TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

Professur für Graphische Datenverarbeitung und Visualisierung

Diplomarbeit

Kommunikationsmechanismen für paralleles, adaptives Level-of-Detail in VR-Simulationen

Verfasser:

Tino Schwarze

geboren am 14. November 1977 in Karl-Marx-Stadt

Betreuer:

Prof. Dr. G. Brunnett

Dipl.-Inf. M. Lorenz

Chemnitz, 5. März 2003

Schwarze, Tino:
Kommunikationsmechanismen für paralleles, adaptives Level-of-Detail in VR-Simulationen
Diplomarbeit, Technische Universität Chemnitz, 2002.

Aufgabenstellung

Die für realistische VR-Simulationen benötigte Modellkomplexität ist nach wie vor nicht zufriedenstellend mit der verfügbaren Rechenleistung zu bewältigen. Die daraus resultierende Latenz zwischen Nutzer-Aktion und System-Reaktion behindert den angestrebten immersiven Eindruck der Simulation.

Ein verbreiteter Ansatz zur Beschleunigung von VR-Simulationen sind Level-of-Detail-Techniken. Dabei werden von sehr komplexen Objekten mehrere vereinfachte Stufen verwendet um Rechenaufwand zu sparen. Problematisch dabei ist die Erzeugung der Vereinfachungen. Dies geschieht in herkömmlichen Systemen manuell bei der Modellierung oder automatisch mit zeitaufwändigem Präprozessing. Für viele Anwendungsgebiete ist dies jedoch inakzeptabel. Eine Generierung der Vereinfachungen bei Bedarf ist wünschenswert. Aufgrund des hohen Aufwandes bietet sich die Parallelisierung dieses Schrittes an.

In der Studienarbeit „Paralleles, adaptives Level-of-Detail für VR-Simulationen“ wurde die grundlegende Realisierbarkeit dieses Ansatzes zur Beschleunigung von VR-Simulationen gezeigt. Der entwickelte Prototyp ist jedoch rudimentär und bis jetzt noch nicht praktisch sinnvoll einsetzbar.

Im Rahmen der Diplomarbeit soll näher auf die Kommunikationsmechanismen eingegangen und diese weiter optimiert werden. Besonderes Augenmerk soll dabei auf die Wiederverwendung reduzierter Objekte gelegt werden, da vermutlich nur durch die effiziente Nutzung bereits berechneter Objektreduktionen eine tatsächliche Geschwindigkeitssteigerung in der VR-Simulation zu erreichen ist. Dafür werden Methoden zur Verwaltung und Klassifizierung von geometrischen Objekten benötigt (z.B. für die Auswahl geeigneter Reduktionen und die Entscheidung, neue Reduktionen zu erzeugen).

Obwohl die eigentliche Reduktion von Objekten nicht Hauptgegenstand der Arbeit sein soll, werden existierende automatische Verfahren zur Geometrievereinfachung wichtige Kriterien für die Modellierung der Kommunikationsstrukturen liefern. Letztendlich sollen einige der vorgeschlagenen Verbesserung implementiert werden.

Inhaltsverzeichnis

1. Einleitung

Durch die rasante Entwicklung der Informationstechnik in den letzten Jahren werden immer mehr Anwendungsgebiete für anspruchsvolle Visualisierungen erschlossen. Was vor wenigen Jahren teurer Spezialhardware vorbehalten war, läßt sich heute mit Standard-Komponenten realisieren.

Besonders gut läßt sich diese Entwicklung anhand von Computerspielen verfolgen. Während noch vor wenigen Jahren abstrakte Grafiken den Spielverlauf darstellten, erreichen aktuelle Titel nahezu die Qualität computergenerierter Filme, obwohl die Grafikausgabe in Echtzeit berechnet wird. Es ist nicht unwahrscheinlich, daß in wenigen Jahren Spiele verfügbar sind, die auf einem durchschnittlichen Computer Spielfilmqualität erreichen.

Auch außerhalb der Unterhaltungsindustrie werden Visualisierungen immer praktikabler. Maschinen, Flugzeuge, Architektur und vieles mehr werden mit leistungsfähiger CAD-Software interaktiv am Computer entworfen. Der Rechner wird mehr und mehr vom „besseren Reißbrett“ zum Entwurfswerkzeug, das unter anderem einen realistischen Eindruck des Endergebnisses vermitteln kann.

1.1. Modellierung

Zur Erzeugung der virtuellen Gegenstände benötigt man zunächst eine Grundlage für ihre Modellierung um sie für die Verarbeitung mit dem Computer aufzubereiten. Man könnte zum Beispiel eine Kugel definieren als Menge:

$$K := \left\{ x^2 + y^2 + z^2 \leq 1 \quad | \, x, y, z \in R \right\} \tag{1.1}$$

Alternativ kann man eine Kugeloberfläche parametrisieren:

$$K(u, v) = \begin{bmatrix} \sin u \cos v \\ \sin u \sin v \\ \cos u \end{bmatrix}, u \in \left[0, \frac{\pi}{2}\right], v \in [0, 2\pi] \tag{1.2}$$

Weiterhin ist auch eine Annäherung der Kugel durch eine Menge von Polygonen denkbar. Jede dieser Darstellungsmethoden hat für verschiedene Anwendungsgebiete Vor- und Nachteile. Die implizite Darstellung (1.1) eignet sich zum Beispiel gut für Schnittberechnungen, sie ist dafür aber schwieriger zu visualisieren.

Letztendlich ist es von der spezifischen Anwendung abhängig, welche Darstellungsweise für die zu verarbeitenden Objekte optimal ist. Für die Echtzeitdarstellung hat es sich zum Beispiel bewährt, die virtuellen Objekte durch eine Menge von Polygonen zu approximieren, während für CAD-Aufgaben oft CSG[1]-Darstellungen verwendet werden.

1.2. Rendering

Als Rendering bezeichnet man die Transformation einer Modelldatenbasis in eine darstellbare Grafik. Im Unterschied zum Raytracing werden beim Rendering keine Strahlen verfolgt, sondern die Projektion der Objekte auf die Bildebene berechnet.

Der Aufwand für das Rendering einer Szene ist also direkt von deren Komplexität (Anzahl der Objekte, Polygone, Lichtquellen etc.) abhängig. Eine große Menge von Algorithmen wurde entwickelt, um diesen Aufwand möglichst gering zu halten, d.h. unnötige Berechnungen zu vermeiden (z.B. durch effiziente Erkennung nicht sichtbarer Teile) oder komplizierte Berechnungen zu vereinfachen (z.B. durch Approximation physikalischer Phänomene).

1.3. Virtual Reality

Unter Virtual Reality (VR) versteht man das Gesamtproblem der Erschaffung einer virtuellen (nicht tatsächlich vorhandenen, scheinbaren) Realität. Nimmt man den Begriff Virtual Reality wörtlich, so gehören zu einer vollständigen Simulation alle fünf Sinne des Menschen. Allgemein wird VR jedoch als vorwiegend optische Simulation verstanden, die teilweise mit akustischen und haptischen Reizen aufgewertet wird.

[1] Constructive Solid Geometry. Ein Objekt wird aus einer Menge von Grundprimitiven mit Hilfe von booleschen Operationen (Schnitt, Vereinigung, Differenz) erzeugt.

Die Computergrafik setzt sich im Kontext der VR primär mit den visuellen Eindrücken und der Interaktivität auseinander. Das Ziel ist ein möglichst immersiver Gesamteindruck, das bedeutet, daß der Nutzer der Simulation so weit in diese „eintaucht", daß er sie als real wahrnimmt. Hauptkriterien dabei sind unter anderem die Reaktionszeit, der dreidimensionale Eindruck sowie eine möglichst intuitive Interaktion.

Zu diesem Zweck wird eine Reihe von Spezialhardware eingesetzt – angefangen von 3D-Eingabegeräten (wie z. B. Datenhandschuh und SpaceMouse[2]) über spezielle Hardware (3D-Grafikbeschleuniger) bis zu 3D-Ausgabegeräten (HMD[3] oder Shutterbrillen[4]). Das größte Problem der VR ist nach wie vor die Reaktionszeit der Simulation. Wirklich realistisch wirkende Szenarien sind meist so komplex, daß sie auch teure Spezialhardware überfordern, d.h. die Berechnung eines neuen Bildes nach einer Veränderung in der Szene dauert so lange, daß der Betrachter dies bemerkt – „Es ruckelt."

Für einen flüssigen Bewegungsablauf und damit realistischen Eindruck gelten 25 Bilder pro Sekunde als untere Grenze. Möchte man einen 3D-Eindruck vermitteln, benötigt man die doppelte Menge an Bilddaten (je ein Bild pro Auge), also rein rechnerisch mindestens 50 Bilder pro Sekunde. Noch höhere Anforderungen stellt die Augmented Reality (AR)[5], da hier schon eine kleine Kopfbewegungen des Beobachters zu einer beträchtlichen Diskrepanz zwischen realer Welt und überlagerter Darstellung führen kann. Wird die überlagerte Darstellung nicht schnell genug aktualisiert, stellt sich ein „Schwimmeffekt" ein.

[2] Ein Joystick-ähnliches Eingabegerät, mit dem direkt alle 6 Freiheitsgrade (Translation und Rotation in jeweils 3 Dimensionen) kontrolliert werden können.

[3] Head Mounted Display. Ein Anzeigegerät, daß wie eine Brille getragen wird und für jedes Auge ein anderes Bild darstellt, wodurch ein realistischer 3D-Eindruck erweckt werden kann.

[4] Eine Alternative zum HMD. Eine Brille, die computergesteuert synchron zum Anzeigegerät (z. B. Monitor) abwechselnd das linke oder rechte Auge des Betrachters verdeckt. Dadurch kann jedem Auge ein anderes Bild angezeigt und ein 3D-Eindruck erweckt werden. Vorteil: Wenn die Synchronisierung per Funk oder Infrarot erfolgt, können mehrere Betrachter die gleiche Szene 3-dimensional betrachten, zum Beispiel wenn diese an eine Wand projiziert wird.

[5] Augmented Reality. „Überlagerte Realität." Die AR befaßt sich mit der Überlagerung von realen durch virtuelle Bilder. Ein beliebtes Beispiel für eine AR-Anwendung ist ein Wartungstechniker, dem vom Rechner ein Schema einer komplexen Maschine in das Sichtfeld eingeblendet wird. Auch die Instrumenteneinblendungen im Cockpit von Kampfflugzeugen sind der AR zuzurechnen.

> Especially rotation of the head can be very fast, in the order of 300 degrees per second and faster if necessary, and in such a situation a latency of 10 ms will already amount to a misplacement of 3 degrees. It has been shown experimentally that humans can detect latencies as low as 8 ms and probably less ... For mobile AR tasks such as tourist information, remote maintenance and support, we estimate that 10 ms will be acceptable ...
>
> [18]

Eine Latenz von 10 Millisekunden entspricht einer Bildwiederholrate von 100 Bildern pro Sekunde, für zwei Augen somit 200 Bilder pro Sekunde. Diese geringen Latenzzeiten werden für AR benötigt, da im Unterschied zur VR der Nutzer die reale Welt gleichzeitig mit der simulierten Realität wahrnimmt. Diskrepanzen zwischen realer Welt und überlagerter Darstellung zerstören deshalb sehr schnell die Illusion der erweiterten Realität.

Für VR Simulationen sind die Anforderungen nicht so extrem hoch, da der Nutzer ausschließlich mit der simulierten Realität interagiert und somit keinen Bezug zur realen Welt wahrnimmt. Es ist jedoch wahrscheinlich, daß eine höhere Simulationslatenz Ermüdungserscheinungen beim Nutzer begünstigt.

Der derzeitige Stand der Technik ist selbst für VR-Simulationen noch nicht leistungsfähig genug. Deshalb müssen Wege gefunden werden, um den Aufwand für die Berechnung eines Bildes beträchtlich zu reduzieren.

1.4. Beschleunigung von VR-Simulationen

Die Beschleunigung von 3D-Darstellungen im Allgemeinen und von VR-Simulationen im Speziellen ist Gegenstand aktiver Forschung. Es kann und soll an dieser Stelle nicht erschöpfend auf die Thematik eingegangen werden – statt dessen werden exemplarisch zwei Ansätze vorgestellt, die für diese Arbeit von besonderer Bedeutung sind.

1.4.1. Level-of-Detail

Mit Level-of-Detail (LOD) bezeichnet man eine Technik zur Reduzierung der Objektkomplexität für die Darstellung. Es wird davon ausgegangen, daß ein

komplexes Objekt in bestimmten Situationen (z.B. sehr weit vom Betrachter entfernt) nicht in allen Einzelheiten erkennbar ist und somit durch eine weniger komplexe Approximation ersetzt werden kann. In der Tat macht es wenig Sinn, ein Fahrzeug mit sämtlicher Inneneinrichtung, allen Karosseriedetails und Lackspiegelungen zu berechnen, wenn es im erzeugten Bild nur durch wenige Pixel repräsentiert wird und nicht von einem Quader mit passender Farbe zu unterscheiden wäre.

Um Rechenzeit einzusparen werden von geeigneten Objekten manuell oder automatisch mehrere Repräsentationen (Levels, Stufen) erzeugt, die jeweils unterschiedlich detailliert sind. Für das Rendering wird dann eine passende Detailstufe ausgewählt, zum Beispiel in Abhängigkeit vom Abstand des Objektes zum Betrachter.

1.4.2. Parallelisierung

Eine weitere Möglichkeit zur Beschleunigung von VR-Anwendungen ist ihre Parallelisierung. Crockett unterscheidet „functional parallelism, data parallelism, and temporal parallelism“ [2].

Funktionelle Parallelisierung

Der Darstellungsprozeß wird in einzelne Funktionen zerlegt, die parallel arbeiten und eine Pipeline bilden, d.h. jede Funktionseinheit gibt ihre Ergebnisse an die nächste weiter. Sobald die Pipeline gefüllt ist arbeiten alle Funktionseinheiten parallel und man erhält einen Geschwindigkeitsgewinn proportional zur Anzahl der Funktionseinheiten. Allerdings ist die Pipeline nur so schnell wie ihre langsamste Einheit (Flaschenhalsprinzip). Diese Art der Parallelisierung wird meist in Hardware realisiert.

Datenbasierte Parallelisierung

Anstatt die einzelnen Verarbeitungsschritte zu parallelisieren werden hier die zu verarbeitenden Daten auf mehrere identische Verarbeitungseinheiten verteilt. Der Geschwindigkeitsgewinn ist proportional zur Anzahl der Verarbeitungseinheiten (die durch ökonomische und technische Gegebenheiten beschränkt wird). Problematisch ist jedoch im allgemeinen der Transport der Daten (dazu zählen auch die erzeugten Bilddaten) von und zu den Einheiten.

Zeitliche Parallelisierung

Dieser Ansatz ist für Echtzeitanforderungen ungeeignet, jedoch für Animationen in hoher Qualität einsetzbar. Ähnlich wie bei der datenbasierten Parallelisierung wird eine Anzahl identischer Verarbeitungseinheiten bereitgestellt, die jeweils ein Bild der Animation zugewiesen bekommen und dann berechnen. Letztendlich bleibt der Aufwand für die Berechnung eines einzelnen Bildes gleich, es wird die Zeit für die Berechnung der Gesamtanimation reduziert.

1.5. Zusammenfassung

Trotz großer Fortschritte sowohl technischer als auch algorithmischer Natur lassen die derzeitigen Möglichkeiten viele Wünsche für VR-Simulationen offen. Jeder Geschwindigkeitsgewinn kann sofort durch eine höhere Komplexität der Szene kompensiert werden – es ist noch nicht absehbar, wann genügend Rechenleistung für realistische Simulationen zur Verfügung stehen wird, oder wie Erikson[4] es formuliert:

> For every computer graphics system, there exists a model complex enough to bring its performance to a crawl.

Die Unterhaltungsindustrie stellt bereits erstaunlich realistische Szenarien in Echtzeit auf Standard-Computern dar. Eine VR-Simulation unterscheidet sich jedoch beträchtlich von einem Computerspiel. Einerseits sind die Spielwelten sorgfältig manuell entworfen während VR-Szenarien oft auf 3D-Scans realer Objekte oder hochdetailierten CAD-Daten basieren[6]. Andererseits wird an die VR der Anspruch der Simulation einer realen Welt gestellt.

Dies bedeutet im Unterschied zum Computerspiel, daß die Szenerie viel detailierter modelliert werden muß. Während im Spiel die Bewegungsfreiheit des Nutzers oft stark eingeschränkt wird (es dürfen nur die vorgesehenen Wege beschritten werden, alle Objekte abseits sind lediglich Fassaden), ist ein Ziel der VR die natürliche, inituitive Bewegung in der Simulation. Wenn also in der Simulation ein Gebäude eine Tür hat, so soll sich diese auch

[6] Eine typische Spielszene beinhaltet mehrere zehntausend Polygone während ein einzelnes Gebäude in einer VR-Simulation ohne weiteres aus mehreren hunderttausend Polygonen bestehen kann.

öffnen lassen und das Innere des Gebäudes preisgeben – im Spiel verzichtet man schlichtweg auf die Modellierung des Inneren, wenn das Gebäude nicht Teil des vorgesehenen Spielverlaufs ist. Die möglichst intuitive Simulation macht es weiterhin erforderlich, viel mehr Dynamik und somit letztendlich auch viel mehr Details zu modellieren. Während im Spiel eine Hausfassade durch ein Polygon mit detailreicher Textur repräsentiert werden kann ist dies bei VR-Simulationen nicht erwünscht, da dies die Interaktionsmöglichkeiten stark einschränkt – ein Fenster, daß nur in der Fassadentextur abgebildet ist, kann eben nicht geöffnet werden. Durch die größere Bewegungsfreiheit des Nutzers und die optimalerweise stereoskopische Betrachtung der VR-Simulation fällt außerdem die fehlende Räumlichkeit der Fassadendetails schnell unangenehm auf, da der Nutzer die Diskrepanz zwischen bildlicher Darstellung und räumlicher Situation zumindest unterbewußt wahrnimmt.

Aufbauend auf die vorangegangene Studienarbeit[25] richtet sich das Hauptaugenmerk dieser Arbeit auf die parallele, automatische Erzeugung von Level-of-Detail-Stufen für eine VR-Simulation und die dafür notwendige Kommunikation. Dazu werden als nächstes die durch die Studienarbeit geschaffenen praktischen Voraussetzungen beschrieben und Verbesserungsvorschläge vorgestellt.

Danach werden bestehende Verfahren zur automatischen Objektvereinfachung vorgestellt, auf ihre Eignung untersucht und wichtige Aspekte für die angestrebte Anwendung herausgearbeitet. Die effiziente Wiederverwendung reduzierter Objekte ist Gegenstand des vierten Teils dieser Arbeit. Es werden die damit verbundenen Problem erörtert und Lösungsvorschläge unterbreitet.

Im fünften Teil der Arbeit wird die Architektur und Funktionsweise des entwickelten Systems näher erläutert. Letztendlich werden die erzielten Ergebnisse zusammengefaßt und es werden Perspektiven für weitereführende Arbeiten aufgezeigt.

2. Voraussetzungen

Bevor näher auf die Objektreduktion eingegangen wird, sollen zunächst der Ansatz beschrieben und die praktischen Voraussetzungen betrachtet werden. In der Studienarbeit „Paralleles, adaptives Level-of-Detail für VR-Simulationen“ wurden die Grundlagen für ein System zur verteilten Reduktion von Objekten erarbeitet und ein Prototyp implementiert[25].

2.1. Idee

VR-Simulationen beinhalten oft sehr komplexe geometrische Objekte. Komplexe Objekte erfordern einen hohen Rechenaufwand für die Visualisierung, der sich durch die Modellierung verschiedener Detailstufen (Level-of-Detail) reduzieren läßt. Bislang geschieht diese Modellierung manuell oder durch aufwändiges Präprozessing. Wünschenswert ist jedoch eine automatische Generierung der Detailstufen bei Bedarf. Aufgrund des hohen Rechenaufwandes bietet sich eine Parallelisierung dieses Schrittes an.

Die Ausgangsdaten für VR-Simulationen sind typischerweise eine Menge von geometrischen Objekten, die in einem azyklischen gerichteten Graph logisch organisiert sind, dem sogenannten Szenegraph. Diese Art der Darstellung hat sich in der Praxis bewährt, da mit Hilfe des Graphen die hierarchischen Abhängigkeiten der einzelnen Objekte gut abgebildet werden können[1]. Offensichtlich bietet sich eine datenbasierte Parallelisierung an - jeweils ein Teil(graph) der Szene wird einer der mehrfach vorhandenen Reduktionseinheiten zugewiesen. Die Zuweisung der Aufträge zu Redukti-

[1] z.B. für das Modell eines Menschen: Der Hals schließt oben an den Torso an, der Kopf befindet sich über dem Hals, an der linken und rechten Seite des Kopfes ist jeweils ein Ohr. Wird jetzt der Körper bewegt, überträgt sich die Bewegung automatisch auf Hals, Kopf und Ohren. Wird der Hals gedreht, drehen sich sowohl Kopf und Ohren entsprechend.

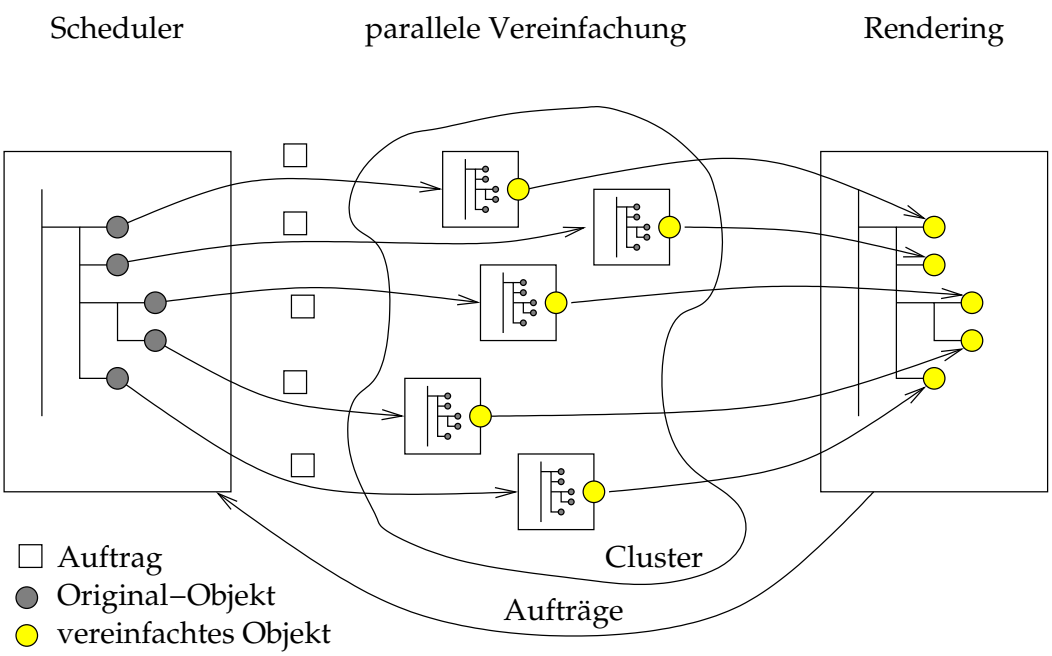


Abbildung 2.1.: Prinzip der parallelen Objektreduktion

onseinheiten übernimmt dabei eine spezielle Schedulingkomponente. Somit ergibt sich das in Abb. 2.1 dargestellte Prinzip für die parallele Objektreduktion.

2.2. Aufbau und Funktionsweise des Prototypen

Der Prototyp ist keine eigenständige Applikation. Stattdessen ist er als Plugin für den CADaVR[2]-Viewer ausgeführt (siehe Abbildung 2.2). Er besteht aus drei Hauptkomponenten:

1. Einer Schnittstelle zur VR-Simulation (zur Extraktion der originalen und Integration der reduzierten Geometriedaten) - DynVLod_CadvisPlugin.
2. Einer Schedulingkomponente zur Verteilung der Reduktionsaufträge auf die Reduktionskomponenten - ReducerScheduler.

[2]Eine textbasierte Schnittstelle für die Beschreibung virtueller, dynamischer Welten. Siehe auch [29].

