

# Redesign von Cgi3D

in Hinblick auf Portierbarkeit und  
Performanceoptimierung

Diplomarbeit, vorgelegt von Bernhard Schwall

Rheinische Friedrich-Wilhelms-Universität Bonn  
Institut für Informatik III

November 1998

# Versicherung

Hiermit versichere ich an Eides Statt, daß ich die vorliegende Arbeit „Redesign von Cgi3D in Hinblick auf Portierbarkeit und Performanceoptimierung“ selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Bonn, den 16. November 1998

Bernhard Schwall

# Inhaltsverzeichnis

<b>Motivation</b> . . . . .	1
<b>1 Übersicht über die vorgefundene Struktur</b> . . . . .	<b>2</b>
1.1 Die Klassenstruktur . . . . .	2
1.2 Funktionalität in t_cgi3d . . . . .	2
1.3 Funktionalität in gh_* . . . . .	2
1.4 Funktionalität der Softwarelösung . . . . .	3
<b>2 Ansätze zum Redesign von Cgi3D</b> . . . . .	<b>4</b>
2.1 Ansätze im Hinblick auf bessere Portierbarkeit . . . . .	4
2.1.1 Hardwareunabhängiges t_Cgi3D . . . . .	4
2.1.2 Hardwareabhängige Funktionen in t_RenderScene . . . . .	4
2.2 Ansätze im Hinblick auf neue Funktionalitäten . . . . .	5
2.3 Ansätze zur Optimierung der Geschwindigkeit . . . . .	5
<b>3 Übersicht über die neue Struktur</b> . . . . .	<b>7</b>
3.1 Funktionalität von t_Cgi3D . . . . .	7
3.2 Funktionalität von t_RenderScene . . . . .	7
<b>4 Aspekte der Softwarelösung</b> . . . . .	<b>9</b>
4.1 Unterschied zwischen full screen und single line Z-Buffer Algorithmus . . . . .	9
4.2 Probleme bei der Integration von single line Z-Buffer in das neue Konzept . . . . .	10
<b>5 Geschwindigkeitsoptimierungen für verschiedene Plattformen</b> . . . . .	<b>11</b>
5.1 Verwendung von Arrays zur Übergabe der Daten an die Hardware . . . . .	11
5.2 Triangle Strip Lists zur Minimierung der Daten . . . . .	12
5.2.1 Verwendeter Algorithmus . . . . .	13
5.2.2 Probleme des verwendeten Algorithmus . . . . .	14
5.3 Array-Optimierung bei Direct3D . . . . .	15
5.3.1 Verwendeter Algorithmus . . . . .	15
5.3.2 Probleme des verwendeten Algorithmus . . . . .	16
5.4 Erzielbare Datenreduktion . . . . .	16
<b>6 Erweiterte Funktionalität der neuen Version</b> . . . . .	<b>18</b>
6.1 area lights . . . . .	18
6.2 3D-Texturen . . . . .	19
6.2.1 Erster, wieder verworfener Algorithmus . . . . .	20
6.2.2 Der zur Zeit implementierte Algorithmus . . . . .	20
6.3 Volume Rendering . . . . .	21
<b>7 Korrektheit der Darstellung im Vergleich zu Raytracing</b> . . . . .	<b>25</b>

<b>8 Aufwand zur Änderung und Portierung</b>	<b>29</b>
8.1 Aufwand bei internen Änderungen, z.B. bei der Datenstruktur	29
8.2 Aufwand zur Portierung auf neue Hardware	29
<b>9 Performancetests</b>	<b>31</b>
9.1 Testplattformen	31
9.2 Ergebnisse und Bewertung	32
9.2.1 Ergebnisse unter Unix/SGI	33
9.2.2 Ergebnisse unter Unix/Linux	35
9.2.3 Ergebnisse unter Windows 95	36
9.3 Vergleich der neuen Treiber untereinander	37
<b>10 Ausgewählte Möglichkeiten zur weiteren Verbesserung</b>	<b>38</b>
10.1 Allgemeine Verbesserungen	38
10.2 OpenGL	39
10.3 Direct3D	40
10.4 XGL	40
10.5 Softwarerenderer	41
<b>11 Ausblick auf neue Versionen der Renderbibliotheken</b>	<b>42</b>
11.1 OpenGL 1.2	42
11.2 DirectX 6.0	42
<b>12 Zusammenfassung der Ergebnisse</b>	<b>45</b>
<b>13 Hinweise zur Portierung auf neue Hardwareplattformen</b>	<b>47</b>
<b>A Unterstützte Renderbibliotheken</b>	<b>49</b>
<b>B Klassenreferenz</b>	<b>52</b>
<b>C Environmentvariable</b>	<b>69</b>
C.1 Tips zur Einstellung der Variablen	70
<b>D CD-ROM Inhalt</b>	<b>72</b>
<b>Literaturverzeichnis</b>	<b>73</b>

# Motivation

Die bisherige Implementation von Cgi3D stammt in ihren Grundzügen aus dem Jahre 1994[Web94]. Sie wurde als low level Grafikschnittstelle zur Ausgabe von dreidimensionalen Objekten des Minimal Rendering Toolkits[Fel94] (kurz MRT) entwickelt. Das MRT ist eine Plattform zur Beschreibung und Darstellung von dreidimensionalen Szenen. Neben der Szenenbeschreibung ermöglicht es die Ausgabe der Szene mittels Raytracing oder Radiosity.

Ziel der ursprünglichen Arbeit war es, die Ausgabe der Szene in Echtzeit plattformunabhängig sowohl mittels Software, als auch bei vorhandener Hardwarebeschleunigung mittels Hardware zu bewerkstelligen. Echtzeit heißt hierbei, daß die Ausgabe eines Bildes anstatt in mehreren Minuten bis Stunden, wie bei Raytracing üblich, innerhalb von Sekundenbruchteilen bzw. weniger Sekunden geschieht.

Mit der Zeit hat sich gezeigt, daß dieser Ansatz zwar von der Idee her in Ordnung war, die Implementierung jedoch einige Schwächen enthielt. Durch den technischen Fortschritt im Bereich der 3D-Hardwarebeschleunigung ist heutzutage die Unterstützung von verschiedenen Hardwareplattformen wichtig geworden. Insbesondere die Unterstützung von Windows 95/NT mit dessen Unterstützung von 3D-Hardware durch Direct3D machte es nötig, die Ausgabe auch unter Direct3D zu ermöglichen. Die Implementierung eines Direct3D Renderers im Rahmen des ursprünglichen Cgi3D war jedoch recht kompliziert.

In dieser Arbeit geht es deshalb darum, die Schwachstellen der alten Implementation von Cgi3D zu erkennen und ein Konzept zu entwickeln, durch das die Portierung von Cgi3D auf eine neue Hardwareplattform erleichtert wird.

Ein weiteres Ziel ist es, die durch neuere Versionen von Renderbibliotheken wie OpenGL 1.1 eingeführten Möglichkeiten zur Beschleunigung der Ausgabe zu unterstützen.

Da durch die zunehmende Leistungsfähigkeit der Grafikhardware eine immer exaktere Ausgabe der Objekte ermöglicht wird, ist der letzte Punkt dieser Arbeit der Abbildungsqualität auf verschiedenen Hardwareplattformen gewidmet. Hierbei wird untersucht, inwieweit die Ausgabe mittels Cgi3D der Berechnung der Szene mittels Raytracing entspricht und wo noch Schwächen sind.

# Kapitel 1

## Übersicht über die vorgefundene Struktur

In diesem Kapitel wird ein Überblick gegeben über die interne Struktur von Cgi3D, so, wie es seit dem Jahre 1994 an der Universität Bonn verwendet und ständig erweitert wird.

### 1.1 Die Klassenstruktur

Cgi3D ist bisher in drei Teile unterteilt: `t_cgi3d.cpp`, `gh_*.cpp` und der Softwarerenderer. Die hardwareabhängigen `gh_*` Dateien bilden einen Teil des Cgi3D. Dies bedeutet, daß eine `gh_*` Datei einen Teil des `t_Cgi3D`-Objektes darstellt und auch auf die privaten Daten von `t_Cgi3D` Zugriff hat. Somit handelt es sich bei den `gh_*` Dateien um einen Teil des `t_Cgi3D`-Objektes. Dies erklärt, warum es nicht möglich ist, im ausführbaren Programm mehrere verschiedene Hardwareplattformen zu unterstützen. Eine Ausnahme bildet hier nur der Softwarerenderer. Dieser ist in mehrere externe Objekte unterteilt, welche aber wiederum sehr eng miteinander verbunden sind.

### 1.2 Funktionalität in `t_cgi3d`

Die high-level Schnittstelle von Cgi3D befindet sich in den Quelltexten `t_cgi3d.*` und `t_cgi3dp.cpi`. Diese implementieren die Routinen, die der Benutzer verwenden kann, sowie einige interne Hilfsfunktionen. Die eigentlichen Aufgaben von Cgi3D, Fenster darstellen und Objekte zeichnen, werden dabei an Cgi, Routinen in `gh_*` Dateien oder den Softwarerenderer weitergeleitet. Routinen in `t_cgi3d.*` und `t_cgi3dp.cpi` selber kümmern sich weder um das Durchlaufen der Szene noch um die Darstellung eines Objektes. Ebenso werden Einstellungen, die der Benutzer ändert, nur in internen Variablen zwischengespeichert, von wo aus der Renderer diese wieder abfragen kann.

### 1.3 Funktionalität in `gh_*`

Für jede unterstützte Hardware existiert eine `gh_*` Datei. Diese ist für die Initialisierung der Hardware, das Anzeigen der Szene und das Picking<sup>1</sup> zuständig. Die `gh_*` Datei implementiert einen Teil des `t_Cgi3D`-Objektes, so daß bei der Compilierung von Cgi3D entschieden werden muß, welche Hardware unterstützt werden soll. Dies gilt auch dann, wenn verschiedene Hardwareunterstützungen auf einer Plattform möglich sind (z.B. XGL und OpenGL).

---

<sup>1</sup>Picking ist das interaktive Auswählen von Objekten oder Polygonen in einer Szene

Jede `gh_*` Datei implementiert im wesentlichen drei Funktionsblöcke:

1. Initialisierung der Hardware. Hierzu zählt hauptsächlich:
  - Test auf Hardwarebeschleunigung
  - Darstellung des Fensters
  - Aktivierung der Hardwarebeschleunigung
  - Setzen des Beleuchtungsmodells
2. Darstellung der Szene. Dies kann unterteilt werden in folgende: Schritte
  - Setzen der Kamera
  - Setzen der Lichter
  - Durchlaufen der Szene
  - Übergabe der einzelnen Objekte in Form von Polygonen an die Hardware
3. Picking

Diese Aufzählung macht deutlich, daß die gesamte Arbeit der Darstellung der Szene in den Routinen der `gh_*` Datei geschieht. Insbesondere kümmern sich die Routinen nicht nur um die eigentliche Ansteuerung der Hardware, sondern erledigen auch das Durchlaufen der Szene und der einzelnen Objekte.

Des weiteren sind die Aufgaben der einzelnen Routinen in den `gh_*` Dateien nicht klar unterteilt. So ist z.B. die Routine `t_Cgi3D::executeScene` für die komplette Darstellung der Szene inklusive Durchlaufen, Zerlegen der Objekte und Übergabe der Polygone an die Hardware verantwortlich. Dadurch wird es sehr schwierig, eine neue Hardware in das bestehende System zu integrieren.

## 1.4 Funktionalität der Softwarelösung

Die Softwarelösung kümmert sich um die Darstellung der Szene, wenn keine dedizierte Hardware vorhanden ist, oder wenn eine Darstellungsoption gewählt wurde, die von der Hardware nicht unterstützt wird (z.B. Phong Shading). Die Softwarelösung ist in vier Klassen unterteilt:

1. `t_ActiveTriangle`
2. `t_Rendering`
3. `t_Illumination`
4. `t_NpcPoint`

Diese Aufteilung dient vor allem dazu, mit Unterstützung des objektorientierten Paradigmas von C++ eine einfache Möglichkeit zu bieten, zwischen verschiedenen Arten der Beleuchtung und Schattierung zu wechseln.

## Kapitel 2

# Ansätze zum Redesign von Cgi3D

In diesem Kapitel werden Ansätze zur Redesign von Cgi3D aufgezeigt, die im Verlauf der Arbeit aufgegriffen und näher erläutert werden.

### 2.1 Ansätze im Hinblick auf bessere Portierbarkeit

Ein Ziel dieser Diplomarbeit ist es, das Design von Cgi3D so zu verändern, daß eine Portierung auf eine neue Hardware einfacher möglich ist. Dazu sollte der hardwareabhängige und hardwareunabhängige Teil strikt getrennt werden. Außerdem sollte es möglich sein, zur Laufzeit eine spezielle Hardwarebeschleunigung auszuwählen.

#### 2.1.1 Hardwareunabhängiges t\_Cgi3D

Neben der Weiterleitung von Funktionsaufrufen an Cgi (Fenster öffnen, Tastaturbehandlung) gehören der Aufruf der Setup-Funktionen und die Auswahl der Hardware zum hardwareunabhängigen Teil von Cgi3D.

Des weiteren gehört systematisch gesehen auch das Durchlaufen der Szene und das Zerlegen der einzelnen Objekte in Polygone zum hardwareunabhängigen Teil von Cgi3D.

Bei der Verwendung von einigen Beschleunigungsmöglichkeiten der Renderer, so z.B. von Triangle Strip Lists, ist es nötig, daß ein Objekt in einer ganz speziellen Reihenfolge durchlaufen und an die Hardware übergeben wird (nähere Erläuterungen hierzu finden sich in Kapitel 5.2).

Bei der Portierung auf IRIS Performer 2.2[Wap99] ergab sich das Problem, daß der Performer den Szenengraphen selbständig optimiert. Somit ist ein Durchlaufen der Szene auf Seiten von Cgi3D nutzlos und verlangsamt die Ausgabe nur unnötig.

Aus den erwähnten Gründen ist es deshalb sinnvoll, sowohl das Durchlaufen der Szene, als auch das Zerlegen der einzelnen Objekte in Polygone in einer Basisklasse für die Hardwareansteuerung zu implementieren. Auf diese Weise können die Routinen, falls benötigt, im Hardwarerenderer überladen werden. Falls dies aber nicht nötig ist, so müssen sie auch nicht für jeden Renderer neu implementiert werden, so daß konzeptionelle Änderungen nur einmal in der Basisklasse vorgenommen werden müssen.

#### 2.1.2 Hardwareabhängige Funktionen in t\_RenderScene

Der hardwareabhängige Teil von Cgi3D ist im Objekt t\_RenderScene gekapselt. Von diesem wird pro unterstützter Hardware ein neues Objekt abgeleitet (z.B. t\_RenderOpenGL oder t\_RenderDirect3D).

In `t_RenderScene` sind folgende hardwareabhängigen Operationen gekapselt:

1. Hardwaresetup
2. Ausgabe der kompletten Szene an die Hardware
3. Ausgabe eines Objektes an die Hardware
4. Picking

Hierbei ist zu unterscheiden zwischen der Ausgabe der kompletten Szene (2) und der Ausgabe eines Objektes (3). Die Ausgabe der Szene durchläuft diese und gibt die Objekte Objektweise an die Ausgaberroutine für einzelne Objekte des Renderers weiter. Dieser zerlegt das Objekt dann in Polygone und gibt diese mit Hilfe der Hardware aus. Eine getrennte Ausgaberroutine für Objekte hat neben der besseren Übersichtlichkeit des Quelltextes vor allem den Vorteil, daß es hiermit möglich ist, Attribute pro Objekt zu setzen. So ist es z.B. möglich, zwischen Flat-Shading und Gouraud-Shading umzuschalten, oder aber, falls die Hardware dies erfordert, für bestimmte Objekte einen speziellen Renderer der Hardware zu verwenden. Die Attributumschaltung ist allerdings im derzeitigen Szenengraphen des MRT nicht vorgesehen.

## 2.2 Ansätze im Hinblick auf neue Funktionalitäten

In den letzten Jahren sind in das MRT einige Neuerungen eingeflossen, deren Nutzung im Zuge des Redesigns von Cgi3D auch mit diesem ermöglicht werden soll. Außerdem wurden einige Erweiterungen angedacht, welche sich in Zukunft als recht nützlich erweisen könnten.

- Durch das Verwenden einer Basisklasse für alle Hardwarerenderer ist die Unterstützung neuer Objekttypen des MRT oder neuer Darstellungsmethoden recht einfach. So mußte z.B. die Simulation von 3D-Texturen mittels 2D-Texturen nur einmal implementiert werden (siehe hierzu Kapitel 6.2). Da die Ausgabe von 2D-Texturen bereits von allen Hardwarerenderern unterstützt wurde, war hierbei direkt die Ausgabe der 3D-Texturen auf allen unterstützten Plattformen möglich.
- Durch die Ausgabe der Szene auf Objektbasis ergeben sich neue Möglichkeiten für eine zeitorientierte Ausgabe der Szene. So ist es denkbar, einen Renderer zu implementieren, der für die Ausgabe einer Szene nur eine gewisse Zeit benötigen darf. Dieser Ansatz wird in Kapitel 10.1 weiter verfolgt.
- Durch die Aufteilung der Szenenwiedergabe in mehrere Blöcke ist es möglich, für jedes ausgegebene Objekt die Attribute wie Schattierung oder Textur zu verändern. Zur Zeit ist dies im Szenengraphen jedoch noch nicht vorgesehen, aber für die Zukunft geplant.

## 2.3 Ansätze zur Optimierung der Geschwindigkeit

Neben dem Redesign von Cgi3D im Hinblick auf einfachere Portierbarkeit ist ein Aspekt dieser Arbeit die Geschwindigkeitsoptimierung der Ausgabe. Hierzu bieten sich die erweiterten Funktionen von OpenGL 1.1 bzw. Direct3D an, als da wären:

- Übergabe der Koordinaten in Form von Arrays anstatt einzeln:  
Hierbei werden die Punktkoordinaten der Polygone zunächst in einem Array gespeichert und dieses Array dann mit einem Aufruf an die Hardware übergeben. Dadurch verringert sich die Anzahl der Aufrufe erheblich, da anstatt von drei Aufrufen pro

Punkt (Koordinate, Normale und Farbe plus evtl. Texturkoordinate) nur ein Aufruf für 200 Punkte (bzw. die Größe des Arrays) stattfinden muß.

- Triangle Strip Lists:

Hierbei werden nicht mehr für jedes Dreieck alle drei Eckpunkte an die Hardware übergeben. Es wird vielmehr versucht, zusammenhängende Dreiecke in Form einer Liste zu übergeben. Dabei bildet die Kante zwischen zwei Eckpunkten des vorangehenden Dreiecks eine Kante des nächsten Dreiecks, so daß für dieses nur noch ein neuer Eckpunkt an die Hardware übergeben werden muß. Dadurch verringert sich die Anzahl der zu übergebenden Punkte an die Hardware von  $3 * N$  auf  $3 + (N - 1)$ , wobei  $N$  die Anzahl der Dreiecke ist. Näheres hierzu findet sich in Kapitel 5.2.

- Vermeidung von Mehrfachinitialisierungen:

Durch das Aufteilen einiger ursprünglicher Funktionen in `set*` und `init*` Funktionen (z.B. `setCamera` und `initCamera`) besteht für den Renderer eine einfache Möglichkeit festzustellen, ob sich das betreffende Objekt seit der letzten Ausgabe der Szene geändert hat (im Beispiel muß die Kamera nicht neu an die Hardware übergeben werden, falls `setCamera` nicht aufgerufen wurde). Dies kann erhebliche Geschwindigkeitsvorteile mit sich bringen, insbesondere bei den Routinen `setScene` und `initScene`, da ohne erneuten Aufruf der Funktion `setScene` dem Renderer direkt bekannt ist, daß sich an der Szene nichts geändert hat. Falls die Hardware dies unterstützt, so kann die (alte) Szene nun direkt aus dem Cache der Hardware ausgegeben werden.

## Kapitel 3

# Übersicht über die neue Struktur

Cgi3D ist in die beiden Objekte `t_Cgi3D` und `t_RenderScene` unterteilt. `t_Cgi3D` enthält keine hardwareabhängigen Aufrufe. Vielmehr beinhaltet es das Interface zwischen dem Benutzer und Cgi3D und leitet die Aufrufe an die korrekte Stelle weiter (z.B. Cgi oder `t_RenderScene`). Weiterhin gehören zu Cgi3D noch die Objekte `t_ActiveTriangle`, `t_NpcPoint`, `t_SoftwareIllumination` und `t_SoftwareRendering`, welche den Software-renderer bilden.

### 3.1 Funktionalität von `t_Cgi3D`

Das Objekt `t_Cgi3D` enthält die Definition von Cgi3D, also den Teil, der dem Benutzer zugänglich ist

Alle Funktionen von Cgi3D, die der Anwender aufrufen kann, werden in `t_Cgi3D` behandelt. Dieses behandelt die Aufrufe entweder selber (z.B. indem es Statusvariablen zwischenspeichert), oder aber es leitet den Aufruf direkt an ein anderes Objekt weiter, z.B. an Cgi oder auch an den hardwareabhängigen Teil von Cgi3D.

Um dem Benutzer eine Abfrage seiner gesetzten Daten zu ermöglichen, werden diese in Statusvariablen zwischengespeichert. Dies hat den Vorteil, daß `t_RenderScene` keine Abfragemöglichkeit für diese Daten enthalten muß. Dadurch ist es möglich, Daten direkt an die Hardware weiterzureichen, sofern dies von der Hardware unterstützt wird. Falls `t_RenderScene` die Daten doch zwischenspeichern muß, so bedeutet dies keinen allzu großen Speicherplatzverlust, da es sich jeweils nur um einfache Integerwerte oder um Pointer handelt. Da diese Pointer Referencepointer[[Fis95](#)] sind, besteht hierbei nicht die Gefahr, daß Daten durch den Anwender gelöscht werden, obwohl `t_RenderScene` diese noch benötigt.

### 3.2 Funktionalität von `t_RenderScene`

Das Objekt `t_RenderScene` bildet die Basisklasse für den hardwareabhängigen Teil von Cgi3D. Es kapselt alle Aufrufe, die direkt auf die Hardware zugreifen:

- Das Hardwaresetup für die komplette Szene. Hierbei werden die Grafikhardware initialisiert und die Kamera und Lichter gesetzt.
- Ein übergebenes Objekt in der gewünschten Beleuchtungsart darstellen. Hierbei wird das Objekt polygonweise durchlaufen und jedes Polygon evtl. in Dreiecke zerlegt. Danach werden diese Dreiecke an die Hardware übergeben (oder im Falle des Softwarerenderers per Software dargestellt).

Außerdem enthält `t_RenderScene` aber auch einige Funktionen, die eigentlich hardwareunabhängig sind. Hierzu zählen:

- Das Durchlaufen der kompletten Szene.
- Das Zerlegen eines Objektes in seine Polygone.

Diese Funktionen wurden bewußt in `t_RenderScene` ausgegliedert, da sie für einige Hardwarebibliotheken überladen werden müssen. So arbeiten einige Bibliotheken auf dem Szenengraphen und optimieren diesen selber (z.B. IRIS Performer). Bei anderen Bibliotheken ist es notwendig, das Objekt in einer ganz speziellen Reihenfolge zu durchlaufen, um die Polygone in eben dieser Reihenfolge an die Hardware zu übergeben (siehe Kapitel 5.2).

Da aber die Grundfunktionalität im Objekt `t_RenderScene` gekapselt ist, ist es recht einfach möglich, von diesem Objekt ein neues abzuleiten und dort die gewünschte Funktion zu überladen. Sollte sich nun an Cgi3D etwas grundlegendes in der Datenstruktur ändern, so muß trotzdem nur das Objekt `t_RenderScene` geändert werden (und natürlich die überladenen Funktionen ebenfalls). Wurde aber für eine Hardware die geänderte Funktion nicht überladen, so muß auch der Quelltext für diese Hardware nicht geändert werden.

Dadurch, daß alle Hardwarerenderer (und natürlich auch der Softwarerenderer) von `t_RenderScene` abgeleitet sind, ist es möglich, zur Laufzeit den gewünschten Renderer auszuwählen. So kann z.B. zwischen einem nicht optimierten und einem optimierten Renderer umgeschaltet werden, ohne das Programm neu übersetzen zu müssen.

# Kapitel 4

## Aspekte der Softwarelösung

Bei der Softwarelösung handelt es sich um eine Anpassung der alten Softwarelösung an die neue Struktur von Cgi3D. Dabei wurden vor allem alle Friend-Abhängigkeiten<sup>1</sup> der Objekte untereinander und von t\_Cgi3D eliminiert, indem die Objekte die benötigten Daten nun bei ihrer Initialisierung erhalten oder aber diese über öffentliche Funktionen von t\_Cgi3D abfragen.

Bei Verwendung des full screen Z-Buffer Algorithmus erfolgt wie auch bei den Hardwarelösungen die Darstellung auf Objektbasis. Somit basiert dieser Renderer auf der selben Basis wie die Hardwarerenderer und unterstützt die selbe Funktionalität wie diese.

### 4.1 Unterschied zwischen full screen und single line Z-Buffer Algorithmus

Bei der Darstellung einer dreidimensionalen Szene auf einem zweidimensionalen Ausgabegerät ist neben der perspektivisch korrekten Abbildung der Punkte auch die Entfernung verdeckter Kanten und Flächen nötig. Dies geschieht bei der Softwarelösung dadurch, daß neben dem eigentlichen Bildspeicher, der das berechnete Bild enthält, ein weiterer Speicher reserviert wird. Dieser Speicher enthält für jeden dargestellten Punkt die Tiefeninformationen im dreidimensionalen Raum und wird Z-Buffer genannt.

Für die Implementierung dieses Z-Buffers gibt es zwei verschiedene Möglichkeiten:

- Es wird für das gesamte Bild ein Z-Buffer angelegt (full screen Z-Buffer).

Der Vorteil ist, daß zu jeder Zeit der Berechnung auf jeden beliebigen Punkt des Z-Buffers zugegriffen werden kann. Somit ist die Reihenfolge der Darstellung der Objekte nicht vorgeschrieben.

Der Nachteil dieses Verfahrens ist, daß für jeden Punkt des Bildspeichers weiterer Speicher für den Z-Buffer benötigt wird. Somit erhöht sich der Speicherbedarf im Vergleich zum Bedarf für das reine Bild erheblich. Bei 32 Bit Genauigkeit für den Z-Buffer werden 3 Byte für die eigentliche Farbinformation und 4 Byte für den Z-Buffer benötigt.

- Das Bild wird Zeile für Zeile von oben nach unten aufgebaut (single line Z-Buffer).  
Der Vorteil ist, daß der Z-Buffer nur für die gerade aktuelle Zeile benötigt wird. Hierdurch wird der Speicherverbrauch im Vergleich zum full screen Z-Buffer erheblich reduziert.

