Universität Zürich
Institut für Informatik

Entwicklung eines parallelen OpenGL Polygon-Renderers

Semesterarbeit im Fach Informatik
vorgelegt von
Tobias Wolf
Näfels GL, Schweiz
Matrikelnummer 97-913-107

Angefertigt am
Institut für Informatik
der Universität Zürich
Prof. Dr. Renato Pajarola

Betreuer: Stefan Eilemann
Abgabe der Arbeit: 30.09.2007

Zusammenfassung

Equalizer ist ein Open Source Framework für verteiltes, paralleles Rendering. Es setzt auf OpenGL auf, einer plattformunabhängigen Programmierschnittstelle für Computergrafik. Im Rahmen dieser Semesterarbeit wird auf der Basis von Equalizer und OpenGL ein Programm zur effizienten Darstellung von polygonalen Daten entwickelt.

In einem ersten Schritt wird eine effiziente Datenstruktur für die Verwaltung der polygonalen Daten umgesetzt. In einem zweiten Schritt wird ein optisch ansprechendes Rendering für die Darstellung dieser Daten implementiert. Das Ergebnis dieser Arbeit ist eine neue Beispielapplikation für Equalizer.

Abstract

Equalizer is an open source framework for distributed, parallel rendering. It is built upon OpenGL, a platform independant application programming interface for computer graphics. In the course of this semester thesis an application for the efficient presentation of polygonal data is developed, based upon Equalizer and OpenGL.

In a first step, an efficient data structure for the storage of the polygonal data is developed. In a second step, a visually appealing rendering for the presentation of that data is implemented. The result of this thesis is a new example application for Equalizer.

Inhaltsverzeichnis

1. Einleitung

Die Leistung von Grafikhardware hat sich in den letzten Jahren kontinuierlich und rasant weiter entwickelt. Mit höherer Leistungsfähigkeit gehen aber auch höhere Ansprüche einher. Es besteht der Wunsch, immer komplexer werdende Daten mit steigendem Datenvolumen in höheren Auflösungen zu visualisieren. Trotz des Fortschrittes gelangen tradionelle Computergrafik Systeme in gewissen Anwendungsgebieten an ihre Grenzen, denn obwohl sich die Hardwareleistung in regelmässigen Abständen verdoppelt, wachsen die produzierten und darzustellenden Datenmengen mindestens genau so schnell [10].

Dem ähnlich gelagerten Problem einer zu geringen Rechenleistung einzelner Computersysteme wurde begegnet, indem mehrere verbundene Rechnersysteme die gewünschten Berechnungen verteilt aber parallel ausführen. Analog dazu lässt sich auch im Bereich der Computergrafik die notwendige Arbeit auf mehrere Grafiksysteme verteilen, welche dann einzelne Teilbereiche parallel bearbeiten und die Teilergebnisse wo notwendig wieder zu einem Gesamtbild zusammensetzen.

1.1. Aufgabenstellung

Equalizer [4][5] ist ein Lösungsansatz für die im letzten Abschnitt vorgestellte Problematik. Es handelt sich bei Equalizer um ein Open Source Framework für verteiltes, paralleles *Rendering*[1], welches sich zur Zeit in Entwicklung befindet. Es setzt auf *OpenGL* [8] auf, der bedeutendsten Programmierschnittstelle für plattformübergreifende, anspruchsvolle Grafikprogrammierung.

Ziel der vorliegenden Semesterarbeit ist die Entwicklung einer Applikation für die effiziente und optisch ansprechende Darstellung von polygonalen Daten, auf Basis des Equalizer Frameworks und der OpenGL Programmierschnittstelle. Ein bereits existierendes Beispielprogramm wird als Grundlage verwendet, und soll am Ende durch das Ergebnis dieser Arbeit vollständig ersetzt werden.

Die erste Teilaufgabe besteht darin, eine effiziente Datenstruktur für die Verwaltung von

[1] Englische Begriffe werden im Glossar am Ende des Dokumentes übersetzt oder erklärt.

polygonalen Daten umzusetzen. Die Wahl fiel hierbei auf eine besondere Baumstruktur, den sogenannten *kd-Tree* [17].

Die polygonalen Daten in Form von *Vertices* und *Triangles* werden aus Dateien im PLY Dateiformat [11] gelesen, und anschliessend in die Baumstruktur einsortiert. Die fertige Datenstruktur erlaubt sowohl ein einfaches *View Frustum Culling* als auch ein effizientes Rendering. Da das Erstellen des kd-Trees ein sehr aufwändiger Prozess ist, werden die Daten für zukünftige Verwendung in einem eigenen Binärformat zwischengespeichert.

Die zweite Teilaufgabe schliesslich ist die Implementierung eines Rendering Verfahrens für die optisch ansprechende Darstellung der polygonalen Daten.

Die Datenstruktur ist optimiert für die Verwendung von *Vertex Buffer Objects* (VBOs) [15], welche entsprechend auch für das Rendering verwendet werden sollen. Zur Verbesserung der visuellen Qualität sollen verschiedene Algorithmen einsetzt werden, welche mit Hilfe der *Open GL Shader Language* (GLSL) [9] entwickelt werden und direkt auf der Grafikhardware ablaufen. Damit auch ältere OpenGL Umgebungen, welche weder VBOs (ab Version 1.5) noch GLSL (ab Version 2.0) unterstützen, die Applikation ausführen und darstellen können, wird eine OpenGL Version 1.1 kompatible Lösung ebenfalls implementiert.

1.2. Gliederung

Der Aufbau der Arbeit sieht wie folgt aus. Nach der Einführung in diesem Kapitel beschäftigt sich Kapitel 2 mit dem grundliegenden Aufbau der Applikation. Weil dazu auch ein gewisses Verständnis der Equalizer Architektur notwendig ist, werden wichtige Elemente daraus kurz vorgestellt.

Kapitel 3 widmet sich ausführlich dem ersten Teil der Aufgabe. Es wird die bisher verwendete Datenstruktur erklärt und deren Schwächen aufgezeigt. Die neue Datenstruktur wird vorgestellt und auch auf deren konkrekte Implementierung eingegangen.

Der zweite Teil der Aufgabe wird in Kapitel 4 behandelt. Es werden zunächst die verschiedenen möglichen OpenGL Rendermodi vorgestellt, und dann deren Verwendung in der bestehenden und neuen Applikation aufgezeigt. Anschliessend werden die unterschiedlichen Algorithmen des *Shading* diskutiert, sowie deren Anwendung sowohl in Standard OpenGL als auch mit GLSL.

Den Abschluss der Arbeit stellt Kapitel 5 dar, in welchem das Ergebnis der Arbeit kurz zusammengefasst sowie ein Ausblick auf weitere zukünftige Erweiterungen der Applikation geworfen wird.

2. Applikationsarchitektur

In diesem Kapitel wird die generelle Architektur der Applikation diskutiert. Da grosse Teile der Architektur durch das Equalizer Framework beeinflusst werden, sollen zu Beginn des Kapitels die wichtigsten Konzepte aus Equalizer kurz vorgestellt werden. Im Anschluss daran wird dann auf die Applikation selbst eingegangen.

2.1. Equalizer

In dieser Sektion sollen einige Aspekte von Equalizer kurz beleuchtet werden, die einen Einfluss auf die Architektur und das Design der Applikation haben. Dies sind zum einen eine handvoll Equalizer Klassen, und die zur Verfügung gestellten Dekompositionsmodi zum anderen.

2.1.1. Klassen

Equalizer abstrahiert die verschiedenen Systemresourcen in mehreren Klassen. Im folgenden sollen die für das Verständnis der Beispielapplikation notwendigen Klassen kurz vorgestellt werden. Tabelle 2.1 stellt die beschriebenen Klassen gegen Ende des Abschnittes zusammengefasst dar.

An oberster Stelle der Resourcenverwaltung steht eine Konfiguration, welche von der Klasse *eq::Config* repräsentiert wird. Die Konfiguration enthält zum einen eine Beschreibung der zu verwendenden physikalischen und logischen Resourcen, zum anderen eine Definition wie die verschiedenen Arbeitsschritte im Detail auf die einzelnen Resourcen zu verteilen sind.

An nächster Stelle stehen Knoten der Klasse *eq::Node*, welche einzelne Computer innerhalb des Rendering Clusters symbolisieren. Jeder solche Knoten besitzt eine oder mehrere Grafikkarten, die von der Klasse *eq::Pipe* repräsentiert werden. *Pipes* enthalten ihrerseits wiederum Fenster der Klasse *eq::Window*, welche einzelne OpenGL *Drawables*

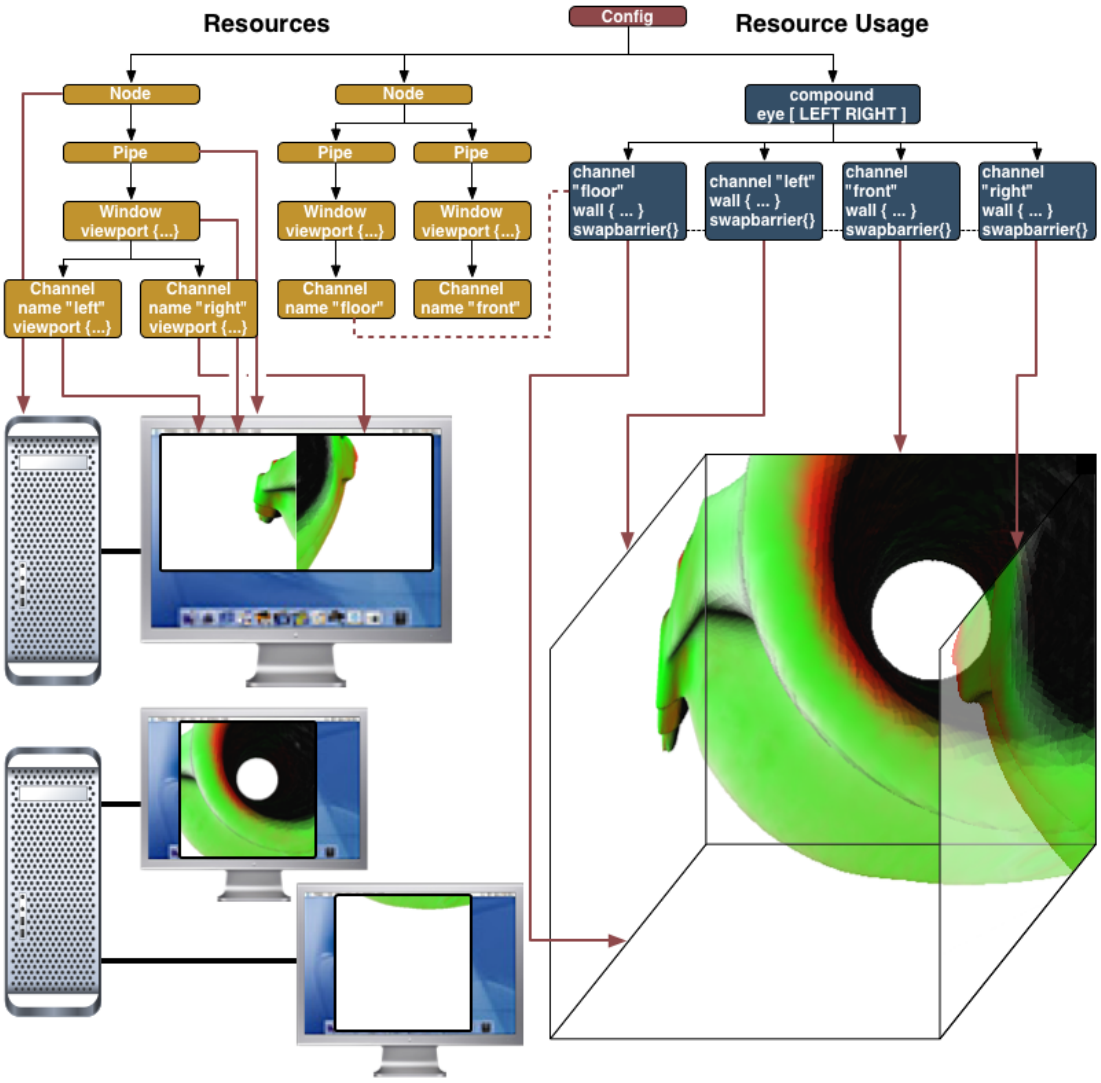

Abbildung 2.1.: Eine Equalizer Beispielkonfiguration (Quelle: [5])

und *Contexts* verwalten. Jedes Fenster kann abschliessend noch mehrere *Viewports* besitzen, die durch die Klasse *eq::Channel* repräsentiert werden. Resourcen auf der selben Pipe teilen sich vorhandene OpenGL Objekte wie beispielsweise Display Listen, Texturen und VBOs.