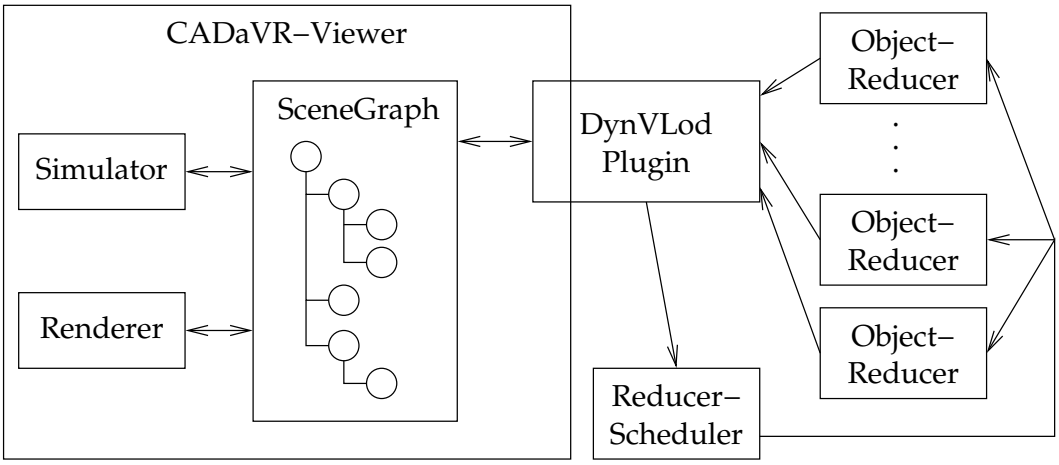


Abbildung 2.2.: Architektur des Prototyps aus der Studienarbeit [25]

3. Den Reduktionskomponenten - ObjectReducer und darauf aufbauende Komponenten.

Die einzelnen Komponenten kommunizieren mittels CORBA miteinander, so daß die objektorientierte Ausführung des Gesamtsystems auch für die systeminterne Kommunikation zum Tragen kommt. Der ReducerScheduler hat dabei eine zentrale Funktion, denn alle anderen Komponenten wenden sich an diesen zur Etablierung der Kommunikation – die Reduktionskomponenten melden sich dort nach dem Start an und das DynVLod_CadvisPlugin sendet seine Reduktionsaufträge dorthin.

2.3. Zielplattform

Die primäre Zielplattform des Prototypen aus der Studienarbeit ist das 2002 an der Professur für graphische Datenverarbeitung installierte HP Visualize Center II. Zur Reduktion der Geometriedaten sollen Rechner des Chemnitzer Linux Cluster ()[27], kurz CLiC, verwendet werden.

2.3.1. Renderingplattform

Das HP Visualize Center II besteht aus drei HP Workstations J6700 mit jeweils zwei HP-PARISC Prozessoren mit 750 MHz sowie 4 GB Arbeitsspeicher. Die

Workstations sind untereinander mit Gigabit-Ethernet vernetzt und zusätzlich mit Fast-Ethernet (100 MBit/s) an das Universitätsnetz angeschlossen. Jede der Workstations verfügt über eine Visualize FX10B Grafikkarte. Die Grafikkarten sind über einen Panoram Integrator 3 Matrix Switcher mit drei Videobeamern Electrohome Marquee LC8500 mit Fast Phosphor () Röhren verbunden. Für die Erzeugung des 3D-Eindruckes kommen Crystal Eyes Long Range Active Stereo Shutter Glasses zum Einsatz, die mit Hilfe von Infrarot-Sendern zum Bild synchronisiert werden.

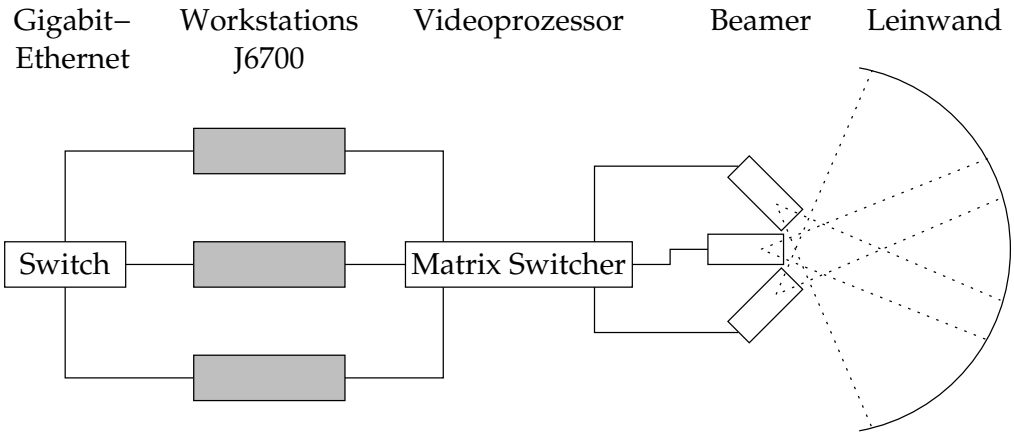


Abbildung 2.3.: Schematischer Aufbau des HP Visualize Center II

Wie in Abb. 2.3 dargestellt, überlappen die Projektionsflächen der Videobeamer. Der Videoprozessor berechnet dafür eine Überblendung zwischen benachbarten Bildern um den Eindruck eines fortlaufenden Bildes hervorzurufen. Mittels eines speziellen Softwarelayers der Workstations können die Grafiksysteme der Workstations derart miteinander verbunden werden, daß sie sich wie ein einziger, größerer Bildschirm verhalten. Für den Anwender (bzw. Programmierer) ist also die Verteilung der Grafik völlig transparent.

2.3.2. Reduktionsplattform

Der Chemnitzer Linux Cluster besteht aus 528 Knoten mit jeweils einem Pentium-III Prozessor (800 MHz) und 512 MB Arbeitsspeicher. Für die Kommunikation im Cluster stehen zwei physisch getrennte Netzwerke mit jeweils 100 MBit/s Bandbreite zur Verfügung. Das sogenannte Servicenetz dient dem Zugang zum Knoten, der Überwachung und dem Zugriff auf

das verteilte AFS-Dateisystem. Das sogenannte Kommunikationsnetz dient ausschließlich der Cluster-internen Kommunikation. Es ist kompressionsfrei, d.h. alle Knoten können mit voller Bandbreite gleichzeitig senden und empfangen, ohne das es zu Kollisionen im Ethernet kommt. Das Servicenetz im CLiC ist mit Gigabit-Ethernet an das Universitätsnetz angeschlossen.

2.4. Aktueller Stand und Kritik

Der vorhandene Prototyp realisiert die Übertragung der Szenedaten zu den Reduktionskomponenten (im folgenden Reduzierer genannt) sowie die Integration der reduzierten Objekte in die Szene. Weiterhin werden alle Objekte der Szene, die als LOD-Objekte ausgezeichnet sind, zur Reduktion herangezogen. Zu Testzwecken wurde eine Reduktion auf die Bounding Box[3] implementiert.

Eine praktische Verwendbarkeit ist jedoch noch nicht gegeben. Das ist zum einen auf eine fehlende sinnvolle Objektreduktion und zum anderen auf die unvollständige Übetragung der Szenedaten zurückzuführen. So unterstützt der Prototyp zum Beispiel keinerlei Materialien – der Reduzierer hat also nicht einmal die Möglichkeit, die Farbe einer Fläche zu ermitteln oder für die Reduktion zu spezifizieren.

Auf weitere Unzulänglichkeiten und Lösungsvorschläge bzw. realisierte Lösungen wird im folgenden eingagangen.

2.4.1. SceneGraph-Abbildung

Bislang war die Abbildung der Szenedaten für die Netzwerkübertragung unvollständig. Materialien, Texturen und Lichtquellen waren den Reduktionskomponenten nicht zugänglich. Diese Unzulänglichkeiten wurden im Rahmen dieser Arbeit behoben. Den Reduzierern steht jetzt eine vollständige Abbildung der Szene zur Verfügung.

Um zu überprüfen, daß die Übertragung der Szenedaten verlustlos ist (also tatsächlich eine 1:1 Abbildung der Szenedaten für die Netzwerkübertragung erfolgt), wurde ein Reduzierer implementiert, der ein Äquivalent des

[3] Kleinster Quader, der ein Objekt komplett einschließt. Aus Effizienzgründen sind die Bounding Boxes meist achsenparallel. Verwendet werden Bounding Boxes zum Beispiel zur Vereinfachung von Kollisionserkennung oder für die Markierung von Objekten zur Interaktion.

Originalobjektes als „Reduktion“ liefert. Dieser Reduzierer wurde erfolgreich getestet.

2.4.2. Redundanz

Bei der Vervollständigung der Szenedatenübertragung wurden zur Redundanzvermeidung Materialien und Texturen als Szene-globale Ressourcen implementiert, die innerhalb der Szene nur durch eine eindeutige Ganzzahl referenziert werden.

Nach wie vor kritisch ist eine Veränderung der Szenegeometrie. Ein Aktualisierungsverfahren, das nur die tatsächlichen Änderungen übeträgt, ist hier wünschenswert. Dessen Implementation würde jedoch den Rahmen dieser Arbeit sprengen, da der gesamte Szenegraph auf Veränderungen überwacht werden müsste und für jedes Attribut im Szenegraph eigene Schnittstellen (im Netzwerkprotokoll) für dessen Veränderung etabliert werden müßten.

2.4.3. Objektreduktion

Wie bereits erwähnt findet bislang keine praktisch verwendbare Objektreduktion statt. Dies hat unter anderem zur Folge, daß über die zur Reduktion notwendigen Parameter bislang nur spekuliert werden konnte. Eine Auswahl bekannter Reduktionsverfahren wird in Kapitel 3 vorgestellt und diskutiert.

2.4.4. Wiederverwendung von Reduktionen

Der im Prototyp verwendete Bounding-Box-Reduzierer legt eine Wiederverwendung berechneter Reduktionen nahe, da sich die Bounding Box nur ändern kann, wenn die Geometrie des Objektes verändert wird. Erste Überlegungen für eine Wiederverwendung sind bereits in den Prototyp eingeflossen – so kann ein Reduzierer zu jedem reduzierten Objekt optionale Validitätskriterien liefern. Über den Charakter dieser Kriterien konnte jedoch nur spekuliert werden, auch wurden die Kriterien nicht ausgewertet. In Abschnitt 4.1 wird auf diesen Komplex näher eingegangen.

2.5. Zusammenfassung

Nachdem mit der Studienarbeit [25] erste Grundlagen und praktische Voraussetzungen geschaffen wurden, ist das Ziel dieser Arbeit die Optimierung der Kommunikation und die effiziente Wiederverwendung berechneter Reduktionen zum Erreichen einer tatsächlichen Geschwindigkeitssteigerung.

Im nächsten Schritt werden bestehende Verfahren zur automatischen Generierung von Objektreduktionen vorgestellt. Danach werden die Grundlagen für eine Wiederverwendung von Reduktionen erarbeitet.

3. Automatische Generierung von LOD-Stufen

LOD-Techniken sind mittlerweile Bestandteil nahezu aller Toolkits für VR-Simulationen und Echtzeit-Rendering. Durch die Modellierung mehrerer, verschieden komplexer Versionen desselben Objektes kann man den Aufwand für dessen Darstellung mitunter beträchtlich verringern. Die Detailstufen wurden anfangs noch manuell modelliert. Es liegt jedoch nahe, diese mühselige Aufgabe zu automatisieren.

Seit einigen Jahren ist die automatische Vereinfachung von Objekten Gegenstand aktiver Forschung. Es gibt eine Vielzahl von Veröffentlichungen zu diesem Thema. Hier soll nur die Vereinfachung polygonaler Modelle diskutiert werden.

3.1. Messung von Objektähnlichkeit

Ziel der Vereinfachung eines polygonalen Modelles M ist es, ein Modell M' mit weniger Polygonen zu erhalten, welches das Original möglichst gut annähert. Für die Messung der Abweichung vom Original existieren sehr viele verschiedene Metriken. Im Allgemeinen hängt die Wahl des Ähnlichkeitskriteriums von der Anwendung ab. Beim Einsatz von Objektreduktionen zur Renderingbeschleunigung ist sicherlich die optische Ähnlichkeit das ausschlaggebende Kriterium. Garland stellt dazu fest[9]:

> However, in almost all cases, researchers in the field of simplification have chosen to use similarity of shape as the primary criterion for evaluating approximation quality. Not only do shape-based metrics appear to be more computationally convenient, but they are also more appropriate in non-rendering applications such as finite element analysis.

Das Grundproblem von Ähnlichkeitsmetriken, die direkt die Unterschiede im Aussehen (d.h. in der gerenderten Darstellung) messen, ist, daß das Aussehen eines Modells vom Betrachterstandpunkt abhängig ist. Die Wahl der Referenzbetrachtungsstandpunkte beeinflußt also das Ergebnis der Metrik enorm. Garland führt weitere Probleme an:

> If we neglect some important part of the viewpoint space, we may very well remove perceptually significant features. And since each sample may involve an expensive rendering step, we cannot make many samples. Indeed, rendering the models for comparison is likely to be quite expensive; simplification is generally performed on models which are prohibitively expensive to render in the first place. [9]

Es existieren eine Reihe von Metriken, die geometrische Ähnlichkeit messen, z.B. den maximalen Abstand eines Punktes in M von M' oder den durchschnittlichen Abstand. Solche globalen Metriken sind jedoch sehr rechenaufwändig und deshalb ungeeignet zur Steuerung des eigentlichen Reduktionsalgorithmus. Oft werden einfachere Approximationen verwendet.

3.2. Reduktionsverfahren

Heckbert und Garland bieten in [10] einen Überblick des umfangreichen Gebiets der Reduktion polygonaler Flächen. Sie unterscheiden dabei „Height Fields“ (Höhenprofile), parametrisierte Flächen, mannigfaltige Flächen[1], mannigfaltige Flächen mit Rand und nicht-mannigfaltige Flächen. Im Rahmen dieser Arbeit werden beliebige polygonale Modelle verwendet. Es muß also davon ausgegangen werden, daß nicht-mannigfaltige Objekte vorliegen. In der Praxis ist dies tatsächlich der Fall.

Leider gibt es nur sehr wenige Reduktionsalgorithmen, die nicht-mannigfaltige Ausgangsobjekte verarbeiten können. Heckbert und Garland[10] zählen dazu gerade drei Algorithmen, die im Anschluß an diese Einleitung kurz vorgestellt werden. Heckbert und Garland sprechen dem Ansatz

[1] Die infinitesimale Nachbarschaft jedes Punktes einer mannigfaltigen Fläche ist topologisch äquivalent zu einer Scheibe. Wenn man eine mannigfaltige Fläche trianguliert, gehört jede Kante zu genau zwei Dreiecken. Trianguliert man eine mannigfaltige Fläche mit Rand, so gehört jede Kante entweder zu genau einem oder zu genau zwei Dreiecken.

von Schroeder et al. (siehe [24]) zwar die Fähigkeit ab, nicht-mannigfaltige Objekte zu verarbeiten, jedoch stellt Andújar [1] fest: „Geometric domain: Unrestricted [...] Topology domain: Unrestricted“ (aus einer Tabelle). In der originalen Veröffentlichung von Schroeder et al. werden keinerlei Einschränkungen für die Topologie des Ausgangsobjektes erwähnt. Der Algorithmus dürfte durchaus für nicht-mannigfaltige Objekte geeignet sein.

Grundsätzlich lassen sich die Algorithmen zur Geometrievereinfachung in topologieerhaltende und topologieverändernde Verfahren unterteilen. Die Topologie eines Objektes wird zum Beispiel verändert, wenn Durchbrüche entfernt oder ursprünglich separate Objekte verbunden werden. Es gibt Anwendungen, für die eine Topologieänderung inakzeptabel ist (z.B. die Visualisierung schematischer Moleküle). Für VR-Simulationen im Allgemeinen sind topologiemodifizierende Verfahren jedoch vorteilhaft, da sie größeren Spielraum für die Approximation des Originalobjektes haben.

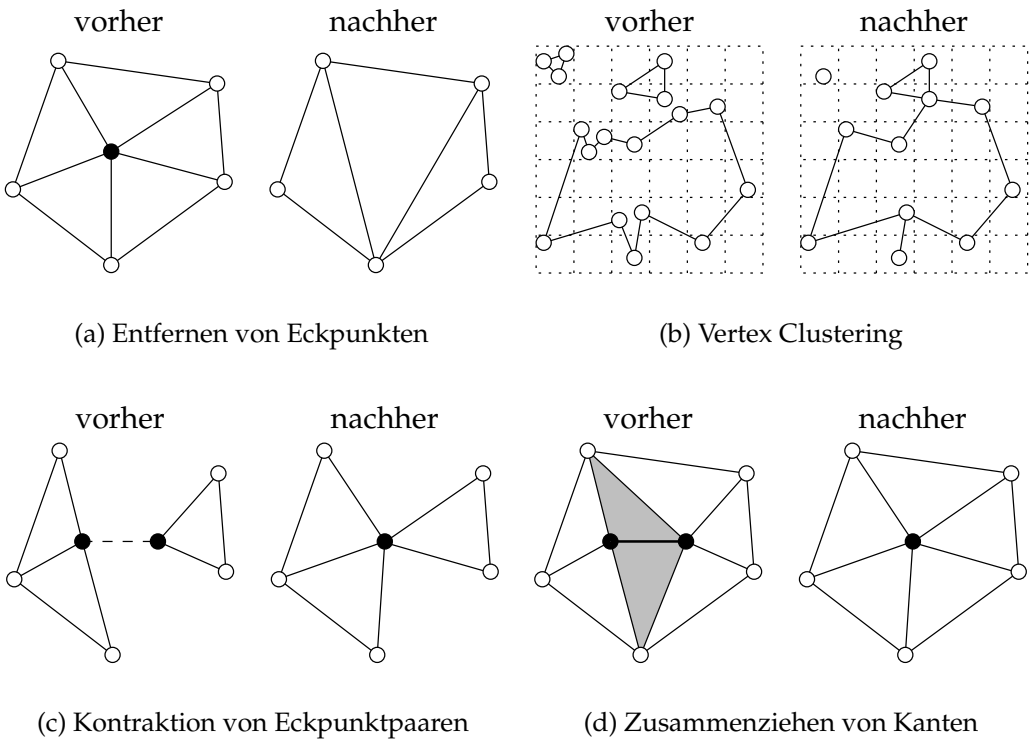
Eine weitere Klassifizierung der Algorithmen basiert auf dem Vorgehen bei der Vereinfachung. Während dezimierende Algorithmen ausgehend vom Originalmodell durch Entfernung oder Kontraktion von Eckpunkten, Kanten oder Flächen eine Vereinfachung erzeugen, beginnen verfeinernde Algorithmen mit einer groben Approximation des Originals und fügen nach und nach Details hinzu. Da die Generierung der ersten Näherung eines Objektes prinzipiell sehr schwierig ist und in der Regel nur unter der Bedingung der Topologieerhaltung eine brauchbare Ausgangsbasis für einen verfeinernden Algorithmus liefert, sind solche Algorithmen hier nicht vertreten[2].

Die im Folgenden gezeigten Beispiele für reduzierte Objekte der einzelnen Algorithmen basieren alle auf dem sogenannten „Stanford Bunny“, dessen Original in Abbildung 3.2 zu sehen ist.

3.2.1. Decimation

Der Algorithmus von Schroeder et al.[24] iteriert über alle Eckpunkte des Objektes und entfernt Eckpunkte, sowie dazugehörige Flächen, wenn ein bestimmtes Kriterium erfüllt wird. Das dabei entstehende Loch wird dann neu trianguliert (siehe Abb. 3.1(a)). Dieser Prozeß wird so lange wiederholt,

[2]Die Erzeugung der ersten Näherung für mannigfaltige Objekte ist an sich schon schwierig genug, es dürfte also noch eine zusätzliche Schwierigkeit darstellen, dies für nicht-mannigfaltige Objekte zu bewerkstelligen.



(a) Entfernen von Eckpunkten

(b) Vertex Clustering

(c) Kontraktion von Eckpunktpaaren

(d) Zusammenziehen von Kanten

Abbildung 3.1.: Basisoperationen für dezimierende Reduktionsalgorithmen (Abb. nach [10])

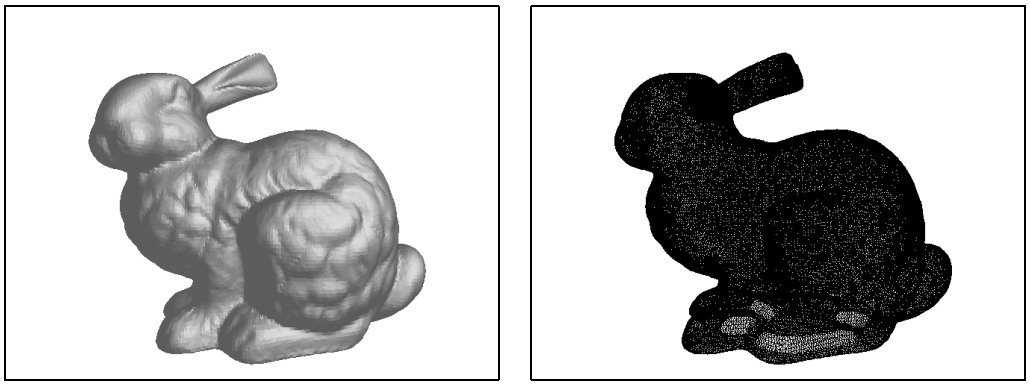


Abbildung 3.2.: Der Stanford-Bunny im Original (69.451 Dreiecke).

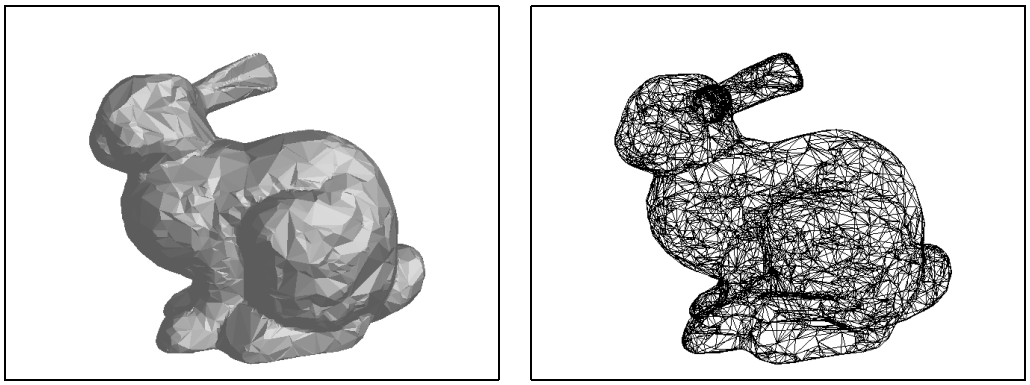


Abbildung 3.3.: Reduktion mit vtkDecimatePro (6.944 Dreiecke).

bis ein Abbruchkriterium (z.B. Grad der Reduktion oder maximaler Fehler) erfüllt wird.

Das Visualization Tool Kit (VTK)[3] enthält zwei Implementationen von Varianten dieses Algorithmus. Die erste, vtkDecimate, wurde erweitert um ein globales Fehlermaximum und die Möglichkeit Topologiemodifikationen zu erlauben. Die zweite Implementation, vtkDecimatePro ergänzt zusätzlich die Option, das Originalobjekt vor oder während der Reduktion in Teile zu zerlegen. Dies dient der besseren Topologieerhaltung, verringert aber natürlich den Spielraum für die Reduktion. Eine Reduktion des beliebten Stanford-Bunny[4] mit vtkDecimatePro ist in Abbildung 3.3 zu sehen.

3.2.2. Uniform Vertex Clustering

Rossignac und Borrel[20] schlagen einen Algorithmus vor, der das Modell ausgehend von seiner Bounding-Box in ein gleichmäßiges räumliches Raster einteilt und dann innerhalb jeder Rasterzelle einen repräsentativen Eckpunkt auswählt. Alle Eckpunkte innerhalb einer Zelle werden dann zum ausgewählten Punkt kontrahiert (siehe Abb. 3.1(b)). Dieses Verfahren kann sehr effizient implementiert werden und wurde auch kommerziell von IBM als Teil des „3D Interaction Accelerator“ verwertet.