---

<sup>1</sup>Als Friend bezeichnet man in C++ eine Klasse, die auf die privaten Daten einer anderen Klasse zugreifen darf.

Der Nachteil dieser Methode ist, daß die auszugebenden Polygone räumlich sortiert werden müssen. Dadurch ist eine nachträgliche Einfügung von zusätzlichen Objekten in die Szene nicht möglich.

Im ursprünglichen Cgi3D waren beide Versionen des Z-Buffers implementiert. Diese sollten natürlich auch übernommen werden. Wie aus der obigen Beschreibung jedoch ersichtlich ist, ist dies beim single line Z-Buffer nicht ohne Probleme möglich. Diese Probleme werden im nächsten Abschnitt näher erörtert.

## 4.2 Probleme bei der Integration von single line Z-Buffer in das neue Konzept

Bei Verwendung des single line Z-Buffer Algorithmus muß vor Beginn der Darstellung dem Algorithmus die komplette Szene bekannt sein. Dies bedeutet, daß eine Integration in das neue Konzept von Cgi3D nicht so einfach möglich ist.

Aus diesem Grund mußte auf die Implementierung eines Teiles der neuen Funktionalität für den single line Z-Buffer verzichtet werden. Bei Verwendung des single line Z-Buffer Algorithmus wird nur die Ausgabe einer kompletten Szene unterstützt (was zur Zeit also für den Benutzer keinen Unterschied macht). Da das Durchlaufen der Szene innerhalb des Softwarerenderers geschehen muß (dieser muß die Objekte in Abhängigkeit von deren Position durchlaufen) mußte die Routine `renderScene` von `t_RenderScene` überladen werden. Somit muß der single line Z-Buffer Renderer ebenfalls angepaßt werden, wenn sich die Datenstruktur der Szene ändern sollte. Ebenso sind die in 2.2 beschriebenen erweiterten Möglichkeiten von Cgi3D mit diesem Renderer nicht möglich.

Vom Autor wird deshalb der Einsatz des single line Z-Buffer Renderers nur dort für sinnvoll erachtet, wo die Speicherausstattung einen Einsatz des full screen Z-Buffer Renderers nicht zuläßt. Bei der heutzutage üblichen Hauptspeicherausstattung von 32 Megabyte und mehr sollte dies jedoch nur bei sehr großen Auflösungen nötig sein. Bei Ausgabe mit einer Auflösung von  $1280 \times 1024$  werden etwa 9 Megabyte Hauptspeicher für das Bild und full screen Z-Buffer mit 32 Bit Tiefe benötigt.

Somit scheint es nur bei der Ausgabe auf ein viel höher auflösendes Medium (wie z.B. das Speichern der Bitmap für die Ausgabe auf einen Drucker) sinnvoll, den single line Z-Buffer Algorithmus einzusetzen.

## Kapitel 5

# Geschwindigkeitsoptimierungen für verschiedene Plattformen

Ein Ziel dieser Arbeit ist es, die Geschwindigkeit der Darstellung auf den verschiedenen Hardwareplattformen zu optimieren. Hierbei sind im allgemeinen zwei Ansätze möglich:

- Optimierung des ersten Aufbaus der Szene, also eine möglichst schnelle Übergabe der Szenenbeschreibung an die Hardware.
- Optimierung der Animation einer unveränderten Szene. Dies baut einerseits auf dem ersten Punkt auf, da die Hardware übergebene Daten optimiert zwischenspeichert. Somit kann die Ausgabe schneller erfolgen, wenn die Hardware die Daten bereits möglichst kompakt erhält.

Ein weiterer Punkt ist jedoch die Optimierung des Quelltextes. Hierbei geht es darum, daß Angaben für die Hardware, welche sich von Bild zu Bild nicht ändern, auch nur einmal an diese übergeben werden müssen (z.B. die Beleuchtung). Dies ist allerdings hardwareabhängig. So muß bei OpenGL die Beleuchtung für jede Darstellung der Szene übergeben werden, da die Koordinaten der Lichter fest mit der Kamera verbunden sind. Bei Direct3D reicht es hingegen, die Lichter mit der ersten Darstellung der Szene zu setzen.

### 5.1 Verwendung von Arrays zur Übergabe der Daten an die Hardware

Ein erster Ansatz zur Optimierung bietet eine Erweiterung in OpenGL 1.1, welche in Direct3D bereits standardmäßig verwendet wird.

Unter OpenGL 1.0 erfolgt die Übergabe der Szenendaten an die Hardware, indem für jeden Eckpunkt eines Polygons zwei bis drei Funktionen aufgerufen werden:

- `glNormal()`
- `glVertex()`
- `glTexCoord()`, falls das Objekt eine Textur besitzt.

Bei der Verwendung von Arrays für die Übergabe der Daten wird nur noch ein einziger Aufruf je Satz von Arrays (unter Umständen auch für das komplette Objekt) benötigt. Hierbei werden die benötigten Daten zunächst in zwei oder drei (bei Verwendung von Texturen) Arrays gespeichert. Diese Arrays werden dann in einem einzigen Aufruf an die Hardware

übergeben. Die Anzahl der Aufrufe der Hardware ist also nur noch von der Größe der Arrays abhängig. Das heißt, wenn man für die Arrays so viel Speicher bereitstellt, daß alle Daten eines Objektes in die Arrays passen, dann ist pro Objekt wirklich nur ein Aufruf nötig. Ansonsten wird das Objekt in mehrere Teile aufgeteilt und bei vollen Arrays jeweils ein Teil des Objektes an die Hardware übergeben. Da ein Objekt theoretisch aus beliebig vielen Polygonen bestehen kann, ist eine Überprüfung auf „volle“ Arrays auf jeden Fall nötig. Bei Tests hat sich herausgestellt, daß ab einer Arraygröße von 200 Punkten keine signifikante Performancesteigerung mehr zu erzielen ist. Deshalb wird in der aktuellen Implementierung diese Größe gewählt. Sie kann jedoch leicht durch Änderung der Datei `t_renogl.hh` und Neucompilierung von `Cgi3D` geändert werden.

Bei Direct3D wird diese Methode standardmäßig zur Datenübergabe verwendet. Hierbei werden für jeden Eckpunkt die entsprechende Koordinate, Normale und Texturkoordinaten in einem Feld zusammengefaßt. Dieses Array wird dann zusammen mit einem Indexarray an Direct3D übergeben. Dieses Indexarray enthält für jeden Eckpunkt genau einen Index auf einen Eintrag im Koordinatenarray. Somit wäre es bereits hier möglich, Eckpunkte mit identischen Normalen nur einmal zu übergeben. Da dies aber eine Nummerierung der einzelnen Eckpunkte erfordert, wird dieser Ansatz nicht verfolgt. Vielmehr wird diese Nummerierung erst für die optimierte Übergabe der Daten eingeführt (näheres hierzu ist in Kapitel 5.3 zu finden).

## 5.2 Triangle Strip Lists zur Minimierung der Daten

Die größte Geschwindigkeitsoptimierung verspricht eine optimierte Übergabe der Daten an die Hardware. Hierfür wurde in OpenGL 1.1 die Möglichkeit eingeführt, sogenannte Triangle Strip Lists zu übergeben. Hierbei werden die Dreiecke nicht mehr einzeln, sondern als zusammenhängende Liste übergeben. Dadurch werden für  $N$  Dreiecke nicht mehr  $3 * N$  Punkte, sondern nur  $3 + (N - 1)$  Punkte übergeben. Für große  $N$  ergibt sich somit näherungsweise eine Reduzierung der Datenmenge auf  $1/3$ .

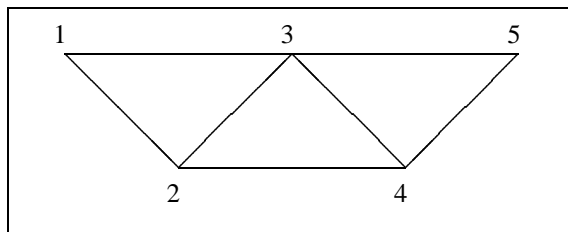


Bild 1: Beispiel für eine Triangle Strip List

Bei der Übergabe der in Bild 1 dargestellten Dreiecke als einzelne Dreiecke müssen die Punkte in der Reihenfolge (1,2,3),(3,2,4),(3,4,5) übergeben werden (die Punkte eines Dreiecks werden entgegen dem Uhrzeigersinn übergeben).

Bei der Übergabe als Triangle Strip List wird jeder Punkt nur einmal übergeben. Für das erste Dreieck der Liste also (1,2,3). Die Kante zwischen den letzten beiden Punkten (also Kante (2,3)) bildet in entgegengesetzter Laufrichtung die erste Kante des nächsten Dreiecks (also Kante (3,2)). Der nächste übergebene Punkt (also Punkt (4)) ist die dritte Ecke des zweiten Dreiecks. Somit bilden also die Kanten (3,2), (2,4) und (4,3) das nächste Dreieck der Triangle Strip List. Mit dem nächsten übergebenen Punkt wird die Triangle Strip List nun analog fortgesetzt (also lauten die Kanten des nächsten Dreiecks (3,4), (4,5) und (5,3)).

Da die darzustellenden Objekte von `Cgi3D` nicht aus Listen von Dreiecken, sondern aus einzelnen Dreiecken bestehen, müssen möglichst lange zusammenhängende Dreieckslisten gefunden werden.

### 5.2.1 Verwendeter Algorithmus

Der verwendete Algorithmus ist eine exakte Umsetzung der in Kapitel 5.2 beschriebenen Methode zum Aufbau einer Triangle Strip List. Da durch die Reihenfolge der Übergabe der Koordinaten des ersten Dreiecks bereits festgelegt ist, welches benachbarte Dreieck des Objektes das nächste Dreieck der Triangle Strip List ist, muß dort nur noch getestet werden, ob dieses Dreieck bereits in einer früheren Liste enthalten ist oder nicht. Ist es bereits in einer Liste enthalten so ist die aktuelle Liste an diesem Dreieck beendet und kann ausgegeben werden. Ist das Dreieck noch nicht enthalten, so wird die dritte Koordinate des Dreiecks der aktuellen Triangle Strip List hinzugefügt und die Entscheidung mit dem nächsten Dreieck fortgesetzt. Dies wird so lange wiederholt, bis alle Dreiecke des Objektes behandelt wurden.

Es ist denkbar, daß es eine bessere Triangulierung des Objektes mittels Triangle Strip Lists gibt als die durch diesen Algorithmus erzeugte. Besser bedeutet dabei, daß weniger Triangle Strip Lists benötigt werden und diese Listen länger sind als die zur Zeit erzeugten. Ohne Kenntnis der Topologie des zugrundeliegenden Objektes ist zur Erzeugung besserer Triangle Strip Lists nur ein heuristischer Ansatz geeignet, d.h. das Objekt müßte mehrmals mit verschiedenen Listen trianguliert und diese Listen dann miteinander verglichen werden. Eine Veränderung der Listen ist nur auf folgende Weise möglich:

- Wahl einer anderen Übergabereihenfolge für die Punkte des ersten Dreiecks:

Eine anderer Triangulierung des Objektes als die durch den verwendeten Algorithmus gegebene Triangulierung ist dadurch möglich, daß die Koordinaten des ersten Dreiecks in einer anderen Reihenfolge übergeben werden, also z.B. anstatt (1,2,3) die Reihenfolge (2,3,1) gewählt wird. Hierdurch bleibt die Orientierung des ersten Dreiecks immer noch erhalten (die Punkte werden entgegen dem Uhrzeigersinn übergeben), die zwei letzten übergebenen Punkte sind jedoch andere als bei der verwendeten Implementierung. Hierdurch ist auch die Lage des zweiten Dreiecks der Triangle Strip List eine andere (anstatt des Dreiecks, welches an die Kante (2,3) grenzt, wird jetzt das Dreieck verwendet, welches an die Kante (3,1) grenzt).

- Wahl eines anderen Startdreiecks:

Durch Wahl eines anderen Startdreiecks für die erste Triangle Strip List entstehen mit hoher Wahrscheinlichkeit andere Triangle Strip Lists als diejenigen, die durch den verwendeten Algorithmus erzeugt werden.

Bei beiden Methoden der Veränderung des Algorithmus ist aber für allgemeine Objekte nicht vorhersagbar, welche Wahl die beste ist. Um bei der Erzeugung der Triangle Strip Lists nicht zu viel Rechenzeit zu verwenden, wurde deshalb die vom MRT übergebene Reihenfolge für die Dreiecke beibehalten.

Das MRT verfügt über einen Iterator, welcher alle Dreiecke des Objektes abläuft. Für den Beginn einer jeden Triangle Strip List wird mittels dieses Iterators ein Dreieck des Objektes ausgewählt. Ist dieses noch nicht in einer Triangle Strip List enthalten, so dient es als erstes Dreieck der neuen Triangle Strip List. Für dieses Dreieck wird nun das nach obiger Vorschrift beschriebene nächste Dreieck ermittelt (ohne den Iterator zu verwenden). Ist dieses Dreieck noch in keiner Liste enthalten, so wird es der aktuellen Liste hinzugefügt. Ansonsten wird die Liste beendet und an die Hardware übergeben. Anschließend wird mittels des Iterators das nächste noch nicht ausgegebene Dreieck bestimmt und die Methode fortgesetzt.

Da der Iterator garantiert alle Dreiecke des Objektes abläuft, braucht nicht abgetestet zu werden, ob alle Dreiecke ausgegeben wurden. Dies meldet der Iterator automatisch.

Bisher wurde immer nur von Dreiecken gesprochen. Objekte des MRT können jedoch auch Polygone mit mehr als drei Ecken enthalten. Da Triangle Strip Lists dies aber nicht können, bedürfen diese Polygone einer besonderen Behandlung.

Vierecke werden dabei in zwei Dreiecke zerlegt und diese beiden Dreiecke der aktuellen Triangle Strip List hinzugefügt.

Polygone mit mehr als vier Ecken werden als einzelne Polygone übergeben, da OpenGL in der Lage ist, diese Polygone direkt auszugeben. Alle zur Zeit im MRT vorhandenen Objekte werden durch Dreiecke oder Vierecke approximiert. Nur bei der Erzeugung von CSG-Objekten<sup>1</sup> können Objekte entstehen, welche Polygone mit mehr als vier Ecken enthalten.

## 5.2.2 Probleme des verwendeten Algorithmus

Das größte Problem bei der Verwendung von Triangle Strip Lists ist die Erzeugung der Strips. Diese sollen möglichst lang sein, um eine effektive Übergabe an die Hardware zu ermöglichen. Außerdem sollten möglichst alle Dreiecke des Objektes von Triangle Strip Lists erfaßt werden.

Selbstverständlich müssen alle Dreiecke des Objektes angezeigt werden. Die letzte Aussage soll nur andeuten, daß nach Möglichkeit alle Dreiecke eines Objektes in einer Triangle Strip List, die länger als nur ein Dreieck ist, enthalten sein sollten. Dreiecke, die nicht zu einer Liste gehören, müssen als einzelne Dreiecke ausgegeben werden, was wiederum zu erhöhtem Datenaufkommen führt. Da es jedoch ohne Kenntnis der Topologie des zugrunde liegenden Objektes nicht möglich ist, optimale Triangle Strip Lists zu erzeugen, bleiben mit hoher Wahrscheinlichkeit einige Dreiecke des Objektes übrig, welche zu keiner Liste gehören.

Ein weiteres Problem besteht in der Übergabe der Normalen für die einzelnen Dreiecke der Triangle Strip List.

Unter OpenGL ist jedem Punkt genau eine Normale zugeordnet. Somit darf ein benachbartes Dreieck nur dann der Liste hinzugefügt werden, wenn die Normalen der benachbarten Dreiecke in beiden geteilten Eckpunkten übereinstimmen. Ansonsten erhält bei Kanten im Objekt ein Punkt eine falsche Normale, was unweigerlich zu sichtbaren Fehlern bei der Darstellung des Objektes führt.

Hierdurch kann es vorkommen, daß bei der Übergabe eines Objektes in Form einer Triangle Strip List diese Übergabe zusammen mit der Erzeugung der Triangle Strip List langsamer ist als wenn das Objekt direkt in Form von Dreiecken ausgegeben würde. Diese Fälle treten besonders drastisch auf bei Objekten mit großen Flächen und scharfen Kanten wie z.B. Würfel oder Pyramiden. Bei den übrigen Objekten von Cgi3D ist dieses Problem jedoch nicht so groß, da es sich bei den meisten Objekten um Approximationen von stetigen mathematischen Objekten (z.B. Kugel) handelt, wobei die Normalen für alle angrenzenden Dreiecke in einem Eckpunkt auf Grund der stetigen Flächenkrümmung identisch sind.

Ein letztes Problem soll auch nicht verheimlicht werden. Da das Durchlaufen des Objektes in einer ganz speziellen Reihenfolge geschehen muß, mußte dieser Durchlauf im optimierten Renderer selber vorgenommen werden. Hierbei wird die Funktion `displayObjectTM` überladen. Diese muß deshalb dann, wenn sich die Datenstruktur des Objektes ändern sollte, ebenfalls angepaßt werden. Somit entfällt für den optimierten OpenGL-Renderer der unter 3.2 erwähnte Vorteil der neuen Struktur von Cgi3D.

---

<sup>1</sup>CSG-Objekte sind Objekte, welche durch mathematische Verknüpfung mehrerer Objekte entstehen

## 5.3 Array-Optimierung bei Direct3D

Unter Direct3D besteht zwar auch die Möglichkeit, Triangle Strip Lists zu erzeugen. Dies ist aber nur im Immediate Mode von Direct3D möglich. Der in der jetzigen Implementierung von Cgi3D verwendete Retained Mode bietet diese Möglichkeit nicht, ist dafür aber erheblich einfacher zu programmieren. Der Unterschied zwischen Immediate Mode und Retained Mode von Direct3D wird in Anhang A genauer erläutert.

Für die Übergabe der Objektdaten an Direct3D gibt es im Retained Mode grundsätzlich zwei Möglichkeiten, `Direct3DRMMesh` und `Direct3DRMMeshBuilder`. Beides sind Direct3D-Objekte (also etwas ähnliches wie C++-Klassen<sup>2</sup>), welche neben den 3D-Objektkoordinaten auch Farbe und Textur verwalten. Der Unterschied zwischen beiden besteht in der Art der Datenübergabe und der Geschwindigkeit der Ausgabe. So haben beide Direct3D-Objekte ihr Vor- und Nachteile, welche im folgenden kurz vorgestellt werden sollen, da sie entscheidend für die Geschwindigkeit der Ausgabe, und somit für ihre Verwendung im optimierten Direct3D-Renderer sind.

- **Direct3DRMMesh**

Dies ist das Direct3D-Objekt, welches für die normale Ausgabe unter Direct3D verwendet wird. Die Datenübergabe erfolgt mittels zweier Arrays, wobei das erste alle Daten des Objektes (Koordinaten, Normalen, Texturkoordinaten) enthält, während das zweite Array ein sogenanntes Index-Array ist. Dieses Array enthält Einträge darüber, wieviele Ecken jedes Polygon hat und welcher Eintrag des ersten Arrays zu welchem Polygon gehört.

- **Direct3DRMMeshBuilder**

Dies ist das Direct3D-Objekt, welches für die Beschleunigung der Ausgabe verwendet wird. Die Datenübergabe erfolgt mittels dreier Arrays. Das erste Array enthält die Koordinaten der Vertices, das zweite Array enthält die Normalen und das dritte Array ist wiederum ein Index-Array, welches ähnlich zum oben beschriebenen Index-Array aufgebaut ist. Nur enthält es diesmal je Polygonecke zwei Einträge, einen für die Koordinate und einen weiteren für die Normale. Wie man sieht, sind hier also Koordinate und Normale voneinander getrennt, so daß es möglich ist, für verschiedenen Polygone eine Koordinate, aber mehrere Normalen zu verwenden, wie dies z.B. bei den Ecken eines Würfels nötig ist.

Wie man aber an der Beschreibung auch sieht, gibt es hier keine Möglichkeit, die Texturkoordinaten für ein Objekt in Form eines Arrays mit zu übergeben. Vielmehr bietet der `Direct3DRMMeshBuilder` nur einen Funktionsaufruf, mit dem man für eine bereits an ihn übergebene Koordinate eine Texturkoordinate setzen kann. Der `Direct3DRMMeshBuilder` hat den Vorteil, daß die Ausgabe eines einmal an Direct3D übergebenen Objektes erheblich schneller erfolgt als mittels `Direct3DRMMesh`. Ein Geschwindigkeitsvergleich hierzu befindet sich in Kapitel 9.

### 5.3.1 Verwendeter Algorithmus

Wie man an der Ausführung im vorigen Kapitel erkennt, erscheint es sinnvoll, `Direct3DRMMeshBuilder` für die Ausgabe der Objekte zu verwenden. Probleme bereiten dabei nur texturierte Objekte. Für diese ist nach Übergabe der Koordinaten an `Direct3DRMMeshBuilder` noch ein zusätzlicher Durchlauf durch das Objekt nötig, um pro Koordinate eine Texturkoordinate zu setzen. Dies ist ein sehr zeitaufwendiges Verfahren, da neben dem erneuten Durchlaufen des Objektes noch ein weiterer Funktionsaufruf von `Direct3DRMMeshBuilder` pro Texturkoordinate hinzukommt.

---

<sup>2</sup>Direct3D ist nicht in C++, sondern in C implementiert. Ein Direct3D-Objekt ist ein sogenanntes COM-Objekt, welches für den Benutzer jedoch ähnlich einem C++-Objekt angesehen werden kann [[Zer97a](#)]

Aus diesem Grund hat sich der Autor dazu entschlossen, in der optimierten Version des Direct3D-Renderers nur nichttexturierte Objekte mit `Direct3DRMMeshBuilder` auszugeben. Texturierte Objekte werden weiterhin mittels `Direct3DRMMesh` an Direct3D übergeben, da eine Kombination beider Funktionen ohne Probleme möglich ist.

Um auch die Möglichkeit zu erörtern, in wie weit es sinnvoll ist, die Daten für die Übergabe an `Direct3DRMMeshBuilder` zu reduzieren, wurde dies ebenfalls in den Direct3D-Renderer aufgenommen. Das Problem hierbei ist, daß die Koordinaten und Normalen des MRT-Objektes nicht numeriert sind. Somit muß diese Numerierung, die für die Indizierung der Koordinaten unter Direct3D unbedingt erforderlich ist, von Cgi3D selber vorgenommen werden. Für jede neue zu übergebende Koordinate muß deshalb das gesamte Koordinatenarray durchsucht werden. Bei Tests hat sich gezeigt, daß dies sehr zeitaufwendig ist. Das Ergebnis ist zwar ein reduziertes Datenvolumen, welches an Direct3D übergeben wird. Bei der Geschwindigkeit der Ausgabe konnte jedoch auf der getesteten Hardware keine Beschleunigung festgestellt werden. Vielmehr wird die erste Übergabe der Daten an Direct3D verlangsamt, wie in Kapitel 9 nachzulesen ist. Da dies aber durchaus von der verwendeten Hardware abhängen kann, wurde diese Implementierung nicht verworfen. Sie kann durch Setzen der Compilerdefinition `useArrayOpti` im Quelltext `t_rend3d.hh` aktiviert werden. Standardmäßig ist sie jedoch deaktiviert.

### 5.3.2 Probleme des verwendeten Algorithmus

Neben dem bereits erwähnten Problem bei der Übergabe der Texturkoordinaten an `Direct3DRMMeshBuilder` führt besonders der Versuch, die Arrays selber zu optimieren, zu Problemen. Da es leider nicht möglich ist, auf einfache Weise an die interne Datenstruktur des Objektes zu gelangen, müssen für den Vergleich der Koordinaten jeweils alle drei Koordinaten (X, Y und Z-Wert) verglichen werden. Wäre ein Zugriff auf die interne Datenstruktur möglich, so würde ein Vergleich der Reference-Pointer auf die Koordinate bzw. die Normale eines Punktes genügen. Allerdings würde dieser Zugriff auf die interne Datenstruktur der Datenkapselung von C++ widersprechen, weshalb diese Methode nicht weiter verfolgt wurde (bei der Verwendung der Triangle Strip Lists unter OpenGL geschieht genau dieser Zugriff mit den in Kapitel 5.2.2 erwähnten Problemen). Ein möglicher Ansatz zu eine elegante Lösung des Zugriffs auf die privaten Daten wird in Kapitel 10.3 gegeben.

Ein weiteres Problem stellt sich bei Performancetests heraus. Die Übergabe der Daten an `Direct3DRMMeshBuilder` erfolgt langsamer als an `Direct3DRMMesh`. Dies ist unabhängig davon, ob versucht wird, die Arrays für `Direct3DRMMeshBuilder` zu optimieren oder nicht. Somit eignet sich die optimierte Version des Direct3D-Renderers besser für die Wiedergabe von Animationen, bei denen sich die Szene nicht ändert. Wird jedoch die Szene verändert, so sollte der Standardrenderer für Direct3D verwendet werden.

Somit bleibt es dem Benutzer überlassen, für welchen Direct3D-Renderer er sich entscheidet. Einen optimalen Direct3D-Renderer für alle Anwendungen gibt es bei der aktuellen Implementierung nicht.

## 5.4 Erzielbare Datenreduktion

Anhand eines kleinen Beispiels soll hier verdeutlicht werden, welche Datenreduktion mit den verwendeten Algorithmen möglich ist. Als Beispielobjekt wird eine Kugel betrachtet. Diese hat den Vorteil, daß jede Ecke von mehreren Dreiecken geteilt wird und die Normale für diese Ecke in allen Dreiecken identisch ist. In diesem Beispiel wird keine Textur verwendet, da dies für die Beispielrechnung nicht relevant ist und diese nur unübersichtlicher machen würde.

Die Kugel wird durch 80 Dreiecke angenähert (dies ist die Standardapproximation des MRT). In der normalen Implementierung müssen hierfür 80 Dreiecke an die Hardware

übergeben werden. Somit sind  $80 \cdot 3 = 240$  Koordinaten und 240 Normalen an die Hardware zu übergeben. Da eine Koordinate ebenso wie eine Normale aus 3 Fließkommawerten besteht, müssen also  $240 \cdot 6 = 1440$  Fließkommazahlen an die Hardware übergeben werden. Unter Direct3D kommt noch ein Indexarray hinzu, welches für jeden Punkt einen Index auf dessen Koordinate im Koordinatenarray enthält. Somit sind zusätzlich noch 240 Integerwerte zu übergeben.

Bei der Verwendung von Triangle Strip Lists entstehen 20 Listen mit einer Länge zwischen 22 und 3 Punkten. Insgesamt werden 120 Koordinaten und 120 Normalen an die Hardware übergeben. Somit müssen  $120 \cdot 6 = 720$  Fließkommazahlen an die Hardware übergeben werden. Dies ist eine Reduzierung auf exakt die Hälfte der ursprünglichen Daten.