Abbildung 2.1 stellt ein Beispiel einer solchen Konfiguration dar. Es handelt sich um eine CAVE™(Cave Automatic Virtual Environment), mit zwei Computern (Nodes), drei Grafikkarten (Pipes) und zugehörigen Fenstern (Windows), die zusammen vier verschiedene Kanäle (Channels) rendern, die den Projektionen auf die vier Wände entsprechen.

Neben den oben genannten Resourcenklassen spielt noch eine weitere Klasse eine wichtige Rolle für das Verständnis der Applikation. Da mehrere Kopien der Applikation im

Cluster laufen, müssen sich diese gemeinsam genutzte Daten auf irgend eine Weise teilen können. Equalizer stellt dafür verteilte Objekte der Klasse *eqNet::Object* bereit, die sofern benötigt auch Versionierung unterstützen.

Das Equalizer Framework sieht vor, dass Applikationen eigene Subklassen von den genannten Basisklassen ableiten. Innerhalb der verschiedenen Klassen stehen eine Reihe von vordefinierten *Task* Methoden zur Verfügung, welche von der Applikation wiederum nach Bedarf überschrieben werden können um die gewünschte Funktionalität zu erreichen. Eine detaillierte Beschreibung der Equalizer Task Methoden findet sich in [6].

Equalizer Klasse	**Bedeutung**
eq::Config	Beschreibung vorhandener Resourcen und deren Verwendung
eq::Node	Einzelner Computer innerhalb des Rendering Clusters
eq::Pipe	Grafikkarte(n) des jeweiligen Knotens
eq::Window	OpenGL Drawables auf der entsprechenden Pipe
eq::Channel	Viewports innerhalb des zugehörigen Fensters
eqNet::Object	Verteilte Objekte mit Unterstützung für Versionierung

Tabelle 2.1.: Wichtige Equalizer Klassen und deren Bedeutung

2.1.2. Dekompositionsmodi

Zum Abschluss dieser Sektion soll noch ein Blick auf die verschiedenen Dekompositionsmodi geworfen werden die Equalizer unterstützt, da diese ebenfalls einen gewissen Einfluss auf das Applikationsdesign haben. Zur Übersicht sind diese in Tabelle 2.2 gegen Ende wieder gemeinsam aufgeführt.

2D: Im 2D Modus wird der zweidimensionale Bildschirmraum in mehrere Kacheln aufgeteilt (siehe Abbildung 2.2). Verschiedene Rendereinheiten bearbeiten nur ihren eigenen Teil, und die Teilergebnisse werden am Ende zu einem Gesamtbild zusammengefügt. Damit jede Rendereinheit effizient arbeitet, sollte die Applikation View Frustum Culling einsetzen und die Datenstruktur entsprechende Unterstützung bieten.

DB: Im DB Modus wird die Datenstruktur aufgeteilt. Jede Rendereinheit ist für einen bestimmten Anteil der polygonalen Daten zuständig und rendert nur diese (siehe Abbildung 2.3). Die Zusammensetzung erfolgt dann mit Hilfe der Tiefenpufferinformation. Damit dieser Modus funktioniert muss die Datenstruktur zusätzliche Informationen enthalten, die Teilbereiche auf eindimensionale Intervalle abbilden.

Eye: Der Eye Modus wird im Stereo Rendering verwendet, verschiedene Rendereinheiten sind für die Ansicht welche jeweils ein bestimmtes Auge sieht verantwortlich. Dieser Modus hat keinen Einfluss auf das Applikationsdesign.

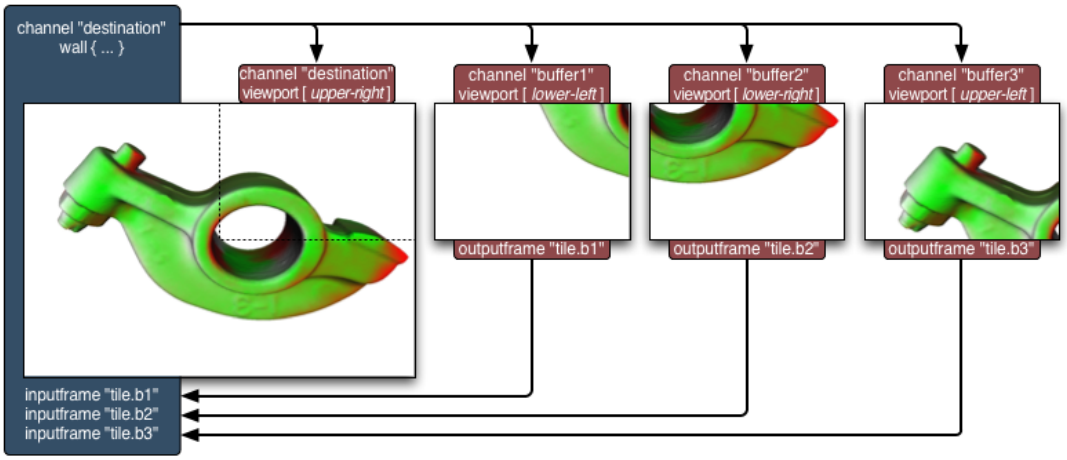

Abbildung 2.2.: Equalizer 2D Dekompositionsmodus (Quelle: [5])

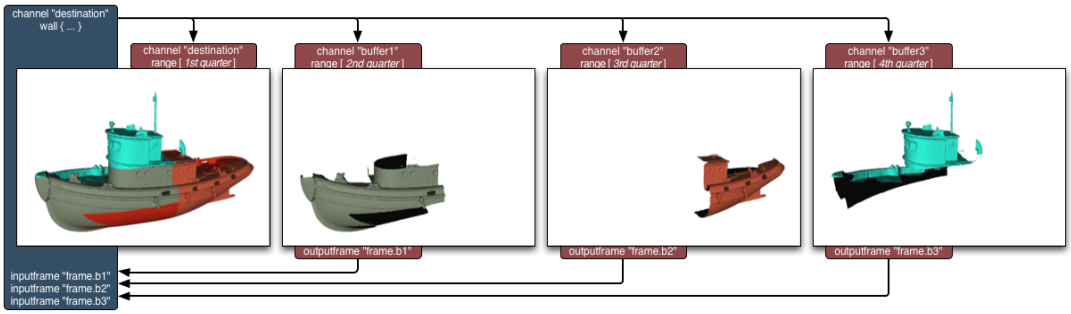

Abbildung 2.3.: Equalizer DB Dekompositionsmodus (Quelle: [5])

Neben diesen einfachen Dekompositionen unterstützt Equalizer auch beliebige Kombinationen derselben. Dies hat jedoch auf die Applikation keinen weiteren Einfluss und wird darum hier nicht weiter erläutert.

Modus	**Auswirkungen**
2D	Applikation und Datenstruktur sollten Frustum Culling unterstützen
DB	Datenstruktur sollte auf eindimensionale Intervalle abbildbar sein
Eye	Keine besonderen Auswirkungen

Tabelle 2.2.: Equalizer Dekompositionsmodi und deren Auswirkung auf die Applikation

2.2. eqPly

Nachdem nun die grundlegenden Aspekte von Equalizer behandelt wurden, welche Einfluss auf Applikationsarchitektur und -design nehmen, wird in dieser Sektion nun die Applikation selbst betrachtet. Sie trägt den Namen **eqPly**.

Aufgrund der relativ klar vorgegebenen Struktur für Equalizer Programme weicht die Architektur der neuen Applikation kaum von der des alten Beispielprogrammes ab. Abbildung 2.4 zeigt die Zusammenhänge der Klassen sowohl untereinander als auch in Relation zu Equalizer. Da der Aufbau von *eqPly* in [3] schon ausführlich behandelt wird, beschränken sich die folgenden Ausführungen auf ein Minimum. Die Klassen betreffend Datenstruktur und Rendering (*mesh::Vertex**) werden gesondert in Kapitel 3 und 4 behandelt.

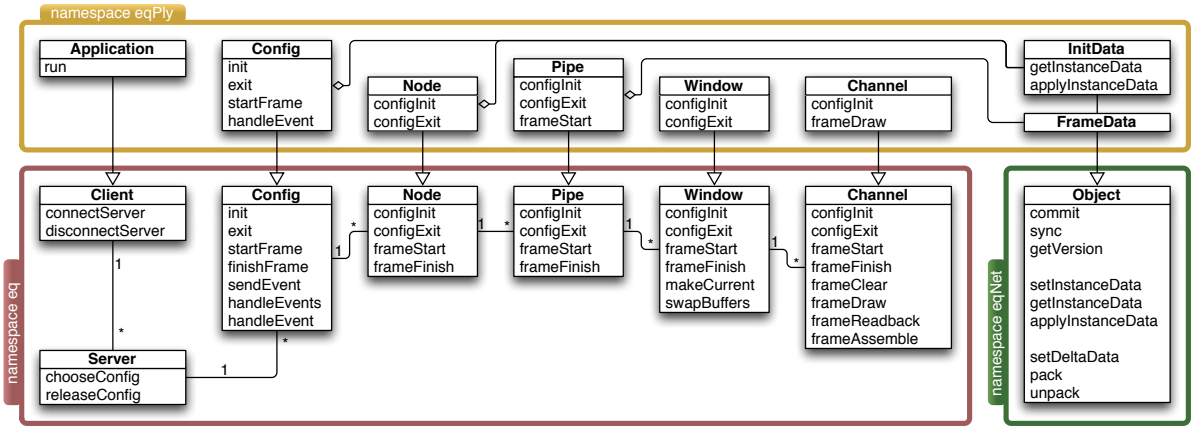

Abbildung 2.4.: Zusammenhänge der Klassen von eqPly und Equalizer (Quelle: [3])

2.2.1. Datenklassen

Die Datenklassen sind verteilte Objekte, und somit von *eqNet::Object* abgeleitet. *eqPly::InitData* kapselt den Knoten übergreifenden, aber statisch bleibenden Zustand der Applikation. Hauptsächlich sind dies der Pfad zum Modell, ein Schalter ob der neue VBO Rendermodus verwendet werden soll, und die ID des zu verwendenden *Frame* Datenobjektes.

eqPly::LocalInitData ist von *eqPly::InitData* abgeleitet und enthält den statischen, lokalen Knoten spezifischen Zustand, bespielsweise den Port zur Kommunikation mit einem eventuell vorhandenen *Head Tracker*, oder ob der Knoten beim Rendering Farben des Modells nutzen oder ignorieren soll. Die Klasse ist auch zuständig für das Parsen der Kommandozeilenparameter.

eqPly::FrameData ist ein versioniertes, verteiltes Objekt und enhält dynamische Daten die sich von einem Frame zum anderen verändern können. Dies sind in *eqPly* vor allem die Rotations- und Translationsmatrizen für die Kameraposition, welche sich sowohl durch Benutzerinteraktion als auch durch kontinuierliche Animation fortlaufend ändert.