> This method is extremely general, as it can operate on any set of triangles (not just manifolds), it can achieve arbitrary simplifica-

[3] http://www.vtk.org/

[4] siehe http://www.graphics.stanford.edu

> tion levels, and it can even eliminate small objects or otherwise change the topology of a surface. Unfortunately, it does not preserve detail well [...]
>
> [10]

Hauptschwäche des Algorithmus ist das gleichmäßige Raster. Der Algorithmus ist mit dem Resampling eines Bildes vergleichbar – die Platzierung des Rasters hat starken Einfluß auf das Ergebnis.

3.2.3. Floating Cell Clustering nach Low und Tan

Low und Tan[15] erweitern Uniform Vertex Clustering und beseitigen einige Schwächen. So wird kein reguläres Raster mehr verwendet, sondern eine Menge von Zellen (die eine beliebige einfache Form haben können, z.B. Würfel oder Kugel), die um Eckpunkte mit hohem Gewicht platziert werden. Weiterhin verbessern sie die Heuristik zur Gewichtung der Eckpunkte.

3.2.4. Quadric Error Metrics

Garland und Heckbert stellen in [6] einen Algorithmus vor, der auf „Pair Contraction“ basiert. Eine „Pair Contraction“ bezeichnet die Kontraktion zweier Eckpunkte zu einem neuen und die anschließende Entfernung degenerierter Flächen (siehe Abb. 3.1(c)). Zusätzlich wird eine Fehlermetrik eingeführt, die auf Quadriken basiert und während der Reduktion akkumuliert werden kann. Für nähere Informationen sei auf [8] verwiesen.

Später erweiterten Garland und Heckbert die Quadric Error Metrics (QEM) auf Flächenattribute wie Farbe und Textur[7]. Die Algorithmen liegen in einer frei verfügbaren Implementation namens QSlim vor[5], die beide Varianten des Algorithmus (ohne Attribute: QSlim, mit Attributen: PropSlim) enthält. Eine Reduktionn von QSlim zeigt Abbildung 3.4.

Auf Quadriken basierende Fehlermetriken wurden auch für andere Verfahren verwendet. So kombiniert Lindstrom diese in [14] mit Uniform Vertex Clustering und beschreibt einen Algorithmus zur Reduktion von Datenmengen, die den verfügbaren Arbeitsspeicher überschreiten. Eine Implementation dieses Algorithmus ist ebenfalls im VTK unter der Bezeichnung vtkQuadricClustering verfügbar (siehe Abb. 3.5 für ein Reduktionsbeispiel). Hoppe[12] stellt später eine neue QEM vor, die weniger Speicherplatz

[5] `http://www.uiuc.edu/~garland/software/qslim.html`

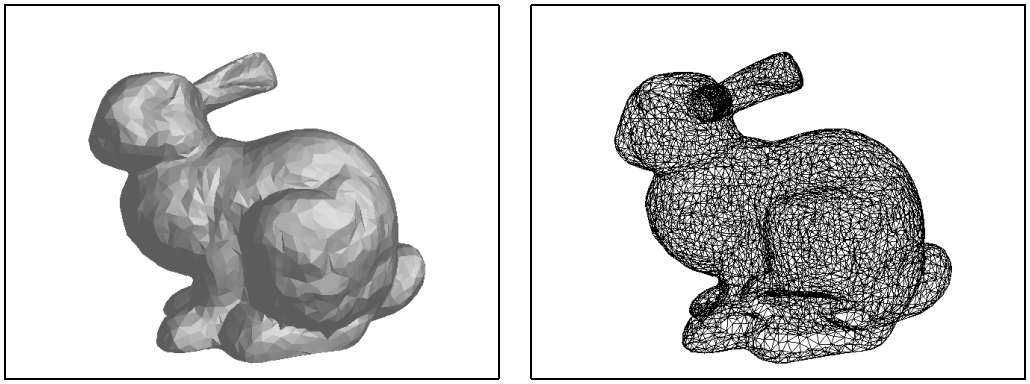


Abbildung 3.4.: Stanford-Bunny reduziert mit QSlim (6.944 Dreiecke).

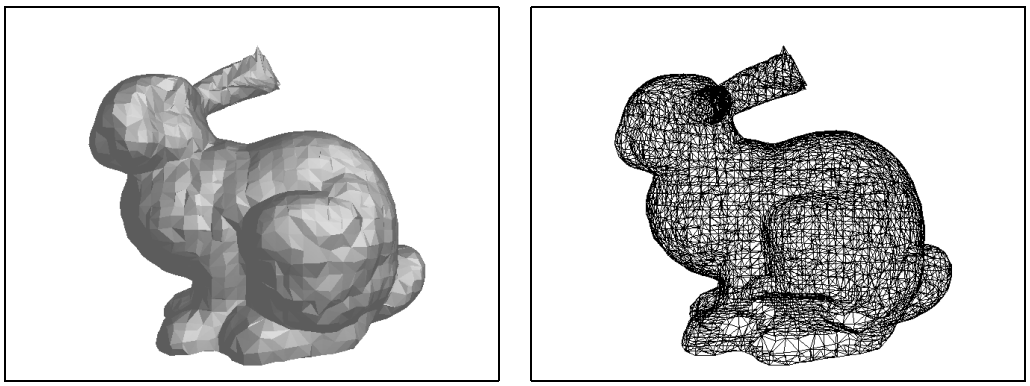


Abbildung 3.5.: Reduktion mit vtkQuadricClustering (6.941 Dreiecke).

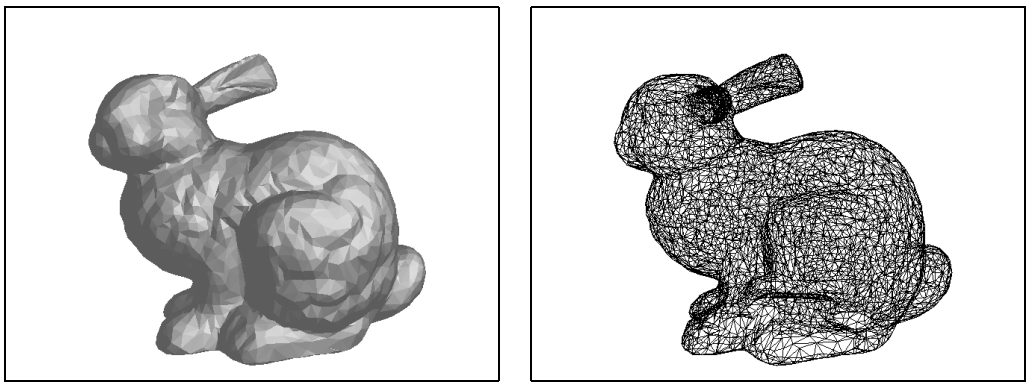


Abbildung 3.6.: Reduktion mit vtkQuadricDecimation (6.449 Dreiecke).

benötigt und schneller auszuwerten ist. Auch dieser Algorithmus ist als vtkQuadricDecimation Teil des VTK, ein Reduktionsergebnis zeigt Abbildung 3.6.

3.2.5. Impostors

Impostors (sinngemäß: Attrappen) bezeichnen die Ersetzung eines komplexen Objektes durch ein Polygon mit einer passenden Textur. Die Reduktion der Objektkomplexität erfolgt also weniger durch geometrische Modifikation als durch Rendering des Objektes. Schaufler und Stürzlinger [21] stellen ein Verfahren zur Beschleunigung von Rendering jeglicher Art vor, daß auf einem hierarchischen Raumunterteilungsverfahren basiert und die Attrappen dynamisch bei Bedarf generiert, sowie einen Cache zur Wiederverwendung generierter Bilder realisiert.

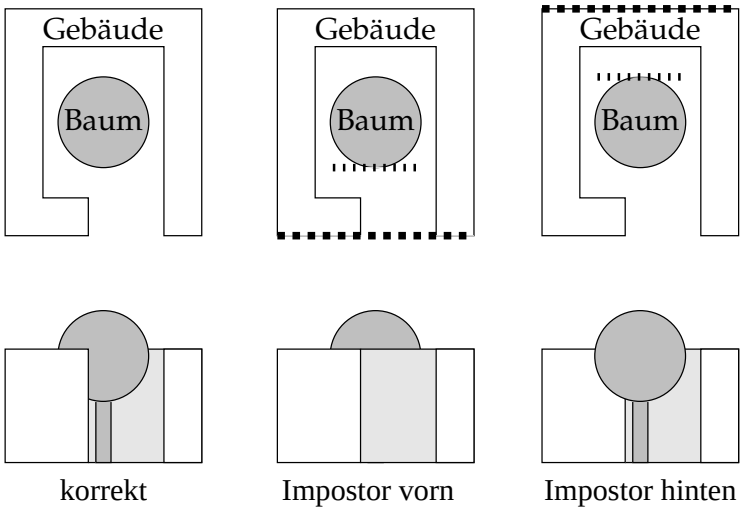


Abbildung 3.7.: Grenzen der Anwendung von Impostors

Impostor-Techniken sind auch für die vorliegende Anwendung denkbar. Es ergeben sich jedoch einige Probleme. So ist für die Generierung der Texturen hardwarebeschleunigtes Rendering wünschenswert, die Rechner des CLiC unterstützen dies jedoch nicht[6]. Auch ergeben sich durch den objektorientierten Ansatz (ein Reduktionsauftrag bezieht sich auf ein Objekt der

[6] Allgemein dürfte hardwareunterstütztes Rendering direkt in den Speicher nur sehr selten auf gängigen Plattformen verfügbar sein – bei Linux ist die 3D-Unterstützung zum Beispiel im Grafiktreiber des X11-Servers angesiedelt und steht somit nicht ohne grafische Oberfläche zur Verfügung.

Szene) zwangsläufig Artefakte, wie Abbildung 3.7 illustriert – ganz gleich, wie man die Polygone für die Attrappen von Haus und Baum platziert, es werden in vergleichbaren Konstellationen immer Objektteile fälschlich überdeckt. Schaufler und Stürzlinger sind von letzterem Problem nicht betroffen, da sie die Geometrie der gesamten Szene in die Raumunterteilung einbeziehen und nicht objektweise vorgehen.

Ein szene-globaler Reduktionsansatz kommt jedoch für VR-Simulationen nicht in Frage. Raumunterteilungsverfahren sind im Allgemeinen sehr rechenaufwändig und werden deshalb meist als Präprozessingschritt realisiert. Durch die in der VR-Simulation vorhandene Dynamik ändert sich praktisch die Geometrie der Szene während der Interaktion (wird ein Objekt innerhalb der Szene bewegt, so ändert sich aus globaler Sicht die Szenegeometrie). Für einen szene-globalen Reduktionsansatz würde also ein Raumunterteilungsverfahren benötigt, welches entweder echtzeitfähig ist – dies ist aufgrund der Szenekomplexität und angesichts existierender Verfahren unwahrscheinlich –, oder dessen Datenstrukturen inkremell aktualisierbar sind um die Veränderungen der Szene durch Interaktion nachzuvollziehen – ein entsprechendes Verfahren ist dem Autor derzeit nicht bekannt.

3.2.6. Impostors mit Tiefeninformation

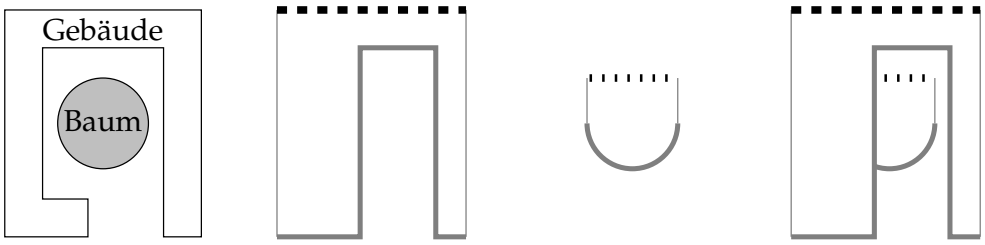


Abbildung 3.8.: Impostors mit Depth Map

Speichert man zu den Texturinformationen der Impostors zusätzlich Tiefeninformationen (eine Depth Map, siehe Abb. 3.8), so lassen sich die Impostors öfter wiederverwenden, indem mittels Relief-Texturing (siehe z.B. Hilbert [11] für eine Funktionsbeschreibung) weitere Ansichten generiert werden

können. Das Problem der Fehlüberdeckung kann jedoch erst durch die Verwendung sogenannter Nailboards gelöst werden. Dabei werden die Tiefenwerte aus der Depth Map entsprechend transformiert und in den z-Buffer des Renderers eingetragen. Dieses Vorgehen erfordert jedoch einen speziellen Renderer und ist deshalb hier nicht geeignet.

3.3. Reduktionsparameter

Tabelle 3.1 bietet einen Überblick über die Parameter der einzelnen Reduktionsalgorithmen, mit denen der Grad der Reduktion beeinflußbar ist. Leider beschäftigen sich nur Low und Tan sowie Rossignac und Borrel mit dem praktischen Einsatz ihrer Algorithmen zur Renderingbeschleunigung. Alle anderen Autoren sehen den Zweck ihrer Algorithmen vorwiegend im Präprozessing unter menschlicher Aufsicht.

Für die Clusteringalgorithmen bietet es sich an, die Größe der Rasterzellen so zu wählen, daß ihre Projektion auf die Bildebene einem Pixel entspricht. Low und Tan stellen dazu fest: „This criterion can generally be relaxed to allow a one-cell-width-to-X-pixel-width basis where $X > 1$. Experiments have shown that good values for X range from 5 to 10, depending on how well the perceptual information are preserved.“ [15] Ähnliche Überlegungen gelten für Algorithmen, die einen maximalen Fehler garantieren können und deren Fehlermetrik auf einem Abstand basiert. Man wählt den zulässigen Fehler so, daß der Abstand eines Eckpunktes im reduzierten Objekt zum Originalobjekt in der Bildebene kleiner als ein Pixel ist.

Rossignac und Borrel unterbreiten lediglich Vorschläge für den praktischen Einsatz ihres Algorithmus. Sie schlagen unter anderem vor, für Echtzeit-Interaktion vereinfachte Modelle zu verwenden und qualitativ hochwertiges Rendering später im Batch-Verfahren abzuwickeln. Auch auf Level-of-Detail-Techniken gehen sie ein, beschränken sich jedoch auf sehr allgemeine Aussagen:

> Simplification levels may be selected depending on the viewpoint. Models that are further away from the viewer are displayed with less details. The distance to the viewer may be estimated using precomputed spherical or other simple bounds for each object. [20]

Decimation nach Schroeder et al.[24] (Impl. vtkDecimatePro) • Reduktionsfaktor (nur garantiert für bestimmte Parameter, u.a.: Topologieänderung erlaubt, Maximalfehler nicht begrenzt) • Maximalfehler • diverse implementationsspezifische Parameter
Uniform Vertex Clustering nach Rossignac und Borrel[20] • Größe der Rasterzellen • Ausrichtung des Rasters
Floating Cell Clustering nach Low und Tan[15] • Größe der Zellen um Eckpunkte mit hohem Gewicht
Quadric Error Metrics nach Garland und Heckbert[6] (Impl. QSlim) • Reduktionsfaktor
Quadric Error Metrics nach Hoppe[12] (Impl. vtkQuadricDecimation) • Maximalkosten für Kantenkontraktion • maximale Anzahl zu kontrahierender Kanten
QEM mit Clustering nach Lindstrom[14] (Impl. vtkQuadricClustering) • Größe der Gitterzellen • diverse Parameter

Tabelle 3.1.: Reduktionsbeeinflussende Parameter der Algorithmen

Offen bleibt, wie man den Reduktionsfaktor für Algorithmen berechnet, die nur einen solchen Faktor als Parameter anbieten. Denkbar ist eine Heuristik, die für alle Polygone des Modells die Fläche in der Bildebene berechnet (oder schätzt) und dann vom Reduktionsalgorithmus nur die Anzahl der Polygone anfordert, die eine Fläche von mehr als einem Pixel einnehmen.

3.4. Zusammenfassung

Es existiert eine große Anzahl von Algorithmen zur automatischen Reduktion von polygonalen Modellen. Für die angestrebte Anwendung kommen jedoch nur solche in Betracht, die Eingabedaten beliebiger Topologie verarbeiten können. Sechs hier vorgestellte Algorithmen erfüllen dieses Kriterium, vier davon liegen in freien Implementationen vor und können deshalb praktisch getestet werden.

Problematisch gestaltet sich die Integration der Algorithmen in eine VR-Simulation, da die reduktionsbeeinflussenden Parameter oft nur heuristisch ermittelt werden können. Ein weiteres Problem stellen attributierte Modelle dar. Zwar beschäftigen sich sowohl Hoppe als auch Garland und Heckbert mit der Reduktion solcher Modelle, jedoch unterstützt einzig der Algorithmus PropSlim aus der Implementation QSlim die Reduktion mit Einbeziehung der Attribute. In der vorliegenden Version von vtkQuadricDecimation ist dies zwar vorgesehen, jedoch als nicht-funktional gekennzeichnet und deaktiviert. Letztendlich unterstützt QSlim jedoch nicht die komplexen, hierarchischen Oberflächenattribute von VR-Modellen – lediglich eine Textur, Normalen und Farben für Eckpunkte werden berücksichtigt. Die Reduktion von komplexen Modellen (z.B. eines Hauses), die verschiedenste Materialien und Texturen beinhalten ist mit den vorgestellten Verfahren nicht ohne weiteres möglich.

Trotz der genannten Probleme stellen die verfügbaren Implementationen eine gute Basis für praktische Experimente dar.

4. Effiziente Wiederverwendung von Reduktionen

Praktische Erfahrungen legen nahe, daß eine Steigerung der Simulationsgeschwindigkeit nur möglich ist, wenn die einmal berechneten Objektreduktionen effizient wiederverwendet werden können. So dauert etwa die Reduktion des Stanford Bunny mit dem verhältnismäßig schnellen vtkQuadricClustering Algorithmus auf einem PC mit Athlon 1.2 GHz Prozessor ca. 0,5 Sekunden. Die meisten anderen getesteten Implementationen benötigen dafür deutlich länger - bis zu mehreren Minuten.

Ein weiteres Argument für die Notwendigkeit effizienter Wiederverwendung liefert die begrenzte Netzwerkbandbreite. Wird als Zielplattform das HP Visualize Center zusammen mit dem CLiC für die Reduktion anvisiert, so stehen gerade einmal 100 MBit/s für den Empfang der reduzierten Objekte zur Verfügung. Die oben genannte Reduktion des Standford Bunny bestand aus 3196 Eckpunkten und 6437 Dreiecken. Allein für die Übertragung der Geometriedaten werden mindestens 131801 Byte benötigt (abgeleitet aus dem CORBA-Interfacebeschreibungen). Optimistisch gerechnet läßt sich diese Geometrie also nicht mehr als 94 mal pro Sekunde übertragen. Bei einer angestrebten Bildwiederholrate von 50 Bildern pro Sekunde lassen sich also pro Bild nicht einmal zwei Objekte mit ähnlichem Umfang empfangen.

Das Problem der effizienten Wiederverwendung läßt sich in zwei Bereiche unterteilen, die miteinander interagieren: Die Kriterien für die Wiederverwendbarkeit von reduzierten Objekten einerseits und die Verwaltung der reduzierten Objekte (dem Objektcache) andererseits.

4.1. Kriterien für die Wiederverwendbarkeit

Ziel der Vereinfachung von geometrischen Objekten ist eine Einsparung von Rechenaufwand für das Rendering bei de facto gleichbleibendem visuellen Eindruck. Der Betrachter soll möglichst nicht bemerken, daß das Objekt modifiziert wurde. Geht man davon aus, daß in einer gegebenen Situation eine adäquate Reduktion eines Objektes berechnet wurde, so stellt sich die Frage, welche Parameter die Wiederverwendbarkeit des Objektes beschränken. Zum Beispiel erübrigt sich die Neuberechnung der Reduktion, wenn die VR-Simulation völlig statisch ist, d.h. keinerlei Veränderungen stattfinden.

Im Folgenden werden einige Parameter und deren Plausibilität vorgestellt.

4.1.1. Objektgeometrie

In jedem Fall ist die Reduktion eines Objektes von der Objektgeometrie abhängig. Dies bedeutet jedoch nicht, daß sich die Geometrie der Reduktion zwangsläufig ändern muß. Ändert sich zum Beispiel nur ein kleines Detail (z.B. wenn eine Türklinke bewegt wird), so kann dies unter Umständen keine Auswirkung auf die Reduktion haben (z.B. weil der Reduktionsalgorithmus die Türklinke als nicht-sichtbares Detail klassifiziert und nicht weiter berücksichtigt).

Da über die Eigenschaften des Reduktionsalgorithmus keine generellen Aussagen getroffen werden können, muß im Allgemeinen davon ausgegangen werden, daß alle Reduktionen eines Objektes hinfällig sind, sobald die Geometrie des Originalobjektes geändert wird. Da die aktuelle Version der CADaVR-Schnittstelle keine Geometriemodifikationen zuläßt, ist diese Möglichkeit nicht implementiert.

Zur Geometrie des Originalobjektes können auch die lokalen Transformationen aller seiner Kinder gezählt werden – findet „innerhalb“ des Objektes (d.h. im Teilgraph, der das Objekt als Wurzel hat) eine Transformationsänderung statt, so entspricht dies einer Änderung der Geometrie des Originalobjektes, da die Reduktionsaufträge immer für Teilgraphen des Szenegraph vergeben werden. Diese Art der Änderung wird von der Implementation korrekt unterstützt.

4.1.2. Relative Position zum Betrachterstandpunkt

Die Position eines Objektes aus Sicht des virtuellen Betrachters ist das offensichtlichste Kriterium für die Wiederverwendbarkeit. Befindet sich der Betrachter „nah“ an einem Objekt, so nimmt dessen visuelle Repräsentation eine größere Fläche ein, wird also durch mehr Pixel repräsentiert und erlaubt somit eine detailiertere Darstellung. Ist der Betrachter dagegen „weit“ vom Objekt entfernt, wird es durch weniger Pixel repräsentiert und es können somit aufgrund des festen Pixelrasters weniger Details abgebildet werden.

Wird für eine gegebene Konstellation von Objekt und Betrachter eine geeignete Reduktion berechnet, so gilt es zu ermitteln, für welche anderen Konstellationen diese auch adäquat ist. Es liegt nahe, einen Bereich im Kameraraum für die Position des Objektes zu spezifizieren, in dem sich die Reduktionsparameter nicht signifikant ändern.

Dieses Kriterium für die Wiederverwendbarkeit wurde in der Studienarbeit [25] unter Verwendung einer Bounding Box realisiert (siehe Abb. 4.1). Die Reduktionskomponente lieferte zur reduzierten Geometrie eine Bounding Box, in der sich das Objekt befinden muß, damit die Reduktion theoretisch wiederverwendet werden kann – praktisch war dies nicht implementiert. Retrospektiv erscheint die Nutzung einer Bounding Box nicht sinnvoll, da keinerlei Vorteile gegenüber dem gängigen Ansatz für LOD-Verfahren zu erkennen sind, der die einzelnen Detailstufen eines Objektes entsprechend der Entfernung zur Kamera auswählt.