Bei Verwendung der Array-Optimierung unter Direct3D müssen genau 80 Koordinaten und 80 Normalen übergeben werden. Somit sind  $80 \cdot 6 = 480$  Fließkommazahlen an die Hardware zu übergeben. Hinzu kommt allerdings noch das Indexarray, welches bei der Array-Optimierung 2 Einträge je Punkt (Koordinate und Normale), also 6 Einträge je Dreieck enthält. Das Array hat somit eine Größe von  $80 \cdot 6 = 480$  Integerwerten. Dies ist zwar von Datenaufkommen her recht viel (genau so viel wie Fließkommazahlen und doppelt so viel wie bei der normalen Übergabeweise der Koordinaten), es muß aber bedacht werden, daß die Fließkommazahlen von der Hardware nochmals weiterverarbeitet werden, wohingegen die Integerwerte nur als Indizes dienen. Die Anzahl der Daten, welche von der Hardware zu bearbeiten sind, reduziert sich also auf ein Viertel der Originaldaten, dafür verdoppelt sich die Größe des Indexarrays.

## Kapitel 6

# Erweiterte Funktionalität der neuen Version

Neben der Restrukturierung von Cgi3D geht es bei dieser Arbeit auch darum, bereits in MRT eingeflossene Neuerungen unter Cgi3D zu implementieren bzw. darzulegen, wie eine Implementierung aussehen könnte.

### 6.1 area lights

Bei area lights handelt es sich um Lichtquellen, die nicht punktförmig sind, sondern eine gewisse Ausdehnung haben. Alle Hardwarerenderer unterstützen jedoch nur Punktlichtquellen. Deshalb müssen area lights bei der Ausgabe der Szene mittels Hardwarerenderer simuliert werden.

Das MRT unterscheidet zwischen zwei Arten von area lights, `t_DistLight` und `t_LightObject`.

`t_DistLight` wird verwendet, um im Raytracer area lights zu simulieren und auf diese Weise Schattenverläufe darzustellen. Bei ihnen handelt es sich um eine kreisförmig ausgedehnte, leuchtende Scheibe, welche jedoch in der Szene, auch wenn sie im Blickfeld des Betrachters liegen sollte, nicht sichtbar ist. Deshalb wird sie in Cgi3D nur durch eine Punktlichtquelle simuliert.

`t_LightObject` sind Objekte mit einer emissiven Oberfläche, also Objekte, die selber leuchten. Diese werden beim Radiosityverfahren als Lichtquellen eingesetzt und sind für den Betrachter sichtbar. Somit müssen sie auch von Cgi3D dargestellt werden.

Der realistischste Ansatz zur Darstellung von area lights in Cgi3D wäre nun, jeder Fläche des Objektes eine gerichtete Punktlichtquelle zuzuordnen. Hierdurch würde jede Fläche zu einer Lichtquelle, so daß das Objekt genau in die richtige Richtung strahlen würde. Außerdem könnte man große Flächen in mehrere Lichtquellen unterteilen, so daß z.B. auch Leuchtstoffröhren, welche durch ein langes, schmales Rechteck dargestellt werden, korrekt simuliert würden.

Dies ist aber nur eine Wunschvorstellung, da eine Realisierung an der Hardware scheitert. Die Berechnung von Lichtquellen ist eines der größten Performanceprobleme bei der Darstellung der Szene. Deshalb erlaubt auch moderne Hardware nur die Definition einer bestimmten Menge von Lichtquellen. Bei OpenGL sind dies im Allgemeinen 8 Stück. Diese Anzahl reicht zwar dazu aus, die Szene korrekt zu beleuchten, aber bei weitem nicht, um leuchtende Objekte in mehrere Lichtquellen zu unterteilen.

Deshalb wurde bei der Darstellung von area lights mit Cgi3D ein anderer Ansatz verfolgt. In das Zentrum der Boundingbox<sup>1</sup> des leuchtenden Objektes wird eine Punktlicht-

<sup>1</sup>Eine Boundingbox ist der kleinste umschließende, achsenparallele Würfel, der das gesamte Objekt, oder bei

quelle gesetzt und die Flächen des Objektes erhalten eine emissive Farbe. Hierdurch erscheint das Objekt leuchtend, wenn es im Sichtbereich des Betrachters ist. Die emissive Farbe ist aber eine reine Simulation für den Betrachter und trägt nichts zur Beleuchtung der Szene bei.

## 6.2 3D-Texturen

3D-Texturen sind prozedurale Texturen, welche nicht, wie 2D-Texturen, aus einer einfachen Bitmap bestehen, die auf das Objekt geklebt wird. Vielmehr handelt es sich um sogenannte solid textures, also Texturen, die sich durch das gesamte Objekt erstrecken. Diese Texturen kann man auch als prozedurale Texturen bezeichnen, da sie durch eine mathematische Funktion berechnet werden.

Eine Darstellung von 3D-Texturen wird in den zur Zeit verwendeten Renderbibliotheken nicht von Hause aus unterstützt (siehe Kapitel 10.2 für einen Ansatz zur Unterstützung unter OpenGL 1.2). Deshalb mußte ein Weg gefunden werden, 3D-Texturen durch 2D-Texturen zu simulieren. Hierdurch wird bereits die erste Beschränkung aufgezeigt, die dieses Verfahren gegenüber einer echten 3D-Textur hat.

Um eine 2D-Textur auf ein Objekt abzubilden, ist es nötig, für jeden Punkt des Objektes einen entsprechenden Punkt in der Textur zu finden. Das MRT bietet hierzu die Funktion **mapInvers**, die genau dieses entsprechende Mapping durchführt. Da diese Funktion verständlicherweise objektabhängig ist, muß sie für jedes Objekt neu implementiert werden. Zur Zeit ist dies für einige Objekte des MRT jedoch noch nicht geschehen. Deshalb ist es bei diesen Objekten nicht möglich, die 3D-Textur mittels der Hardware zu simulieren. Eine Ausgabe der 3D-Textur mittels des Softwarerenderers ist hingegen möglich, da dieser direkt die Farbberechnung des Objektes verwendet, und somit direkt die 3D-Textur abfragt.

Die ideale Vorgehensweise, um eine 2D-Textur aus einer 3D-Textur zu erzeugen, sieht folgendermaßen aus:

Für jedes Pixel der 2D-Textur wird ein korrespondierender Wert auf dem Objekt im dreidimensionalen Raum ermittelt. Die Texturfarbe der 3D-Textur wird an dieser Koordinate ermittelt und in der 2D-Textur gespeichert.

Da jedoch das MRT keine Möglichkeit bietet, aus einer 2D-Texturkoordinate einen Punkt auf einem Objekt zu ermitteln, muß die Vorgehensweise genau umgedreht werden.

Es werden auf dem Objekt möglichst viele, nahe beieinander liegende Punkte ermittelt. Mittels der Funktion **mapInvers** des Objektes wird dann die Koordinate in der 2D-Textur bestimmt. Die an der Objektkoordinate ermittelte 3D-Texturfarbe wird in die 2D-Textur eingetragen.

Das Problem besteht nun darin, möglichst so viele Punkte auf dem Objekt abzutasten, daß alle Pixel der 2D-Textur nach Beendigung des Algorithmus gesetzt sind.

Im folgenden werden zwei Algorithmen vorgestellt, welche zur Erzeugung von 2D-Texturen aus 3D-Texturen verwendet wurden. Der erste hier beschriebene Algorithmus führte allerdings nicht zum gewünschten Erfolg. Da er jedoch durch passende Ergänzungen durchaus dazu in der Lage sein könnte, einige Probleme der aktuellen Implementierung zu beheben, soll er hier ebenfalls erläutert werden. Auf die nötigen Ergänzungen wird in Kapitel 10.1 näher eingegangen.

---

einer Teilszene alle Objekte der Teilszene ganz enthält

### 6.2.1 Erster, wieder verworfener Algorithmus

Um aus einer 3D-Textur eine 2D-Textur für das betreffende Objekt zu erzeugen, muß dieses Objekt abgetastet werden. Dazu müssen Punkte auf dem Objekt ermittelt werden, welche dann mit der Funktion **mapInvers** des Objektes zu einer Koordinate in der 2D-Textur führen.

Der erste Ansatz nimmt nun den direkten Weg, um die Koordinaten auf dem Objekt zu ermitteln:

- Das auszugebende Objekt wird Polygon für Polygon durchlaufen
- Für jeden Eckpunkt eines Polygons wird der Texturwert mit Hilfe der original 3D-Textur berechnet.
- Mit Hilfe der Funktion **mapInvers** wird für jeden Eckpunkt die korrespondierende Koordinate in der 2D-Textur ermittelt und der Texturwert in der 2D-Textur gespeichert.

Ist das gesamte Objekt durchlaufen, so ist auch die 2D-Textur für die Anzeige dieses Objektes fertig und kann mit Hilfe der Hardware ausgegeben werden.

Zu erwarten war, daß die so erzeugte Textur einige schwarze Stellen enthielt, da das Objekt nicht kontinuierlich, sondern nur punktuell gesampelt wurde.

Das Ergebnis des Algorithmus sah jedoch, vorsichtig ausgedrückt, mehr nach einer schwarzen Textur mit bunten Flecken, als nach einem Abbild der 3D-Textur mit schwarzen Flecken aus. Auch bei Verkleinerung der Polygone, also einer exakteren Triangulierung des Objektes, wurde das Ergebnis nur unwesentlich besser.

Bei genauerer Betrachtung der Ausgabe von OpenGL stellte sich auch schnell heraus, welchen Grund das schlechte Ergebnis hatte. Da OpenGL für jedes Pixel eines gerenderten Objektes die Textur abfragt, war die gewünschte Textur nur zufällig an den Stellen, wo mehrere Polygone sehr eng beisammen liegen, oder die Polygone sehr klein sind, korrekt.

Die einzige Lösung, dieses Problem etwas zu entschärfen ist, die Größe der 2D-Textur stark zu verkleinern. Da dabei aber natürlich die Detailauflösung verloren geht, ist dieser Ansatz nicht praktikabel. Ein möglicher, jedoch sehr aufwendiger Ansatz zur Verbesserung dieses Algorithmus ist in Kapitel 10.1 beschrieben.

### 6.2.2 Der zur Zeit implementierte Algorithmus

Da der in Kapitel 6.2.1 beschriebene Algorithmus nicht zu einem optisch ansprechenden Ergebnis führte, mußte ein Weg gefunden werden, das Objekt exakter abzutasten. Weil die Methode, welche in Kapitel 10.1 beschrieben wird, vom Autor als zu aufwendig sowohl in der Implementierung, als auch in der Laufzeit eingeschätzt wird (sie läuft im Endeffekt auf eine Teilimplementierung oder Wiederverwertung des Softwarerenders hinaus), wurde ein anderer Ansatz gesucht.

Die Abtastung des Objektes erfolgt von „außerhalb“ des Objektes. Das Objekt wird von allen Seiten mit Strahlen „beschossen“. An jedem Schnittpunkt zwischen Strahl und Objekt wird der 3D-Texturwert ermittelt, und an der korrespondierenden Stelle der 2D-Textur eingetragen.

Im Einzelnen führt dies zu folgender Vorgehensweise:

- Die Boundingbox des Objektes wird ermittelt. Hierdurch erhält man sechs achsenparallele Rechtecke, die sich sehr gut als Ausgangspunkt für die Strahlen eignen.
- Jedes Rechteck wird in viele gleichmäßig verteilte Punkte aufgeteilt. Hierbei wird in Richtung der längeren Seite mit mehr Punkten abgetastet als in Richtung der kürzeren Seite.

- Von jedem dieser Punkte ausgehend wird ein Strahl senkrecht zur Ebene des Rechtecks in Richtung des Objektes berechnet.
- Dieser Strahl wird mit dem Objekt geschnitten. Falls ein Schnittpunkt existiert, so werden für diesen Schnittpunkt der 3D-Texturwert und die 2D-Texturkoordinate bestimmt und der Wert in die 2D-Textur eingetragen.
- Dies wird so lange wiederholt, bis alle Punkte auf allen Rechtecken der Boundingbox behandelt wurden.

Bei dieser Art der Abtastung besteht nun die Möglichkeit, die Genauigkeit der Abtastung durch Veränderung der Anzahl der Samples pro Rechteck zu erhöhen. Da dies aber zu Lasten der Geschwindigkeit geht, ist sowohl dieser Parameter, als auch die Größe der erzeugten 2D-Textur vom Benutzer durch die Funktion **set3DtextureQuality** oder durch Environmentvariablen wählbar.

Der Parameter **3DmapSize** legt die horizontale und vertikale Auflösung der erzeugten 2D-Textur fest. Von dieser Auflösung ist die Qualität der späteren Darstellung der 2D-Textur abhängig. Der Wert muß eine Zweierpotenz sein, da einige Hardwarerenderer dies voraussetzen. Ansonsten würde die Textur vom Renderer wieder verkleinert, was zu unnützem zusätzlichem Rechenaufwand führt.

Der Parameter **3DminSamples** gibt an, mit wie vielen Samples ein Rechteck mindestens gesampelt wird. Beim Sampeln wird für die horizontale und vertikale Richtung eine unterschiedliche Auflösung gewählt. Hierbei wird die kürzeste Kante der Boundingbox mit **3DminSamples** Samples abgetastet. Die übrigen Seiten erhalten proportional zum Seitenverhältnis mehr Abtastpunkte.

Das Ergebnis dieses Algorithmus ist zwar keineswegs ideal, bei passenden Parametern läßt sich jedoch die Struktur der 3D-Textur sehr wohl erkennen. Bei der Wahl von Parametern, die zu einer akzeptablen Laufzeit für die Erzeugung einer 2D-Textur führen (im Bereich zwischen 5 und 20 Sekunden auf dem Testrechner, siehe Kapitel 9), ergeben sich allerdings immer noch einige „Löcher“ in der 2D-Textur, die sich als schwarze Punkte auf dem Objekt bemerkbar machen. Diese Fehlerfarben fallen jedoch bei üblichen Objekt- und Texturgrößen nicht mehr stark ins Gewicht und können gegebenenfalls von Benutzer durch Wahl anderer Parameter minimiert werden.

## 6.3 Volume Rendering

Unter Volume-Rendering versteht man die Visualisierung von dreidimensionalen Datenfeldern. Diese Datenfelder enthalten in einer dreidimensionalen Matrix Dichteinformationen über das zu visualisierende Objekt (z.B. CT-Daten<sup>2</sup> oder Rauchfelder) Diese Visualisierung wurde in das MRT von Michael Pietsch im Rahmen seiner Diplomarbeit [Pie97] Ende 1997 eingefügt. In seiner Arbeit beschreibt Herr Pietsch zwei unterschiedliche Methoden, die zur Darstellung von Volumendaten mittels eines polygonbasierten Renderers geeignet wären.

In Kapitel 2.2.2 [Pie97] beschreibt er eine polygonale Approximation, welche auf dem *Marching Cubes Algorithmus* [LC87] von Lorence & Cline beruht. Hierbei werden ISO-Flächen<sup>3</sup> aus dem Datensatz extrahiert. Diese Flächen werden durch Polygone dargestellt und können als normales Objekt mittels Cgi3D ausgegeben werden. Dieser Ansatz weist jedoch einige Probleme auf, welche Herr Pietsch folgendermaßen beschreibt: „... daß Oberflächen generiert werden, die unter Umständen Lücken oder Löcher beinhalten können.“. Ein weiteres Problem, welches der Autor dieser Diplomarbeit sieht, ist, daß prinzipbedingt eine ISO-Fläche für einen speziellen Dichtewert erzeugt wird. Soll nun der Dichtewert verändert werden, so ist auch die Erzeugung einer neuen ISO-Fläche erforderlich. Des weiteren ist mit diesem Ansatz eine Darstellung von Rauch nicht möglich, da die erzeugte

<sup>2</sup>Computertomographie

<sup>3</sup>ISO-Flächen sind Flächen gleichbleibender Dichte

ISO-Fläche nicht transparent ist. Es würde also entweder der Rauch selber dargestellt, aber hinter (oder in) ihm liegende Objekte nicht, oder aber nur die Objekte würden dargestellt, jedoch der Rauch selber nicht.

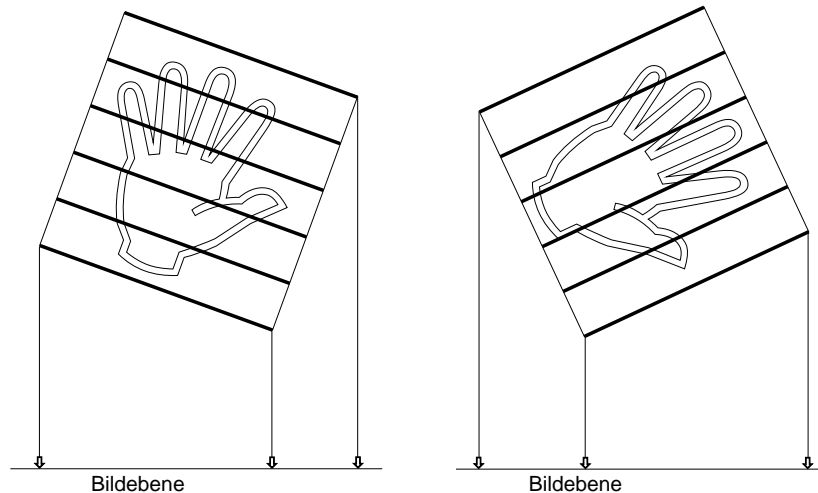


Bild 2: Abbildung der Texturen auf die Bildebene, basierend auf [Pie97]

Aus diesen Gründen erscheint dem Autor der zweite Ansatz, den Herr Pietsch zur Visualisierung von dreidimensionalen Datenfeldern vorschlägt, der geeigneter zu sein. Das in Kapitel 2.3.4 [Pie97] beschriebene Verfahren beruht auf der Darstellung der Datenfelder durch teiltransparente Texturen. Hierzu wird das Datenfeld durch mehrere Texturen abgebildet. Herr Pietsch schlägt hierfür zwei verschiedene Methoden vor, von denen jedoch hier nur auf die zweite Methode eingegangen werden soll, da die erste Methode blickrichtungsabhängig ist und somit eine Neuberechnung der Texturen für jede Darstellung des Objektes erfordern würde. Die zweite Methode (siehe Bild 2) benötigt diese Berechnung hingegen nur einmal im initialen Schritt. Das Datenfeld wird in mehrere Ebenen aufgeteilt, wobei jede Ebene einen Schnitt durch das Datenfeld repräsentiert. Um mit dieser Aufteilung das Objekt von jeder Seite aus darstellen zu können, müssen drei Sets von Texturen erzeugt werden, die in Abhängigkeit vom Winkel des Betrachters zum Objekt ausgewählt werden (Blick von vorne/hinten, Blick von rechts/links und Blick von oben/unten). Hierbei darf jeweils nur ein Set von Texturen aktiv sein, da jedes Set die gesamte Dichteinformation des Datenfeldes enthält.

In Bild 2 sind zwei der drei benötigten Sets dargestellt. Das linke Bild zeigt ein Set für die Darstellung von vorne/hinten, das rechte Bild für die Darstellung von links/rechts. Zur Erzeugung eines Sets von Texturen und zugehörigen Flächen für Cgi3D muß folgendermaßen vorgegangen werden:

- Das Datenfeld wird in mehrere parallel hintereinanderliegende Rechtecke in der benötigten „Blickrichtung“ unterteilt. Diese sollten parallel zu den Kanten des Datenfeldes gewählt werden, da auf diese Weise eine einfache Bestimmung und schnelle Erzeugung der Texturen möglich ist. Auf jeden Fall muß jede „Blickrichtung“ senkrecht zu den beiden anderen „Blickrichtungen“ stehen. In Bild 2 sind diese Rechtecke durch dicke Linien dargestellt.
- Für jede dieser Flächen wird ein Rechteck als Objekt für Cgi3D bestimmt.
- Für jedes Rechteck wird eine Textur aus dem Datenfeld bestimmt. Ein Texel<sup>4</sup> enthält dabei neben der Farbinformation einen sogenannten Alpha-Wert. Dieser Wert bestimmt die Transparenz des jeweiligen Texels und wird dadurch bestimmt, daß die

<sup>4</sup>Ein Texel (texture element) ist ein Element der Textur, also im Normalfall ein Pixel

Dichtewerte des Datenfeldes zwischen der letzten erzeugten Textur und der neuen Textur in Blickrichtung aufmultipliziert werden. Hierdurch enthält das Texel genau den Dichtewert, der dem Datenfeld zwischen dem vor ihm liegenden Texel und sich selber entspricht, wodurch keine Dichteinformationen des Datenfeldes verloren gehen. Für eine genaue Beschreibung dieser Berechnung siehe [Pie97], Kapitel 2.3.2.

- Dieser Vorgang wird für alle Rechtecke eines Textursets durchgeführt.

Nachdem die Textursets zusammen mit den Rechtecken erzeugt wurden, müssen diese durch Cgi3D ausgegeben werden. Hierbei muß die Ausgabe eines Sets mit der Ebene beginnen, welche als nächste zum Betrachter liegt, damit die Alpha-Werte in der korrekten Reihenfolge von der Hardware aufmultipliziert werden. Bei dieser Ausgabe könnte durch Wahl eines Grenzwertes für Alpha bestimmt werden, welche ISO-Fläche wiedergegeben werden soll.

Die Wiedergabe eines Volumenobjektes mit der aktuellen Version von Cgi3D erfordert einige Ergänzungen des Objektes `t_RenderScene`. Dies ist aus drei Gründen nötig:

1. Ein Volumenobjekt besteht aus mehreren einzelnen Objekten, nämlich seinen Rechtecken mit den zugehörigen Texturen. Die derzeitig unterstützten Objekte verwenden pro Szenenobjekt genau eine zusammenhängende Oberfläche mit einer einzigen Textur.
2. Abhängig von der Blickrichtung des Betrachters zum Objekt muß das geeignete Texturset ausgewählt werden. Die Auswahl geschieht dadurch, daß dasjenige Set gewählt wird, dessen Flächennormale mit dem Augvektor (dem Vektor zum Betrachter) den kleinsten Winkel bildet.
3. Wie bereits erwähnt, muß die Ausgabe der Rechtecke eines Textursets in der korrekten Reihenfolge geschehen.

Der erste Punkt ließe sich noch durch die Aufteilung des Volumenobjektes in mehrere Objekte, welche alle als einzelne Objekte in die auszugebende Szene eingefügt würden, lösen.

Der zweite Punkt ist jedoch mit zur Zeit gegebenen Mitteln nicht zu realisieren, da sich hierbei unter Umständen das Objekt je nach Blickrichtung ändert. Die Integration dieses Punktes ist am elegantesten dadurch zu lösen, daß er zusammen mit einer neuen Struktur des Szenengraphen implementiert wird. Wie in Kapitel 8.1 beschrieben, ist die Erweiterung des Szenengraphen um sogenannte Aktionsknoten geplant. In einem solchen Knoten könnte dann, abhängig von Blickwinkel des Betrachters, das korrekte Texturset für die nächste Darstellung der Szene gewählt werden. In derselben Routine wäre dann auch Punkt drei, die korrekte Ausgabereihenfolge der Rechtecke eines Textursets, zu lösen.

Durch diesen Lösungsansatz würde eine Änderung an Cgi3D selber für Volumenobjekte unnötig. Da auch, wie im Kapitel 10.2 näher erläutert, keine der für die Entwicklung verwendeten Hardwarerenderer einen Alpha-Buffer unterstützt, wurde dieses Verfahren nicht implementiert. Der Autor hält es für sinnvoll, diese Implementierung erst im Rahmen der neuen Struktur des Szenengraphen durchzuführen.

Ein Problem, das sich bei der oben beschriebenen Erzeugung der Textursets aus dem Volumenobjekt ergibt, ist die Aufmultiplikation der Dichtewerte. Da dies im Grunde genommen blickrichtungsabhängig ist, ist die ausgegebene Dichteverteilung nur dann korrekt, wenn der Betrachter genau senkrecht zum Texturset steht. Dieses Problem ist um so größer, je größer der Abstand der einzelnen Texturen untereinander ist. Falls der Abstand der Texturen genau dem Abstand der Voxels<sup>5</sup> des Datenfeldes entspricht, so ist die Ausgabe stets korrekt, da jedes Texel die Dichtedaten von genau einem Voxel enthält. Allerdings kann dadurch die Anzahl der Texturen recht groß werden. Die von Herrn Pietsch

---

<sup>5</sup>Ein Voxel (volume texture element) ist ein Volumenelement, also hier ein Dichteintrag im Datenfeld

verwendeten CT-Daten haben eine Auflösung von  $256 \times 256 \times 109$ . Somit würden sich zwei Textursets mit einer Größe von  $256 \times 109$  Pixeln und ein Set mit der Größe  $256 \times 256$  Pixeln ergeben. Die ersten beiden Sets hätten jeweils 256 Texturen, das letzte Set hätte 109 Texturen. Somit ergäbe sich ein Datenaufkommen von etwa 21 Millionen Pixeln, also etwa 64 MB Daten alleine für die Texturen. Da dies wohl auch eine gut ausgestattete Workstation überfordert, muß deshalb ein Kompromiß zwischen Datenmenge und Qualität geschlossen werden.

Bei der Wiedergabe des Datenfeldes mittels der in Kapitel 10.2 vorgestellten 3D-Texturfähigkeit von OpenGL 1.2 würde sich dieses Datenvolumen jedoch auf ein Drittel, also auf ca. 21 MB reduzieren, womit die Wiedergabe möglich sein sollte.

## Kapitel 7

# Korrektheit der Darstellung im Vergleich zu Raytracing

Ziel einer jeden Darstellung ist es, die vom Benutzer konstruierte Szene möglichst naturgetreu darzustellen. Hierbei bedeutet naturgetreu in erster Linie, daß die Darstellung möglichst einer Fotografie entsprechen sollte. Um überhaupt eine Vergleichsmöglichkeit für die Qualität der Ausgabe mittels eines Hardwarerenders zu haben, wird im folgenden die Ausgabe der Renderer mit der Ausgabe des im MRT vorhandenen Raytracers verglichen. Ziel der Hardwarerenderer im Cgi3D sollte es also sein, der Ausgabe einer Berechnung durch den Raytracer möglichst zu entsprechen. Als weiteres Ziel von Cgi3D darf aber auch die Geschwindigkeit der Ausgabe nicht vernachlässigt werden. Aus diesem Grund müssen zeitweise Kompromisse zwischen erreichbarer Darstellungsqualität und Ausgabegeschwindigkeit geschlossen werden.

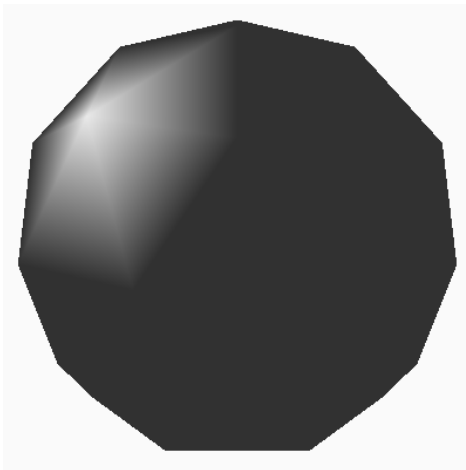


Bild 3: Standardapproximation

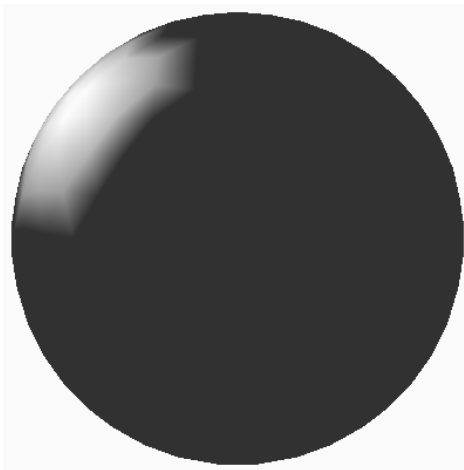


Bild 4: beste Approximation

Die Genauigkeit der Darstellung, einmal abgesehen von ihrer Beleuchtung, hängt im wesentlichen davon ab, wie genau das Objekt durch die polygonale Darstellung approximiert wird. Als Beispiel betrachte man Bild 3 und Bild 4. Bei der dort abgebildeten Kugel handelt es sich um die Approximation mit einer unterschiedlichen Anzahl von Polygonen. In Bild 3 wurden 80 Dreiecke verwendet (dies ist die Standardapproximation des MRT), in Bild 4 wurden 1280 Dreiecke verwendet (dies ist die maximale Qualität, welche das MRT anbietet). Man erkennt eindeutig den Unterschied in der Darstellung, ganz besonders am Rand der Kugel. Die Kugel in Bild 3 sieht eher wie ein Vieleck denn wie eine Kugel aus. Hingegen ist in Bild 4 nicht zu verkennen, um welches Objekt es sich handelt. Allerdings

benötigt die Approximation von Bild 4 auch 16 mal so viele Dreiecke wie diejenige von Bild 3. Da dies besonders bei der Wiedergabe von Animationen ein entscheidender Faktor ist (je mehr Polygone angezeigt werden müssen, desto länger dauert die Ausgabe eines Bildes), muß ein Kompromiß zwischen Datenmenge und Darstellungsqualität gefunden werden. Hinzu kommt noch, daß Objekte, die sich weiter weg vom Betrachter befinden, in der Wiedergabe eine kleinere Auflösung haben und somit durch weniger Polygone angenähert werden könnten als Objekte, die nahe beim Betrachter liegen.

Der Raytracer arbeitet direkt auf dem mathematischen Objekt, so daß hier die Auflösung des Objektes nur von der verwendeten Auflösung der Ausgabe abhängt (Bilder 3 bis 6 wurden mit einer Auflösung von  $400 \times 400$  Pixeln berechnet). Bei genauer Betrachtung von Bild 4 und Bild 6 läßt sich deshalb immer noch ein leichter Vorteil des Raytracers gegenüber der Ausgabe der optimalen Triangulierung der Kugel feststellen (die Rundungen erscheinen noch etwas runder als beim Hardwarerenderer).

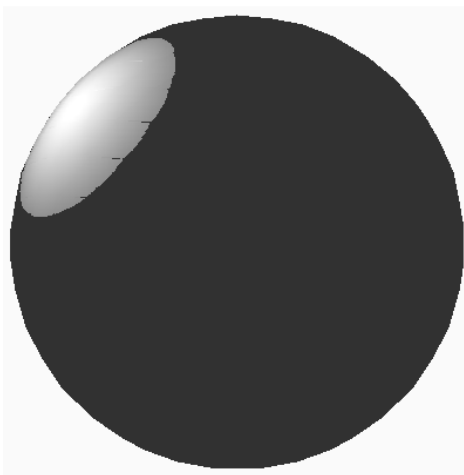


Bild 5: Software Phong Shading

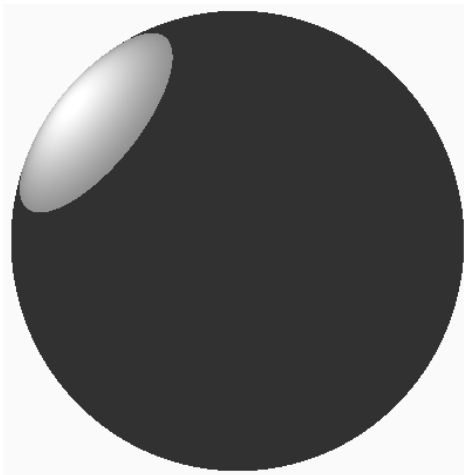


Bild 6: Raytracing

Ein weiterer Aspekt der Darstellungsqualität ist die Auswertung der Beleuchtung.

Der Softwarerenderer verwendet beim Beleuchtungsmodell Diffuse eine rein ambiente und diffuse Reflexion ohne Berücksichtigung der spiegelnden Materialkomponente. Die verwendete Beleuchtungsgleichung lautet folgendermaßen:

$$I = I_a * k_a + f(d)I_{lq}(N * L)k_d$$

Hierbei ist  $I$  die zu berechnende Intensität,  $I_a$  die (globale) ambiente Intensität,  $k_a$  der ambiente Reflexionskoeffizient,  $f(d)$  eine Funktion zur entfernungsabhängigen Abschwächung der Intensität (hierdurch wird die Entfernung zwischen beleuchtetem Punkt und Lichtquelle berücksichtigt),  $I_{lq}$  die Intensität einer Lichtquelle,  $N$  die Oberflächennormale am Punkt der Intensitätsberechnung,  $L$  der Vektor zur Lichtquelle an diesem Punkt und  $k_d$  der diffuse Reflexionskoeffizient.

Diese Gleichung wird für die drei Grundfarben rot, grün und blau getrennt ausgewertet. Bei diesem Beleuchtungsmodell werden keine Highlights, also Spiegelungen der Lichtquelle auf dem Objekt, erzeugt. Dieses Beleuchtungsmodell ist nur sinnvoll mit Flat- oder Gouraud-Shading zu verwenden. Für die Verwendung von Phong-Shading sollte das im folgenden beschriebene Beleuchtungsmodell gewählt werden.

OpenGL, Direct3D und XGL verwenden das Phong Beleuchtungsmodell, welches beim Softwarerenderer als Phongillumination bezeichnet wird. Dieses wertet neben der diffusen Reflexion auch die spiegelnde Reflexion aus. Die Beleuchtungsgleichung lautet:

$$I = I_a * k_a + f(d)I_l(k_d(N * L) + k_s(A * R)^c)$$

Zusätzlich zu den oben beschriebenen Werten sind hierbei  $k_s$  der spiegelnde Reflexionskoeffizient,  $A$  der Vektor vom Punkt der Intensitätsberechnung zum Betrachter,  $R$  der reflektierte Lichtvektor und  $c$  der Materialkoeffizient, der die Größe des Reflexionskegels bestimmt.

Die Verwendung der Phong-Beleuchtungsgleichung darf für die Qualität der Ausgabe keinesfalls mit einer Ausgabe mittels Phong-Shading verwechselt werden. Alle Bibliotheken verwenden nur ein Gouraud-Shading zur Ermittlung der Farbe innerhalb eines Polygons. Dabei wird für jeden Eckpunkt des Polygons die Farbe mittels der Beleuchtungsgleichung bestimmt (also mittels der Phong-Beleuchtungsgleichung). Aus diesen Farben der Eckpunkte wird dann aber mittels bilinearer Interpolation die Farbe des Polygoninneren ermittelt. Phong-Shading, so wie es der Softwarerenderer bei der Option PhongShading verwendet, wertet dagegen die Beleuchtungsgleichung an jeder Stelle des Polygons erneut aus. Hierbei werden die Oberflächennormalen innerhalb des Polygons durch Interpolation über die Normalen in den Eckpunkten ermittelt.

Den Unterschied dieser beiden Methoden erkennt man bei Betrachtung von Bild 3 und 4. Bild 3 wurde mittels OpenGL mit Gouraud-Shading dargestellt. Es zeigt zwar deutlich das Spotlight, welches sich links oberhalb der Kugel befindet, der Verlauf des Lichtkegels auf der Kugel ist jedoch nicht korrekt. Im Vergleich hierzu zeigt Bild 4 die korrekte Darstellung des Lichtkegels, da hier für jedes darzustellende Pixel die Beleuchtungsgleichung ausgewertet wird. Hierdurch läßt sich auch der Helligkeitsunterschied innerhalb eines Polygons darstellen.

Der Raytracer (Bild 5) verwendet ebenfalls die Phong-Beleuchtungsgleichung, muß jedoch die Oberflächennormalen nicht interpolieren, da er direkt mit dem mathematischen Objekt arbeitet.

Zur korrekten Darstellung einer Szene gehören auch Schatten. Prinzipbedingt stellen die Hardwarerenderer keinen Schatten dar. Dies liegt daran, daß die Hardware jedes Objekt getrennt für sich alleine betrachtet und ausgibt. Einzig die Tiefeninformation wird im Z-Buffer gespeichert, um verdeckte Polygone entfernen zu können. Dieses Problem ist auf einfache Weise nur dadurch zu umgehen, daß die Szene nur durch eine einzige Punktlichtquelle, welche genau im Zentrum der Kamera sitzt, beleuchtet wird. Hierdurch liegen alle Schatten, die Objekte werfen, genau in dem von den Objekten verdeckten Sichtbereich, so daß auch bei Berechnung der Schatten diese für den Betrachter unsichtbar wären. Eine Möglichkeit zur Ausgabe echter Schatten wäre natürlich die Verwendung eines anderen Algorithmus, wie z.B. des Shadow-Z-Buffer Algorithmus, wie er in [Wat92] beschrieben ist.

Zum Abschluß des Vergleiches soll noch kurz auf die Darstellung von Transparenz eingegangen werden. Die zur Zeit implementierten Hardwarerenderer, und auch der Softwarerenderer, verwenden für die Darstellung von transparenten Objekten Stipple Patterns, welche teiltransparente Texturen sind. Hierdurch ist die Darstellung von Transparenz nur in sehr geringem Umfang möglich. Erstens ist das Objekt entweder voll transparent (wenn die Textur durchsichtig ist), oder es ist nicht transparent. Der Eindruck der teilweisen Transparenz entsteht nur dadurch, daß sich transparente und nichttransparente Pixel abwechseln. Eine echte Abschwächung der Intensität von Objekten, die räumlich gesehen hinter dem transparenten Objekt liegen, ist hiermit nicht möglich. Mit dieser Methode der Darstellung ist deshalb nur eine Annäherung der echten Transparenz zu erreichen, ganz abgesehen davon, daß eine Simulation des Brechungsindex des Materials auf diese Weise nicht möglich ist. Eine besserer Methode zur Darstellung von Transparenz ist in Kapitel 10.2 beschrieben, welche mit der aktuell vorhandenen Hardware jedoch nicht realisierbar ist.

Insgesamt läßt sich also sagen, daß die Darstellung einer Szene nur unter ganz bestimmten Voraussetzungen in die Nähe der Qualität einer Berechnung durch den Raytracer kommt. Der Softwarerenderer erreicht das gesteckte Ziel bei Verwendung von Phong-Shading noch am ehesten, benötigt hierfür aber auch die meiste Zeit. Die Darstellung der Szene mittels Gouraud-Shading kann bei entsprechend feiner Triangulierung der Szene aber durchaus dazu dienen, einen Gesamteindruck der Szene so, wie sie später von Raytracer berechnet würde, zu erhalten (abgesehen von den fehlenden Schatten).

Bei alledem sollte jedoch nicht vergessen werden, daß die Darstellung der Szene mittels Hardware nicht nur das Ziel verfolgt, möglichst exakt zu sein. Vielmehr wird immer ein Kompromiß zwischen Exaktheit der Darstellung und Geschwindigkeit der Ausgabe zu schließen sein.

## Kapitel 8

# Aufwand zur Änderung und Portierung

Da sowohl das MRT, als auch Cgi3D ein System ist, das ständig weiterentwickelt und verbessert wird, läßt es sich dabei nicht vermeiden, daß sich auch einige grundlegende Dinge wie z.B. Datenstrukturen im System ändern. Dieses Kapitel beschreibt, wie hoch der Aufwand ist, Cgi3D an solche angenommenen Änderungen anzupassen. Da sich während des Schreibens der Diplomarbeit einige solche Änderungen ergaben, beruhen die Angaben nicht nur auf Schätzungen, sondern auch auf den Erfahrungen, die der Verfasser bei diesen Änderungen gemacht hat.

### 8.1 Aufwand bei internen Änderungen, z.B. bei der Datenstruktur

Mit internen Änderungen sind hier nicht Änderungen an Cgi3D selber, sondern vielmehr Änderungen an Cgi, MRT oder den Generic-Klassen des MRT gemeint. Einige dieser Änderungen können sich auch insoweit auf Cgi3D auswirken, als sie z.B. den Aufbau der Objekte oder das Durchlaufen der Szene für die Anzeige betreffen. Letzteres trat während der Entwicklung der neuen Struktur von Cgi3D auf, als das grundsätzliche Durchlaufen einer Szene beim Übergang von MRT 207 zu MRT 208 geändert wurde.

Zur Anzeige einer Szene muß diese verständlicherweise komplett durchlaufen werden. Durch die strikte Trennung von Cgi3D zwischen hardwareabhängigen und -unabhängigen Funktionen waren die nötigen Änderungen nur im Grundobjekt `t_RenderScene`, von dem alle Hardwarerenderer abgeleitet sind, einmal zentral durchzuführen.

Ebenso sähe es bei einer Änderung oder Erweiterung des Szenegraphen aus. Für einen späteren Zeitpunkt ist die Einführung von sogenannten Aktionsknoten in den Szenegraphen geplant. Diese sollen dann eine zentrale Funktion zur interaktiven Änderung des Szenegraphen übernehmen, so daß z.B. durch Änderung eines Knotens im Graphen ein ganzer Teilast eine andere Schattierung oder eine neue Position in der Szene erhält.

Zur Auswertung dieser Aktionsknoten ist es nur nötig, das Grundobjekt `t_RenderScene` zu ändern. Eine Ausnahme bildet nur den Softwarerenderer, der single line Z-Buffer verwendet. Da dieser, wie in Kapitel 4 beschrieben, den Szenegraphen selber durchläuft, muß er ebenfalls angepaßt werden.

### 8.2 Aufwand zur Portierung auf neue Hardware

Zur Unterstützung einer neuen Hardware unter Cgi3D ist es notwendig, ein neues Objekt von `t_RenderScene` abzuleiten und dort alle benötigten Funktionen zu implementie-

ren. Funktionen, welche nicht direkt von der Hardware abhängen, müssen dabei nicht neu implementiert werden. Zu diesen Funktionen zählen unter anderem das Durchlaufen des Szenengraphen und das Durchlaufen der einzelnen Objekte. Welche Funktionen zu implementieren sind, wird in Kapitel 13 genauer beschrieben.

Da alle Funktionen für die Hardwareansteuerung klar gegliedert sind, muß für die erste erfolgreiche Unterstützung einer neuen Hardware nicht unbedingt die komplette Funktionalität implementiert werden. So kann z.B. die Unterstützung von Texturen, oder sogar die Auswahl des Shadings zunächst einmal unberücksichtigt bleiben und man kann sich voll und ganz auf die eigentliche Darstellung der Polygone beschränken. Wenn dies funktioniert, ist es dann ein Einfaches (aus Sicht von Cgi3D, nicht aus Sicht der Hardwareansteuerung), noch fehlende Funktionen durch Implementierung der passenden Funktionen in Cgi3D zu ergänzen. Hierdurch wird auch die Fehlersuche vereinfacht, da Fehler einfacher einer speziellen Funktion zuzuordnen sind.

# Kapitel 9

## Performancetests

Zum Test der Geschwindigkeit der neuen Version von Cgi3D und zum Vergleich mit der alten Version wurde ein kurzes Programm implementiert, welches eine beliebige Szenenbeschreibung einliest und in einem Cgi3D-Fenster darstellt. Die Einstellungen der Kamera werden aus der Szenenbeschreibung übernommen. Nach dem ersten Darstellen der Szene wird die Kamera in 200 Schritten zu je 3,6 Grad um den Lookpoint der Kamera gedreht. Der Up-Vektor der Kamera bleibt dabei unangetastet. Somit rotiert die Kamera zweimal komplett um den Punkt der Szene, auf den die Kamera blickt.

Während der Rotation wird die Szene nicht geändert. Es wird die Zeit gemessen, welche für die 200 Darstellungen benötigt wird. Dabei geht die Zeit, die für den ersten Aufbau der Szene benötigt wird, nicht mit in die Messung ein. Hierdurch wird also die reine Zeit gestoppt, welche Cgi3D für das erneute Darstellen der Szene durch die Hardware benötigt.

Um jedoch nicht nur die Hardwaregeschwindigkeit zu messen, erfolgt ein zweiter Test mit demselben Programm, wobei aber der Hardwarerenderer nun dazu aufgefordert wird, jedes Objekt bei jedem Schritt neu an die Hardware zu übergeben. Hier gehen dann auch die Zeiten mit ein, die Cgi3D selber benötigt, um das Objekt möglichst ideal an die Hardware weiterzureichen.

### 9.1 Testplattformen

Zum Messen der Ausgabegeschwindigkeiten von Cgi3D wurde mit dem in Kapitel 9 beschriebenen Programm die Szene `all.msdc`<sup>1</sup> in der Größe 400×400 Pixel mit Gouraud-Shading dargestellt. Die Größe wurde gewählt, um auch mit der Softwaredarstellung eine erträgliche Testzeit zu erhalten.

Die Tests auf dem PC wurden auf einem AMD K6/233 mit einer Matrox Mystique vorgenommen. Die Bildschirmauflösung betrug jeweils 1024×768 mit 16 Bit Farbtiefe.

Unter Linux wurde der OpenGL-Treiber Mesa 2.6 verwendet, unter Windows 95 der OpenGL 1.1 Treiber von Microsoft.

Die Tests auf der SGI fanden auf einer Indigo 2 mit 12 Bit Grafikauflösung statt, welche hardwarebeschleunigtes Doublebuffering mit OpenGL ermöglicht. Als Betriebssystem war IRIX 6.2 installiert, so daß dort nur eine OpenGL 1.0 Version vorhanden war. Da diese aber die Erweiterung `GL_EXT_vertex_array` besitzt, wurden hierbei auch die Arrays zur beschleunigten Ausgabe verwendet.

Bei allen Tests wurde darauf geachtet, daß auf dem Computer außer dem Autor selbst kein Benutzer einen Prozeß gestartet hatte. Auf dem PC stand Cgi3D somit die gesamte Rechenzeit zur Verfügung. Auf der SGI, welche an ein Netz angeschlossen ist, konnte dieses nicht gewährleistet werden. Die Tests wurden jedoch auf allen Plattformen mehrmals

---

<sup>1</sup>Die Szene `all.msdc` ist im Archiv `scenes.zip` des MRT enthalten

durchgeführt und die Ergebnisse gemittelt, wobei extreme Ausreißer außer Acht gelassen wurden.

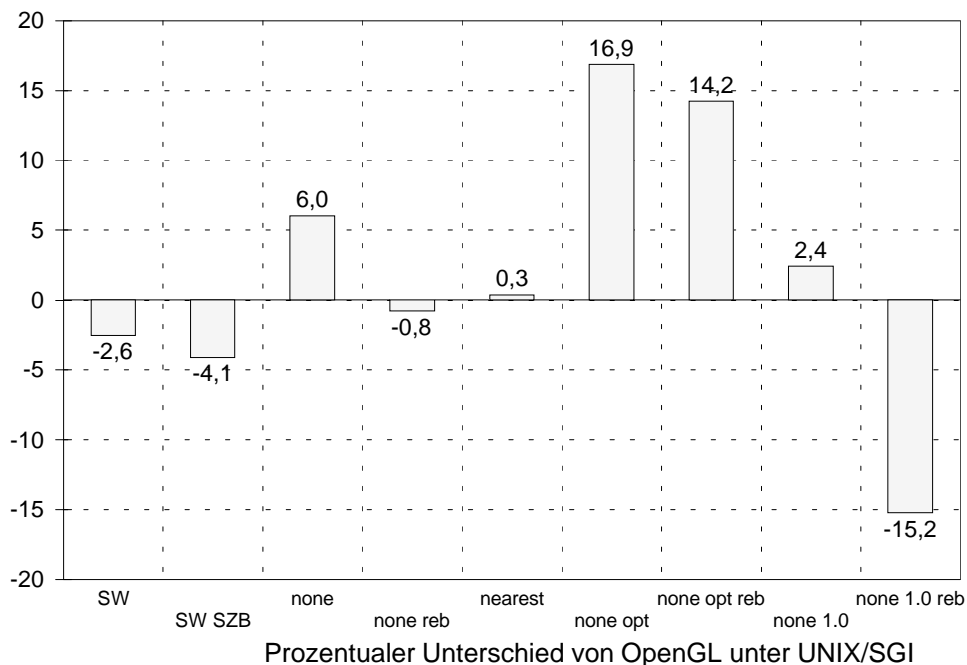
## 9.2 Ergebnisse und Bewertung

In den folgenden Kapiteln sind die Testergebnisse sowohl als Tabellen, wie auch vergleichend in Bildern dargestellt. Die Bilder zeigen dabei jeweils den prozentualen Geschwindigkeitsgewinn (oder Verlust) der neuen Version gegenüber der alten Version von Cgi3D an. Die Tabellen geben die exakten Meßwerte in Sekunden wieder. Die Zahlenangabe 208 in der Beschriftung der Tabellen steht dabei für die alte Version von Cgi3D. Da in den Tabellen aus Platzgründen die Testkriterien gekürzt wiedergegeben werden müssen, folgt zunächst die Erläuterung der Fußnoten für alle aufgeführten Testkriterien:

1. Die Ausgabe erfolgte mittels des Softwarerenderers von Cgi3D mit full screen Z-Buffer. Es wurde Gouraud-Shading verwendet.
2. Die Ausgabe erfolgte mittels des Softwarerenderers von Cgi3D mit single line Z-Buffer. Es wurde Gouraud-Shading verwendet.
3. Die Ausgabe erfolgte mittels des Softwarerenderers von Cgi3D mit full screen Z-Buffer. Es wurde Gouraud-Shading verwendet. Das Ergebnis wurde nicht auf dem Bildschirm ausgegeben, sondern nur im Hintergrundspeicher gehalten.
4. Die Ausgabe erfolgte mittels OpenGL 1.1 bzw. mit dem Standardrenderer von Direct3D ohne Texturen
5. Die Ausgabe erfolgte mittels OpenGL 1.1 bzw. mit dem Standardrenderer von Direct3D ohne Texturen. Bei jeder neuen Ausgabe der Szene wurde diese erneut an die Hardware übergeben, die Displaylisten wurden nicht verwendet.
6. Die Ausgabe erfolgte mittels OpenGL 1.1 bzw. mit dem Standardrenderer von Direct3D mit eingeschalteten Texturen in der Option „nearest“. Diese wählt aus der Textur jeweils das nächstliegende Pixel für die Darstellung und ist die schnellste Methode der Texturdarstellung.
7. Die Ausgabe erfolgte mittels OpenGL 1.1 oder Direct3D ohne Texturen. Bei OpenGL wurden Triangle Strip Lists für die Übergabe der Objekte verwendet, unter Direct3D der optimierte Treiber.
8. Die Ausgabe erfolgte mittels OpenGL 1.1 oder Direct3D ohne Texturen. Bei OpenGL wurden Triangle Strip Lists für die Übergabe der Objekte verwendet, unter Direct3D der optimierte Treiber. Bei jeder neuen Ausgabe der Szene wurde diese erneut an die Hardware übergeben, die Displaylisten wurden nicht verwendet.
9. Die Ausgabe erfolgte mittels OpenGL 1.0 ohne Texturen. Hierbei wurden bewußt die OpenGL 1.1 Erweiterungen im Quelltext von Cgi3D deaktiviert.
10. Die Ausgabe erfolgte mittels OpenGL 1.0 ohne Texturen. Hierbei wurden bewußt die OpenGL 1.1 Erweiterungen im Quelltext von Cgi3D deaktiviert. Bei jeder neuen Ausgabe der Szene wurde diese erneut an die Hardware übergeben, die Displaylisten wurden nicht verwendet.
11. Die Ausgabe erfolgte mittels des optimierten Direct3D Renderers. Dabei wurde die Array-Optimierung im Quelltext aktiviert.
12. Die Ausgabe erfolgte mittels des optimierten Direct3D Renderers. Dabei wurde die Array-Optimierung im Quelltext aktiviert. Bei jeder neuen Ausgabe der Szene wurde diese erneut an die Hardware übergeben, die Displaylisten wurden nicht verwendet.

## 9.2.1 Ergebnisse unter Unix/SGI

Ergebnisse auf SGI unter UNIX		
	SGI	SGI, 208
Software <sup>1</sup>	2:32.5	2:28.7
Software SZB <sup>2</sup>	3:35.1	3:26.6
none <sup>4</sup>	0:15.6	0:16.6
none, reb.all <sup>5</sup>	0:39.0	0:38.7
nearest <sup>6</sup>	0:29.3	0:29.4
none, opt <sup>7</sup>	0:13.8	—
none, opt, reb.all <sup>8</sup>	0:33.2	—
none, OpenGL 1.0 <sup>9</sup>	0:16.2	0:16.6
none, OpenGL 1.0, reb.all <sup>10</sup>	0:44.6	0:38.7



Beim Vergleich der Zeiten unter UNIX kann man zwei grundlegende Ergebnisse unterscheiden:

1. Die Anzeige konnte mit Hilfe der nicht optimierten Treiber beschleunigt (um ca. 6%) werden. Hierdurch zeigt sich, daß trotz der Aufteilung der ursprünglichen Routinen und der Verwendung einer Basisklasse für alle Renderer die Geschwindigkeit keinesfalls abgenommen hat. Vielmehr konnte dieser zusätzliche interne Aufwand (neben der Aufteilung des Renderers in mehrere Unterfunktionen muß pro Funktionsaufruf eine Referenz mehr berücksichtigt werden, da alle Funktionen virtuelle Funktionen sind) durch die Verwendung von Arrays zur Übergabe der Daten an die Hardware mehr als kompensiert werden. Der zusätzliche Rechenaufwand für die Erzeugung der Arrays, der bei der ständigen neuen Übergabe der Daten an die Hardware mit einfließt, ist mit unter 1% sehr gering.

