innerhalb des Shaders nur die Berechnung des Alphawerts relevant, RGB und Tiefenwert sind hier nicht von Bedeutung. Für jedes Texel soll das Minimum seines Alphawertes und seiner Nachbarn bestimmt werden. Bereits als ungültig markierte Texel werden dazu im Shader ohne Veränderung als Ausgangs-Alphawert übergeben. Für sie ist keine weitere Bearbeitung notwendig, daher kann die Berechnung hier bereits abgebrochen werden. Für gültige Texel soll getestet werden, ob sie sich nahe eines ungültigen Texels befinden. Dazu werden die vier in Abbildung 5.10 gezeigten Nachbartexel ausgewertet (Elementarraute). Durch Übergabe der Texturauflösung kann dabei innerhalb des Shaders auf beliebige Texel der Textur zugegriffen werden. Mit der dadurch im Shader bekannten Texturgröße können Nachbartexel gezielt durch einen Offset in den Texturkoordinaten angesprochen werden.

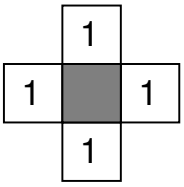


Abbildung 5.10: Filter für die Expansion der als ungültig markierten Bereiche

Nur falls alle Nachbartexel als gültig markiert sind, d. h. das Texel befindet sich nicht unmittelbar bei einer ungültigen Region, soll auch das betrachtete Fragment gültig bleiben.

Daher basiert die Berechnung des neuen Alphawerts auf der Summe der Alphawerte der Nachbartexel:

$$\alpha_{frag} = \begin{cases} 1, & \text{für } \sum_{t=0}^{3} \alpha_t > \alpha_c \\ 0, & \text{sonst} \end{cases} \tag{5.5}$$

wobei für α_c ein beliebiger Wert im Intervall $[3,0 \ldots 4,0]$ verwendet werden kann.

Die aus dem Render-Durchlauf resultierende Alphamaske wird anschließend wieder im Alphakanal der Textur gespeichert.

Da innerhalb eines Shaders nicht auf die Werte anderer Fragmente zugegriffen werden kann, sind zusätzliche Iterationen direkt innerhalb des Shaders nicht möglich. Auch hier kann jedoch durch wiederholte Ausführung eine Verstärkung des Effekts erreicht werden. Der fehlende Zugriff auf benachbarte Fragmente wird somit durch Multipass-Rendering mit wahlfreiem Zugriff auf einzelne Texel des vorherigen Ergebnisses innerhalb des Shaders kompensiert.

In Abbildung 5.11 ist links ein Beispiel für die Expansion der verdeckten Bereiche mit der Methode der alternierenden Skalierungen für verschiedene Iterationen zu sehen, in der mittleren Spalte erfolgten die Skalierungen fortschreitend. Rechts wird auf die gleiche Alphamaske das gerade beschriebene Verfahren angewendet. Es ist zu sehen, dass durch dieses die Ausdehnung der Bereiche gleichmäßiger erfolgt, was auf die Wahl des kleinen Filters zurückzuführen ist. Somit wird bei dieser Methode weniger gültige Farbinformation fälschlicherweise verworfen. Es sind jedoch mehr Iterationen erforderlich, um einen vergleichbaren Expansionseffekt zu erreichen. Wie in Kapitel 7.2.2 zu sehen sein wird, resultiert dies in etwas längeren Ausführungszeiten. Zudem läuft dieses Verfahren nur auf moderner programmierbarer Grafik-Hardware.

5.2 Hole Filling

5.2.1 Motivation

Auch nach der Gewinnung und Fusion vieler Texturen können immer noch Bereiche in den Polygonen existieren die noch nie in einem Kamerabild sichtbar waren. Entsprechend sind für diese Bereiche keine Farbinformationen vorhanden. Das in Kapitel 5.1 beschriebene Verfahrung zur Unterdrückung der Artefakte reduziert die vorhandene Farbinformation noch weiter. Solche störenden “Löcher” in den Texturen verringern den Gesamteindruck der dargestellten Szene für den Benutzer drastisch (rosa Bereiche in Abb. 5.12). Anstatt fotorealistisch zu wirken, offenbaren sich die Bilder als synthetisch. Daher müssen diese Löcher vor dem Benutzer verborgen werden (“Inpainting”). Das Ziel ist dabei keine möglichst korrekte Rekonstruktion der fehlenden Information, die natürlich ohnehin nicht möglich wäre. Statt dessen sollen die Löcher mit Farbinformation gefüllt werden, die dem Benutzer ein subjektiv empfunden möglichst stimmiges Gesamtbild liefern.

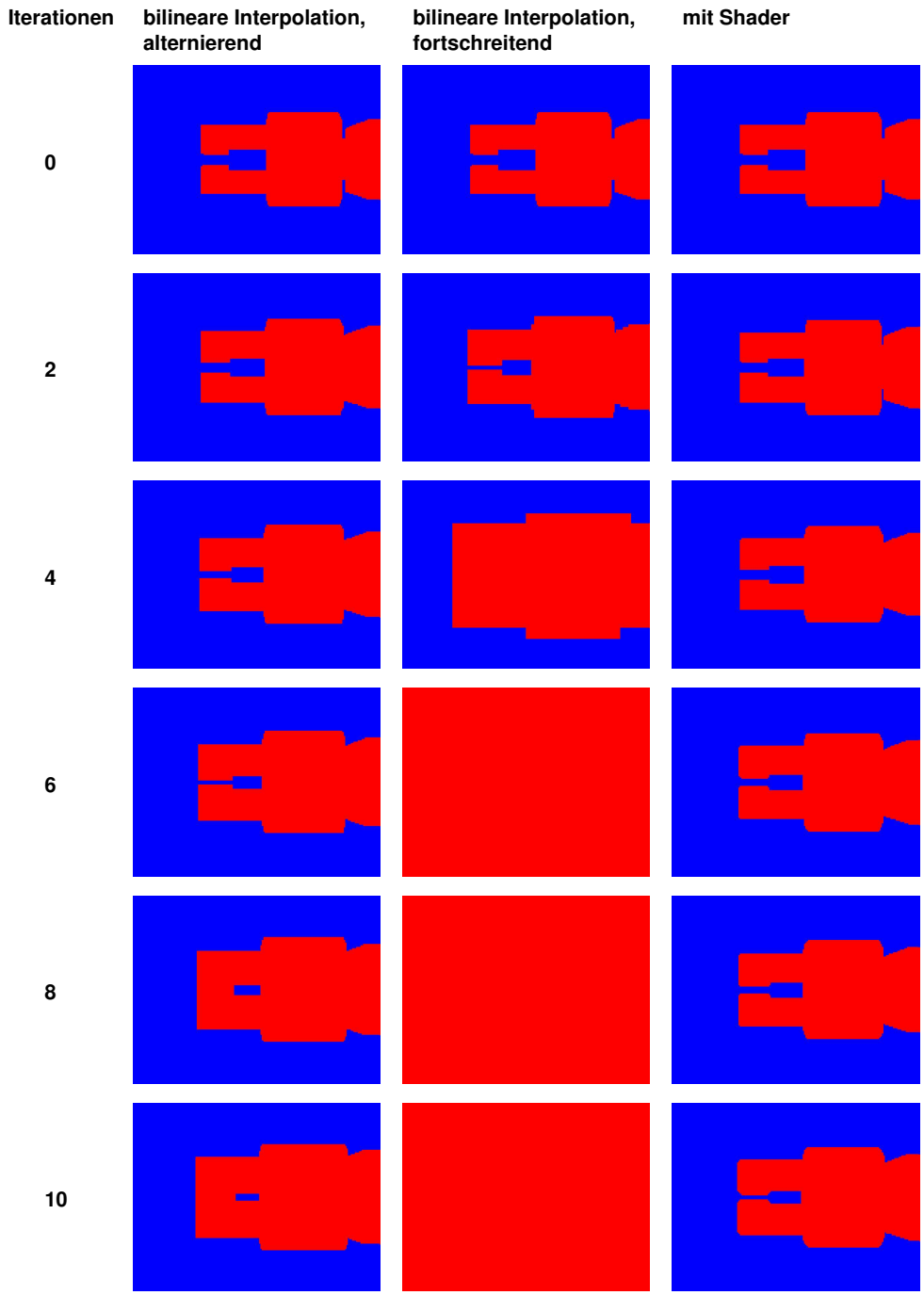


Abbildung 5.11: Vergleich der Alphamasken bei den verschiedenen Verfahren in Abhängigkeit von der Anzahl Iterationen

In [22] wird zum Zweck des Hole Fillings eine Retriangulierung mit Unterteilung von in den Kamerabildern vollständig sichtbaren und vollständig verdeckten Polygonen durchgeführt. Den Eckpunkten der verdeckten Polygone wird anschließend eine Art Mittelwert des nächstliegenden sichtbaren Polygons zugeordnet. Die Interpolation der Farbwerte erfolgt schließlich mit Gouraud-Shading. Eine andere Möglichkeit ist das Füllen dieser

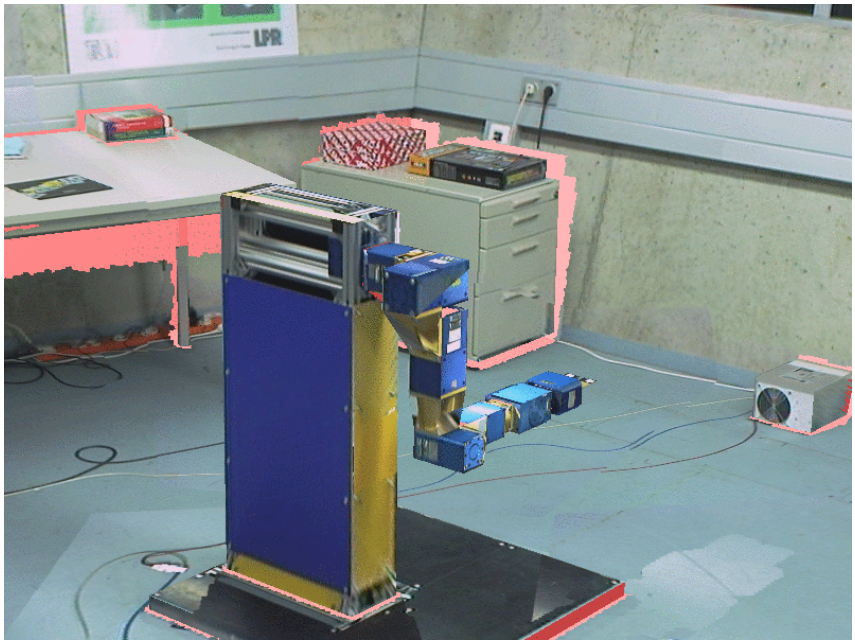


Abbildung 5.12: Synthetische Szene mit Löchern in den Texturen (rosa)

Löcher im Bildraum [23, 59]. Für jeden darzustellenden Frame muss dabei ein Hole Filling Algorithmus durchlaufen werden, was natürlich die für die Darstellung erforderliche Rechenzeit erhöht. Außerdem entstehen dadurch, dass die Interpolation keinen Bezug zwischen Bildpunkten und zugehörigem Polygon herstellt und somit auch zwischen Farben unterschiedlicher Polygone interpoliert, neue Artefakte. Bei einer Bewegung der Betrachterposition tritt durch die ständig neu kalkulierte Interpolation auch Flimmern auf; vor allem durch Auf- und Verdeckung von Polygonen entsteht zusätzliche Unruhe.

Im verwendeten Ansatz werden die Löcher daher statt dessen im Texturraum gefüllt. Dadurch wird der beschriebene Algorithmus nicht pro Frame, sondern jeweils nur einmalig, bei der Generierung der jeweiligen Textur, durchlaufen.

5.2.2 Pyramiden-basierter Ansatz

Als Lösung werden die Farbwerte von den Rändern der Löcher verwendet, um die Löcher schrittweise zu füllen. Eine Operation für das Füllen der Löcher durch Interpolation steht unter OpenGL nicht zur Verfügung. Im Folgenden wird beschrieben wie der gewünschte Effekt trotzdem auf der Grafik-Hardware realisiert werden kann.

Für jedes Polygon wird dazu eine *Texturpyramide* aufgebaut. Das hier vorgestellte Verfahren ähnelt grundsätzlich der z. B. in [73] beschriebenen Bandpasszerlegung mit *Gauß-Pyramiden*. Die Differenzen der einzelnen Stufen bilden eine *Laplace-Pyramide*. Die Anwendung erfolgt in der vorliegenden Arbeit auf Texturen statt auf Bilder. Von Bedeutung

ist hier erneut vor allem die Realisierung auf der Grafik-Hardware zur Beschleunigung. Zusätzlich müssen hier die Löcher beim Aufbau der Pyramide berücksichtigt werden.

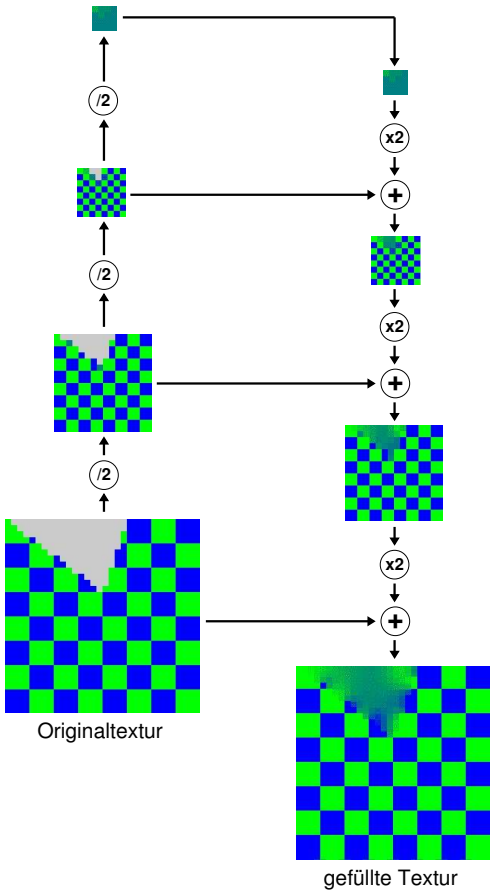


Abbildung 5.13: Textur-Pyramide, links: Aufbau, rechts: Kollabieren

Begonnen wird der Aufbau der Pyramide mit der Textur in Originalgröße als Basis. Bei jedem Schritt werden anschließend Höhe und Breite der Textur halbiert. Der Alphawert eines Texels einer Stufe wird aus den vier betreffenden Texeln der vorhergehenden Stufe mit

$$\alpha_{i+1} = \begin{cases} 1, & \text{für } \sum_{t=0}^{3} \alpha_{i,t} > 0 \\ 0, & \text{für } \sum_{t=0}^{3} \alpha_{i,t} = 0 \end{cases} \tag{5.6}$$

bestimmt. Der Farbwert ist nur für $\alpha_{i+1} = 1$ definiert, seine Berechnung erfolgt mit

$$\underline{C}_{i+1} = \frac{\sum_{t=0}^{3} \alpha_{i,t} \underline{C}_{i,t}}{\sum_{t=0}^{3} \alpha_{i,t}} \tag{5.7}$$

Muss die Textur nur noch in eine Richtung verkleinert werden, d. h. in die andere Richtung ist bereits die Größe 1 erreicht, so gilt entsprechend:

$$\alpha_{i+1} = \begin{cases} 1, & \text{für } \sum_{t=0}^{1} \alpha_{i,t} > 0 \\ 0, & \text{für } \sum_{t=0}^{1} \alpha_{i,t} = 0 \end{cases} \tag{5.8}$$

$$\underline{C}_{i+1} = \frac{\sum_{t=0}^{1} \alpha_{i,t}\underline{C}_{i,t}}{\sum_{t=0}^{1} \alpha_{i,t}} \tag{5.9}$$

Erfolgt nur noch eine Halbierung der Textur in der Breite, so liegen die beiden beteiligten Texel horizontal beisammen. Bei Verkleinerung in der Höhe befinden sie sich entsprechend vertikal nebeneinander.

Die Gleichungen entsprechen jeweils einer Interpolation der Farbwerte der *gültigen* Texel. Jede Stufe ergibt sich aus der vorherigen Stufe durch eine Tiefpassfilterung dieser Texel. Das Resultat wird als Textur gespeichert und ist die Eingangstextur für die nächste Interpolationsstufe. Die letzte Stufe enthält schließlich nur noch ein einziges Texel. (Abbildung 5.13, links). Insgesamt sind

$$I_{F1} = log_2(max(s_n, t_n)) - 1 \tag{5.10}$$

Durchläufe erforderlich, um die vollständige Pyramide aufzubauen.

5.2.3 Subsampling und selektive Interpolation

Die Gleichungen können nicht direkt in OpenGL-Kommandos umgesetzt werden, da in der Grafik-Pipeline die Interpolation *vor* dem Alphatest durchgeführt wird (s. Abb. 2.2 und 2.4). Während der Interpolation bei Texturierung kann daher nicht zwischen gültigen und ungültigen Texeln unterschieden werden. Auf Grund der fest vorgegebenen Art der Interpolation kann auch nicht die seit Version 1.4 im OpenGL-Core enthaltene automatische Generierung von Mip-Maps verwendet werden.

Statt dessen erfolgt zunächst viermal ein Subsampling der Textur, sodass jede der resultierenden Subtexturen eines der Texel enthält, das für die Interpolation für das korrespondierende Texel der nächsten Stufe verwendet wird. Abbildung 5.14 verdeutlicht dies links.

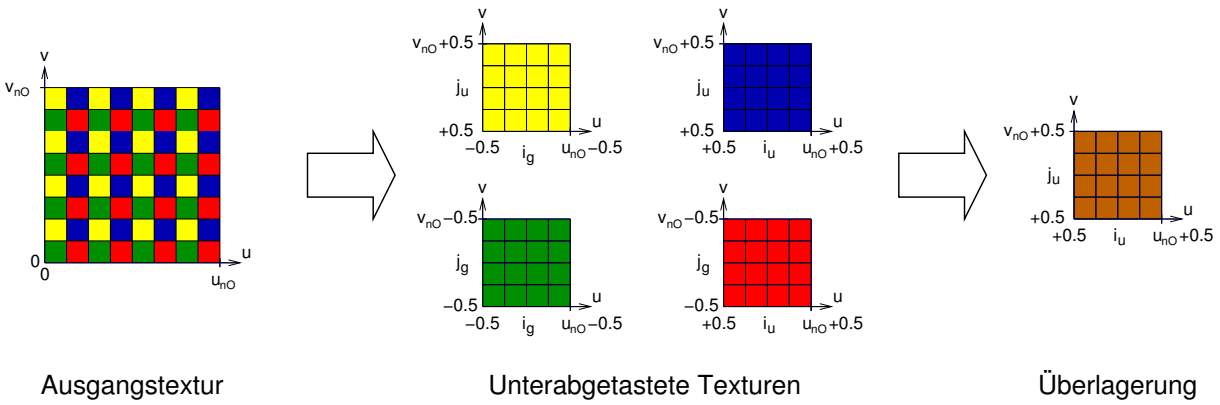


Abbildung 5.14: Subsampling und Überlagerung einer Textur

Dies wird durch viermaliges Zeichnen der Ursprungs-Textur mit Nearest-Neighbor-Filterung erreicht. Beim Rendern müssen dazu die geraden Texelkoordinaten

$$\{(i_g, j_g) \quad | \quad (\{0, 2, 4, \cdots, 2^m - 2\}, \{0, 2, 4, \cdots, 2^n - 2\})\} \tag{5.11}$$

bzw. die ungeraden Texelkoordinaten

$$\{(i_u, j_u) \quad | \quad (\{1, 3, 5, \cdots, 2^m - 1\}, \{1, 3, 5, \cdots, 2^n - 1\})\} \tag{5.12}$$

auf die korrespondierenden Bildschirmkoordinaten

$$\{(x_f, y_f) \quad | \quad (0 \leq i < 2^{m-1}, 0 \leq i < 2^{n-1})\} \tag{5.13}$$

abgebildet werden. Für die exakte Berechnung der Texelkoordinaten der Subsampling Texturen muss jeweils ein Offset für die Texturkoordinaten angegeben werden. Es ergeben sich damit für eine Textur der Größe $(u, v) = (2^m, 2^n)$ folgende Texturkoordinaten für das Subsampling:

$$(u_{min_g}, v_{min_g}) = (-\frac{1}{2}, -\frac{1}{2}); \qquad (u_{max_g}, v_{max_g}) = (2^m - \frac{1}{2}, 2^n - \frac{1}{2}) \tag{5.14}$$

$$(u_{min_u}, v_{min_g}) = (\frac{1}{2}, -\frac{1}{2}); \qquad (u_{max_u}, v_{max_g}) = (2^m + \frac{1}{2}, 2^n - \frac{1}{2}) \tag{5.15}$$

$$(u_{min_u}, v_{min_u}) = (\frac{1}{2}, \frac{1}{2}); \qquad (u_{max_u}, v_{max_u}) = (2^m + \frac{1}{2}, 2^n + \frac{1}{2}) \tag{5.16}$$

$$(u_{min_g}, v_{min_u}) = (-\frac{1}{2}, \frac{1}{2}); \qquad (u_{max_g}, v_{max_u}) = (2^m - \frac{1}{2}, 2^n + \frac{1}{2}) \tag{5.17}$$

Die vier so gewonnenen Texturen müssen anschließend entsprechend den Gleichungen 5.6 bis 5.9 zu einer einzelnen kombiniert werden (Abb. 5.14, rechts). Für die Realisierung mit konfigurierbarer Hardware kommen die seit der GeForce256-Generation auf NVidia-Grafikkarten verfügbaren *Register Combiners* zum Einsatz. Diese OpenGL-Erweiterung ersetzt, wenn aktiviert, unter anderem die Standard Textur-Umgebung und die Berechnung der Farbsumme pro Fragment durch Kombination mehrerer konfigurierbarer Stufen. Der Aufbau für eine GeForce3-Grafikkarte ist in Abbildung A.1 skizziert. Bei Aktivierung werden zunächst eine oder mehrere *General Combiner* Stufen (Abbildungen A.2 und A.3) durchlaufen, eine *Final Combiner* Stufe berechnet schließlich Farbe und Alphawert des Fragments. Ein Satz von Registern (Tabelle A.1), die unter anderem die entsprechende Texelfarbe der Textur-Einheiten enthalten, fungiert als Input (bis zu vier Register) und Output der einzelnen Stufen. Über die *Combiner Functions* (Tabellen A.2 und A.3) werden die Ausgangsdaten und in der letzten Stufe der neue Farbwert und Alphawert für das Fragment bestimmt. Auf die Eingangswerte können vor der eigentlichen Funktion ein paar Operationen angewandt werden (Tabelle A.4), für die Ausgangswerte können bestimmte Skalierungen und Offsets erfolgen (Tabelle A.5).

Für die Ausführung des beschriebenen Algorithmus ist mindestens eine GeForce3-Grafikkarte erforderlich. Ihre vier Textureinheiten und die acht General Combiner Stufen

ermöglichen eine korrekte Interpolation der Farbwerte nach den Gleichungen 5.6 bis 5.7 in einem Durchgang. Dabei erfolgt eine Realisierung mit sechs General Combiner Stufen, die im Folgenden kurz beschrieben werden. Die Nutzung der zusätzlichen Funktionalität der Final Combiner Stufe ist nicht erforderlich.