2.2.2. Abgeleitete Klassen

Wie im Abschnitt 2.1.1 bereits beschrieben, leiten Equalizer Applikationen eine Reihe von Basisklassen ab und überschreiben darin die Task Methoden um ihre gewünschte Funktionsweise zu implementieren. Genau dies geschieht in den Klassen *eqPly::Config, eqPly::Node, eqPly::Pipe, eqPly::Window* und *eqPly::Channel*.

eqPly::Config ist die Steuerung der Applikation. Die Klasse verwaltet die Primärkopien der *eqPly::InitData* und *eqPly::FrameData* Objekte und behandelt die Registrierung dieser verteilten Objekte. Sie initialisiert und kommuniziert mit einem eventuell vorhandenen Head Tracker, ist für die Ereignisbehandlung zuständig und aktualisiert sämtliche Frame relevanten Daten zu Beginn eines neuen Frames. Dazu gehört insbesonders die neue Position der Kamera.

Die Aufgaben von *eqPly::Node* sind das *Mapping* der verteilten statischen *eqPly::InitData* auf eine lokale Kopie sowie das Laden des Polygonmodells aus der in *eqPly::InitData* übermittelten Datei.

Da nicht von allen Rendereinheiten zur gleichen Zeit derselbe Frame gerendert werden muss, unterhält *eqPly::Pipe* ein Mapping des dynamischen *eqPly::FrameData* Objekts. Zu Beginn eines neuen Frames wird die ID des zu rendernden Frames übermittelt, und die Klasse synchronisiert ihre Kopie der *eqPly::FrameData* mit der gewünschten Version.

eqPly::Window enthält den *VertexBufferState*, welcher den *State* für das Rendering der Datenstruktur sowie die auf der selben Pipe gemeinsam genutzten OpenGL Objekte verwaltet. *eqPly::Window* ist auch zuständig für das Laden des Equalizer Logos, die Initialiserung der OpenGL *Extensions* sowie der Auswahl des zu verwendenden Rendermodus. Auf die State Klasse und OpenGL Extensions wird in Kapitel 4 noch näher eingegangen.

eqPly::Channel leistet die Hauptarbeit der Applikation. Die Klasse ist zuständig für das Zeichnen des Logos, die Berechnung des Frustums, für die Anwendung der verschiedenen Modell- und Projektionstransformationen sowie das View Frustum Culling und Rendering des Modells.

2.2.3. Hilfsklassen

Zwei weitere Module sind Bestandteil von *eqPly*. Zum einen handelt es sich um die Dateien *ply.h* und *plyfile.cpp*, welche die Bibliothek für das Laden der PLY Dateien darstellen. Da sie ursprünglich in reinem C geschrieben wurde, ist die Bibliothek weder objektorientiert noch besonders einfach zu verwenden.

Zum anderen sind da noch die Dateien *tracker.h* und *tracker.cpp*. Dabei handelt es sich um eine Klasse zur Kommunikation mit einem Head Tracker, der von *eqPly* bei Vorhandensein verwendet werden kann um Kopfbewegungen eines Benutzers zu verfolgen und die Ansicht entsprechend anzupassen.

2.2.4. Hauptprogramm

Die *eqPly::Application* Klasse ist das Gerüst der Applikation. Zusammen mit der *main* Funktion werden die notwendigen Klassen instanziert, sowie die benötigten Schritte für das Aufsetzen einer Equalizer Applikation in die Wege geleitet. Da diese ausführlich in [3] beschrieben sind, wird an dieser Stelle nicht weiter darauf eingegangen.

3. Datenstruktur

In diesem Kapitel wird die erste Teilaufgabe der Arbeit behandelt, die Datenstruktur. In einem ersten Schritt wird die alte Datenstruktur analysiert und erläutert sowie deren Probleme aufgezeigt. In einem zweiten Schritt werden dann die Anforderungen an die neue Datenstruktur spezifiziert und das gewählte Design sowie die schlussendliche Implementierung vorgestellt.

3.1. Alte Datenstruktur

Der Kern der alten Datenstruktur ist ein *Bounding Box Octree* [18]. Die *Bounding Boxes* sind achsenparallele Quader welche die enthaltenen Daten minimal umschliessen. Der *Octree* ist eine Baumstruktur zur binären Raumpartitionierung bei der ein Knoten entweder keine oder acht Kinder hat.

Ein Knoten des Baumes wird solange mit Daten befüllt, bis eine vordefinierte Grenze erreicht ist. Bei Überschreiten dieser Grenze wird die Bounding Box des Knotens in acht gleich grosse Oktanten unterteilt, welche die neuen Kinder des Knotens sind. Die im Knoten enthaltenen Daten werden gemäss ihrer neuen Zugehörigkeit auf die Kinder verteilt. Dieser Vorgang wird solange wiederholt, bis ein Blatt genug Platz für die Daten bietet, oder eine gewisse maximale Baumtiefe erreicht wurde.

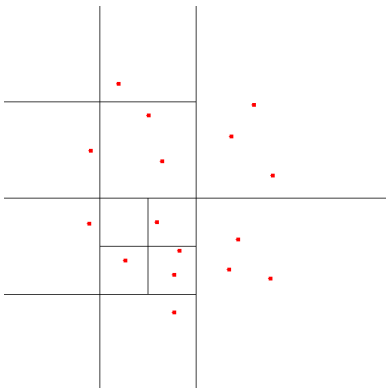

Abbildung 3.1.: 2D Raumaufteilung mittels Quadtree

Abbildung 3.1 soll dies anhand eines grafischen Beispiels verdeutlichen. Um die Komplexität einer dreidimensionalen Darstellung zu vermeiden, wird der zweidimensionale Fall eines *Quadtree* (Teilung in vier Quadranten) mit einzelnen Punkten als Daten gezeigt. Die Datemenge umfasst hier 16 Punkte, und die Grenze pro Knoten beträgt 3 Punkte. Erzeugt wurde die Abbildung mit Hilfe von [14].

3.1.1. Analyse

Das alte Modell wird primär repräsentiert durch eine Klasse *PlyModel*. Es handelt sich dabei um eine Template Klasse, die als Template Argument den Typ der zu speichernden *Faces* (in diesem Fall Triangles) benötigt. Dieser Typ sieht konkret wie folgt aus:

```
template<class VertexType>
struct NormalFace
{
    float      normal[3];
    VertexType vertices[3];
};
```

Es wird nur eine Flächennormale gespeichert, und der Typ der drei Vertices ist wiederum ein Template Argument. Es stehen zwei verschiedene Vertextypen zur Auswahl, eine erste Variante die nur Vertexpositionen beinhaltet, und eine zweite die zusätzlich noch eine Farbe speichert:

```
struct Vertex
{
    float pos[3];
};

struct ColorVertex : public Vertex
{
    float color[3];
};
```

Der Kern der Klasse *PlyModel* ist die Kapselung des Bounding Box Octrees, entsprechend enthält sie einen eigenen Typ für die Bounding Boxes mit denen die polygonalen Daten effektiv verwaltet werden:

```
struct BBox
{
    BBox*          parent;
    BBox*          next;
    BBox*          children;
```

```
    Vertex          pos[2];
    Vertex          cullBox[2];

    vmml::Vector4f  cullSphere;

    size_t          nFaces;
    FaceType*       faces;

    float           range[2];

    void*           userData;
};
```

Die Datenstruktur enthält Zeiger zu ihrem Elternknoten, dem nächsten Nachbarn auf gleicher Baumtiefe sowie den potentiellen Kindern. Darüber hinaus wird die dem Knoten entsprechende Bounding Box und eine davon abgeleitete Bounding Sphere (erlaubt effizienteres View Frustum Culling) gespeichert, ebenso ein dem Anteil der enthaltenen Daten entsprechendes eindimensionales Intervall (benötigt für Equalizer DB Dekomposition) sowie die Anzahl der enthaltenen Triangles und ein Zeiger auf eine Liste derselben.

Die Klasse *PlyModel* bietet basierend auf dieser Struktur eine Reihe von Methoden an. Das Erstellen und Füllen des Bounding Box Octrees wird dabei genau so unterstützt wie das Speichern und Laden einer binären Repräsentation. Die Klasse verwaltet ebenfalls die Hauptkopie der Liste aller Triangles und bietet eine Reihe von Hilfsfunktionen wie zum Beispiel das Skalieren der Modelldaten oder der Berechnung von Flächennormalen und Bounding Spheres.

Die Methoden für das Einlesen der Daten aus PLY Dateien sind in einer eigenen Klasse *PlyFileIO* untergebracht, welche die schon in Abschnitt 2.2.3 erwähnte Biobliothek dafür nutzt. Das Rendering der alten Datenstrukur ist vollständig von dieser externalisiert, und befindet sich in *eqPly::Channel*.

3.1.2. Probleme

Sowohl die verwendete Octree Baumstruktur als auch die Implementierung des Polygonmodells offenbart eine Reihe von Schwächen. Im folgenden sollen zunächst die Probleme des Octrees aufgezeigt werden.

Bei Überschreiten der Füllgrenze pro Knoten wird der Raum immer in acht gleich grosse Oktanten unterteilt, dies geschieht ohne Rücksicht auf die tatsächliche Verteilung der Daten. In der Folge ist die Blattgrösse nur sehr schlecht zu kontrollieren, sie kann zwischen leer bis hinauf zur Füllgrenze variieren. Ein weiterer Effekt dieser fehlenden

Kontrolle in Bezug auf die Knotengrössen ist, dass die Baumtiefe in verschiedenen Bereichen des Baumes stark schwanken kann, je nach Dichte der Daten in diesem Bereich. Das Fazit ist, dass der Octree in dieser Form äussert schlecht ausbalanciert ist.

Auch die konkrete Implementierung zeigt eine Reihe von Schwächen. Die PLY Biobliothek alloziert für mehrere Rückgabevariablen Speicher, der von den aufrufenden Methoden in *PlyFileIO* nicht vollständig freigegeben wird. Die Folge sind mögliche Speicherlecks beim Einlesen der PLY Dateien.

Das Modell verwaltet direkt Triangles, und speichert darum gemeinsam genutzte Vertices mehrfach. Informationen über den Aufbau der polygonalen Daten wie welche Triangles welche Vertices teilen (und entsprechend benachbart sind) gehen verloren. Dies und die Tatsache dass nur Flächennormalen gespeichert werden, führt dazu dass beim Rendering nur Flat Shading verwendet werden kann, und andere Shading Verfahren aufgrund von fehlenden Informationen gar nicht realisiert werden können.

Die mehrfache Template Benutzung führt zur leicht kryptischen Deklaration eines konkreten Modells der Form **typedef** PlyModel< NormalFace< ColorVertex > > Model;. Darüber hinaus ist der Implementierung die C Herkunft noch deutlich anzusehen (Verwendung von C Arrays, malloc() usw.), und sie ist allgemein eher unübersichtlich und ihre Funktionsweise nur schwer nachzuvollziehen. Ein letztes Problem ist, dass die alte Datenstruktur in ihrer Implementierung sehr viel Speicher verbraucht.

3.2. Neue Datenstruktur

Aufgrund der nicht vorhandenen Balancierung des Octrees und der schlechten Kontrolle über die Blattgrössen, wurde ein kd-Tree [17] als neuer Kern der Datenstruktur ausgewählt.

Der kd-Tree ist wie der Octree eine Sruktur zur binären Raumpartitionierung, die Aufteilung erfolgt jedoch nach einem anderem Verfahren, was zu einer Reihe von Vorteilen führt. Der grundlegende Algorithmus zur Erstellung eines kd-Trees läuft wie folgt ab.

In jedem Schritt werden die Daten zunächst entlang einer bestimmten Achse sortiert, die Sortierachse rotiert dabei mit der Rekursionstiefe. Danach wird der Median ermittelt, und die Datenmenge in zwei Häften geteilt. Die eine enthält Objekte kleiner als der Median, und die andere Objekte grösser oder gleich dem Median. Beide Hälften werden rekursiv weiter verarbeitet, bis die gewünschte Blattgrösse erreicht wird.

Abbildung 3.2 soll dies anhand eines grafischen Beispiels verdeutlichen. Er wird der zweidimensionale Fall gezeigt. Die verwendeten Daten sind die selben 16 Punkte wie

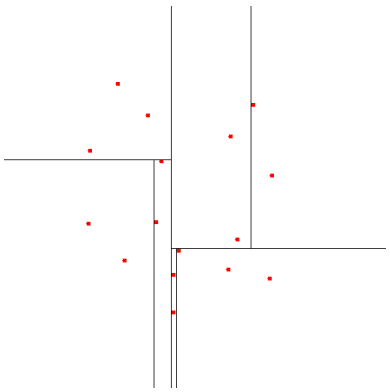

Abbildung 3.2.: 2D Raumaufteilung mittels kd-Tree

im Quadtree Beispiel in Abbildung 3.1, und die Grenze für die Blattgrösse ist ebenfalls wieder 3. Die erste Aufteilung spaltet den Raum auf der x-Achse, die zweite auf der y-Achse, und die dritte und letzte wiederum auf der x-Achse.

Wie aus der Beschreibung des Algorithmus und dem Beispiel ersichtlich sein sollte, hat ein so erstellter kd-Tree mehrere Vorteile: Die Anzahl Knoten fällt geringer aus als beim Quadtree/Octree, weil keine leeren Bereiche entstehen. Da die Raumteilung immer anhand eines Datenwertes erfolgt, sind die geteilten Bereiche (fast) gleich gross, und die Baumtiefe ist ebenfalls gleichmässig. Im Endeffekt ist dieser Baum also beinahe perfekt ausbalanciert.

3.2.1. Anforderungen

Es werden an die neue Datenstruktur eine Reihe von Anforderungen gestellt, die zum Teil schon durch die Wahl des kd-Tree erfüllt werden, zum anderen aber im Design und in der Implementierung Berücksichtigung finden müssen. Die folgenden Anforderungen sind zum Teil neu, und galten zum Teil auch schon für die alte Datenstruktur und müssen entsprechend auch von der neuen erfüllt werden.

Erstens soll die Baumstruktur besser ausbalanciert sein, und eine genauere Kontrolle über die Blattgrössen aufgrund der Verwendung von VBOs beim Rendering erreicht werden. Diese beiden Punkte werden direkt durch die kd-Tree Auswahl erfüllt.

Zweitens sollen die Informationen über die Struktur der polygonalen Daten erhalten bleiben, also die Nachbarschaftsbeziehungen zwischen Triangles. Dies einerseits, um bei Speicherung und Rendering Vorteile aus den gemeinsam genutzten Vertices ziehen zu können, und andererseits um Vertexnormalen anstatt von Flächennormalen berechnen zu können. Diese erlauben erst die Verwendung besserer Shading Algorithmen.

Drittens soll der neue Code heutigen objektorientierten Ansprüchen genügen, die Möglichkeiten von C++ besser ausnutzen und natürlich einfach verständlich und gut wartbar sein.

Da Equalizer selbst Unterstützung für mehrere Betriebssysteme (Mac OS X, Linux, Windows) sowohl in 32 als auch 64 Bit bietet und auf unterschiedlichen Prozessorfamilien (mit Little- und Big-Endian Datenformat) läuft, müssen die entsprechenden Eigenheiten auch in Hinblick auf die neue Datenstruktur mit berücksichtigt werden.

Schliesslich soll die neue Datenstruktur auch wieder effizientes View Frustum Culling und ein Mapping der Baumbereiche auf ein eindimensionales Intervall für die DB Dekomposition von Equalizer unterstützen.

3.2.2. Design

Basistypen

Um die verwendeten Datentypen flexibel zu halten, aber auf eine übermässige Verwendung von Templates zu verzichten, wurde entschieden die Basistypen der Datenstruktur in Form von eigenen Typedefs zu bilden.

```
typedef vmml::Vector3< GLfloat >      Vertex;
typedef vmml::Vector4< GLubyte >      Color;
typedef vmml::Vector3< GLfloat >      Normal;
typedef size_t                        Index;
typedef GLushort                      ShortIndex;
```

Besondere Erwähnung verdient hierbei die Definition von zwei verschiedenen Indextypen. Grund hierfür ist, dass für effizientes Rendering die verwendeten Indices im Bereich eines Short Datentyps liegen sollen. Das Einsortieren der polygonalen Daten in den kd-Tree arbeitet noch mit den globalen, grösseren Indices. Sobald jedoch ein Blatt erzeugt wird, werden die darin enthaltenen Triangles neu indexiert. Dies impliziert natürlich, dass ein Blatt maximal so viele unterschiedliche Vertices enthalten darf wie im Short Bereich Platz finden.

Auf Basis dieser Primärtypen sowie einer Wrapper Hilfsklasse für kleine Arrays mit fester Grösse werden dann die folgenden zusammengesetzten Typen definiert.

```
typedef vmml::Vector3< Index >        Triangle;
typedef ArrayWrapper< Vertex, 2 >     BoundingBox;
typedef vmml::Vector4< float >        BoundingSphere;
typedef ArrayWrapper< float, 2 >      Range;
```

Effizientes View Frustum Culling lässt sich einfacher mit Bounding Spheres als mit Bounding Boxes realisieren, da der Überlappungstest einfacher ausfällt. Die Struktur wird selber nur die Bounding Spheres speichern, Boxes kommen aber noch beim Aufbau temporär zum Einsatz, da sich diese besser zusammenfassen lassen.

Mesh Klassen

Nachdem die Basistypen vorgestellt wurden, können nun die Klassen betrachtet werden. Da die Datenstruktur Polygonmeshes verwaltet, wird für alle Belange der Datenstruktur der eigene Namespace *mesh* verwendet.

mesh::VertexData speichert die polygonalen Daten in ihrer Ursprungsform, zur Verwendung kommen C++ STL Vektoren. Darüber hinaus besitzt die Klasse alle notwendigen Methoden um PLY Dateien einzulesen, die Bounding Box des gesamten Modells zu ermitteln, das Modell zu skalieren, die Normalen zu berechnen sowie die Triangles für die Erzeugung des kd-Trees zu sortieren.

```
class VertexData
{
public:
    VertexData();

    bool readPlyFile( const char* file, const bool ignoreColors );
    void sort( const Index start, const Index length,
               const Axis axis );
    void scale( const float baseSize );
    void calculateNormals( const bool vertexNormals );
    void calculateBoundingBox();
    const BoundingBox& getBoundingBox() const;

    std::vector< Vertex >   vertices;
    std::vector< Color >    colors;
    std::vector< Normal >   normals;
    std::vector< Triangle > triangles;

private:
    ...
    BoundingBox _boundingBox;
};
```

mesh::VertexBufferData verwaltet die sortierten und reindexierten Daten für den kd-Tree. Während der Erstellung des Baumes werden die Elemente von *mesh::VertexData* in die Vektoren dieser Klasse kopiert, entsprechend kann *mesh::VertexData* nach Erstellen des Baums direkt entsorgt werden. Da *mesh::VertexBufferData* Bestandteil des kd-Tree

ist, besitzt es wie alle Klassen des Baumes selbst Methoden zum Speichern seiner Daten in einen Stream bzw. zum Laden aus einem memory-mapped File.

```
class VertexBufferData
{
public:
    void clear();
    void toStream( std::ostream& os );
    void fromMemory( char** addr );

    std::vector< Vertex >       vertices;
    std::vector< Color >        colors;
    std::vector< Normal >       normals;
    std::vector< ShortIndex >   indices;

private:
    ...
};
```

mesh::VertexBufferState ist eine Hilfsklasse, die den Renderstate für den kd-Tree abstrahiert. Dies sind insbesondere der zu verwendende Rendermodus (siehe Kapitel 4.1), die Verwendung von Farben sowie die Verwaltung von OpenGL Objekten und Extension Functions. Die Klasse selbst ist abstrakt, konkrete Versionen sind für eine stand-alone Nutzung *mesh::VertexBufferStateSimple* und *mesh::EqVertexBufferState* für die Verwendung mit Equalizer.

```
class VertexBufferState
{
public:
    virtual bool useColors() const;
    virtual void setColors( const bool colors );
    virtual RenderMode getRenderMode() const;
    virtual void setRenderMode( const RenderMode mode );

    virtual GLuint getDisplayList( const void* key );
    virtual GLuint newDisplayList( const void* key );
    virtual GLuint getBufferObject( const void* key );
    virtual GLuint newBufferObject( const void* key );

    virtual void deleteAll();

    virtual const GLFunctions* getGLFunctions() const;

protected:
    VertexBufferState( const GLFunctions* glFunctions );
    virtual ~VertexBufferState();
```

```
    const GLFunctions*   _glFunctions;
    bool                 _useColors;
    RenderMode           _renderMode;

private:
};
```

mesh::VertexBufferBase ist die abstrakte Basisklasse für alle anderen Baumklassen. Sie definiert das Interface welches alle Typen von Knoten anbieten müssen, und implementiert bereits Basisfunktionalität soweit dies möglich ist. Im Gegensatz zur alten Datenstruktur ist das Rendering nun Bestandteil der Struktur selbst. Entsprechend muss jeder Knoten eine Methode für das Rendering zur Verfügung stellen. Weitere öffentliche Methoden sind die Rückgabe der Kinder und der Bounding Spheres für das externe View Frustum Culling sowie des abgedeckten Intervalls für die Equalizer DB Dekomposition.

Zusätzlich besitzt die Klasse Methoden für das Speichern und Laden ihrer Daten, für das Erstellen des kd-Trees sowie der Aktualisierung der Bounding Sphere und des Intervalls. Da alle diese Methoden von Klienten nur auf dem Wurzelknoten aufgerufen werden sollen, sind sie auf allgemeiner Knotenebene *protected*.

```
class VertexBufferBase
{
public:
    virtual void render( VertexBufferState& state ) const;

    const BoundingSphere& getBoundingSphere() const;
    const float* getRange() const;

    virtual const VertexBufferBase* getLeft() const;
    virtual const VertexBufferBase* getRight() const;

protected:
    VertexBufferBase();
    virtual ~VertexBufferBase();

    virtual void toStream( std::ostream& os );
    virtual void fromMemory( char** addr,
                             VertexBufferData& globalData );

    virtual void setupTree( VertexData& data, const Index start,
                            const Index length, const Axis axis,
                            const size_t depth,
                            VertexBufferData& globalData );
    virtual BoundingBox updateBoundingSphere();
    virtual void updateRange();
```

```
    void calculateBoundingSphere( const BoundingBox& bBox );

    BoundingSphere   _boundingSphere;
    Range            _range;

private:
};
```

mesh::VertexBufferNode ist die Klasse für reguläre Knoten die kein Blatt sind. Sie ist abgeleitet von *mesh::VertexBufferBase* und selbst Basisklasse für *mesh::VertexBufferRoot*. Die Klasse besitzt mit Ausnahme einer Hilfsfunktion keine neuen Methoden, implementiert aber einerseits die abstrakten Methoden von *mesh::VertexBufferNode* und überschreibt andererseits sofern notwendig bestehende virtuelle Methoden.

```
class VertexBufferNode : public VertexBufferBase
{
    ...

private:
    size_t countUniqueVertices( VertexData& data, const Index start,
                                const Index length ) const;

    VertexBufferBase*    _left;
    VertexBufferBase*    _right;
};
```

mesh::VertexBufferRoot ist die Klasse für den Wurzelknoten des Baumes, und abgeleitet von *mesh::VertexBufferNode*. Sie verwaltet die Hauptkopie der polygonalen Daten, und stellt den öffentlichen Zugang für Klienten dar. Entsprechend zu den *protected* Methoden der Basisklassen bietet sie zugehörige *public* Methoden an.

Das Rendering der Datenstruktur ist entweder durch einen einzelnen Aufruf der entsprechenden Methode auf dem Wurzelknoten möglich, oder durch mehrere direkte Aufrufe der *render()* Methoden der regulären bzw. Blattknoten, wie es zum Beispiel bei Verwendung des externen View Frustum Culling geschieht. In beiden Fällen ist vor und nach dem Rendering auf der Wurzel *begin-* und *endRendering()* aufzurufen. Diese beiden Methoden kümmern sich um den Auf- und Abbau des für das Rendering notwendigen OpenGL States.

```
class VertexBufferRoot : public VertexBufferNode
{
public:
    virtual void render( VertexBufferState& state ) const;

    void beginRendering( VertexBufferState& state ) const;
    void endRendering( VertexBufferState& state ) const;
```

```
    void setupTree( VertexData& data );
    bool writeToFile( const char* filename );
    bool readFromFile( const char* filename );
    bool hasColors() const;

protected:
    virtual void toStream( std::ostream& os );
    virtual void fromMemory( char* start );

private:
    bool constructFromPly( const char* filename );

    VertexBufferData _data;
};
```

Die letzte Mesh Klasse ist *mesh::VertexBufferLeaf*, bei welcher es sich um die Klasse für Blattknoten handelt. Sie hält Referenzen auf die globalen polygonalen Daten in der Wurzel, sowie Start und Länge der Bereiche innerhalb der globalen Daten für welche sie zuständig ist. Während die *render()* Methoden der übrigen Knoten nicht mehr machen als an die Kinder zu delegieren, sind die Blätter für das tatsächliche Rendering zuständig.

```
class VertexBufferLeaf : public VertexBufferBase
{
    ...

private:
    void setupRendering( VertexBufferState& state ) const;
    void renderImmediate( VertexBufferState& state ) const;
    void renderDisplayList( VertexBufferState& state ) const;
    void renderBufferObject( VertexBufferState& state ) const;

    VertexBufferData&    _globalData;
    Index                _vertexStart;
    ShortIndex           _vertexLength;
    Index                _indexStart;
    Index                _indexLength;
    mutable bool         _isSetup;
};
```

3.2.3. Implementierung

Nachdem das Design der neuen Datenstruktur im Detail erklärt wurde, soll nun auf einige ausgewählte Details der Implementierung eingegangen werden. Für die vollständigen Details der Implementierung sei auf den Source Code auf der beiliegenden CD-ROM verwiesen.

Die neue Datenstruktur wurde zwar in erster Hinsicht für die Verwendung in der Equalizer Beispielapplikation entwickelt, sollte aber dennoch unabhängig von Equalizer selbst bleiben um unter Umständen auch in anderen Projekten einsetzbar zu sein. Da für beide Einsatzzwecke an einigen Stellen unterschiedlicher Code notwendig ist, wurde eine Lösung mit Makros und bedingt compiliertem Code verwendet. Die Datei *typedefs.h* bietet einerseits die Einstellung ob Equalizer verwendet werden soll (#**define** EQUALIZER), und andererseits die Definitionen welche Streams für Logausgaben (#**define** MESHINFO etc.) und welche Form von Assertions (#**define** MESHASSERT) genutzt werden sollen. In der Datei wird auch ein NullStream definiert, der bei stand-alone Benutzung verwendet werden kann um gewisse Logausgaben komplett zu verwerfen.

Generell ist zur Implementierung zu sagen, dass intern verwendete Methoden im Fehlerfall Ausnahmen vom Typ *mesh::MeshException* werfen, während die vom Klienten aufrufbaren Methoden ihren Erfolg über die Rückgabe von *boolean* Werten signalisieren. Besonders unter Windows verlangsamt der Einsatz von Debugging Code verschiedene Bereiche des Baumaufbaus so stark, dass selbst mittelgrosse PLY Modelle nicht mehr in nützlicher Frist verarbeitet werden.

VertexData

mesh::VertexData verwendet für das Einlesen der PLY Dateien die selbe Bibliothek wie die alte Beispielapplikation. Der Code selbst wurde aber ganz neu geschrieben, und bemüht sich nun Speicherlecks zu vermeiden. In Bezug auf die Sortierfunktion für Triangles ist anzumerken, dass diese nach willkürlich festgelegter Art vorgeht (siehe Code). Ein anderes Sortierverfahren würde den Zweck ebenfalls erfüllen, es muss nur eine gleichbleibende Ordnung über die Triangles definiert werden.

VertexBufferNode

Aus *mesh::VertexBufferNode* soll kurz betrachtet werden, wie die Umsetzung des kd-Tree Algorithmus in der Praxis nun tatsächlich aussieht. *countUniqueVertices()* ist eine Hilfsfunktion welche die Anzahl einzigartiger Vertices innerhalb des betrachteten Bereiches ermittelt. Das Reindexieren der Indices und der Aufbau der sortierten polygonalen

Daten findet in der *setupTree()* Methode von *mesh::VertexBufferLeaf* statt. Die Konstrukte mit dem *static_cast* die hier und an einigen anderen Stellen verwendet werden, sind notwendig um die *protected* Eigenschaft der verwendeten Methoden zu umgehen.

```
void VertexBufferNode::setupTree( VertexData& data, const Index start,
                                  const Index length, const Axis axis,
                                  const size_t depth,
                                  VertexBufferData& globalData )
{
    data.sort( start, length, axis );
    const Index median = start + ( length / 2 );

    // left child will include elements smaller than the median
    const Index leftLength    = length / 2;
    const bool  subdivideLeft =
        countUniqueVertices( data, start, leftLength ) > LEAF_SIZE ||
        depth < 3;

    if( subdivideLeft )
        _left = new VertexBufferNode;
    else
        _left = new VertexBufferLeaf( globalData );

    // right child will include elements equal or greater median
    const Index rightLength    = ( length + 1 ) / 2;
    const bool  subdivideRight =
        countUniqueVertices( data, median, rightLength ) > LEAF_SIZE ||
        depth < 3;

    if( subdivideRight )
        _right = new VertexBufferNode;
    else
        _right = new VertexBufferLeaf( globalData );

    // move to next axis and continue contruction in the child nodes
    const Axis newAxis = static_cast< Axis >( ( axis + 1 ) % 3 );
    static_cast< VertexBufferNode* >
        ( _left )->setupTree( data, start, leftLength, newAxis,
                              depth + 1, globalData );
    static_cast< VertexBufferNode* >
        ( _right )->setupTree( data, median, rightLength, newAxis,
                               depth + 1, globalData );
}
```

VertexBufferLeaf

Die entsprechende *setupTree()* Methode in *mesh::VertexBufferLeaf* sieht wie folgt aus. Sie sortiert die Daten ein letztes Mal nach der neuen Achse, und reindexiert die Vertices dann damit alle Indices innerhalb des Short Bereichs liegen. Während dem Reindexieren werden die Daten ebenfalls sortiert in die globale *mesh::VertexBufferData* Struktur eingefügt. Werden vereinzelte Vertices auch über Blattgrenzen hinweg noch gemeinsam genutzt, werden die entsprechenden Daten durch diesen Schritt dupliziert.

```
void VertexBufferLeaf::setupTree( VertexData& data, const Index start,
                                  const Index length, const Axis axis,
                                  const size_t depth,
                                  VertexBufferData& globalData )
{
    data.sort( start, length, axis );
    _vertexStart = globalData.vertices.size();
    _vertexLength = 0;
    _indexStart = globalData.indices.size();
    _indexLength = 0;

    const bool hasColors = ( data.colors.size() > 0 );

    // stores the new indices (relative to _start)
    map< Index, ShortIndex > newIndex;

    for( Index t = 0; t < length; ++t )
    {
        for( Index v = 0; v < 3; ++v )
        {
            Index i = data.triangles[start + t][v];
            if( newIndex.find( i ) == newIndex.end() )
            {
                newIndex[i] = _vertexLength++;
                // assert number does not exceed SmallIndex range
                MESHASSERT( _vertexLength );
                globalData.vertices.push_back( data.vertices[i] );
                if( hasColors )
                    globalData.colors.push_back( data.colors[i] );
                globalData.normals.push_back( data.normals[i] );
            }
            globalData.indices.push_back( newIndex[i] );
            ++_indexLength;
        }
    }
}
```

Die für das Rendering relevanten Methoden von *mesh::VertexBufferLeaf* werden in Kapitel 4 genauer betrachtet, und entsprechend hier nicht weiter behandelt.

VertexBufferRoot

Der letzte fehlende Bestandteil des Baumaufbaus ist die Methode die der Client in *mesh::VertexBufferRoot* aufruft um den Vorgang zu starten. Sie ist nicht viel mehr als ein öffentlicher Wrapper für die *protected* Methode. Nach dem eigentlichen Baumaufbau werden ebenfalls rekursiv die Bounding Spheres sowie die Intervalle für die Equalizer DB Dekomposition berechnet. Anzumerken ist hierbei, dass die Methode *updateBoundingSphere()* temporär mit Bounding Boxes arbeitet und an den Aufrufer zurückliefert, weil das Verschmelzen der Bounding Volumes damit einfacher ist als bei der direkten Verwendung von Bounding Spheres.

```
void VertexBufferRoot::setupTree( VertexData& data )
{
    // data is VertexData, _data is VertexBufferData
    _data.clear();
    VertexBufferNode::setupTree( data, 0, data.triangles.size(),
                                 AXIS_X, 0, _data );
    VertexBufferNode::updateBoundingSphere();
    VertexBufferNode::updateRange();
}
```

Es wurden bereits plattformspezifische Eigenheiten erwähnt, die es bei der Implementierung ebenfalls zu berücksichtigen gilt. *mesh::VertexBufferRoot* verwendet in der Hinsicht beispielsweise zwei Hilfsfunktionen, um einerseits zu testen ob es sich um ein 32 oder 64 Bit System handelt, und andererseits ob das System die Daten im Little- oder Big-Endian Format verwaltet. Für jede mögliche Kombination werden eigenständige binäre Repräsentationen des kd-Trees abgespeichert.

```
size_t getArchitectureBits()
{
    return ( sizeof( void* ) * 8 );
}

bool isArchitectureLittleEndian()
{
    unsigned char test[2] = { 1, 0 };
    short x = *( reinterpret_cast< short* >( test ) );
    return ( x == 1 );
}
```

Mit diesen ausgewählten Betrachtungen einzelner Implementierungssdetails ist das Ende der ersten Teilaufgabe der vorliegenden Arbeit erreicht.

4. Rendering

In diesem Kapitel wird die zweite Teilaufgabe der Arbeit behandelt, das Rendering. In der ersten Sektion werden die drei OpenGL Rendermodi beschrieben welche die neue Applikation grundsätzlich unterstützt, sowie die konkrete Implementierung der entsprechenden Rendermodi in Bezug auf die Applikation. In der zweiten Sektion werden anschliessend die möglichen Shading Algorithmen vorgestellt, und zu deren besseren Verständnis kurz das Beleuchtungsmodell rekapituliert. Abgeschlossen wird das Kapitel mit einem Ausblick auf weitere Algorithmen die das Rendering optisch attraktiver gestalten können.

4.1. OpenGL Rendermodi

Die alte Beispielapplikation hat für das Rendering ausschliesslich Display Listen verwendet. Die neue Applikation wird nun um die Möglichkeit erweitert, für das Rendering Vertex Buffer Objects [15] zu verwenden.

4.1.1. Immediate Mode

Immediate Mode ist die einfachste Möglichkeit im Umgang mit OpenGL. Kommandos werden einzeln abgesetzt und direkt ausgeführt. Dies bedeutet für den konkreten Fall des Renderings unserer polygonalen Daten, dass für jedes zu zeichnende Triangle je drei Aufrufe für das Setzen von Vertexpositionen, Normalen und Farbinformationen notwendig sind. Im folgenden der entsprechende Auszug aus *mesh::VertexBufferLeaf*.

```
inline
void VertexBufferLeaf::renderImmediate( VertexBufferState& state )
{
    glBegin( GL_TRIANGLES );
    for( Index offset = 0; offset < _indexLength; ++offset )
    {
        Index i = _vertexStart +
                  _globalData.indices[_indexStart + offset];
```

```
        if( state.useColors() )
            glColor4ubv( &_globalData.colors[i][0] );
        glNormal3fv( &_globalData.normals[i][0] );
        glVertex3fv( &_globalData.vertices[i][0] );
    }
    glEnd();
}
```

Immediate Mode hat aber gravierende Nachteile in Hinblick auf die Performance. Der Overhead für die häufigen Funktionsaufrufe und das einzelne Durchreichen der Daten ist signifikant. Immediate Mode taugt deshalb in anspruchsvollen Anwendungen heute praktisch nur noch zu Testzwecken. Die entsprechende Methode ist in *mesh::VertexBufferLeaf* nur für Vergleichszwecke vorhanden, und weil der selbe Code auch für die Nutzung von Display Listen Verwendung findet.

4.1.2. Display Lists

Display Listen bilden den ersten Ausweg zur signifikanten Leistungssteigerung. Neue Display Listen werden mit glGenLists() angefordert, und lassen sich einfach erstellen indem vorhandener Immediate Mode Code mit den Befehlen glNewList() und glEndList() umschlossen wird. Sämtliche Kommandos die zwischen diesen Zeilen abgesetzt werden, kompiliert OpenGL zu einer Display Liste. Aufgerufen wird eine Display Liste durch einen einzelnen Befehl, glCallList().

In unserer Applikation lässt sich auf Basis des oben bereits dargestellten Immediate Mode Codes ohne Probleme eine neue Display Liste anlegen. Dies wird einmalig in der Methode *vertexBufferLeaf::setupRendering()* erledigt, danach steht die Display Liste für weitere Rendervorgänge zur Verfügung.

```
glNewList( displayList, GL_COMPILE );
renderImmediate( state );
glEndList();
```

Das Rendering in *mesh::VertexBufferLeaf* reduziert sich damit auf:

```
inline
void VertexBufferLeaf::renderDisplayList( VertexBufferState& state )
{
    GLuint displayList = state.getDisplayList( this );
    glCallList( displayList );
}
```

Aktuelle OpenGL Treiber optimieren die Kommandos beim Kompilieren einer Display

Liste stark, und speichern die Display Listen direkt auf der Grafikhardware. Das Ergebnis ist in vielen Fällen eine kaum zu verbessernde Renderleistung, solange die Display Listen nicht signifikant zu gross oder klein sind.

Display Listen haben aber auch Nachteile, auch wenn diese für diese Applikation nicht ins Gewicht fallen. Display Listen können keine Befehle enthalten, welche den OpenGL Client State verändern. Findet sich solch ein Befehl innerhalb eines Display Listen Blocks, wird er direkt ausgeführt, aber nicht mit in die Liste kompiliert. Weiterhin taugen Display Listen nur für statische Objekte, da eine kompilierte Display Liste nachträglich nicht mehr verändert werden kann.

4.1.3. Vertex Buffer Objects

Vertex Buffer Objects, kurz VBOs [15], sind ursprünglich eine Extension von OpenGL und ab Version 1.5 fester Bestandteil der Spezifikation. Kurz formuliert erlauben es VBOs, diverse Daten wie Vertexpositionen, Normalen und Farben nicht nur blockweise mit einem Aufruf zu spezifizieren, sondern diese werden im Idealfall auch direkt auf der Grafikhardware gespeichert. VBOs taugen dabei nicht nur für statische Daten, sondern können auch für Daten verwendet werden die sich dynamisch ändern. Damit der Treiber die Zugriffe optimieren kann, wird beim Erstellen eines VBO angegeben um welche Verwendung es sich handelt.

Obwohl praktisch alle aktuellen Grafiktreiber OpenGL in Versionen jenseits von 1.5 unterstützen, ist die Methode die VBO Funktionen anzusprechen nicht immer die gleiche. Unter Mac OS X sind die Funktionen beispielsweise direkt verfügbar. Unter Windows hingegen stellt Microsoft nur veraltete OpenGL Bibliotheken zur Verfügung und die Funktionen können nur über Funktionszeiger angesprochen werden, die vor der ersten Verwendung auch noch ermittelt werden müssen. Unter Linux hat sich während der Entwicklung dieser Applikation ein gemischtes Bild gezeigt. Während auch hier Funktionszeiger ermittelt werden müssen, sind die Funktionen in einigen Fällen unter den standard OpenGL Namen (z.B. glBindBuffer()) anzutreffen, während in anderen Fällen nur das Extension Namensschema (z.B. glBindBufferARB()) verwendet werden kann.

Um diese Problematik vom Code der die VBOs nutzen möchte in der vorliegenden Applikation zu entbinden, wurde die Initialisierung der Funktionszeiger und deren zentrale Speicherung in eine Klasse *GLFunctions* ausgelagert, die von der Applikation über den *VertexBufferState* angesprochen werden kann.

VBOs müssen vor der Verwendung ähnlich zu Display Listen erst initialisiert werden. Eine neues VBO wird zunächst über glGenBuffers() angefordert. Danach wird ein VBO mit glBindBuffer() aktiviert und via glBufferData() mit Daten gefüllt. Es gibt dabei zwei Arten von Buffer Objects. Die erste Art speichert Vertexdaten wie Positionen, Normalen

und Farben, die zweite speichert Indexdaten. Im letzteren Fall spricht man dann auch von Element Buffer Objects, kurz EBOs. Es können entweder einzelne VBOs für die verschiedenen Typen von Vertexdaten angelegt werden, oder die Daten verwoben in einem einzigen VBO gespeichert werden. Siehe dazu auch [16].

Für das Rendering der Daten im kd-Tree wurde der erste Ansatz gewählt. Einmalig werden die VBOs in *mesh::VertexBufferLeaf::setupRendering()* initialisiert, und stehen fortan für das Rendering zur Verfügung.

```
GLuint buffers[4];

buffers[VERTEX_OBJECT] = state.newBufferObject( ... );
gl->bindBuffer( GL_ARRAY_BUFFER, buffers[VERTEX_OBJECT] );
gl->bufferData( GL_ARRAY_BUFFER, _vertexLength * sizeof( Vertex ),
                &_globalData.vertices[_vertexStart], GL_STATIC_DRAW );

...

buffers[INDEX_OBJECT] = state.newBufferObject( ... );
gl->bindBuffer( GL_ELEMENT_ARRAY_BUFFER, buffers[INDEX_OBJECT] );
gl->bufferData( GL_ELEMENT_ARRAY_BUFFER,
                _indexLength * sizeof( ShortIndex ),
                &_globalData.indices[_indexStart], GL_STATIC_DRAW );
```

Um die in den VBOs und dem EBO vorbereiteten Daten zu rendern, ist zunächst jedes Buffer Object wieder mit glBindBuffer() zu aktivieren. Für jedes VBO ist danach mit glXYZPointer() das Format der Daten anzugeben. Nach dem Aktivieren des EBO können die Daten schliesslich mit einem Aufruf von glDrawElements() gezeichnet werden.

```
inline
void VertexBufferLeaf::renderBufferObject( VertexBufferState& state )
{
    const GLFunctions* gl = state.getGLFunctions();
    GLuint buffers[4];
    for( int i = 0; i < 4; ++i )
        buffers[i] = state.getBufferObject( ... );

    if( state.useColors() )
    {
        gl->bindBuffer( GL_ARRAY_BUFFER, buffers[COLOR_OBJECT] );
        glColorPointer( 4, GL_UNSIGNED_BYTE, 0, 0 );
    }
    gl->bindBuffer( GL_ARRAY_BUFFER, buffers[NORMAL_OBJECT] );
    glNormalPointer( GL_FLOAT, 0, 0 );
    gl->bindBuffer( GL_ARRAY_BUFFER, buffers[VERTEX_OBJECT] );
    glVertexPointer( 3, GL_FLOAT, 0, 0 );
    gl->bindBuffer( GL_ELEMENT_ARRAY_BUFFER, buffers[INDEX_OBJECT] );
```

```
    glDrawElements( GL_TRIANGLES, _indexLength, GL_UNSIGNED_SHORT, 0 );
}
```

Was ebenfalls noch beachtet werden muss ist, dass vor dem Rendering der VBOs der Vertex Array State vom Client aktiviert werden muss. Dies erledigt in der Applikation die Methode *mesh::VertexBufferRoot::beginRendering()*.

Wie schon beim Vergleich des Rendering Code ersichtlich wird, benötigen VBOs bedeutend mehr Funktionsaufrufe als Display Listen. Entsprechend gibt es für die Grösse von VBOs eine Untergrenze bei deren Unterschreitung die Leistung einbricht. Ebenfalls zu beachten ist dass die Leistung von VBO Rendering stark zusammenfällt, wenn die falschen Datentypen verwendet werden. Der Hintergrund ist dabei, dass die Grafikhardware in diesem Fall bei jedem Rendering Konvertierungen ins korrekte, intern verwendete Format vornehmen muss. Für weitere Benchmarks und Hinweise zur Verwendung von VBOs, siehe auch [2].

Obwohl bei der Entwicklung der vorliegenden Applikation alle Hinweise in Bezug auf Datenformate und Puffergrössen beachtet wurden, bleibt die Leistung des VBO Rendering in einigen Fällen aus bisher unbekannten Gründen dennoch hinter den Display Listen zurück.

4.2. Shading Algorithmen

Nachdem die grundlegenden Rendermöglichkeiten in der letzten Sektion gezeigt wurden, sollen in dieser Sektion nun die verschiedenen Shading Algorithmen sowohl in der Theorie erklärt als auch in der praktischen Implementierung vorgestellt werden. Als Hintergrund wird dazu zunächst das Beleuchtungsmodell diskutiert, welches in den meisten Echtzeit Computergrafik Systemen verwendet wird. Eine ausführliche Behandlung dieser Themen findet sich beispielsweise in [1].

4.2.1. Beleuchtungsmodell

Das heutzutage in der (Echtzeit) Computergrafik am häufigsten anzutreffende Beleuchtungsmodell stammt von Phong und wurde bereits vor mehr als 30 Jahren publiziert. Es bildet die Realität nicht exakt ab, erreicht aber in vielen Situationen akzeptable bis gute Ergebnisse.

Das Phong Modell ist ein lokales Beleuchtungsmodell, es berücksichtigt nur Reflektionen die ein Objekt selbst durch direkte Bestrahlung einer Lichtquelle erzeugt. Lichtreflektio-

nen die von anderen Objekten stammen werden nicht in Betracht gezogen. Um diesen Verlust auszugleichen, enthält das Modell eine Komponente für globales *Ambient Light*. Das reflektierte Licht besteht aus zwei Komponten, dem *Diffuse Light* sowie dem *Specular Light*).

$$I_{total} = I_{ambient} + I_{diffuse} + I_{specular}$$

Unterschiedliche Materialien reflektieren Licht auf unterschiedliche Weise. Um dies zu modellieren, werden jedem Objekt drei Materialkoeffizienten zugewiesen, welche den drei Lichtkomponenten entsprechen und bestimmen welches Licht vom Objekt wie stark reflektiert wird. Neben diesen drei Koeffizienten wird auch noch ein vierter verwendet, der beeinflusst wie stark die Objektoberfläche spiegelt und *Shininess* genannt wird.

Ambient Light

Das Ambient Light ist global, beleuchtet also jeden Punkt eines jeden Objektes mit der gleichen Intensität. Wenn die Intensität des Ambient Light L_a ist und der entsprechende Materialkoeffizient k_a, so beträgt die reflektierte Intensität

$$I_a = k_a L_a.$$

Diffuse Light

Diffuse Light wird von einer rauen Oberfläche in alle Richtungen gleichmässig reflektiert, und erscheint somit für jeden Beobachter gleich. Allerdings ist die reflektierte Intensität abhängig von der Lage der Oberfläche und des Lichtes zueinander. Im Phong Modell wird diese Komponente daher mit Hilfe des Winkels zwischen dem Lichtvektor l und der Flächennormalen n modelliert. Da der Cosinus des Winkels bei Einheitslänge der Vektoren dem Skalarprodukt entspricht, ergibt sich für die Intensität somit

$$I_d = k_d L_d (l \cdot n).$$

Specular Light

Spiegelnde Reflektionen stammen von glatten Oberflächen. Das Licht fällt aus Richtung des Lichtvektors l ein, und wird in Richtung des Reflektionsvektors r zurückgeworfen. Der davon sichtbare Anteil hängt vom Winkel zwischen dem (perfekten) Reflektionsvektor und dem Beobachter ab. Nicht jede Oberfläche reflektiert allerdings perfekt, und die daraus resultierende Streuung der tatsächlichen Reflektion und deren Einfluss auf die

Intensität wird durch den *Shininess* Koeffizienten α modelliert. Mit v als Einheitsvektor in Richtung des Beobachters, beträgt die Intensität also

$$I_s = k_s L_s (r \cdot v)^\alpha.$$

Vollständige Formel

Wenn die drei verschiedenen Komponenten nun zusammengesetzt werden, und möglicherweise negative Komponenten ausgeschlossen werden, ergibt sich die folgende vollständige Formel für das Phong Beleuchtungsmodell (unter Vernachlässigung dass Intensität mit zunehmender Entfernung zwischen der Lichtquelle und dem Objekt abfällt):

$$I = k_a L_a + k_d L_d max(l \cdot n, 0) + k_s L_s max((r \cdot v)^\alpha, 0)$$

Um die Berechnung von r zu vermeiden, hat Blinn eine Modifikation des Phong Modells vorgeschlagen. Er verwendet den sogenannten *Halfway* Vektor $h = \frac{l+v}{|l+v|}$ und approximiert dann $r \cdot v$ in der Formal mit $n \cdot h$. Um ein optisch ähnliches Ergebnis wie beim reinen Phong Modell zu erhalten, muss nach dieser Approximation der *Shininess* Koeffizient α angepasst werden.

4.2.2. Flat Shading

Theoretisch unterscheiden sich sowohl der Lichtvektor l als auch die Richtung zum Beobachter v für jeden Punkt einer Oberfläche. Um die Shading Berechnung zu vereinfachen, können sowohl die Lichtquelle als auch der Beobachter als unendlich weit entfernt betrachtet werden, womit aus beiden Vektoren konstante Richtungen werden. Für plane Oberflächen ist die Flächennormale n ebenfalls an jedem Punkt konstant.

Wenn alle drei Vektoren konstant sind, muss das Shading der Oberfläche nur an einem Punkt berechnet werden. Das Ergebnis sind Flächen die über den ganzen Bereich dieselbe Farbtönung tragen. Dieser Algorithmus wird als *Flat Shading* bezeichnet.

Weil die alte Datenstruktur nur Flächennormalen gespeichert hat, war Flat Shading der einzige Algorithmus der mit der alten Beispielapplikation realisiert werden konnte. Da dies optisch nicht ansprechend und auch nicht mehr zeitgemäss ist, wird in der neuen Applikation auf Flat Shading ganz verzichtet.

4.2.3. Gouraud Shading

Eür ein korrektes Shading gekrümmter Oberflächen wäre es notwendig, die betroffenen Vektoren an jedem einzelnen Punkt zu kennen. Eine brauchbare Approximation ist es, das Shading nur an Vertices zu berechnen und dann über die Oberfläche hinweg zu interpolieren. Dieses Verfahren wird *Gouraud Shading* genannt.

Für Gourauf Shading wird je eine Normale pro Vertex benötigt. Diese lässt sich durch Mittelung der Flächennormalen sämtlicher angrenzender Flächen vernünftig schätzen. Die neue Applikation unterstützt Gouraud Shading direkt, da unmittelbar nach Einlesen des Modells die entsprechenden Vertexnormalen berechnet werden.

4.2.4. Phong Shading

Gouraud Shading hat immer noch einen entscheidenen Nachteil. Weil das Shading nur an den Vertices berechnet und über die Öberfläche interpoliert wird, gehen spiegelnde Reflektionen inmitten einer Fläche verloren oder werden über die gesamte Fläche verschmiert.

Eine weitere Verbesserung ist, anstelle des Shading die Normalen selbst zu interpolieren. Dieser Algorithmus ist das *Phong Shading*. Das Problem mit Phong Shading ist, dass Shading nicht mehr pro Vertex berechnet werden kann, sondern per Pixel berechnet werden muss. Dies war bis zur Einführung von programmierbaren Fragment Shadern in der Grafikhardware nicht in Echtzeit möglich.

Die neue Applikation enthält ein Gerüst für den Einsatz von Vertex und Fragment Shadern, die mit der OpenGL Shading Language geschrieben wurden. Als einfache Beispiele werden Shader mitgeliefert, welche das Phong Shading implementieren.

4.3. Weitere Algorithmen

Aufgrund fehlender Zeit konnten leider keine weiteren Algorithmen recherchiert und umgesetzt werden, welche die optische Attraktivität der Darstellung weiter erhöhen würden. Da aber ein Gerüst zur Unterstützung von GLSL Shadern vorhanden ist, sollten sich weitere Algorithmen zu einem späteren Zeitpunkt ohne grosse Probleme zur Applikation hinzufügen lassen.

5. Zusammenfassung

Diese Semesterarbeit hatte zwei Ziele. Zum ersten die Umsetzung einer neuen Datenstruktur für die effiziente Verwaltung von polygonalen Daten, und zum zweiten ein ebenfalls effizientes aber auch optisch ansprechendes Rendering dieser Daten.

Der erste Teil der Arbeit wurde vollständig umgesetzt, die Implementierung der neuen kd-Tree Datenstruktur erfüllt sämtliche in Abschnitt 3.2.1 gestellten Anforderungen. Der einzige spürbare Nachteil im Vergleich zur alten Datenstruktur ist ein bedeutend langsamerer Aufbau des Baumes. Da nach der erstmaligen Erstellung aber sowieso binäre Repräsentationen gespeichert werden und für weitere Aufrufe verfügbar sind, ist dieser Punkt vernachlässigbar. Der Speicherverbrauch wurde auf weniger als ein Viertel reduziert, wie die folgende Tabelle 5.1 anhand der Grösse binärer 32 Bit Repräsentationen aufzeigt. Es findet keine Kompression statt. Die für die Messungen verwendeten Modelle stammen aus [12].

Modell	**Binary Octree [MB]**	**Binary kd-Tree [MB]**	**Verhältnis [%]**
Armadillo	27.72	6.09	21.96
Dragon	69.83	15.34	21.97
Happy Buddha	87.16	19.05	21.86
Lucy	n/a	494.58	n/a
Stanford Bunny	5.56	1.20	21.56

Tabelle 5.1.: Unterschied Speicherbedarf der Datenstrukturen

Momentan wird der kd-Tree von jedem Renderknoten vollständig geladen. Eine denkbare Erweitung für die Verwendung mit der Equalizer DB Dekomposition wäre beispielsweise, ein teilweises Laden nur der benötigen Bereiche zu ermöglichen. Die Effizienz des View Frustum Cullings liesse sich unter Umständen verbessern, indem statt beim Aufbau die Sortierachse zu rotieren jeweils diejenige mit der längsten Dimension gewählt würde.

Des weiteren steht die Ausgliederung der Mesh Klassen in eine eigenstände Biobliothek aus, um diese auch für andere Projekte verwenden zu können. Als Folge daraus wäre auch eine GLUT Applikation denkbar, welche die Benutzung der Mesh Klassen in einer stand-alone Umgebung demonstriert.

Der zweite Teil der Arbeit erreicht die definierten Ziele teilweise. Das neue Rendering mit Vertex Buffers Objects ist implementiert und funktioniert. Auch Rendering mit Display Listen wird als OpenGL 1.1 kompatible Lösung weiterhin angeboten, und bleibt zur Zeit noch die Standardeinstellung. VBO Verwendung kann über einen Kommandozeilenparameter aktiviert werden.

Ein Vergleich der Renderingperformance zwischen der alten und der neuen Applikation findet sich in folgender Tabelle 5.2. In einigen Fällen bleibt die VBO Performance aus bisher ungeklärten Gründen hinter den Erwartungen zurück, während sie in anderen Fällen einen markanten Leistungsvorteil zeigt. Neben den Modellen aus [12] wurden hierfür auch Modelle von [13] verwendet.

Modell	**Anzahl Triangles**	**eqPly (alt) DL [FPS]**	**eqPly (neu) DL [FPS]**	**eqPly (neu) VBO [FPS]**
Stanford Bunny	69'451	406.82	436.67	617.00
Rocker Arm	80'354	445.61	445.57	301.67
Goddess 1	274'822	142.20	162.45	70.94
Armadillo	345'944	118.30	136.43	92.17
Boat	776'374	29.25	20.06	28.47
Dragon	871'414	17.06	45.02	38.19
Goddess 2	1'047'330	10.02	9.32	42.99
Hip	1'060'346	11.10	9.24	29.37
Happy Buddha	1'087'716	9.54	29.27	34.88

Tabelle 5.2.: Vergleich Renderingperformance altes und neues eqPly

Um den möglichen Performance Unterschieden des VBO Renderings auf die Spur zu kommen, wurde noch die erzeugte Anzahl Blattknoten sowie die durchschnittliche Anzahl Vertices und Triangles pro Blatt ermittelt. Die Ergebnisse in Tabelle 5.3 erlauben aber auch keine gesicherten Rückschlüsse.

Modell	**Anzahl Triangles**	**Anzahl Blätter**	**Triangles pro Blatt**	**Vertices pro Blatt**
Stanford Bunny	69'451	16	4'340	2'329
Rocker Arm	80'354	16	5'022	2'797
Goddess 1	274'822	16	17'176	9'252
Armadillo	345'944	16	21'621	11'218
Boat	776'374	32	24'261	12'769
Dragon	871'414	32	27'231	14'138
Goddess 2	1'047'330	32	32'729	17'430
Hip	1'060'346	32	33'135	17'307
Happy Buddha	1'087'716	32	33'991	17'509

Tabelle 5.3.: Statistische Daten über die Ausnutzung der Blätter im kd-Tree

Das Rendering ist durch die standardmässige Nutzung von Gouraud anstelle von Flat Shading bereits optisch deutlich ansprechender als in der alten Applikation. Dies ist in Abbildungen 5.1 und 5.2 am Beispiel des RockerArm Modells zu sehen.

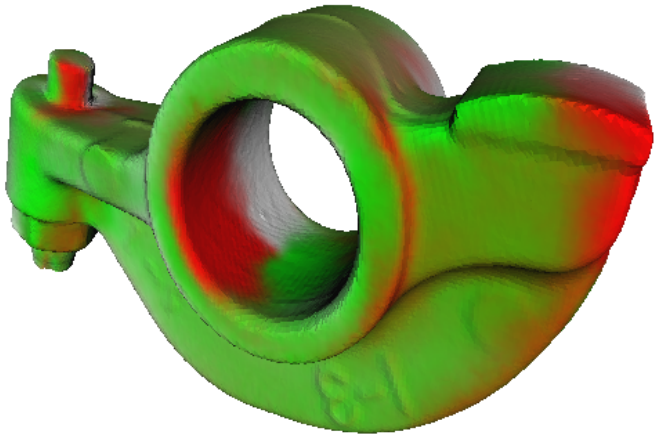

Abbildung 5.1.: RockerArm Modell mit Flat Shading

Der Einsatz von GLSL Shadern und Phong Shading verbessert dies je nach Licht- und Materialeigenschaften noch einmal. Weitere Algorithmen konnten aufgrund mangelnder Zeit leider nicht mehr realisiert werden, das vorhandene Gerüst sollte das Hinzufügen weiterer Shader Algorithmen zu einem späteren Zeitpunkt jedoch problemlos ermöglichen.

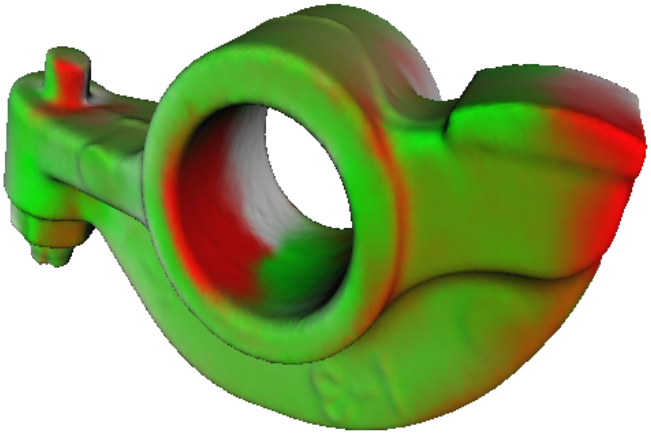

Abbildung 5.2.: RockerArm Modell mit Gouraud Shading

A. Inhalt der CD-ROM

Die der Semesterarbeit beiliegende CD-ROM besitzt den folgenden Inhalt:

Pfad	**Beschreibung**
zusfg.pdf	Zusammenfassung der Arbeit in Deutsch
abstract.pdf	Zusammenfassung der Arbeit in Englisch
semester.pdf	Elektronische Version der Arbeit
latex	LaTeX Quellen der Arbeit
eq	Equalizer SVN Snapshot per 30. September 2007

Tabelle A.1.: Inhalt der CD-ROM

Obwohl die CD-ROM einen kompletten Equalizer SVN Snapshot enthält, wird dringend empfohlen bei Bedarf eine aktuelle Version aus dem Repository auszuchecken [7]. Sowohl Equalizer als auch eqPly befinden sich noch in aktiver Entwicklung.

Der Source Code der ursprünglichen Beispielapplikation befindet sich im Verzeichnis *eq/src/examples/eqPly/*. Das Ergebnis dieser Arbeit, der Source Code der neuen Applikation, liegt in *eq/src/examples/eqPlyNew/*.

Um die Applikationen auszuführen ist ein vollständiges Kompilieren von Equalizer notwendig. Unter Mac OS X und Linux reicht es aus, dazu in das *eq/src* Verzeichnis zu wechseln und *make* auszuführen. Für Windows enthält *eq/src/VS2005* eine Visual Studio 2005 Solution. Die notwendigen Voraussetzungen und Hinweise für das Kompilieren unter den verschiedenen Betriebssystemen finden sich in den *README* Dateien sowie der *FAQ* im Verzeichnis *eq/src/*.

Glossar

Context	OpenGL Kontext.
Drawable	OpenGL Zeichenbereich.
Extensions	Erweiterungen von OpenGL, welche den Einzug in den OpenGL Hauptstandard noch nicht geschafft haben.
Faces	Flächen aus denen sich ein Objekt zusammensetzt.
Frame	Einzelbild einer Bildfolge.
Head Tracker	Gerät zur Verfolgung der Kopfbewegung mit Hilfe externer Referenzpunkte.
kd-Tree	K-dimensionaler Baum, eine Datenstruktur aus der Informatik zur räumlichen Aufteilung.
Light, Ambient	Umgebungslicht.
Light, Diffuse	Von rauen Oberflächen gleichmässig in alle Richtungen reflektiertes Licht.
Light, Specular	Von glatten Oberflächen spiegelnd reflektiertes Licht.
Mapping	Abbildung oder Zuordnung.
Pipe	Grafikkarte.
Rendering	Erzeugung eines digitalen Bildes aus einer Bildbeschreibung.
Shading	Simulation der Oberflächen grafischer Objekte anhand der Beleuchtung und Materialeigenschaften.
Shininess	Glanz, ein Koeffizient der angibt wie stark eine Objektoberfläche einfallendes Licht spiegelt.
State	Zustand.

Task	Aufgabe.
Triangles	Dreiecke. Elementare Objekte der Computergrafik aus denen komplexere Objekte typischerweise zusammengesetzt sind.
Vertices	Eckpunkte grafischer Objekte.
View Frustum Culling	Eliminierung von Objekten die ausserhalb des sichtbaren Bereiches liegen vor dem Rendering.
Viewport	Sichtbereich innerhalb eines Fensters.

Literaturverzeichnis

[1] Angel, Edward: *Interactive Computer Graphics. A Top-Down Approach Using OpenGL.* Addison Wesley, 4. Auflage, 2005.

[2] Bavoil, Louis: *OpenGL VBO Benchmarks.* `http://www.sci.utah.edu/~bavoil/opengl/` (letzter Abruf im September 2007).

[3] Eilemann, Stefan: *Equalizer Programming Guide.* `http://www.equalizergraphics.com/documents/Developer/ProgrammingGuide.pdf` (letzter Abruf im September 2007).

[4] Eilemann, Stefan und Renato Pajarola: *The Equalizer Parallel Rendering Framework.* Technischer Bericht IFI-2007.06, Department of Informatics, University of Zurich, Switzerland, Juni 2007.

[5] *Equalizer - Parallel Rendering.* `http://www.equalizergraphics.com/` (letzter Abruf im September 2007).

[6] *Equalizer - Task Methods.* `http://www.equalizergraphics.com/documents/design/taskMethods.html` (letzter Abruf im September 2007).

[7] *Equalizer - Downloads.* `http://www.equalizergraphics.com/downloads.html` (letzter Abruf im September 2007).

[8] *OpenGL - The Industry Standard for High Performance Graphics.* `http://www.opengl.org/` (letzter Abruf im September 2007).

[9] *OpenGL Shading Language.* `http://www.opengl.org/documentation/glsl/` (letzter Abruf im September 2007).

[10] Pajarola, Renato: *Why large-data visualization is not solved.* Unpubliziert.

[11] *PLY - Polygon File Format.* `http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/` (letzter Abruf im September 2007).

[12] *The Stanford 3D Scanning Repository.* `http://graphics.stanford.edu/data/3Dscanrep/` (letzter Abruf im September 2007).

[13] *Cyberware 3D Scanner Samples FTP Directory.* `ftp://ftp.cyberware.com/pub/samples/` (letzter Abruf im September 2007).

[14] *Spatial Index Demos.* `http://donar.umiacs.umd.edu/quadtree/index.html` (letzter Abruf im September 2007).

[15] *Vertex Buffer Objects - Whitepaper.* `http://www.spec.org/gwpg/gpc.static/vbo_whitepaper.html` (letzter Abruf im September 2007).

[16] *oZone3D.Net Tutorials - Vertex Buffer Objects.* `http://www.ozone3d.net/tutorials/opengl_vbo.php` (letzter Abruf im September 2007).

[17] *Wikipedia - kd-Tree.* `http://en.wikipedia.org/wiki/Kd-tree` (letzter Abruf im September 2007).

[18] *Wikipedia - Octree.* `http://en.wikipedia.org/wiki/Octree` (letzter Abruf im September 2007).