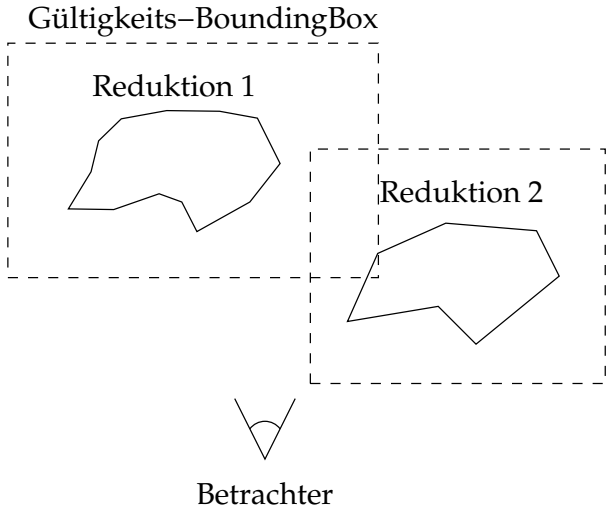


Abbildung 4.1.: Einschränkung der Gültigkeit durch Bounding-Box

Im Gegenteil hat die Verwendung einer Bounding Box als Gültigkeitsbereich gravierende Nachteile. So läßt sich z.B. eine Entfernungsabhängigkeit nicht modellieren. Weiterhin ergeben sich große Probleme bei der Organisation des Cache – für das effiziente Auffinden einer geeigneten Reduktion müßte in der Menge der Gültigkeits-Bounding-Boxes eine mehrdimensionale Suche durchgeführt werden. Gaede und Günther stellen in [5] fest: „[...] at present no access method has proven itself to be superior to all it's competitors in whatever sense." Und obwohl eine Reihe von Algorithmen bekannt sind, die für diese Suche in Betracht kämen, sind diese alle kompliziert zu implementieren. Dazu kommt, daß es sich hier um eine ausgesprochen anspruchsvolle Anwendung handelt, da sich die Bounding Boxes überlappen können und außerdem die Forderung nach effizientem Einfügen *und* Löschen neuer Bounding Boxes gestellt wird – viele der in [5] vorgestellten Algorithmen wurden für Datenbanken entworfen und stellen somit eher effizientes Suchen und Einfügen in den Vordergrund.

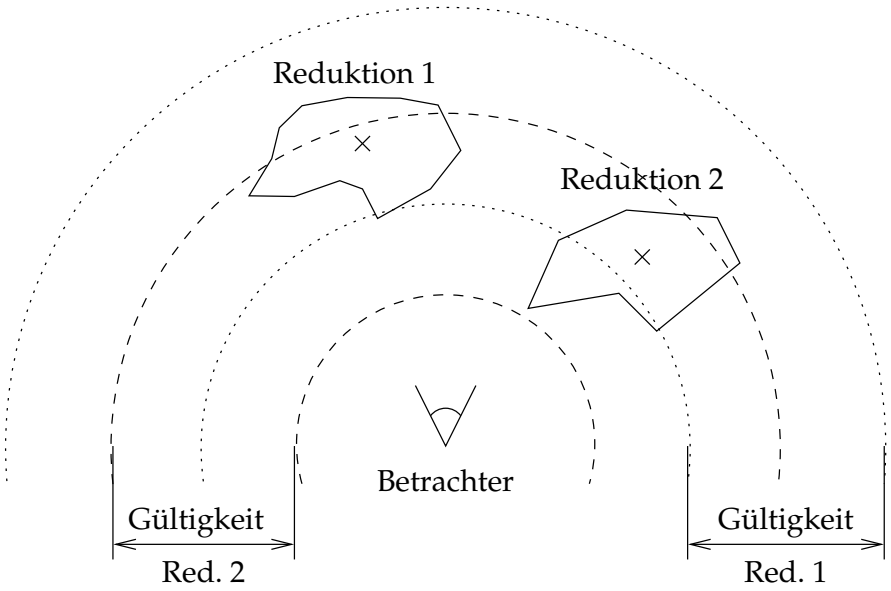


Abbildung 4.2.: Einschränkung der Gültigkeit durch Abstand

Für jedes reduzierte Objekt kann die Reduktionskomponente also einen Entfernungsbereich spezifizieren, in dem die Reduktion gültig ist (siehe Abbildung 4.2). Der Abstand zum Betrachter wird dabei ausgehend vom Mittelpunkt der Bounding Box des Originalobjektes berechnet.

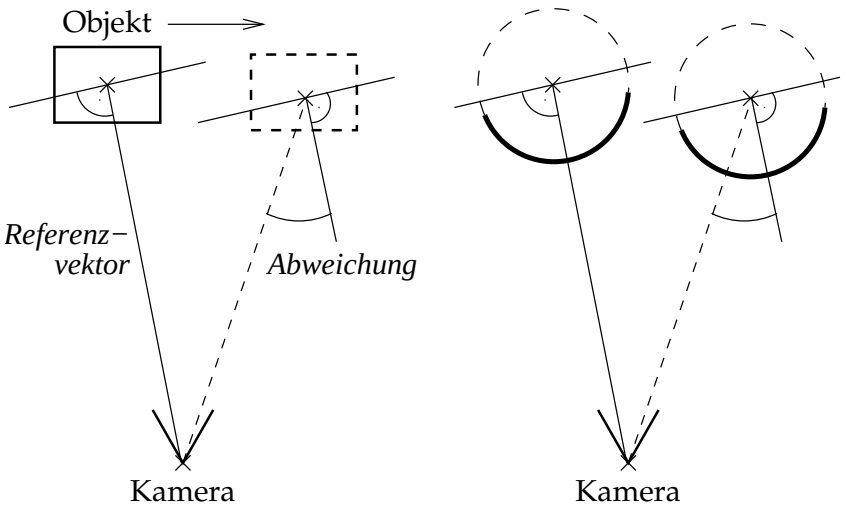
4.1.3. Blickwinkel auf das Objekt

Obwohl die in 3.2 vorgestellten Verfahren zur Objektreduktion sämtlich unabhängig von einem virtuellen Betrachter operieren, existieren Algorithmen, die den Betrachter in die Reduktion einbeziehen. Stellvertretend sei hier der Ansatz von Xia und Varshney [30] genannt. In einem Präprozessing-Schritt wird ein sogenannter Merge-Tree berechnet, der dann beim Rendering verwendet wird, um abhängig vom Betrachterstandpunkt und der Beleuchtungssituation den Detailgrad der Objektbereiche zu bestimmen. So werden die Silhouette und Gebiete mit starken Beleuchtungsänderungen mit mehr Details (d.h. Eckpunkten und Dreiecken) versehen während zum Beispiel Rückseiten stark vereinfacht werden. Da dieser Ansatz auf ein enges Zusammenspiel mit dem Renderer angewiesen ist, kommt dieser Algorithmus hier nicht in Frage.

Es ist jedoch klar, daß eine Änderung des Blickwinkels auf das Objekt bei ähnlichen Algorithmen einen Einfluß auf die Reduktion hat und somit das reduzierte Objekt nur für gewisse Blickwinkel verwendbar ist. Eine andere Art von Blickwinkelabhängigkeit kann das Verfahren vtkQuadricClustering (siehe Abschnitt 3.2.4) bedingen. Das Raster für die Clusterzellen könnte optimalerweise parallel zu den Koordinatenachsen im Kamerakoordinatensystem ausgerichtet werden, um möglichst wenig Artefakte beim Rendering zu provozieren.

Um eine maximale Abweichung vom Blickwinkel definieren zu können stellt sich zunächst die Frage, wie der Blickwinkel auf ein Objekt definiert werden kann. Es bietet sich an, daß die Reduktionskomponente einen Referenzvektor errechnet, der 0 Grad Blickwinkelabweichung definiert (siehe Abbildung 4.3(a)). Weiterhin erscheint es sinnvoll, diesen Vektor per Definition auf den Mittelpunkt der Bounding Box des Originalobjektes zu beziehen. Alternativ könnte der Referenzvektor implizit dem Vektor Kamera – Bounding-Box-Mittelpunkt entsprechen, dies bedeutet allerdings zusätzliche Schwierigkeiten wenn die Reduktion asynchron zum Rendering abläuft, da dieser Vektor dann für jeden ausstehenden Reduktionsauftrag gespeichert und nach Eintreffen des Ergebnisses der Reduktion zugeordnet werden müsste.

Die Abweichung vom Referenzvektor wird innerhalb des Weltkoordinatensystems berechnet. Sie ist damit also unabhängig von der Orientierung der Kamera.



(a) Berechnung der Winkelabweichung

(b) Blickwinkelabhängigkeit durch Rückseitenentfernung

Abbildung 4.3.: Einschränkung der Gültigkeit durch Blickwinkel

4.1.4. Beleuchtungsverhältnisse

Wenn das Reduktionsverfahren Renderingschritte beinhaltet, so ist das reduzierte Objekt abhängig von der aktuellen Beleuchtungssituation. Da die aktuelle Beleuchtungssituation eine große Menge von Parametern beinhaltet (Position, Status und Farbe aller Lichtquellen), wird nicht versucht, beleuchtungsabhängige Reduktionen wiederzuverwenden. Stattdessen werden diese sofort verworfen, wenn sich die Beleuchtungssituation ändert. Es genügt als Kriterium also die Information, ob die Reduktion von der Beleuchtung abhängig ist.

4.1.5. Bildparameter

Das Ziel der Reduktion ist die Entfernung von Details, die durch den Prozess der Rasterisierung auf diskrete Bildpunkte sowieso verloren gehen würden und somit nichts oder nicht signifikant zum visuellen Eindruck beitragen. Die Reduktionskomponenten erhalten Kenntnis über die Kameratransformation sowie die Größe des zu rendernden Bildes, um die reduzierten Objekte optimal an die aktuellen Anzeigeverhältnisse anpassen zu können.

Ändert sich die Bildgröße oder die Projektionstransformation, so ist davon auszugehen, daß die bisherigen Reduktionen nicht mehr optimal auf die Bildgeometrie zugeschnitten sind. Wird das Bild verkleinert, so enthalten die Reduktionen potentiell zu viele Details; wird das Bild vergrößert, so enthalten die Reduktionen potentiell zu wenige Details. Im ersten Fall wird unnötig Rechenzeit für die Darstellung der nicht sichtbaren Details verwendet, im letzteren Fall nimmt die Qualität der Darstellung ab.

In diesem Kontext stellt sich die Frage, wie bei Stereo-Bildern die beiden Teilbilder behandelt werden können. Es bietet sich an, für das linke bzw. rechte Bild jeweils die Kameraposition entsprechend anzupassen, so daß das Rendering eines Stereo-Bildes auf das Rendering zweier monoskopischer Bilder zurückgeführt wird. In diesem Fall können die Cache-Mechanismen optimal greifen und es ist außer bei blickwinkelabhängigen Reduktionen nicht mit Mehraufwand für die Berechnung zusätzlicher Reduktionen zu rechnen.[1]

4.2. Cacheverwaltung

Die Verwaltung von Caches ist ein sehr komplexes Thema. Es läßt sich hier in folgende Teilprobleme unterteilen:

- Lokation des Cache
- Größe des Cache
- Auffinden passender Reduktionen im Cache
- Freigabe von Cachespeicher

4.2.1. Cachelokation

In einer verteilten Applikation ist es wünschenswert, den insgesamt in der Applikation verfügbaren Arbeitsspeicher optimal zu nutzen. Ein verteilter Cache wäre also optimal. Aufgrund der beschränkten Bandbreite zum Renderknoten ist dies jedoch nicht praktikabel. Der Cache muß also auf dem Renderknoten gehalten werden. Unabhängig davon könnten natürlich die

[1] Die stereoskopische Darstellung wurde nicht implementiert.

Reduktionskomponenten eigene Caches verwalten, insofern die beschriebenen Kriterien für die Wiederverwendbarkeit beachtet werden[2]. Dieser Ansatz wird in Abschnitt 6.2.5 (Seite 66) näher diskutiert.

4.2.2. Cachegröße

Optimal wäre eine dynamische Anpassung der Cachegröße an den verfügbaren freien Speicher. Praktisch läßt sich der freie physikalische Speicher (d.h. ohne virtuellen Speicher, der mit Hilfe der Festplatten emuliert wird) unter Unix-artigen Betriebssystemen nicht zuverlässig feststellen (es existiert nicht einmal ein standardisierter Systemaufruf um die Menge des verfügbaren Speichers zu bestimmen[3]). Außerdem ist nicht ausgeschlossen, daß ein anderer Teil der VR-Simulation plötzlich größere Speichermengen anfordert (es könnte z.B. eine physikalische Simulation gestartet werden). Es kann also leider keine optimale Cachegröße geben, sondern diese muß manuell vorgegeben (und eventuell experimentell ermittelt) werden.

Für die manuelle Vorgabe der Cachegröße bieten sich zwei Parameter an: Anzahl der Objekte im Cache und maximaler Speicherbedarf der Objekte im Cache. Der Speicherbedarf eines reduzierten Objektes ist jedoch sehr schwierig zu ermitteln, da ein Objekt aus mehrere Instanzen von Klassen der SceneGraph-Bibliothek zusammengesetzt ist. Als einfache Heuristik könnte man alternativ die Punkte, Polygone und Texturen in Betracht ziehen, die zum Objekt gehören. Im ungünstigsten Fall ist das Originalobjekt – und damit potenziell auch seine Reduktion – so komplex, daß nur ein Objekt vom Cache aufgenommen wird.

Um die Implementation möglichst einfach und übersichtlich zu halten, wird die Cachegröße durch die Anzahl der Objekte im Cache beschränkt.

[2] In der Tat versucht der Scheduler, einen Auftrag vorzugsweise dem Reduzierer zuzuteilen, der zuletzt eine Reduktion für das angeforderte Objekt erzeugt hat. Somit ist bei ausreichender Anzahl von Reduzierern eine Affinität von Originalobjekt zu Reduzierer gegeben. Die auf VTK-Implementationen basierenden Reduktionskomponenten nutzen diesen Umstand, um eine erneute Konvertierung der Geometriedaten in die VTK-Datenstrukturen zu vermeiden.

[3] Letztendlich wäre mit einer Möglichkeit, den verfügbaren Speicher zu bestimmen, nicht viel gewonnen, da dies den Programmierer nur zu unzuverlässigen Heuristiken verleitet. Unix-artige Systeme sind in der Mehrzahl Multiuser- und Mulittaskingsysteme mit unterschiedlichsten Einsatzzwecken. Es ist daher im Allgemeinen günstiger, die Zuteilung von speziellen Pufferspeichern manuell durch den Administrator konfigurieren zu lassen.

4.2.3. Auffinden passender Reduktionen

Das nächste Teilproblem des Objektcaching ist das Auffinden einer Reduktion passend für den aktuellen Zustand der VR-Simulation. Einfach und effizient für Caches mit wenigen Objekten ist die lineare Suche. Für größere Caches wären jedoch Verfahren mit asymptotisch logarithmischer Laufzeit wünschenswert.

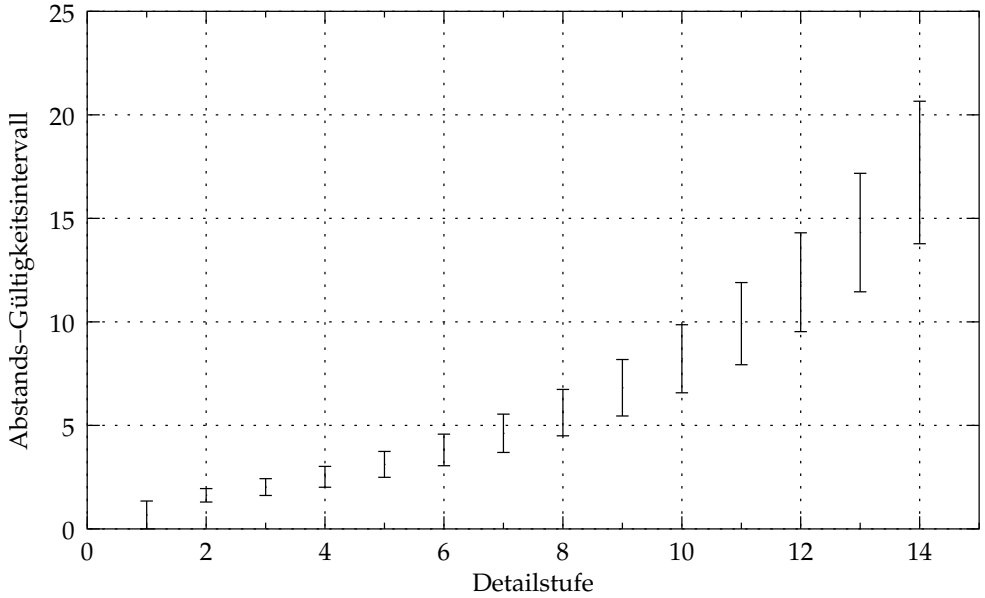


Abbildung 4.4.: Beispiel für Abstands-Gültigkeitsintervalle (Standford-Bunny, reduziert mit QuadricClusteringReducer)

Es bietet sich ein zweistufiges Suchverfahren an: Zuerst werden alle Reduktionen ermittelt, die nach dem Abstands-Kriterium in Frage kommen und diese werden dann entsprechend ihrem Blickwinkel-Kriterium getestet. Die Suche innerhalb der Intervalle beginnt beim Intervall, dessen untere Grenze am nächsten unterhalb des derzeitigen Abstandes liegt. Damit ist gewährleistet, daß ein möglichst hoher Reduktionsgrad verwendet wird, außerdem müssen dann nur die Intervalle geprüft werden, deren untere Grenze noch niedriger ist. Die Intervalle können also in einem Suchbaum, geordnet nach ihren unteren Grenzen organisiert werden.

Obwohl Abb 4.4 nahelegt, daß die Suche abgebrochen werden kann, sobald die obere Intervallgrenze den aktuellen Abstand unterschreitet, kann dies nicht vorausgesetzt werden. Es wäre zum Beispiel denkbar, daß ein Algorithmus für einen anderen Blickwinkel ein Abstandsintervall mit höherer

Obergrenze geliefert hat – die Reduktionen in Abb. 4.4 waren als blickwinkelunabhängig gekennzeichnet.

Derzeit wird die erste Reduktion verwendet, deren Kriterien der aktuellen Betrachtungssituation genügen. Wünschenswert wäre hier eine vom Anwender änderbare Präferenz der einfachsten bzw. detailliertesten den Kriterien genügenden Reduktion. Da jedoch kein Komplexitätsmaß für die vorliegenden Reduktionen verfügbar ist, muß darauf verzichtet werden.

4.2.4. Freigabe von Cachespeicher

Auch die Freigabe von Cachespeicher ist im Allgemeinen ein komplexes Thema. Die vorliegende Implementation gibt bei Überschreitung der maximalen Cachegröße die „ältesten“ Reduktion frei. Optimal wäre jedoch eine Gewichtung der Reduktionen im Cache nach der Wahrscheinlichkeit, daß sie in Kürze benötigt werden. Dafür wäre jedoch eine Metrik notwendig, die aufgrund der verschiedenen (und optionalen) Verwendungskriterien sehr kompliziert zu definieren und zu berechnen wäre. Diese Metrik könnte außerdem für die Ermittlung einer vorläufig-optimalen Reduktion nützlich werden, die verwendet wird, während die optimale Reduktion noch berechnet wird.

Eine weitere Strategie für die Cachefreigabe ist die sogenannte LRU-Strategie (Least Recently Used, am Längsten nicht benutzt). Dabei wird für jeden Cacheeintrag ein Zeitstempel für die letzte Verwendung mitgeführt und dann werden zuerst die Cacheeinträge mit dem niedrigsten Zeitstempel freigegeben. Die LRU-Strategie stellt eine Heuristik für die oben genannte Wahrscheinlichkeit dar.

Für die LRU-Strategie läßt sich recht einfach ein Worst-Case-Szenarion konstruieren. Der Nutzer bewegt sich zunächst aus großer Entfernung x auf ein Objekt zu, passiert das Objekt (die für Entfernung x passende Reduktion wird aus dem Cache entfernt), läßt es hinter sich und entfernt sich wieder davon (es werden keine Reduktionen benötigt und damit auch keine Zeitstempel aktualisiert, da mittels View Frustum Culling das Objekt zeitig als nicht-sichtbar klassifiziert wird). Dann dreht sich der Nutzer um 180 Grad und blickt wiederum aus Entfernung x auf das Objekt. Jetzt wird für Entfernung x eine neue Reduktion erzeugt, jedoch die nächstdetailliertere Reduktion aus dem Cache entfernt. Bewegt sich der Nutzer nun erneut auf das Objekt zu, so wird die jeweils als übernächstes benötigte Reduktion durch die

als nächstes benötigte aus dem Cache verdrängt. Die Folge wäre eine starke Verlangsamung der Simulation. Inwiefern diese Heuristik für die vorliegende Anwendung sinnvoll ist ließe sich mit Untersuchungen über das Bewegungsverhalten von Nutzern in VR-Simulationen abschätzen.

4.3. Zusammenfassung

Ohne effiziente Wiederverwendung einmal berechneter Objektreduktionen dürfte im vorliegenden Anwendungsfall keine Beschleunigung der VR-Simulation zu erreichen sein. Hauptargumente für diese Annahme sind der Zeitbedarf für die Erzeugung der Reduktionen und die begrenzte Netzwerkbandbreite.

Die Wiederverwendbarkeit von reduzierten Objekten kann von der Position des Objektes relativ zum Betrachterstandpunkt, dem Blickwinkel auf das Objekt (im Koordinatensystem der Kamera oder der Welt) und den Beleuchtungsverhältnissen abhängen. In jedem Fall ist die Reduktion abhängig von der Geometrie des Originalobjektes.

Aufgrund der recht komplexen Kriterien für die Wiederverwendbarkeit von Objektreduktionen gestaltet sich die Cacheverwaltung schwierig. In der vorliegenden Implementation wurde Wert auf Einfachheit und Übersichtlichkeit gelegt. Die Suche nach passenden Objektreduktionen erfolgt zweistufig. Zuerst werden alle Reduktionen mit passenden Entfernungskriterium ermittelt und diese werden dann auf das Blickwinkelkriterium getestet. Es wird die erste Reduktion verwendet, die gültige Kriterien aufweist.

5. Systemarchitektur

5.1. Vorüberlegungen

Höherer Realismus in Virtual Reality Anwendungen geht im Allgemeinen mit höherer Komplexität der Simulation einher. Je detaillierter ein Objekt innerhalb einer Simulation dargestellt werden soll, desto komplexer wird seine Modellierung. VR-Simulationen beinhalten also sehr komplexe Objekte.

Da der Aufwand für die Darstellung eines Objektes direkt von dessen Komplexität abhängt, d.h. in diesem Fall von der Anzahl der Eckpunkte und Polygone, erreicht eine umfangreiche Simulation schnell die Grenzen der verfügbaren Technik. Je umfangreicher eine Simulation ist (im Sinne von räumlicher Ausdehnung der simulierten Realität und somit Objektanzahl), desto häufiger treten Objekte nur als Randerscheinungen auf, d.h. sie befinden sich am Rande oder außerhalb des gegenwärtigen Fokus des Benutzers, wie zum Beispiel Bäume, wenn sich der Nutzer innerhalb eines Hauses bewegt und diese somit nur vereinzelt oder weit entfernt wahrnimmt. Weiterhin wird durch die Rasterisierung im Renderingprozess die Menge der darstellbaren Details eingeschränkt – im ungünstigsten Fall befindet sich die Abbildung eines oder mehrerer Objekte innerhalb eines einzelnen Pixels und ist somit nicht von einem Polygon mit passender Oberflächenfarbe zu unterscheiden.

5.1.1. Dynamisches Level-of-Detail

Ziel vieler Optimierungen für VR-Simulationen ist es also, Berechnungen möglichst zu vermeiden, die auf den Simulationseindruck keine Auswirkungen haben. So ist es zum Beispiel üblich, mittels View Frustum Culling sehr früh im Renderingprozess Objekte auszusortieren, die sich vollständig

außerhalb der Sichtpyramide des Betrachters (dem View Frustum) befinden. Wird für ein Objekt festgestellt, daß es potenziell sichtbar ist, so erscheint es sinnvoll, für die Darstellung eine Repräsentation des Objektes zu wählen, die für die aktuelle Betrachtungssituation den geringsten Rechenaufwand benötigt. Level-of-Detail (LOD) mit einer statischen Anzahl von Detailstufen ist ein dafür weit verbreitetes Verfahren (siehe Abb. 5.1).

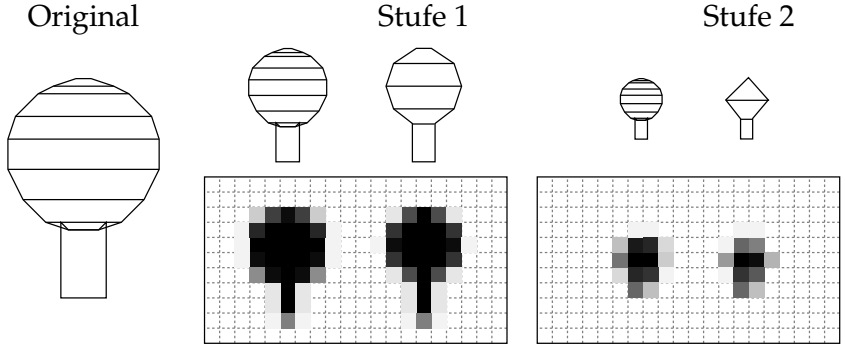


Abbildung 5.1.: LOD für ein einfaches Objekt

Eine adaptive Generierung von Detailstufen bietet sich als Alternative an, da dann aufwändiges Präprozessing oder die manuelle Modellierung der Detailstufen entfällt. Weiterhin erlaubt eine bedarfsorientierte Erzeugung der vereinfachten Objektrepräsentation eine optimale Anpassung an die jeweilige Betrachtungssituation. Die Auswahl der Objektrepräsentation bei statischem LOD erfolgt in der Regel in Abhängigkeit vom Abstand der Kamera zum Objekt. Diese Entfernung wird jedoch innerhalb des stetigen Modellraumes ermittelt und ist unabhängig von der diskreten Ausdehnung des rasterisierten Objektes im Bildraum. Während also eine Detailstufe für eine bestimmte Bildgröße (z.B. Handy-Display mit 160×100 Pixeln) optimal ist, kann sie für ein Bild mit deutlich mehr Pixeln (z.B. Mehrsegmentprojektion mit 2560×768 Pixeln) inadäquat sein, da sie viel zu grob ist (siehe Abbildung 5.2).

5.1.2. Parallele Generierung der Detailstufen

Zur Generierung der Detailstufen (Reduktion) können die in Kapitel 3 vorgestellten Verfahren verwendet werden. Diese Algorithmen sind sehr rechenaufwändig. Deshalb bietet sich eine Wiederverwendung einmal berechneter Vereinfachungen an. Weiterhin ist die Reduktion optimal paral-

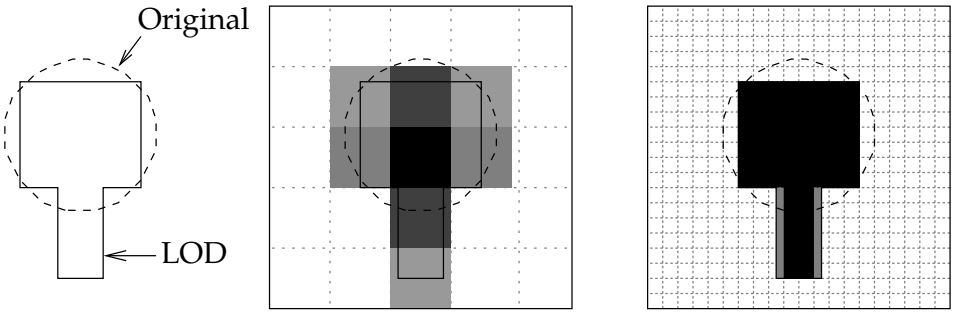


Abbildung 5.2.: Statisches LOD berücksichtigt die Bildgeometrie nicht

lelisierbar, da sie objektweise vorgenommen wird. Es werden also Reduktionsaufträge formuliert, die von einer parallelen Applikation (oder Komponente) bearbeitet werden.

Die parallele Komponente wiederum wird in zwei Bestandteile zerlegt: Die Verteilung der Aufträge und die Verarbeitung der Aufträge. Diese Unterteilung hat den Vorteil, daß die parallele Applikation unabhängig von der Simulation laufen und auf Aufträge warten kann und somit der Start des Simulationsprogrammes vereinfacht wird.

5.1.3. Verteilung der Szenedaten

Für die Reduktion eines Objektes wird dessen Originalgeometrie benötigt. Sie muß also dem parallelen Programm zugänglich gemacht werden. Dabei stellt sich die Frage, ob grundsätzlich jede Reduktionskomponente eine Kopie des gesamten Szenegraph erhält oder nur die Teile, die für die Reduktion erforderlich sind. Eine weitere Möglichkeit ist die gleichmäßige Verteilung von Szenegraphteilen über alle Reduktionskomponenten. (Siehe dazu Abbildung 5.3.)

Bestimmte Teile des Szenegraph werden prinzipiell immer für die Reduktion benötigt (z.B. um die Position des Objektes im Kameraraum zu bestimmen), also ließen sich im einfachsten Fall lediglich die Geometriedaten nur bei Bedarf zuteilen. Optimalerweise (aber auch am kompliziertesten) wäre die Verteilung von Szenegraphteilen und Geometrie genau nach Bedarf.

Von Vorteil wäre die bedarfsorientierte Zuteilung der Daten, wenn der Speicherbedarf der Geometriedaten die Ressourcen der Reduktionsrechner überschreitet (die Reduktionsverfahren benötigen oft grosse Speichermengen für die Reduktion). Praktisch werden jedoch bei Speicherknappheit

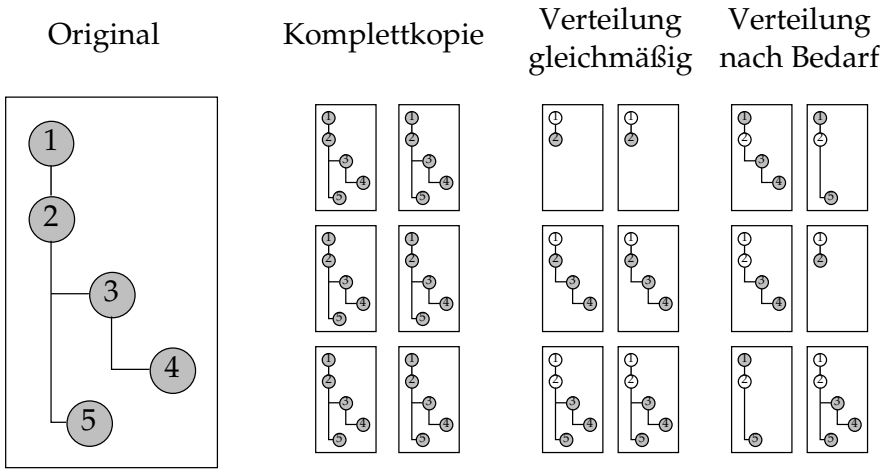


Abbildung 5.3.: Szenedatenverteilung auf die Reduktionskomponenten

nicht benötigte Teile des Szenegraph vom Betriebssystem ausgelagert, so daß durch eine geschickte Zuteilung der Reduktionsaufträge der vom Betriebssystem simulierte (virtuelle) Arbeitsspeicher mit genutzt werden kann (im Falle der Knoten des CLiC stehen damit 1 GB Speicher pro Rechner zur Verfügung).

Versucht man, Speicherknappheit auf den Reduktionsrechnern durch eine bedarfsorientierte Verteilung der Geometriedaten zu vermeiden, so wirft dies eine Reihe von Fragen auf, die zum Teil bereits in Abschnitt 4.2.2 (Seite 42) diskutiert wurden. Das grundlegende Problem besteht in der Abschätzung des verfügbaren Speichers einerseits und des Speicherbedarfs der zu verteilenden Daten andererseits – es handelt sich um hierarchische Datenstrukturen. Zudem würde für die Kontrolle der Verteilung ein recht umfangreiches Protokoll benötigt. Entweder übernimmt die zentrale Auftragsverteilung diese Aufgabe, dann muß sie den Reduktionskomponenten u.a. mitteilen, welche Teile des Szenegraph zu löschen sind um anderen Platz zu schaffen, oder die Reduktionskomponenten müssen von der Auftragsverteilung Szenedaten anfordern.

Hier wird die Variante „Komplettkopie“ gewählt (siehe Abb. 5.3), da sie am einfachsten zu implementieren ist und keine gravierenden Nachteile hat. In allen anderen Varianten entsteht ein zusätzlicher Kommunikations- und Implementationsaufwand, der durch simples einmaliges Versenden des kompletten Szenegraph vollständig vermieden wird.

5.1.4. Verarbeitung der Reduktionsergebnisse

Die Reduktionskomponenten senden die Ergebnisse der Reduktion direkt zum Auftraggeber, also der Simulation. Diese integriert die neu empfangene Repräsentation des Objektes in den Szenegraph als weitere Detailstufe. Wie bereits in Kapitel 4 auf Seite 35 erläutert, läßt sich eine Steigerung der Simulationsgeschwindigkeit nur durch eine Wiederverwendung bereits berechneter Detailstufen erreichen. Die Detailstufen werden also gleichzeitig in einen Cache zur späteren Wiederverwendung aufgenommen.

5.2. Logische Architektur

Mit der Aufnahme der reduzierten Objekte in den Cache schließt sich der Kreis der in Abbildung 5.4 gezeigten logischen Architektur für die verteilte Objektreduktion.

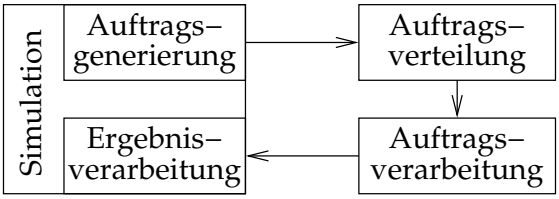


Abbildung 5.4.: Logische Komponenten der Reduktion

Die verteilte Objektreduktion findet also in einem Kreislauf statt: Die Simulation fordert Reduktionen an, erzeugt also Aufträge, die von der Auftragsverteilung möglichst geschickt auf die mehrfach vorhandenen Verarbeitungskomponenten verteilt werden. Die Ergebnisse der Verarbeitung (die reduzierten Objekte) werden wiederum direkt zur Simulation gesendet.

5.3. Modularisierung

5.3.1. Modularisierung der verteilten Reduktion

Die Auftragsgenerierung und die Ergebnisverarbeitung arbeiten eng zusammen, da letztendlich nur neue Aufträge generiert werden, wenn noch keine passenden Ergebnisse (d.h. Reduktionen mit passenden Kriterien)

vorliegen. Die Modularisierung der verteilten parallelen Applikation entspricht der logischen Architektur: Die Auftragsgenerierung und Ergebnisverarbeitung sind Teil der Komponente DynVLod_CadvisPlugin. Die Auftragsverteilung ist in der Komponente ReducerScheduler realisiert, die Auftragsverarbeitung in ObjectReducer und davon abgeleiteten Programmen.

5.3.2. Modularisierung der Simulation

Die Modularisierung im Gesamtsystem gestaltet sich etwas komplexer. Um eine hohe Wiederverwendbarkeit zu erreichen, ist die Simulation in sich unterteilt (siehe Abb. 5.5).

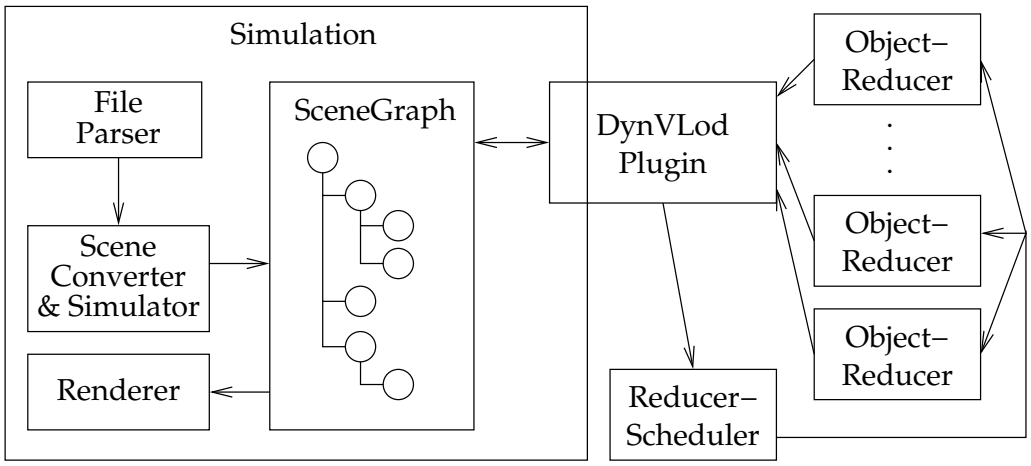


Abbildung 5.5.: Modularisierung

Der erste Schritt zur Modularisierung besteht in der Entkopplung von Simulation und Eingabedaten. Ziel dieser Entkopplung ist die Unterstützung verschiedenster Eingabeformate. Derzeit wird primär das CADaVR-Format (siehe [29]) eingesetzt. Dieses Dateiformat wird von einem Parser (Modul Parser) in Datenstrukturen überführt. Da diese Datenstrukturen natürlich vom Dateiformat abhängen, wird eine formatunabhängige Repräsentation der Szene (Modul SceneGraph) benötigt. Durch diesen zentralen, formatunabhängigen Szenegraph ist eine gute Wiederverwendbarkeit gegeben.

Für die ebenfalls dateiformatabhängige Szenedynamik (z.B. Interaktionsmöglichkeiten) wird ein weiteres Modul (CadavrSimulator) eingeführt. Ein Teil dieses Moduls ist auch für die Überführung der dateiformatspezifischen Datenstrukturen in den zentralen Szenegraph zuständig. Angesichts

der Komplexität der CADaVR-Schnittstelle erscheint es nicht sinnvoll, den Simulator selbst zu abstrahieren, stattdessen sind in SceneGraph allgemeine Schnittstellen integriert, die für das Simulationsmodul benötigt werden.

Im letzten Schritt wird auch der Renderer ausgegliedert. Dies vereinfacht spätere Experimente mit alternativen Renderern, die z.B. Schattenberechnung unterstützen könnten. Außerdem wird somit eine grunsätzliche Abhängigkeit des Simulationsprogrammes vom derzeit verwendeten World Toolkit (WTK) vermieden und es können zukünftig auch andere Renderingbibliotheken zum Einsatz kommen, wie zum Beispiel OpenSceneGraph (`http://www.openscenegraph.org`) oder MAVERIK (`http://www.gnu.org/software/maverik/`).

5.3.3. Trennung von Simulation und verteilter Reduktion

Die Simulation ist vollständig von der verteilten adaptiven Objektreduktion entkoppelt. Damit wird Funktionalität ausgelagert, die nicht zum Kern der VR-Simulation gehört und somit weitere wissenschaftliche Untersuchungen auf Basis der VR-Simulation (ohne Reduktion) erschweren könnte. Ein weiterer Grund für die Entkopplung von Simulation und Reduktion ist die Abhängigkeit der Reduktion von CORBA, dessen Implementationen im Allgemeinen recht umfangreich sind. Letztendlich ist die verteilte Reduktion kein essentieller Bestandteil der Simulation und sollte deshalb rein prinzipiell nicht zu stark damit verbunden sein.

5.4. Physische Architektur

Die physische Architektur des Systems wird vorwiegend von der Netzwerk-Infrastruktur und der Verteilung der Rechenkapazitäten bestimmt. Das HP Visualize Center II im VR-Labor der Professur für Graphische Datenverarbeitung und Visualisierung wird als Renderingplattform verwendet. Es ist mittels Fast-Ethernet mit dem Gigabit-Backbone des Universitätsrechenzentrums (URZ) verbunden. Zur Reduktion der Geometriedaten bietet sich der Einsatz von Knoten des CLiC an, da diese speziell für paralleles Rechnen installiert wurden. Es können praktisch jedoch auch beliebige andere Rechner verwendet werden, die per Netzwerk erreichbar sind.

Somit liegen die Lokationen von Simulation, Auftragsgenerierung und Ergebnisempfang nahe. Die Auftragsverteilung wird optimalerweise auch auf

einem Knoten des CLiC angesiedelt, da diese Komponente als Multiplikator auftritt, indem sie globale Daten von der Simulation empfängt, an alle Reduktionskomponenten weiterleitet und so die auf Kommunikation optimierte Infrastruktur des Clusters gut genutzt werden kann. Für die Verteilung dieser Daten (wie z.B. Bildparameter und Geometriedaten) bietet sich die Verwendung von Multicasting an (da jeder Knoten identische Daten erhält), jedoch ist die Implementation der dafür geeigneten CORBA-Spezifikation MIOP (Multicast InterORB Protocol) noch nicht vollständig.

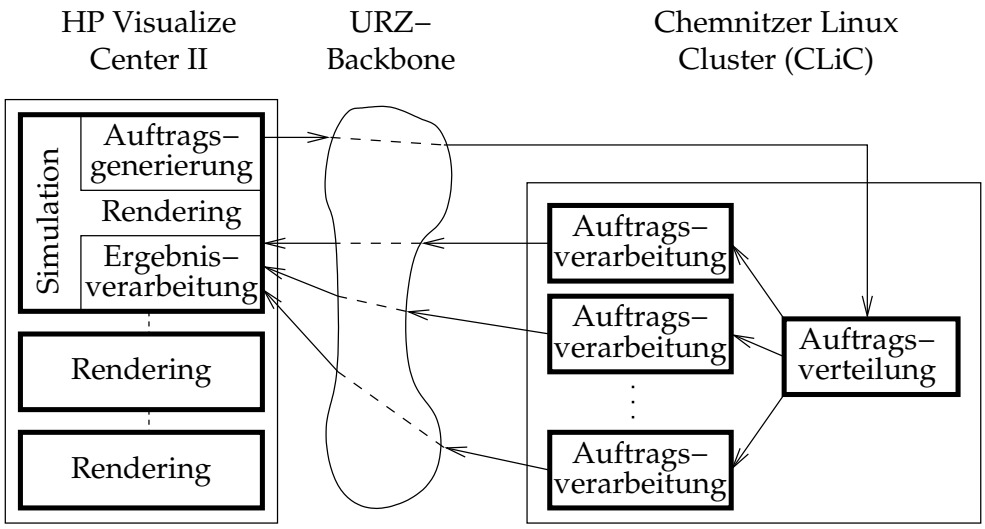


Abbildung 5.6.: Physische Komponenten der Reduktion (fett umrahmt: Jeweils ein Rechner mit zugeordneten Aufgaben)

In Abbildung 5.6 wird die Zusammenarbeit der physischen Komponenten illustriert. Die Auftragsgenerierung und Ergebnisverarbeitung läßt sich nicht sinnvoll auf eine eigene Maschine verlagern, da sie sehr eng mit der Simulation zusammenarbeitet und zudem direkten Zugriff auf die Datenstrukturen der Simulation benötigt. Die Verteilung des Rendering auf die einzelnen Maschinen des Visualize Center übernimmt transparent für den Anwender bzw. Programmierer eine Softwareschicht namens „HP Distributed 3D Single Logical Screen“, die zum Lieferumfang des Visualize Center gehört. Die Simulation läuft auf der sogenannten Master-Workstation (siehe [19]), die mit den Slave-Workstations per Gigabit-Ethernet verbunden ist.

5.5. Kommunikationsinfrastruktur

Für die Kommunikation innerhalb des verteilten Systems wurde in der Studienarbeit [25] CORBA gewählt, da es den Anforderungen am besten genügt. In Abbildung 5.7 ist die CORBA-basierte Kommunikation hervorgehoben.

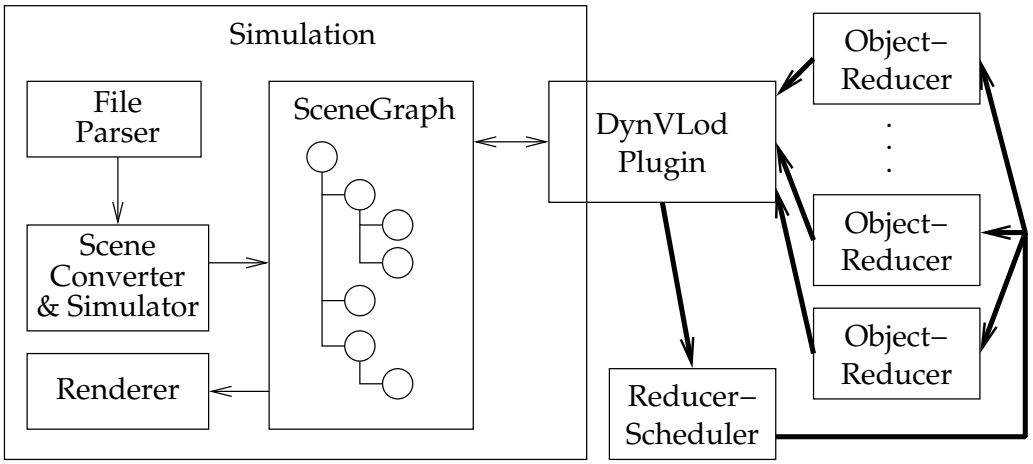


Abbildung 5.7.: CORBA-basierte Kommunikation in der Applikation

Die gegenüber MPI[1] niedrigere Performanz wird aufgewogen durch die High-Level-Schnittstelle, die den objektorientierten Programmieransatz der Gesamtapplikation optimal unterstützt. Bei der Verwendung vom MPI müßte zunächst eine umfangreiche Infrastruktur geschaffen werden, die voraussichtlich große Teile der CORBA-Schnittstellen nachahmen würde (z.B. Serialisierung und Deserialisierung von Datenstrukturen, Aufruf von Methoden von Objektinstanzen auf anderen Rechnern, Auffinden der verteilten Komponenten). Letztendlich würde durch diese Infrastruktur vermutlich ein großer Teil von MPIs Performanzvorteil wieder verloren gehen (CORBA erreichte bei den Messungen in [25] ca. 80 % des Durchsatzes von MPI).

Statt „das Rad neu zu erfinden“ kann man durch die Verwendung von CORBA einen ausgereiften Standard einsetzen, für den mehrere freie Implemen-

[1] Message Passing Interface, ein Standard für die Kommunikation von verteilten Programmen. Es existieren mehrere hochoptimierte, freie MPI-Implementationen, u.a. MPIch (`http://www-unix.mcs.anl.gov/mpi/mpich/`) und LAM-MPI (`http://www.lam-mpi.org/`).

tationen existieren, von denen einige kommerziell unterstützt und produktiv eingesetzt werden[2].

Zusätzlich erlaubt der Einsatz von CORBA die Implementation der Reduktionskomponenten in jeder beliebigen Programmiersprache für die eine CORBA-Implementierung existiert. So ist es zum Beispiel möglich, Reduktionsalgorithmen in Java, Python oder Fortran zu implementieren, ohne daß dies einen Einfluß auf andere Teile der Applikation hat.

Zur Etablierung der Kommunikation in der verteilten Applikation wird zusätzlich der CORBA-NamingService benötigt, der hier jedoch nicht als Teil der Applikation betrachtet wird. Obwohl die Verwendung dieses Namensdienstes für die Auflösung einer einzelnen Objektreferenz auf den ersten Blick nicht sinnvoll erscheint (statt der Lokation der Verteilungskomponente muß die des Namensdienstes den Komponenten der Applikation mitgeteilt werden), ergeben sich in der Praxis Vereinfachungen durch die verwendete CORBA-Implementation TAO[3,4].

5.6. Datenflüsse im System

Zum besseren Verständnis der Arbeitsweise des Systems werden an dieser Stelle zwei Hauptdatenflüsse vorgestellt: Die Verteilung der Szenedaten nach erstmaliger Aktivierung der Reduktion sowie die Verarbeitung eines Reduktionsauftrages.

5.6.1. Verteilung der Szenedaten

Die Szenedaten werden bei erstmaliger Aktivierung der Reduktion an alle Reduktionskomponenten verteilt. Dabei werden zunächst die Szenedaten aus der SceneGraph-Darstellung in CORBA-spezifische Datenstrukturen überführt, die aus der IDL-Beschreibung generiert wurden. Dann erfolgt ein Aufruf der Verteilungskomponente mit Übergabe der Szenedaten

[2] Zum Beispiel wird TAO von Object Computing, Inc. (`http://www.ociweb.com/`) kommerziell vertrieben und unterstützt.

[3] The ACE ORB, eine CORBA-Implementierung, die am „Center for Distributed Object Computing“ der Washington University in St. Louis' begonnen wurde. Siehe [23].

[4] TAO sieht für die Angabe des NamingService spezielle Möglichkeiten vor, die für die Verteilungskomponente nachgebildet werden müssten. Durch die Verwendung des NamingService kann der Anwender die dokumentierten Verfahren von TAO nutzen (z.B. Umgebungsvariablen oder Multicasting).

(siehe Abbildung 5.8), dabei werden die Geometriedaten durch die CORBA-Implementierung in eine Binärdarstellung überführt (entsprechend dem IIOP-Protokoll), die per Netzwerk versendet und vom Empfänger wieder in die CORBA-Repräsentation überführt wird.

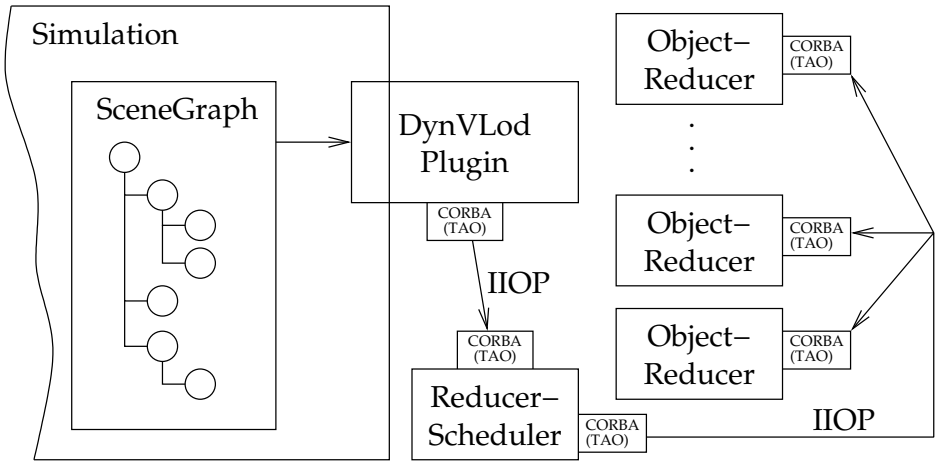


Abbildung 5.8.: Datenfluß für Verteilung der Szenedaten

Die Auftragsverteilungskomponente wiederum übergibt die Daten iterativ allen Reduktionskomponenten (dieser Schritte könnte, wie bereits erläutert, per Multicasting beschleunigt werden). Die Reduktionskomponenten speichern dann die Szene (siehe Abbildung 5.9). In Abschnitt 6.2.2, Seite 64 wird noch einmal auf die Thematik der mehrfachen Datenkonvertierung eingegangen.

5.6.2. Verarbeitung eines Reduktionsauftrages

Wird für ein Objekt festgestellt, daß keine gültige Reduktion vorhanden ist, so wird vom Modul DynVLod_CadvisPlugin eine neue Reduktion angefordert. Die Parameter für das zu rendernde Bild liegen der Auftragsverteilung bereits vor, es werden also nur die Nummer des zu reduzierenden Objektes sowie eine Objektreferenz für den Empfänger der Reduktion an die Verteilungskomponente übergeben.

Die Verteilungskomponente ermittelt nun einen passenden Reduzierer (möglichst den, der das gleiche Objekt zuletzt reduziert hat) und reicht den Auftrag an diesen weiter. Zuvor prüft sie jedoch, ob der Reduzierer bereits

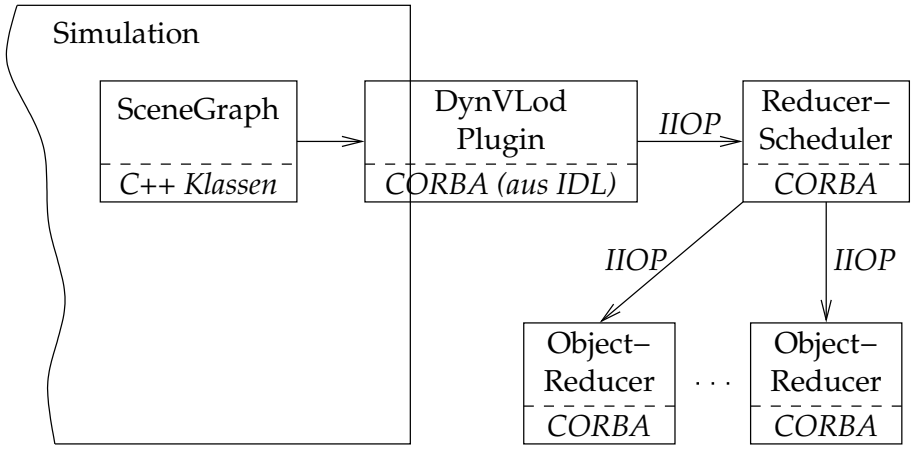


Abbildung 5.9.: Datenkonvertierungen während Verteilung der Szenedaten. Es sind jeweils das verarbeitende Modul und die verwendete Repräsentation vermerkt.

die Parameter des zu rendernden Bildes erhalten hat und übergibt diese gegebenenfalls zusammen mit ausstehenden Transformationsänderungen im Szenegraph.

Wie die Reduktionskomponente den Auftrag verarbeitet ist prinzipiell offen. In den Beispielimplementationen (wie z.B. QuadricClusteringReducer) ergab sich jedoch die Notwendigkeit, die CORBA-spezifischen Datenstrukturen (entsprechend der IDL-Beschreibung) in andere Strukturen zu überführen, da die jeweiligen Algorithmen Teil externer Bibliotheken waren. Grundsätzlich erscheint diese Konvertierung notwendig, da ein Reduktionalgorithmus die Geometrie immer mit zusätzlichen Informationen versehen muß (in C++ können zu einer Instanz einer Klasse zur Laufzeit keine klassenfremden Daten hinzugefügt werden; Python z.B. läßt dies jedoch zu). Zusätzlich erschwert wird die Konvertierung durch den Umstand, daß alle getesteten Reduktionslagorithmen nicht mit hierarchischen Geometrierepräsentationen arbeiten, sondern lediglich mit unstrukturierten Mengen von Polygonen. Dies verhindert zum Beispiel die einmalige Konvertierung der gesamten Szene in für den Algorithmus verwendbare Strukturen, da nicht bekannt ist, für welche Objekte Reduktionen benötigt werden. Letztendlich müssen die algorithmus-spezifischen Datenstrukturen wieder in CORBA-spezifische zurückgewandelt werden, um die Reduktion zum anfordernden Objekt senden zu können.

In Abbildung 5.10 ist der entsprechende Datenfluß dargestellt. Der

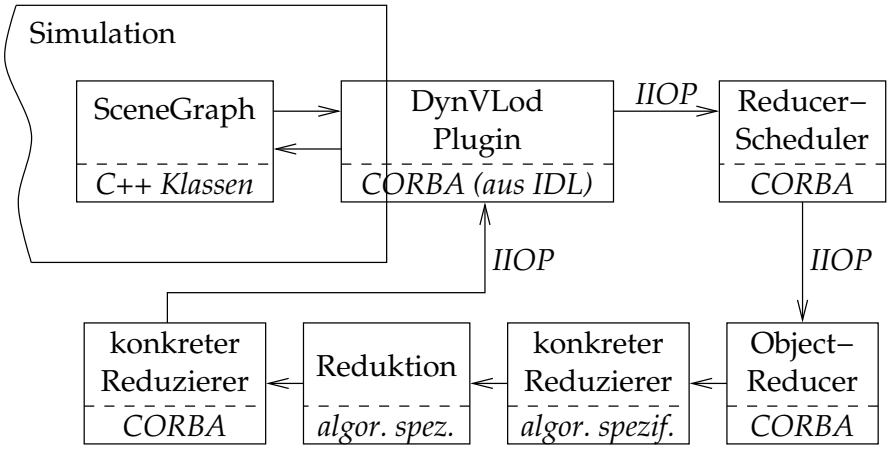


Abbildung 5.10.: Datenfluß für die Reduktion eines Objektes mit Angabe der verwendeten Repräsenation der Daten

Empfänger der Reduktion ist dabei architektonisch Teil des DynVLod_CadvisPlugin, strukturell jedoch Teil des SceneGraph, da das empfangende Objekt als Knoten in den Szenegraph eingebunden ist.

5.7. Bestimmung reduzierbarer Objekte

Bislang ungeklärt ist die Frage, welche Objekte für die Reduktion in Frage kommen. Dafür wurde ein sehr einfacher Ansatz gewählt: Alle LOD-Objekte im Szenegraph werden für die automatische Reduktion herangezogen. Dahinter steht die Überlegung, daß Objekte, die explizit als LOD modelliert sind, so komplex sind, daß sie die Darstellungsgeschwindigkeit vermutlich signifikant beeinflussen. Weiterhin ergibt sich damit eine einfache Möglichkeit, Objekte im CADaVR-Dateiformat als reduzierbar zu kennzeichnen: Man fügt direkt davor ein LodObject ein, das nur ein Kindobjekt hat. Ist das Objekt in einer eigenen Datei definiert, so läßt sich die Import-Fähigkeit des CADaVR-Formates nutzen und es sind keine Modifikationen an der Original-Datei notwendig.

5.8. Zusammenfassung

Die logische Architektur des Gesamtsystems ergibt sich aus den Anforderungen für die verteilte Reduktion für adaptives Level-of-Detail. Es werden

die Hauptkomponenten „Simulation“, „Auftragsgenerierung“, „Auftragsverteilung“, „Auftragsverarbeitung“ und „Ergebnisverarbeitung“ identifiziert. Für eine verbesserte Wiederverwendbarkeit werden in der Simulation ein zentraler Szenegraph, der Renderer sowie Dateiparser und -konverter unterschieden. Die Realisierung der Schnittstelle zur verteilten Objektreduktion als Plugin macht die Simulation vollständig unabhängig von der Reduktion.

Die physische Architektur wird durch die Netzwerkinfrastruktur und die verfügbaren Rechenkapazitäten bestimmt. Sie ähnelt der logischen Architektur sehr stark, da sich diese seit Beginn an den praktischen Gegebenheiten orientierte.

6. Ergebnisse

Das in Kapitel 5 vorgestellte System zur verteilten, adaptiven Objektreduktion wurde implementiert. Eine Reihe von Reduktionsalgorithmen wurden untersucht und praktisch getestet.

Der Algorithmus vtkQuadricClustering wurde aufgrund seiner hohen Geschwindigkeit und der verhältnismäßig einfach zu bestimmenden Reduktionsparameter für eine Referenzimplementation verwendet. Alle anderen Algorithmen reduzieren lediglich auf einen festen Prozentsatz der Flächen oder Eckpunkte des Originalmodells und erzeugen damit eine einzelne statische LOD-Stufe, die beliebig oft wiederverwendet werden kann. Der vorgestellte Ansatz kann also als Generalisierung von statischem Level-of-Detail verstanden werden.

Es stellte sich jedoch heraus, daß die Algorithmen in den vorliegenden Implementierungen für typische VR-Szenen wenig geeignet sind. So kann lediglich ein Algorithmus, PropSlim, überhaupt Attribute der Modelle wie Texturkoordinaten, Normalen und Farben berücksichtigen. Alle anderen Algorithmen können diese Attribute nicht verarbeiten, sie gehen bei der Reduktion verloren. Selbst PropSlim ist damit noch nicht für beliebige VR-Modelle geeignet, denn ein einzelnes Objekt einer Szene enthält in der Regel verschiedene Materialien und ein Material ist nicht nur durch eine Farbe, sondern durch weitere Eigenschaften wie Transparenz, diffuse und spekulare Reflektion, Textur etc. definiert.

Dabei ist hervorzuheben, daß diese Einschränkung von PropSlim nicht prinzipieller Natur ist, sondern durch eine geeignete Neuimplementierung behoben werden könnte.

Eine weitere Einschränkung der vorliegenden Implementation besteht darin, daß nach dem Erzeugen der Reduktionsaufträge auf die Ergebnisse gewartet wird, bevor das Rendering fortgesetzt wird. Dieses Vorgehen hat vor allem praktische Gründe – eine asynchrone Verarbeitung der Ergebnisse wirft eine Reihe neuer Probleme auf, die später noch diskutiert werden.

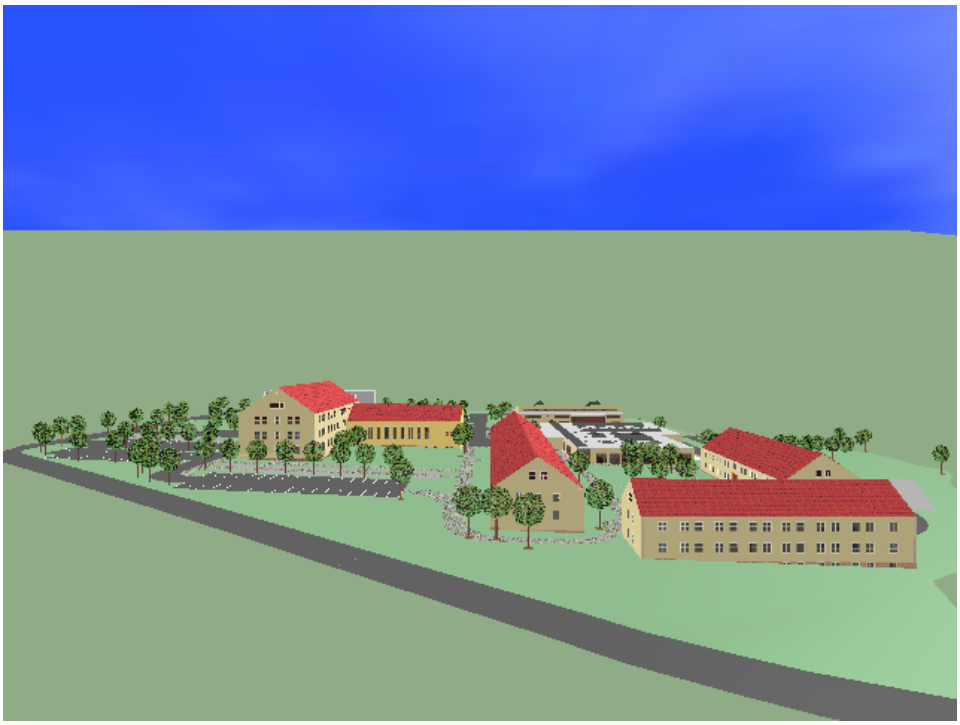
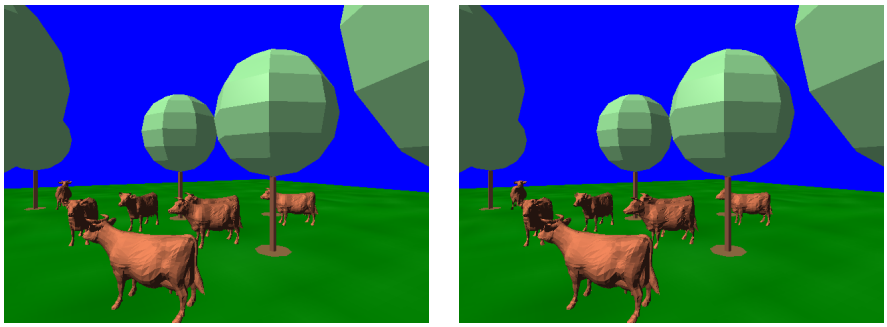


Abbildung 6.1.: Eine für die Reduktion ungeeignete Szene. Die Häuser enthalten verschiedenste Materialien.

6.1. Benchmark

Um die Leistungsfähigkeit des vorgestellten Systems zu bestimmen, wurden Messungen durchgeführt. Zu diesem Zweck wurde eine Szene mit einigen Objekten und einer Animation erstellt (siehe Abbildung 6.2). Die Animation dauert 300 Simulationssekunden, die Simulationsauflösung wurde für die Messung auf 0.1 Sekunde festgesetzt. Die Simulation lief auf der Master-Workstation des Visualize Center, für die Reduktion wurden 8 Knoten des CLiC verwendet, davon 1 für die Auftragsverteilung.

Gemessen wurde die Latenz zwischen Beginn und Ende des Rendering. Die Ergebnisse der Messung sind in Abbildung 6.3 dargestellt. Gut zu erkennen sind die durch das Warten auf Reduktionsergebnisse hervorgerufenen kurzen Anstiege der Renderinglatenz. Weiterhin ist sofort ersichtlich, daß die Reduktion tatsächlich zu einer Geschwindigkeitssteigerung führt – Bilder, für die keine neuen Reduktionen berechnet werden müssen, werden deutlich schneller gerendert als ohne Reduktion.



(a) ohne Reduktion (b) mit Reduktion

Abbildung 6.2.: Screenshot der Benchmarkszene

Insgesamt ergibt sich im Mittel eine Bildrate von 17,38 Bildern pro Sekunde ohne Reduktion und 21,61 Bildern pro Sekunde mit Reduktion. Trotz des zusätzlichen Rechenaufwandes für die Erzeugung der Reduktionen und die damit verbundenen Wartezeiten wird also eine leichte Geschwindigkeitssteigerung erreicht. Mit Fortsetzung der Simulation ist ein zunehmender Geschwindigkeitsvorsprung der Variante mit Reduktion zu erwarten, da die verwendete Animation den Cache bereits gefüllt hat und im Folgenden nur noch wenige Neuberechnungen von Reduktionen notwendig sind.

Bei geeigneten Szenen ist also nach der Berechnung einiger Reduktionen mit einer Geschwindigkeitssteigerung zu rechnen.

6.2. Ausblick

Im Laufe der Erstellung dieser Arbeit und der Implementierung der Applikation(en) mußten immer wieder Kompromisse eingegangen werden. Es zeigte sich eine Reihe von Verbesserungsmöglichkeiten, die jedoch über das Anliegen dieser Arbeit hinausgehen.

6.2.1. Intelligentere Auftragserzeugung

So liegt es zum Beispiel nahe, nur Aufträge zu erzeugen, falls ein signifikant reduziertes Ergebnis zu erwarten ist. Der QuadricClusteringReducer hat zum

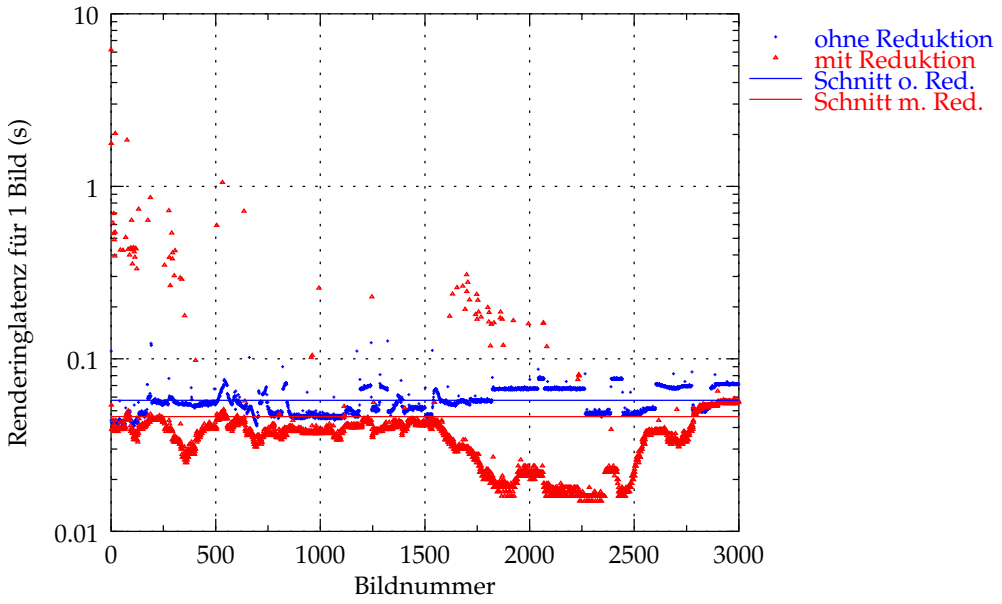


Abbildung 6.3.: Benchmark der verteilten Reduktion (mit Reduzierer QuadricClusteringReducer)

Beispiel eine Obergrenze für die Rasterauflösung. Ab einer bestimmten Entfernung zur Kamera (abhängig von der Bildgeometrie) werden also keine detaillierteren Reduktionen mehr erzeugt. Solche und ähnliche Abhängigkeiten könnten von einer Heuristik bei der Auftragserzeugung erkannt werden und somit würde Rechenzeit gespart. In diesem Zusammenhang muß auch erwähnt werden, daß die Originalobjekte derzeit bei aktivierter Reduktion nicht verwendet werden. Damit sind natürlich bei der Betrachtung aus nächster Nähe nicht alle Details verfügbar.

Denkbar ist in diesem Zusammenhang auch eine Ergänzung der Schnittstelle zwischen Reduzierer und Simulation. Bislang wird davon ausgegangen, daß der Reduzierer für das gewünschte Objekt eine Reduktion erzeugen kann. Es besteht jedoch auch die Möglichkeit, daß ein Reduktionsalgorithmus für das Objekt gar nicht geeignet ist (z.B. wenn Algorithmen eingesetzt werden sollen, die bestimmte topologische Eigenschaften der Objekte voraussetzen und diese nicht gegeben sind). Diesen Umstand könnte der Reduzierer der Simulation mitteilen und für das betreffende Objekt würden keine Reduktionen mehr angefordert.

6.2.2. Effizientere Geometriedatenübertragung

Auch im Bereich der Geometriedatenübertragung werden Reserven für eine deutliche Steigerung der Geschwindigkeit gesehen. Aufgrund des Designs von CORBA und SceneGraph können die SceneGraph-Datenstrukturen nicht direkt für den Versand über das Netzwerk verwendet werden. Es ist also eine Konvertierung in spezielle, mit CORBA IDL beschriebene Datenstrukturen notwendig (siehe Abbildung 5.9).

Taubin und Rossignac beschreiben in [26] ein Verfahren zur Geometriekompression, das nicht selten eine Reduktion der Daten auf ein Fünfzigstel des Originalumfangs erreicht. Somit ließen sich deutlich mehr reduzierte Objekte pro Bild empfangen. Es liegt also nahe, die Geometriedaten mit Hilfe spezieller Algorithmen in eine binäre Repräsentation zu überführen und diese dann mit CORBA-Mitteln zu versenden[1]. Damit würde außerdem das Kopieren der SceneGraph-Datenstrukturen in CORBA-spezifische eingespart. Die Konvertierung der CORBA-spezifischen Datenstrukturen in einen Datenstrom für den Netzwerkversand würde dann durch die Geometriekompression ersetzt.

Die Verwendung von Geometriekompression erscheint vielversprechend und ließe sich mit relativ wenigen Änderungen in die bestehende Applikation integrieren. Lediglich der Versand der Szene, deren Empfang, sowie der Versand der Reduktionen und deren Empfang wären von dieser Änderung betroffen. Der Nachteil der Geometriekompression ist die Aufgabe einiger Vorteile von CORBA, namentlich der Plattform- und Programmiersprachenunabhängigkeit. Diese Nachteile lassen sich jedoch kompensieren durch eine präzise Spezifikation des Binärdatenstroms der Geometriedaten und entsprechende Implementation (Stichwort: Network Byte Order und Host Byte Order) sowie die Reimplementierung der Geometriekompression und -dekompression für Programmiersprachen neben C++.

6.2.3. Ausnutzung von Redundanz in der Szene

In VR-Szenen kommt es häufig vor, daß ein einzelnes Objekt mehrfach instanziiert wird (z.B. Bäume). Die Geometrie eines Objektes wird nur einmal spezifiziert und an anderer Stelle lediglich referenziert, um ein weiteres

[1] Für solche Zwecke sieht CORBA eine Octet Sequence vor, also einen binären Datenstrom. In vielen CORBA-Implementationen ist die Verarbeitungen von Octet Sequences speziell optimiert und deshalb sehr performant.

Objekt zu erzeugen. Im CADaVR-Dateiformat wird dieses Vorgehen indirekt durch das Importieren anderer Dateien unterstützt. Es erscheint also logisch, diese Redundanz zu nutzen und die Reduktionen des Objektes für alle Instanzen des Objektes zu verwenden.

Die Verwendung von Objekt-Instantiierung führt jedoch zu einigen Problemen speziell im Kontext von VR-Simulationen. So gilt es zum Beispiel zu klären, was passiert, wenn Veränderungen an einer Instanz vorgenommen werden (z.B. in Form einer Animation). Im Allgemeinen soll dann nur genau eine Instanz betroffen sein, man müßte also veränderte Informationen getrennt von gemeinsamen Informationen speichern und einen Copy-on-Write-Mechanismus vorsehen. Dieses Problem wird zusätzlich dadurch verschärft, daß prinzipiell Instanzen von beliebigen Knoten des Szenegraph erzeugt werden können, also jeweils ganze Teilgraphen betroffen sind – eine Veränderung eines Objektes innerhalb des instantiierten Teilgraph soll sich nur auf eine Instanz auswirken. Zusätzlich sollte es natürlich auch die Möglichkeit geben, das Ur-Objekt aller Instanzen zu verändern, so daß alle Instanzen die Veränderung übernehmen. Was passiert dann bei gegenläufigen Änderungen (zuerst wurde eine Instanz verändert, dann das gleiche Attribut des Ur-Objektes)?

Angesichts dieser Probleme wird keine praktikable Möglichkeit für eine Implementation von Instanzen gesehen, obwohl dies größere Geschwindigkeitsgewinne verspricht.

6.2.4. Entkopplung von Rendering und Reduktion

Wie am Anfang dieses Kapitels bereits bemerkt, wird das Rendering erst fortgesetzt, wenn alle angeforderten Reduktionen eingetroffen sind. Dies führt zu deutlich wahrnehmbaren Wartezeiten und stört damit den immersiven Eindruck der Simulation. Prinzipiell ist ein asynchroner Rendermodus denkbar. So könnte, während die angeforderte Reduktion noch nicht verfügbar ist, entweder die Originalgeometrie oder die am besten geeignete bereits verfügbare Reduktion für das Rendering verwendet werden.

Die Entkopplung von Rendering und Reduktion ist jedoch nicht unproblematisch. Entweder es wird immer nur eine Reduktion angefordert oder es können auch mehrere Reduktionsaufträge nacheinander erzeugt werden. Im ersten Fall besteht die Gefahr, daß mit Eintreffen der Reduktion diese für die mittlerweile veränderte Situation in der Szene nicht mehr verwendbar ist. Diese Wahrscheinlichkeit wird um so höher, je schneller sich der

relative Position des Objektes zur Kamera ändert und je länger die Reduktion des Objektes dauert. Läßt man mehrere ausstehende Reduktionen pro Objekt zu, so kann es außerdem passieren, daß eine Menge von Reduktionsaufträgen bei der Auftragsverteilung auflaufen, weil die Reduktionskomponenten diese nicht schnell genug bearbeiten können (bei 20 Bildern pro Sekunde würden etwa 20 mal so viele Reduktionsaufträge versendet werden).

Das Grundproblem des asynchronen Rendering ist, daß die Simulation die Gültigkeitskriterien der Reduktionen erst mit dem Reduktionsergebnis erfährt. Somit können keine Annahmen in Bezug auf die Verwendbarkeit der Reduktionen getroffen werden. Auch können deshalb Reduktionen nicht vorausschauend angefordert werden (z.B. durch Extrapolation der Bewegung des Beobachters oder der Objekte).

Grundsätzlich erscheint ein asynchroner Rendermodus sinnvoll, da er eine Verbesserung des immersiven Eindrucks bewirkt. Für seine Realisierung werden jedoch vermutlich zusätzliche Schnittstellen zwischen Reduktion und Simulation notwendig. Eine Grundannahme des vorliegenden Systems müßte verworfen werden: Die Simulation weiß nichts über die Art der Reduktion und kann diese nicht direkt beeinflussen. Wird diese Annahme fallen gelassen, so ist auch der parallele Einsatz verschiedener Reduktionsalgorithmen denkbar oder eine Verkettung mehrerer Algorithmen. Die Auswahl des Algorithmus obläge dann der Simulation; diese könnte zusätzlich noch einen abstrakten Reduktionsparameter oder einen erwünschten Gültigkeitsbereich zur Steuerung der Reduktion spezifizieren.

6.2.5. Weitere Cache-Verbesserungen

Für den Cache könnten die Ressourcen der zahlreichen Reduktionsrechner herangezogen werden. Optimal wäre dabei eine $1 : 1$ Zuordnung von reduzierbarem Objekt zu Reduktionsrechner (im bereits diskutierten asynchronen Rendermodus kann dies jedoch zu einem Engpass führen). Dieser Rechner könnte dann eine große Menge von Reduktionen speichern und bei Bedarf ohne Neuberechnung an die Simulation senden. In der derzeitigen Konfiguration erscheint dies jedoch nicht sinnvoll, da einerseits die Reduktionsalgorithmen mitunter große Speichermengen für die Berechnungen benötigen und andererseits die Speicherausstattung des Visualize Center mit 4 GB Arbeitsspeicher deutlich über den Reduktionsrechnern mit 512 MB liegt.

Verkompliziert wird dieser Ansatz durch die derzeitigen Parameter für einen Reduktionsauftrag – dieser besteht lediglich aus einer Objektnummer und implizit der aktuellen Kameraposition sowie den Bildparametern. Der Reduzierer müßte also für jede Reduktion die aktuellen Kameraparameter speichern und hätte nur diese als Suchschlüssel im Cache zur Verfügung. Die Verwendung eines verteilten Cache erscheint also ohne spezielle Schnittstellen nicht sinnvoll.

Innerhalb der Simulation besteht allerdings noch Verbesserungspotenzial bei der Cache-Verwaltung. Wie in Abschnitt 4.2.4 (Seite 44) bereits erläutert, wird in der aktuellen Implementation die „älteste“ Reduktion freigegeben, d.h. alle Reduktionen werden in einem FIFO gespeichert und somit in der gleichen Reihenfolge gelöscht, in der sie aufgenommen wurden. Eine optimale Freigabestrategie würde zunächst die vorhandenen Reduktionen gewichten nach der Wahrscheinlichkeit, daß sie „in Kürze“ wieder benötigt wird. Diese Gewichtung könnte auf Basis der Abweichung zwischen derzeitigem Abstand Objekt–Betrachter und Abstandsintervall der Reduktion sowie derzeitigem Blickwinkel auf das Objekt und Blickwinkelgültigkeit der Reduktion erfolgen.

6.2.6. Verteiltes Rendering

Die vorliegende Infrastruktur läßt sich nicht nur zur verteilten Reduktion nutzen. Denkbar ist auch die Verwendung für verteiltes Rendering. Dabei würde die Rücksendung der Reduktionen zur Simulation entfallen und durch Rendering des Objektes ersetzt werden. Mit geeigneter Hardware ließen sich dann die gerenderten Objekte zu einem Gesamtbild kombinieren.

6.3. Zusammenfassung

Trotz der genannten Probleme und Verbesserungsmöglichkeiten lassen sich mit der vorliegenden Applikation bei geeigneten Szenen deutliche Geschwindigkeitssteigerungen erzielen. „Geeignet“ bedeutet dabei, daß die Szene Objekte enthält, die von vtkQuadricClustering reduziert werden können und nicht mehr als ein Material enthalten.

A. Dokumentation

Im Folgenden werden die Schritte beschrieben, die zum Kompilieren und Ausführen der Applikation notwendig sind. Alle dabei verwendeten Shellkommandos sind für eine Bourne-Shell (wie z.B. GNU bash) ausgelegt. Kommandozeilen, die die verfügbare Seitenbreite überschreiten würden sind Shell-üblich mittels Backslash „\“ am Zeilenende gekennzeichnet und werden auf der folgenden Zeile fortgesetzt.

Alle hier beschriebenen Programme sind sowohl unter HP-UX 11.0 sowie Linux lauffähig. Dabei sind die Sourcecodes multiplattformfähig, das heißt der gleiche Sourcebaum kann parallel Kompilate für verschiedene Plattformen bereitstellen – es sind also keine separaten Kopien des Sourcecodes für die Kompilation auf verschiedenen Plattformen notwendig.

Als Compiler wird der GNU C Compiler in der Version 3.1 oder neuer empfohlen.

Zur Vereinfachung der folgenden Schritte wird davon ausgegangen, daß sich alle Sourcecodes etc. in einem Verzeichnis `BASEDIR` befinden, das auf allen beteiligten Rechnern unter dem gleichen Namen erreichbar ist (also z.B. ein Verzeichnis im AFS).

A.1. Kompilieren der Applikation

Die Hauptbestandteile der Applikation sind die CADvis-Portierung für Unix und die Komponenten für die verteilte Reduktion. Letztere benötigen eine CORBA-Implementation, so daß diese vorher vorbereitet werden muß.

Für den Fall, daß Sourcecode aus dem CVS-Repository der Professur für graphische Datenverarbeitung und Visualisierung ausgecheckt werden muß, sind vorher die Umgebungsvariablen `CVSROOT`, `CVS_SERVER` und `CVS_RSH` zu setzen:

```
export CVS_RSH=ssh
export CVS_SERVER=/usr/local/bin/cvs
export CVSROOT=rad:/home/CVSROOT/Repository
```

Das CVS-Repository befindet sich auf dem Rechner `rad.informatik.tu-chemnitz.de`, hier kurz mit `rad` bezeichnet.

A.1.1. CADvis

Als erstes wird die CADvis-Portierung für Unix benötigt, da die verteilte Reduktion darauf aufbaut. Die dieser Arbeit entsprechende Version ist im CVS-Repository mit `tisc_Diplomarbeit` markiert. Sie kann mit

```
cd $BASEDIR
cvs checkout -r tisc_Diplomarbeit -d CADvis-src \
   CADvis/src
```

aus dem CVS in das lokale Verzeichnis `CADvis-src` kopiert werden.

Für die Kompilierung benötigt man den GNU C Compiler (Version $\geq$ 2.95.3), flex++ (Version $\geq$ 2.5.4), bison++ (Version $\geq$ 1.21-8) und WTK (Version $\geq$ 8). Falls bison++ nicht vorhanden ist, findet sich eine Version im Archiv `CADvis-src/Tools/bison++-1.21-8.tar.bz2`.

Eventuell sind noch systemspezifische Einstellungen in der Datei `CADvis-src/Common/Make.defs` notwendig (siehe z.B. die Variablen `WTKDIR`, `X11DIR`, `LEX`, `BISONXX`).

Die Kompilierung aller notwendigen Bibliotheken wird nun durch den Aufruf von GNU make (unter HP-UX und IRIX als `gmake` verfügbar) initiiert:

```
cd $BASEDIR/CADvis-src
gmake
```

A.1.2. CORBA-Implementation

Wie bereits erwähnt, wird als CORBA-Implementation TAO verwendet. Die Kompilierung von TAO ist recht umfangreich und benötigt viel Plattenplatz (siehe dazu die TAO Dokumentation). Fertig kompilierte Versionen von TAO (und der benötigten ACE-Bibliothek) für HP-UX und Linux finden sich im Verzeichnis `/afs/tu-chemnitz.de/project/dynvlod`. Die notwendigen Umgebungsvariablen setzt man durch Einbinden der Datei `setup.ace` im genannten Verzeichnis:

```
. /afs/tu-chemnitz.de/project/dynvlod/setup.ace
```

Danach verweisen die Umgebungsvariablen ACE_ROOT und TAO_ROOT auf die Verzeichnisse mit den Kompilaten für die aktuelle Plattform. Außerdem wird LD_LIBRARY_PATH entsprechend angepaßt. Wenn die vorkompilierte Version nicht eingesetzt werden soll oder kann, so sind lediglich diese drei Umgebungsvariablen entsprechend anzupassen. (Achtung: Der Sourcecode von ACE und TAO ist nicht multiplattformfähig. Es wird also für jede Plattform ein eigener Sourcebaum benötigt. Siehe dazu die ACE-Dokumentation.)

Damit die entsprechenden Einstellungen nicht in die Login-Skripte übernommen werden müssen, empfiehlt es sich, ein Skript `setup.ace` in $BASEDIR anzulegen, welches die notwendigen Einstellungen vornimmt. Bei Verwendung der oben genannten kompilierten Version genügt stattdessen:

```
cd $BASEDIR
ln -s /afs/tu-chemnitz.de/project/dynvlod/setup.ace .
```

A.1.3. Verteilte Reduktion

Die dieser Arbeit entsprechende Version der verteilten Reduktion ist im CVS-Repository ebenfalls mit `tisc_Diplomarbeit` markiert. Sie kann mit

```
cd $BASEDIR
cvs checkout -r tisc_Diplomarbeit -d tisc-DA \
   tisc/Diplomarbeit/Implementation
```

in das lokale Verzeichnis `tisc-DA` kopiert werden. Prinzipiell können die Komponenten jetzt kompiliert werden. Jedoch benötigen alle sinnvoll einsetzbaren Reduktionsalgorithmen zusätzliche Bibliotheken. Die zusätlichen Bibliotheken müssen nur für die Plattform(en) kompiliert werden, die zur Reduktion eingesetzt werden sollen.

VTK

Einige Reduktionskomponenten basieren auf dem Visualization Toolkit (VTK). Implementiert und getestet wurde mit der VTK Version 4.0. Der Original-Sourcecode ist im CVS-Repository gespeichert, zur Kompilation wird zusätzlich das Tool CMake benötigt, welches ebenfalls im CVS abgelegt ist:

```
cd $BASEDIR/tisc-DA
cvs checkout -d sources \
   tisc/Diplomarbeit/Material/vtk40Src.tar.gz
cvs checkout -d sources \
   tisc/Diplomarbeit/Material/CMake1.4.6-src-unix.tar.gz
```

Für die Kompilation von VTK und CMake sei auf die dazugehörige Dokumentation verwiesen. Wichtig ist, daß die Erzeugung dynamischer Bibliotheken bei der VTK-Kompilierung aktiviert wird. Es wird empfohlen, den VTK-Sourcecode in die Verzeichnisse `tisc-DA/VTK.Linux` bzw. `tisc-DA/VTK.HP-UX` zu kopieren und zu kompilieren, da er dort automatisch gefunden wird. Alternativ kann mit der Umgebungsvariable `VTK_ROOT` der Pfad zum VTK spezifiziert werden.

Im Allgemeinen sollte mit folgenden Schritten CMake und VTK kompiliert werden können[1]:

```
cd $BASEDIR/tisc-DA
gtar xfz sources/CMake1.4.6-src-unix.tar.gz
mv CMake-1.4.6 CMake-1.4.6.‘uname‘
cd CMake-1.4.6.‘uname‘
./configure
gmake
cd $BASEDIR/tisc-DA
gtar xfz sources/vtk40Src.tar.gz
mv VTK VTK.‘uname‘
cd VTK.‘uname‘
# Datei CMakeLists.txt editieren:
# Option BUILD_SHARED_LIBS auf ON setzen
../CMake-1.4.6.‘uname‘/Source/cmake
gmake
```

QSlim

QSlim ist eine Implementation der von Michael Garland entwickelten Reduktionsalgorithmen, die auf Quadriken basieren (siehe [8]). Eine leicht modifizierte Variante der Version 2.0 von QSlim ist im Verzeichnis `tisc-DA/qslim-2.0` verfügbar. Die Kompilierung ist unkompliziert:

[1] Die Kompilierung auf HP-UX funktioniert derzeit nicht.

```
cd $BASEDIR/tisc-DA/qslim-2.0/mixkit/build
rm config.cache config.log config.status
./configure
cd ../src
gmake
```

Kompilieren

Nachdem VTK und QSlim kompiliert wurden, können die restlichen Komponenten der verteilten Reduktion kompiliert werden:

```
cd $BASEDIR/tisc-DA
gmake
```

Die fertigen Programme und Bibliotheken befinden sich danach im Unterverzeichnis `bin.Linux` bzw. `bin.HP-UX`.

A.2. Starten der Applikation

Die verteilte Applikation verwendet CORBA für die Kommunikation. Als zentrale Anlaufstelle dient die Auftragsverteilung, ReducerScheduler. Um diese Komponente im Netzwerk zu finden, wird der CORBA-NamingService verwendet.

A.2.1. CORBA-NamingService

Eine Implementation des CORBA-NamingService ist Teil von TAO. Der NamingService kann prinzipiell auf jedem beliebigen Rechner gestartet werden, der von allen anderen für die Applikation benötigten Rechnern erreichbar ist[2]. Da der NamingService nur beim Start der verteilten Applikation, beim Start der Simulation und beim Beenden der verteilten Applikation benötigt wird, verursacht er keine nennenswerte Belastung in Form von Rechenlast oder Netzwerkverkehr. Er kann also zum Beispiel auf dem Simulationsrechner gestartet werden.

[2] Aus bislang ungeklärten Gründen funktioniert der NamingService nicht unter Linux.

Zur Etablierung der Kommunikation mit dem NamingService dient eine IOR[3]. Die IOR des NamingService wird von diesem in eine Datei `$BASEDIR/ns.ior` geschrieben, so daß die anderen Komponenten der verteilten Applikation mit Hilfe dieser Datei Kontakt zum NamingService aufnehmen können.

```
$TAO_ROOT/orbsvcs/Naming_Service/Naming_Service \
   -o $BASEDIR/ns.ior &
```

Auf den HP-UX-Rechnern des Visualize Center wird leider die IOR nicht korrekt erzeugt – statt dem vollständigen Rechnernamen wird nur der Kurzname eingetragen, so daß Rechner in anderen DNS-Domänen die Objektreferenz nicht auflösen können[4]. Abhilfe schafft in diesem Fall die explizite Angabe des Hostnamens, z.B.:

```
$TAO_ROOT/orbsvcs/Naming_Service/Naming_Service \
   -o $BASEDIR/ns.ior\
   -ORBEndpoint iiop://luzie.informatik.tu-chemnitz.de &
```

Der NamingService erzeugt in der Regel keinerlei Ausgaben. Seine Funktion kann geprüft werden mit dem Programm nslist:

```
$TAO_ROOT/utils/nslist/nslist \
   -ORBInitRef NameService=file:///$BASEDIR/ns.ior
```

A.2.2. Verteilte Reduktion

An dieser Stelle wird zunächst das allgemeine Vorgehen erläutert und später die notwendigen Ergänzungen bei Verwendung des CLiC beschrieben.

Im nächsten Schritt wird die Autragsverteilung ReducerScheduler gestartet. Wie bereits erwähnt, registriert sich ReducerScheduler beim NamingService,

[3] Interoperable Object Reference. Innerhalb des CORBA-Standards definierte Schnittstelle zur Beschreibung einer Objektreferenz. In einer IOR sind sowohl die Adresse als auch das Protokoll für den Zugriff auf ein Objekt kodiert. Dabei kann ein Objekt über verschiedene Protokolle und Adressen gleichzeitig erreichbar sein.

[4] Diese ungewöhnliche Systemkonfiguration ist für das Funktionieren des bereits erwähnten HP Distributed 3D Single Logical Screen erforderlich.

benötigt also dessen Adresse (IOR). Am einfachsten läßt sich diese mit der Umgebungsvariable `NameServiceIOR` angeben.

Für den Start von ReducerScheduler steht ein einfaches Shell-Skript zur Verfügung, daß automatisch das passende plattformspezifische Programm ausführt:

```
export NameServiceIOR=file:///$BASEDIR/ns.ior
cd $BASEDIR/tisc-DA/ReducerScheduler
./run
```

Nun können die Reduzierer gestartet werden. Auch hier wird die IOR des NamingService per Umgebungsvariable übergeben. Die einzelnen Reduktionsalgorithmen sind als Plugins für das Programm `objectreducerserver` realisiert. Für jeden Reduktionsalgorithmus existiert ein Shell-Skript, um den Start zu vereinfachen, z.B.:

```
export NameServiceIOR=file:///$BASEDIR/ns.ior
cd $BASEDIR/tisc-DA/ObjectReducer
./run.qclust
```

A.2.3. Simulation

Letztendlich kann die VR-Simulation selbst gestartet werden. Die in Abschnitt 6.1 auf Seite 61 beschriebene, für die Benchmarks verwendete Szene befindet sich im Unterverzeichnis `Modelle` der Implementation. Das Simulationshauptprogramm ist noch nicht vollständig vom Renderer entkoppelt und deshalb Teil des Moduls WtkRenderer. Für die verteilte Reduktion ist wiederum die Angabe der IOR des NamingService notwendig. Aufgrund der bereits erwähnten Probleme mit den Hostnamen im Visualize Center muß dem Reduktionsplugin ein zusätzlicher Parameter übergeben werden.

```
export NameServiceIOR=file:///$BASEDIR/ns.ior
cd $BASEDIR/tisc-DA/Modelle
$BASEDIR/CADvis-src/bin.‘uname‘/wtkrendertest --plugin \
  $BASEDIR/tisc-DA/bin.‘uname‘/DynVLod_CadvisPlugin.* \
  --pluginopt "DynVLod:-ORBEndpoint \
  iiop://rad.informatik.tu-chemnitz.de" \
  cow_hurd_bench.cad
```

Alternativ kann dieser komplexe Schritt mit Hilfe des Skriptes `tisc-DA/view` erfolgen. (Das Skript `tisc-DA/viewr` veranlaßt zusätzlich die Aktivierung der verteilten Reduktion direkt nach dem Start der Simulation.)

A.3. Verwendung des CLiC

Sollen für die Objektreduktion Knoten des CLiC verwendet werden, so muß das Batchsystem OpenPBS des Clusters verwendet werden (siehe dazu die Webseiten zum CLiC für nähere Informationen [27]). In `$BASEDIR/tisc-DA` liegt ein Beispielskript `clic_job`, welches eine Anzahl von Knoten auf dem Cluster anfordert und eine interaktive Shell startet (für die Benutzung ist ein Account auf dem CLiC notwendig):

```
cd $BASEDIR/tisc-DA
# Anzahl der Knoten als Argument (Standard: 4)
./clic_job 8
# nach zwei OpenPBS-Meldungen sollte eine Shell auf
# einem CLiC-Knoten geöffnet werden
```

Mit Hilfe des Skriptes `boot` können dann die Auftragsverteilung sowie die Reduktionskomponenten gestartet werden (1 Knoten wird abei für die Verteilung, alle restlichen reservierten Knoten für Reduzierer verwendet, der Start der Reduzierer erfolgt per `ssh`[5]). Das Skript prüft unter anderem, ob `TAO_ROOT` gesetzt ist und bindet `$BASEDIR/setup.ace` ein, falls dies nicht der Fall ist. Die zu startende Reduktionskomponente wird als Argument übergeben:

```
cd $BASEDIR/tisc-DA
./boot ObjectReducer/run.qclust
```

Beendet wird dieser Teil der Applikation durch `Return`.

[5] Eine funktionierende, keybasierte Authentifizierung wird hier vorausgesetzt. D.h. man sollte sich ohne Passworteingabe per von einem URZ-Rechner zum nächsten bewegen können, sonst funktioniert das Skript nicht. Siehe dazu die `man`-Pages von `ssh` und `ssh-keygen`.

A.4. Bedienung

Die Navigation innerhalb der Simulation erfolgt vorwiegend mit Hilfe der Maus. Der Einsatz anderer Eingabegeräte (wie z.B. SpaceMouse) ist vorgesehen, jedoch nicht implementiert. Die virtuelle Kamera wird manipuliert, indem man bei gedrückter Maustaste die Maus verschiebt. Die Wirkung der einzelnen Maustasten ist in Tabelle A.1 erläutert.

linke Taste	Bewegung vor und zurück, Schwenk nach links und rechts
mittlere Taste	Schwenk nach oben und unten, Rotation des Bildes
rechte Taste	Bewegung nach links und rechts / nach oben und unten

Tabelle A.1.: Navigation mit der Maus

Zusätzlich steht eine Reihe von Tastenkombinationen zur Verfügung mit denen die Simulation und die verteilte Reduktion beeinflußt werden können. Eine Liste der verfügbaren Kommandos erhält man mit `h`. In Tabelle A.2 sind die wichtigsten Kommandos aufgelistet.

`q`	Simulation beenden
`h`	verfügbare Kommandos auflisten
`w`	Drahtgittermodus an-/ausschalten
`f`	aktuelle Rendergeschwindigkeit ausgeben (Bilder pro Sekunde)
`g`	Screenshot anfertigen (gespeichert in `screen????.bmp`)
`R`	Verteilte Reduktion aktivieren
`S`	zwischen reduzierten und Originalobjekten umschalten
`p`	Struktur des Szenegraph ausgeben

Tabelle A.2.: Die wichtigsten Tastaturkommandos für die Simulation

B. Interfacebeschreibung in IDL

```
module DynVLod
{
    // used for name service look up
    const string ReducerScheduler_name = "ReducerScheduler";

// first part: scene graph represented in CORBA

    // we reference scene nodes by their ID - using CORBA references would be overkill
    // (the decision here is to threat scene nodes like data rather than active objects)
    typedef long NodeId;

    // materials are scene-global and identified by an ID to prevent
    // sending the same material (probably textures etc.) multiple times
    // (-1 means: no material specified / used)
    typedef long MaterialId;

    // textures are similar to materials: scene-global and identified by an ID
    // (-1 means: no texture specified / used)
    typedef long TextureId;

    typedef float Matrix[4][4];

    typedef float Vertex[3];

    union OptionalVertex switch (boolean)
    {
        case TRUE:
            Vertex    value;
    };

    struct Quaternion
    {
        float     s;
        Vertex    v;
    };
```

```
struct Color // CHECKME: Rather use float values - they need four times the space
{
    octet    r;
    octet    g;
    octet    b;
};

union OptionalColor switch (boolean)
{
    case TRUE:
        Color    value;
};

struct TexCoord
{
    float    u;
    float    v;
};

union OptionalTexCoord switch (boolean)
{
    case TRUE:
        TexCoord    value;
};

enum TriBool // tri-state-bool which allows to discriminate "not set"
{
    TB_Unknown,
    TB_True,
    TB_False
};

struct BoundingBox
{
    // implicit: the bounding box is valid only iff (min[i] <= max[i]) for all i=0..2
    // else it is considered to be empty / invalid
    Vertex    min;
    Vertex    max;
};

union OptionalBoundingBox switch (boolean)
{
    case TRUE:
        BoundingBox value;
};

struct Range
```

```
{
    float     min;
    float     max;
};

union OptionalRange switch (boolean)
{
    case TRUE:
        Range    value;
};

struct ViewAngleConstraint
{
    // The viewing angle is specified by a "zero degree" direction
    // and a maximum deviation angle. The direction vertex is supposed to
    // point from the center of the object's bounding box to the camera.
    Vertex    zero_direction;
    // Maximum deviation angle (calculated in world coordinate frame).
    // -> no depency on camera orientation but only on camera position.
    float     max_angle;
};

union OptionalViewAngle switch (boolean)
{
    case TRUE:
        ViewAngleConstraint        value;
};

struct Transformation
{
    Vertex              position;
    Quaternion          orientation;
    // Not to allow scaling via object transformation simplifies things a lot.
    // Vertex           scaling;
};

struct RenderAttributes
{
    TriBool             double_sided;
    TriBool             smooth;
    OptionalColor       color;
    MaterialId          material_id; // -1 = none
    TextureId           texture_id; // -1 = none
};

union OptionalRenderAttributes switch (boolean)
{
    case TRUE:
```

```
        RenderAttributes        value;
};

struct Point
{
    Vertex                      position;
    OptionalVertex              normal;
    OptionalTexCoord            tex_coord;
    OptionalRenderAttributes    render_attributes;
};

struct Polygon
{
    sequence<unsigned long>     point_indices;
    OptionalRenderAttributes    render_attributes;
};

struct Faceset
{
    sequence<Point>             points;
    sequence<Polygon>           polygons;
    OptionalRenderAttributes    render_attributes;
};

typedef sequence<Faceset> FacesetSequence;

enum LightType
{
    AmbientLight,
    DirectedLight,
    PointLight,
    SpotLight
};

struct LightColor
{
    Color   ambient;
    Color   diffuse;
    Color   specular;
};

struct PointLightAttributes
{
    Vertex  position;
    // Vertex       attenuation;
};

struct SpotLightAttributes
```

```
{
    Vertex      position;
    Vertex      direction;
    float       angle;
    float       exponent;
    //Vertex        attenuation;
};

union LightTypeAttributes switch (LightType)
{
    case DirectedLight:
        Vertex                  dl_direction;
    case PointLight:
        PointLightAttributes    pl_attributes;
    case SpotLight:
        SpotLightAttributes     sl_attributes;
};

enum NodeType
{
    SimpleNode,     // all nodes can have childs, so this is kind of a group node
    GeometryNode,   // a node with geometry
    LightNode       // a light
};

struct LightNodeAttributes
{   // common attributes are not in LightTypeAttributes
    float               intensity;
    LightColor          color;
    LightTypeAttributes attributes;
};

union NodeAttributes switch (NodeType)
{
    case GeometryNode:
        FacesetSequence     facesets;
    case LightNode:
        LightNodeAttributes ln_attributes;
};

struct SceneNode
{
    string              name;
    NodeId              id;         // >= 0, unique for the scene
    NodeId              parent_id;  // -1 = no parent
    Transformation      transform;  // transformation affects all children
    NodeAttributes      attributes;
    sequence<NodeId>    childs;
```

```
};

typedef sequence<SceneNode> SceneNodeSequence;

struct Material
{
    MaterialId      id;         // >= 0, must be unique for the scene
    OptionalColor   ambient;
    OptionalColor   diffuse;
    OptionalColor   specular;
    OptionalColor   emission;
    float           shininess;  // default: 1.0
    float           opacity;    // default: 1.0
    //string        name;       // TODO: Do we need a material name?
};

typedef sequence<Material> MaterialSequence;

struct Texture
{
    TextureId       id;         // >= 0, must be unique for the scene
    unsigned long   size_u;
    unsigned long   size_v;
    // For now, we transmit an octet sequence for simplicity.
    // Transmitting sequence< sequence< Color > > could be considered.
    // Transport format: RGBA data, 4 byte / pixel, left-to-right
    // CHECKME: top-to-bottom or bottom-to-top?
    sequence<octet> data;
};

typedef sequence<Texture> TextureSequence;

// For more flexibility on the sending and receiving side, all nodes
// get an ID and are referenced by their ID. This will also ease the
// development of an scene-update-protocol.
struct WholeScene
{
    // These are implicitly associative arrays since each of the items
    // has a unique ID.
    MaterialSequence        scene_materials;
    TextureSequence         scene_textures;
    SceneNodeSequence       scene_nodes;
    NodeId                  root_node;
};

// second part: description of distributed application interfaces

struct TransformationUpdate
```

```
{
    NodeId              scene_object_id;
    Transformation      transform;
};

typedef sequence<TransformationUpdate> TransformationUpdateSequence;

interface ReducedObjectReceiver; // forward declaration

// information valid for the next frame
struct FrameParameters
{
    // wraps around after about 500 days at 100 frames/s
    unsigned long    frame_no;

    // changed object frames
    // TODO: more complex scene updating?
    TransformationUpdateSequence      updates;

    // transformation world coordinates -> modelview coordinates
    Matrix    modelview_matrix;
    // transformation modelview coordinates -> normalized device coordinates
    Matrix    projection_matrix;

    // frame/picture width in pixels
    unsigned long    frame_width;
    // frame/picture height in pixels
    unsigned long    frame_height;
};

struct ReductionRequest
{
    NodeId                    node_id;
    ReducedObjectReceiver     result_receiver;
};

typedef sequence<ReductionRequest> ReductionRequestSequence;

struct ReductionValidityConstraints
{
    // Everything specified here is given in modelview (=camera) coordinates.

    // 1st) Range where this reduction is valid.
    //
    // For validity, the distance between the camera and the object's center
    // needs to be in the given range.
    //
    // Value semantics:
```

```
    // - valid everywhere: no range given
    // - valid only there: range.min = range.max = object's distance from camera
    // - else: valid inside given range
    OptionalRange     range;

    // 2nd) view angle where this reduction is valid.
    //                                                                    330
    // Value semantics:
    // - valid eveywhere: no angle given
    // - valid only there: angle = 0
    // - else: reduction valid iff vector camera-to-object-center
    //         does not deviate more than angle degree from given zero-degree vector
    OptionalViewAngle     view_angle;

    // Specifies whether the reduction somehow depends on the lights
    // affecting the object (e.g. when parts of the object are rendered
    // onto textures or the reduction algorithm takes highlights into account)   340
    boolean               lighting_dependent;
};

interface ObjectReducer
{
    // hand this reducer instance over a copy of the scene we're working on
    boolean LoadMasterScene (
        in WholeScene    scene
        );
                                                                           350
    // update global per-frame information and transformations
    void StartNextFrame (
        in FrameParameters    parameters
        );

    oneway void ReduceObject (
        in ReductionRequest   reduction_request
        );

    void Shutdown ();                                                      360
};

interface ReducedObjectReceiver   // this is where the reductions are sent to
{
    void ReceiveReducedObject (
        in WholeScene                    reduced_object,
        in ReductionValidityConstraints  validity_constraints
        );

};                                                                         370
```

```
    interface ReducerScheduler
    {
        // set parameters for the next frame and update transformations
        void StartNextFrame (
            in FrameParameters   parameters
            );

        void EnqueueReductionRequest (
            in ReductionRequest  reduction_request                    380
            );

        // shortcut for batch operation: enqueue multiple requests at once
        // to save communication overhead
        void EnqueueReductionRequests (
            in ReductionRequestSequence reduction_requests
            );

        // assign this reducer an ID which it uses to send back heartbeats
        // after work completion                                      390
        long RegisterObjectReducer (
            in ObjectReducer      reducer
            );

        // distribute scene data
        boolean LoadMasterScene (
            in WholeScene   scene
            );

        // tell the scheduler that ObjectReducer <id> is ready to perform work   400
        oneway void ReceiveHeartbeat (
            in long id
            );

        // tell scheduler that ObjectReducer <id> shuts down
        // note: uncompleted task might get lost
        void UnregisterObjectReducer (
            in long id
            );
                                                                      410
        void Shutdown ();
    };

};
```

C. Implementation

Die Implementation des vorgestellten Systems zur parallelen, adaptiven Generierung von Objektreduktionen besteht aus ca. 9 000 Zeilen C++ Sourcecode. Zum Gesamtverständnis ist weiterhin eine Dokumentation des CADaVR-Viewers sinnvoll, der aus weiteren 31 000 Zeilen Sourcecode besteht. Im Folgenden werden beide Teile der Implementation näher beschrieben.

C.1. Entwurfskriterien

Die Hauptkriterien für die Implementation sind:

- Portabilität (Plattformunabhängigkeit)
- Wiederverwendbarkeit
  - Unabhängigkeit vom Renderer
  - Unabhängigkeit vom Eingabedatenformat
- Erweiterbarkeit

C.2. Systemkomponenten

Um den Kriterien der Wiederverwendbarkeit und Erweiterbarkeit zu genügen wurde das Gesamtsystem in folgende Hauptkomponenten zerlegt.

C.2.1. Hauptkomponenten

CADaVR-Viewer Eine Applikation zur Visualisierung und Simulation von VR-Szenen, die im CADaVR-Dateiformat vorliegen.

DynVLod_CadvisPlugin Das Plugin für die verteilte Objektreduktion. Die Realisierung als Plugin stellt sicher, daß der CADaVR-Viewer weder von CORBA noch von der verteilten Reduktion abhängig ist (siehe auch Abb. 2.2, Seite 18).

ReducerScheduler Scheduling-Komponente, die anfallende Reduktionsaufträge auf die Reduktionskompoenten verteilt.

ObjectReducer Abstrakte Reduktionskomponente mit Infrastruktur für die Reduktionsverfahren. Auf dieser Komponente basieren die einzelnen Reduktionsverfahren wie z.B. QuadricClusteringReducer.

Die Komponenten für die verteilte Reduktion benötigen zusätzlich eine CORBA-Implementation für die Kommunikation. Zu diesem Zweck wurde TAO ausgewählt.

C.2.2. Komponenten des CADaVR-Viewers

Der CADaVR-Viewer wurde nochmals modularisiert, um sowohl Erweiterbarkeit als auch Plattformunabhängigkeit zu verbessern. So ist der Renderer explizit außerhalb der SceneGraph-Komponente angesiedelt, um einen späteren Austausch des Renderers zu unterstützen.

Common Basisfunktionalitäten aller Module. Infrastruktur für Debugging, Klassen für Berechnungen mit Vektoren, Matrizen und Quaternionen, Smart Pointer[1].

SceneGraph Infrastruktur für die Verwaltung eines Szenegraphs. Beinhaltet keinen funktionsfähigen Renderer, sondern spezifiziert nur das Interface.

[1] Über Templates realisierte C++-Klassen, die sich wie Pointer verhalten, die Speicherverwaltung aber vereinfachen, indem sie z.B. Referenzzähler mitführen und Instanzen automatisch zerstören, wenn diese nicht mehr referenziert werden. Siehe auch C.3.1, Seite 88.

Parser CADaVR-Parser. Liest eine Datei im CADaVR-Format ein und erzeugt Datenstrukturen, die den Dateiinhalt repräsentieren.

CadavrSimulator Das SceneGraph-Modul kann die Daten aus einer CADaVR-Datei nicht direkt verarbeiten. Deshalb konvertiert das CadavrSimulator-Modul die Daten des CADaVR-Parsers und kann die Dynamik entsprechend verarbeiten.

WtkRenderer Eine Implementation eines Renderers für SceneGraph, die WTK als Renderingtoolkit verwendet.

C.3. Beschreibung der Komponenten

Die einzelnen Komponenten (außer das Modul Common) werden zu dynamischen Bibliotheken kompiliert, die je nach Zielplattform im Verzeichnis `src/lib.`*`plattform`* abgelegt werden.

C.3.1. Modul „Common“

Im Modul Common sind einige grundlegende Funktionen vereint: Smart Pointers und Vektorarithmetik. Die meisten Funktionen und Klassen sind dabei mit Hilfe von C++-Templates[2] realisiert.

Smart Pointers

Es werden zwei verschiedene Arten von „Smart Pointer“ zur Verfügung gestellt: TSmartPtr und TRefCountPtr. Die Smart-Pointer-Implementationen sind jeweils mit Templates realisiert, sie hängen also vom Typ der zu verwaltenden Objekte ab.

[2] Eine Technik zur Parametrisierung von Funktionen und Klassen durch Datentypen. So ist die Implementierung einer Klasse für Vektorarithmetik prinzipiell unabhängig vom Datentyp der Vektorkomponenten (in diesem Fall float oder double). Man parametrisiert also die Klasse mit diesem Datentyp und der Compiler generiert bei Bedarf die Implementation je nach benötigtem Basisdatentyp. Die STL (Standard-Template-Library) ist z. B. Teil des C++-Standards und enthält eine umfangreiche Sammlung von Templates, z.B. zur Verwaltung von Listen, assoziativen Arrays etc.

TSmartPtr Mit TSmartPtr lassen sich komfortabel Instanzen verwalten, die nicht immer oder nur temporär benötigt werden. Wird der TSmartPtr zerstört, so wird auch automatisch die verwaltete Instanz zerstört. Wird der TSmartPtr kopiert, so wird auch eine Kopie der verwalteten Instanz erzeugt – ein wichtiger Unterschied zur herkömmlichen Pointer-Semantik. Weiterhin ist TSmartPtr nützlich, wenn Instanzen nicht immer vorhanden sind. Zusammengefasst: TSmartPtr repräsentiert die Instanz eines Objektes und unterstützt die Modellierung des Zustandes „nicht vorhanden". Er darf nicht mit polymorphen Klassen verwendet werden – für diesen Fall existiert TClonedSmartPtr, der einen virtuellen Copy-Konstruktor voraussetzt.

TRefCountPtr TRefCountPtr dient der Verwaltung von Instanzen, die mehrfach referenziert werden können. Es wird ein Referenzzähler außerhalb der Objektinstanz geführt. Erreicht der Referenzzähler Null, so wird die Instanz zerstört. Hauptnachteil dieses Smart Pointers ist die Realisierung des Referenzzählers ausserhalb der Objektinstanz. So darf ein herkömmlicher Pointer nur genau einmal in einen TRefCountPtr überführt werden, weil sonst zwei unabhängige Referenzzähler für die gleiche Instanz erzeugt werden, was in jedem Fall zur vorzeitigen Zerstörung der Instanz führt.

Diesen Nachteil behebt die ReferencedPtr-Implementation innerhalb des SceneGraph-Moduls, jedoch mit dem Nachteil, daß alle referenzzählbaren Objekte von der Klasse ReferenceCounted abgeleitet sein müssen.

Vektorarithmetik

Vektoren bilden die Basis für polygonale Objektmodellierung. Um den Umgang mit Vektoren möglichst komfortabel zu gestalten, stehen drei Templates zur Verfügung: ctVertex, ctMatrix und ctQuaternion. Alle Klassen sind mit dem Basisdatentyp parametrisiert, so daß man je nach Bedarf zwischen einem Vektor mit Komponenten des Typs float oder double wählen kann. Die Klassen überladen die entsprechenden Operatoren um eine intuitive Verwendung zu erlauben.

C.3.2. Modul „SceneGraph"

Das SceneGraph-Modul besteht aus ca. 40 Klassen, die verschiedene Aspekte eines Szenegraph implementieren. An dieser Stelle sollen nur die wich-

tigsten Teile und grundlegende Funktionen des Moduls beschrieben werden – für die detailierte Interface-Dokumentation sei auf die generierte Dokumentation im Verzeichnis `Devel-Docs/Doxygen` und den Sourcecode verwiesen.

Watchable und Trigger

Da z.B. der Renderer nicht integraler Bestandteil des SceneGraph-Moduls ist, muß es eine Möglichkeit geben, diesem Veränderungen am Szenegraph effizient mitzuteilen. Zu diesem Zweck wird ein Callback-System benötigt. Dieses ist mit Hilfe der Templates Watchable und Trigger bzw. Trigger2 realisiert. Watchable dient dabei als Basisklasse für „überwachbare“ Klassen und stellt grundlegende Funktionen und Typdefinitionen bereit, z.B.:

```
#include "SceneGraph/ReferenceCounted.hh"
#include "SceneGraph/ReferencedPtr.hh"
#include "SceneGraph/DestructionWatchable.hh"

namespace SceneGraph
{

class Object;

class Object
:   public ReferenceCounted,
    public DestructionWatchable<Object>
{
public:
    typedef Watchable<Object> watchable_type;
    typedef Watchable<Object>::trigger::type trigger_type;

    typedef ReferencedPtr<Object> child_pointer;
// ...
    /** trigger used to pass the parent as well as the affected child */
    typedef Watchable<Object>::trigger2<child_pointer>::type child_trigger_type;

    /** Add trigger to be called before object is removed from scene graph. ... */
    void AddTrigger_Removal (trigger_type* trigger)
    { Watchable<Object>::AddTrigger (mRemovalTriggers, trigger); };

    void RemoveTrigger_Removal (trigger_type* trigger)
    { Watchable<Object>::RemoveTrigger (mRemovalTriggers, trigger); };
// ...
    /** Add trigger to be called by AddChild. ... */
```

```
    void AddTrigger_AddChild (child_trigger_type* trigger)
    { Watchable<Object>::AddTrigger2<child_pointer> (mAddChildTriggers,
            trigger); };

    void RemoveTrigger_AddChild (child_trigger_type* trigger)
    { Watchable<Object>::RemoveTrigger2<child_pointer> (mAddChildTriggers,
            trigger); }
// ...
protected:
    /** triggers called on removal of object */                    40
    Watchable<Object>::trigger::list_type          mRemovalTriggers;

    /** triggers called on addition of a child */
    Watchable<Object>::trigger2<child_pointer>::list_type          mAddChildTriggers;

};

}   // namespace SceneGraph
```

Trigger und Trigger2 stellen eine Art Funktionsobjekt dar, dessen virtuelle Funktion `Notify` zur Benachrichtigung aufgerufen wird. `Trigger::Notify` erhält dabei als Parameter nur einen Pointer zum auslösenden Objekt während `Trigger2::Notify` ein zusätzliches Argument übergeben wird, dessen Typ ein Templateparamter von Trigger2 ist. Um die Verwendung von Triggern zu vereinfachen existieren die Klassen mem_fun_trigger_t und mem_fun2_trigger_t. Diese erlauben es, eine beliebige Funktion einer anderen Klasse als Trigger-Ziel anzugeben. Die Hilfsfunktionen `mem_fun_trigger` und `new_mem_fun_trigger` erleichtern die Verwendung noch weiter (und sparen eine Menge Tipparbeit für die u.U. recht umfangreichen Templateparamter):

```
// ** Auszug WtkRenderer/Renderer.hh **

#include "SceneGraph/Object.hh"
#include "SceneGraph/Renderer.hh"

namespace Wtk
{
// ...

class Renderer                                                     10
:   public SceneGraph::Renderer
{
public:
    /** Default constructor. ... */
```

```
    explicit Renderer (SceneGraph::Object::pointer root_node);
// ...
    /** Trigger to be called when a child is added to an object. ... */
    void AddChild_trigger_callback (SceneGraph::Object* parent,
        SceneGraph::Object::child_pointer child
        );

private:
// ...
    SceneGraph::Object::child_trigger_type*    mAddChildTrigger;
// ...
}; // class Renderer

} // namespace Wtk

// ** Auszug WtkRenderer/Renderer.cc **

namespace Wtk
{
// ...

Renderer::Renderer(SceneGraph::Object::pointer root_node)
:   mAddChildTrigger (
        SceneGraph::new_mem_fun_trigger (    // create trigger once
            &Renderer::AddChild_trigger_callback, this)
        ),
// ...
{
// ...
}

// ...
// add same trigger to multiple objects
obj.AddTrigger_AddChild (mAddChildTrigger);
// ...

void
Renderer::AddChild_trigger_callback (SceneGraph::Object* parent,
    SceneGraph::Object::child_pointer child)
{
// ...
}

// ...

} // namespace Wtk
```

Dieses Callback-System hat sich in der Praxis bewährt, obowhl einige Wünsche offen bleiben. So wäre es von Vorteil, wenn ein Trigger vorübergehend blockiert werden könnte, um z.B. umfangreiche Änderungen an den überwachten Daten vorzunehmen. Weiterhin müssen die Trigger immer explizit ausgelöst werden und die Funktionen zum Hinzufügen von Triggern müssen explizit deklariert werden (die Funktionalität wird jeweils von Watchable zur Verfügung gestellt, so daß die Implementationen der Funktionen Einzeiler sind). Außerdem müssen die Zugriffe auf überwachte Daten komplett kontrolliert werden – es dürfen also keine nicht-konstanten Referenzen auf ein überwachtes Datum aus der überwachenden Klasse herausgegeben werden (dies ist auch sonst zu vermeiden, da sonst u.U. Invarianten der Klasse durch externe Modifikation zerstört werden können). Es scheint mit C++-Mitteln keine bessere Lösung zu geben. Die Alternative, für jeden zu überwachenden Datentyp eine Proxy-Klasse zu schreiben und direkten Zugriff darauf zu gewähren, kann unter den Gesichtspunkten der Kapselung und des Aufwandes nicht befürwortet werden.

Grundlegende Eigenschaften eines Szeneknotens

Die Knoten des Szenegraph sind Instanzen der Klasse Object oder davon abgeleiteter Klassen. Jeder Knoten hat eine eindeutige Identifikationsnummer (ID, vorzeichenlose Ganzzahl), die zum Beispiel für die Assoziation von zusätzlichen externen Informationen verwendet werden kann. Weiterhin bietet die Basisklasse Object eine Menge von Standardfunktionen für die Verwaltung des Szenegraph: Hinzufügen / Entfernen von Kindknoten, Hinzufügen / Entfernen von Triggern für verschiedenste Modifikationen, Ermittlung der Bounding Box und Bounding Sphere, Zugriff auf die Kindknoten.

Die Basisklasse Object ist von SceneGraph::ReferenceCounted abgeleitet und kann somit für ReferencedPtr benutzt werden. In der Tat werden die Kindknoten als ReferencedPtrs gespeichert, so daß sie automatisch zerstört werden, wenn keine Referenz mehr darauf existiert. Herkömmliche Pointer auf Knoten sollten also nur temporär gehalten werden (eine Ausnahme ist der Pointer zum Elternknoten, da sonst eine zirkuläre Referenz entstehen würde, die eine Zerstörung der Knoten verhindert).

Visitor-Pattern für Szenegraph-Traversierung

Für Anwendungen, die eine Hierarchie polymorpher Klassen beinhalten, stellt sich oft die Frage, wie man den Typ einer Instanz ermittelt. Die Kindknoten eines Object werden immer als ReferencedPtr auf ein Object gespeichert. Um zu vermeiden, daß an vielen Stellen im Sourcecode umfangreiche `if`-Kaskaden verstreut werden, die mit Hilfe von `dynamic_cast` den Typ der Instanz ermitteln, wird das Visitor-Pattern verwendet.

Das Visitor-Pattern funktioniert wie folgt: Um eine Operation auf einem Knoten auszuführen, die vom Knotentyp abhängig ist, wird die virtuelle Methode `accept` der Instanz mit dem Visitor als Parameter aufgerufen. