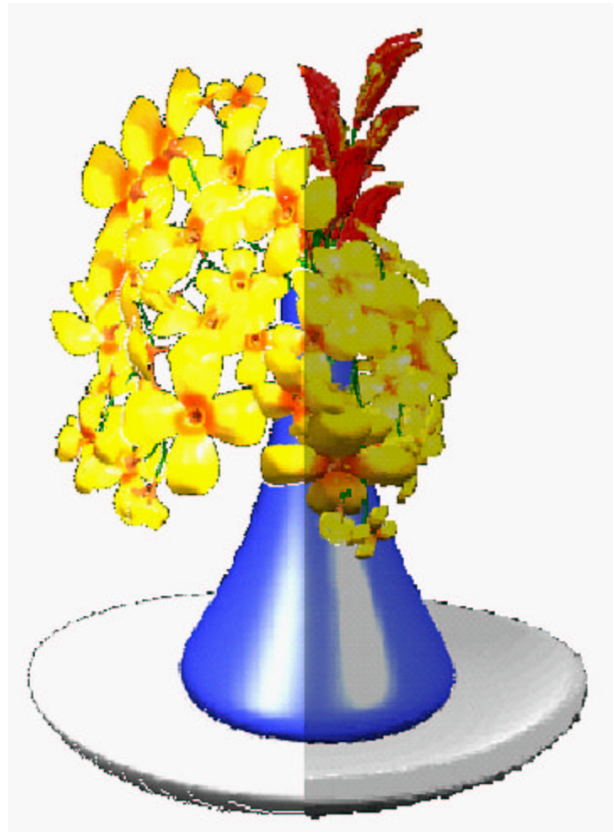


# Erweiterung eines geometriebasierten Szenengraphen um punktbasierte Objekte



**Marco Nef**

Semesterarbeit  
Sommersemester 2001

Computer Graphics Lab  
Departement Informatik, ETH Zürich

Prof. Dr. Markus Gross  
Stephan Würmlin



# Abstract

The *blue-c* is a joint research project between several institutes at ETH Zürich, Switzerland. The research aims to build a new collaborative virtual environment where users can immerse into virtual worlds, meet, work and communicate with each other. The system records users by video cameras and projects a virtual environment simultaneously. From video images a three-dimensional representation of the users is generated and inserted into the virtual environment.

The work presented in this thesis extends an existing 3D Image-Warping algorithm by the ability of **shading** the transformed object according to the scene environment. Because the output data of the warper is image-based, an integration of this data into the geometry-based and as such illuminable world is needed. Only with this homogenization of the two completely different worlds it is possible to realize a visually correct result when rendering the warped **point objects**. Such a high degree of realism of the visualization is very important for the users of a *blue-c* portal to really be able to immerse into the virtual environment.

In a first step a prototype in OpenGL is implemented, which in a second step is implemented in OpenGL Performer, using the *blue-c* API.

The basis of this work is a 3D Image-Warping algorithm that itself is an extension of an algorithm developed at MIT. It supports multiple reference images and knows processing on several processors and incrementally evaluating the warping equation.



# Zusammenfassung

Das *blue-c* Projekt ist ein interdisziplinäres Forschungsprojekt an der Eidgenössischen Technischen Hochschule ETH in Zürich, welches zum Ziel hat, eine kollaborative virtuelle Umgebung zu entwickeln. Es soll möglich werden, dass mehrere Personen in die virtuelle Welt eintauchen, dass sie sich dort treffen, miteinander kommunizieren und zusammenarbeiten können. Das System nimmt Bilder mittels Videokameras auf und projiziert gleichzeitig eine virtuelle Umgebung in denselben Raum. Aus den Videobildern wird eine 3D-Repräsentation der Benutzer generiert und in die virtuelle Umgebung eingefügt.

Diese Arbeit erweitert eine vorhandene Implementierung eines 3D Image-Warping Algorithmus um die Möglichkeit der **Schattierung**. Da jedoch die Ausgabedaten des Warpers bildbasiert sind, ist eine Integration dieser Daten in die geometriebasierte und damit beleuchtbare virtuelle Welt erforderlich. Erst durch diese Homogenisierung der beiden unterschiedlichen Welten werden die für das *blue-c* Projekt benötigten, visuell richtig empfundenen Resultate beim Rendering der gewarpten **Punktobjekte** möglich. Für die Benutzer eines *blue-c* Protals ist ein hoher Realitätsgrad der Visualisierung Voraussetzung, um in die virtuelle Realität eintauchen zu können.

In einem ersten Schritt wird ein Prototyp in OpenGL implementiert, welcher anschliessend nach OpenGL Performer portiert wird.

Der zugrundeliegende 3D Image-Warping Algorithmus ist eine Erweiterung eines am MIT entwickelten Algorithmus und berücksichtigt mehrere Referenzbilder. Ausserdem ist er in der Lage, die Berechnung parallel auf mehreren Prozessoren und inkrementell auszuführen.



## Semesterarbeit von Marco Nef

# ERWEITERUNG EINES GEOMETRIEBASIERTEN SZENENGRAPHEN UM PUNKTBASIERTE OBJEKTE

### Einleitung

Die *blue-c* ist eine kollaborative, immersive virtuelle Umgebung, welche zurzeit als Polyprojekt an der ETH entwickelt wird. Ziel dieses Projektes ist der Aufbau einer Virtual Reality Umgebung, in welcher sich Benutzer im virtuellen Raum treffen, miteinander kommunizieren und zusammenarbeiten können. Das System erfasst Benutzer über Videokameras und projiziert gleichzeitig eine virtuelle Umgebung. Aus den Videobildern wird ein dreidimensionales Bild des Benutzers erzeugt und in die andere virtuelle Umgebung eingefügt.

### Ziel

Ziel dieser Semesterarbeit ist die Erweiterung eines geometriebasierten Szenengraphen, wie zum Beispiel OpenGL Performer, um punktbasierte Objekte. Dazu soll der Szenengraph von Performer um einen neuen Nodetypen erweitert werden, welcher dann auch nachträglich Operationen wie *Shading* oder *Shadowing* am Objekt ermöglichen soll.

### Aufgabenstellung

Die Aufgaben dieser Semesterarbeit können in folgende Komponenten aufgeteilt werden:

- *Erweiterung des Szenengraph von OpenGL Performer* zur Repräsentation von punktbasierten Objekten. Dazu soll der Szenengraph um einen neuen Node-Type erweitert werden, der in einer ersten Implementation schon vorhanden ist.
- *Implementation von "Deferred Operations" am punktbasierten Objekt:*
  - *Per-Point Re-Shading* des Objekts mit den Beleuchtungsattributen der Performer Szene. Es sollen verschiedene Verfahren ausgewählt, implementiert und auf ihre Performance untersucht werden. Dazu müssen aus den Daten des punktbasierten Objekts zuerst Normalen abgeschätzt werden. Es soll weiter versucht werden, die Leistung eines PC-Systems mit GeForce2-Karte auszuschöpfen.
  - *Shadowing*. Das Objekt wirft einen Schatten in die Szene, welcher generiert und dargestellt werden muss.
- *Integration und Test der Implementation am ersten Prototypen der blue-c* unter Benutzung des darin integrierten blue-c Warpers, der die punktbasierten Daten des Objektes liefert.

Als Entwicklungsplattform dient eine sgi Workstation, die Anwendung wird auf dem blue-c Rendering-System, einer sgi Onyx3200, getestet. Zusätzlich soll die Implementation auf einen Linux-PC portiert und dort getestet werden. Die Implementation soll in C++, OpenGL und Performer erfolgen.

Die Semesterarbeit steht unter der Obhut von Prof. Dr. Markus Gross und wird von Stephan Würmlin betreut.

*Ausgabe:* Montag, 2. April 2001

*Abgabe:* Freitag, 6. Juli 2001

(Prof. Dr. M. Gross)



# Inhaltsverzeichnis

<b>Abstract</b>	
<b>Zusammenfassung</b>	
<b>Aufgabenstellung</b>	
<b>1 Einleitung</b>	1
1.1 Polyprojekt <i>blue-c</i>	1
1.1.1 <i>Real, virtuell?</i>	1
1.1.2 <i>Einsatzgebiete</i>	2
1.2 Vorarbeiten, Image Warping	2
1.3 Motivation und Ziele	3
1.4 Aufbau der Arbeit	4
<b>2 Normalenberechnung und Schattierung</b>	5
2.1 Mathematische Grundlagen	5
2.2 Konstante Schattierung	5
2.3 Gouraud Shading	6
2.4 Phong Shading	7
<b>3 Prototyp eines Warpers mit OpenGL</b>	9
3.1 OpenGL	9
3.1.1 <i>Vorteile für Entwickler</i>	9
3.1.2 <i>API Hierarchie</i>	10
3.1.3 <i>Pipeline</i>	10
3.2 Implementierung	11
3.2.1 <i>Normalenberechnung</i>	11
3.2.2 <i>Integration der Normalen in die Szene</i>	13
3.2.3 <i>Lichtquelle</i>	14
<b>4 Prototyp eines Warpers mit OpenGL Performer</b>	15
4.1 OpenGL Performer	15
4.1.1 <i>Szenengraph</i>	15
4.1.2 <i>Multi-Process Environment</i>	17
4.1.3 <i>Shared Memory Arena</i>	18
4.2 <i>blue-c</i> API	19
4.3 Implementierung	19
<b>5 Resultate</b>	21
5.1 Visueller Eindruck	21
5.2 Performanz	21
5.3 Optimierungen	24
5.3.1 <i>Koordinatensysteme</i>	24
5.3.2 <i>Normalenberechnung</i>	24
5.3.3 <i>Hardwareunterstützung</i>	25
5.3.4 <i>Glanzpunkte</i>	26
<b>6 Zusammenfassung und Ausblick</b>	27
6.1 Ausblick	27
<b>Referenzen</b>	29



# 1

## Einleitung

Die vorliegende Semesterarbeit ist ein Teil des *blue-c* Projektes [1,4,7], einer kollaborativen, immersiven, virtuellen Umgebung, welche zur Zeit als Polyprojekt an der Eidgenössischen Technischen Hochschule ETH in Zürich entwickelt wird.

Dieses erste Kapitel gibt einen groben Überblick über das *blue-c* Projekt und beschreibt vorangegangene Arbeiten, auf denen diese Semesterarbeit aufbaut.

### 1.1 Polyprojekt *blue-c*

Zum jetzigen Zeitpunkt ist die Informationstechnologie respektive die Hard- und Softwareentwicklung an einem Punkt angekommen, wo man in der Lage ist, virtuelle Welten (*virtual environments, virtual realities*) in Echtzeit in einer Qualität zu generieren, dass der Betrachter diese künstlich generierte Welt als eigene zu akzeptieren bereit ist. Durch Computerspiele sind solche Technologien der breiten Öffentlichkeit bereits zugänglich, in der militärischen wie auch der zivilen Ausbildung werden hochwertige virtuelle Systeme seit einiger Zeit benutzt, zum Beispiel in Form von Flugsimulatoren zur Ausbildung von Piloten.



#### 1.1.1 Real, virtuell?

All diesen bestehenden virtuellen Systemen ist gemein, dass sie aus einem System bestehen, welches eine virtuelle Realität generiert, sowie einer Person, die sich in dieser künstlichen Welt bewegt. Interaktivität findet in dem Sinne statt, dass die in die virtuelle Welt eintretende Person die Möglichkeit hat, sich dort zu bewegen und unter Umständen diese Welt auch zu manipulieren. Es kann auch sein, dass durch den Computer generierte Figuren der Person entgegen treten.

Das auf drei Jahre ausgelegte *blue-c* Projekt beinhaltet Grundlagenforschung zur Entwicklung einer virtuellen Umgebung der nächsten Generation mit dem Schwerpunkt, die Interaktion zwischen Menschen, die sich physisch weit distanziert voneinander befinden, zu ermöglichen. Diese Idee ist vergleichbar mit einer Videokonferenz, wobei der grosse Unterschied zu beachten ist, dass hier ein vollständig dreidimensionales virtuelles Environment gebildet wird.

Um eine Zusammenarbeit mehrerer Teilnehmer über beliebige Distanz in Echtzeit zu ermöglichen, werden die sich in einem Portal befindlichen Benutzer live aufgenommen (siehe Abbildung 1.2) und gleichzeitig in die virtuelle Szene projiziert. Echtzeit-Analyse der Videobilder

wird Farb- und Tiefeninformation liefern, welche benötigt wird, um die Benutzer in die virtuelle, dreidimensionale Umgebung einzufügen. Via Hochgeschwindigkeitsnetzwerk werden die Daten zwischen den Benutzern über grössere Distanzen übertragen. Somit wird es ihnen ermöglicht, sich in einem virtuellen Raum zu treffen, zu kommunizieren und zusammenzuarbeiten.

Fähigkeiten von *blue-c* werden sein:

- Dreidimensional dargestellte Repräsentationen von Personen
- Bewegungen und Gespräche in Echtzeit
- Interaktion zwischen Mensch und Computermodell

### 1.1.2 Einsatzgebiete

Die am *blue-c* beteiligten Projektgruppen (Computer Graphics Lab, Center of Product Development, Computer Vision Group, Computer Aided Architectural Design Group) bringen vor allem durch die Interdisziplinarität diverse Einsatzideen ein.

In der Architektur wird es dank einer virtuellen Umgebung möglich sein, geplanten, aber noch nicht gebauten Gebäuden zu begehen. In der Automobil- oder Flugzeugindustrie wird es möglich sein, eine virtuelle Entwicklungswerkstätte zu schaffen, wo verschieden, über den Globus verteilte Teams zusammenarbeiten können. Eine medizinische Anwendung wäre eine von Ärzten in verschiedenen Städten gemeinsam durchgeführte Diagnose oder Operation. Nicht zu vergessen sind natürlich die diversen denkbaren Realisierungen unter der Federführung der Unterhaltungsindustrie. Das Holodeck lässt grüssen...

## 1.2 Vorarbeiten, Image Warping

Die vorliegende Arbeit baut auf einer Semesterarbeit von Stefan Hösli [13] auf. Darin wurde ein dreidimensionaler Image-Warping Algorithmus nach [2] entwickelt und implementiert.

Die vorliegende Semesterarbeit realisiert keinen Warping-Algorithmus weshalb hier nicht auf den mathematischen Hintergrund und die Funktionsweise solcher Algorithmen eingegangen wird. Der interessierte Leser sei auf die genannten Referenzen verwiesen.

In [13] wurden verschiedene Image-Warping Algorithmen implementiert. Für das *blue-c* Projekt wurde anschliessend die Entscheidung getroffen, nur den sogenannten *SheetFlashWarper* zu benutzen, welcher folgende Charakteristika hat:

- Mehrere Referenzbilder können verwendet werden
- Verdeckungsprobleme sind gelöst
- Durch Splatting werden Löcher verhindert
- Berechnung ist auf mehrere Prozessoren parallelisierbar
- Progressives Warping ist möglich

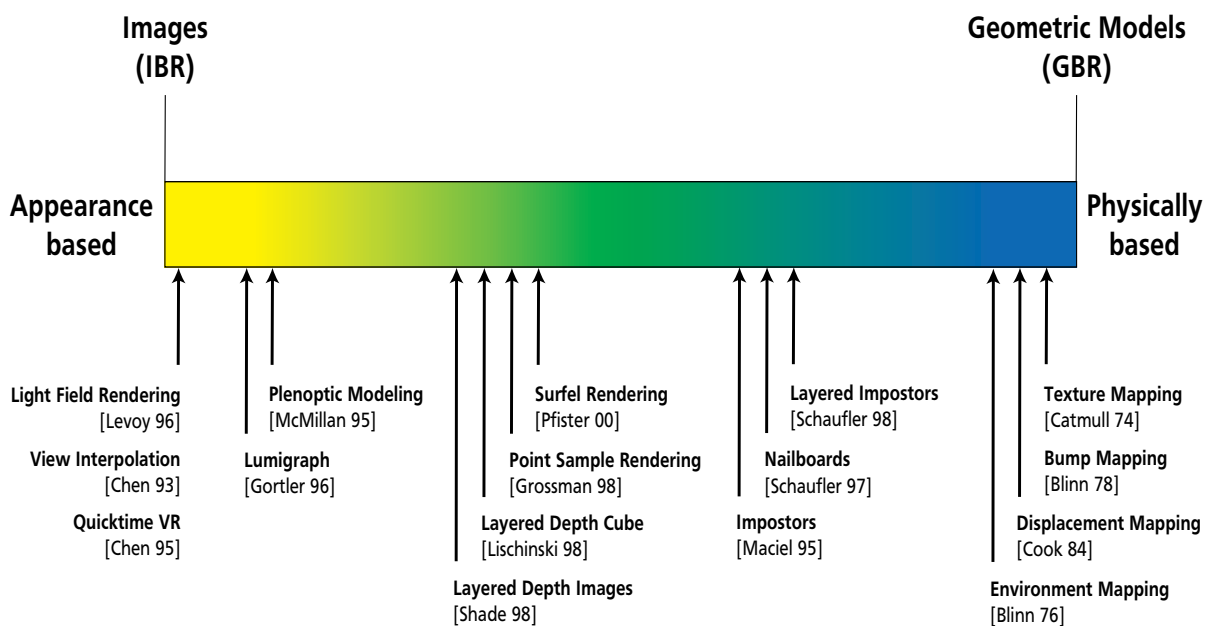
Die Ausgabe des Warping-Algorithmus ist eine Punktwolke. Etwas genauer ausgedrückt handelt es sich um ein Bild mit der Sicht von der aktuellen Position des Beobachters auf das darzustellende Objekt. Diese Information ist zweidimensional. Zusätzlich ist der Abstand der einzelnen Bildpunkte vom Beobachter bekannt, woraus zusammen mit der Position und Blickrichtung des Beobachters die dreidimensionale Position der Bildpunkte berechnet werden kann.

## 1.3 Motivation und Ziele

Voraussetzung für die photorealistische Darstellung von dreidimensionalen Daten ist eine adäquate Beleuchtung der Szene. Im Zusammenhang mit dieser Beleuchtung sollen die dargestellten Oberflächen entsprechend ihrer Position zur Lichtquelle schattiert werden.

Beim *geometriebasierten Rendering*<sup>1</sup> werden Objekte respektive ihre Oberflächen durch ihre Geometrie und Position im Raum beschrieben. Es ist daher mit einfacher Vektormathematik möglich, Normalen der einzelnen Oberflächensegmente zu berechnen, welche die Ausrichtung des darzustellenden und zu beleuchtenden Elements beschreiben. Ist diese Ausrichtung bekannt, so kann der Anteil des von der Lichtquelle<sup>2</sup> ausgesandten Lichts berechnet werden, der das Flächenelement beleuchtet. Diese Beleuchtungsintensität wie auch die Farbe der Beleuchtungsquelle bestimmen die zu rendernde Farbe, welche je nach Ausrichtung heller oder dunkler als die im Modell für das betrachtete Element definierte Farbe sein kann, bei farbigem Licht gar eine andere Farbe sein kann.

Ganz anders sieht die Situation beim *bildbasierten Rendering*<sup>3</sup> aus. Es gibt keine geometrische Beschreibung der darzustellenden Objekte, sondern es werden von Kameras aufgenommene Bilder gewarpt und anschliessend dargestellt. Weil Kameras nur beleuchtete Objekte filmen können, enthalten die entstehenden Bilder immer eine Beleuchtung und damit auch Glanzpunkte oder matte Stellen.



**Abbildung 1.1:** Spektrum der Möglichkeiten zwischen geometriebasiertem (GBR) und bildbasiertem (IBR) Rendering (aus [9]).

Im Rahmen des *blue-c* Projektes werden die geometriebasierte und die bildbasierte Welt verbunden (vergleiche Abbildung 1.1). Die Generierung der virtuellen Repräsentanten der Teilnehmer wird mit Videokameras realisiert. Die entstehenden Bilder sind also bereits beleuchtet. Andere Objekte, welche in die virtuelle Welt eingefügt werden, haben eine geometrische

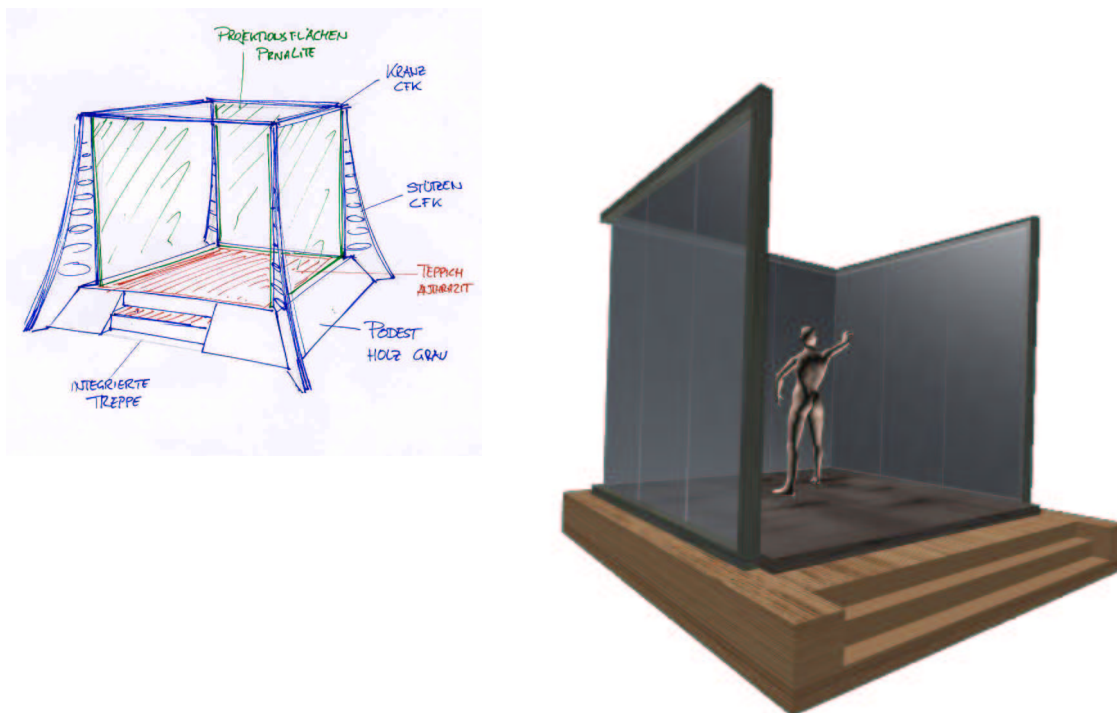
1. geometry-based rendering
2. Selbstverständlich können auch mehrere Lichtquellen benutzt werden
3. image-based rendering

Beschreibung. Um eine möglichst realistische Darstellung mit Schattierung und Schattenwurf erreichen zu können, ist es daher unabdingbar, eine Beleuchtung einzuführen.

Ziel dieser Semesterarbeit ist es, eine Methode zu erarbeiten, die es möglich macht, punkt-basierte Objekte in einer geometriebasierten Welt visuell richtig zu beleuchten. Der bestehende Prototyp eines Warpers in OpenGL von Stefan Hösli [13] soll um Schattierung erweitert werden und schliesslich nach OpenGL Performer portiert werden.

## 1.4 Aufbau der Arbeit

Ebenso wie der Ablauf der Semesterarbeit ist auch diese Niederschrift in zwei zentrale Teile unterteilt, welche in den Kapiteln 3 und 4 im Anschluss an einige mathematische Grundlagen beschrieben werden. Es handelt sich dabei um die beiden Prototypen für das *blue-c* Projekt, wovon je einer in OpenGL und in OpenGL Performer implementiert ist. Im Anschluss daran werden die Ergebnisse diskutiert und einer kritischen Prüfung unterzogen. Das letzte Kapitel ist schliesslich noch einem kurzen Ausblick gewidmet.



**Abbildung 1.2:** Design-Studie für die Portale der *blue-c*.

# 2

## Normalenberechnung und Schattierung

Die Voraussetzung, um in OpenGL eine realistische Beleuchtung mit der entsprechenden Schattierung (*shading*) des dargestellten Objektes einzuführen, sind Normalen. In diesem Kapitel werden die mathematischen Grundlagen der Normalenberechnung und Schattierung beschrieben. Der Leser möge beachten, dass es sich beim Shading nicht um Schattenwurf (*shadowing*) auf andere Objekte handelt.

### 2.1 Mathematische Grundlagen

Im mathematischen Sinne sind *Normalen* Vektoren, welche senkrecht auf einer Fläche stehen, die selber durch Vektoren aufgespannt wird. In der Computergraphik sind solche Flächen meist Dreiecke, im allgemeinen Fall jedoch Polygone, welche aber ihrerseits in Dreiecke zerlegt werden können. Die Dreiecke sind bestimmt durch die Eckpunkte, deren Koordinaten durch einen Punktvektor bestimmt werden. Das führt zu folgendem Bild

Die Normale  $\vec{N}$  wie in 2.1 wird folgendermassen mit dem Vektorprodukt<sup>1</sup> berechnet:

$$\vec{N} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 \cdot b_3 - a_3 \cdot b_2 \\ a_3 \cdot b_1 - a_1 \cdot b_3 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{bmatrix}$$

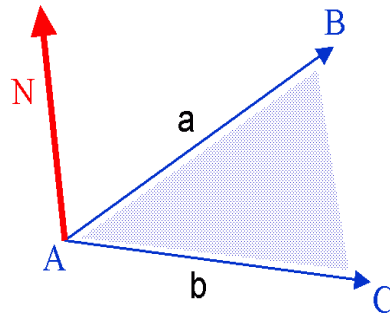
Es gibt diverse Möglichkeiten, auf einem Polygon Normalen für die Schattierung zu bestimmen, was zu verschiedenen Realitätsgraden beim Rendering führt. Die wichtigsten Schattierungsmodelle werden im folgenden nach [3] beschrieben.:

### 2.2 Konstante Schattierung

Unter der speziellen Voraussetzung, dass sich die Lichtquelle im Unendlichen befindet oder das zu beleuchtende Polygon sehr klein (kleiner als ein Pixel) ist, genügt der einfachste Fall:

---

1. entspricht dem Kreuzprodukt

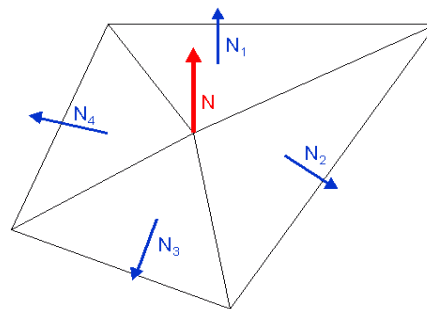


**Abbildung 2.1:** Dreieck, aufgespannt durch die Vektoren  $\vec{a}$ ,  $\vec{b}$ , zusammen mit dem zugehörigen Normalenvektor  $\vec{N}$ .

Das sogenannte *Flat Shading* oder *Constant Shading*. Dabei muss die Beleuchtungsgleichung nur einmal pro Polygon berechnet werden, was zu einem konstanten Farbwert über die gesamte durch das Polygon aufgespannte Fläche führt.

### 2.3 Gouraud Shading

Für realistischere Resultate muss man die Form und die Krümmung der approximierten Fläche berücksichtigen. Beim Gouraud Shading wird die Schattierung nicht mit den Flächennormalen berechnet, sondern es werden Eckpunktintensitäten über die Fläche interpoliert. Das bedingt die Berechnung der Eckpunktnormalen, welche der Mittelung der Normalen der angrenzenden Flächen entsprechen.



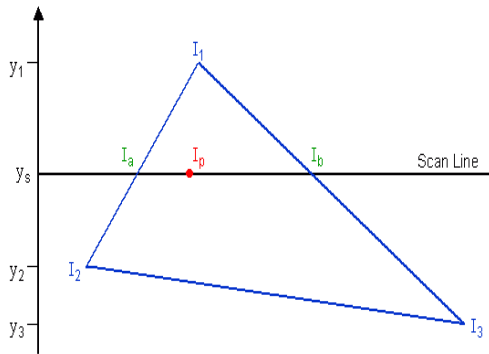
**Abbildung 2.2:** Eckpunkt- und Flächennormalen.

Das Gouraud Shading erfolgt in vier Schritten:

- Berechnung der Flächennormalen aller Flächen
- Berechnung der Normalen  $\vec{N}_V$  zu jedem Vertex  $\vec{V}$  als

$$\vec{N}_V = \frac{\sum_{i=1}^n \vec{N}_i}{\left| \sum_{i=1}^n \vec{N}_i \right|}$$

- Berechnung des gewünschten Beleuchtungsmodells an jedem Eckpunkt
- Bilineares Interpolieren der Intensität  $I_P$  gemäss Abbildung 2.3 für jeden diskreten Punkt  $P$  des Polygons



$$I_a = I_1 - (I_1 - I_2) \cdot \frac{y_1 - y_S}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_3) \cdot \frac{y_1 - y_S}{y_1 - y_3}$$

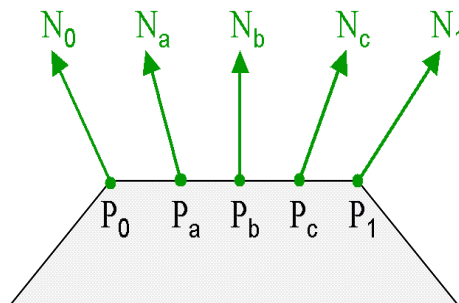
$$I_P = I_b - (I_b - I_a) \cdot \frac{x_b - x_P}{x_b - x_a}$$

**Abbildung 2.3:** Interpolation der Intensität entlang der Polygonkanten und Scan-Linien.

Obwohl Gouraud Shading oft verwendet wird und auch häufig in Hardware realisiert ist, mangelt das Verfahren hauptsächlich in zwei Punkten: Es liefert speziell im Inneren inkorrekte Glanzpunkte, und die abrupten Intensitätswechsel, die beim Flat Shading auftreten, werden zwar abgeschwächt, jedoch nicht ganz eliminiert. Ausserdem treten bei relativ zur Pixelgrösse sehr grossen Polygonen Interpolationsartefakte auf und spekulare Effekte sind nicht darstellbar.

## 2.4 Phong Shading

Im Gegensatz zur Interpolation von Intensitäten werden beim Phong Shading die Flächennormalen interpoliert. Dazu wird an jedem Punkt  $P$  entlang einer Polygonkante die Flächennormale interpoliert und damit das entsprechende Beleuchtungsmodell ausgewertet. Dies ergibt eine genauere Approximation der Oberfläche.



**Abbildung 2.4:** Interpolation des Normalenvektors beim Phong-Shading.

Sind die Flächennormalen nicht explizit gegeben, so kann die Eckpunktnormale über die Vektorprodukte aller anliegenden Kantenvektoren durch Mittelung berechnet werden.



# 3

## Prototyp eines Warpers mit OpenGL

Dieses Kapitel beginnt mit einer kurzen Übersicht über die OpenGL API. Weil diese Graphikbibliothek sehr weit verbreitet und damit auch bekannt ist, werden hier keine technischen Details geliefert. Es werden einzig Gründe aufgezeigt, weshalb sich die Arbeit mit OpenGL lohnt. Wichtiger ist die daran anschliessende Beschreibung der Implementation des Warping-Prototypen in OpenGL.

### 3.1 OpenGL

*OpenGL* [8] ist die erste Adresse, wenn es um die Entwicklung interaktiven 2D und 3D Graphikanwendungen geht. Seit seiner Einführung 1992 wurde OpenGL die am meisten benutzte und unterstützte API für die Computergraphik. Es gibt tausende von damit entwickelten Applikationen auf einer Mehrheit aller Computersysteme. OpenGL fördert Innovation und beschleunigt die Applikationsentwicklung durch das zur Verfügung stellen einer grossen Palette von Funktionen für Rendering, Texture Mapping, Spezialeffekte und andere Visualisierungsmechanismen.

#### 3.1.1 Vorteile für Entwickler

OpenGL bietet diverse Vorteile für Entwickler von graphischen Anwendungen:

- **Industriestandard**  
Ein unabhängiges Konsortium, das *OpenGL Architecture Review Board*, definiert und kontrolliert die OpenGL-Spezifikation. Dank seiner breiten Unterstützung in der Industrie ist OpenGL der einzige wahre offene, herstellerunabhängige Graphikstandard für diverse Plattformen.
- **Stabilität**  
OpenGL ist bereits seit fast zehn Jahren für eine Vielzahl von Plattformen erhältlich. Erweiterungen der Spezifikation werden kontrolliert und Updates genügend früh publiziert, damit die Entwickler Änderungen an ihren Applikationen anbringen können. Dank Rückwärtskompatibilität ist das Funktionieren älterer Anwendungen garantiert.

- **Verlässlichkeit und Portabilität**  
Alle OpenGL-Anwendungen produzieren eine konsistente Bildschirmausgabe auf sämtlicher die OpenGL API unterstützender Hardware, unabhängig von Betriebssystem und Fenstermanager.
- **Erweiterbarkeit**  
Dank seinem durchdachten und zukunftsgerichteten Design erlaubt OpenGL den Zugriff auf neue Hardwareentwicklungen über den OpenGL Erweiterungsmechanismus in der API.
- **Skalierbarkeit**  
OpenGL API-basierte Anwendungen können auf einem breiten Spektrum von Systemen betrieben werden: Von PCs über Workstations bis Supercomputer. Deshalb können Applikationen auf die jeweilig vom Entwickler gewählte Zielumgebung skaliert werden.
- **Einfache Benutzung**  
OpenGL besitzt ein intuitives Design. Effiziente OpenGL-Routinen benötigen typischerweise weniger Programmcode als vergleichbare Graphikbibliotheken. Zusätzlich kapseln die OpenGL-Treiber die darunterliegende Hardware, sodass sich der Applikationsentwickler nicht um technische Details kümmern muss.
- **Gute Dokumentation**  
Es wurde eine Vielzahl Bücher über OpenGL publiziert (zum Beispiel [5]). Zusätzlich ist eine grosse Anzahl an Quellcodes frei erhältlich, was die Benutzung von OpenGL einfach und billig macht.

### 3.1.2 API Hierarchie

Wie in Abbildung 3.1 für Unix und Windows gezeigt, ist die OpenGL Bibliothek in drei Hierarchiestufen aufgeteilt. Zuoberst befindet sich das Anwendungsprogramm. Dieses bekommt vom Fenstermanager (Xlib, GDU) Ereignismeldungen (Maus, Tastatur), kann aber auch Einfluss auf die Darstellung der Fenster nehmen. Soll eine graphische Ausgabe innerhalb eines Fensters getätigt werden, so wird der zugehörige Auftrag an OpenGL übergeben. Der Zugriff kann entweder über GLU, welches als Schnittstelle zu OpenGL Quadrics, NURBS, komplexe Polygone, Matrix-Operationen und noch mehr unterstützt, geschehen, oder direkt über OpenGL. Letztere Form entspricht dem Normalfall.

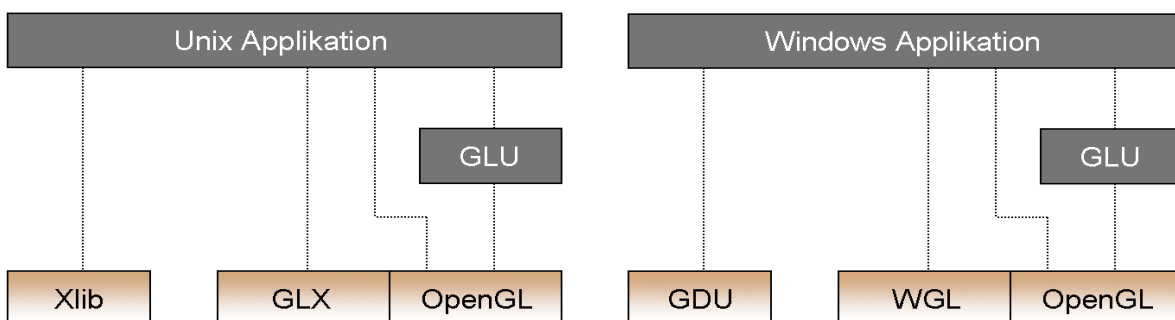
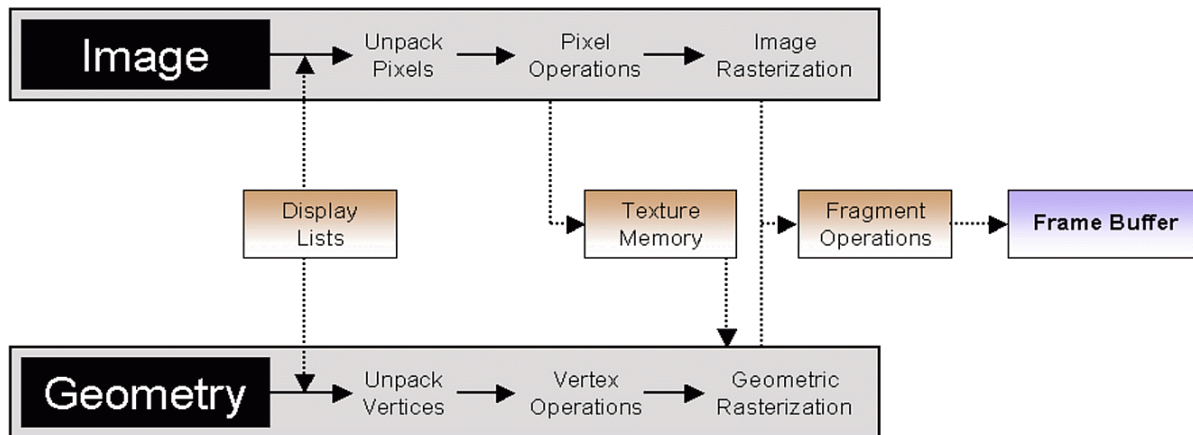


Abbildung 3.1: API Hierarchie in OpenGL für Unix und Windows.

### 3.1.3 Pipeline

Viele OpenGL-Methoden werden spezifisch für Punkte, Linien, Polygone und Bitmaps verwendet, andere Methoden kontrollieren die Ausgabe, zum Beispiel das Aktivieren von Antia-

lasing oder Texturen. Eine letzte Art von Methoden greift direkt auf den Framebuffer zu und manipuliert denselben.



**Abbildung 3.2:** Verarbeitungs pipelines für Bilder und Geometriedaten in OpenGL.

In Abbildung 3.2 werden die beiden Graphik-Pipelines für Bilder und Geometrien gezeigt. Bevor die Daten mehrere Bearbeitungsstufen durchlaufen, werden sie aus der *Display List* entnommen, die alle darzustellenden Primitiven enthält. In beiden Pipelines werden die Primitiven anschliessend interpretiert und in die Bestandteile aufgeschlüsselt. Als nächster Schritt werden Operationen angewandt, bevor die Daten rasterisiert werden, um sie stückweise an den Framebuffer zu senden.

## 3.2 Implementierung

Nach einer kurzen Übersicht über OpenGL wird im Folgenden der unter OpenGL entwickelte Warping-Prototyp beschrieben. Als Programmiersprache wurde C++ ([12] liefert eine gute Einführung) verwendet.

### 3.2.1 Normalenberechnung

Weil der gesamte aus [13] vorhandene Code in C++ implementiert ist, wurde auch für die Normalen eine eigene Klasse `calcNormal` kreiert.

```

typedef MgcVector3 sNormal;

class calcNormal
{ public : calcNormal(PlanarView *_v,float *_Matr,int w,int h);
        ~calcNormal();
        sNormal *getNormal(int x,int y);
        sNormal *getNormal(int index);
        sNormal *getNormal(int index,sNormal *adr)
        int getNumNormals();
}
  
```

Als Parameter werden dem Constructor der Klasse die aktuelle View, ein Zeiger auf die zweidimensionale Matrix mit den Tiefendaten der Punkte des darzustellenden, gewarpten Bildes sowie Grössenangaben (Höhe, Breite) des Bildes übergeben. Die Schnittstelle ist ausserordentlich eng gehalten und stellt nur gerade die Methode `getNormal` zum Abfragen der Nor-

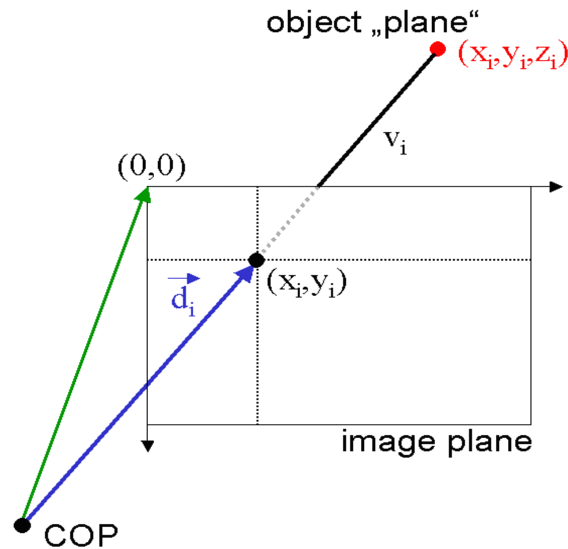
malen in drei Varianten und eine Methode `getNumNormals`, die die Anzahl vorhandener Normalen zurückgibt, zur Verfügung. Diese Anzahl entspricht der Anzahl gewarppter Punkte.

Bereits beim Aufruf des Constructors werden sämtliche Berechnungen ausgeführt und die berechneten Normalen in einer Matrix gespeichert. In einem ersten Schritt müssen die Koordinaten aller gewarppter Punkte im Raum berechnet werden. Das kann mittels einer Koordinatentransformation realisiert werden.

In Gleichung 3.1 wird gezeigt, wie vom *COP* (*Center of Projection*) über den Vektor  $\vec{d}_i$  und die Tiefenangabe  $v_i$  aus der Tiefenmatrix die Koordinaten des Punktes  $P_i = (x, y, z)$  im Objektraum berechnet werden können:

$$\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} = COP + \vec{d}_i \cdot \frac{|\vec{d}_i| + v_i}{|\vec{d}_i|} \quad (3.1)$$

Diese Berechnung wird der Verständlichkeit halber in Abbildung 3.3 dargestellt:



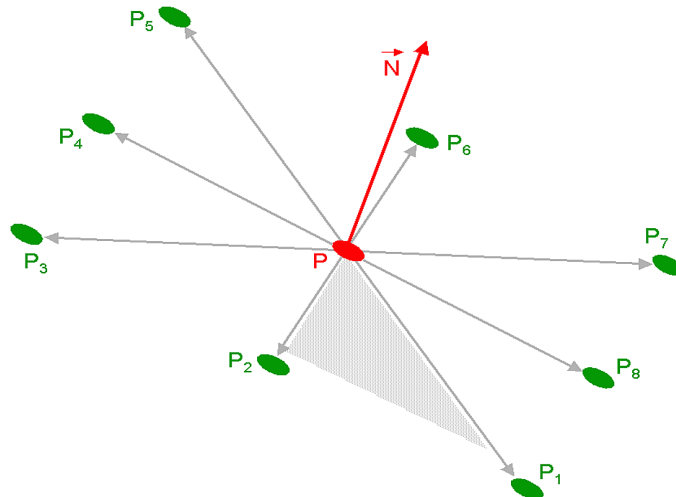
**Abbildung 3.3:** Berechnung der Koordinaten des Punktes  $P_i$  mit den Koordinaten  $(x, y, z)$  im Objektraum.

Der Vektor  $\vec{d}_i$  ist berechenbar, weil Ausrichtung und Position des gewarppten Bildes (*image plane*) im Raum sowie die Grösse desselben bekannt sind.

Sobald alle Punktkoordinaten im Objektraum bekannt sind, kann mit der eigentlichen Normalenberechnung wie in Kapitel 2 beschrieben begonnen werden. Dazu werden sämtliche Nachbarpunkte eines Punktes  $P$  betrachtet und wie in Abbildung 3.4 gezeigt Dreiecke gebildet.

Für jedes Dreieck  $i$  wird nach Gleichung 3.2 eine Flächennormale  $\vec{N}_i$  berechnet, welche zur Normalen  $\vec{N}$  gemittelt werden:

$$\vec{N} = \frac{\sum_i \vec{N}_i}{\left| \sum_i \vec{N}_i \right|} \quad (3.2)$$



**Abbildung 3.4:** Berechnung der Normalen eines Punktes unter Berücksichtigung der Punkte in der Nachbarschaft.

Unbedingt zu beachten ist dabei, dass nicht jeder Punkt in der Tiefenmatrix vorhanden sein muss. Deshalb können an den Objektgrenzen nicht acht Dreiecke respektive Flächennormalen gebildet werden.

Es stellt sich noch die Frage, wann die Normalen berechnet werden sollen. Es gibt zwei Möglichkeiten dafür:

- Nach jedem Warping haben sich Position und Ausrichtung der Punkte im Raum geändert. Deshalb sollen nach jedem Warping-Vorgang alle Normalen neu berechnet werden.
- Beim Warping eines einzigen Bildes wird ausschliesslich mit den Punkten desselben gearbeitet. Daher ist es möglich, bereits zu Beginn die Normalen der Punkte zu berechnen und beim Warping-Vorgang für die benötigte Position und Ausrichtung im Raum zu transformieren.

Bei mehreren Referenzbildern kann diese Methode sehr aufwendig werden, falls die Farben der Punkte anhand mehrere Bilder, auf denen sie sichtbar sind, berechnet werden. In diesem Fall müssen auch die Normalen der einzelnen Bilder miteinander verrechnet werden. Einfacher kann dieses Problem mit Z-Buffering gelöst werden, was keinen zusätzlichen Aufwand bedeutet.

Bei dieser Fragestellung ist natürlich zu bedenken, wie oft sich die Bilddaten ändern, also die Normalen auf jeden Fall neu berechnet werden müssen. In einem Echtzeitsystem wie der Zielplattform *blue-c* werden ununterbrochen neue Bilder akquiriert, weshalb es mit grosser Wahrscheinlichkeit nicht möglich sein wird, den Mehraufwand für die initiale Berechnung aller Normalen aller Referenzbilder später zu amortisieren. Aus diesem Grund wurde im Prototypen die erste Variante gewählt, wenn auch alle zur Verfügung stehenden Testdaten statische Szenen sind.

### 3.2.2 Integration der Normalen in die Szene

OpenGL benötigt die Normalen für die Berechnung der Beleuchtung bei der Ausgabe der Bilddaten. Deshalb war es einfach, die Normalen in die Szene des Prototypen zu integrieren. Wann immer ein Punkt ausgegeben wird, wird auch gleich seine Normale mittels `glNormal3f` ausgegeben. Den Rest erledigt, eine geeignete Beleuchtung (siehe nächster Abschnitt) vorausgesetzt, OpenGL selbständig.

### **3.2.3 Lichtquelle**

Damit eine Schattierung auch wirklich sichtbar wird, ist es notwendig, eine oder mehrere Lichtquellen in die Szene einzufügen. Ein Lichtquelle besitzt eine Position im Raum, eine Richtung, einen Strahlungskonus, ambiente, diffuse und spekulare Intensität, sowie eine Farbe.

Im Prototypen wurde eine weisse Beleuchtung verwendet. Zusätzlich wird die Möglichkeit geboten, diese Lichtquelle mit der Maus um das gewarpte Objekt herum zu bewegen. Dadurch können verschiedenste Beleuchtungssituationen (von vorne, von der Seite, von hinten) interaktiv simuliert werden.

# 4

## Prototyp eines Warpens mit OpenGL Performer

Dieses Kapitel gibt eine grobe Übersicht über OpenGL Performer. Um den Rahmen einer Semesterarbeit nicht zu sprengen, wird die Beschreibung auf die für die Implementierung des Prototypen benötigte Funktionalität beschränkt. Für weitergehende Informationen wird auf [10] und [11] verwiesen.

Im Anschluss an die Beschreibung von OpenGL Performer wird eine ebenso kurze Einführung in die *blue-c* API gegeben, auf welcher künftige Applikationen für die virtuelle Umgebung der *blue-c* entwickelt werden, bevor schliesslich eine Beschreibung des Prototypen folgt.

### 4.1 OpenGL Performer

OpenGL Performer ist ein mächtiges und umfassendes Programmierinterface für Entwickler von graphischen Echtzeitsimulationen und anderen hochleistungsorientierten, dreidimensionalen graphischen Anwendungen. Es vereinfacht die Entwicklung von komplexen Applikationen für Simulationen, Virtuelle Realitäten, interaktive Unterhaltungsformen, CAD und so weiter.

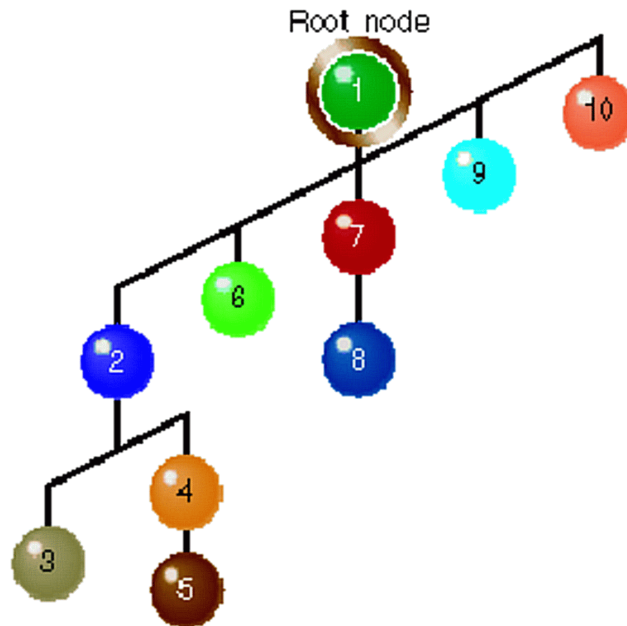
Die OpenGL Performer API wurde als Layer über die OpenGL Graphikbibliothek gebaut und ist ein fest in das Onyx 3000 Visualisierungssystem und in die Visual Workstations von SGI eingebauter Bestandteil. Sie ist kompatibel zu allen graphischen Plattformen von SGI, welche unter den Betriebssystemen IRIX oder Linux laufen, und erreicht auf beiden maximale Performanz. Dadurch ist die API eine flexible, intuitive, toolkit-basierte Plattform für alle Entwickler, die hochperformante Lösungen auf SGI Systemen oder PC Systemen unter Linux entwickeln möchten.

#### 4.1.1 Szenengraph

Das zentrale Element jeder graphischer Ausgabe mit OpenGL Performer ist der *Szenengraph* (siehe Abbildung 4.1). Alle zu rendernden Elemente einer Szene müssen als Knotenobjekte in den Szenengraphen eingefügt werden. Durch die Reihenfolge des Einfügens wird eine Hierarchie der Knoten bestimmt, welche die Reihenfolge der Abarbeitung durch die Rendering-Engine bestimmt.

Die Traversierungsordnung für den Baum ist die Tiefensuche. Dies ermöglicht das Erstellen von Objekten, welche je nach darzustellender Grösse verschiedene Auflösungsstufen ausgeben

können. Ist die benötigte Stufe erreicht, so kann eine weiter Verfolgung des Astes in die Tiefe gestoppt werden.



**Abbildung 4.1:** Darstellung eines Szenengraphen und der enthaltenen Knotenhierarchie (aus [10]).

Jedes Element im Szenengraphen ist ein Knoten. OpenGL Performer stellt eine grosse Bibliothek von verschiedenen Knotenklassen zur Verfügung, welche beliebig erweiterbar sind. Die folgende Auflistung gibt einen groben Überblick, was alles in den Szenengraphen eingefügt werden kann:

- `pfScene`-Objekt als Wurzel des Szenengraphen stellt die Schnittstelle zwischen der Applikation und der Szene dar. Über dieses Objekt wird auf die in der Szene enthaltenen Objekte zugegriffen.
- `pfGroup`-Objekte ermöglichen, mehrere Knotenobjekte zusammenzufassen. Dadurch kann ein Knoten mit beliebig vielen Kindern realisiert werden.
- Der eigentliche Knotentyp für darzustellende Szenenelemente ist `pfGeode`. Ein solcher Knoten beschreibt die Geometrie eines Objektes inklusive einer Bounding Box. Solche Elemente können beliebig komplex aufgebaut sein (z.B. Fahrzeug in einer Autosimulation).
- Lichtquellen werden ebenfalls in den Szenengraphen eingehängt. Anhand der Traversierungsordnung und der Position der Lichtquelle innerhalb des Baumes kann bestimmt werden, welche Knoten beleuchtet werden sollen. Normalerweise wird man aber bedacht sein, dass die Lichtquellen die höchste Priorität aller Knoten haben und folglich die gesamte Szene beleuchten.
- Wie bereits oben erwähnt, ist es möglich, über einen Knoten verschiedene Levels of Detail zu definieren, welche je nach Auflösung gebraucht werden.

- Ein weiterer Knotentyp bestimmt geometrische Transformationen. Dadurch wird eine Transformation der Kinder des Knotens vor der Ausgabe ermöglicht. Mit dieser Funktionalität genügt es zum Beispiel, bei der Darstellung eines Autos ein einziges Rad zu modellieren. Die restlichen drei Räder werden als Kopien des ersten Szeneknoten an die richtige Position in die Szene transformiert.

Mit OpenGL Performer werden noch weitere spezialisierte Knotenklassen mitgeliefert, welche aber in der für diese Semesterarbeit relevanten Programmierarbeit nicht gebraucht wurden und deshalb hier nicht weiter beschrieben werden.

#### 4.1.2 Multi-Process Environment

Eine wichtige Eigenschaft von OpenGL Performer ist, dass es sich um ein Multi-Prozess System handelt. Hauptsächlich drei Prozesse sind in jeder Applikation enthalten:

- Application Process
- Culling Process
- Drawing Process

Der *Application Process* zeichnet sich für die Bearbeitung von Benutzereingaben wie auch das automatische Ändern der Szene verantwortlich. Ein Beispiel wäre die Steuerung der durch den Computer gesteuerten Wagen in einer Rennsimulation.

Im *Culling Process*<sup>1</sup> werden die Bounding Boxes der Knoten im Szenengraph abgefragt. Mit dieser Information wird entschieden, welche Objekte sich in der aktuellen Ansicht befinden und deshalb ausgegeben werden müssen. Die Aufgabe dieses Prozesses kann damit als Optimierung der graphischen Ausgabe betrachtet werden, da es keinen Sinn machen würde, wertvolle Rechenzeit für die Visualisierung von nicht sichtbaren Objekten aufzuwenden.

Der *Drawing Process* schliesslich führt die Ausgabe der im Culling Process als sichtbar gekennzeichneten Objekte durch. Dafür durchquert er den Szenengraphen und fordert die sichtbaren Objekte auf, sich in einem definierten Kontext auszugeben.

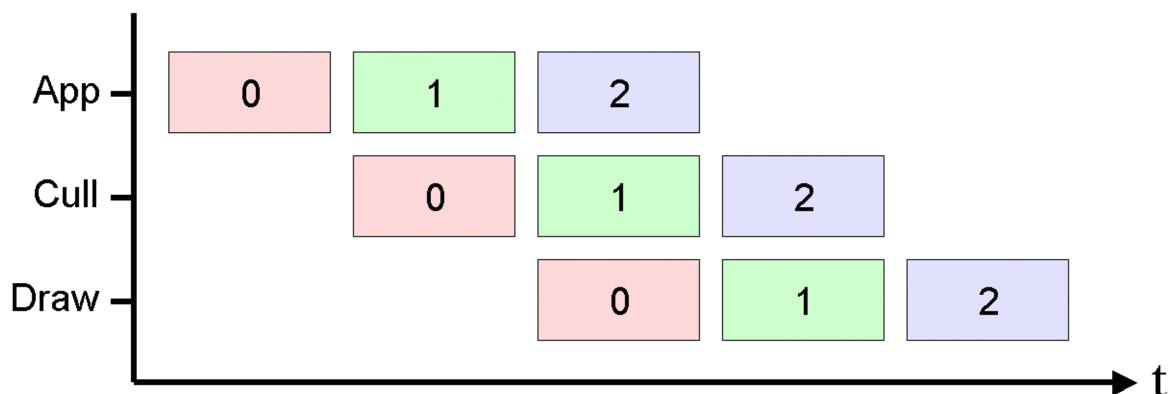


Abbildung 4.2: Abarbeitung der drei Prozesse in der Performer-Pipeline.

Die Prozessausführung wird in einer Pipeline organisiert (siehe 4.2). Zu Beginn wird der Applikationsprozess ausgeführt. Sobald er seine Arbeit getan hat, wird der Culling-Prozess für die Weiterverarbeitung gestartet. Anschliessend wird die Kontrolle an den Drawing-Prozess weitergereicht, der wie beschrieben die Ausgabe übernimmt. Weil es sich beim Prozessmana-

1. engl. für aussuchen, pflücken



Die Problematik besteht darin, dass jeder Prozess seinen eigenen geschützten Speicherbereich besitzt und somit keinen Zugriff auf die Daten der anderen Prozesse hat. Die Lösung ist, dass die Performer-Bibliothek bei der Initialisierung einen globalen Speicherbereich erstellt, auf welchen die einzelnen Prozesse mittels spezialisierten Bibliotheksroutinen zugreifen können.

Die sogenannte *Shared Memory Arena* (siehe 4.3) ist also ein Topf, in dem sich jeder Prozess die benötigten Daten holen kann. Voraussetzung ist, dass die einzelnen Prozesse derart programmiert sind, dass sie wirklich alle benötigten Daten in der Shared Memory Arena ablegen. In der Praxis hat sich gezeigt, dass dies sehr fehleranfällig und schwierig zu debuggen ist, weil manche scheinbar lokale Variablen im globalen Kontext eine Rolle spielen.

Damit alle Objekte in einer Applikation im globalen Datenpool gespeichert sind, müssen sie von der Performer-Klasse `pObject` abgeleitet sein. Diese Objekte werden immer in der Shared Memory Arena gehalten.

## 4.2 *blue-c* API

In einem Unterprojekt des *blue-c* Projektes [4] wird momentan eine Programmierbibliothek *blue-c* API entwickelt, welche es als weiterer Layer über dem OpenGL Performer ermöglicht, mit wenig Aufwand Anwendungen für die virtuelle Umgebung des *blue-c* Projektes zu erstellen. Für die Implementierung des Prototypen eines Warpers in der Performer-Umgebung wurde allerdings nur ein kleiner Teil der besagten API verwendet.

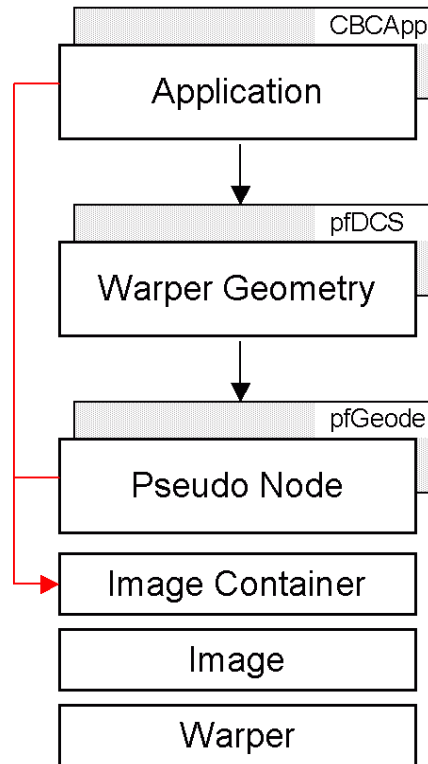
Die Applikationsklasse `CBCApp` bildet die Grundlage für eine in das *blue-c* System integrierte Anwendung. Die vererbten Routinen starten automatisch die benötigten Ausgabekanäle, welche frei konfigurierbar sind. Es werden ausserdem Funktionen für den Zugriff auf den Szenengraphen und die Shared Memory Arena zur Verfügung gestellt. Wichtig sind auch die während den diversen Prozessphasen aufgerufenen Callback-Methoden `PostMPInit` für die Initialisierung der Applikation nach dem Start der *blue-c* Umgebung, `FrameUpdate` für die Berechnung der neuen Szene und `OnMessage` für die Behandlung von Benutzereingaben. Die weiteren Callback-Methoden wurden hier nicht verwendet.

Diverse Ereignissklassen ermöglichen eine einfache klassenorientierte Evaluierung der Art des eingetroffenen Ereignisses. Es stehen Klassen für Maus- und Tastatureingaben zur Auswahl.

## 4.3 Implementierung

Zum Schluss soll jetzt noch ein Überblick über die Funktionsweise des Prototypen Performer-Warpers gegeben werden. Die Aufgabe bestand darin, den im dritten Kapitel beschriebenen OpenGL-Prototypen unter Benutzung der *blue-c* API in das Performer-Environment zu portieren. Dazu war es notwendig, einen Szenengraphen zu implementieren und die bereits vorhandenen Objekte in diesen zu integrieren. Der zweite wichtige Aspekt war die Aufteilung der Applikationslogik auf mehrere Prozesse.

In 4.4 wird der Informationsfluss innerhalb des Prototypen gezeigt. Das von der Klasse `CBCApp` abgeleitete Applikationsobjekt generiert während der Initialisierung einen Ast bestehend aus einem Geometrie- und einem Pseudoknoten, welche beide in den Szenengraphen eingehängt werden. Die Aufgabe des Geometrieknotens ist, der Rendering-Engine anzuzeigen, dass er ein darzustellendes Objekt repräsentiert. Der Pseudoknoten ist der Repräsentant der gewarperten Punktwolke, also der Integrator der punktbasierten Daten in die geometrische Welt. Weiter implementiert das Applikationsobjekt die Behandlungsroutine für jegliche auftretende



**Abbildung 4.4:** Datenfluss in der Prototyp-Anwendung.

Ereignisse wie Mausklicks oder Tastatureingaben. Erfolgt ein gültiges Kommando mit Auswirkung auf die Szene, so wird die entsprechende Funktionalität im ImageContainer aufgerufen.

Man kann den Pseudoknoten in der bestehenden Implementierung aber auch als Schnittstelle zum alten Prototypen verstehen. Er implementiert ausschliesslich den Culling- und Drawing-Callback, welche von den entsprechenden Prozessen aufgerufen werden. Dabei werden die darunterliegenden Funktionen des ImageContainer-Objektes aufgerufen.

# 5

## Resultate

Nachdem eine Methode für das Re-Shading von Punktwolken sowie die beiden implementierten Prototypen beschrieben wurden, gilt es nun, die erreichten Ergebnisse zu bewerten und kritisch zu hinterfragen. Im Anschluss daran werden Möglichkeiten aufgeführt, wie insbesondere die Performanz verbessert werden kann.

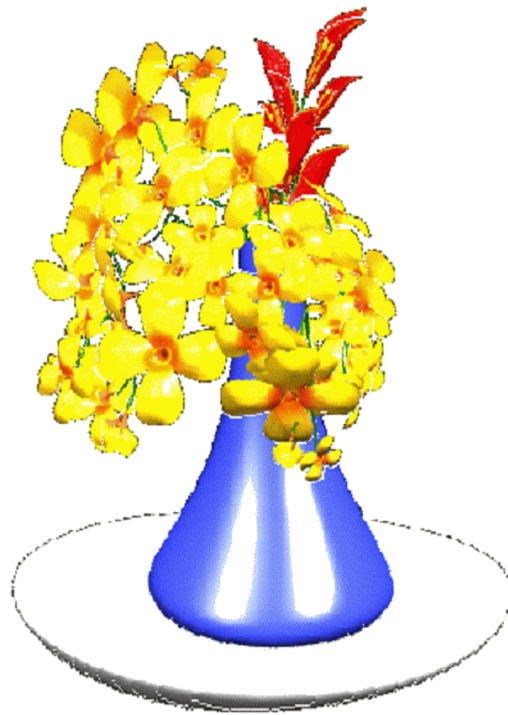
### 5.1 Visueller Eindruck

Bei der Betrachtung der Abbildungen in diesem Kapitel ergibt sich der subjektive Eindruck, dass diejenigen Bilder, auf welche die Schattierung angewandt wurde, realistischer in die Szene integriert erscheinen als die Referenzbilder ohne Re-Shading. Alleine durch diese Verbesserung gegenüber dem alten Resultat sollte der zusätzliche Aufwand für die Berechnung der Normalen gerechtfertigt sein.

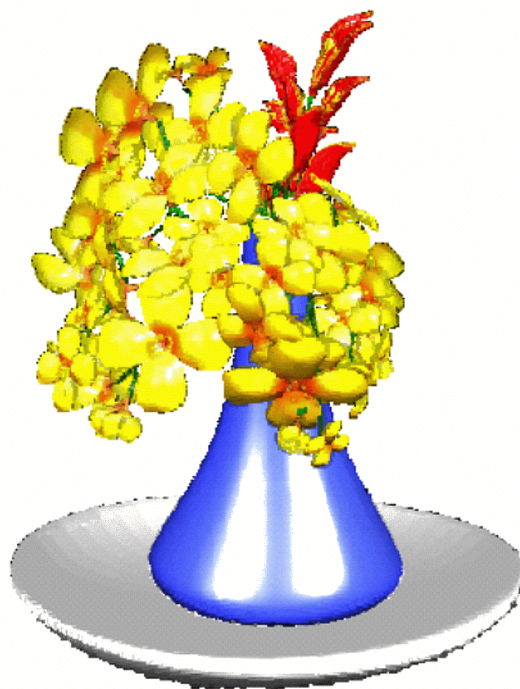
Abbildung 5.3 zeigt insbesondere, dass es mit der angewandten Methode möglich ist, die Textur eines bildbasierten Objektes in unbeleuchteten Bereichen abzdunkeln. Weniger stark fällt die Aufhellung einer stark beleuchteten Textur in Abbildung 5.2 im Vergleich zum Referenzbild auf. Da aber die Szene im *blue-c* Projekt während der Akquisition der Bilddaten durch Video-Kameras sehr stark ausgeleuchtet ist, entsteht durch die schwache Aufhellung kein Problem. Ganz im Gegenteil sind alle Bilddaten bereits sehr stark beleuchtet und müssen deshalb ausschliesslich schattiert werden. Damit scheint das Ziel, die bildbasierte und die geometriebasierte Welt zu verbinden, erreicht. Es ist möglich, Punktobjekte subjektiv richtig zu beleuchten und damit den Umgebungsbedingungen in der virtuellen Welt anzupassen.

### 5.2 Performanz

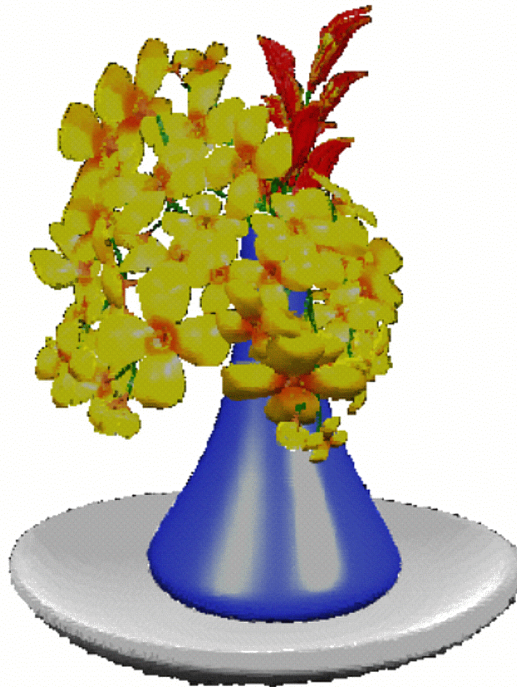
Die Betrachtung der Performanz-Messungen für die OpenGL-Implementierung mit eingeschaltetem Splatting und Shading auf einer SGI Octane lassen Ernüchterung aufkommen. Der Prototyp schafft beim in den Abbildungen 5.1 bis 5.3 gezeigten Modell mit rund 80'000 Punkten nur gerade 1.8fps. Eine solche Leistung ist für ein Echtzeitsystem, wie es die *blue-c* sein soll, völlig unbrauchbar.



**Abbildung 5.1:** Blumenvase ohne Schattierung.



**Abbildung 5.2:** Vase mit Schattierung, Beleuchtung von vorne.



**Abbildung 5.3:** Vase mit Schattierung, Beleuchtung von hinten, woraus eine Verdunkelung der Szene resultiert.

Trotzdem ist dieses Resultat nicht wirklich repräsentativ. Für diese Aussage gibt es gleich mehrere Begründungen:

- Im Rahmen einer Semesterarbeit war es aus zeitlichen Gründen nicht möglich, die Implementation von Stefan Hösli [13] neu zu schreiben. Aus diesem Grund wurde ein Design gewählt, welches die für das Shading benötigten Normalen losgelöst vom Warping-Algorithmus berechnet. Als Folge dieses Ansatzes werden viele Berechnungen, insbesondere auch rechenintensive Koordinatentransformationen, mehrfach ausgeführt, anstatt die bestehenden Resultate wieder zu verwenden.
- Insbesondere im Prototypen in OpenGL Performer wird mit diversen unterschiedlichen Koordinatensystemen gearbeitet (Modellkoordinaten, Warperkoordinaten, Performerkoordinaten), was einen unbezifferbaren Mehraufwand an rechenintensiven Matrixoperationen bedeutet.
- Die überaus performante Möglichkeit von OpenGL, Vertices inklusive Farbe und Normalen in Arrays mit einem einzigen Prozeduraufruf zu übergeben, kann wegen dem Splatting nicht benutzt werden. Der Grund liegt in der Art der Datenrepräsentation. Im Prototyp werden die Splats als Kreisflächen eines bestimmten Umfangs ausgegeben. Diese Flächen sind aber nicht geometrisch im dreidimensionalen Raum definiert, sondern nur als Grösse der zu zeichnenden Punkte. OpenGL unterstützt keine Arrays für Punktgrössen.

Es war nie ein Ziel, die entwickelten Prototypen direkt für das *blue-c* Projekt zu verwenden. Vielmehr galt es zu zeigen, dass das Einfügen von Punktobjekten in die Geometriewelt inklusive richtiger Schattierung funktioniert, und eine Beispielanwendung für OpenGL Performer zu implementieren. In diesem Sinne ist das Ziel der Semesterarbeit erreicht.

Weil das erklärte Ziel des *blue-c* Projektes ist, die virtuelle Umgebung mittels preisgünstiger Standardhardware zu realisieren, müssen diese Resultate dennoch Sorgen bereiten. Es wird niemals möglich sein, eine virtuelle Repräsentation eines Menschen mit weniger als 100'000 Punkten realistisch darzustellen.

Aus diesem Grund wird es notwendig sein, einen immensen Aufwand in die Optimierung und enge Integration der Waring- und Shading-Algorithmen zu stecken. Im folgenden Abschnitt werden einige Ideen gegeben, wie ein solcher Effort Erfolg zeigen könnte.

### 5.3 Optimierungen

Um insbesondere die Performanz der Algorithmen zu verbessern, werden in diesem Abschnitt einige Optimierungs-Möglichkeiten vorgeschlagen.

#### 5.3.1 Koordinatensysteme

Wie bereits im Abschnitt 5.2 beschrieben, werden im Performer-Prototyp verschiedene Koordinatensysteme benutzt. Das erste solche System ist dasjenige in den mit Alias gerenderten Testdaten. Diese Koordinaten werden im Waring-Prototypen in das Warper-Koordinatensystem transformiert und schliesslich für die Ausgabe in das Koordinatensystem des OpenGL Performer.

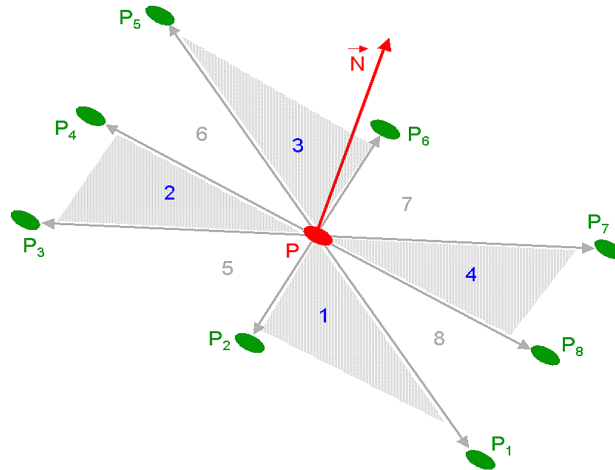
Die *blue-c* API definiert die Koordinaten im Performer auf Metergrössen. Dadurch steht in der Rendering-Engine ein normiertes und intuitives Koordinatensystem zur Verfügung, welches in der Realisierung der *blue-c* unbedingt vom Beginn der Akquisition der Bilder her gebraucht werden soll. Damit wird verhindert, dass Koordinaten transformiert werden müssen. Ein zweiter Vorteil ist der Grössenbereich der benötigten Daten: Alle zu erfassenden Objekte werden in einem Bereich von einigen Millimetern bis etwa zwei Metern liegen, weshalb keine hochgenaue Gleitkommadarstellung benutzt werden muss.

#### 5.3.2 Normalenberechnung

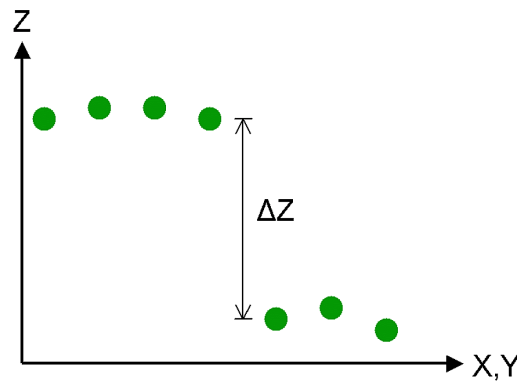
Wie in Kapitel 3 beschrieben, werden in den aktuellen Prototypen jeweils alle Punkte in der Nachbarschaft des gerade betrachteten Punktes in die Normalenberechnung einbezogen (Vergleich Abbildung 3.4). Anhand von Abbildung 5.4 wird eine andere Möglichkeit der Berechnung der Normalen eines Vertices demonstriert.

Um den Aufwand der Normalenberechnung zu minimieren, kann es sich lohnen, nicht alle Normalen der angrenzenden Dreiecksflächen in die Berechnung einzubeziehen. Vielmehr soll eine geschickte Auswahl getroffen werden. Dies kann geschehen, indem man nur maximal eine konstante Anzahl Normalen in die Mittelung einbezieht. Die Wahl der Kandidaten ist natürlich äusserst willkürlich, kann höchstens heuristisch getroffen werden. Lohnenswert ist eine möglichst regelmässige Verteilung der verwendeten Dreiecksflächen rund um den betrachteten Punkt. Dazu werden die Dreiecke in der in Abbildung 5.4 beschriebenen Reihenfolge in die Berechnung einbezogen. Falls mit einem konstanten Faktor 4 gearbeitet wird, deckt das Resultat bei diesem Vorgehen bereits alle Nachbarknoten ab.

Mit der angewandten Brute-Force Methode, einfach alle oder eine bestimmte Anzahl Nachbarpunkte in die Berechnung der Normalen einzubeziehen, besteht ein Randproblem, welches in Abbildung 5.5 dargestellt ist. Da nämlich für jedes Pixel nur eine Farb- und eine Tiefeninformation vorhanden sind, kann nicht garantiert werden, dass zwei Nachbarpixel zur selben Oberfläche des Objektes gehören. Mit diesen Punkten eine Normale zu bilden, führt zu einer falschen Schattierung. In Experimenten hat sich trotzdem gezeigt, dass der Effekt, da er sowieso an den



**Abbildung 5.4:** Reihenfolge für die optimierte Berechnung der Punktnormalen aus den Flächennormalen der mit den Nachbarpunkten gebildeten Dreiecke.



**Abbildung 5.5:** Tiefenunterschied  $\Delta Z$  zwischen zwei Nachbarpunkten, die nicht zur selben Oberfläche gehören.

Rändern von Objekten oder in tiefenkomplexen Objekten auftritt, kaum oder gar nicht sichtbar ist.

Damit ergibt sich ein weiteres Kriterium für die Auswahl der zu verwendenden Dreiecksflächen in der Nachbarschaft: Wann immer der Tiefenunterschied  $\Delta Z$  ( $Z$ -Achse) zwischen zwei Nachbarpixeln einen zu definierenden maximalen Wert  $\Delta Z_{max}$  überschreitet, soll der Nachbarpunkt nicht in die Normalenberechnung einbezogen werden. Dadurch sollte sich aus oben genanntem Grund auch eine bessere Qualität des Re-Shadings ergeben. Man beachte, dass durch das Wegfallen eines Nachbarpunktes gleich zwei Nachbarflächen nicht mehr berücksichtigt werden können.

### 5.3.3 Hardwareunterstützung

Moderne Graphikkarten bieten weit mehr an Funktionalität als die Darstellung von Pixeln. Neueste Modelle wie zum Beispiel die *GeForce 3* von NVIDIA mit der *nFiniteFX Engine* [6] enthalten hochgradig optimierte, programmierbare Prozessoren mit einer Fülle an Funktionen, die im Zusammenhang mit der graphischen Ausgabe von dreidimensionalen Daten häufig benö-

tigt werden. Wann immer möglich, sollte in Applikationen versucht werden, die Unterstützung dieser Graphikbeschleuniger anzunehmen.

Durch geeignete Programmierung solcher Graphikbeschleuniger neuester Generation können die Berechnung von Normalen und die Beleuchtung der Punktobjekte der Hardware überlassen werden, wodurch mehr Rechenleistung für andere Berechnungen zur Verfügung steht.

Es muss aber unbedingt angemerkt werden, dass durch die Benutzung hardwarenaher, proprietärer Funktionalitäten, wie dies bei Graphikbeschleunigern der Fall ist, die Gruppe der möglichen Anwender kleiner wird.

Eine andere Möglichkeit, die mathematischen Funktionen einer Graphikengine auszunutzen, wäre, die Punkte geometrisch zu Dreiecken zu verbinden. Die meisten aktuellen Graphikkarten implementieren Spezialisierungen für die Verarbeitung und insbesondere die Schattierung von Dreiecken.

#### **5.3.4 Glanzpunkte**

Ein grundsätzliches Problem des bildbasierten Rendering sind Glanzpunkte auf dem Objekt. Solche entstehen, wenn die Beleuchtung während der Bildakquisition nicht genügend diffus ist. Mittels spezieller, wahrscheinlich sehr heuristischer Filterung dürfte es möglich sein, solche Glanzpunkte aus den Bilddaten zu entfernen. Ob dies allerdings in Echtzeit und bei sich ununterbrochen ändernder Szene möglich ist, bleibt offen. Im Rahmen dieser Semesterarbeit kann nicht weiter auf diese Problematik eingegangen werden.

# 6

## Zusammenfassung und Ausblick

Im Rahmen des *blue-c* Projektes wird momentan an der ETH Zürich Zentrum ein erster Prototyp eines Portals für die Akquisition und das Rendering der virtuellen Umgebung in Originalgrösse aufgebaut. Nach erfolgreichen Tests im nächsten Jahr wird ein zweites Portal an der ETH Zürich Hönggerberg erstellt werden.

Für ein auf die Position des Benutzers im Portal angepasstes Rendering der virtuellen Szene ist es notwendig, die von den Kameras aufgenommenen Bilddaten mittels 3D Image-Warping zu transformieren. Damit der Benutzer aber auch wirklich in die computergenerierte Welt eintauchen kann, also die virtuelle Welt nicht mehr von der realen Umwelt zu unterscheiden mag, genügt es nicht, die Bilder zu transformieren. Vielmehr ist auch notwendig, die Szene richtig zu beleuchten, wozu die bildbasierte mit der geometriebasierten Welt verbunden werden muss.

In dieser Arbeit wurde gezeigt, wie die Verbindung zwischen den beiden Extrema des Spektrums von bild- und geometriebasiertem Rendering hergestellt werden kann. Damit ist der Weg geebnet, diverse Semester- und Diplomarbeiten zum eigentlichen Prototypen der *blue-c* zu kombinieren.

### 6.1 Ausblick

Ende des aktuellen Jahres soll das erste *blue-c* Portal funktionsfähig sein. Danach werden viele äusserst interessante Applikationen realisierbar sein. Weil ich dann im Computer Graphics Lab meine Diplomarbeit schreiben werde, werde ich die Möglichkeit haben, diesen wichtigen Moment zu erleben, worauf ich mich sehr freue.





## Referenzen

- [1] Computer Graphics Lab, ETH Zürich. *The blue-c Homepage*. <http://blue-c.ethz.ch>, 12.07.2001.
- [2] Leonard McMillan. "An Image-Based Approach to Three-Dimensional Computer Graphics." Ph. D. Thesis, University of North Carolina at Chapel Hill, 1997.
- [3] Markus H. Gross. *Graphische Datenverarbeitung*. Lecture Notes, Computer Science Departement, ETH Zürich, 1999.
- [4] Martin Näf. "The blue-c: Software Integration of Virtual Worlds, Live Media-Streams and Collaboration." ETHZ Ph.D. Proposal, November 2000.
- [5] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner. *OpenGL Programming Guide*. Addison Wesley Publishing Company, Third Edition, 1999.
- [6] NVIDIA Corporation. *NVIDIA nFiniteFX Engine: Programmable Vertex Shaders*. <http://www.nvidia.com/docs/lo/67/SUPP/vertexshaders.pdf>, 11.07.2001.
- [7] Oliver G. Staadt, Martin Näf, Edouard Lamboray, Stephan Würmlin. "JAPE: A Prototyping System for Collaborative Virtual Environments." In *Eurographics 2001*, 2001.
- [8] OpenGL. *The Industry's Foundation for High Performance Graphics*. <http://www.opengl.org>, 12.07.2001.
- [9] Ronald Peikert, Oliver Staadt. *Graphische Datenverarbeitung 2*. Lecture Notes, Computer Science Departement, ETH Zürich, 2001.
- [10] SGI Tech Reports. *OpenGL Performer 2.4 Getting Started Guide*. <http://www.sgi.com/software/performer/manuals.html>, 11.07.2001.
- [11] SGI Tech Reports. *OpenGL Performer 2.4 Programmer's Guide*. <http://www.sgi.com/software/performer/manuals.html>, 11.07.2001.
- [12] Stanley B. Lippman. *Essential C++*. C++ In-Depth Series, 1999.
- [13] Stefan Hösli. "3D Point Warping." Semesterarbeit, Departement Informatik, ETH Zürich, 2000/2001.