Combiner Stufen 1-3 (Abb. 5.15):

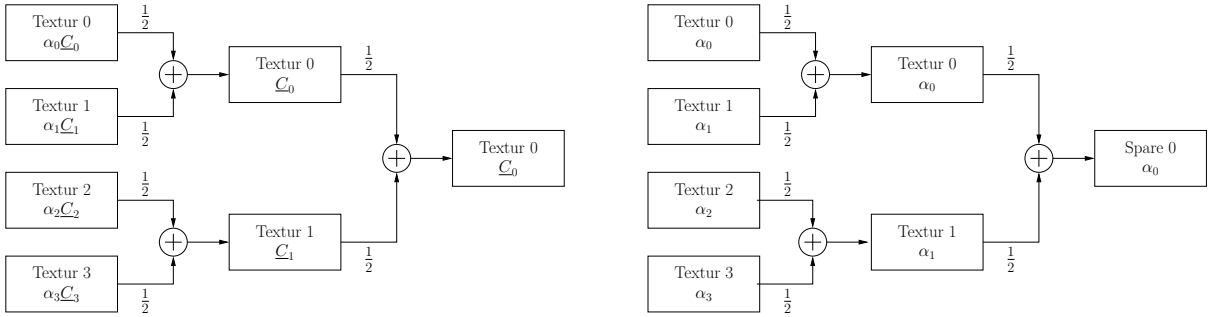


Abbildung 5.15: Register Combiners Stufen 1-3, links: RGB, rechts: α

Die Farbvektoren und Alphawerte der vier Subsampling-Texturen werden in den Texureinheiten 0 bis 3 gespeichert. Die Farbvektoren der Texel mit gleichen Texelkoordinaten in den vier Texturen werden mit ihren Alphawerten multipliziert und mit einer jeweiligen Gewichtung von $\frac{1}{4}$ addiert:

$$\underline{C}_{Stage1-3} = \frac{1}{4}(\alpha_0\underline{C}_0 + \alpha_1\underline{C}_1 + \alpha_2\underline{C}_2 + \alpha_3\underline{C}_3) \tag{5.18}$$

Die Alphawerte werden ebenfalls mit einer jeweiligen Gewichtung von $\frac{1}{4}$ addiert:

$$\alpha_{Stage1-3} = \frac{1}{4}(\alpha_0 + \alpha_1 + \alpha_2 + \alpha_3) \tag{5.19}$$

Combiner Stufe 4 (Abb. 5.16):

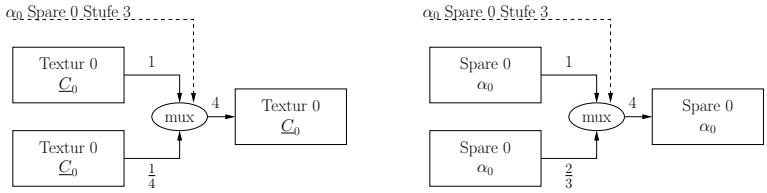


Abbildung 5.16: Register Combiners Stufe 4, links: RGB, rechts: α

Ist der Alphawert der dritten Combiner Stufe $\alpha_{Stage1-3} < 0.5$, so bedeutet dies, dass nur ein Texel der ursprünglichen Textur als sichtbar markiert war. Der zugehörige Farbvektor $\underline{C}_{Stage1-3}$ wird mit dem Faktor vier gewichtet. Ist der Alphawert $\alpha_{Stage1-3} \geq 0.5$, dann ist

mehr als ein Texel als sichtbar markiert und der Farbvektor $\underline{C}_{Stage1-3}$ wird in dieser Stufe nicht verändert:

$$\underline{C}_{Stage4} = \begin{cases} 4\,\underline{C}_{Stage1-3}, & \text{für } \alpha_{Stage1-3} < 0.5 \\ \underline{C}_{Stage1-3}, & \text{sonst} \end{cases} \tag{5.20}$$

Für $\alpha_{Stage1-3} < 0.5$ wird der Alphawert ebenfalls mit dem Faktor vier gewichtet, um den zugehörigen Farbvektor $\underline{C}_{Stage1-3}$ für die nächsten Combiner Stufen nun als gültig zu markieren und ihn damit vor weiteren Bearbeitungsschritten zu schützen. Ist der Alphawert $\alpha_{Stage1-3} \geq 0.5$, wird er mit dem Faktor $\frac{2}{3}$ gewichtet, um die Fallunterscheidung (mux) der nächsten Combiner Stufe zu ermöglichen:

$$\alpha_{Stage4} = \begin{cases} 4\,\alpha_{Stage1-3}, & \text{für } \alpha_{Stage1-3} < 0.5 \\ \frac{2}{3}\,\alpha_{Stage1-3}, & \text{sonst} \end{cases} \tag{5.21}$$

Combiner Stufe 5 (Abb. 5.17):

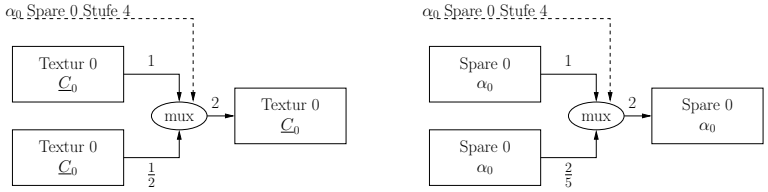


Abbildung 5.17: Register Combiners Stufe 5, links: RGB, rechts: α

Ist der Alphawert der vierten Combiner Stufe $\alpha_{Stage4} < 0.5$, dann sind zwei Texel aus den Subsampling Texturen als sichtbar markiert. Der zugehörige Farbvektor $\underline{C}_{Stage4}$ wird mit dem Faktor zwei gewichtet. Bei $\alpha_{Stage4} \geq 0.5$ bleibt der Farbvektor $\underline{C}_{Stage4}$ wieder unverändert:

$$\underline{C}_{Stage5} = \begin{cases} 2\,\underline{C}_{Stage4}, & \text{für } \alpha_{Stage4} < 0.5 \\ \underline{C}_{Stage4}, & \text{sonst} \end{cases} \tag{5.22}$$

Für $\alpha_{Stage4} < 0.5$ wird der Alphawert ebenfalls mit dem Faktor zwei gewichtet. Bei einem Alphawert $\alpha_{Stage4} \geq 0.5$ wird er mit dem Faktor $\frac{2}{5}$ gewichtet:

$$\alpha_{Stage5} = \begin{cases} 2\,\alpha_{Stage4}, & \text{für } \alpha_{Stage4} < 0.5 \\ \frac{2}{5}\,\alpha_{Stage4}, & \text{sonst} \end{cases} \tag{5.23}$$

Combiner Stufe 6 (Abb. 5.18):

In der sechsten Combiner Stufe wird der Farbvektor $\underline{C}_{Stage5}$ mit dem Faktor $\frac{4}{3}$ gewichtet, falls der Alphawert $\alpha_{Stage5} < 0.5$ ist (drei Texel gültig). Bei $\alpha_{Stage5} \geq 0.5$ bleibt der Farbvektor $\underline{C}_{Stage5}$ wieder, wie in den vorherigen beiden Stufen, unverändert (alle vier Texel gültig):

$$\underline{C}_{Stage6} = \begin{cases} \frac{4}{3}\,\underline{C}_{Stage5}, & \text{für } \alpha_{Stage5} < 0.5 \\ \underline{C}_{Stage5}, & \text{sonst} \end{cases} \tag{5.24}$$

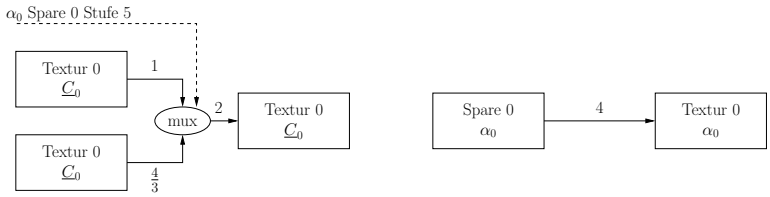


Abbildung 5.18: Register Combiners Stufe 6, links: RGB, rechts: α

Der Alphawert α_{Stage5} wird in der sechsten Combiner Stufe mit dem Faktor vier gewichtet, um einen Alphawert $\alpha_{Stage6} \geq 1$ zu erhalten:

$$\alpha_{Stage6} = 4\,\alpha_{Stage5} \tag{5.25}$$

Sind alle vier Alphawerte der Subsampling Texturen $\alpha = 0$, werden sie durch die Operationen in den sechs Combiner Stufen nicht verändert. Das resultierende Texel ist immer noch mit einem Alphawert $\alpha = 0$ markiert. In allen anderen Fällen wird ein neuer Farbvektor $\underline{C}_{neu}$ nach Gleichung 5.7 berechnet und der zugehörige Alphawert auf $\alpha = 1$ gesetzt (Clamping der Werte $\alpha > 1$).

Anzahl sichtbarer Texel	0	1	2	3	4
Combiner Stufe	**Alphawerte**				
Combiner Stufe 1-3	0	**0.25**	0.5	0.75	1
Combiner Stufe 4	0	1	**0.333**	0.5	0.666
Combiner Stufe 5	0	0.8	0.666	**0.4**	0.533
Combiner Stufe 6	0	1	1	1	1

Tabelle 5.1: Alphawerte in den Combiner Stufen in Abhängigkeit von der Anzahl der gültigen Texel

In Tabelle 5.1 sind alle möglichen Alphawerte in den einzelnen Combiner Stufen dargestellt. Die Werte, die in den einzelnen Stufen die jeweils getestete Anzahl gültiger Texel detektieren, sind hervorgehoben[1)]. Die Skalierungen werden mit der internen Skalierung und dem Constant Color Register durchgeführt. Dafür ist es notwendig, dass in jeder Combiner Stufe unterschiedliche Werte für Constant Color angegeben werden können, weshalb zusätzlich die OpenGL-Erweiterung *Register Combiners 2* erforderlich ist. Diese ist auch aufgrund der Anzahl der verwendeten Combiner Stufen notwendig.

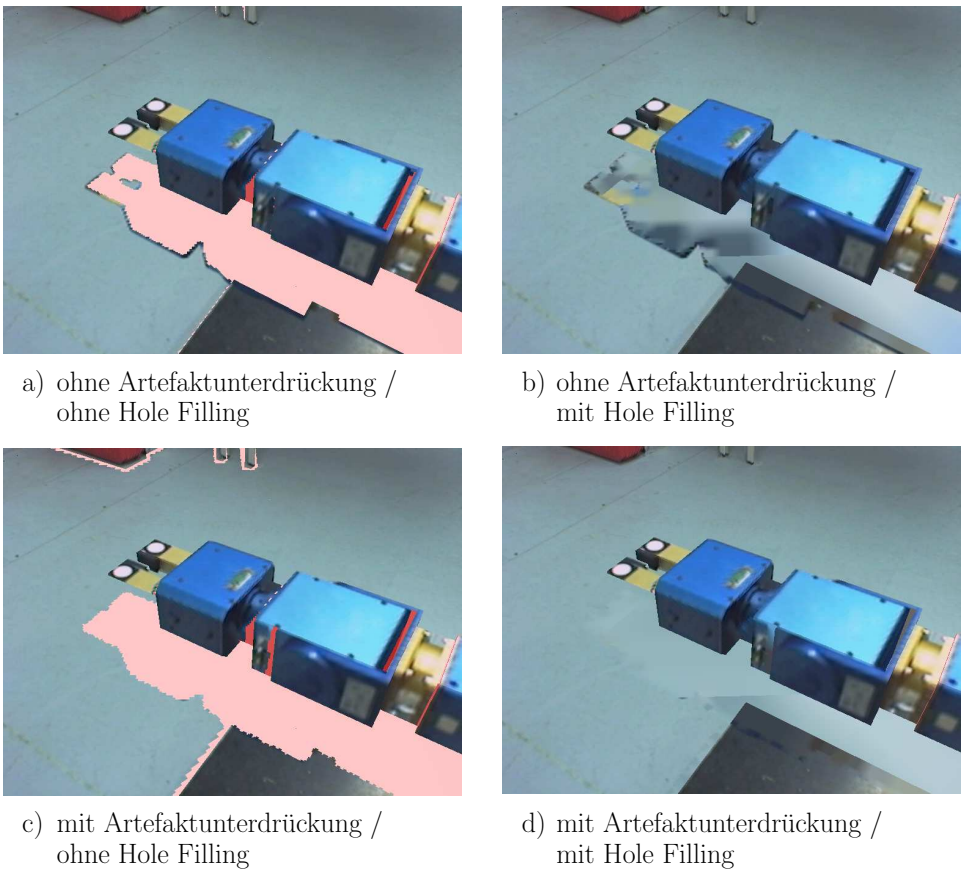
Die in Kapitel 5.2.2 angesprochene Interpolation mit jeweils nur noch zwei Texeln ist hier nicht beschrieben, sie erfolgt analog.

[1)] Die Alphawerte könnten auch anders gewählt werden; entscheidend ist nur, dass die Grenzwerte im richtigen Intervall liegen.

5.2.4 Kollabieren der Pyramide

Nachdem die Pyramide vollständig aufgebaut wurde, wird sie kollabiert (Abb. 5.13, rechts). Dazu wird die Standard Textur-Umgebung der Grafik-Pipeline verwendet, d. h. die Register Combiners sind hier wieder deaktiviert. Die kleinste Textur (1x1 Texel) wird zunächst auf 2x2 Texel vergrößert. Die gültigen Texel der 2x2-Textur der Pyramide überschreiben unter Verwendung eines Alphatests den Inhalt dieses Ergebnisses. Das Resultat wird auf 4x4 Texel vergrößert und dient erneut als Input für die Summe. Dies wird fortgesetzt, bis die Originalgröße erreicht ist. Die Textur enthält nun keine Löcher mehr, falls mindestens ein Texel der ursprünglichen Textur gültig war. Bei den Vergrößerungen wird jeweils bilineare Interpolation verwendet, um einen weichen Übergang zwischen den fusionierten Stufen in der gefüllten Textur zu erzielen.

Um die Fusion mit Texturen aus neuen Kamerabildern zu ermöglichen, muss in den Texturen markiert sein, welche Texel direkt aus einer Aufnahme stammen und welche durch Interpolation entstanden sind. Daher wird im Anschluss an den Hole Filling Algorithmus die ursprüngliche Alphamaske restauriert.



a) ohne Artefaktunterdrückung / ohne Hole Filling

b) ohne Artefaktunterdrückung / mit Hole Filling

c) mit Artefaktunterdrückung / ohne Hole Filling

d) mit Artefaktunterdrückung / mit Hole Filling

Abbildung 5.19: Hole Filling anhand einer Beispielszene

Die Abbildungen 5.19 a und b zeigen den Effekt, wenn Hole Filling ohne vorherige Artefaktunterdrückung durchgeführt wird. Die fehlerhaften Texel an den Rändern der als

verdeckt markierten Bereiche werden beim Hole Filling in das Innere der Löcher interpoliert. Dadurch werden die Fehler sogar noch sichtbarer und störender. Bei vorheriger Unterdrückung der Artefakte verschwindet dieser Effekt für das dargestellte Beispiel nahezu vollständig (Abb. 5.19 c und d).

In Abbildung 5.20 ist die gleiche Szene wie in Abbildung 5.12 zu sehen. Diesmal wurden die Löcher mit der beschriebenen Methode gefüllt. Es ist hier kaum mehr zu erkennen an welchen Stellen die Löcher waren.

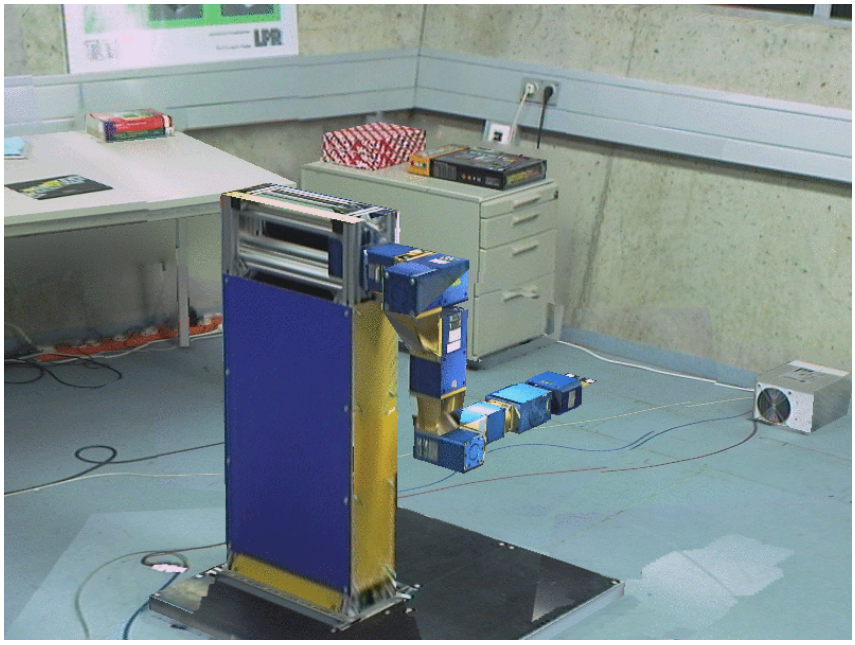


Abbildung 5.20: Synthetische Szene mit gefüllten Löchern in den Texturen

5.2.5 Hole Filling mit programmierbarer Hardware

Mit programmierbarer Hardware kann eine iterative Interpolation der Farbwerte von den Rändern der Löcher nach innen realisiert werden. Dabei kommt, ähnlich wie in Kapitel 5.1.4, ein Fragment Shader zum Einsatz. Auch hier wird dabei der Shader beim Zeichnen der Textur mit $\mathbf{M}_t$ verwendet. Gezielte Zugriffe auf einzelne Texel sind erneut durch Offset in den Texturkoordinaten durch die übergebene Texturauflösung möglich. Diesmal erfolgt jedoch nicht nur eine Manipulation des Alphawerts, sondern es werden auch die RGB-Werte im Shader berechnet.

Zunächst erfolgte dabei eine Umsetzung mit dem in Abbildung 5.21 links zu sehenden Filter. Dieses wird nur auf ungültige Texel, d. h. mit $\alpha < 0.5$, angewandt. Das verwendete Filter ähnelt einem Tiefpassfilter, jedoch werden auch hier nur gültige Texel

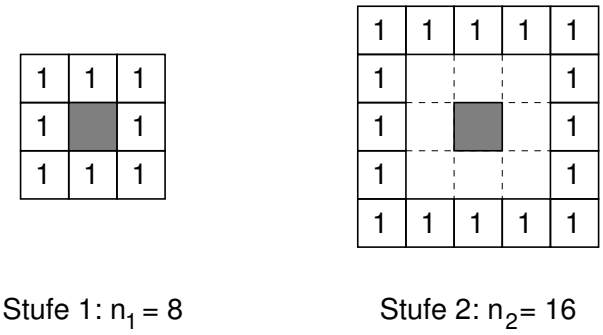


Abbildung 5.21: Filter für den Hole Filling Algorithmus

berücksichtigt. Dazu werden die Gleichungen 5.6 und 5.7 von vier beteiligten Texeln auf n_1 Texel erweitert:

$$\alpha_{i+1}^{stage1} = \begin{cases} 1, & \text{für } \sum_{t=0}^{n_1-1} \alpha_{i,t} > 0 \\ 0, & \text{für } \sum_{t=0}^{n_1-1} \alpha_{i,t} = 0 \end{cases} \tag{5.26}$$

$$\underline{C}_{i+1}^{stage1} = \frac{\sum_{t=0}^{n_1-1} \alpha_{i,t} \underline{C}_{i,t}}{\sum_{t=0}^{n_1-1} \alpha_{i,t}} = \frac{\underline{c}_1}{a_1} \tag{5.27}$$

Mit diesem Filter konnten die Löcher zwar gefüllt werden, es entstanden jedoch deutlich sichtbare Kanten, wie das Beispiel in Abbildung 5.23 b zeigt. Diese resultieren daraus, dass jeweils nur ein einzelnes gültiges Texel in der Diagonalen erfasst wurde. Dieses wurde bei jeder Iteration ohne Interpolation mit anderen Texeln weiter übernommen. Dieser störende Effekt kann auch durch eine alleinige Vergrößerung der Filtermaske nicht vermieden werden. Als Lösung wurde ein auf dem ersten Filter aufbauendes zweistufiges Verfahren realisiert. Für $\alpha_{i+1}^{stage1} = 0$ wird die Ausführung des Shaders dazu abgebrochen, d. h. das Texel bleibt ungültig (Abb. 5.22, Fall 1). Für $\alpha_{i+1}^{stage1} > 0$, d. h. mindestens eins der beteiligten Texel ist gültig (Abb. 5.22, Fall 2), wird mit einem weiteren Filter eine zweite Stufe durchlaufen (Abb. 5.21, rechts). Diese berechnet die endgültigen Alpha- und RGB-Werte dann aus

$$\alpha_{i+1}^{stage2} = 1 \tag{5.28}$$

$$\underline{C}_{i+1}^{stage2} = \frac{\sum_{t=0}^{n_1+n_2-1} \alpha_{i,t} \underline{C}_{i,t}}{\sum_{t=0}^{n_1+n_2-1} \alpha_{i,t}} = \frac{\underline{c}_1 + \sum_{t=n_1}^{n_1+n_2-1} \alpha_{i,t} \underline{C}_{i,t}}{a_1 + \sum_{t=n_1}^{n_1+n_2-1} \alpha_{i,t}} \tag{5.29}$$

Durch die Berücksichtigung der zusätzlichen n_2 Texel im Umkreis des ersten Filters wird sichergestellt, dass immer mehrere Texel für die Interpolation verwendet werden. In Abbildung 5.23 c ist zu sehen, dass dadurch harte Kanten bei der Interpolation vermieden werden. Ein Sonderfall ist die Erfassung eines einzelnen *isolierten* gültigen Texels durch das erste Filter (Abb. 5.22, Fall 3). Für diesen Fall ist die Verwendung dieses einzelnen Texels für die Interpolation jedoch gerechtfertigt. Die in der ersten Stufe berechneten Summen $\underline{c}_1$ und a_1 können für die Berechnung der Gesamtsummen in der zweiten Stufe genutzt werden.

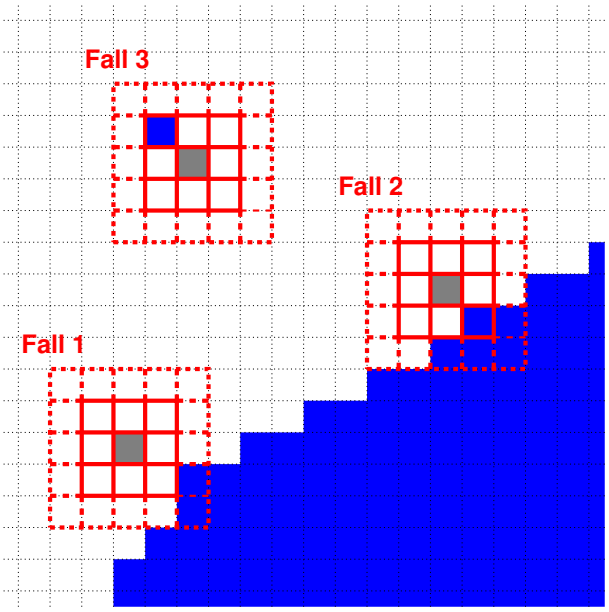


Abbildung 5.22: Erfassung gültiger Texel mit der ersten Filter-Stufe

Ein Problem bei diesem Verfahren ist, dass

$$I_{F2} = max(s_n - 1, t_n - 1) \tag{5.30}$$

Iterationen erfolgen müssen, um sicher zu stellen, dass alle Löcher gefüllt sind. Im Gegensatz zum pyramiden-basierten Ansatz, bei dem die Anzahl der Iterationen logarithmisch mit der Texturauflösung ansteigt (s. Gleichung 5.10) nimmt sie hier somit linear zu. Dies führt zu einem drastischen Anstieg der erforderlichen Rechenzeit. Wünschenswert ist es, keine Iterationen mehr durchzuführen, sobald keine Texel mehr geschrieben werden müssen, d. h. sobald alle Löcher gefüllt sind. Zu diesem Zweck kommt die in OpenGL 1.5 eingeführte *Occlusion Query* zum Einsatz. Diese liefert die Anzahl Fragmente zurück, die den Tiefentest bei den zwischen einem Start- und einem Stopbefehl erfolgten Rendering-Operationen bestanden haben.

Um diese Funktionalität nutzen zu können, muss erreicht werden, dass nur diejenigen Texel den Tiefentest bestehen, die sich in einer Iteration *verändern*. Vor der ersten Iteration wird daher das Rechteck des Texturobjekts zunächst einmal ohne Shader gezeichnet, um den Tiefenpuffer auf einheitliche Werte zu initialisieren. Der Tiefenpuffer wird anschließend schreibgeschützt. Beim Zeichnen des Rechtecks während der Iterationen wird dann der zuvor beschriebene Fragment Shader ausgewertet. Dieser wurde jedoch so erweitert, dass zusätzlich zum Setzen des jeweiligen RGBA-Werts auch eine Manipulation des Tiefenwerts erfolgt. Ist das Texel bereits gültig, so erfolgt ein Offset des Tiefenwerts um $+z_{off}$. Auch wenn in der ersten Stufe detektiert wird, dass das Texel in dieser Iteration ungültig bleibt ($\alpha_{i+1}^{stage1} = 0$), wird zum Tiefenwert des Fragments $+z_{off}$ addiert. Wird das Texel jedoch in dieser Iteration von ungültig auf gültig gesetzt, so beträgt der Offset $-z_{off}$. Mithilfe des Tiefentests liefert die Occlusion Query so die Anzahl der veränderten

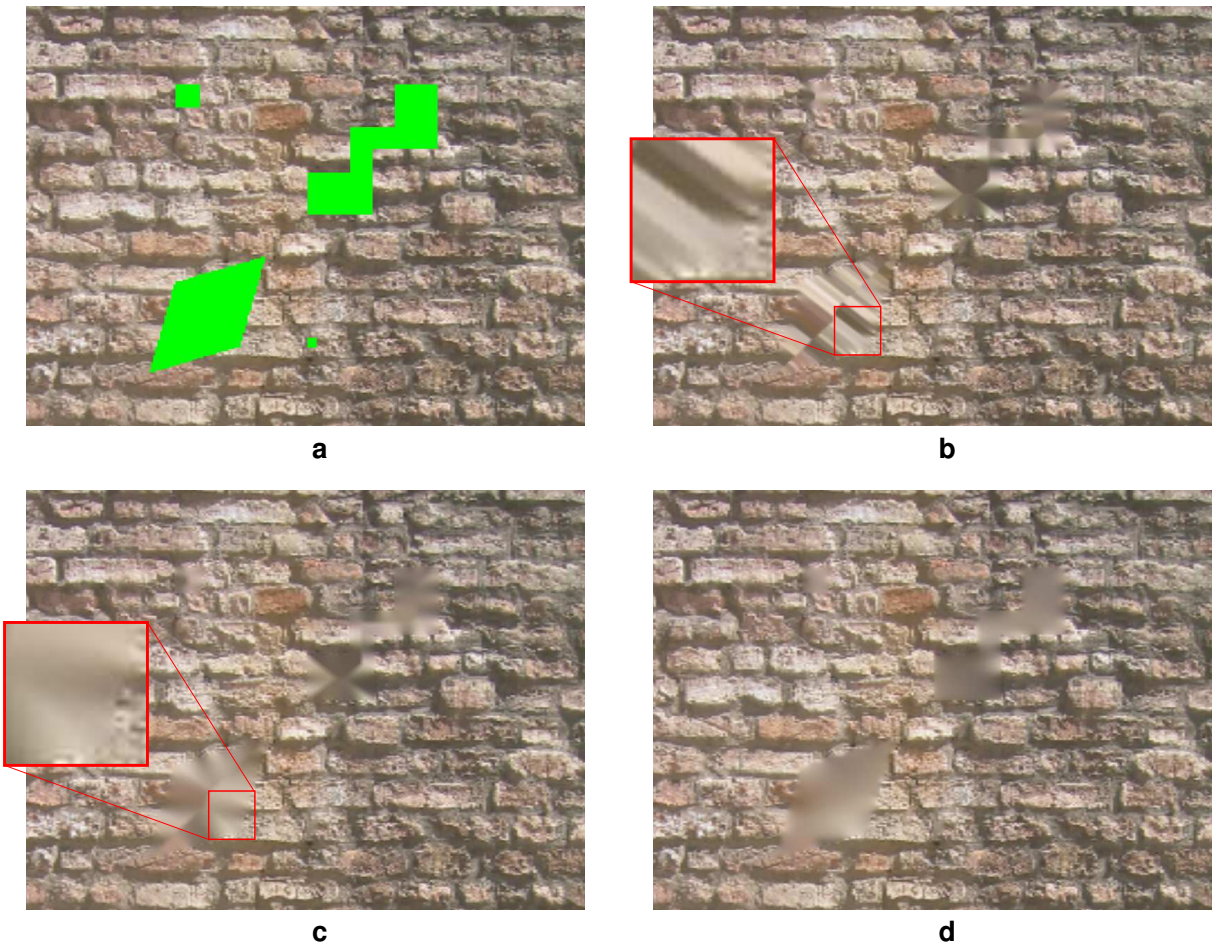


Abbildung 5.23: Hole Filling: a) Textur mit Löchern, b) gefüllt mit Shader, Stufe 1, c) zweistufig gefüllt, d) mit Texturpyramide

Texel. Ist das Ergebnis Null, so sind bereits alle Löcher gefüllt: es sind keine weiteren Iterationen mehr erforderlich.

Durch die daraus resultierende Abhängigkeit der Anzahl der Iterationen von der Größe der Löcher sinkt die erforderliche Rechenzeit für die Textur eines Polygons mit zunehmender vorhandener Farbinformation. In der Initialisierungsphase des Systems, in der erste Texturen für die Polygone extrahiert werden, sind die Ausführungszeiten am größten. Diese Phase ist jedoch noch zeitunkritisch. Bei späteren Aktualisierungen sind dann bereits immer mehr gültige Texel vorhanden, daher können die verbleibenden Löcher immer schneller gefüllt werden.

Vergleich mit pyramiden-basiertem Ansatz

Ein Vorteil dieser Methode gegenüber dem vorgestellten pyramiden-basierten Ansatz ist ihre Unabhängigkeit vom Hersteller der Grafikkarte. Die Verwendung der Register Combiners bei der selektiven Interpolation kann zwar auch durch den Einsatz eines entspre-

chenden Shaders vermieden werden, dadurch wird aber einer der Vorteile dieses Ansatzes, auch auf älterer Hardware lauffähig zu sein, zunichte gemacht.

Ein klarer Vorteil des pyramiden-basierten Ansatzes sind die kürzeren Ausführungszeiten (s. Kap. 7.2.2). Erst sobald nur noch kleine Löcher vorhanden sind, kommt die zweite Methode durch das zuvor beschriebene Verfahren zum vorzeitigen Abbruch des Algorithmus zu vergleichbar kurzen Rechenzeiten. Durch die in das Innere der Löcher zunehmende Tiefpassfilterung sind auch die Farbübergänge beim pyramiden-basierten Ansatz weicher (Abb. 5.23 d). Im Gegensatz dazu sind in der Mitte der gefüllten Löcher bei Einsatz der zweiten Methode teilweise die Übergänge deutlich sichtbar. Dies begründet sich daraus, dass hier nur die unmittelbare Nachbarschaft eines interpolierten Texels zu seinem Farbwert beiträgt. Dadurch treffen in der Mitte der Löcher interpolierte Farbwerte aufeinander, die aus an gegenüberliegenden Rändern befindlichen Farbwerten entstanden sind.

6 Asynchrone Textur-Aktualisierung

6.1 Motivation

In den bisherigen Kapiteln wurde für die Generierung der Texturen von denselben Ressourcen der Grafikkarte Gebrauch gemacht, die auch für die Darstellung der synthetischen Szene für den Benutzer verwendet wurden. Daher war es nicht möglich, beide Vorgänge parallel ablaufen zu lassen.

Als Folge musste für die Dauer der Gewinnung der Texturen aus einem Kamerabild die Darstellung der Szene für den Benutzer unterbrochen werden. Wie in Kapitel 7.2.2 gezeigt, können diese Unterbrechungen, vor allem in Abhängigkeit von der Szene, bis zu mehreren hundert Millisekunden dauern. Dies ist für einen Einsatz des prädiktiven Displays in einer realen Umgebung nicht akzeptabel. Um dem Benutzer den Eindruck eines flüssigen Bildes zu geben, muss die Szene mit mindestens 33 Hz gezeichnet werden [77].

In Kapitel 4.4 wurde bereits erläutert, dass neben geometrischen Änderungen der Szene auch zusätzliche sichtbare Änderungen, wie z. B. in der Beleuchtung, sichtbar gemacht werden müssen. Eine ständige Aktualisierung der Texturen ist somit zwingend erforderlich. In diesem Kapitel wird beschrieben, wie es möglich ist sowohl die Darstellung der künstlichen Szene als auch die Generierung der Texturen parallel auf derselben Grafikkarte ablaufen zu lassen. Für den Benutzer soll dies dabei transparent erfolgen.

6.2 Extraktion der Texturen mit Off-Screen Rendering

Für die Textur-Generierung wird ein Puffer benötigt, in den zu Beginn das Kamerabild kopiert wird, und in dem daraufhin Rendering-Operationen ausgeführt werden können. Dies war bei den bisherigen Beschreibungen der Backbuffer $\mathcal{F}_\mathcal{B}$ des für die Darstellung verwendeten Bereichs des Framebuffers. Der zugehörige Frontbuffer $\mathcal{F}_\mathcal{F}$ wurde zudem für weitere für die Extraktion der Texturen notwendige Rendering-Operationen verwendet. Es wäre zwar möglich alleine $\mathcal{F}_\mathcal{B}$ für die Textur-Gewinnung zu nutzen und für das Zeichnen nur Single Buffering zu verwenden, dies würde jedoch ein störendes Flackern der Darstellung für den Benutzer bedeuten (vgl. Kapitel 2.2).

Um eine flüssige Darstellung der Szene auch während der Textur-Extraktion zu ermöglichen, ist es somit nötig die Textur-Generierung ohne Verwendung des Darstellungs-

Framebuffers durchzuführen. Zum Einsatz kommt dabei das in Kapitel 2.7 beschriebene *Off-Screen Rendering*.

6.2.1 Prinzip

Die Darstellung der Szene und die Textur-Generierung sollen somit hinsichtlich der verwendeten Framebuffer voneinander entkoppelt werden, so dass beide Vorgänge *unabhängig voneinander* ablaufen können. Dazu wird bei den für die Textur-Generierung erforderlichen Render-Operationen anstelle des sichtbaren Framebuffers ein Pixel Buffer verwendet. Auf diese Weise wird eine Darstellung der Szene auch während der Textur-Generierung ermöglicht, da der Backbuffer für die Textur-Generierung nicht mehr benötigt wird und somit allein für die Darstellung der Szene zur Verfügung steht. Die Zustände der beiden Tasks sind durch die zwei Rendering Contexts ebenfalls jeweils gekapselt.

Es wird die in Kapitel 2.7 aufgezeigte Möglichkeit genutzt, Texturobjekte von den betreffenden Rendering Contexts gemeinsam zu nutzen: Im Off-Screen Buffer werden die Texturen erzeugt, im sichtbaren Framebuffer können sie dann für die Darstellung der texturierten Szene genutzt werden. Abbildung 6.1 veranschaulicht diesen Sachverhalt. Während der Rendering Context der Textur-Extraktion $\mathcal{RC}_\mathcal{P}$ somit schreibend auf die Texturen zugreift, werden sie im Rendering Context der Darstellung $\mathcal{RC}_\mathcal{F}$ ausschließlich gelesen.

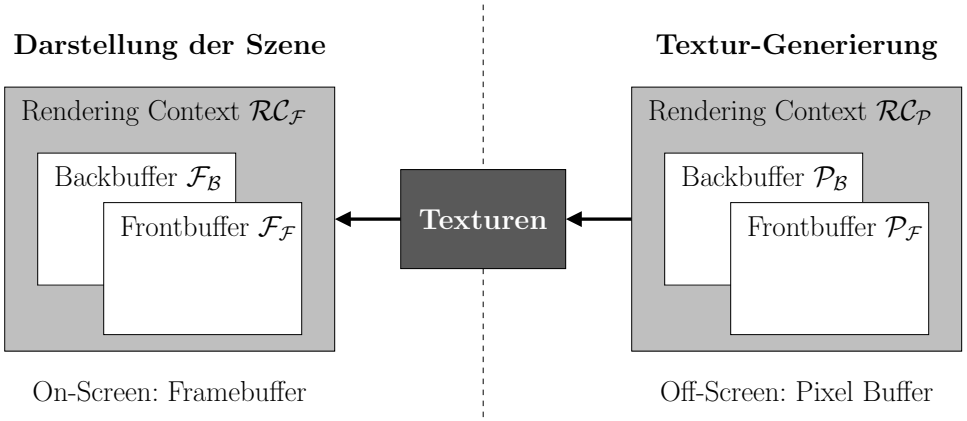


Abbildung 6.1: Trennung zwischen Textur-Generierung und Darstellung der Szene

Dabei sind für den Zeitpunkt ab dem neue Texturen verwendet werden zwei Strategien denkbar. Zum Einen können sie unmittelbar nach ihrer Gewinnung bei der Darstellung zum Einsatz kommen. Da die Extraktion jedoch üblicherweise länger als die Zeit zwischen zwei dargestellten Frames dauert, kann dies zu Unruhe in der Darstellung durch plötzliches Erscheinen neuer Texturen für einzelne Polygone führen. Weiterhin wird so der Vorteil aufgegeben, dass gerade Bereiche mit Texturen, die aus dem gleichen Kamerabild gewonnen wurden, ein stimmiges Gesamtbild ergeben. Daher werden die Texturen bei der Extraktion zunächst gepuffert. Erst wenn alle Texturen eines Kamerabildes extrahiert wurden, werden diese neuen Bildinformationen in die Darstellung übernommen.

6.2.2 Konfiguration des Pixel Buffers

Konfiguration

Die verschiedenen Konfigurationsmöglichkeiten für Pixel Buffer wurden bereits in Kapitel 2.7 erläutert. Die auf den verwendeten NVIDIA-Grafikkarten zur Verfügung stehenden Konfigurationen haben dabei die folgenden Eigenschaften gemeinsam:

- Als *Farbmodus* wird der RGBA-Modus verwendet.
- Die Tiefe des *Color-Buffers* beträgt 8 Bit pro Pixel für jeden Farbkanal (Rot, Grün und Blau).
- Es ist ein *Stencil-Buffer* vorhanden, auf den 8 Bit pro Pixel entfallen, sowie ein *Accumulation-Buffer* mit jeweils 16 Bit pro Pixel für den Rot-, Grün-, Blau- und Alphakanal. Diese beiden Puffer werden hier jedoch nicht benötigt und bleiben daher ungenutzt.

Um die benötigte Funktionalität zur Verfügung zu stellen, muss die Konfiguration des Pixel Buffers zusätzlich folgende speziellen Eigenschaften aufweisen:

- Auch beim Pixel Buffer kommt *Double Buffering* zum Einsatz.
- Die Color-Buffer müssen neben den drei Farbkanälen Rot, Grün und Blau zusätzlich einen *Alphakanal* beinhalten, auf den ebenfalls 8 Bit pro Pixel entfallen.
- Es wird ein *Depth-Buffer* mit einer Tiefe von mindestens 24 Bit pro Pixel benötigt.

Der Alphakanal sowie der Depth-Buffer werden unter anderem für die Verdeckungsrechnung und für die Maßnahmen zur Artefaktunterdrückung benötigt. Auf die Verwendung von Double Buffering wird in Kapitel 6.2.3 eingegangen.

Der Pixel Buffer wird darüber hinaus mit der Option *Preserved Contents* erzeugt, d. h. im Falle eines Ressourcen-Konflikts können die Inhalte des Pixel Buffers nicht verloren gehen, sondern werden gegebenenfalls in den Hauptspeicher ausgelagert.

Größe des Pixel Buffers

Hinsichtlich der Größe des Pixel Buffers (d. h. seiner Höhe und Breite in Pixeln) bestehen folgende Anforderungen:

$$b_P \geq max\{\lceil d_x \rceil_2, r_{max}\} \tag{6.1}$$
$$h_P \geq max\{\lceil d_y \rceil_2, r_{max}\} \tag{6.2}$$

d. h.

1. Höhe und Breite des Pixel Buffers müssen mindestens der Höhe bzw. Breite des Kamerabildes entsprechen und außerdem eine Potenz von Zwei darstellen und
2. die Größe des Pixel Buffers muss mindestens der maximalen Größe der normalisierten Texturen entsprechen.

Die erste Bedingung resultiert zum einen daraus, dass für die Textur-Extraktion das Kamerabild vollständig in den Pixel Buffer geladen werden muss. Weiterhin müssen auch die aus dem Kamerabild extrahierten, perspektivischen Texturen vollständig in den Pixel Buffer gezeichnet werden können[1)]. Die Größe der perspektivischen Texturen ist dabei durch die Größe des Kamerabildes beschränkt, wobei jedoch die Höhe und Breite jeweils zur nächsten Zweierpotenz aufgerundet werden müssen.

Auch die normalisierten Texturen müssen während des Vorgangs der Textur-Generierung mehrfach in den Pixel Buffer gezeichnet werden, sodass daraus die zweite Bedingung für die Größe des Pixel Buffers resultiert. Die maximale Größe der normalisierten Texturen wird dabei als statisch angenommen.

6.2.3 Modifizierter Ablauf der Textur-Generierung

Der Ablauf der Textur-Gewinnung muss angepasst werden, um sie nun unter Zuhilfenahme eines Pixel Buffers anstatt des sichtbaren Framebuffers erfolgen zu lassen. Dadurch ist es nun jederzeit möglich, die Textur-Generierung während ihres Ablaufs für die Darstellung der Szene zu unterbrechen. Erforderlich ist dafür letztlich nur ein Wechsel des aktuellen Rendering Contexts. Die wesentlichen Punkte des dazu modifizierten Ablaufs werden in diesem Kapitel erläutert. Abbildung 6.2 zeigt das zugehörige Flussdiagramm.

Die Textur-Generierung im Pixel Buffer gliedert sich in folgende Schritte, wobei davon ausgegangen wird, dass der Pixel Buffer und der dazugehörige Rendering Context bereits erzeugt worden sind:

1. **Überprüfung der Größe des Pixel Buffers**

   Die Bedingungen, nach denen sich Höhe und Breite des Pixel Buffers richten, wurden bereits in Kapitel 6.2.2 beschrieben. Eine Überprüfung der Größe des Pixel Buffers ist nur dann erforderlich, wenn die Größe der Kamerabilder variabel ist oder wenn die maximale Größe der normalisierten Texturen nicht fest vorgegeben ist. Falls sich die ursprüngliche Größe des Pixel Buffers dann als nicht ausreichend erweist, muss dieser gegebenenfalls gelöscht und mit geänderter Größe neu erzeugt werden.

2. **Wechsel in den zum Pixel Buffer gehörenden Rendering Context**

   Alle weiteren Vorgänge zur Textur-Generierung finden nun im Pixel Buffer statt. Der sichtbare Framebuffer (und auch dessen zugehöriger Rendering Context) bleiben somit von der Textur-Generierung unbeeinflusst.

[1)] Dies ist unter anderem bei der Artefaktunterdrückung erforderlich.

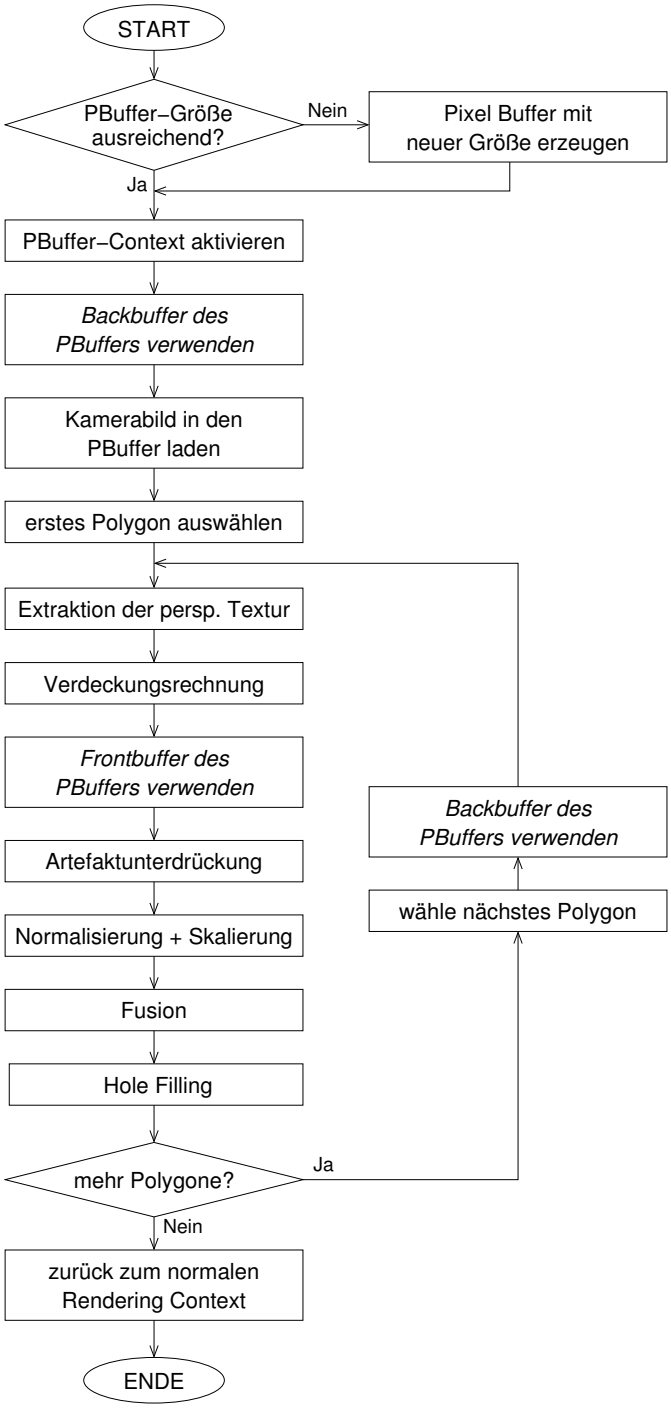


Abbildung 6.2: Ablauf der Textur-Generierung mittels Off-Screen Rendering

3. **Laden des Kamerabildes in den Pixel Buffer**

   Der erste Schritt zur eigentlichen Textur-Generierung besteht darin, das Kamerabild in den Pixel Buffer zu laden. Normalerweise befinden sich die Bilddaten zuvor im Hauptspeicher des Rechners; von dort werden sie dann mit Hilfe des Befehls

`glDrawPixels` in den Pixel Buffer kopiert. Bei diesem Vorgang findet während der Textur-Generierung der einzige Transfer von Bilddaten zwischen dem Hauptspeicher und dem Grafikspeicher statt.

4. **Gewinnung und Bearbeitung der Texturen**

   Hierzu gehören die Extraktion der perspektivischen Texturen, die Verdeckungsrechnung, die Artefaktunterdrückung, die Normalisierung und Skalierung der Texturen, die Fusion mit evtl. bereits vorhandenen Texturen sowie das Füllen der verbliebenen Löcher. Diese Schritte werden für jedes Polygon einzeln ausgeführt. Relevant sind dabei nur diejenigen Polygone, die in dem Ausschnitt des Szenenmodells, der dem Kamerabild entspricht, enthalten sind und für die somit aus dem Kamerabild Textur-Information gewonnen werden kann.

   Nachdem die oben genannten Schritte für ein bestimmtes Polygon durchlaufen sind, wird das fertige Texturbild mittels des Befehls `glCopyTexSubImage2D` aus dem Pixel Buffer in das dem Polygon zugeordnete Texturobjekt kopiert und kann fortan für die Darstellung der Szene verwendet werden. Ein zeitaufwändiger Umweg über den Hauptspeicher ist somit nicht erforderlich.

5. **Wechsel zurück in den Rendering Context des sichtbaren Framebuffers**

   Nach Beendigung der Textur-Generierung kann nun wieder in den Rendering Context des sichtbaren Framebuffers zurückgeschaltet werden.

Die Punkte 3 und 4 unterscheiden sich nicht wesentlich von der Textur-Generierung ohne Pixel Buffer. Lediglich die *Reihenfolge des Ablaufs* wurde geändert und im Gegensatz zur Textur-Generierung im sichtbaren Framebuffer wird beim modifizierten Ablauf mit *Double Buffering* gearbeitet.

Bei der bisherigen Methode erfolgte zuerst für sämtliche Polygone die Extraktion der perspektivischen Texturen und die Verdeckungsrechnung. Anschließend wurden nochmals alle Polygone abgearbeitet, um die weitere Verarbeitung der Texturen durchzuführen. Der Grund dafür ist, dass sich das Kamerabild für die Extraktion der perspektivischen Texturen und die Verdeckungsrechnung im Framebuffer befinden muss. Bei der Durchführung der restlichen Schritte wird das Kamerabild im Framebuffer jedoch überschrieben. Mit der beschriebenen Vorgehensweise wird so vermieden, dass das zeitaufwändige Laden des Kamerabildes für jedes Polygon erneut erfolgen muss. Der Nachteil dabei ist jedoch, dass nach der Extraktion der perspektivischen Textur und der Verdeckungsrechnung die erhaltene perspektivische Textur jedes Polygons zwischengespeichert werden muss, wofür jedem Polygon ein weiteres Texturobjekt zugeordnet werden muss. Dies erhöht erstens den Bedarf an Grafikspeicher, zweitens erweist sich das Erzeugen von Texturobjekten unter gewissen Umständen als sehr zeitaufwändig (siehe dazu Kapitel 2.5.1).

Es ist daher erstrebenswert die Zwischenspeicherung der Texturen zu vermeiden. Dafür bietet es sich bei der Verwendung des Pixel Buffers an, auch hier Double Buffering zu verwenden. Das Kamerabild wird dann zu Beginn in den Backbuffer $\mathcal{P}_\mathcal{B}$ des Pixel Buffers

geschrieben. Dort kann es während der kompletten Textur-Extraktion verbleiben, da Artefaktunterdrückung, Normalisierung, Fusion und Hole Filling stets im Frontbuffer $\mathcal{P}_\mathcal{F}$ des Pixel Buffers ablaufen, während für die Extraktion der perspektivischen Texturen und die Verdeckungsrechnung $\mathcal{P}_\mathcal{B}$ verwendet werden kann. Auf diese Weise können die einzelnen Schritte der Textur-Generierung direkt nacheinander erfolgen, ohne dass eine Zwischenspeicherung von Texturen erforderlich wäre. Die Rollen von Front- und Backbuffer sind hierbei willkürlich gewählt.

6.3 Scheduling

Im vorigen Kapitel wurde beschrieben, wie durch den Einsatz von Off-Screen Rendering die Textur-Generierung und die Darstellung der Szene parallel ablaufen können, ohne sich gegenseitig störend zu beeinflussen. Durch einen Wechsel des aktuellen Rendering Contexts kann die Textur-Generierung praktisch jederzeit unterbrochen werden, um das texturierte Szenenmodell im sichtbaren Framebuffer zu zeichnen.

Um nun eine vom Benutzer als flüssig wahrgenommene Szenendarstellung zu erreichen, ist mindestens eine Bildrate von $F_d = 33$ Frames pro Sekunde nötig. Die Textur-Generierung muss demnach, wie in Abbildung 6.3 angedeutet, alle 30 Millisekunden unterbrochen werden, um ein neues Bild der Szene zu zeichnen. Um dies zu gewährleisten, wird im Programmablauf von Intervall-Timern Gebrauch gemacht.

In diesem Kapitel werden zunächst zwei verschiedene Typen von Intervall-Timern – *QTimer* und *Signal-Timer* – vorgestellt. Im Anschluss daran wird das erarbeitete Scheduling-Konzept, bei dem beide der genannten Intervall-Timer zum Einsatz kommen, erläutert.

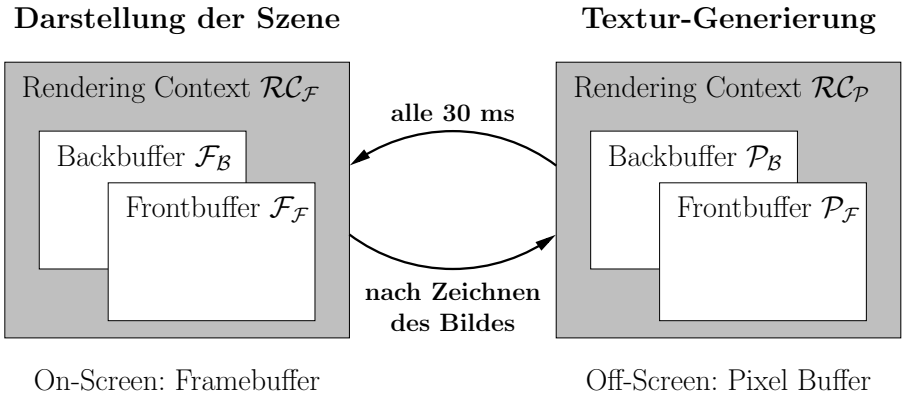


Abbildung 6.3: Scheduling zur Gewährleistung der Framerate

6.3.1 Intervall-Timer

Ein Intervall-Timer dient generell dazu, den Ablauf eines Programms zu gewissen Zeitpunkten zu unterbrechen und eine bestimmte Aktion auszuführen. Dazu wird eine Funk-

tion, der so genannte *Timer-Handler*, aufgerufen, in welchem der Programmcode enthalten ist, der bei der Unterbrechung abgearbeitet werden soll. Nachdem die Ausführung dieser Funktion beendet ist, wird mit dem Programm an derjenigen Stelle fortgefahren, an der es zuvor unterbrochen worden war.

A. QTimer

Die in dieser Arbeit verwendete Qt-Bibliothek [83] bietet neben der Anbindung von OpenGL und den Möglichkeiten zur Erstellung einer grafischen Benutzeroberfläche unter anderem auch einen Intervall-Timer an. Die dazugehörige Klasse trägt den Namen `QTimer`. Alle folgenden Angaben beziehen sich auf die in dieser Arbeit verwendete Version 3.1 der Qt-Bibliothek.

Einrichten und Verwenden des Timers. Die Einrichtung eines QTimers erfolgt durch folgende Schritte:

1. *Erzeugen einer Instanz* der Klasse QTimer
2. *Zuweisung der Handler-Funktion* über den Signal/Slot-Mechanismus (siehe unten)
3. *Starten des Timers* mit dem gewünschten Zeitintervall

Bei dem in Punkt 2 genannten *Signal/Slot-Mechanismus* handelt es sich um ein spezielles Qt-spezifisches Verfahren, bei dem objektübergreifend durch das Aussenden eines so genannten "Signals" eine bestimmte Funktion ("Slot") aufgerufen werden kann. Das Signal wird in diesem Fall durch den QTimer beim Ablauf des Zeitintervalls ausgesendet, beim Slot handelt es sich dementsprechend um die Handler-Funktion des Timers. Für nähere Informationen zum Signal/Slot-Mechanismus sei auf die Dokumentation der Qt-Bibliothek verwiesen. Anzumerken ist an dieser Stelle noch, dass die hier beschriebenen Signale und die später erläuterten Signale des Systemtimers trotz Namensgleichheit nichts miteinander zu tun haben.

Beim Starten des QTimers kann gewählt werden, ob nach Ablauf des ersten Zeitintervalls der Timer erneut gestartet werden soll, oder ob es sich um einen "Single-Shot" handeln soll. Im ersten Fall läuft der Timer so lange weiter, bis er durch einen entsprechenden Befehl explizit beendet wird, im zweiten Fall wird der Timer nach Ablauf des ersten Intervalls automatisch gestoppt.

Funktionsweise. In der Qt-Bibliothek werden zur Steuerung vieler Funktionen *Events* benutzt. Diese werden häufig durch Aktionen des Benutzers ausgelöst, wie etwa durch das Klicken der Maus oder das Drücken einer Taste. Wenn ein solches Event auftritt, wird es zunächst in eine Warteschlange (*Event Queue*) eingereiht. Damit die in dieser Warteschlange gespeicherten Events abgearbeitet werden, muss sich der Programmablauf in der *Main Event Loop* befinden. Dies ist der Zustand, in dem in einer Endlosschleife auf Events, wie Aktionen des Benutzers, gewartet wird. Befindet sich der Prozess dagegen in

einer anderen Funktion, werden die Events nur auf einen expliziten Befehl hin abgearbeitet (`QApplication::processEvents()`).

Auch durch den QTimer wird beim Ablaufen des entsprechenden Zeitintervalls ein Event generiert (*Timer-Event*). Bei der Abarbeitung dieses Timer-Events wird dann die dem Timer zugewiesene Handler-Funktion aufgerufen. Hieraus ergibt sich das Problem, dass sich der Programmablauf für die zeitgerechte Ausführung des Handlers in der oben angesprochenen Main Event Loop befinden muss. Solange dies nicht der Fall ist, bleibt der Timer blockiert. Abbildung 6.4 zeigt dies an einem Beispiel. Zwar kann die Abarbeitung der Event-Queue explizit veranlasst werden, um dabei jedoch die Genauigkeit des Timers zu bewahren, muss dies in Zeitabständen geschehen, die deutlich kleiner sind als das Zeitintervall des Timers.

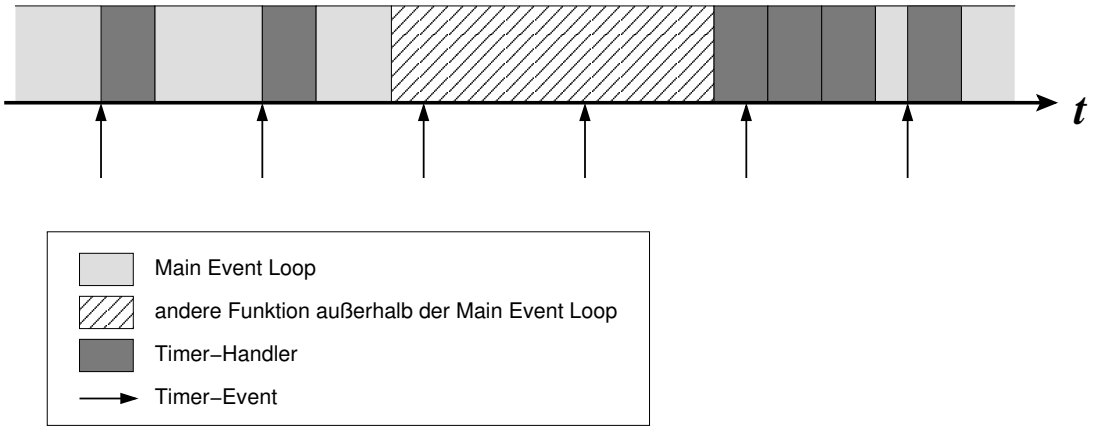


Abbildung 6.4: Funktionsweise des QTimers an einem Beispiel

Ohne regelmäßiges Aufrufen des Befehls `QApplication::processEvents()` bewirkt der QTimer daher nur dann eine zeitgerechte Unterbrechung des Programms, wenn sich dieses in der Main Event Loop befindet. Gerade während der Textur-Generierung ist dies jedoch nicht der Fall, so dass hierfür eine andere Lösung gefunden werden muss.

B. Systemtimer unter Linux

Als Alternative zum QTimer bietet auch das Betriebssystem Linux einen Intervall-Timer an. Dieser beruht nicht auf dem im vorigen Kapitel vorgestellten Event-Mechanismus, sondern bedient sich so genannter *Signale*. Im Folgenden wird zunächst dieses Signal-Konzept kurz erläutert. Nähere Informationen dazu finden sich zum Beispiel in [38].

Das Signal-Konzept unter Linux. Unter Signalen versteht man Interrupts, die entweder von der Hardware oder von der Software erzeugt werden, wenn während der Programmausführung besondere Ereignisse eintreten. Ein Linux-Timer zeigt so mittels eines Signals den Ablauf eines eingestellten Zeitintervalls an.

Wenn ein Prozess ein solches Signal empfängt, gibt es drei Möglichkeiten, wie darauf reagiert werden kann:

1. Das Signal kann *ignoriert* werden, d. h. es erfolgt keine weitere Reaktion.
2. Es kann die vom System voreingestellte Aktion (*Default-Aktion*) ausgeführt werden. In den meisten Fällen, insbesondere bei durch Fehlersituationen ausgelösten Signalen, führt dies zur Beendigung des betreffenden Prozesses.
3. Es kann ein *durch den Benutzer definierter Signal-Handler* aufgerufen werden. Bei einem Signal-Handler handelt es sich um eine Callback-Funktion: Dem System wird die Adresse dieser Funktion übergeben, wodurch diese beim Eintreffen eines Signals automatisch ausgeführt werden kann.

Es ist auch möglich, Signale zu *blockieren*. Das Blockieren hat dabei zur Folge, dass im Falle des Eintreffens des betreffenden Signals die voreingestellte Aktion (d. h. die Default-Aktion oder der Aufruf des benutzerdefinierten Signal-Handlers) solange nicht ausgeführt wird, bis die Blockierung für dieses Signal wieder aufgehoben wird oder bis das Signal explizit ignoriert wird.

Ein Signal ist grundsätzlich immer blockiert während der zugehörige Signal-Handler (oder die Default-Aktion) ausgeführt wird. Demnach kann ein Signal-Handler nicht mehrfach ausgeführt werden.

Treffen während der Blockierung eines bestimmten Signals mehrere Signale dieser Art ein, so bleibt nur das erste Signal in Wartestellung, wohingegen die nachfolgenden Signale verworfen werden. Für Signale existiert somit keine Warteschlange, wie dies bei den Events der Qt-Bibliothek der Fall ist.

Einrichten und Verwenden des Timers. Um einen Linux-Systemtimer einzurichten und zu starten, sind folgende Schritte erforderlich:

1. **Zuweisen der entsprechenden Handler-Funktion**

   Das beim Ablauf eines Timer-Intervalls ausgelöste Signal trägt den Namen "`SIGALRM`" (in Anlehnung an den Begriff "alarm clock"). Diesem Signal kann mit dem Befehl `sigaction()` (siehe [38]) die entsprechende Handler-Funktion zugewiesen werden, die durch den Timer aufgerufen werden soll.

2. **Spezifizieren der Dauer der Zeitintervalle**

   Im Gegensatz zu QTimern besteht bei einem Linux-Systemtimer die Möglichkeit, für das erste Zeitintervall nach dem Starten des Timers sowie für die folgenden Zeitintervalle unterschiedliche Werte anzugeben. Zur Aufnahme dieser Werte dient eine Struktur des Typs `struct itimerval`.

   Die Dauer der Zeitintervalle wird dabei in Mikrosekunden angegeben. Tatsächlich bietet der Timer jedoch nur eine Auflösung von 10 Millisekunden, d. h. die angege-

benen Werte werden gegebenenfalls intern auf ein Vielfaches von 10 Millisekunden aufgerundet. Dies entspricht der üblichen Auflösung des Linux-Schedulers [5].

3. **Starten des Timers**

   Zum Starten des Systemtimers bietet Linux die Funktion `setitimer()` an, an welche die in Punkt 2 angelegte Struktur übergeben wird. Derselbe Befehl kann auch zum Stoppen des Timers verwendet werden, indem die übergebenen Zeitintervalle auf Null gesetzt werden.

   Die Möglichkeit eines "Single-Shots", d. h. das automatische Stoppen des Timers nach Ablauf des ersten Intervalls, ist bei diesem Timer-Typ nicht möglich. Um diesen Effekt dennoch zu erreichen, muss der Timer beim ersten Aufruf des Signal-Handlers manuell gestoppt werden.

Funktionsweise. Wie bereits weiter oben erläutert wurde, führt das Ablaufen des Timers zum Senden des Signals SIGALRM, worauf der aktuelle Prozess unverzüglich unterbrochen und die Handler-Routine des Timers aufgerufen wird. Falls das Signal jedoch gerade blockiert ist, bleibt es bis zum Aufheben der Blockierung in Wartestellung. Weitere Signale, die während der Blockierung eintreffen, werden verworfen. In Abbildung 6.5 ist dieser Sachverhalt in einem Diagramm dargestellt.

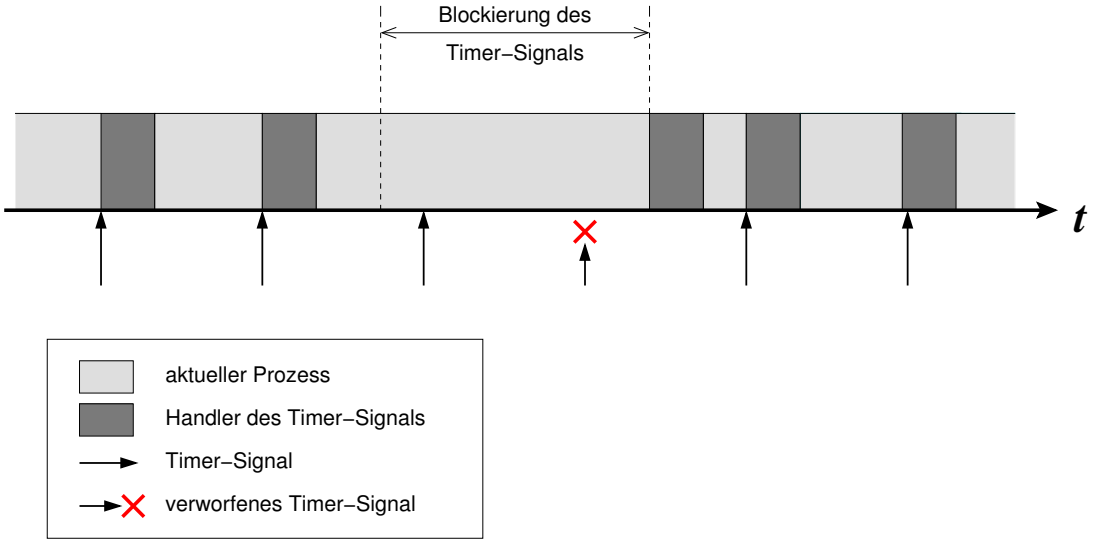


Abbildung 6.5: Funktionsweise des Systemtimers an einem Beispiel

Einschränkung bezüglich der im Signal-Handler aufgerufenen Funktionen. Durch die Möglichkeit des Linux-Systemtimers, einen Prozess jederzeit asynchron zu unterbrechen, können somit auch Befehle während ihrer Abarbeitung unterbrochen werden. Dies führt zu der Einschränkung, dass alle Befehle, die bei der Texturgewinnung aufgerufen werden, *reentrant*, d. h. mehrfach aufrufbar sein sollten. Falls dies nicht gewährleistet ist, kann

es zu Fehlern in der Programmausführung kommen, wenn das Timer-Signal eine nicht-reentrante Funktion unterbricht und diese Funktion durch den Handler erneut gestartet wird.

In dieser Hinsicht kann es auch zu Problemen kommen, wenn im Timer-Handler OpenGL-Befehle aufgerufen werden, da die meisten dieser Funktionen in der in dieser Arbeit verwendeten OpenGL-Implementierung nicht reentrant sind. In diesem Fall sollte daher sichergestellt sein, dass Timer-Signale nicht während der Abarbeitung von OpenGL-Befehlen auftreten. Dies kann zum Beispiel durch das Blockieren des Signals erreicht werden kann.

6.3.2 Gewährleistung der Framerate

Im Folgenden wird das in dieser Arbeit entwickelte Scheduling-Konzept erläutert. Das Ziel ist es, die Darstellung mit möglichst konstanter Framerate erfolgen zu lassen. Dies ist jedoch eine *weiche Realzeit-Bedingung*, da ein Abfallen der Framerate zwar den Benutzer beeinträchtigt, jedoch kein sicherheitskritisches Problem darstellt. Die verbleibende Rechenzeit auf CPU und GPU soll für die Textur-Aktualisierung zur Verfügung stehen. Die Darstellung hat somit eine höhere Priorität als die Textur-Gewinnung. Dies ist notwendig, um dem Benutzer eine möglichst flüssige Darstellung der Szene zu präsentieren. Da kein Realzeit-Betriebssystem zum Einsatz kommt, kann eine stabile Framerate natürlich nicht wirklich garantiert werden. Die hier beschriebenen Verfahren zielen darauf ab sie aber trotzdem möglichst stabil zu halten.

Bei der Umsetzung kommen sowohl der QTimer als auch der Signal-Timer zum Einsatz. Beide rufen bei ihrem Ablauf letztlich die gleiche Darstellungsfunktion auf.

Anwendung des QTimers

Der QTimer wird immer dann verwendet, wenn gerade *keine Textur-Generierung* stattfindet, da sich in dieser Zeit der Programmablauf in der Main Event Loop (siehe Kapitel 6.3.1) befindet und daher auch durch den QTimer eine stabile Framerate (ohne manuelles Aufrufen von `QApplication::processEvents()`) erreicht wird. Ein Einsatz des Systemtimers ist hier ungeeignet, da während der Abarbeitung der Main Event Loop auch nicht-reentrante Funktionen aufgerufen werden, deren Unterbrechung durch den Systemtimer dann zu Fehlfunktionen führen könnte.

Die Aufgabe der Handler-Funktion des QTimers besteht hier lediglich darin, das Zeichnen eines neuen Frames zu veranlassen. Weitere Schritte, wie sie in der Handler-Funktion des Systemtimers (siehe folgender Abschnitt) erforderlich sind, werden hier nicht benötigt.

Anwendung des Systemtimers

Um auch während der Textur-Generierung den QTimer einsetzen zu können, wäre ein periodisches Aufrufen des bereits angesprochenen Befehls `QApplication::processEvents()` für eine Abarbeitung der Timer-Events erforderlich. Zur Erzielung einer halbwegs guten Genauigkeit des Timers müsste der Aufruf dieses Befehls in Zeitabständen erfolgen, die deutlich geringer sind als die Dauer eines Timer-Intervalls. Dies wäre jedoch nur möglich, wenn auch alle einzelnen Befehle der Textur-Generierung innerhalb solch kurzer Zeit abgearbeitet werden könnten. Ein Beispiel, bei dem dies nicht gewährleistet ist, ist das Laden eines Kamerabildes von der Festplatte oder über das Netzwerk. Dabei wurden auf dem verwendeten System bis zu 60 Millisekunden gemessen.

Während der Textur-Generierung wird daher statt des QTimers der Linux-Systemtimer genutzt. Durch die Verwendung des Signal-Konzepts ist der Linux-Systemtimer in der Lage, eine Unterbrechung der Textur-Generierung auch während der Abarbeitung länger dauernder Befehle hervorzurufen. Die Handler-Funktion stellt sich hier jedoch etwas komplizierter dar, als dies beim QTimer der Fall ist, da vor dem Zeichnen eines neuen Bildes noch einige andere Schritte nötig sind.

Im Folgenden werden die einzelnen Vorgänge im Handler des Systemtimers erläutert. Ein zugehöriges Flussdiagramm ist in Abbildung 6.6 dargestellt. Die eingekreisten Ziffern neben den Elementen des Diagramms entsprechen der Nummerierung der folgenden Absätze.

1. **Überlastbehandlung**

   Die für das Zeichnen des texturierten Szenenausschnitts benötigte Zeit hängt neben anderen Faktoren stark von der Anzahl der texturierten Polygone ab, die in diesem Ausschnitt enthalten sind. Bei einer komplexen Szene kann es dabei passieren, dass die gewünschte Framerate nicht erreicht werden kann, d. h. die Abarbeitung der gesamten Handler-Funktion dauert länger als das Zeitintervall des Timers. In diesem Fall wird das nächste Timer-Signal bereits ausgelöst, *bevor* die aktuelle Handler-Funktion beendet ist. Dadurch wird diese im Anschluss daran sofort erneut aufgerufen. Dies kann schließlich zur Folge haben, dass für die Textur-Generierung keine Zeit mehr zur Verfügung steht und stattdessen nur noch die Zeichen-Routine durchlaufen wird.

   Um eine vollständige Blockierung der Texturgewinnung zu vermeiden, wird am Beginn der Handler-Funktion die Zeitdifferenz seit dem letzten Verlassen des Handlers gemessen. Unterschreitet diese Zeit einen bestimmten Schwellwert t_{diff}, so wird angenommen, dass in der Zwischenzeit die Textur-Generierung keine Fortschritte gemacht hat. Statt die Szene zu zeichnen, wird in diesem Fall die Handler-Funktion sofort wieder verlassen. Zuvor wird jedoch der Systemtimer neu gestartet, wobei die Dauer des ersten Intervalls auf t_{min} gesetzt wird. Diese Zeit steht somit für die Textur-Generierung zur Verfügung, bevor das nächste Timer-Signal ausgelöst wird.

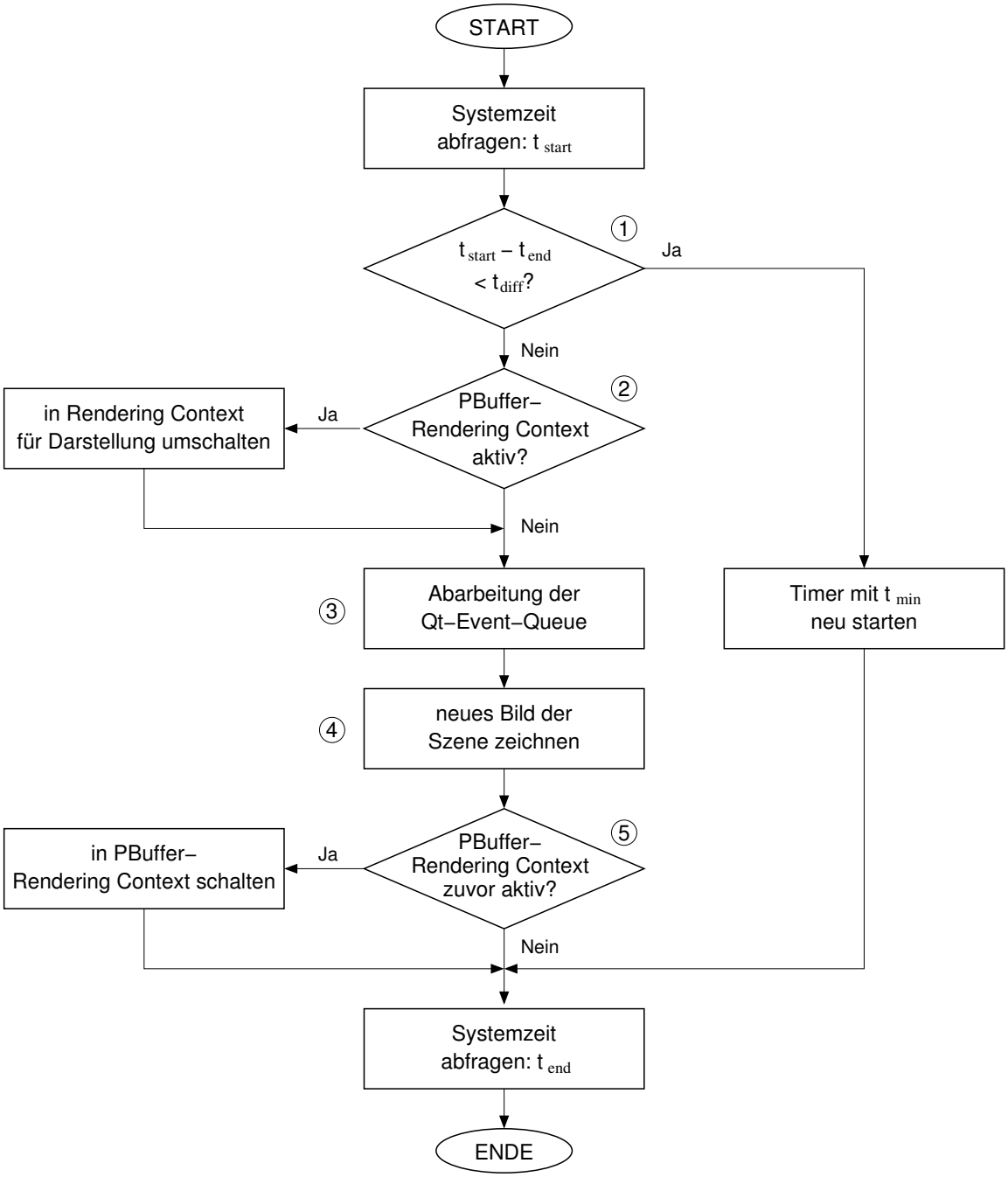


Abbildung 6.6: Handler-Routine des Systemtimers

Für t_{diff} haben sich in Versuchen 100 Mikrosekunden als brauchbarer Wert erwiesen. Für t_{min} wurde der unter Linux kleinste mögliche Timer-Wert 10 Millisekunden verwendet.

Die Framerate wird durch diese Methode natürlich weiter gesenkt. Dies ist jedoch akzeptabel, da das Verfahren ohnehin nur zum Einsatz kommt, wenn die aktuelle Framerate F_a bereits unter die gewünschte Framerate F_d gefallen ist.

2. **Wechsel in den Rendering Context des sichtbaren Framebuffers**

   Im nächsten Schritt wird überprüft, welcher Rendering Context gerade aktiv ist. Ist dies der Rendering Context des Pixel Buffers, so wird für die Darstellung der Szene in den Rendering Context des sichtbaren Framebuffers umgeschaltet.

3. **Abarbeitung der Event-Queue**

   Als nächstes wird der Befehl `QApplication::processEvents()` ausgeführt, um eventuell eingetroffene Qt-Events abzuarbeiten. Dies ist nötig, um auch während der Textur-Generierung auf Aktionen des Benutzers (Mausklicks, Tastatureingaben etc.) reagieren zu können.

   Vor dem Eintritt in die Textur-Generierung muss deshalb auch der Event-Timer gestoppt werden, da dessen Timer-Events sonst ebenfalls berücksichtigt und in Konflikt mit dem hier verwendeten System-Timer geraten würden.

4. **Zeichnen des neuen Bildes**

   Der vierte Schritt besteht schließlich darin, das neue Bild der Szene zu zeichnen.

5. **Wechsel zurück in den Rendering Context des Pixel Buffers**

   Vor der Rückkehr aus der Handler-Funktion wird nun wieder der Rendering Context des Pixel Buffers aktiviert, falls dieser zuvor aktiv war.

Blockieren des Timers während der Ausführung von nicht-reentranten Funktionen

Wie bereits in Kapitel 6.3.1 erwähnt, muss die Unterbrechung der Ausführung einer nicht-reentranten Funktion durch Timer-Signale vermieden werden, falls diese Funktion auch im Signal-Handler aufgerufen wird.

Auch die meisten OpenGL-Funktionen sind bei den verwendeten Treibern von dieser Einschränkung betroffen. Deswegen ist es erforderlich, während der Ausführung von OpenGL-Funktionen, die im Laufe der Textur-Generierung aufgerufen werden, das Timer-Signal zu blockieren. Nur so können Konflikte zwischen einer durch das Timer-Signal unterbrochenen und einer im Timer-Handler aufgerufenen OpenGL-Funktion vermieden werden.

Eine Blockierung des Timers kann natürlich den Aufruf des Timer-Handlers und damit die zeitgerechte Darstellung des nächsten Frames verzögern. Von Bedeutung ist dies vor allem dann, wenn die Abarbeitung eines einzelnen OpenGL-Befehls so lange dauert, dass die Blockierungsdauer des Timers nicht mehr deutlich geringer ist als die Dauer eines Timer-Intervalls. Solche Fälle sind jedoch relativ selten und treten vor allem dann auf, wenn nur noch wenig verfügbarer Grafikspeicher vorhanden ist und Texturdaten deshalb in den Hauptspeicher ausgelagert werden müssen (vgl. Kapitel 2.5.1). Im Normalfall hingegen wirkt sich die Blockierung des Timers nur unwesentlich auf die Stabilität der Framerate aus.

6.4 Zeitkritische Grafik-Operationen

Durch das in Kapitel 6.3 vorgestellte Scheduling-Konzept ist es möglich das Szenenmodell auch während der Textur-Generierung mit einer nahezu konstanten Bildwiederholungsrate darzustellen. Um für den Bediener eines Telepräsenz-Systems eine möglichst große Realitätsnähe zu erreichen, sollte diese Framerate dabei so hoch wie möglich sein. Zwar reicht für eine als flüssig wahrgenommene Szenendarstellung eine Rate von 33 Bildern pro Sekunde aus. Von den Zeitabständen zwischen den einzelnen Bildern ist es jedoch auch abhängig, mit welcher Verzögerung der menschliche Operator eine visuelle Rückmeldung seiner Aktionen erhält. Insbesondere bei einer raschen Bewegung des Kopfes kann sich eine Verzögerung von 30 Millisekunden, wie sie bei einer Rate von 33 Bildern pro Sekunde auftreten kann, bereits als störend erweisen.

Mit steigenden Anforderungen bezüglich der Framerate gewinnen auch Fragen nach der Ausführungsdauer verschiedener Grafik-Operationen an Bedeutung. Je kürzer die Zeitabstände zwischen den einzelnen Frames sind, desto weniger Zeit steht letztlich für das Zeichnen eines Frames zur Verfügung. Dasselbe gilt für die Textur-Generierung, da diese in der verbleibenden Zeit zwischen dem Zeichnen der Frames stattfindet. Durch das in Kapitel 6.3.1 angesprochene Problem, dass der System-Timer während der Ausführung von OpenGL-Befehlen blockiert werden muss, erhöhen sich aber insbesondere die Anforderungen an die Ausführungszeiten dieser Befehle: Um die Schwankungen der Framerate durch die Blockierung des Timers zu minimieren, muss die Blockierungszeit deutlich unter der Dauer eines Timer-Intervalls, d. h. dem Zeitabstand zweier Frames, bleiben.

Dieses Kapitel geht daher auf diejenigen Grafik-Operationen ein, deren Zeitbedarf sich in dieser Hinsicht als besonders kritisch herausgestellt hat. Dies betrifft insbesondere die Erzeugung und Verwendung von Texturobjekten, den Transfer von Daten zwischen dem Hauptspeicher und dem Grafikspeicher, sowie das Umschalten zwischen verschiedenen Rendering Contexts.

6.4.1 Texturobjekte

Auslagerung von Texturobjekten in den Hauptspeicher

In Kapitel 2.5.1 wurde bereits erwähnt, dass OpenGL zur Speicherung von Texturen *Texturobjekte* verwendet. Ein Texturobjekt beinhaltet neben diversen Parameterdaten vor allem die Bilddaten einer Textur. Es benötigt daher eine bestimmte Menge an Speicherplatz, die dabei hauptsächlich von der Größe der Textur und vom internen Speicherformat der Bilddaten (d. h. Anzahl der Bits pro Texel) abhängt. Eine Textur mit 1024×1024 Texeln, die im RGBA-Format mit jeweils 32 Bit pro Texel gespeichert ist, benötigt beispielsweise 4 MB an Speicher.

Für die Speicherung von Texturobjekten wird normalerweise der lokale Speicher der Grafikkarte verwendet, wodurch ein schneller Zugriff des Grafikprozessors auf die Texturdaten

ermöglicht wird. Reicht der Grafikspeicher nicht für die Speicherung aller vorhandenen Texturobjekte aus, können diese jedoch auch in den Hauptspeicher ausgelagert werden. Dabei wird gewöhnlich eine *LRU*-Strategie (*Least-Recently Used*) verwendet, d. h. diejenigen Texturobjekte, die am längsten nicht mehr verwendet wurden, werden als erstes ausgelagert. Alternativ können Texturobjekten auch verschiedene Prioritäten zugewiesen werden, wodurch zuerst die Texturobjekte ausgelagert werden, denen die niedrigste Priorität zugewiesen wurde.

Problematisch bei der Auslagerung von Texturobjekten ist, dass sowohl das Auslagern selbst als auch der Zugriff auf ausgelagerte Texturobjekte sehr zeitaufwändig ist. Der genaue Zeitbedarf lässt sich hierbei nicht vorhersagen, je nach Anzahl und Größe der Texturen kann sich der benötigte Zeitbedarf in beiden Fällen aber bis in den Sekundenbereich bewegen. Für das beschriebene System bedeutet dies, dass sowohl die Darstellung der Szene als auch die Textur-Generierung dadurch für mehrere Sekunden blockiert werden können.

Aus diesem Grund ist es zweckmäßig, den Speicherbedarf der Texturen von vornherein so zu dimensionieren, dass alle Texturobjekte sicher im lokalen Speicher der Grafikkarte Platz finden. Um dies zu erreichen, kann zum Beispiel die maximale Größe der normalisierten Texturen herabgesetzt werden. Dies hat jedoch auch zur Folge, dass die betreffenden Texturen mit einer geringeren Auflösung dargestellt werden, wodurch die Bildqualität für den Betrachter sinkt. Eine andere Möglichkeit ist, die Texturen komprimiert abzuspeichern, worauf im Folgenden noch ausführlicher eingegangen wird. Auch hierbei verringert sich jedoch die Qualität der dargestellten Texturen.

Anlegen neuer Texturobjekte

Auch das Erzeugen neuer Texturobjekte zählt zu den Vorgängen, deren Zeitbedarf nicht ohne weiteres vorhergesagt werden kann. Zwar beträgt im Normalfall die dafür benötigte Zeit auch bei großen Texturen nicht mehr als wenige Millisekunden. Vor allem in zwei Fällen kann der Zeitbedarf für das Erzeugen eines neues Texturobjekts jedoch erheblich steigen:

- Falls der zur Verfügung stehende *Grafikspeicher für das Anlegen des neuen Texturobjekts nicht ausreicht*, kann es infolgedessen zum Auslagern anderer Texturobjekte kommen, was dann zu den im vorigen Abschnitt beschriebenen Zeitverlusten führt.
- Aber auch durch eine *starke Fragmentierung* des Grafikspeichers, die zum Beispiel durch ein häufiges Löschen und Neuerzeugen von Texturobjekten verursacht werden kann, können erhebliche Performance-Einbußen beim Anlegen von Texturobjekten entstehen, selbst wenn der insgesamt noch verfügbare Grafikspeicher für die neue Textur ausreicht.

Beide Fälle können letztlich dazu führen, dass die benötigte Zeit für das Erzeugen eines neuen Texturobjekts die Dauer eines Timer-Intervalls um ein Vielfaches übersteigt. Neben der Einhaltung des verfügbaren Grafikspeichers ist es daher wichtig, auch eine

Fragmentierung des Grafikspeichers zu vermeiden, indem das Löschen und Neuerzeugen von Texturobjekten auf ein Minimum beschränkt wird.

Es wurde daher die Strategie verfolgt für jedes Polygon nur ein einziges Mal ein Texturobjekt anzulegen: wenn das betreffende Polygon erstmals die Textur-Generierung durchläuft. Dieses Texturobjekt bleibt dem Polygon anschließend fest zugeordnet. Wenn das Polygon dann erneut die Textur-Generierung durchläuft, werden zwar die Bilddaten der Textur aktualisiert, es muss aber kein neues Texturobjekt erzeugt werden. Auf diese Weise kann auch auf das Löschen von Texturobjekten vollständig verzichtet werden, wodurch Fragmentierung vermieden wird. Das Texturobjekt für die temporäre perspektivische Textur wird ebenfalls in ausreichender Größe einmal zu Beginn angelegt. Dieses Objekt wird für die perspektivische Textur *aller* Polygone verwendet.

Textur-Komprimierung

Wie in Kapitel 2.5.2 beschrieben, ist S3TC ein *asymmetrisches Kompressionsverfahren*. Für Anwendungen deren Texturen statisch sind ist dies unproblematisch. Sie können vorab komprimiert und in dieser Form auf die Grafikkarte geladen werden. Durch die reduzierte Datenmenge läuft dies dabei sogar noch schneller ab, als mit unkomprimierten Texturen. Jede Manipulation der Texturen (wie z. B. das Kopieren von Bilddaten in ein Texturobjekt) zur Laufzeit erfordert jedoch eine erneute Komprimierung der entsprechenden Daten. Dadurch erhöht sich die Dauer der Operation auf ein Vielfaches. Durch die in dieser Arbeit beschriebene große Anzahl an Manipulationen an den Texturen ist der entstehende Zeitbedarf nicht akzeptabel.

Die Komprimierung kann aber trotz des hohen Zeitbedarfs auch bei dynamischer Generierung der Texturen zur Laufzeit Vorteile bringen. Wenn der zur Verfügung stehende Grafikspeicher nicht ausreicht, um sämtliche Texturobjekte unkomprimiert darin unterzubringen, kann der Zeitbedarf der Textur-Komprimierung den einer Auslagerung von Texturen in den Hauptspeicher aufwiegen. Zudem lässt sich die Zeit, die zur Textur-Komprimierung benötigt wird im Gegensatz zur Dauer der Auslagerung von Texturen zumindest grob vorhersagen.

6.4.2 Datentransfer zwischen Haupt- und Grafikspeicher

Eine besonders zeitaufwändige Operation im Bereich der Computergrafik ist der Transfer von Daten zwischen dem Hauptspeicher und dem lokalen Speicher der Grafikkarte. Dieser Abschnitt beschäftigt sich deshalb mit zwei Fällen, in denen der schnelle Datentransfer vom Haupt- in den Grafikspeicher von Bedeutung ist.

Bilddaten

Vor der Gewinnung der Texturen aus einem Kamerabild muss dieses, wie in den Kapiteln 4.1.1 und 6.2.3 beschrieben, zunächst in den Pixel Buffer geladen werden. Dieser Vorgang kann jedoch, in Abhängigkeit von der Größe des Kamerabildes, so lange dauern, dass die Genauigkeit des System-Timers dadurch merklich beeinflusst wird, da auch während der Ausführung von `glDrawPixels`, wie in Kapitel 6.3.1 erläutert, eine Blockierung des System-Timers erforderlich ist.

Die Lösung ist denkbar einfach: Anstatt das Kamerabild in einem Zug zu kopieren, wird dieser Vorgang in mehrere Schritte aufgespaltet. Es wird dann immer nur ein Teil des Kamerabildes kopiert und anschließend die Blockierung des Timers kurzzeitig aufgehoben, um ein eventuell anstehendes Timer-Signal abzuarbeiten. Nach der erneuten Blockierung des Timers kann der nächste Teil des Kamerabildes in den Grafikspeicher geschrieben werden.

Abbildung 6.7 veranschaulicht diese Vorgehensweise, wobei hier als Beispiel eine Aufteilung des Kopiervorgangs in zwei Schritte erfolgt.

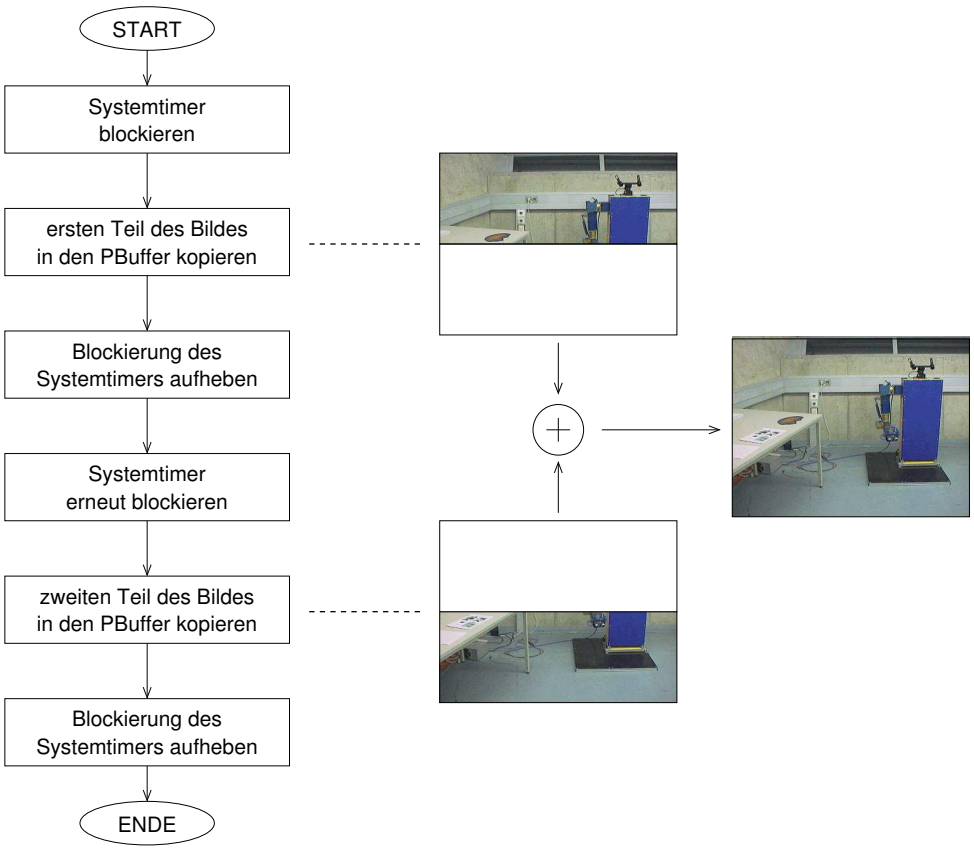


Abbildung 6.7: Laden eines Kamerabildes in mehreren Schritten

Vertex-Daten

Für die Darstellung der texturierten Szene müssen für jeden Frame eine Reihe von Daten pro Vertex vom Hauptspeicher zum Grafikprozessor übertragen werden. Dabei handelt es sich zumindest um die Koordinaten der Eckpunkte jedes Polygons, häufig zusätzlich noch um die dazugehörigen Normalenvektoren, die von OpenGL für eine korrekte Berechnung der Beleuchtung des Szenenmodells benötigt werden. Die Daten sämtlicher Polygone werden für gewöhnlich in einem so genannten *Vertex-Array* im Hauptspeicher abgelegt. Der Zugriff auf die Daten eines bestimmten Polygons erfolgt dann durch Auswahl des entsprechenden Eintrags dieses Vertex-Arrays. Von der Geschwindigkeit dieses Datenzugriffs ist es unter anderem abhängig, wie viel Zeit das Zeichnen eines Frames in Anspruch nimmt.

Der entscheidende Vorteil bei der Verwendung von Vertex-Arrays liegt darin, dass von der CPU zunächst nur die betreffenden Indizes des Vertex-Arrays an den Grafikprozessor geliefert werden müssen. Die eigentlichen Daten können dann vom Grafikprozessor über DMA (Direct Memory Access) aus dem Hauptspeicher geladen werden. Die Verwendung von AGP oder PCIe bringt dabei zusätzliche Geschwindigkeitsvorteile. Seit OpenGL 1.5 ist es mit den *Vertex Buffer Objects* auch möglich Vertex-Daten direkt im lokalen Grafikspeicher abzulegen. Dadurch kann der Transfer zwischen Hauptspeicher und Grafikspeicher für diese Daten minimiert werden. Mit der schnellen Anbindung des Grafikspeichers an die GPU können hohe Frameraten erreicht werden.

6.4.3 Wechsel des Rendering Contexts

Immer wenn der Vorgang der Textur-Generierung zum Zeichnen eines neuen Bildes durch ein Timer-Signal unterbrochen wird, muss zunächst in den Rendering Context des sichtbaren Framebuffers umgeschaltet werden. Vor der Wiederaufnahme der Textur-Generierung erfolgt ein Wechsel zurück in den Rendering Context des Pixel Buffers (siehe Kapitel 6.3.2).

Diese Umschaltvorgänge erweisen sich teilweise als sehr zeitaufwändig. Der Zeitbedarf für einen Wechsel des aktuellen Contexts bewegt sich beim verwendeten System etwa in der Größenordnung von einer Millisekunde. In einzelnen Fällen kann der Zeitbedarf jedoch die Dauer eines Zeitintervalls zwischen zwei Frames deutlich übersteigen, was dann zu einem Einbruch der Framerate bei der Szenendarstellung führt. Der Grund dafür ist eine implizite Ausführung des Befehls `glFinish` vor dem Wechsel des Rendering Contexts. Durch diesen Befehl wird der Programmablauf so lange angehalten, bis sämtliche Grafikoperationen, die bis zu diesem Zeitpunkt durch entsprechende OpenGL-Befehle gestartet worden sind, vollständig abgearbeitet sind, d. h. die Pipeline wird geleert.

Der Zeitbedarf für den Wechsel des Rendering Contexts kann in akzeptablen Grenzen gehalten werden, indem gelegentlich während des Programmablaufs der Befehl `glFinish` explizit aufgerufen wird. Diese Aufrufe verursachen natürlich ein wenig zusätzliche Rechenzeit, da die Pipeline geleert und damit kurzzeitig nicht voll genutzt werden kann. Je öfter sie jedoch erfolgen, desto mehr lässt sich der Zeitbedarf eines einzelnen Aufrufes

senken, so dass auf diese Weise eine Beeinträchtigung für den Benutzer weitestgehend vermieden werden kann.

7 Weitere Betrachtungen

7.1 Übertragung der Kamerabilder

Neue Kamerabilder können zu jedem Zeitpunkt asynchron von der Teleoperatorseite zur Operatorseite gesendet werden. Die nächste Textur-Extraktion kann jedoch immer erst gestartet werden, sobald die Verarbeitung des vorherigen Kamerabildes abgeschlossen ist. Daher wurde ein Bilder-Server zur Entkopplung der empfangenen Kamerabilder und des aktuell zu verarbeitenden Bildes realisiert (Abb. 7.1).

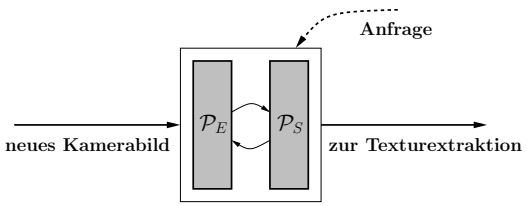


Abbildung 7.1: Entkopplung des Bildempfangs

Die Entkopplung erfolgt dabei durch die Verwendung von Double Buffering. In den Empfangspuffer $\mathcal{P}_E$ wird das jeweils empfangene Kamerabild geschrieben. Sobald dieser Vorgang abgeschlossen ist, werden die Rollen der beiden Puffer vertauscht. Wird ein neues Kamerabild empfangen, bevor das letzte zur Verarbeitung übernommen wurde, so wird dieses überschrieben. Dies ist gerechtfertigt, da in einem solchen Fall ohnehin nicht ausreichend Zeit für eine Verarbeitung zur Verfügung steht. Um keine Bilder verwerfen zu müssen, wäre alternativ zum Double Buffering auch die Verwendung einer Queue denkbar. Dies würde jedoch zu einer Verarbeitung zunehmend älterer Bilder führen, sobald die zeitlichen Abstände mit denen die Bilder empfangen werden die Dauer der Textur-Extraktion unterschreiten.

Die Übernahme der Bilder vom Bilder-Server wird vom prädiktiven Display initiiert (Pulling). Dazu fragt das prädiktive Display, wenn gerade keine Textur-Extraktion stattfindet, in regelmäßigen Abständen beim Bilder-Server an, ob ein neues Kamerabild im Sendepuffer $\mathcal{P}_S$ vorliegt. Diese Anfragen müssen durch die Entkopplung nicht mit der Empfangsrate der Kamerabilder beim Bilder-Server synchron sein.

Zusammen mit den Kamerabildern wird jeweils die Position $\mathbf{P}_t$ abgespeichert, von der aus das Bild aufgenommen wurde. Diese Zuordnung ist für die Gewinnung der Texturen notwendig (s. Kap. 4.1).

Die Übertragung der Bildpakete vom und zum Server wurde mit CORBA realisiert. Dadurch ist es auch problemlos möglich den Server auf einem anderen Rechner laufen zu lassen. Dies hat sich als nützlich erwiesen, um ein Abfallen der Framerate der Darstellung durch die Rechenbelastung beim Empfang eines neuen Bildes zu vermeiden.

7.2 Extraktion der Texturen

7.2.1 Ablauf

In Abbildung 7.2 ist der Ablauf der Texturgewinnung als Blockschaltbild dargestellt. Die Funktionsweise der einzelnen Komponenten ist in den Kapiteln 4 und 5 beschrieben.

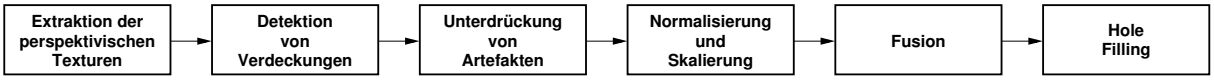


Abbildung 7.2: Ablauf der Texturgewinnung

Bemerkenswert ist, dass die Normalisierung der Texturen *nach* der Artefaktunterdrückung stattfindet. Dies begründet sich daraus, dass die durch die in Kapitel 5.1.1 beschriebenen Ursachen hervorgerufenen Artefakte im Bildraum der Kameraaufnahme entstehen (Abbildung 7.3, links und Mitte). Bei der Normalisierung der Texturen werden auch die Artefakte transformiert. War das Polygon im Kamerabild aus einer schrägen Ansicht zu sehen, so werden die Artefakte dabei teilweise vergrößert (Abbildung 7.3, rechts, rote Pfeile), teilweise verkleinert. Weiterhin ist die Ausdehnung der Artefakte in den normalisierten Texturen abhängig von deren Skalierung.

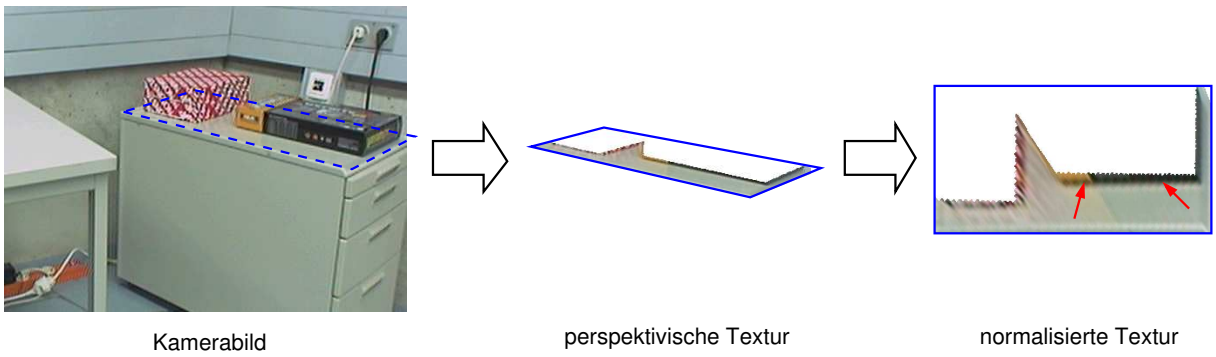


Abbildung 7.3: Größe der Artefakte in perspektivischer und normalisierter Textur

Die Vergrößerung der als verdeckt markierten Bereiche im Texturraum der normalisierten Textur würde daher nicht mehr genau diejenigen Texel berücksichtigen, bei denen am wahrscheinlichsten ist, dass sie fälschlicherweise dem verdeckten Polygon zugeordnet wurden.

7.2.2 Ausführungszeiten

Das prädiktive Display wurde auf unterschiedlichen x86-Architekturen mit einer Vielzahl von Grafikkarten getestet. Die hier präsentierten Ergebnisse wurden auf einem *AMD Athlon XP* mit 2,2 GHz und 1GB Hauptspeicher erzielt. Als Grafikkarte kam ein AGP-Modell der Firma *Leadtek* mit einer *NVIDIA GeForce 6800 GT* GPU und 256 MB Grafikspeicher zum Einsatz. Auf dem PC lief als Betriebssystem Fedora Core 1 Linux, als Grafikkarten-Treiber diente Version 66.29 mit Unterstützung für OpenGL 1.5.

Die Messergebnisse wurden, sofern nicht anders genannt, bei dem in 7.3.1 vorgestellten Szenario mit einem antropomorphen Roboter erzielt. Die Szene bestand aus insgesamt 1936 Polygonen. Die Darstellung erfolgte mit 24 bit Farbtiefe, die Texturen wurden im RGBA8-Format (32 bit) gespeichert. Es wurde jeweils der Mittelwert bei der Verarbeitung von insgesamt 320 Kamerabildern bestimmt. Pro Bild wurden dabei je nach Kameraposition die Texturen von zwischen 15 und 344 Polygonen verarbeitet. Die entzerrten Kamerabilder hatten eine Auflösung von 751x570 Pixel, die Darstellung erfolgte mit 800x600 Pixel.

Sofern nicht anders angegeben, betrugen bei der Normalisierung die Skalierungsfaktoren d=500 Texel/Meter und r_{max}=256 Texel. Die Artefaktunterdrückung erfolgte mit GLSL in jeweils 3 Iterationen. Beim Hole Filling kam das pyramiden-basierte Verfahren zum Einsatz. Für die zunächst beschriebenen Messungen erfolgte parallel zur Textur-Extraktion keine Darstellung.

Verarbeitungsschritt	**Rechenzeit [ms]**	**Anteil [%]**
Extraktion der persp. Textur	14,6	11,6
Verdeckungsrechnung	4,6	3,7
Artefaktunterdrückung	24,1	19,2
Normalisierung und Skalierung	4,8	3,8
Fusion	2,3	1,8
Hole Filling	66,2	52,7
Sonstiges	9,1	7,2
Gesamt	125,7	100

Tabelle 7.1: Durchschnittliche Ausführungszeiten der einzelnen Komponenten der Textur-Extraktion für ein Kamerabild

Tabelle 7.1 zeigt die durchschnittlichen **Ausführungszeiten der einzelnen Komponenten der Textur-Extraktion** für das Beispielszenario. Auffällig ist, dass das Hole Filling über die Hälfte der Rechenzeit beansprucht und auch die Artefaktunterdrückung im Vergleich zu den anderen Komponenten lange dauert. Unter "Sonstiges" fällt vor allem das Laden des Kamerabilds vom Hauptspeicher in den Grafikspeicher. Die Gesamtzeit zeigt, dass etwa acht Bilder pro Sekunde verarbeitet werden konnten. Für die Messungen wurden die Kamerabilder von der Festplatte geladen, dieser Vorgang dauerte durchschnittlich 19,1 ms. Dieser Wert ist jedoch für die Praxis irrelevant, da dort statt dessen der in Kapitel 7.1 beschriebene Server mit niedriger Zugriffszeit zum Einsatz kommt.

Iterationen	Rechenzeit [ms]	
	bilineare Interpolation	mit Shader
0	0	0
2	15,3	18,6
4	25,7	29,5
6	36,1	40,5
8	46,5	51,5
10	56,9	62,4

Tabelle 7.2: Durchschnittliche Ausführungszeiten für die Artefaktunterdrückung in Abhängigkeit von der Methode und Anzahl der Iterationen

In Tabelle 7.2 sind die durchschnittlichen **Ausführungszeiten für die Artefaktunterdrückung** zu sehen. Die Testfälle entsprechen denjenigen aus Abbildung 5.11. Eine Unterscheidung zwischen alternierendem und fortschreitendem Verfahren für die bilineare Interpolation wird hier nicht gemacht, da beide Methoden nahezu identische Messergebnisse liefern. Es ist zu sehen, dass bei gleicher Anzahl Iterationen die bilineare Interpolation geringfügig schneller ist als die Lösung mit Fragment Shader. Wie in Kapitel 5.1.4 beschrieben steht der pro Iteration geringeren Ausdehnung der ungültigen Bereiche bei dem Verfahren mit Shader die bessere Erhaltung der ursprünglichen Form dieser Bereiche gegenüber.

Methode	Rechenzeit [ms]		
	komplexe Szene	bereits teilweise textur. Objekt	bereits stark textur. Objekt
Pyramiden-Ansatz	66,2	58,3	4,8
Shader, ohne Occ. Query	3014,9	892,1	160,7
Shader, mit Occ. Query	1752,5	404,2	3,7

Tabelle 7.3: Durchschnittliche Dauer für das Hole Filling

Tabelle 7.3 zeigt die durchschnittliche **Dauer für das Hole Filling** mit den verschiedenen Methoden. Dabei wurden drei Testfälle betrachtet. Bei der komplexen Szene im Szenario des antropormorphen Roboters erzielt der pyramiden-basierte Ansatz deutlich kürzere Ausführungszeiten. Die sehr hohen Werte des Ansatzes mit Fragment Shader lassen sich unter anderem dadurch erklären, dass mit den Kamerabildern nicht die komplette Szene abgedeckt wurde. Dadurch musste das Hole Filling für Polygone, von denen nur ein kleiner Teil sichtbar war, für das komplette restliche Polygon erfolgen. Das zweite Beispiel zeigt die Rechenzeit für ein Objekt aus 139 Polygonen, für das vor den Messungen bereits teilweise Texturen vorhanden waren. Die Rechenzeit für die Lösung mit Shader nähert sich bei Verwendung des Abbruchskriteriums mit Occlusion Query daher an die des pyramiden-basierten Ansatzes an. Im letzten Beispiel wurde ein sehr einfaches Objekt (Würfel) vorab bereits nahezu vollständig mit Texturen versehen. Es ist zu sehen, dass, sobald nur noch sehr wenige Löcher gefüllt werden müssen, die Lösung mit Shader und Occlusion Query sogar schneller sein kann als der pyramiden-basierte Ansatz.

d [Texel/Meter]	r_{max} [Texel]	Rechenzeit [ms]
50	512	118,3
100	64	100,9
100	256	112,5
100	512	128,9
500	64	104,8
500	256	125,7
500	512	464,7

Tabelle 7.4: Abhängigkeit der Dauer der Textur-Extraktion von der Auflösung der Texturen

In Tabelle 7.4 ist die durchschnittliche **Dauer für die Textur-Extraktion** in Abhängigkeit von den Parametern für die **Skalierung der normalisierten Texturen** zu sehen. Solange alle Texturen in den Grafikspeicher passen, sind hier für die getesteten Parameter Differenzen von maximal knapp 30 Prozent zu erkennen. Bei $d = 500\,\text{Texel/Meter}$, $r_{max} = 256\,\text{Texel}$ konnte noch eine schnelle Verarbeitung bei gleichzeitig guter Texturqualität erzielt werden, weshalb diese Werte für dieses Szenario als optimal genommen wurden. Bei $d = 500\,\text{Texel/Meter}$, $r_{max} = 500\,\text{Texel}$ gab es ein verstärktes Ansteigen der Rechenzeit und starke Schwankungen bei den Messwerten. Die Ursache war, dass der lokale Grafikspeicher nicht mehr ausreichend groß war und die Texturen daher teilweise temporär in den Hauptspeicher ausgelagert werden mussten.

Framerate F_d [Bilder/s]	**Rechenzeit [ms]**	
	Ansicht wenige Polygone	**Ansicht viele Polygone**
ohne Darstellung	125,7	125,7
25	147.4	156,7
33	156,9	171,3
50	179,9	209,9
100	332,8	613,7

Tabelle 7.5: Abhängigkeit der Dauer der Textur-Extraktion von der Framerate

Tabelle 7.5 zeigt die durchschnittliche **Verarbeitungszeit für ein Kamerabild bei gleichzeitiger Darstellung** der Szene. Die Verarbeitungszeit ist in diesem Fall von der gewünschten Framerate F_d und der Komplexität der dargestellten Szene abhängig. Auch bei paralleler Darstellung mit ergonomischen 50 Hz können im Mittel noch 4 bis 5 Bilder pro Sekunde verarbeitet werden, sogar bei Ansicht einer relativ komplexen Szene. Selbst bei einer Ansicht mit vielen Polygonen konnte bei einer gewünschten Framerate von $F_d = 50\,Hz$ diese nahezu konstant erreicht werden, d. h. $F_a \approx F_d$. Die in Kapitel 6.3.2 beschriebene Überlastbehandlung kommt hier somit noch nicht zum Einsatz. Aber auch bei $F_d = 100\,Hz$ konnte bei vielen sichtbaren Polygonen noch eine mittlere tatsächliche Bildrate $\overline{F}_a = 97\,Hz$ erreicht werden. Die kleinste gemessene Bildrate betrug hier $F_{a,min} = 90\,Hz$. Durch die Überlastbehandlung bekommt auch in diesem rechenintensiven Fall die Textur-Extraktion noch Rechenzeit auf CPU und GPU zugeteilt.

7.3 Test-Applikationen

Das auf der in dieser Arbeit beschriebenen Textur-Extraktion basierende prädiktive Display wurde in verschiedenen Szenarien getestet. Diese sollen hier kurz beschrieben und interessante Aspekte hervorgehoben werden.

7.3.1 Antropomorpher Roboter

Die meisten in dieser Arbeit gezeigten Beispielbilder zeigen Ausschnitte eines Szenarios mit dem Roboter *MinERVA* (s. Abb. 1.1). Dieser aus Modulen der Firma *Amtec* aufgebaute Roboter verfügt im Wesentlichen über einen dem Menschen kinematisch nachempfundenen Arm mit sieben Freiheitsgraden und einem Zweifinger-Greifer, sowie einen Stereo-Kamerakopf mit Pan-Tilt-Einheit. Der Roboterarm wird in dem Telepräsenz-Szenario von einem Operator über einen in einem Griff montierten Positions-Tracker gesteuert. Die prädizierte Szene wird dem Benutzer auf einem HMD präsentiert, eine optionale Stereo-Darstellung erhöht dabei den räumlichen Eindruck. Ein zusätzlicher am HMD befestigter Positions-Tracker bestimmt die Position des Betrachters in der Szene. Die Aufgabe des Benutzers in dem Szenario ist das Greifen eines Objektes. Dabei werden dem Benutzer wahlweise die Kamerabilder gezeigt, die über eine simulierte Verzögerungsstrecke variabel verzögert werden können oder er sieht eine prädizierte Ansicht.

Abbildung 7.4: links: prädiziertes Greifen eines Objektes aus Sicht des Benutzers, rechts: Augmentierung

Abbildung 7.4 zeigt links eine typische prädizierte Szene, wie sie der Benutzer sieht. Im dargestellten Ausschnitt wurde ein Objekt gegriffen und mitgeführt. Das Modell für das Szenario wurde vorab durch manuelles Vermessen modelliert. Fehlende haptische Rückkopplung wurde durch eine Drahtgitter-Darstellung des Greifers bei detektierter Kollision so weit wie möglich kompensiert. Während das texturierte Modell sich in einem solchen Fall nicht weiterbewegt, entspricht die zusätzliche Darstellung der Hand des

Roboters der tatsächlichen Position des Benutzers. Diese Methode wurde auch verwendet, wenn der Benutzer seine Hand an eine Position bewegte, die der Roboter aufgrund seiner Kinematik nicht erreichen kann (Abb. 7.4 rechts). In diesem Szenario wurde auch die Aktualisierung der Texturen erfolgreich getestet.

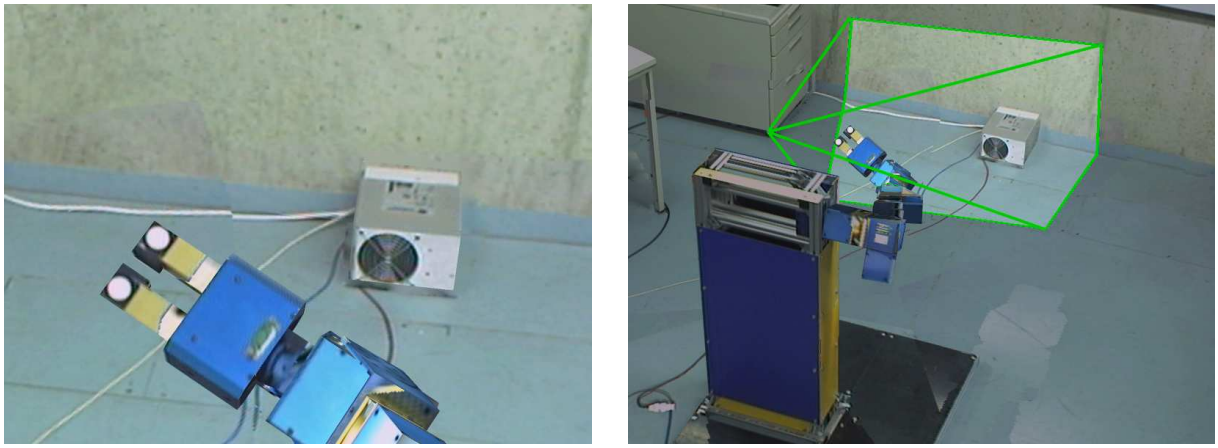


Abbildung 7.5: Operator/Betrachter-Betrieb: links: Sicht des Operators, rechts: Sicht des Betrachters

Weiterhin erfolgte in dieser Anwendung die Realisierung eines *Operator/Betrachter-Betriebs*. Dabei wird einem Operator, wie zuvor beschrieben, die prädizierte Szene präsentiert, mit der er auch interagieren kann (Abb. 7.5, links). Zusätzlich wird einem oder mehreren weiteren Benutzern die Möglichkeit gegeben die gleiche Szene zu betrachten. Dabei sind sie lediglich passiv anwesend und können ausschließlich ihre Betrachterposition kontrollieren. Diese Anwendung ist beispielsweise zu Lernzwecken sinnvoll, wo der Lernende zunächst die Aktionen eines Experten beobachten soll. Abbildung 7.5 zeigt rechts eine prädizierte Ansicht für einen Betrachter. Entscheidend ist in dieser Anwendung, dass der Betrachter auch erkennen kann welchen Bildausschnitt der Operator gerade sieht. Dies wird durch Augmentierung der Position des Operators und Darstellung von dessen Sichtvolumen als Rampenlicht erreicht.

7.3.2 Industrie-Roboter

In einem weiteren Szenario wurde ein *KUKA* Industrieroboter mithilfe von *Phantom*-Eingabegeräten gesteuert. Die Aufgabe des Bedieners dieses Demonstrators war das Greifen und Einfügen eines Kolbens in einen Motorblock (Abb. 7.6, links). Der Demonstrator war eine Kooperation mehrerer Teilprojekte des SFB 453, eine Beschreibung von Teilen des Szenarios ist in [80] zu finden.

Bei der Anwendung des prädiktiven Displays in diesem Szenario wurde die Verzögerung nicht simuliert, sondern entstand tatsächlich durch die Übertragung zwischen Operator und Teleoperator, die sich an unterschiedlichen Standorten befanden (Operator: TU München, Teleoperator: Deutsches Zentrum für Luft- und Raumfahrt in Oberpfaffenho-

Abbildung 7.6: links: Einfügen eines Kolbens in einen Motorblock, rechts: Prädizierte Ansicht

fen) und variierte zwischen 40 ms und 130 ms. Auch in diesem Szenario erfolgte die geometrische Modellierung des Roboters und seiner Umgebung, sowie die Registrierung der Kamerabilder, vorab manuell. Im laufenden Betrieb wurde keine Aktualisierung der Texturen durchgeführt. Abbildung 7.6 zeigt rechts eine Prädiktion der Ansicht des Roboters bei Mitführung des gegriffenen Kolbens.

7.3.3 Herz-Chirurgie

Ein häufiges Problem in der minimal-invasiven Chirurgie ist das geringe Field-of-View der endoskopischen Kameras. Da sich diese üblicherweise relativ nah am Operationsgebiet befinden, ist für den Operateur nur ein kleiner Ausschnitt der Szene sichtbar. Diese Situation wird für den Patienten gefährlich, sobald die Instrumente aus dem Blickfeld bewegt werden. Durch das üblicherweise bei der Steuerung der Instrumente erforderliche häufige Indexing ("Umgreifen") neigt der Chirurg in einem solchen Fall dazu die Hand-Auge-Koordination zu verlieren. Die Bewegungen der Instrumente als Folge seiner Handbewegungen sind für ihn nicht mehr sichtbar, was zu einer Verletzung des Patienten führen kann. Die Relokalisierung der Instrumente ist für den Chirurgen eine langsame und ermüdende Aufgabe. Aus diesem Grund kam als dritte Anwendung das prädiktive Display bei einem Experimentalaufbau für minimal-invasive chirurgische Eingriffe am Beispiel von Herzoperationen zum Einsatz. In dem Szenario wurden dabei in einem künstlichen Thorax an einem Modell eines Herzens Knoten geknüpft (Abb. 7.7) [65, 72].

In dieser Anwendung war weniger eine *zeitliche* als eine *räumliche* Prädiktion das Ziel. Im Gegensatz zu den beiden vorherigen Szenarien wurde hier das Modell nicht manuell bestimmt, sondern mit den in [78] und [79] beschriebenen Methoden aus einem vom Trokarpunkt aus aufgenommenen Bildpaar rekonstruiert. Mit den aus einem dieser Bilder

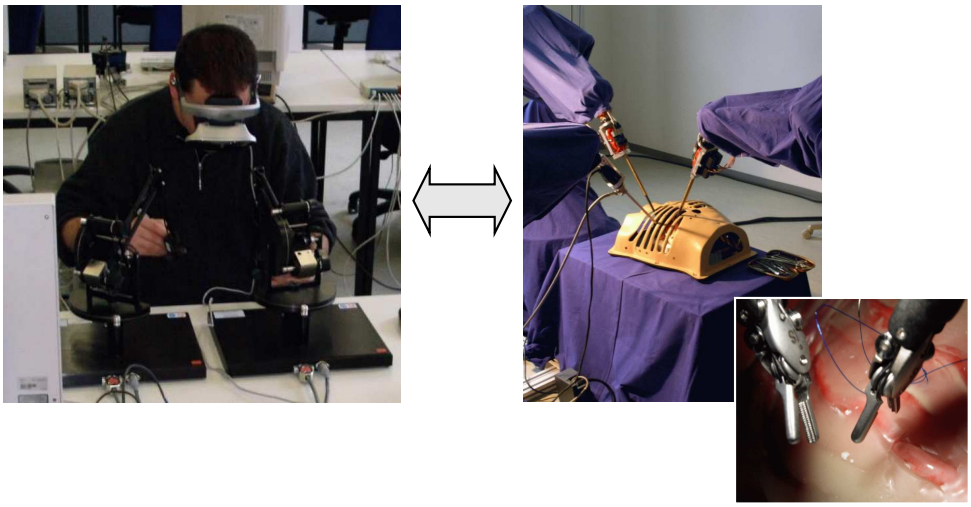


Abbildung 7.7: Minimal-invasive Chirurgie an einem künstlichen Herzen mit einem Roboter (MIRS)

gewonnenen Texturen konnte dann dem Operateur die Szene in einer fotorealistischen virtuellen Übersicht präsentiert werden. Abbildung 7.8 zeigt links eine solche Szene. Die schwarzen Löcher sind Bereiche, die bei der Rekonstruktion der Szene nicht mit Polygonen modelliert werden konnten. Durch eine Abbildung von Polygongröße auf Texturauflösung mit $d = 10.000\,\text{Texel/Meter}$ konnte bei der Textur-Extraktion trotz der eher mikroskopischen Abmessungen dieses Szenarios mit etwa 5.000 Polygonen ein Überlaufen des Texturspeichers verhindert werden. Für die Anwendung war das zusätzliche Einblenden der Instrumente an ihren aktuellen Positionen auf Basis der gemessenen Lage der Instrumente erforderlich (Abb. 7.8, Mitte). Dem Operateur war somit eine Lokalisation der Instrumente auch außerhalb des Kamerabilds möglich. Um dem Operateur eine Zuordnung zur Ansicht der realen Kamera zu ermöglichen, wurde deren Blickfeld, ähnlich wie bei der in Kapitel 7.3.1 beschriebenen Anwendung, in die Ansicht als eine Art Rampenlicht augmentiert (Abb. 7.8, rechts).

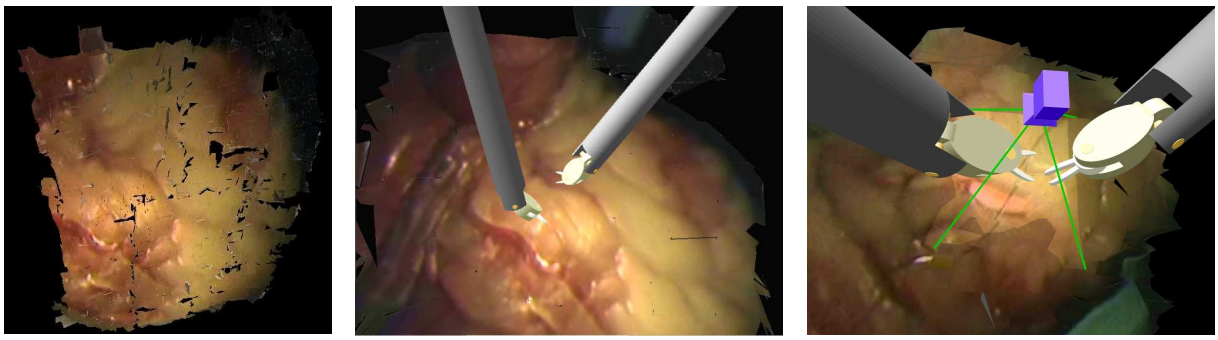


Abbildung 7.8: Prädizierte Übersichtsbilder in einem Chirurgie-Szenario: links: texturiertes Modell eines Herzens, Mitte: Szene mit Instrumenten, rechts: zusätzliche Augmentierung der Kamera

8 Zusammenfassung

Um dem Benutzer einer Telepräsenzanwendung effektives Arbeiten zu ermöglichen, ist es zwingend erforderlich, ihm das visuelle Ergebnis seiner Aktionen ohne spürbare Verzögerung zu präsentieren. Prädiktive Displays ermöglichen dies auch für Anwendungen mit Latenzzeiten im Übertragungskanal und/oder begrenzter Bandbreite, indem eine Vorhersage der zu erwartenden Bilder erfolgt. Zumeist stellen die Bilder dabei einfache polygonale Szenen dar, die auch als solche zu erkennen sind. In der vorliegenden Arbeit wurden Verfahren für die Realisierung eines neuartigen fotorealistischen prädiktiven Displays vorgestellt. Fotorealismus wird durch die Verwendung von Bildinformationen aus den (verzögert eintreffenden) Kamerabildern bei der Darstellung eines benutzerseitig lokalen 3D-Modells erreicht. Die besondere Herausforderung war dabei die automatische Extraktion und Weiterverarbeitung von Texturen aus den Kamerabildern. Ziel war es, auf Basis dieser Texturen möglichst realistische und für den Benutzer stimmige synthetische Ansichten zu generieren.

Damit der Benutzer Änderungen in der Umgebung des Roboters wahrnehmen kann, ist es erforderlich, die Texturen nicht nur einmalig zu gewinnen, sondern sie statt dessen im laufenden Betrieb ständig zu aktualisieren. Die Gewinnung muss dabei möglichst schnell ablaufen, um die Aktualität der verwendeten Texturen zu garantieren. Zur Zeit verfügbare 3D-Grafikkarten sind für die schnelle Verarbeitung von Bilddaten optimiert und können auch für viele weitere parallelisierbare Aufgaben genutzt werden. Um dies für die Verarbeitung der enormen Datenmengen bei der Gewinnung der Texturen auszunutzen, wurden alle Algorithmen mit dem Ziel entwickelt, möglichst große Teile direkt auf der Grafikkarte ablaufen zu lassen. Damit verbunden war jedoch auch die Beschränkung auf die durch die Grafik-API zur Verfügung gestellte Funktionalität.

Für das prädiktive Display ist die Speicherung vollständiger Kamerabilder und ihre Verwendung für die Texturierung nicht ausreichend, da damit die für eine realistische Visualisierung erforderliche zusätzliche Nutzung der Bildinformation aus älteren Aufnahmen nicht erreicht werden kann. Aus diesem Grund erfolgte die Verwaltung der Texturen statt dessen pro Polygon. Für die Extraktion der Texturen sind dabei lediglich das Kamerabild, das polygonale Modell zum Zeitpunkt der Aufnahme sowie intrinsische und extrinsische Kameraparameter erforderlich. Mit diesen Daten konnten zunächst Texturen als Bildausschnitte pro Polygon gewonnen werden. Diese wurden in normalisierte Texturen transformiert, um eine einheitliche Verwaltung sowie eine Skalierung zur Regulierung des benötigten Bedarfs an Texturspeicher zu ermöglichen. Weiterhin musste berücksichtigt werden, dass ein Kamerabild keine Farbinformationen für Teile von Objekten enthalten kann, die sich aus Kamerasicht in der Szene hinter anderen Objekten befinden. Daher

wurde eine Methode zur Detektion von Verdeckungen realisiert, die sich den auf der Grafikkarte vorhandenen Tiefenpuffer zunutze macht. Ungültige Bereiche wurden dabei für die weitere Verarbeitung in den Texturen markiert. Da die Roboterkameras üblicherweise nicht starr sind, sondern der Blickrichtung des Benutzers folgen, sind die Verdeckungen in zeitlich aufeinander folgenden Bildern unterschiedlich. Um möglichst viel verfügbare Farbinformation zu verwenden, aber auch gleichzeitig die virtuellen Ansichten aktuell zu halten, wurde eine Fusion mit älteren Texturen bei Priorisierung der aktuellsten Farbinformationen realisiert.

Die Erfassung der Daten kann in realen Anwendungen immer nur eine Näherung sein. So sind bei der Extraktion der Texturen extrinsische und intrinsische Kameraparameter nur Schätzungen und auch das Modell approximiert die reale Welt mit Polygonen nur. Diese und weitere Ursachen führen unweigerlich zu Zuordnungen von Bereichen des Kamerabildes zu falschen Polygonen. Besonders wenn irrtümlicherweise Polygonen im Hintergrund der Szene Bildbereiche zugeordnet werden, die eigentlich zu Polygonen im Vordergrund gehören, kann dies zu sehr störenden Artefakten bei der anschließenden Darstellung führen. Bei heute gängigen Verfahren zur manuellen Extraktion von Texturen werden diese Bereiche in der Regel von Hand selektiert und verworfen. Da für ein prädiktives Display aber eine Extraktion ohne menschliche Interaktion erforderlich ist, musste die Behandlung solcher Gebiete statt dessen automatisch erfolgen. Weil aus den Bildinformationen keine sichere Aussage über die korrekte Zuordnung von Bildinformationen zu Polygonen getroffen werden kann, ist ein Kompromiss zwischen dem Verwerfen von möglichst viel falschen Bildinformationen und dem Erhalt von möglichst viel korrekten Farbinformationen notwendig. Aus diesem Grund wurden Methoden entwickelt, um Farbinformationen zu verwerfen, von denen am wahrscheinlichsten ist, dass sie dem falschen Polygon zugeordnet wurden. In Testanwendungen konnten durch diese Maßnahmen nahezu alle derartigen Artefakte entfernt werden.

Auch nach der Verarbeitung von vielen Kamerabildern gibt es fast immer noch Bereiche der Szene, die bisher in keinem Kamerabild sichtbar waren und für die somit keine Farbinformation vorhanden ist. Die gerade genannten Methoden zur Unterdrückung von Artefakten reduzieren die pro Polygon verfügbaren Farbinformationen weiter. Um dem Benutzer ein stimmiges und vollständiges Gesamtbild präsentieren zu können, war das Füllen solcher Löcher in den Texturen unumgänglich. Auch diese Aufgabe wird in vielen herkömmlichen Anwendungen von Hand vollzogen oder es werden komplexe und langsame Verfahren eingesetzt. Um kurze Ausführungszeiten zu erreichen, wurden in der vorliegenden Arbeit Methoden entwickelt, die direkt auf der Grafikkarte Löcher in den Texturen durch Interpolation vorhandener Farbinformation füllen können. Kleinere Löcher sind auf diese Weise fast nicht mehr wahrzunehmen. Erst bei großen Löchern innerhalb einer Textur mit viel Struktur sind diese Bereiche zu erkennen. In der Praxis sind aber auch diese Bereiche üblicherweise nicht besonders störend, zumal durch die Aktualisierung der Texturen bei Nachführung der Kameras solche Löcher ohnehin nur temporär vorhanden sind.

Gleichzeitig zur Aktualisierung der Texturen muss dem Benutzer ständig ein flüssiges Bild präsentiert werden. Der durch Nutzung der Grafikkarte durch beide Tasks entstehen-

de Ressourcen-Konflikt konnte durch Parallelisierung bei gemeinsamer Texturverwaltung und unter Berücksichtigung zeitkritischer Grafik-Operationen gelöst werden. Ein für diese Anwendung entwickeltes Scheduling-Konzept garantiert dabei eine nahezu konstante Framerate bei der Darstellung, während die verbleibende Rechenzeit für die Extraktion der Texturen genutzt werden kann.

Die gemessenen Ausführungszeiten bestätigten die durch die intensive Ausnutzung der Grafik-Hardware erhoffte schnelle Verarbeitung der Texturen. Durchschnittlich konnten pro Sekunde aus etwa acht Kamerabildern Texturen gewonnen werden. Selbst bei gleichzeitiger Darstellung der Szene mit ergonomischen 50 Hz wurden noch etwa fünf Kamerabilder pro Sekunde verarbeitet. Mit diesen Ergebnissen kann eine ständige Aktualisierung der Texturen gewährleistet werden, was einen Einsatz des fotorealistischen prädiktiven Displays in praktischen Anwendungen rechtfertigt.

Ein Test des prädiktiven Displays in einem vollständigen geschlossenen System inklusive Modellaufbau konnte aufgrund der vergleichsweise langsamen Modellrekonstruktion nicht erfolgen. Die Funktionsfähigkeit der beschriebenen Methoden wurde jedoch anhand von drei Anwendungen belegt, in denen das Gesamtsystem aus Textur-Extraktion und Visualisierung zum Einsatz kam. Die Hauptanwendung war die telepräsente Steuerung eines Siebengelenk-Roboters bei Darstellung von wahlweise prädizierter Szene oder realer Bilder mit simulierter Verzögerung über ein Head-Mounted-Display. Das Ziel der Präsentation fotorealistischer Bilder kann dabei als voll erreicht bezeichnet werden. In zahlreichen Demonstrationen konnten viele Personen sich von der Qualität selbst überzeugen und gaben das Empfinden der Realitätsnähe subjektiv stets als sehr gut an. Die häufig gestellte Frage “Sehe ich gerade die reale oder die virtuelle Ansicht?” belegt dies. So konnten auch ungeübte Testpersonen ein Objekt mit Hilfe eines Roboters erfolgreich greifen, während sie die virtuelle Ansicht präsentiert bekamen. Erfolgreiche Tests des prädiktiven Displays wurden auch bei der telepräsenten Steuerung eines räumlich entfernten Industrieroboters bei tatsächlich vorhandenen Latenzzeiten durchgeführt. Für ein medizinisches Demonstrationsszenario wurde das prädiktive Display schließlich zur Generierung von Übersichtsbildern für die Lokalisation der Instrumente bei minimal-invasiven Herzoperationen erweitert. In dieser Anwendung wurde es auch erfolgreich an Modellen gestestet, die mittels Rekonstruktion generiert wurden.

A Anhänge

A.1 Register Combiners

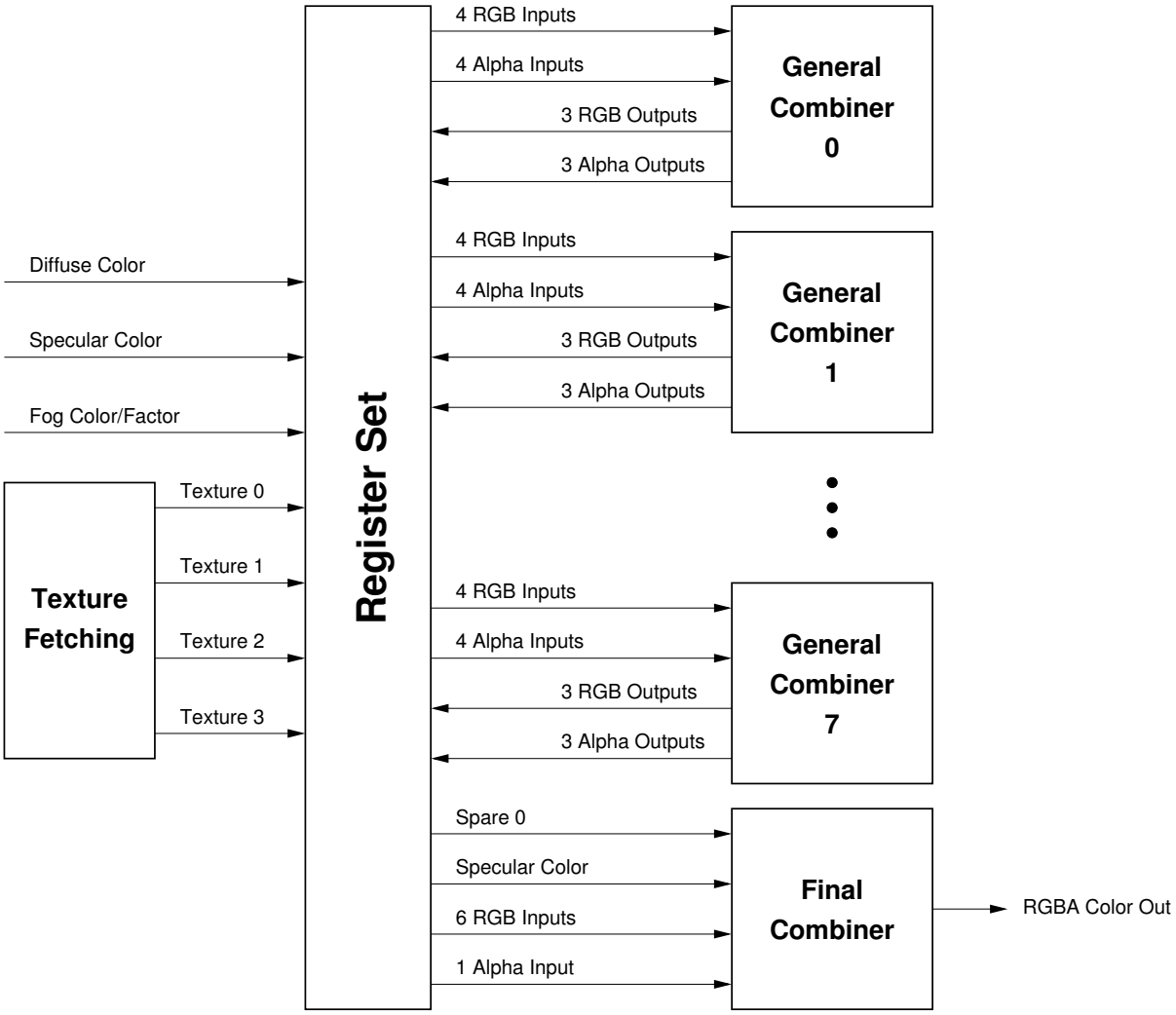


Abbildung A.1: Register Combiners der GeForce3

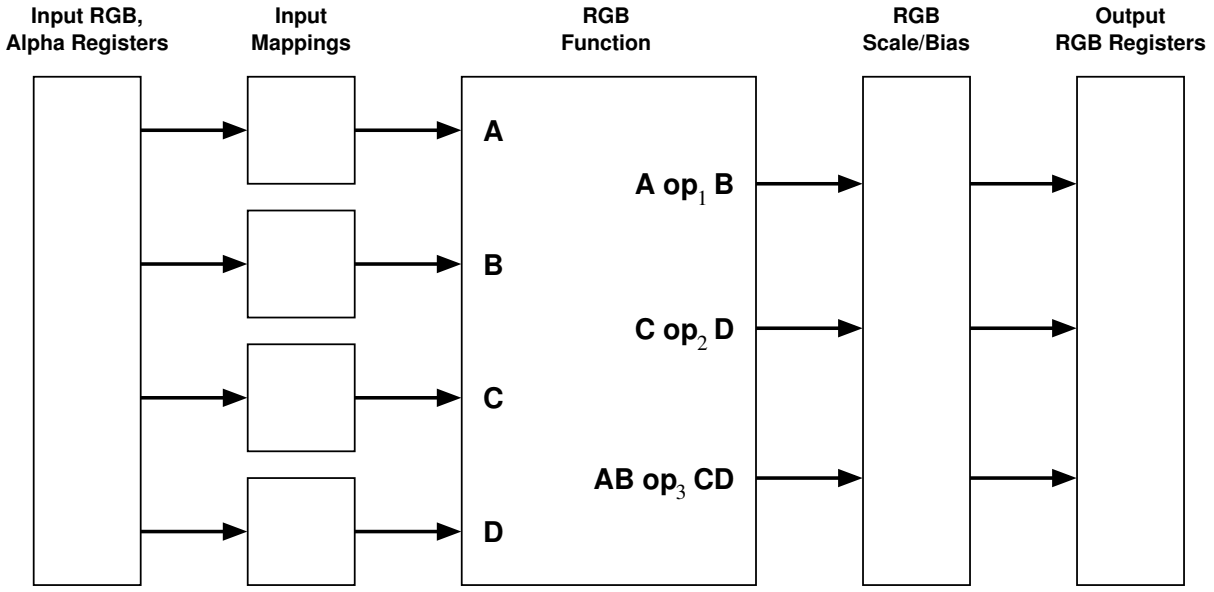


Abbildung A.2: General Combiner Stufe der RGB-Werte

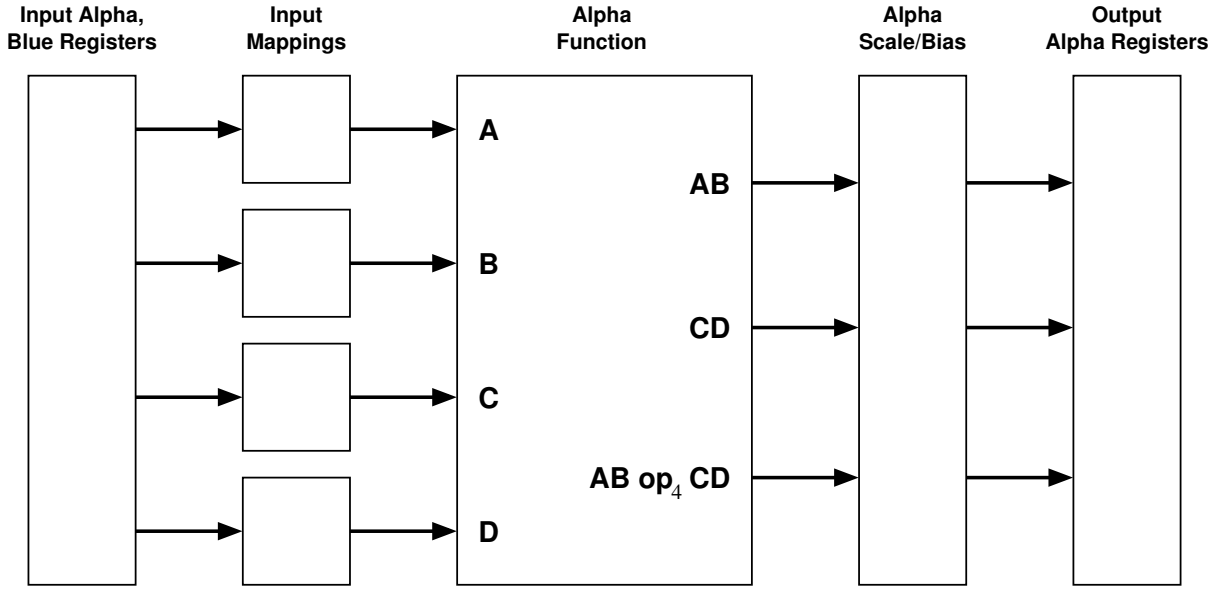


Abbildung A.3: General Combiner Stufe der Alphawerte

Register	**Read**	**Write**
Diffuse Color	yes	yes
Specular Color	yes	yes
Texture 0	yes	yes
Texture 1	yes	yes
Texture 2	yes	yes
Texture 3	yes	yes
Spare 0	yes	yes
Spare 1	yes	yes
Constant Color 0	yes	no
Constant Color 1	yes	no
Fog Color and Factor	RGB only	no
Zero	yes	no
Discard	no	yes

Tabelle A.1: Register der General Combiners

Name		**Funktion** $\underline{C} = (r, g, b)$	
A *op* B	AB	$\underline{C}_{AB}$	$(r_A r_B, g_A g_B, b_A b_B)$
	$A \cdot B$	$\underline{C}_{A \cdot B}$	$(r_A r_B + g_A g_B + b_A b_B,$ $r_A r_B + g_A g_B + b_A b_B,$ $r_A r_B + g_A g_B + b_A b_B)$
C *op* D	CD	$\underline{C}_{CD}$	$(r_C r_D, g_C g_D, b_C b_D)$
	$C \cdot D$	$\underline{C}_{C \cdot D}$	$(r_C r_D + g_C g_D + b_C b_D,$ $r_C r_D + g_C g_D + b_C b_D,$ $r_C r_D + g_C g_D + b_C b_D)$
AB *op* CD	$AB + CD$	$\underline{C}_{AB+CD}$	$\underline{C}_{AB} + \underline{C}_{CD}$
	AB mux CD	$\underline{C}_{ABmuxCD}$	$\underline{C}_{AB}$ für $\alpha_{Spare0} < 0.5$ $\underline{C}_{CD}$ für $\alpha_{Spare0} \geq 0.5$

Tabelle A.2: Combiner Function RGB

Name		**Funktion** α	
AB		α_{AB}	$\alpha_A \alpha_B$
CD		α_{CD}	$\alpha_C \alpha_D$
A *op* B	$AB + CD$	α_{AB+CD}	$\alpha_A \alpha_B + \alpha_C \alpha_D$
	AB mux CD	$\alpha_{ABmuxCD}$	$\alpha_A \alpha_B$ für $\alpha_{Spare0} < 0.5$ $\alpha_C \alpha_D$ für $\alpha_{Spare0} \geq 0.5$

Tabelle A.3: Combiner Functions Alpha

Name	**Funktion**	**Mapping**
Signed Identity	$f(x) = x$	$[-1, 1] \rightarrow [-1, 1]$
Signed Negate	$f(x) = -x$	$[-1, 1] \rightarrow [1, -1]$
Unsigned Identity	$f(x) = max(0, x)$	$[0, 1] \rightarrow [0, 1]$
Unsigned Invert	$f(x) = 1 - min(max(0, x), 1)$	$[0, 1] \rightarrow [1, 0]$
Expand Normal	$f(x) = 2 * max(0, x) - 1$	$[0, 1] \rightarrow [-1, 1]$
Expand Negate	$f(x) = -2 * max(0, x) + 1$	$[0, 1] \rightarrow [1, -1]$
Half Bias Normal	$f(x) = max(0, x) - \frac{1}{2}$	$[0, 1] \rightarrow [-\frac{1}{2}, \frac{1}{2}]$
Half Bias Negate	$f(x) = -max(0, x) + \frac{1}{2}$	$[0, 1] \rightarrow [\frac{1}{2}, -\frac{1}{2}]$

Tabelle A.4: Input Mappings

Name	**Funktion**
Scale by $\frac{1}{2}$	$f(x) = \frac{x}{2}$
No Scale	$f(x) = x$
Scale by 2	$f(x) = 2x$
Scale by 4	$f(x) = 4x$
Bias by $-\frac{1}{2}$	$f(x) = x - \frac{1}{2}$
Bias by $-\frac{1}{2}$ and Scale by 2	$f(x) = 2(x - \frac{1}{2})$

Tabelle A.5: Output Scale und Bias

Index

Literaturverzeichnis

[1] *ARTESAS*. http://www.artesas.de.

[2] *ARVIKA*. http://www.arvika.de.

[3] Barth, Michael, Tim Burkert, Christof Eberst, Norbert O. Stöffler und Georg Färber: *Photo-Realistic Scene Prediction of Partially Unknown Environments for the Compensation of Time Delays in Telepresence Applications*. In: *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'00)*, Band 4, Seiten 3132–3137, April 2000.

[4] Bejczy, Antal K., Steven Venema und Won S. Kim: *Role of Computer Graphics in Space Telerobotics: Preview and Predictive Displays*. In: *Cooperative Intelligent Robotics in Space*, Band 1387 der Reihe *Proc. of the SPIE*, Seiten 365–377, November 1990.

[5] Bovet, Daniel P. und Marco Cesati: *Understanding the Linux Kernel*. O'Reilly & Associates, Inc, Januar 2003.

[6] Bresenham, Jack E.: *Algorithm for computer control of a digital plotter*. IBM Systems Journal, 4(1):25–30, 1965.

[7] Buck, Ian, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston und Pat Hanrahan: *Brook for GPUs: Stream Computing on Graphics Hardware*. In: *ACM Transactions on Graphics (SIGGRAPH) 2004*, Band 23, Seiten 777–786, August 2004.

[8] Burkert, Tim und Georg Färber: *Photo-Realistic Scene Prediction*. In: *Mechatronics & Robotics (MECHROB) 2004*, Aachen, Deutschland, 2004.

[9] Burkert, Tim, Jan Leupold und Georg Passig: *Hardware Accelerated Texture Extraction for a Photo-Realistic Predictive Display*. In: Ertl, T., B. Girod, H. Niemann, H.-P. Seidel, E. Steinbach und R. Westermann (Herausgeber): *Vision, Modeling, and Visualization 2003*, Seiten 11–18, Technische Universität München, November 2003.

[10] Burkert, Tim, Jan Leupold und Georg Passig: *Scene Model Acquisition from Stereo Vision for Photo-Realistic Scene Prediction*. In: Knoll, A. (Herausgeber): *Third IEEE International Conference on Humanoid Robots, Workshop on Telepresence*, Technische Universität München, Oktober 2003.

[11] Burkert, Tim, Jan Leupold und Georg Passig: *A Photo-Realistic Predictive Display*. Presence, Seiten 22–43, 2004.

[12] Burkert, Tim und Georg Passig: *A Photo-Realistic Predictive Display for Telepresence Applications*. In: Färber, Georg und Jens Hoogen (Herausgeber): *Advances in Interactive Multimodal Telepresence Systems*, Technische Universität München, März 2001. DFG.

[13] Burkert, Tim und Georg Passig: *Ein Prädiktives Display für Telepräsenzanwendungen*. In: Dillmann, R. (Herausgeber): *Human Centered Robotic Systems*, Band 1, Seiten 75–82, 2002.

[14] Carranza, Joel, Christian Theobalt, Marcus A. Magnor und Hans-Peter Seidel: *Free-viewpoint video of human actors*. In: *SIGGRAPH 03 Conf. Proc.*, Band 22, Seiten 569–577, 2003.

[15] Carr, Nathan A., Jesse D. Hall und John C. Hart: *The Ray Engine*. In: *Proc. of Graphics Hardware*, Seiten 37–46, 2002.

[16] Chen, Shenchang Eric: *QuickTime VR - An Image-Based Approach to Virtual Environment Navigation*. Computer Graphics, 29:29–38, 1995.

[17] Chen, Shenchang Eric und Lance Williams: *View Interpolation for Image Synthesis*. In: *SIGGRAPH 93 Conf. Proc.*, Seiten 279–288, 1993.

[18] Chen, Wei-Wao, , Herman Towles, Lars Nyland, Greg Welch und Henry Fuchs: *Toward a Compelling Sensation of Telepresence: Demonstrating a portal to a distant (static) office*. In: *Visualization 2000*, Seiten 327–333. IEEE, 2000.

[19] Cobzas, Dana und Martin Jägersand: *Tracking and Predictive Display for a Remote Operated Robot using Uncalibrated Video*. In: *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'05)*, 2005.

[20] Colantoni, Philippe, Nabil Boukala und Jérôme Da Rugna: *Fast and Accurate Color Image Processing Using 3D Graphics Cards*. In: Ertl, T., B. Girod, H. Niemann, H.-P. Seidel, E. Steinbach und R. Westermann (Herausgeber): *Vision, Modeling, and Visualization 2003*, Seiten 383–390, November 2003.

[21] Cruz-Neira, Carolina, Daniel J. Sandin und Thomas A. DeFanti: *Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE*. In: *SIGGRAPH 93 Conf. Proc.*, Band 37, Seiten 135–142, 1993.

[22] Debevec, Paul E., George Borshukov und Yizhou Yu: *Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping*. In: *9th Eurographics Rendering Workshop*, Seiten 105–116, Wien, Juni 1998.

[23] Debevec, Paul E., Camillo J. Taylor und Jitendra Malik: *Modeling and Rendering Architecture from Photographs: a hybrid geometry- and image-based approach*. In: *SIGGRAPH 96 Conf. Proc.*, Band 30, Seiten 11–20, 1996.

[24] *DirectX 9.0 graphics, High-Level Shader Language.* http://msdn.microsoft.com/directx.

[25] Ellis, Stephen R., Mark J. Young, Bernard D. Adelstein und Sheryl M. Ehrlich: *Discrimination of changes in latency during head movement.* In: *8th International Conference on Human-Computer Interaction (HCI'99)*, Seiten 1129–1133, München, 1999.

[26] Encarnação, José, Wolfgang Strasser und Reinhard Klein: *Graphische Datenverarbeitung 2.* Oldenbourg, 1997.

[27] Encarnação, José, Wolfgang Strasser und Reinhard Klein: *Graphische Datenverarbeitung 1.* Oldenbourg, 1998.

[28] Fernando, Randy, Mark Harris, Matthias Wloka und Cyril Zeller: *Programming Graphics Hardware.* In: *Eurographics 2004*, 2004.

[29] Ferrell, William R.: *Remote manipulation with transmission delay.* IEEE Transactions in Human Factors in Electronics, 6(1):24–32, 1965.

[30] Foley, James D., Andries Van Dam, Steven K. Feiner und John F. Hughes: *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[31] Fung, James und Steve Mann: *Computer Vision Signal Processing on Graphics Processing Units.* In: *Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)*, Seiten 93–96, Mai 2004.

[32] Fung, James, Felix Tang und Steve Mann: *Mediated Reality Using Computer Graphics Hardware for Computer Vision.* In: *Sixth International Symposium on Wearable Computing*, Seiten 83–89, Seattle, USA, Oktober 2002.

[33] *General-Purpose computation on GPUs.* http://www.gpgpu.org.

[34] Gibson, William: *Neuromancer.* Ace Books, 1984.

[35] Gortler, Steven J., Radek Grzeszczuk, Richard Szeliski und Michael F. Cohen: *The Lumigraph.* SIGGRAPH 96 Conf. Proc., 30:43–54, 1996.

[36] Gross, Markus, Stephan Würmlin, Martin Naef, Edouard Lamboray, Christian Spagno, Andreas Kunz, Esther Koller-Meier, Tomas Svoboda, Luc Van Gool, Silke Lang, Kai Strehlke und Andrew Vande Moere- and Oliver Staadt: *blue-c: A Spatially Immersive Display and 3D Video Portal for Telepresence.* In: *SIGGRAPH 03 Conf. Proc.*, Band 22, Seiten 819–827. ACM, 2003.

[37] Harris, Mark J., William V. Baxter, Thorsten Scheuermann und Anselmo Lastra: *Simulation of cloud dynamics on graphics hardware.* In: *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Seiten 92–101, 2003.

[38] Herold, Helmut: *Linux-Unix-Systemprogrammierung.* Addison-Wesley-Longman, Bonn, 1999.

[39] Hirzinger, Gerd, Bernhard Brunner, J. Dietrich und J. Heindl: *ROTEX – The First Remotely Controlled Robot in Space.* In: *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'94)*, Band 3, Seiten 2604–2611, Los Alamitos, CA, USA, Mai 1994. IEEE Computer Society Press.

[40] Hoff, Kenneth E. III, Tim Culver, John Keyser, Ming Lin und Dinesh Manocha: *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware.* In: *SIGGRAPH 99 Conf. Proc.*, Seiten 277–286, 1999.

[41] Hoff, Kenneth E. III, Tim Culver, John Keyser, Ming C. Lin und Dinesh Manocha: *Interactive Motion Planning Using Hardware-Accelerated Computation of Generalized Voronoi Diagrams.* In: *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'00)*, Seiten 2931–2937, April 2000.

[42] Jägersand, Martin: *Image-based predictive display for high d.o.f. uncalibrated tele-manipulation using affine and intensity subspace models.* Advanced Robotics, 14(8):683–701, Februar 2001.

[43] Joshi, Pushkar und Leslie Ikemoto: *Harnessing the GPU for General Purpose Computation: A Case Study.* Technischer Bericht, EECS, Berkeley University of California, 2004.

[44] Kanade, Takeo, P. J. Narayanan und Peter W. Rander: *Virtualized Reality: Concepts and Early Results.* In: *IEEE Workshop on the Representation of Visual Scenes*, Seiten 69–76, Juni 1995.

[45] Kanade, Takeo, Peter Rander, Sundar Vedula und Hideo Saito: *Virtualized Reality: Digitizing a 3D Time-Varying Event As Is and in Real Time.* In: Yuichi Ohta, Hideyuki Tamura (Herausgeber): *Mixed Reality, Merging Real and Virtual Worlds*, Seiten 41–57. Springer-Verlag, 1999.

[46] Kelshikar, Nikhil, Xenophon Zabulis, Jane Mulligan, Kostas Daniilidis, Vivek Sawant, Sudipta Sinha, Travis Sparks, Scott Larsen, Herman Towles, Ketan Mayer-Patel, Henry Fuchs, John Urbanic, Kathy Benninger, Raghurama Reddy und Gwendolyn Huntoon: *Real-time Terascale Implementation of Tele-immersion.* In: *Terascale Performance Analysis Workshop*, Seiten 33–42, Melbourne, Australien, Juni 2003.

[47] Kessenich, John, Dave Baldwin und Randi Rost: *The OpenGL Shading Language (Specification)*, 2004. Language Version 1.10.

[48] Kilgard, Mark J.: *OpenGL Programming for the X Window System.* Addison-Wesley Publishing Company, 1996.

[49] Kim, Won S.: *Virtual Reality Calibration and Preview/Predictive Displays for Telerobotics.* Presence, 5(2):173–189, 1996.

[50] Krüger, Jens, Thomas Schiwietz, Peter Kipfer und Rüdiger Westermann: *Numerical Simulations on PC Graphics Hardware.* In: *ParSim 2004 (Special Session of EuroPVM/MPI 2004)*, Seiten 442–450, 2004.

[51] Krüger, Jens und Rüdiger Westermann: *Linear algebra operators for GPU implementation of numerical algorithms.* ACM Transactions on Graphics (TOG), 22(3):908–916, 2003.

[52] Lane, J. Corde, Craig R. Carignan, Brook R. Sullivan, David L. Akin, Teresa Hunt und Rob Cohen: *Effects of time delay on telerobotic control of neutral buoyancy.* In: *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'02)*, Seiten 2874–2879, Washington, USA, 2002.

[53] Lang, Andreas: *Transparente Textur-Aktualisierung in einem Virtual-Reality-System.* Diplomarbeit, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 2002.

[54] Lang, Silke, Martin Naef, Markus Gross und Ludger Hovestadt: *IN:SHOP - Using Telepresence and Immersive VR for a New Shopping Experience.* In: Ertl, T., B. Girod, H. Niemann, H.-P. Seidel, E. Steinbach und R. Westermann (Herausgeber): *Vision, Modeling, and Visualization 2003*, Seiten 3–10, Technische Universität München, November 2003.

[55] Lengyel, Eric: *The OpenGL Extensions Guide.* Charles River Media, 2003.

[56] Lengyel, Jed: *The Convergence of Graphics and Vision.* Computer, 31:46–53, 1998.

[57] Levoy, Marc und Pat Hanrahan: *Light Field Rendering.* SIGGRAPH 96 Conf. Proc., 30:31–42, 1996.

[58] Lindholm, Erik, Mark J. Kilgard und Henry Moreton: *A user-programmable vertex engine.* In: *SIGGRAPH 01 Conf. Proc.*, Seiten 149–158, 2001.

[59] Mark, William R.: *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping.* Doktorarbeit, University of North Carolina at Chapel Hill, April 1999.

[60] Mark, William R., R. Steven Glanville, Kurt Akeley und Mark J. Kilgard: *Cg: A system for programming graphics hardware in a C-like language.* In: *SIGGRAPH 03 Conf. Proc.*, Band 22, Seiten 896–907, San Diego, USA, 2003.

[61] Mark, William R., Leonard McMillan und Gary Bishop: *Post-Rendering Image Warping for Latency Compensation.* Technischer Bericht TR96-020, Univ. of North Carolina at Chapel Hill, 1996.

[62] Mark, William R., Leonard McMillan und Gary Bishop: *Post-Rendering 3D Warping.* In: *Symposium on Interactive 3D Graphics*, Seiten 7–16, April 1997.

[63] Matusik, Wojciech, Chris Buehler, Ramesh Raskar, Steven J. Gortler und Leonard McMillan: *Image-Based Visual Hulls.* In: *SIGGRAPH 00 Conf. Proc.*, Seiten 369–374, 2000.

[64] Maughan, Chris und Matthias Wloka: *Vertex Shader Introduction.* Technischer Bericht, NVIDIA Corporation, 2001.

[65] Mayer, Hermann, István Nagy, Alois Knoll, Eva U. Schirmbeck und Robert Bauernschmitt: *Robotic System to Evaluate Force Feedback in Minimally Invasive Computer Aided Surgery.* In: *Proc. of International Design Engineering Technical Conferences*, USA, September 2004. ASME.

[66] McCool, Michael D., Zheng Qin und Tiberiu S. Popa: *Shader Metaprogramming.* In: *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware Conf. Proc.*, Seiten 57–68, 2002.

[67] McCool, Michael D., Stefanus Du Toit, Tiberiu Popa, Bryan Chan und Kevin Moule: *Shader Algebra.* In: *ACM Transactions on Graphics (SIGGRAPH) 2004*, Band 23, Seiten 787–795, 2004.

[68] McMillan, Leonard und Gary Bishop: *Plenoptic Modeling: An Image-Based Rendering System.* SIGGRAPH 95 Conf. Proc., 29:39–46, 1995.

[69] McReynolds, Tom und David Blyth: *Advanced Graphics Programming Techniques using OpenGL*, 1999.

[70] Minsky, Marvin: *Telepresence.* Omni, 1980.

[71] Möller, Tomas und Eric Haines: *Real-Time Rendering.* AK Peters, Ltd., 2002.

[72] Nagy, István, Hermann Mayer, Alois Knoll, Eva U. Schirmbeck und Robert Bauernschmitt: *EndoPAR: An Open Evaluation System for Minimally Invasive Robotic Surgery.* In: *Mechatronics & Robotics (MECHROB) 2004*, Aachen, Deutschland, September 2004. IEEE.

[73] Ogden, J. M., E. H. Adelson, J. R. Bergen und P.J. Burt: *Pyramid-based computer graphics.* RCA Engineer, 30(5):4–15, 1984.

[74] Oliveira, Manuel M., Gary Bishop und David McAllister: *Relief Texture Mapping.* In: *SIGGRAPH 00 Conf. Proc.*, Seiten 359–368, 2000.

[75] *OpenGL Extensions Registry.* http://oss.sgi.com/projects/ogl-sample/registry.

[76] *OpenVIDIA : GPU accelerated Computer Vision Library.* http://www.openvidia.org/.

[77] Osgood, Charles Egerton: *Projection dynamics in perception: Perception of movement.* In: *Method and Theory in Experimental Psychology*, Seiten 243–248, New York, 1953. Oxford University Press.

[78] Passig, Georg: *Umgebungsmodellierung auf der Basis von Stereo-Kamerabildern.* Doktorarbeit, Technische Universtät München, 2005.

[79] Passig, Georg, Tim Burkert und Jan Leupold: *Scene Model Acquisition for a Photo-Realistic Predictive Display and its Application to Endoscopic Surgery.* In: *Mirage 2005*, Seiten 89–97, März 2005.

[80] Preusche, Carsten, Jens Hoogen, Detlef Reintsema, Günther Schmidt und Gerd Hirzinger: *Flexible Multimodal Telepresent Assembly using a Generic Interconnection Framework.* In: *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'02)*, Seiten 3712–3718, USA, 2002.

[81] Purcell, Timothy J.: *Ray Tracing on a Stream Processor.* Doktorarbeit, Stanford University, März 2004.

[82] Purcell, Timothy J., Ian Buck, William R. Mark und Pat Hanrahan: *Ray Tracing on Programmable Graphics Hardware.* In: *SIGGRAPH 02 Conf. Proc.*, Band 21, Seiten 703–712, Juli 2002.

[83] *Qt-Online-Dokumentation.* http://doc.trolltech.com/products/qt.

[84] Roddenberry, Gene: *Star Trek: The Next Generation*, 1986.

[85] Rost, Randi J.: *OpenGL Shading Language.* Addison-Wesley Publishing Company, 2004.

[86] S3 Incorporated: *S3TC DirectX 6.0 standard texture compression*, 1998.

[87] Schroll, Christof: *Erhöhung der Realitätsnähe in einem Virtual Reality System durch Fehlerverschleierung.* Diplomarbeit, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 2001.

[88] Segal, Mark und Kurt Akeley: *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*, 2004.

[89] Segal, Mark, Carl Korobkin, Rolf van Widenfelt, Jim Foran und Paul Haeberli: *Fast shadows and lighting effects using texture mapping.* In: *SIGGRAPH 92 Conf. Proc.*, Band 26, Seiten 249–252, 1992.

[90] Seitz, Steven M. und Charles R. Dyer: *View Morphing.* SIGGRAPH 96 Conf. Proc., 30:21–30, 1996.

[91] Serra, Jean: *Image analysis and mathematical morphology.* Academic Press, London, 1982.

[92] Shreiner, Dave, Mason Woo, Jackie Neider und Tom Davis: *OpenGL Programming Guide.* Addison-Wesley Publishing Company, 4. Auflage, 2003.

[93] *Sonderforschungsbereich 453: Wirklichkeitsnahe Telepräsenz und Teleaktion.* http://www.sfb453.de.

[94] Sonderforschungsbereich 453: *Wirklichkeitsnahe Telepräsenz und Teleaktion.* Finanzierungsantrag 1999-2001, 1998.

[95] Spagno, Christian P. und Andreas M. Kunz: *Construction of a three-sided immersive telecollaboration system.* In: *IEEE Virtual Reality Conference (VR 2003)*, Seiten 37–44, 2003.

[96] Stein, Matthew R.: *Behaviour-Based Control for Time-Delayed Teleoperation.* Doktorarbeit, University of Pennsylvania, 1994.

[97] Sutherland, Ivan E., Robert F. Sproull und Robert A. Schumacker: *A Characterization of Ten Hidden-Surface Algorithms.* Computing Surveys, 4(1):25–30, 1974.

[98] Taubenberger, Thomas: *Photorealistische Modellvisualisierung durch inverses Texture-Mapping.* Diplomarbeit, Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 2000.

[99] Thomas, Geb, Ted Blackmon, Michael Sims und Daryl Rasmussen: *Video engraving for virtual environments.* In: *Proc. of SPIE: Stereoscopic Displays and Virtual Reality Systems IV*, Seiten 462–471, 1997.

[100] Thompson, Chris J., Sahngyun Hahn und Mark Oskin: *Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis.* In: *Proc. of 35th International Symposium on Microarchitecture (MICRO-35)*, Seiten 306–317, 2002.

[101] Towles, Herman, Wei-Chao Chen, Ruigang Yang, Sang-Uok Kum, Henry Fuchs, Nikhil Kelshikar, Jane Mulligan, Kostas Daniilidis, Loring Holden, Bob Zeleznik, Amela Sadagic und Jaron Lanier: *3D Tele-Collaboration Over Internet2.* In: *International Workshop on Immersive Telepresence (ITP 2002)*, Juan-les-Pins, Frankreich, Dezember 2002.

[102] Williams, Lance: *Pyramidal Parametrics.* In: *SIGGRAPH 83 Conf. Proc.*, Band 17, Seiten 1–11, Juli 1983.

[103] Womack, Paula und Jon Leech: *OpenGL Graphics with the X Window System (Version 1.3).* Silicon Graphics Inc., Mountain View, California, 1998.

[104] Woop, Sven, Jörg Schmittler und Philipp Slusallek: *RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing.* In: *SIGGRAPH 05 Conf. Proc.*, 2005.

[105] Würmlin, Stephan, Edouard Lamboray, Oliver G. Staadt und Markus H. Gross: *3D Video Recorder: A System for Recording and Playing Free-Viewpoint Video.* In: Duke, David und Roberto Scopigno (Herausgeber): *Proc. of the 10th Pacific Conference on Computer Graphics and Applications*, Seiten 325–334. Blackwell Publishing Ltd, Oxford, U.K., 2002.