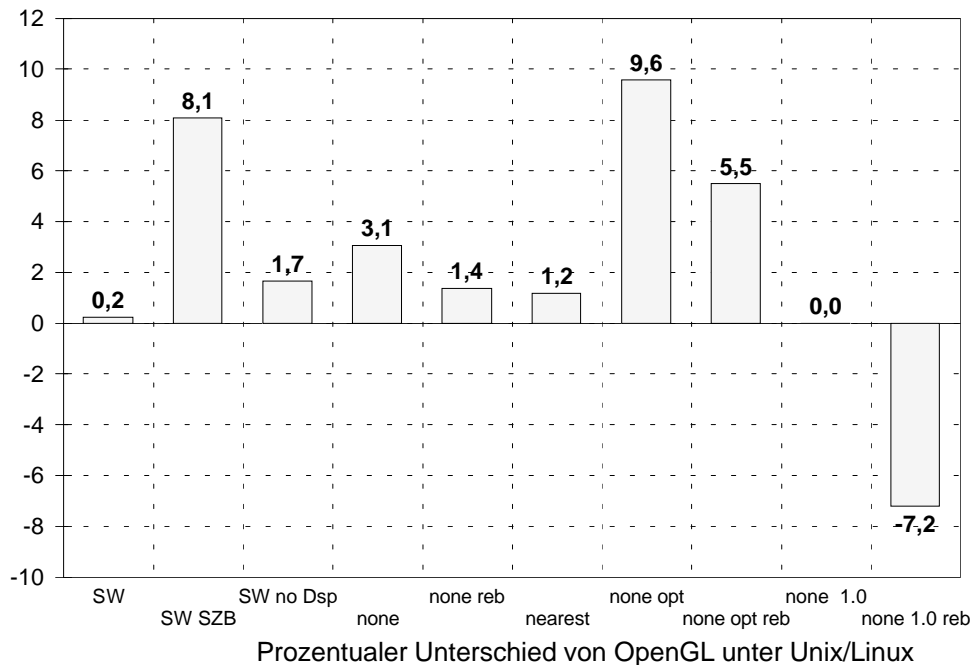
2. Die Verwendung von Triangle Strip Lists unter OpenGL bringt eine Geschwindigkeitssteigerung gegenüber der alten Routine von fast 17%. Auch dann, wenn sich die Szene ständig ändert, fällt der Geschwindigkeitsgewinn mit 14% noch deutlich aus.

Die einzige signifikante Verlangsamung bei Verwendung der neuen OpenGL-Routinen ergibt sich dort, wo nur OpenGL 1.0 ohne SGI-Extensions zur Verfügung steht und die Szene immer wieder erneut aufgebaut werden muß. Hier sind die neuen Routinen um rund 15% langsamer als die alten, was einzig und alleine darauf zurückgeführt werden kann, daß bei Verwendung der neuen Routinen, wie bereits beschrieben, mehr Funktionsaufrufe durchzuführen sind. Da inzwischen jedoch OpenGL 1.1 die verbreitetste Version ist, kann man davon ausgehen, daß diese Verlangsamung nicht zu kritisch ist.

Der Softwarerenderer hat ebenfalls leicht an Geschwindigkeit eingebüßt, was jedoch mit Blick auf die Ergebnisse unter Linux nicht ganz nachzuvollziehen ist. Hier scheint der Compiler die neue Struktur nicht so gut optimieren zu können wie die alte Struktur.

## 9.2.2 Ergebnisse unter Unix/Linux

Ergebnisse auf dem PC unter Linux		
	Linux	Linux, 208
Software <sup>1</sup>	2:08.9	2:09.2
Software SZB <sup>2</sup>	2:41.6	2:55.8
Software, no display <sup>3</sup>	1:10.9	1:12.1
none <sup>4</sup>	0:25.3	0:26.1
none, reb.all <sup>5</sup>	0:28.7	0:29.1
nearest <sup>6</sup>	0:33.5	0:33.9
none, opt <sup>7</sup>	0:23.6	—
none, opt, reb.all <sup>8</sup>	0:27.5	—
none, OpenGL 1.0 <sup>9</sup>	0:26.1	0:26.1
none, OpenGL 1.0, reb.all <sup>10</sup>	0:31.2	0:29.1



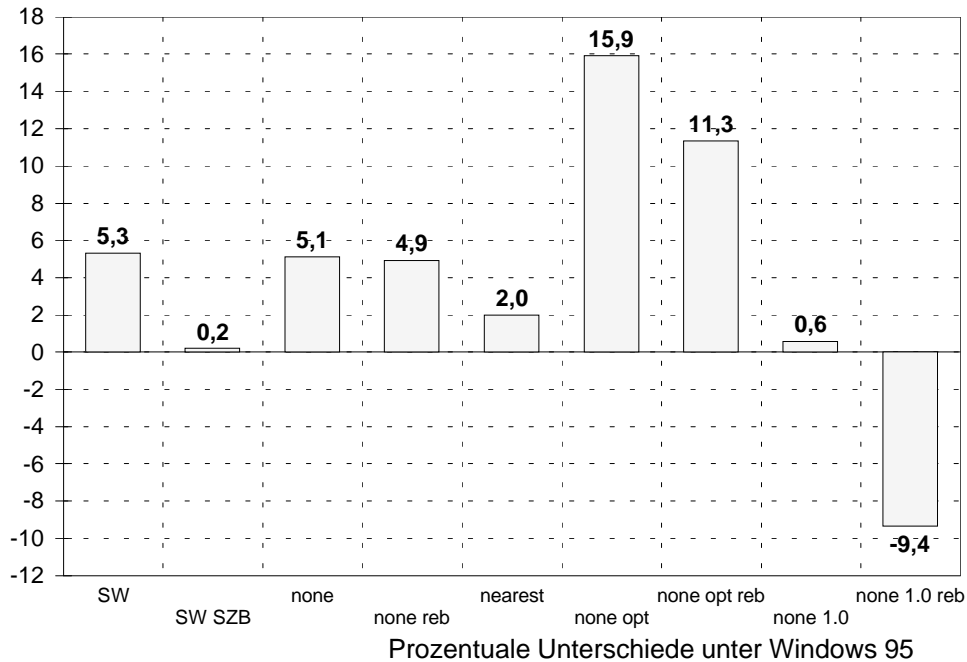
Bei Betrachtung der Ergebnisse unter Linux und beim Vergleich dieser mit den Ergebnissen unter UNIX/SGI wird deutlich, wie sehr der Geschwindigkeitsunterschied zwischen alter und neuer Version von Cgi3D von der verwendeten Hardware abhängig ist. Unter Linux mit Mesa 2.6 ergibt sich für alle Testwerte bis auf einen eine meist deutliche Geschwindigkeitssteigerung (zwischen 1.2% und 9.6%)

Nur bei Verwendung der OpenGL 1.0 Routinen mit dem neuen Renderer ergibt sich ein deutlicher Geschwindigkeitsverlust von ca 7%. Dieser Messwert ist aber für die Praxis nicht relevant, da Mesa in der Version 2.6 OpenGL 1.1 unterstützt und somit niemand hier auf Version 1.0 zurückgreifen muß.

Wie bereits im vorigen Kapitel erwähnt, wurde unter Linux auch der Softwarerenderer beschleunigt (bei single line Z-Buffer um ca. 8%). Dies verdeutlicht, daß der Geschwindigkeitsgewinn (oder Verlust) nicht alleine von der reinen Implementierung innerhalb von Cgi3D abhängt. Vielmehr spielen auch die Optimierungsmöglichkeiten des Compilers eine Rolle.

### 9.2.3 Ergebnisse unter Windows 95

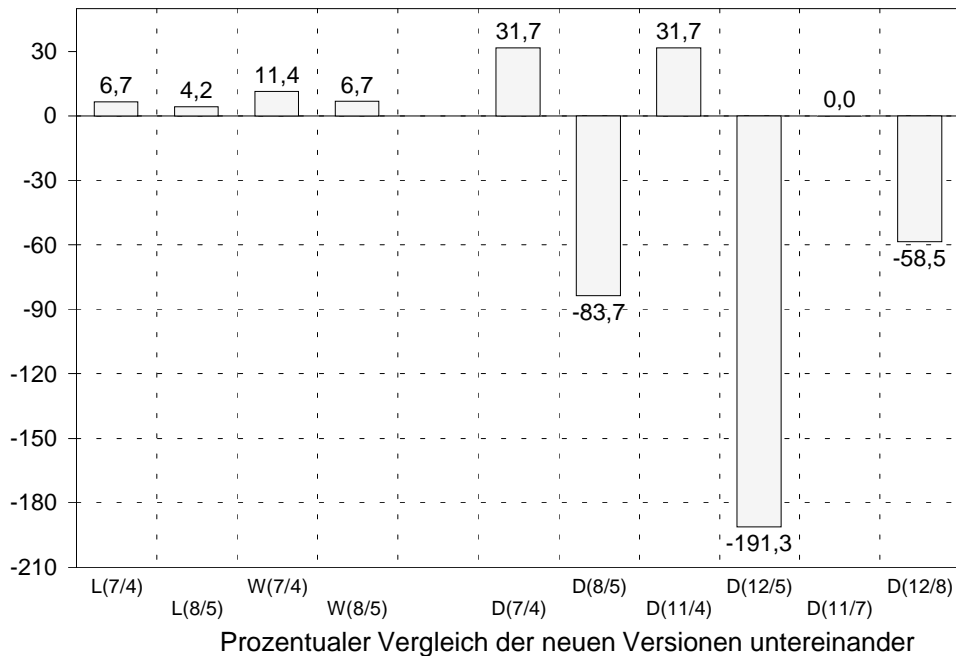
Ergebnisse auf dem PC unter Windows 95			
	Direct3D	OpenGL	OpenGL, 208
Software <sup>1</sup>	4:42.1	4:41.1	4:56.9
Software SZB <sup>2</sup>	4:09.2	4:09.2	4:09.7
none <sup>4</sup>	0:04.1	0:16.7	0:17.6
none, reb.all <sup>5</sup>	0:08.0	0:19.3	0:20.3
nearest <sup>6</sup>	0:04.2	0:29.4	0:30.0
none, opt <sup>7</sup>	0:02.8	0:14.8	—
none, opt, reb.all <sup>8</sup>	0:14.7	0:18.0	—
none, OpenGL 1.0 <sup>9</sup>	—	0:17.5	0:17.6
none, OpenGL 1.0, reb.all <sup>10</sup>	—	0:22.2	0:20.3
none, D3D, array opt <sup>11</sup>	0:04.1	—	—
none, D3D, array opt, reb.all <sup>12</sup>	0:19.8	—	—



Unter Windows sehen die Ergebnisse für OpenGL ähnlich aus wie unter Unix/Linux (Performancegewinn gegenüber der alten Version von 2% bis 16%). Auch hier fällt wieder der Wert für die ständige neue Übergabe der Szene unter reinem OpenGL 1.0 aus der Reihe. Da jedoch auch unter Windows bereits OpenGL 1.1 die Standardversion ist, ist dies für die Praxis nicht relevant.

Die Werte des Direct3D-Renderers können nicht direkt mit den übrigen Werten verglichen werden. Erstens gab es in der alten Version von Cgi3D noch keinen Direct3D-Renderer und zweitens ist dies der einzige Renderer, der auf der Testplattform die Grafikkarte verwendet. Hierdurch lassen sich die enormen Zeitunterschiede erklären (Direct3D ist bis zu 400% schneller als OpenGL).

### 9.3 Vergleich der neuen Treiber untereinander



Neben dem Vergleich der alten mit der neuen Version von Cgi3D ist es auch interessant zu sehen, wie viel Geschwindigkeitsgewinn (oder Verlust) die optimierten Treiber im Vergleich zu den nichtoptimierten Treibern unter Unix und Windows erreichen. Im obigen Bild bezeichnet **L** die OpenGL-Treiber unter Linux, **W** die OpenGL-Treiber unter Windows 95 und **D** den Direct3D-Treiber unter Windows 95. Die Zahlen in Klammern geben jeweils die verglichenen Testkriterien an und entsprechen den Fußnoten in den vorigen Tabellen. Eine genaue Beschreibung dieser Fußnoten kann in Kapitel 9.2 nachgelesen werden. Auf einen Vergleich der Werte von UNIX/SGI wurde bewußt verzichtet, da diese tendentiell ähnlich zu den Werten unter Linux und Windows 95 mit OpenGL sind.

Unter OpenGL gleichen sich die Ergebnisse unter Linux und Windows 95 im wesentlichen. Hier ergibt sich ein Performancegewinn der optimierten Treiber (die Triangle Strip Lists verwenden) gegenüber den normalen Treibern (welche Arrays zur Datenübergabe verwenden) zwischen 4% und 11%. Windows schneidet hierbei allgemein etwas besser ab, was auf die unterschiedlichen OpenGL-Treiber zurückzuführen sein dürfte.

Unter Direct3D ergibt sich durch die Verwendung beider Versionen des optimierten Treibers eine beachtliche Geschwindigkeitssteigerung von fast 32%. Genauso fällt aber direkt auf, daß der optimierte Treiber erheblich länger zur Datenübergabe an die Hardware benötigt. Die vom optimierten Treiber verwendete Version der Datenübergabe an Direct3D ist also nicht dazu geeignet, sich ständig ändernde Szenen auszugeben. Die Werte D(7/4) und D(11/4) belegen außerdem, daß es auf der verwendeten Hardware keinen Unterschied macht, ob die Arrays bereits optimiert sind oder nicht. Da jedoch die Erzeugung und Übergabe der optimierten Arrays (Wert D(12/8)) nochmals etwa 58% langsamer geschieht als die normale Übergabe der Arrays an den optimierten Treiber, scheint die Arrayoptimierung nicht sinnvoll zu sein. Wie im Kapitel 5.3 beschrieben, ist diese Arrayoptimierung im optimierten Treiber deshalb standardmäßig deaktiviert.

## Kapitel 10

# Ausgewählte Möglichkeiten zur weiteren Verbesserung

In diesem Kapitel sollen, getrennt nach Hardwareplattformen, einige Punkte aufgezeigt werden, welche in der neuen Version von Cgi3D aus Sicht des Autors noch verbessert werden könnten.

Diese Verbesserungsvorschläge verlangen teilweise eine neue Datenstruktur für die Objekte oder den Szenengraphen, teilweise bauen sie auch auf (gewünschten) Erweiterungen von Renderbibliotheken auf.

### 10.1 Allgemeine Verbesserungen

Zur Zeit ist es für den Benutzer von Cgi3D nur vorgesehen, eine komplette Szene auszugeben. Mit der neuen Implementierung von Cgi3D wäre es denkbar, auch vom Benutzer aus einzelne Objekte auszugeben. Diese Option wurde jedoch bewußt nicht in die aktuelle Version von Cgi3D aufgenommen, da der Autor es für sinnvoller erachtet, daß der Benutzer auch einzelne Objekte in einer Szene anordnet und dann die Szene darstellt. Dies hat den Vorteil, daß auch Hardwarerenderer, welche nur die Ausgabe einer gesamten Szene unterstützen (z.B. der single line Z-Buffer Softwarerenderer), für diese Art der Ausgabe verwendet werden können. Da es auch bei der Ausgabe einzelner Objekt durch den Benutzer nötig wäre, daß dieser bei jeder Darstellung alle Objekte an Cgi3D übergibt, muß der Benutzer diese sowieso selber speichern. Dieses kann dann auch direkt in einer Szenenbeschreibung geschehen.

Eine Verbesserung der Darstellung von Animationen durch Cgi3D wäre dadurch möglich, daß der Benutzer für die Darstellung einer Szene eine maximal erlaubte Ausgabezeit vorschreiben könnte. Zur Zeit erfolgt die Ausgabe von Cgi3D nicht zeitorientiert, d.h. Cgi3D zeigt unabhängig von der dafür benötigten Zeit einfach die komplette Szene an. Um eine gleichmäßige Animation, also eine bestimmte Anzahl von dargestellten Bildern pro Sekunde zu ermöglichen, müßte es möglich sein, Cgi3D eine maximale Zeit für die Anzeige zuzuteilen. Die Regulierung innerhalb von Cgi3D könnte dadurch erfolgen, daß in einem ersten Testlauf (also einer Art Initialisierungsphase) ohne Anzeige der Objekte auf dem Bildschirm (also nur im Backbuffer bei doublebuffer Darstellung) gemessen wird, wie lange die Hardware für die Anzeige jedes Objektes benötigt. Wenn die Zeit, die für die gesamte Darstellung der Szene benötigt wird, länger ist als die Zeit, die der Benutzer für die Darstellung erlaubt, dann müßten einzelne Objekte, die besonders viel Zeit beanspruchen, mit einer größeren Triangulierung, d.h. mit weniger Polygonen, dargestellt werden. Führt auch dieser Test nicht zum gewünschten Erfolg, so könnten einzelne Teiläste des Szenengraphen nur durch ihre Boundingbox dargestellt werden.

Hierdurch sollte es nun möglich sein, die von Benutzer geforderte Anzeigzeit nicht zu überschreiten. Die Darstellung der Szene wird durch diese Methode zwar vergrößert, dadurch aber eine gleichmäßige Animation gewährleistet.

Diese Veränderungen könnten im Cgi3D-Objekt `t_RenderScene` in der Methode `renderScene()` zentral für alle Hardwareplattformen vorgenommen werden.

In Kapitel 6.2 wurde ein Algorithmus beschrieben, der zur Simulation von 3D-Texturen mittels 2D-Texturen getestet wurde. Mittels einer Erweiterung dieses Algorithmus sollte es möglich sein, die dort beschriebenen Probleme zu beheben und ein Ergebnis zu erzielen, welches exakter ist als das des aktuellen Algorithmus.

Der dort beschriebene Algorithmus bearbeitet das Objekt Polygon für Polygon und ermittelt für jeden Eckpunkt des Polygons den 2D-Texturwert. Da bei den Approximationen der Objekte durch Polygone die Polygone meist so groß sind, daß sie mehrere Texturkoordinaten überdecken, müßte man zur Verbesserung des Ergebnisses also die 3D-Textur auch innerhalb der Polygone abfragen und in die 2D-Textur übertragen. Man müßte also durch Interpolation über die gegebenen Eckpunkte des Polygons das gesamte Polygon abtasten und für jede so erhaltene Koordinate den 3D-Texturwert in die 2D-Textur eintragen. Da die 3D-Textur für den gesamten Raum, und nicht nur für die Oberfläche des Objektes definiert ist, stört es dabei nicht, daß die durch Interpolation erhaltenen Werte nicht unbedingt auf der mathematischen Oberfläche des Objektes liegen. Bei Verwendung dieses Algorithmus dürften besonders Stellen, an denen das Objekt stark gekrümmt ist, genauer Abgetastet werden als mit dem zur Zeit verwendeten Algorithmus.

Bei der aktuellen Implementierung von Cgi3D ist es nur möglich, das Ergebnis des Softwarerenderers als Datei abzuspeichern. Theoretisch ist es auch denkbar, die Ausgabe der Hardwarerenderer in eine Datei zu speichern. Dies könnte auch, bei Verwendung von Hintergrundspeicher, ohne sichtbare Anzeige für den Benutzer geschehen. Jede Grafikbibliothek bietet eine Möglichkeit, das dargestellte Bild auszulesen. Dies kann jedoch recht langwierig sein, wenn das Bild nur Pixel für Pixel ausgelesen werden kann.

## 10.2 OpenGL

Die Darstellung von Transparenz ist in der aktuellen Version von Cgi3D dadurch gelöst, daß sogenannte Stipple Patterns verwendet werden. Das ist ein Textur, welche das Objekt stückweise durchsichtig macht. Dadurch entsteht eine Art Schachbrettmuster auf dem Objekt, bei dem alle schwarzen Felder durchsichtig sind.

Falls die Hardware dies unterstützt, wäre die Darstellung von transparenten Objekten durch die Verwendung eines Alpha-Buffers erheblich realitätsnäher. Der Alpha-Buffer ist ein zusätzlicher Speicher, in welchem die Alpha-Werte eines jeden dargestellten Pixels gespeichert werden. Wird nun über das bereits dargestellte Pixel ein neues Pixel gezeichnet, so werden die Farben dieser beiden Pixel ihren Alpha-Werten und ihrer räumlichen Anordnung entsprechend multipliziert. Wird zum Beispiel in weißes Pixel mit dem Alpha-Wert 0.5 vor ein schwarzes Pixel gezeichnet, so erhält das Zielpixel die Farbe Grau (oder 50% Weiß), da das weiße Pixel 50% der Farbe des darunterliegenden Pixels darstellt. Auf diese Weise erscheinen die Objekte dann durchsichtig.

Da diese Funktion aber nur unterstützt wird, wenn die Hardware dieses auch anbietet, ist die Verwendung des Alpha-Buffers in Cgi3D nur sinnvoll, wenn solche Hardware verfügbar ist. Die zum Zeitpunkt des Schreibens der Diplomarbeit verwendeten Computer unterstützen keinen Alpha-Buffer. Eine Implementierung des Alpha-Buffers im Hauptspeicher bei gleichzeitiger Verwendung der Grafikhardware für die Anzeige auf dem Bildschirm ist nicht möglich, da dies einen schreibenden Zugriff auf den Hauptspeicher des Computers voraussetzen würde.

Mit dem OpenGL Softwaretreiber Mesa wäre unter Linux die Verwendung des Alpha-Buffers möglich gewesen. Neben der Verfügbarkeit des Alpha-Buffers muß jedoch auch die

Szene bei jeder Ausgabe sortiert und alle Objekte von hinten nach vorne (aus Sichtweise der aktuellen Kameraposition) ausgegeben werden. Dies ist nötig, damit auch bei übereinanderliegenden Objekten eine korrekte Farbberechnung stattfinden kann, da der Test des Tiefenwertes vor dem Alpha-Test stattfindet. Somit erscheint die Verwendung des Alpha-Buffers nur sinnvoll, wenn es um eine qualitativ hochwertige Ausgabe der Szene mittels OpenGL geht. Deshalb sollte dies nicht in den jetzigen OpenGL-Renderer einfließen. Vielmehr ist es sinnvoll, einen neuen OpenGL-Renderer abzuleiten und diesen auf Ausgabequalität zu optimieren.

## 10.3 Direct3D

Auch unter Direct3D wäre die Unterstützung des Alpha-Buffers, wie unter OpenGL bereits erwähnt, die beste Möglichkeit, transparente Objekte darzustellen. Da dies aber auch bei Direct3D nur mit Hardwareunterstützung möglich ist (eine Softwareemulation des Alpha-Buffers bei gleichzeitiger Verwendung der Beschleunigerhardware ist bei Direct3D ebenfalls nicht möglich) und die zur Entwicklung von Cgi3D zur Verfügung stehende Hardware dies nicht unterstützte, konnte die Verwendung des Alpha-Buffers nicht implementiert werden. Leider konnte der Dokumentation von Direct3D nicht entnommen werden, ob unter Direct3D ebenfalls eine Sortierung der Szene ihrer Tiefe nach nötig ist, hiervon geht der Autor aber aus, da die Voraussetzungen dieselben wie unter OpenGL sind.

Wie bereits in Kapitel 5.3.2 erwähnt, wäre eine effektivere Reduktion der Arrays als zur Zeit implementiert durch einen direkten Zugriff auf die internen Daten des auszugebenden Objektes möglich. Die eleganteste Lösung hierfür wäre, wenn die Klasse `t_BRep`, welche die Polygonrepräsentation des 3D-Objektes enthält, die Koordinaten und Normalen selber numerieren könnte. Hierdurch würde auch ein zusätzliches Speichern der Daten im Direct3D-Renderer unnötig. Dies wäre dadurch zu realisieren, daß in der Template-Klasse `t_DList`<sup>1</sup> zwei weitere Funktionen eingeführt würden. Mit der ersten Funktion, sie sei hier einmal `generateHash` genannt, würde eine Hashtabelle für alle Elemente der `t_DList` generiert. Für den Zugriff auf die einzelnen Elemente der `t_DList` sollte dann der Array-Operator `[ ]` definiert werden. Mittels dieser beiden Funktionen wäre es möglich, auf die Liste aller Ecken und Normalen eines Objektes zuzugreifen, ohne in Cgi3D die Daten nochmals zwischenspeichern zu müssen. Weil sich zum Zeitpunkt des Zugriffes durch Direct3D auf die `t_DList` diese nicht mehr ändert, würde diese Erweiterung nicht gegen das Konzept der dynamischen, doppelt verketteten Liste verstoßen. Da dieses Konzept jedoch die Erweiterung einer Klasse erfordert, welche nicht zu Cgi3D gehört, und außerdem die Anwendung der Array-Optimierung unter Direct3D auf der verwendeten Testplattform keine Geschwindigkeitssteigerung verspricht, hat der Autor auf eine Realisierung dieses Ansatzes verzichtet.

## 10.4 XGL

Unter XGL wäre die Verwendung von Triangle Strip Lists für die Datenübergabe an die Hardware möglich. Hierzu könnte der optimierte OpenGL-Renderer auf XGL umgesetzt werden. Eine sinnvollere Lösung wäre es sicherlich, eine allgemeine Routine zur Erzeugung von Triangle Strip Lists zu implementieren, da die Voraussetzungen und Beschränkungen, die in Kapitel 5.2 für OpenGL genannt wurden, auch auf XGL zutreffen.

Der Gründe, daß diese Verbesserung nicht in den aktuellen XGL-Renderer eingeflossen sind, liegt insbesondere daran, daß fast alle an der Universität vorhandenen Maschinen nur ein 8 Bit Display und keine Hardwarebeschleunigung besitzen. Die Ausgabe mittels der

---

<sup>1</sup>Eine `t_DList` ist eine doppelt verkettete Liste, auf welche zur Zeit nur über Iteratoren zugegriffen werden kann.

XGL Softwareemulation ist dabei langsamer als die Ausgabe mittels des OpenGL Softwarerenderers. Zur Entwicklung der neuen Version von Cgi3D stand zeitweise ein SUN-Computer mit 24 Bit XGL-Hardwarebeschleunigung zur Verfügung, auf dem Cgi3D auch getestet wurde. Da dies aber ein Mitarbeitercomputer ist, mußte darauf verzichtet werden, andere als strukturelle Verbesserungen gegenüber der alten Version von Cgi3D vorzunehmen.

Desweiteren unterstützt SUN selber inzwischen auch OpenGL. Deshalb erschien es dem Autor sinnvoll, sich eher um die Verbesserung des Open GL-Renderers zu kümmern als den XGL-Renderer weiter zu verfeinern, da Optimierungen des OpenGL-Renderers einen Performancegewinn auf allen Testplattformen erzielen.

## 10.5 Softwarerenderer

Zum gegenwärtigen Zeitpunkt ist die Darstellung von Texturen nur bei der Ausgabe mittels Phongshading korrekt. Dies liegt daran, daß bei allen anderen Beleuchtungsarten die Farbe der Textur nur in den Ecken der Dreiecke bestimmt wird und die Farben innerhalb des Dreiecks durch Interpolation berechnet werden. Für eine korrekte Darstellung der Textur muß diese für jeden dargestellten Punkt des Dreiecks abgefragt werden und es darf nur die Helligkeit innerhalb des Dreiecks interpoliert werden. Da diese Vorgehensweise jedoch einige Umstellungen innerhalb des Softwarerenderers, der komplett von der alten Version von Cgi3D übernommen wurde, bedeuten würden, wurde auf die Implementierung verzichtet.

Im Vergleich zur Softwareemulation von OpenGL mittels Mesa ist die Ausgabe mit dem Softwarerenderer von Cgi3D sehr langsam, was keineswegs nur an der recht langsamen Ausgabe des berechneten Bildes durch Cgi liegt (ein genauer Vergleich hierzu ist in Kapitel 9 zu finden). Dies ist zum Teil sicherlich dadurch zu begründen (und auch zu rechtfertigen), daß Cgi3D als ein offenes System ausgelegt ist. Dies bedeutet, daß es z.B. möglich ist, neue Beleuchtungsgleichungen zu implementieren. Sollte die Geschwindigkeitsoptimierung ein Ziel zukünftiger Verbesserungen von Cgi3D sein, so ist es sicherlich sinnvoller, den Softwarerenderer komplett neu zu designen und zu implementieren, als zu versuchen, ihn intern zu verbessern.

## Kapitel 11

# Ausblick auf neue Versionen der Renderbibliotheken

In diesem Kapitel soll ein Überblick darüber gegeben werden, welche neuen Funktionalitäten die gerade erst veröffentlichten neuen Versionen der Renderbibliotheken OpenGL und Direct3D bieten. Allerdings werden nur die Funktionen berücksichtigt, die aus Sicht von Cgi3D interessant sind.

Eine Unterstützung dieser neuen Versionen in der aktuellen Fassung von Cgi3D war nicht möglich, da die Neuerungen erst während des Entstehens dieser Diplomarbeit spezifiziert wurden und zum Teil noch gar nicht implementiert sind. So gibt es bis zum Abschluß der Arbeit noch keine OpenGL 1.2 Version für Windows, unter Unix unterstützt nur Mesa 3.0 die neuen Funktionalitäten per Software. DirectX 6.0 ist erst etwa zwei Monate vor Ende der Diplomarbeit von Microsoft freigegeben worden. Hierzu werden aber (zumindest für die vom Verfasser verwendete Grafikkarte) auch neue Grafiktreiber benötigt, welche zum aktuellen Zeitpunkt noch nicht verfügbar sind.

Um eine Unterstützung von Direct3D auch unter Windows NT 4.0 zu ermöglichen, wurde der Treiber noch auf Version 3.0 von Direct3D aufgesetzt. Diese ist zumindest in einer Softwareversion im Service Pack 3 von Windows NT enthalten.

### 11.1 OpenGL 1.2

Bei OpenGL 1.2[[Seg98](#)] wurden 3D-Texturen eingeführt. Diese werden dadurch realisiert, daß eine Menge von 2D-Texturen übereinandergelegt werden und dann durch tridirektionale Interpolation der korrekte Texturwert für das darzustellende Pixel ermittelt wird.

Für die Verwendung dieser 3D-Texturen in Cgi3D müssen aus der prozeduralen Textur von Cgi3D mehrere 2D-Texturen erstellt werden, welche OpenGL dann zur Darstellung der 3D-Textur verwendet. Dieser Schritt ist jedoch sehr viel schneller und genauer zu bewerkstelligen als die zur Zeit implementierte Lösung zur Simulation von 3D-Texturen, da hierbei nicht das Objekt, sondern nur die 3D-Textur abgetastet werden muß.

### 11.2 DirectX 6.0

Unter DirectX 6.0[[Mic98](#)] ist es nun endlich möglich, auch im Retained Mode Triangle Strip Lists zu erstellen. Hierzu ist im neuen `MashBuilder3`<sup>1</sup> die Funktion **AddTriangles**

---

<sup>1</sup>`MashBuilder` ist unter Direct3D das Objekt, welches alle Polygone des darzustellenden Objektes enthält und verwaltet. Ebenso verwaltet es deren Farbe und Textur

hinzugekommen. Mit Hilfe dieser Funktion ist es nun möglich, Triangle Strip Lists, Triangle Fans<sup>2</sup> und einzelne Dreiecke in einem Array zusammenzufassen. Dabei ist es sogar möglich, Triangle Strip Lists und Triangle Fans zu mischen. Somit wäre bei passendem Algorithmus eine noch kompaktere Darstellung als mit reinen Triangle Strip Lists möglich.

Es wäre also denkbar, daß die in Kapitel 5.2.2 aufgeführten Probleme nicht unbedingt zum Abbruch der aktuellen Triangle Strip List führen müssen. Vielmehr könnte dort, wo die Normalen der übereinstimmenden Punkte zweier benachbarter Dreiecke nicht mehr übereinstimmen, unter Umständen mit einem Triangle Fan fortgefahren werden. Hierdurch würde dann quasi eine Kurve in der Triangle Strip List entstehen. An diesen Triangle Fan könnte dann wiederum eine neue Triangle Strip List angefügt werden, so wie dies in Bild 7 dargestellt ist.

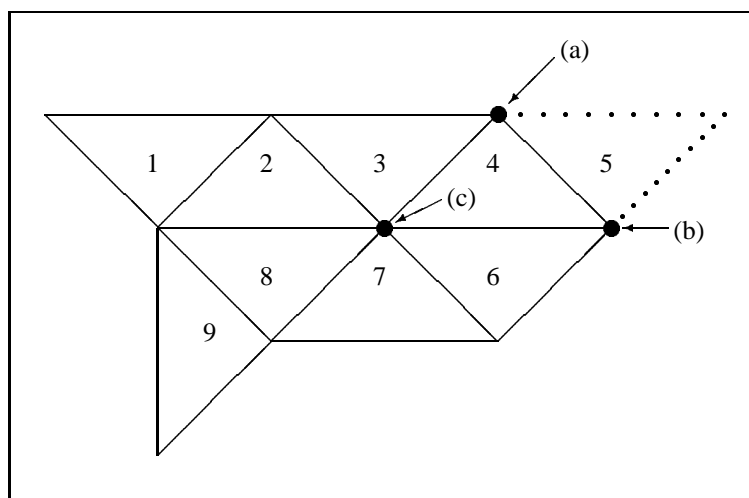


Bild 7: Triangle Strip Liste, gefolgt von einem Triangle Fan

Um möglichst lange Listen aus Triangle Strips und Triangle Fans zu erzeugen ist es nun nicht möglich, die Listen in einem Durchgang zu erzeugen. Vielmehr muß beim ersten Durchgang nur überprüft werden, an welcher Stelle die Triangle Strip List aufhört und ob dort ein Triangle Fan angefügt werden kann. Im Beispiel in Bild 7 sei Dreieck Nr. 5 nicht konform mit der Triangle Strip List, d.h. mindestens eine Normale in Punkt (a) oder (b) ist nicht identisch mit der Normalen in Punkt (a) bzw. (b) von Dreieck Nr. 4. Da aber festgestellt wurde, daß die Normale in Punkt (c) für die Dreiecke 3, 4, 6 und 7 identisch ist, und daß außerdem die Normalen von Dreieck 4 und 6 in Punkt (b) identisch sind (sowie die Normalen in den geteilten Punkten von Dreieck 3 und 4 bzw. 6 und 7) ist es nun möglich, einen Triangle Fan aus den Dreiecken 3, 4, 6 und 7 zu erzeugen. Der gemeinsame aufspannende Punkt für diesen Triangle Fan ist Punkt (c). Somit wird die Triangle Strip List, bestehend aus den Dreiecken 1 und 2 fortgesetzt mit einem Triangle Fan bestehend aus den Dreiecken 3, 4, 6 und 7, wobei die letzte Kante des Triangle Strips gleichzeitig als erste Kante des Triangle Fans dient und der letzte Punkt der Triangle Strip List der aufspannende Punkt des Triangle Fans ist. Im Beispiel aus Bild 7 folgt dann an das Triangle Fan wieder eine Triangle Strip Liste bestehend aus den Dreiecken 8 und 9, welche wiederum direkt an den Triangle Fan ansetzt. Es müssen also auch beim Wechsel zwischen Triangle Strip List und Triangle Fan keine Punkte zweimal in das Array übernommen werden.

Des Weiteren ist es mit dem MashBuilder3 nun auch möglich, in die Arrays für die Triangle Strip List bzw. den Triangle Fan direkt die Texturkoordinaten zu speichern. Somit

<sup>2</sup>Ein Triangle Fan ist eine Menge von Dreiecken, welche alle einen gemeinsamen Punkt und jeweils benachbarte Dreiecke zwei gemeinsame Punkte besitzen. In den gemeinsamen Punkten müssen die Normalen ebenfalls übereinstimmen. Ein Beispiel wäre die Approximation einer Kreisfläche, wobei der gemeinsame Punkt der Mittelpunkt des Kreises ist.

entfällt auch die zeitaufwendige Übergabe der Texturkoordinaten an Direct3D, wie dies bei der Array-Optimierung mit Direct3D 3.0 noch nötig ist (dies ist in Kapitel 5.3 näher erläutert). Dies könnte auch für die Übergabe der Daten in Form von einfachen Dreiecken an Direct3D genutzt werden.

Die Implementierung der Verwendung von Triangle Strip Lists bzw. Triangle Fans ist jedoch nur dann sinnvoll, wenn gleichzeitig die unter 10.3 angegebenen Änderungen an der Template-Klasse `t_DList` des MRT vorgenommen werden, da die Numerierung der Koordinaten und Normalen für die sinnvolle Verwendung des `MashBuilder3` Voraussetzung ist.

## Kapitel 12

# Zusammenfassung der Ergebnisse

Die neue Struktur von Cgi3D bietet zusammen mit den vorgenommenen Erweiterungen gegenüber der alten Version eine Reihe von Vorteilen:

- **Portierbarkeit**

Durch die Aufteilung von Cgi3D in einen hardwareunabhängigen und einen hardwareabhängigen Teil muß zur Portierung von Cgi3D auf eine neue Hardwareplattform nur der Teil neu implementiert werden, welcher direkt auf die Hardware zugreift. Zur Unterstützung verschiedener Arten von Hardwarebeschleunigern folgt die Aufteilung jedoch nicht rein logischen Erwägungen. Einige Hardwarebibliotheken implementieren auch eigentlich hardwareunabhängige Funktionen selber (z.B. optimieren sie den gesamten Szenengraphen). Aus diesem Grund wurden Teile des Renderers, die im Grunde genommen hardwareunabhängig sind, in die Basisklasse des hardwareabhängigen Teils von Cgi3D verlegt. Eine genaue Aufstellung der Aufteilung ist in Kapitel 3 nachzulesen.

- **Erweiterbarkeit**

Durch die Einführung einer Basisklasse für alle Hardwarerenderer müssen Erweiterungen, die von allgemeiner Natur sind, nur einmal implementiert werden und sind danach direkt auf allen Plattformen verfügbar. Ein gutes Beispiel hierfür ist die Simulation von 3D-Texturen durch 2D-Texturen. Die bereits vorgenommenen Erweiterungen an Cgi3D sind in Kapitel 6 zusammengestellt.

- **Zukunftssicherheit**

Da alle Renderer auf Objektbasis arbeiten, ist es ohne groß Probleme möglich, die für die Zukunft des MRT geplanten Änderungen an der Szenenbeschreibung in Cgi3D zu implementieren. So ist z.B. ein Umschalten des Schattierungsmodelles innerhalb der Szenenbeschreibung möglich. In Kapitel 10 sind einige mögliche Erweiterungen von Cgi3D angeführt.

- **Geschwindigkeitssteigerung**

Durch die Verwendung neuerer Versionen der Renderbibliotheken konnte die Geschwindigkeit der Ausgabe im Schnitt um 10% gesteigert werden. Die vorgenommenen Änderungen zur Performanceoptimierung sind in Kapitel 5 aufgeführt. In Kapitel 9 ist ein detaillierter Geschwindigkeitsvergleich der neuen gegenüber der alten Version von Cgi3D nachzulesen.

- **Unterstützung neuer Hardware**

Die Unterstützung von Direct3D erlaubt nun auch mit preisgünstigen Grafikkarten die hardwarebeschleunigte Ausgabe unter Windows 95. Für diese Karten ist meist kein hardwarebeschleunigter OpenGL-Treiber verfügbar. In Anhang A befindet sich eine Auflistung aller zur Zeit von Cgi3D unterstützten Renderbibliotheken.

- **Kontinuität**

Bei allen Veränderungen und Optimierungen wurde darauf geachtet, die bisher für den Benutzer von Cgi3D gegebene Interfacestruktur beizubehalten. Nur dort, wo dies, meist zur Performancesteigerung, erforderlich war, wurden die Parameter geändert oder Funktionen umbenannt. Eine Erläuterung dieser Änderungen wird am Ende von Kapitel 2.3 gegeben. Durch die geringe Zahl der Änderungen ist für einen Benutzer, der die alte Version von Cgi3D bereits kennt, relativ leicht, seine Programme auf die neue Version anzupassen.

## Kapitel 13

# Hinweise zur Portierung auf neue Hardwareplattformen

In diesem Kapitel wird beschrieben, wie man bei der Portierung von Cgi3D auf eine neue Hardwareplattform vorgehen sollte. Um möglichst anschaulich zu sein wird die Methode beschrieben, nach welcher der Autor die Portierung auf Direct3D vorgenommen hat.

Zunächst einmal muß ein neues Objekt definiert werden, welches von `t_RenderScene` abgeleitet wird. `t_RenderScene` bildet das Grundgerüst für alle Hardwareplattformen. Dieses Objekt muß nun Cgi3D bekannt gemacht werden, damit die neue Plattform vom Benutzer auch ausgewählt werden kann. Hierzu muß in der Datei `t_renscn.hh` der Namen der neuen Grafikkhardware in der Struktur `t_GraphicHardware` eingetragen werden. Weiterhin muß in der Datei `t_cgi3dp.cpi` in der Funktion `useGraphicHardware` ein zusätzliches `case` Statement anhand der bereits vorhandenen Beispiele eingefügt werden. Soll die Auswahl der neuen Grafikkhardware auch mittels Environmentvariable möglich sein, so muß auch die Funktion `construct` von Cgi3D (ebenfalls in der Datei `t_cgi3dp.cpi`) entsprechend erweitert werden.

Nach diesen Vorarbeiten kann nun mit der eigentlichen Programmierung der Hardware begonnen werden. An dieser Stelle wird nur beschrieben, welche Routinen auf jeden Fall zu implementieren sind. Eine genaue Beschreibung der Funktionsparameter und eine Angabe, welche Funktionalität in welche Memberfunktion der neu abgeleiteten Klasse gehört, ist in Anhang B zu finden.

Für die erste Implementierung des Hardwarerenderers sind folgende Memberfunktionen des neuen Objektes zu implementieren:

Constructor, Destructor, `initHardware`, `swapBuffers`, `beginScene`, `endScene`, `initCamera`, `initLights`, `setMaterial`, `beginObject`, `endObject`, `addFace` und `renderArrays`.

Die Implementierung aller übrigen Funktionen kann zunächst einmal ignoriert werden, falls passende Standardwerte gesetzt werden z.B. Gouraud-Shading, Verwendung der Beleuchtung, keine Textur.

Wenn mit diesem Grundgerüst die Ausgabe funktioniert, kann man sich daran machen, die restlichen Funktionen zu implementieren. Angefangen von verschiedenen Beleuchtungsarten über Texturen bis hin zum Picking kann so nach und nach der Renderer vervollständigt werden. Da bereits das Grundgerüst funktioniert, sind eventuell auftretende Fehler aller Wahrscheinlichkeit nach einfacher einzugrenzen, als wenn man direkt alle Funktionen implementiert.

Allerdings sollte man von Anfang an im Auge behalten, was noch alles fehlt. Ansonsten kann es leicht passieren, daß man das Grundgerüst auf einer Funktionalität der Hardwarebibliothek aufbaut, welche kein Picking unterstützt oder welche das Einbinden von Texturen nur auf sehr komplizierte Weise zuläßt. Aus diesem Grund ist es sehr ratsam, sich vor der

Wahl der Hardwarebibliothek genau anzusehen, welche Funktionalität diese unterstützt und welche nicht. Der Autor hatte genau an dieser Stelle Probleme bei der Implementierung des Direct3D-Renderers. Der erste Ansatz, welcher in der dem Cgi3D-Renderer zugrunde liegenden Demo verwendet wurde, mußte leider wieder verworfen werden, da das Einbinden von Texturen dabei nur mit Geschwindigkeitseinbußen möglich war. Allerdings mußte bei der beschleunigten Übergabe der Daten, wie im Kapitel 5.3 beschrieben, doch wieder auf diesen Renderer zurückgegriffen werden, was dann zu den in Kapitel 5.3 beschriebenen Problemen mit den Texturen führte.

Zum Abschluß folgt nun noch eine Aufstellung, in welcher Reihenfolge die Standardimplementierung in `t_RenderScene` die einzelnen Funktionen bei der Darstellung eine Szene aufruft. Hiermit ist im Zusammenhang mit der Funktionsbeschreibung in Anhang B zu erkennen, welche Teile der Hardware in welcher Funktion zu setzen sind. Da dies von Hardware zu Hardware unterschiedlich ist, können dazu keine allgemeinen Angaben gemacht werden.

```
t_RenderScene::renderScene
  initHardware()
  beginScene()
  initCamera()
  initLights()
  for each Object:
    Mat = transformationmatrix(Object)
    displayObjectTM(Object, Mat)
    t_RenderScene::displayObjectTM
      if (beginObject(Object) == false) return false
      setMatrix(Mat)
      setTexture(...)
      setMaterial()
      for each Face:
        addFace(Face, Object)
      next Face
      renderArrays()
      endObject()
    t_RenderScene::displayObjectTM END
  next Object
  endScene()
t_RenderScene::renderScene END
```

## Anhang A

# Unterstützte Renderbibliotheken

Die Auswahl der Renderbibliothek erfolgt entweder durch Setzen der entsprechenden Environmentvariablen (diese ist in Anhang C beschrieben), oder durch Aufruf der Funktion **useGraphicHardware** von Cgi3D.

Die aktuelle Version von Cgi3D unterstützt folgende Renderbibliotheken:

- OpenGL

Für OpenGL sind zwei Renderer implementiert, **OpenGL** und **OpenGL\_TSL**, welche in Abhängigkeit der OpenGL-Version ausgewählt werden können.

Zum Zeitpunkt der Compilierung von Cgi3D wird die höchste vorhandene Version von OpenGL zur Übersetzung verwendet. Sollte auf dem Rechner, auf dem das Programm nachher ausgeführt wird, nur eine frühere Version vorhanden sein (z.B. es wurde unter OpenGL 1.1 compiliert, auf dem Rechner ist aber nur Version 1.0 installiert), so schaltet Cgi3D automatisch auf OpenGL 1.0 zurück und die Beschleunigung wird nicht verwendet. Ein Zurückschalten von OpenGL 1.1 auf OpenGL 1.0 mit `GL_EXT_vertex_array` ist nur auf einigen Plattformen möglich. Dies hängt von Hersteller der Include-Dateien für OpenGL ab, welche bei der Übersetzung von Cgi3D verwendet wurden. So ist bei Verwendung von Mesa für die Compilierung von Cgi3D möglich, bei Verwendung der Microsoft OpenGL Header jedoch nicht.

- OpenGL 1.0

Dies ist die erste Version von OpenGL, welche bereits von der alten Version von Cgi3D unterstützt wurde. Für Version 1.0 sind keine speziellen Beschleunigungsvarianten implementiert.

Dieser Renderer ist in dem Objekt **t\_RenderOpenGL** implementiert und wird als Hardware **OpenGL** in Cgi3D ausgewählt.

- OpenGL 1.0 mit `GL_EXT_vertex_array`

Dies ist OpenGL 1.0 mit der Erweiterung, welche es ermöglicht, die Objektdaten als Array an die Hardware zu übergeben. Dies wird in der neuen Version von Cgi3D zur Beschleunigung der Datenübergabe verwendet und ist in Kapitel 5.1 näher beschrieben. Die Erweiterung `GL_EXT_vertex_array` ist in der Version von OpenGL implementiert, welche auf SGI IRIX 6.2 standardmäßig installiert ist.

Dieser Renderer ist ebenfalls in dem Objekt **t\_RenderOpenGL** implementiert und wird gleichfalls als Hardware **OpenGL** in Cgi3D ausgewählt. Ist zum Zeitpunkt der Programmausführung nur die normale OpenGL Version 1.0 verfügbar, so wird automatisch auf den normalen OpenGL-Renderer zurückgeschaltet.

Neben der Möglichkeit, die Daten als Array zu übergeben, besteht weiterhin die Möglichkeit, die Polygone in Form von Triangle Strip Lists zu übergeben. Von dieser Möglichkeit wird zur Beschleunigung von Cgi3D Gebrauch gemacht. Das verwendete Verfahren ist in Kapitel 5.2 näher beschrieben.

Dieser Renderer ist im Objekt **t\_RenderOpenGL\_TSL** implementiert und wird als Hardware **OpenGL\_TSL** in Cgi3D ausgewählt. Bei Wahl dieses Renderers auf einem Computer, der nur OpenGL 1.0 installiert hat, wird automatisch auf den nicht beschleunigten OpenGL 1.0 Renderer zurückgeschaltet.

– OpenGL 1.1

Dies ist die zum Zeitpunkt der Diplomarbeit aktuelle Version. Zur Beschleunigung von Cgi3D bietet sie dieselben Möglichkeiten wie unter „OpenGL 1.0 mit GL\_EXT\_vertex\_array“ beschrieben.

Die Auswahl der Hardware erfolgt wie unter „OpenGL 1.0 mit GL\_EXT\_vertex\_array“ beschrieben.

• Direct3D 3.0

Unter Windows wird Direct3D 3.0[Mic97] für die Ausgabe mit Direct3D benötigt. Dies ist zwar eine recht alte Version, sie bietet aber den Vorteil, daß sie sowohl unter Windows 95, als auch unter Windows NT 4.0 mit Service Pack 3 vorhanden ist. Unter Windows NT wird zwar von Seiten Microsofts leider keine Hardwarebeschleunigung unterstützt, man kann aber den Softwaregestützten Renderer von Direct3D verwenden. Es wird nur eine fensterbasierte Ausgabe unter Direct3D unterstützt. Daneben bietet Direct3D auch eine full screen Ausgabe an. Da Cgi3D jedoch auf Cgi aufsetzt und die Unterstützung der full screen Ausgabe nur unter Umgehung von Cgi möglich wäre, wurde diese Möglichkeit nicht implementiert.

Direct3D bietet zwei verschiedene Modi, welche als Immediate Mode und Retained Mode bezeichnet werden.

Der Immediate Mode setzt direkt auf der Hardware auf. Er besteht im wesentlichen aus einer rendering Pipeline, in die alle Daten zur Ausgabe (Matrizen, Koordinaten) geschrieben werden müssen. Dieses Verfahren ist sehr aufwendig und fehlerträchtig.

Der Retained Mode kann als high level Schnittstelle von Direct3D bezeichnet werden. Die Programmierung im Retained Mode ist ähnlich der von OpenGL und der Programmierer braucht sich nicht um jede Kleinigkeit selber zu kümmern, wie dies im Immediate Mode nötig ist. Allerdings wird durch die Verwendung des Retained Mode ein Teil der möglichen Geschwindigkeit verschenkt. Dies wird jedoch durch die vereinfachte Programmierung im Retained Mode mehr als kompensiert.

Der Direct3D-Renderer ist im Objekt **t\_RenderDirect3D** implementiert und wird als Hardware **Direct3D** in Cgi3D ausgewählt.

Der im Kapitel 5.3 beschriebene optimierte Renderer für Direct3D ist im Objekt **t\_RenderDirect3D2** implementiert und wird als Hardware **Direct3D2** in Cgi3D ausgewählt.

- XGL

Auf SUN Computern wird neben der Verwendung von OpenGL die SUN-eigene Hardwarebeschleunigung XGL unterstützt. Hierbei gibt es jedoch keine speziellen Möglichkeiten zur Beschleunigung gegenüber der Ausgabe mit dem alten Cgi3D.

Dieser Renderer ist im Objekt **t\_RenderXGL** implementiert und wird als Hardware **XGL** in Cgi3D ausgewählt.

- Software

Die Darstellung durch Software ist auf jeder Plattform möglich. Wie im Kapitel 4 erläutert, gibt es zwei Softwarerenderer.

Der Renderer mit full screen Z-Buffer ist im Objekt **t\_RenderGen** implementiert und wird als Hardware **Software** in Cgi3D ausgewählt.

Der Renderer mit single line Z-Buffer ist im Objekt **t\_RenderGenSingleLine** implementiert und wird als Hardware **SoftwareSZB** in Cgi3D ausgewählt.

Daneben gibt es noch einen dritten Softwarerenderer, der im Objekt **t\_RenderGen-File** implementiert ist. Dieser ist von Benutzer nicht auszuwählen und wird von Cgi3D verwendet, wenn der Benutzer die Ausgabe der Szene in eine Datei mittels **renderFile** vornimmt. Dieser Renderer verwendet single line Z-Buffer für die Erzeugung der Datei.

## Anhang B

# Klassenreferenz

### enum t\_illumination

gen/rendctrl.hh

---

**t\_illumination** definiert die möglichen Beleuchtungsmodelle, welche im MRT verwendet werden können. Cgi3D unterstützt nur einen Teil dieser Beleuchtungsmodelle, da ein Teil dieser Modelle für die interaktive Darstellung der Szene nicht sinnvoll erscheint (so z.B. Monochrome oder DepthCuing). Im folgenden werden deshalb nur diejenigen Modelle beschrieben, welche von Cgi3D unterstützt werden.

- |                            |   |
|----------------------------|---|
| <b>DefaultIllumination</b> | Hierbei wird das Beleuchtungsmodell gewählt, welches durch die Environmentvariable CGI3D.IlluminationModel festgelegt ist. Eine genaue Beschreibung aller Environmentvariablen findet sich in Anhang <a href="#">C</a> .  |
| <b>Ambient</b>             | Reine ambiente Beleuchtung, d.h die Lichtquellen werden nicht ausgewertet.  |
| <b>Diffuse</b>             | Ambiente und diffuse Beleuchtung. Hierbei werden sowohl das ambiente Licht, als auch die Lichtquellen zur Beleuchtung der Szene verwendet.  |
| <b>PhongIllumination</b>   | Ambiente und diffuse Beleuchtung, wobei die Phong Beleuchtungsgleichung verwendet wird.   |
| <b>Radiosity</b>           | Hierbei werden weder die ambiente Beleuchtung, noch die Lichtquellen zur Beleuchtung verwendet. Vielmehr ist in jedem Polygon des anzuzeigenden Objektes sein Farbwert gespeichert, welcher ohne Veränderung durch Cgi3D zur Färbung des Polygons verwendet wird. Dieser Ausgabemodus ist, wie sein Name schon sagt, für Szenen gedacht, welche mit dem Radiosity-Verfahren berechnet wurden. |

## enum t\_shading

gen/rendctrl.hh

**t\_shading** definiert die möglichen Schattierungsarten, welche im MRT verwendet werden können. Cgi3D unterstützt nur einen Teil dieser Schattierungsarten. Im folgenden sind deshalb nur diejenigen Schattierungsarten beschrieben, die in Cgi3D ausgewählt werden können.

- DefaultShading** Hierbei wird die Schattierungsart ausgewählt, welche durch die Environmentvariable **CGI3D.ShadingModel** festgelegt ist. Eine genaue Beschreibung aller Environmentvariablen befindet sich in Anhang C.
- Wireframe** Die Szene wird als Drahtgittermodell dargestellt.
- Flat** Die Szene wird mittels Flat-Shading dargestellt. Dies bedeutet, das jedes Polygon eine konstante Farbe erhält.
- Gouraud** Die Szene wird mittels Gouraud-Shading dargestellt. Hierbei wird die Farbe im Inneren des Polygons durch Interpolation über die Farben seiner Eckpunkte berechnet.
- PhongShading** Die Szene wird mittels Phong-Shading dargestellt. Hierbei wird die Normale im Flächeninneren des Polygons durch Interpolation über die Normalen seiner Ecken bestimmt. Die Farbe wird mit Hilfe dieser interpolierten Normalen mittels der Beleuchtungsgleichung berechnet.

## enum t\_GraphicHardware

t\_rensn.hh

**t\_GraphicHardware** definiert alle Renderer, die unter Cgi3D zur Verfügung stehen. Je nach verwendeter Computerhardware sind jedoch nicht immer alle Renderer verfügbar.

- OpenGL** der standardmäßige OpenGL-Renderer
- OpenGL\_TSL** der OpenGL-Renderer, der Triangle Strip Lists verwendet
- XGL** der XGL-Renderer
- Direct3D** der standardmäßige Direct3D-Renderer
- Direct3D2** der Direct3D-Renderer, der Array-Optimierung verwendet
- Software** der Softwarerenderer, der full screen Z-Buffer verwendet
- SoftwareSZB** der Softwarerenderer, der single line Z-Buffer verwendet

## enum t\_TextureMethod

t\_renschn.hh

---

**t\_TextureMethod** definiert verschiedene Methoden, mittels derer eine Textur von der Hardware auf das Objekt ausgegeben werden kann.

<b>DefaultTexture</b>	Es wird die Texturmethode gewählt, welche durch die Environmentvariable <b>CGI3D.filterMethod</b> festgelegt ist. Eine genaue Beschreibung aller Environmentvariablen befindet sich in Anhang C.
<b>NoTexture</b>	Es wird keine Textur angezeigt.
<b>Nearest</b>	Es wird das Pixel der Textur verwendet, welches dem berechneten am nächsten liegt.
<b>Linear</b>	Das Texturpixel wird durch Interpolation über diejenigen Pixel der Textur bestimmt, die dem berechneten am nächsten liegen.
<b>MipmapNearest</b>	Die Textur wird mittels Mipmaps <sup>1</sup> im Speicher gehalten. Für das Texturpixel wird zunächst die geeignetste Textur (im Bezug auf dessen Auflösung) ermittelt und innerhalb dieser Textur wie bei der Methode <b>Nearest</b> das passende Pixel ermittelt.
<b>MipmapLinear</b>	Die Textur wird mittels Mipmaps im Speicher gehalten. Das Texturpixel wird durch bilineare Interpolation aus den naheliegenden Texturen ermittelt.

## enum t\_PickModes

t\_renschn.hh

---

**t\_PickModes** definiert die Rückgabewerte, welche die Funktion **doHWpick** der Hardware-renderer zurückgeben kann. Abhängig von diesem Wert werden die übrigen Werte per Software ermittelt.

<b>PickNotSupported</b>	Die Hardware unterstützt kein Picking oder die Funktionalität ist zur Zeit nicht implementiert. Sowohl Objekt, als auch Fläche müssen per Software ermittelt werden.
<b>NoObjectHit</b>	Die Hardware unterstützt das Picking, unter dem angegebenen Pixel befindet sich jedoch kein Objekt.
<b>ObjectFound</b>	Die Hardware konnte nur das getroffene Objekt ermitteln, die Fläche muß per Software ermittelt werden.
<b>FaceFound</b>	Die Hardware konnte sowohl das Objekt, als auch die getroffene Fläche ermitteln.

---

<sup>1</sup>Mipmaps sind vorberechnete Texturen in unterschiedlichen Auflösungen. Hierdurch kann die Skalierung der Textur für die Ausgabe einfacher erfolgen und die Ergebnistextur ist bei geeignet gewählten Mipmaps besser als bei Verwendung einer einzelnen Textur.

## Rückgabewerte von `getHardwareStatus`

t\_rensch.hh

Dies sind die Werte, welche die Funktion `getHardwareStatus` von `t_Cgi3D` als Ergebnis kodiert in einem `long` zurückgibt. Die Beschreibung erfolgt bitkodiert und nicht mittels `enum`, um in einem Rückgabewert mehrere Möglichkeiten der Hardware zu kodieren.

<b>HW_SHADING_WIREFRAME</b>	0x00000001	Die Hardware unterstützt die Ausgabe als Wireframe, also als Drahtgittermodell.
<b>HW_SHADING_FLAT</b>	0x00000002	Die Hardware unterstützt die Ausgabe mittels Flatshading.
<b>HW_SHADING_GOURAUD</b>	0x00000004	Die Hardware unterstützt die Ausgabe mittels Gouraudshading.
<b>HW_SHADING_PHONG</b>	0x00000008	Die Hardware unterstützt die Ausgabe mittels Phongshading.
<b>HW_ILLUMINATION_AMBIENT</b>	0x00000010	Die Hardware unterstützt die Ausgabe mittels reiner ambienter Beleuchtung.
<b>HW_ILLUMINATION_DIFFUSE</b>	0x00000020	Die Hardware unterstützt die Ausgabe mittels diffuser Beleuchtung.
<b>HW_ILLUMINATION_PHONG</b>	0x00000040	Die Hardware unterstützt die Ausgabe mittels Phong Beleuchtungsgleichung.
<b>HW_ILLUMINATION_RADIOSTY</b>	0x00000080	Die Hardware unterstützt die Ausgabe für Radiosity. Dies bedeutet, daß keine Beleuchtung verwendet wird. Vielmehr ist bei jeder auszugehenden Fläche ihre genaue Farbe gespeichert.

**t\_Cgi3D** ist die Klasse, welche der Anwender zur Darstellung von dreidimensionalen Szenen verwendet. Ein **t\_Cgi3D**-Objekt kann an ein bestehendes **t\_Cgi** Fenster gebunden werden. In diesem Fall erfolgt die Ausgabe in das von **Cgi** geöffnete Fenster. Alternativ kann **Cgi3D** das Fenster auch selber öffnen und verwalten.

#### Öffentliche Elementfunktionen:

<b>Konstruktor</b>	<pre>t_Cgi3D(unsigned sizex,          unsigned sizey,          const char* win_name = CGI3D-Window,          int xpos = t_Cgi::DEFAULT_POSITION,          int ypos = t_Cgi::DEFAULT_POSITION,          int visual = t_Drawable::DEFAULT_VISUAL,          int depth = t_Cgi::DEFAULT_DEPTH,          unsigned border = 1,          const char* dispName = NULL)</pre> <p>Ein neues <b>t_Cgi</b>-Fenster wird mit den übergebenen Parametern geöffnet. Die Dokumentation von <b>t_Cgi</b><a href="#">[Fel93]</a> enthält eine genaue Beschreibung aller Parameter</p>
<b>Konstruktor</b>	<pre>t_Cgi3D(t_Cgi* cgi)</pre> <p>Die Ausgabe von <b>t_Cgi3D</b> erfolgt in das Fenster des übergebenen <b>t_Cgi</b>-Objektes.</p>
<b>initialize</b>	<pre>void initialize(t_Cgi* cgi)</pre> <p>Die Ausgabe von <b>t_Cgi3D</b> erfolgt in das Fenster des übergebenen <b>t_Cgi</b>-Objektes. Ein eventuell von <b>t_Cgi3D</b> erzeugtes Fenster wird dabei geschlossen.</p>
<b>illumination</b>	<pre>void illumination(t_illumination illumi)</pre> <p><b>illumi</b> bestimmt das Beleuchtungsmodell, welches <b>Cgi3D</b> für die Ausgabe der Szene verwendet. Zu beachten ist, daß nicht jeder Renderer jeden Beleuchtungsmodus unterstützt. Die für den gewählten Renderer verfügbaren Beleuchtungsmodelle können mittels <b>getHardwareStatus</b> abgefragt werden.</p>
<b>illumination</b>	<pre>t_illumination illumination() const</pre> <p>Abfrage des aktuell eingestellten Beleuchtungsmodells.</p>
<b>illuminationId</b>	<pre>const char* illuminationId() const</pre> <p>Abfrage des aktuell eingestellten Beleuchtungsmodells als String.</p>
<b>shading</b>	<pre>bool shading(t_shading shading)</pre> <p>Setzt das Shading für die gesamte Szene oder die nächsten auszugebenden Objekte. Der Rückgabewert ist <b>true</b>, falls der Renderer das gewünschte Shading unterstützt, <b>false</b>, falls er dies nicht tut. Bei Rückgabewert <b>false</b> wird das Standardshading des Renderers eingeschaltet (meist Gouraud).</p>
<b>shading</b>	<pre>shading() const</pre> <p>Abfrage des aktuell gesetzten Shadingmodus.</p>

<b>shadingId</b>	const char* shadingId() const Abfrage des aktuell gesetzten Shadingmodus als String.
<b>textureMethod</b>	textureMethod(t_TextureMethod method) Setzt die Methode, mit der die Textur eines Objektes dargestellt wird.
<b>textureMethod</b>	t_textureMethod textureMethod() const; Abfrage der aktuell gesetzten Texturmethode
<b>set3DtextureQuality</b>	void set3DtextureQuality(int mapSize int minSamples)  Dies setzt die Qualität der für die Simulation von 3D-Texturen erzeugten 2D-Textur. <b>mapSize</b> bestimmt dabei die Auflösung der 2D-Textur, <b>minSamples</b> bestimmt die Qualität des Samplings der 3D-Textur. Eine exakte Beschreibung der Parameter ist in Kapitel 6.2 zu finden.
<b>doubleBuffer</b>	void doubleBuffer(bool value) Schaltet die Ausgabe zwischen singlebuffered ( <b>value = false</b> ) und doublebuffered ( <b>value = true</b> ) um.
<b>doubleBuffer</b>	bool doubleBuffer() const Der Rückgabewert ist <b>true</b> , falls Cgi3D doublebuffering für die Ausgabe verwendet.
<b>useGraphicHardware</b>	bool useGraphicHardware(t_GraphicHardware value, bool dblBuffer)  Wählt den Hardwarerenderer, der von Cgi3D verwendet werden soll. Gleichzeitig wird zwischen singlebuffered und doublebuffered Anzeige gewählt. Der Rückgabewert ist <b>true</b> , falls der gewünschte Renderer auf der aktuellen Hardware vorhanden ist. Ist der Rückgabewert <b>false</b> , so ist der gewünschte Renderer nicht verfügbar und der Softwarerenderer wurde eingeschaltet.
<b>useGraphicHardware</b>	t_GraphicHardware useGraphicHardware() const Abfrage der aktuell ausgewählten Grafikhardware bzw. des aktuell eingestellten Renderers.
<b>getHardwareStatus</b>	long getHardwareStatus() const Abfrage der Fähigkeiten, welche der aktuell ausgewählte Renderer unterstützt. Eine genaue Beschreibung der Rückgabewerte befindet sich am Anfang der Beschreibung der Klasse t_Cgi3D.
<b>rebuildAll</b>	void rebuildAll() Nach Aufruf dieser Funktion übergibt Cgi3D beim nächsten Aufruf von <b>render</b> die Triangulierung aller Objekte erneut an die Hardware, auch wenn sich die Szene nicht geändert hat und sich die Daten noch im Cache der Hardware befinden. Diese Funktion ist hauptsächlich zur Performancemessung gedacht. Hiermit läßt sich die Geschwindigkeit zwischen erneuter Datenübergabe und Verwendung des Cache der Hardware ermitteln.

<b>camera</b>	void camera(const t_CameraPtr& c) Setzt die Kamera für die Darstellung der Szene.
<b>camera</b>	const t_CameraPtr& camera() const Abfrage der aktuell gesetzten Kamera.
<b>lights</b>	void lights(const t_LightsPtr& l) Setzt die Beleuchtung für die Szene.
<b>lights</b>	const t_LightsPtr& lights() const Abfrage der aktuell gesetzten Beleuchtung.
<b>xResolution</b>	unsigned int xResolution() const Abfrage der aktuellen horizontalen Größe des Cgi3D-Fensters in Pixel.
<b>yResolution</b>	unsigned int yResolution() const Abfrage der aktuellen vertikalen Größe des Cgi3D-Fensters in Pixel.
<b>backgroundColor</b>	void backgroundColor(const t_Color& color) Setzt die Hintergrundfarbe für die Szene.
<b>backgroundColor</b>	const t_Color& backgroundColor() const Abfrage der aktuell gesetzten Hintergrundfarbe.
<b>scene</b>	void scene(const t_ScenePtr& scene) Setzt der Szene, welche mittels <b>render</b> ausgegeben werden kann.
<b>scene</b>	const t_ScenePtr& scene() const Abfrage der aktuellen Szene.
<b>render</b>	void render(bool swapBuf) Zeigt die mittels <b>scene</b> übergebene Szene an. Ist <b>swapBuf = false</b> so wird bei Verwendung einer doublebuffered Anzeige der Hintergrundbuffer noch nicht angezeigt.
<b>renderFile</b>	bool renderFile(const char* filename, const PnmFileType format, const unsigned int xResolution, const unsigned int yResolution)  Speichert die aktuelle Szene in eine PNM-Datei ab. Da bei der aktuellen Implementierung nur der Softwarerenderer verwendet wird, kann die Auflösung des Bildes unabhängig von der aktuellen Auflösung des Cgi3D-Fensters gewählt werden.
<b>swapBuffers</b>	void swapBuffers() Hiermit wird der Hintergrundbuffer bei einer doublebuffered Anzeige in den Vordergrundbuffer kopiert.
<b>interactor</b>	void interactor() Zeigt die mittels <b>scene</b> übergebene Szene an und öffnet einen 3D-Interaktor zur Modifizierung der Kameraposition.

<b>animator</b>	<pre>void animator(t_CameraPtr* cameraArray,              int keyFrames,              int* inbetweens,              t_CameraFly::InterplTyp interpl,              t_CameraFly::PlayTyp playTyp)</pre> <p>Diese Funktion ermöglicht einen animierten Flug durch die Szene. <b>cameraArray</b> ist ein Array von Kamerapositionen, die angesteuert werden sollen. <b>keyFrames</b> ist die Anzahl der Kamerapositionen. <b>inbetweens</b> gibt an, wieviele Frames jeweils zwischen den einzelnen Kamerapositionen dargestellt werden. <b>interpl</b> beschreibt die Art der Interpolation (Linear<sup>2</sup> oder Quaternion<sup>3</sup>) zwischen den einzelnen Kamerapositionen und <b>playType</b> die Art der Wiedergabe (Loop oder Ping-Pong). Die Animation wird so lange abgespielt, bis eine Taste gedrückt wird.</p>
<b>pick</b>	<pre>bool pick(const t_2DintVector&amp; xy,           t_SurfaceObjectPtr* obj,           t_FaceNormal** face)</pre> <p>Ermittelt das Objekt und die Fläche, welche bei der aktuellen Anzeige am Ort des Pixels <b>xy</b> angezeigt wird. Ist <b>face = NULL</b>, so wird nur das entsprechende Objekt ermittelt.</p>
<b>lineColor</b>	<pre>void lineColor(const t_Colr&amp; color)</pre> <p>Setzt die Farbe für das Zeichnen von Linien mittels <b>line</b>.</p>
<b>line</b>	<pre>void line(const t_2DVector&amp; a,           const t_2DVector&amp; b)</pre> <p>Zeichnet eine Linie über die aktuelle Bitmap.</p>
<b>line</b>	<pre>void line(const t_3DVector&amp; a,           const t_3DVector&amp; b)</pre> <p>Zeichnet eine Linie über die aktuelle Bitmap. Die Punkte werden in den Raum projiziert, die Linie wird jedoch auf die Frontplane der Szene gezeichnet.</p>
<b>cgi</b>	<pre>t_Cgi* cgi()</pre> <p>Abfrage des aktuell verwendeten Cgi-Objektes.</p>

---

<sup>2</sup>Bei linearer Interpolation bewegt sich die Kamera zwischen zwei Keyframes auf einer Geraden, so daß an den Keyframes Unstetigkeiten in der Kamerabahn entstehen können

<sup>3</sup>Mittels Quaternioneninterpolation wird eine Kurve durch alle Keyframes gelegt, so daß eine stetige Bahn für die Kamera entsteht

### **Von Cgi abgeleitete öffentliche Elementfunktionen:**

Die folgenden Funktionen sind direkt von Cgi abgeleitet. Eine genaue Beschreibung der Parameter ist in der Dokumentation von Cgi zu finden.

<b>initLogically</b>	<code>virtual void initLogically(Cin_class inClass, CGI_IX inIndex)</code>  Vorbereitung von Cgi3D für die Abfrage der Tastatur oder Maus
<b>sampleLocator</b>	<code>virtual int sampleLocator (Cvalidity&amp; valid, CGI_IX&amp; trigger, t_2DVector&amp; point)</code>  Abfrage der aktuellen Mausposition.
<b>requestLocator</b>	<code>virtual int requestLocator(Cvalidity&amp; valid, CGI_IX&amp; trigger, t_2DVector&amp; point, t_Real timeout)</code>  Abfrage der aktuellen Mausposition. Dabei wird auf den Druck einer Maustaste gewartet.
<b>sampleLocator</b>	<code>virtual int sampleLocator (Cvalidity&amp; valid, CGI_IX&amp; trigger, t_3DVector&amp; direction)</code>  Abfrage der Blickrichtung in die Szene an der aktuellen Mausposition. Dies ist der Vektor, der vom gewählten Punkt auf der Projektionsebene zum Zentrum der Szene zeigt.
<b>requestLocator</b>	<code>virtual int requestLocator(Cvalidity&amp; valid, CGI_IX&amp; trigger, t_3DVector&amp; direction, t_Real timeout)</code>  Abfrage der Blickrichtung in die Szene an der aktuellen Mausposition. Dies ist der Vektor, der vom gewählten Punkt auf der Projektionsebene zum Zentrum der Szene zeigt. Bei der Abfrage wird auf den Druck einer Maustaste gewartet.
<b>sampleKeyboard</b>	<code>virtual int sampleKeyboard(Cvalidity&amp; valid, char&amp; value)</code>  Abfrage der Tastatur.
<b>requestKeyboard</b>	<code>virtual int requestKeyboard(Cvalidity&amp; valid, char&amp; value, t_Real timeout)</code>  Abfrage der Tastatur. Dabei wird auf einen Tastendruck gewartet.
<b>cgiExecuteDeferredActions</b>	<code>void cgiExecuteDeferredActions() const</code>  Ausführung aller Kommandos durch Cgi, welche dieses intern zwischengespeichert hat.

### Geschützte Elementfunktionen:

**construct** void construct(t\_Cgi\* cgi)

Teil des Konstruktors, der von beiden Konstruktoren verwendet wird

### Geschützte Komponenten

**v\_illumination** t\_illumination v\_illumination  
Aktuelles Beleuchtungsmodell.

**v\_shading** t\_shading v\_shading  
Aktuelles Schattierungsmodell.

**v\_cgi** t\_Cgi\* v\_cgi  
Zeiger auf das mit Cgi3D verbundene Cgi-Object.

**xRes** unsigned int xRes  
Horizontale Auflösung des Cgi3D-Fensters in Pixel.

**yRes** unsigned int yRes  
Vertikale Auflösung des Cgi3D-Fensters in Pixel.

**v\_backgroundColor** t\_Color24Bit v\_backgroundColor  
Aktuelle Hintergrundfarbe.

**v\_camera** t\_CameraPtr v\_camera  
Aktuelle Kamera.

**v\_lights** t\_LightsPtr v\_lights  
Aktuelle Beleuchtung.

**v\_ownCgiWindow** bool v\_ownCgiWindow  
Ist **true**, wenn Cgi3D ein eigenes Cgi-Fenster geöffnet hat.

**v\_renderScene** t\_RenderScene\* v\_renderScene  
Zeiger auf den aktuellen Hardwarerenderer.

**v\_scene** t\_ScenePtr v\_scene  
Aktuelle Szenenbeschreibung.

`t_RenderScene` ist die Basisklasse für alle Hardwarerenderer. Die Hardwarerenderer werden nur innerhalb von `Cgi3D` verwendet. Eine Verwendung durch den Benutzer von `Cgi3D` ist nicht vorgesehen.

Die meisten Funktionen von `t_RenderScene` sind rein virtuelle Funktionen, d.h. sie sind in `t_RenderScene` nicht implementiert. Vielmehr müssen sie in der abgeleiteten Klasse überladen werden. Einige Funktionen, welche hardwareunabhängig sind und von allen Renderern verwendet werden können, sind bereits in der Basisklasse implementiert.

Die Funktionsbeschreibungen enthalten keine Beschreibung, welche Wirkung die Funktion hat. Vielmehr wird hier erläutert, welche Funktionalität in einer abgeleiteten Klasse implementiert werden muß. Zusammen mit Kapitel 13 ergibt sich somit eine genaue Beschreibung, welche Funktionalität in welcher Funktion für einen Hardwarerenderer zu implementieren ist.

#### Öffentliche Elementfunktionen:

Die öffentlichen Funktionen sind diejenigen, welche von `Cgi3D` aufgerufen werden.

<b>Destruktor</b>	<code>flt_RenderScene</code> Im Destruktor muß die Hardware wieder freigegeben und eine eventuelle Bindung an das Fenster von <code>Cgi3D</code> aufgehoben werden.
<b>setBackgroundcolor</b>	<code>void setBackgroundcolor(const t_Color&amp; col)</code> Setzt die Hintergrundfarbe.
<b>setPickingMode</b>	<code>void setPickingMode(bool pickingMode)</code> Ist <b>pickingMode = true</b> , so dient der nächste Aufruf von <b>render</b> dazu, die Szene für ein Picking darzustellen. Dies kann bei einigen Hardwareplattformen nötig sein.
<b>setCamera</b>	<code>void setCamera(const t_CameraPtr&amp; camera)</code> Dies übergibt die Kamera an den Renderer. Hierbei ist die Kamera noch nicht in der Hardware zu setzen. Vielmehr müssen sie zwischengespeichert werden. Ein Aufruf von <b>setCamera</b> zeigt dem Renderer an, daß sich die Kamera verändert hat.
<b>setLights</b>	<code>void setLights(const t_LightsPtr&amp; lights)</code> Dies übergibt die Lichter an den Renderer. Hierbei sind die Lichte noch nicht in der Hardware zu setzen. Vielmehr müssen sie zwischengespeichert werden. Ein Aufruf von <b>setLights</b> zeigt dem Renderer an, daß sich die Lichter der Szene geändert haben.
<b>setShading</b>	<code>bool setShading(t_shading shad)</code> Setzt das Schattierungsmodell für die Ausgabe der nächsten Objekte. Das Schattierungsmodell kann sich von Objekt zu Objekt ändern. Der Rückgabewert ist <b>false</b> , falls die Hardware das gewünschte Schattierungsmodell nicht unterstützt. Dann ist ein Defaultmodell für den Renderer einzustellen.
<b>setIllumination</b>	<code>void setIllumination(t_illumination illumi)</code> Setzt das Beleuchtungsmodell für die gesamte Szene.

<b>initHardware</b>	void initHardware(bool pickingMode) Initialisiert die Hardware. In dieser Funktion ist insbesondere eine Anpassung der Hardware an die aktuelle Fenstergröße von Cgi und das Löschen des Bildspeichers nötig. Außerdem ist darauf zu achten, daß das korrekte Hardwarefenster eingeschaltet wird, falls mehrere Fenster von Cgi3D geöffnet sind. Dies ist der Grund, weshalb <b>initHardware</b> vor dem Aufruf von <b>beginSzene</b> erfolgt. Ist <b>pickingMode = true</b> , so dient die nächste Ausgabe der Szene der Vorbereitung für das Picking.
<b>useGraphicHardware</b>	bool useGraphicHardware() Der Rückgabewert ist <b>true</b> , falls die Hardware korrekt initialisiert werden konnte und eine Ausgabe der Szene mit dieser Hardware möglich ist. <b>false</b> könnte hier jedoch auch bedeuten, daß die Hardware zwar im allgemeinen verfügbar ist, eine spezielle Erweiterung, welche der Renderer benötigt, jedoch nicht installiert ist. Ist der Rückgabewert <b>false</b> , so verwendet Cgi3D diesen Renderer nicht.
<b>getStatus</b>	long getStatus() Der Rückgabewert enthält bitkodiert die Fähigkeiten, welche die Hardware unterstützt. Der Wert ist eine or-Verknüpfung der Werte aus <b>HW_SHADING_*</b> und <b>HW_ILLUMINATION_*</b> .
<b>textureMethod</b>	void textureMethod(t_TextureMethod method) Setzt die Art und Weise, mit der Texturen auf ein Objekt gemappt werden. Ist der Wert <b>NoTexture</b> , so sind die Texturen für die nächsten Objekte zu deaktivieren.
<b>textureMethod</b>	t_TextureMethod textureMethod() Der Rückgabewert ist die aktuell ausgewählte Texturmethode, welche der gewählten möglichst nahe kommen sollte, falls die Hardware den vom Benutzer gewünschten Modus nicht unterstützt.
<b>set3DtextureQuality</b>	void set3DtextureQuality(int mapSize int minSamples) Dies setzt die Qualität der für die Simulation von 3D-Texturen erzeugten 2D-Textur. <b>mapSize</b> bestimmt dabei die Auflösung der 2D-Textur, <b>minSamples</b> bestimmt die Qualität des Samplings der 3D-Textur. Eine exakte Beschreibung der Parameter ist in Kapitel 6.2 zu finden.
<b>doubleBuffer</b>	doubleBuffer() Falls die Hardware doublebuffering unterstützt und dieses aktiviert ist, so ist der Rückgabewert <b>true</b> , ansonsten <b>false</b> .
<b>rebuildAll</b>	rebuildAll() Wenn diese Funktion durch Cgi3D aufgerufen wird, dann muß der Renderer danach alle Objekte erneut an die Hardware übergeben. Eventuell in Displaylisten <sup>4</sup> bereits gespeicherte

<sup>4</sup>Als Displayliste bezeichnet man unter OpenGL den internen Cache der Grafikhardware, in dem diese die Koordinaten und Farben der Ausgabe zwischenspeichert.

Daten sind zu löschen. Dies dient hauptsächlich dem Perfor-  
mancetest, um zu überprüfen, wie schnell die Übergabe der  
Daten an die Hardware geschieht.

#### **setMaterial**

```
void setMaterial(const t_Color& amb,  
                const t_Color& dif,  
                const t_Color& spec,  
                const t_Real& exp,  
                const t_Real& transp,  
                const t_Color& emis)
```

Setzt die Materialeigenschaften für das aktuelle Objekt. Falls  
es genügt, das Material einmal pro Objekt in der Hardware zu  
setzen, so kann dies hier geschehen. Ansonsten müssen die  
Daten gespeichert werden.

#### **setTexture**

```
bool setTexture(t_2DTexture* texture)
```

Setzt die Textur für das aktuelle Objekt. Falls **texture =  
NULL** ist, dann besitzt das Objekt keine Textur. Ansonsten  
enthält **texture** eine 2D-Textur, welche an die Hardware über-  
geben werden muß. Der Rückgabewert ist **false**, falls die Tex-  
tur nicht an die Hardware übergeben werden konnte. Dann  
wird das Objekt ohne Textur ausgegeben.

#### **setMatrix**

```
void setMatrix(const t_4x3Matrix* matrix)
```

Setzt die lokale Transformationsmatrix für das aktuelle Ob-  
jekt.

#### **beginScene**

```
void beginScene()
```

Diese Funktion wird aufgerufen, wenn eine neue Ausgabe der  
Szene beginnt. Beim Aufruf dieser Funktion sind Illuminati-  
on, Lichter und Kamera bereits an den Renderer übergeben,  
aber noch *nicht* mittels **initLights** oder **initCamera** gesetzt  
worden. Hier können Einstellungen, die bei jedem Aufbau der  
Szene gemacht werden müssen, vorgenommen werden.

#### **endScene**

```
void endScene(bool swapBuf)
```

Alle Objekte der Szene sind an die Hardware übergeben wor-  
den. In **endScene** muß nun sichergestellt werden, daß alle  
Objekte von der Hardware bearbeitet und ausgegeben wurden.  
Ist **swapBuf = true** und die Hardware verwendet doublebuf-  
fering, so muß der Hintergrundbuffer in die Anzeige kopiert  
werden.

#### **setScene**

```
void setScene(const t_ScenePtr& scene)
```

Setzt die Szene, welche ausgegeben werden soll. Durch den  
Aufruf von **setScene** wird angezeigt, daß sich an der Szene  
etwas geändert haben kann und daß es deshalb bei der näch-  
sten Ausgabe nicht genügt, nur alle Displaylisten auszugeben.  
Vielmehr muß die Szene komplett durchlaufen und überprüft  
werden.

#### **renderScene**

```
void renderScene(bool swapBuf)
```

Zeigt die aktuelle Szene an. Ist **swapBuf = true** und die  
Hardware verwendet doublebuffering, so muß der Hinter-  
grundbuffer in die Anzeige kopiert werden.

<b>pick</b>	<pre>bool pick(const t_2DintVector&amp; xy,           t_SurfaceObjectPtr* object,           t_FaceNormal** face)</pre> <p>Ermittelt das Objekt und die Fläche, welche bei der aktuellen Anzeige am Ort des Pixels <b>xy</b> angezeigt wird. Hierzu muß die Szene unter Umständen erneut angezeigt werden. Ist <b>face = NULL</b>, so wird nur das entsprechende Objekt ermittelt.</p>
<b>displayObject</b>	<pre>bool displayObject(const t_ObjectPtr&amp; object)</pre> <p>Anzeige eines Objektes ohne lokale Transformationsmatrix. Diese Funktion ist identisch zu <b>displayObjectTM</b>, jedoch wird keine Transformationsmatrix gesetzt. Der Rückgabewert ist <b>false</b>, wenn sich das Objekt bereits in einer Displayliste befand und diese für seine Anzeige verwendet wurde.</p>
<b>displayObjectTM</b>	<pre>bool displayObjectTM(const t_ObjectPtr&amp; object)</pre> <p>Anzeige eines Objektes mit lokaler Transformationsmatrix. Die lokale Transformationsmatrix ist mittels <b>setMatrix</b> an die Hardware zu übergeben. Der Rückgabewert ist <b>false</b>, wenn sich das Objekt bereits in einer Displayliste befand und diese für seine Anzeige verwendet wurde.</p>
<b>swapBuffers</b>	<pre>void swapBuffers()</pre> <p>Wenn die Hardware doublebuffering verwendet, so kopiert diese Funktion den Hintergrundbuffer in die Anzeige.</p>
<b>hardwareID</b>	<pre>t_GraphicHardware hardwareID()</pre> <p>Abfrage der verwendeten Grafikhardware. Hierdurch identifiziert sich die eingestellte Grafikhardware.</p>

#### Geschützte Elementfunktionen:

<b>Konstruktor</b>	<pre>t_RenderScene()</pre> <p>Der Defaultkonstruktor ist als geschützte Elementfunktion definiert, damit der Anwender keine Instanz der Klasse <b>t_RenderScene</b> anlegen kann.</p>
<b>initCamera</b>	<pre>void initCamera()</pre> <p>Dies setzt die mittels <b>setCamera</b> übergebene Kamera in der Hardware. Falls die Hardware dies unterstützt und <b>v_cameraChanged = false</b> ist, so hat sich die Kamera seit dem letzten Aufruf von <b>initCamera</b> nicht verändert und sie muß nicht erneut an die Hardware übergeben werden.</p>
<b>initLights</b>	<pre>void initLights()</pre> <p>Dies setzt die mittels <b>setLights</b> übergebenen Lichter in der Hardware. Falls die Hardware dies unterstützt und <b>v_lightsChanged = false</b> ist, so haben sich die Lichter seit dem letzten Aufruf von <b>initLights</b> nicht verändert und sie müssen nicht erneut an die Hardware übergeben werden. Da vor dem Aufruf von <b>initLights</b> bereits die Hardware mittels <b>initHardware</b> initialisiert und dem Renderer der Beginn einer neuen Szene mitgeteilt wurde, darf <b>setLights</b> auch Objekte mittels <b>displayObject</b> ausgeben. Dies könnte zur Simulation von area Lights verwendet werden.</p>

<b>beginObject</b>	<pre>bool beginObject(const t_ObjectPtr&amp; object,                  const t_ApproximationPtr&amp; aobject,                  bool rebuild)</pre> <p>Mit diesem Aufruf wird der Beginn eines Objektes angezeigt. Hier müssen alle Einstellungen in der Hardware vorgenommen werden, welche für das gesamte Object gelten. Hier muß auch ein Test erfolgen, ob das Objekt bereits in einer Displayliste enthalten ist. Ist dies der Fall und sind sowohl <b>rebuild = false</b>, als auch <b>v_rebuildAll = false</b>, so muß die Displayliste für dieses Objekt aktiviert werden und <b>beginObject</b> wird mit dem Rückgabewert <b>false</b> verlassen.</p>
<b>endObject</b>	<pre>void endObject()</pre> <p>Bei Aufruf dieser Funktion sind alle Polygone des Objektes an die Hardware übergeben. Das Objekt kann somit abgeschlossen werden (z.B. kann die Displayliste beendet werden).</p>
<b>addFace</b>	<pre>void addFace(t_FaceNormal* face,              const t_ObjectPtr&amp; object)</pre> <p>Hierin wird ein Polygon an die Hardware übergeben. Dabei muß das Polygon Eckpunkt für Eckpunkt durchlaufen werden und die passenden Daten an die Hardware übergeben werden. Der Pointer auf das Objekt, zu welchem das Polygon gehört, wird für das Texturemapping benötigt. Die Daten können entweder direkt an die Hardware übergeben werden, oder aber sie werden in einem Feld zwischengespeichert. Ist das Feld gefüllt, so wird es mittels <b>renderArrays</b> an die Hardware übergeben. Das Feld ist als lokale Komponente im abgeleiteten Objekt zu definieren.</p>
<b>renderArrays</b>	<pre>void renderArrays(unsigned degree)</pre> <p>Falls der Renderer Arrays für die Übergabe der Daten an die Hardware verwendet, so müssen hiermit diese Arrays an die Hardware übergeben werden. <b>degree</b> enthält die Anzahl der Ecken eines Polygons, das in den Arrays gespeichert ist. Falls dieser Parameter von der Hardware benötigt wird, so dürfen die Arrays nur Polygone mit <b>degree</b> Ecken enthalten. Anderenfalls ist die Eckenzahl mit in den Arrays gespeichert und <b>degree</b> wird nicht verwendet. Diese Funktion wird auch am Ende eines Objektes aufgerufen, so daß die Arrays nicht notwendigerweise komplett gefüllt sein müssen.</p> <p>Die Arrays müssen als private Komponenten im Renderer gehalten werden. Eine Übergabe der Arrays als Parameter ist nicht sinnvoll, da dies nur einen zusätzlichen Laufzeitnachteil mit sich bringen würde und sich das Format der Arrays von Renderer zu Renderer unterscheidet. Weil <b>renderArrays</b> eine geschützte Elementfunktion ist, besteht keine Notwendigkeit, diese durch zusätzliche Parameter zu verlangsamen.</p> <p>Falls die Hardware keine Arrays zur Datenübergabe verwendet, so kann diese Funktion leer bleiben.</p>
<b>getTexture</b>	<pre>t_2DTexture* getTexture(const t_ObjectPtr&amp; object)</pre> <p>Ermittelt zum übergebenen <b>object</b> die zugehörige 2D-Textur. Hierbei wird, falls das Objekt eine 3D-Textur besitzt, eine korrespondierende 2D-Textur berechnet.</p>

<b>generateTexture</b>	<p>t_2DTexture* generateTexture(const t_ObjectPtr&amp; object)</p> <p>Berechnet für ein Objekt, welches eine 3D-Textur besitzt, die korrespondierende 2D-Textur.</p>
<b>sampleObject</b>	<p>bool sampleObject(const t_ObjectPtr&amp; object,  const t_3DVector&amp; start,  const t_3DVector&amp; dir,  const t_3DVector&amp; addU,  const t_3DVector&amp; addV,  int samplesU,  int samplesV,  t_Color24Bit* textureArray,  bool* colorSet,  int mapSize)</p> <p>Hilfsfunktion für <b>generateTexture</b>. Das <b>object</b> wird von einer Seite seiner Boundingbox aus abgetastet. Die Samplefläche beginnt bei <b>start</b> und erstreckt sich <b>samplesU</b>×<b>samplesV</b> weit in Richtung <b>addU</b>×<b>addV</b>. Dabei werden die Strahlen in Richtung <b>dir</b> auf das Objekt geschossen. Die ermittelten 2D-Texturwerte werden im Array <b>textureArray</b> der Größe <b>mapSize</b>×<b>mapSize</b> gespeichert. <b>colorSet</b>, das ebenfalls diese Größe besitzt, ist <b>true</b>, wenn der 2D-Texturwert für dieses Pixel bestimmt wurde. Der Rückgabewert ist <b>false</b>, wenn das Objekt keine Funktion <b>mapInvers</b> besitzt.</p>
<b>doHWpick</b>	<p>t_PickModes doHWpick(const t_2DintVector&amp; xy,  t_SurfaceObjectPtr* object,  t_FaceNormal** face)</p> <p>Ermittelt das Objekt und die Fläche, welche bei der aktuellen Anzeige am Ort des Pixels <b>xy</b> angezeigt wird, mittels der Hardware. Entsprechend dem Rückgabewert werden die nicht durch die Hardware zu ermittelnden Daten softwaremäßig ermittelt.</p>

### Geschützte Komponenten

<b>v_textureMethod</b>	<p>t_TextureMethod v_textureMethod</p> <p>Die aktuell gewählte Texturmethode.</p>
<b>v_backgroundColor</b>	<p>t_Color v_backgroundColor</p> <p>Die aktuelle Hintergrundfarbe.</p>
<b>v_sceneInProgress</b>	<p>bool v_sceneInProgress</p> <p>Zeig an, daß gegenwärtig eine Szene an die Hardware übergeben wird. Dies wird in <b>beginScene</b> auf <b>true</b> und in <b>endScene</b> auf <b>false</b> gesetzt.</p>
<b>v_doubleBuffer</b>	<p>bool v_doubleBuffer</p> <p>Zeigt an, ob die Hardware doublebuffering für die Ausgabe verwendet oder nicht.</p>
<b>v_pickingMode</b>	<p>bool v_pickingMode</p> <p>Zeigt an, ob die aktuelle Darstellung der Szene für das Picking erfolgt oder nicht.</p>

<b>v_rebuildAll</b>	bool v_rebuildAll Zeigt an, daß alle Objektdaten erneut an die Hardware übergeben werden müssen, auch dann, wenn sich die Objekte nicht verändert haben.
<b>v_cameraChanged</b>	bool v_cameraChanged Zeigt an, daß sich die Kamera seit dem letzten Aufruf von <b>initCamera</b> geändert hat.
<b>v_lightsChanged</b>	bool v_lightsChanged Zeigt an, daß sich die Lichter seit dem letzten Aufruf von <b>initLights</b> geändert haben.
<b>v_sceneChanged</b>	bool v_sceneChanged Zeigt an, daß sich die Szene seit der letzten Ausgabe (also seit dem letzten Aufruf von <b>endScene</b> ) geändert hat.
<b>v_camera</b>	t_CameraPtr v_camera Die aktuelle Kamera.
<b>v_lights</b>	t_LightsPtr v_lights Die aktuellen Lichter.
<b>v_scene</b>	t_ScenePtr v_scene Die aktuelle Szenenbeschreibung.
<b>v_3DmapSize</b>	int v_3DmapSize Horizontale und vertikale Auflösung der 2D-Textur in Pixeln, die bei der Simulation von 3D-Texturen berechnet wird.
<b>v_3DminSamples</b>	int v_3DminSamples Minimale Anzahl an Samples, mit denen die kürzeste Kante der Boundingbox des Objektes für die Simulation einer 3D-Textur abgetastet wird.
<b>gen3Dtextures</b>	Gen3DTexture gen3Dtextures Enthält für jede bereits angezeigte 3D-Texturapproximation einen Zeiger auf den Shader und einen Zeiger auf die erzeugte 2D-Textur, welche in einem struct zusammengefaßt werden. Hierdurch muß bei einer erneuten Anzeige des Objektes die 2D-Textur nicht erneut berechnet werden.
<b>num3Dtextures</b>	int num3Dtextures Die Anzahl der bereits in <b>gen3Dtextures</b> gespeicherten 3D-Texturapproximationen.

## Anhang C

# Environmentvariable

Die Grundeinstellungen für Cgi3D können durch Setzen von Environmentvariablen beeinflußt werden. Diese Einstellungen werden verwendet, wenn vom Benutzer keine Einstellungen im Programm vorgenommen werden oder wenn der Benutzer die Defaulteinstellungen aufruft. Die Einstellungen werden abhängig vom Betriebssystem in verschiedenen Dateien vorgenommen.

Unter Unix werden die Einstellungen in der Datei `.Xresources` im Hauptverzeichnis des Benutzers (also normalerweise in `~/Xresources`) vorgenommen und lauten dort wie folgt (die *kursiv* angegebenen Werte sind die Defaulteinstellungen):

- CGI3D.IlluminationModel:** Ambient, Diffuse, *Phong*  
Hiermit wird das Beleuchtungsmodell gewählt.
- CGI3D.ShadingModel:** Wireframe, Flat, *Gouraud*, Phong  
Hiermit wird das Schattierungsmodell gewählt.
- CGI3D.frameBufferType:** Single, *Double*  
Hiermit kann zwischen der Ausgabe mittels Singlebuffer und Doublebuffer umgeschaltet werden.
- CGI3D.filterMethod:** NoTexture, *Nearest*, Linear, MipmapNearest, MipmapLinear  
Hiermit wird die Methode gewählt, mittels der die Texturen dargestellt werden.
- CGI3D.graphicHardware:** *OpenGL*, OpenGL\_TSL, Direct3D, Direct3D2, XGL, Software, SoftwareSZB  
Hiermit wird die Grafikhardware, welche für die Ausgabe verwendet werden soll, ausgewählt. Selbstverständlich ist nicht jede Hardware auf jeder Rechnerplattform verfügbar. Wird eine nicht verfügbare Hardware gewählt, so wird **Software** als Renderer von Cgi3D ausgewählt. Eine Ausnahme bildet hierbei der OpenGL\_TSL-Renderer. Falls nur OpenGL 1.0 installiert ist, so wird automatisch diese Version gewählt.
- CGI3D.3DmapSize:** Dies ist die Größe der 2D-Textur in Pixeln, die bei der Simulation von 3D-Texturen berechnet wird. Horizontale und vertikale Auflösung sind identisch. Eine exakte Erläuterung dieses Parameters kann in Kapitel 6.2 nachgelesen werden. Die Größe sollte eine Zweierpotenz sein,

da einige Hardwarerenderer ansonsten die Textur wieder verkleinern müssen. Der Standardwert ist `128`.

**CGI3D.3DminSamples:** Dies ist die minimale Anzahl an Samples, mit denen die kürzeste Kante der Boundingbox des Objektes einer 3D-Textur abgetastet wird. Eine exakte Erläuterung dieses Parameters kann in Kapitel [6.2](#) nachgelesen werden. Der Standardwert ist `120`.

Unter Windows erfolgt die Einstellung der Parameter in einer Inidatei, welche im selben Verzeichnis wie die Anwendung stehen muß. Außerdem hat sie den selben Namen wie das Programm, nur mit der Eindung `.ini`. Für das Programm `mrtpen.exe` lautet die Datei also `mrtpen.ini` und befindet sich im selben Verzeichnis wie `mrtpen.exe`.

Die Einstellungen werden hierbei in Gruppen unterteilt, ähnlich, wie dies in der Datei `win.ini` üblich ist. Die Gruppe für Cgi3D beginnt mit **[CGI3D]**. Die Einstellungen haben den selben Namen wie unter Unix, nur entfällt das Präfix **CGI3D.**, und die Trennung zwischen Variable und Wert erfolgt mittels '=' anstatt mittels ':'. Somit lautet eine Einstellung, die unter Unix mit '**CGI3D.ShadingModel: Gouraud**' beschrieben wird, unter Windows '**ShadingModel = Gouraud**'.

## C.1 Tips zur Einstellung der Variablen

Da Cgi3D auf Cgi aufbaut und das Fenster, welches von Cgi geöffnet wird, zur Ausgabe der Szene verwendet, müssen auch die Environmentvariablen von Cgi korrekt eingestellt ein. Insbesondere sind hierbei folgende Variablen zu beachten:

**CGI.doubleBuffer** Diese Variable sollte entweder auf **False** gesetzt werden, oder nicht definiert werden. Ansonsten kann es bei Verwendung von Penguin<sup>1</sup> unter Umständen vorkommen, daß die Ausgabe von Cgi-3D durch die Ausgabe von Penguin wieder gelöscht wird.

**CGI.backingStore** Diese Variable muß je nach verwendetem Renderer und Betriebssystem unterschiedlich gesetzt werden. Unter Unix oder bei Verwendung des Direct3D-Renderers unter Windows sollte sie auf **WhenMapped** gesetzt oder nicht definiert werden.

Bei Verwendung des Direct3D-Renderers kann es ansonsten zu Problemen mit der Darstellung des Hintergrundes kommen. Dies liegt daran, daß Direct3D nur beim ersten Zeichnen der Szene den gesamten Hintergrund darstellt, bei jeder weiteren Ausgabe der Szene jedoch nur dort den Hintergrund neu zeichnet, wo dieser vorher durch ein Objekt bedeckt war. Dadurch bleiben bei einer anderen Einstellung als **WhenMapped** nicht von Objekten bedeckte Teile des Fensters schwarz, da Penguin nach der ersten Ausgabe von Cgi3D das Fenster bei dieser Einstellung nochmals löscht. Dies wurde vom Autor bei Verwendung von Penguin mit der Applikation `mrtpen`<sup>2</sup> beobachtet. Dies könnte jedoch eine Eigenheit des verwendeten Grafikkartentreibers sein und muß auf anderen Plattformen nicht auftreten.

---

<sup>1</sup>Penguin ist ein Bestandteil des MRT, mit welchem sich Menüs, Buttons und Eingabemasken darstellen lassen.  
<sup>2</sup>`mrtpen` ist eine Beispielanwendung, welche zum MRT-Paket gehört. Er erlaubt unter anderem das interaktive Rotieren und Skalieren der Szene.

Bei Verwendung des OpenGL-Renderers unter Unix kann die Einstellung **NotUsefull** zu Programmabstürzen führen. Dies wurde bei Tests mit der Applikation `mrtpen` von Autor beobachtet. Da das Programm bei einer anderen Einstellung von **CGI.backingStore** nicht auftrat, führt der Autor dieses Problem alleine auf die falsche Einstellung zurück.

Bei Verwendung des OpenGL-Renderers unter Windows muß die Variable **CGI.backingStore** jedoch auf **NotUsefull** gesetzt werden, da ansonsten kein gültiges OpenGL-Fenster erzeugt werden kann. Dieses Problem tritt bei allen Testprogrammen, auch solchen ohne Penguin, auf, so daß dies eine Unverträglichkeit des von Cgi erzeugten Fensters mit dem OpenGL-Renderer zu sein scheint.

#### **CGI.visualDepth**

Dies muß auf die korrekte Farbtiefe der aktuellen Anzeige eingestellt sein, sollte jedoch, wenn 15 Bit Farbtiefe verwendet wird, auf 16 gestellt werden, da ansonsten die Ausgabe des Softwarerenders nicht funktioniert, da Cgi nur 16-Bit-Displays kennt.

#### **CGI.visualClass**

Dies muß für 8-Bit-Displays auf **PseudoColor**, und für alle anderen Farbtiefen auf **TrueColor** gestellt sein. Eine Ausnahme bilden nur die SGI-Computer mit 8 Bit Farbtiefe. Hier muß **TrueColor** eingestellt sein, da OpenGL ansonsten kein gültiges Fenster öffnen kann.

Bei falscher Wahl der Farbtiefe (**CGI.visualDepth**) oder der Farbklasse (**CGI.visualClass**) kann es zu Abstürzen des Programms direkt nach dem Start des Programmes, insbesondere bei Verwendung des XGL-Renderers, kommen.

## Anhang D

### CD-ROM Inhalt

text/	schwall198.ps Diese Diplomarbeit im Postscript-Format.
text/	schwall198.pdf Diese Diplomarbeit im PDF-Format.
archiv/alt_208/	Der Quelltext der gesamten Originalversion 208 des MRT.
archiv/alt_400/	Der Quelltext der gesamten Originalversion 400 des MRT.
archiv/neu_208/	Der Quelltext der neuen Version von Cgi3D und der angepaßten Anwendungen für die Version 208.
archiv/neu_400/	Der Quelltext der neuen Version von Cgi3D und der angepaßten Anwendungen für die Version 400.
src/neu_400/	Der Quelltext der gesamten Version 400 mit der neuen Version von Cgi3D und den angepaßten Anwendungen unkomprimiert in der Unix-Version. Da das ISO9660-Filesystem keine Links unterstützt, sind bei dieser Version die Makefiles mehrfach vorhanden.

# Literaturverzeichnis

- [Ben97] Heinzgerd Bendels. *Eine topologische Datenstruktur und ihre Anwendungen im 3D-Graphiksystem MRT.*, Diplomarbeit, Universität Bonn, Institut für Informatik III, März 1997
- [Fel92] Dieter W. Fellner. *Computergrafik*, Reihe Informatik, Band 58, BI-Wissenschaftsverlag, Mannheim, 2. Auflage, 1992.
- [Fel93] Dieter W. Fellner, Martin Fischer. *CGI' C++ Interface Version 1.0 (based on CGI DIS*, Universität Bonn, Institut für Informatik, Mai 1993
- [Fel94] Dieter W. Fellner. *MRT++ design Issues and Brief Reference*, Universität Bonn, Institut für Informatik III, May 1994.
- [Fis95] Martin Fischer. *Reference Pointers for Class Hierarchies*, Forschungsbericht IAI-TR-95-12, Universität Bonn, Institut für Informatik III, August 1995.
- [Hof97] Jan Eric Hoffmann. *Alles fließt. Computer-Animationen mit OpenGL*, PC Magazin DOS, WEKA Computerzeitschriften Verlag GmbH, Poing, Seite 256-263, Juni 1997
- [LC87] William E. Lorensen and Harvey E. Cline. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Computer Graphics (Proc. SIGGRAPH'87), 21(4):163-169, July 1987
- [Mic97] Microsoft Corporation, *Microsoft DirectX 3 SDK*, Redmond, 1997
- [Mic98] Microsoft Corporation, *Microsoft Direct3D Retained Mode SDK, Version 6.0*, Redmond, August 1998
- [Pie97] Michael Pietsch. *Volumenvisualisierung im Rahmen des MRT*, Diplomarbeit, Universität Bonn, Institut für Informatik III, November 1997.
- [Seg98] Mark Segal, Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, Silicon Graphics, Inc., 1992-1998
- [Sil93] Silicon Graphics, Inc. *The OpenGL reference Manual*, Mountain View, California, 1993
- [Sun93] Sun Microsystems, Inc. *The Solaris XGL Graphics Library*, Mountain View, California, 1993
- [Sun97] Sun Microsystems, Inc. *Solaris XGL 3.3 AnswerBook*, 1997, Internet: <http://docs.sun.com:80/>, Links: "Einführung und Übersicht", "Solaris XGL 3.3 AnswerBook"
- [Tho96] Nigel Thompson. *3D Graphics Programming for Windows 95*. Microsoft Press, Redmond, Washington, 1996

- [Wap99] Armin Wappenschmidt. *Modellierung einer Benutzerschnittstelle für eine Virtual Workbench*, Diplomarbeit, Universität Bonn, Institut für Informatik III, voraussichtlich Mai 1999
- [Wat92] Alan Watt, Mark Watt. *Advanced Animation and Rendering Techniques, Theory and Practice*. ACM Press, Addison-Wesley, 1992.
- [Web94] Johannes Weber. *3D-CGI++*, *Ein Plattform-unabhängiges 3D Grafik Interface basierend auf CGI++*, Diplomarbeit, Universität Bonn, Institut für Informatik III, August 1994.
- [Woo] Woo, Neider, Davis. *OpenGL Programming Guide*, Second Edition, Addison-Wesley.
- [Zer97a] Klaus Zerbe. *Bausatz. Einführung in COM, DirectX und ActiveX*, C't, Verlag Heinz Heise GmbH & Co KG, Hannover, Heft 8, Seite 286-291, 1997
- [Zer97b] Klaus Zerbe. *Dreiecksfüller. C++-Klassenbibliothek vereinfacht Direct3D*, C't, Verlag Heinz Heise GmbH & Co KG, Hannover, Heft 9, Seite 280-291, 1997

# CD-ROM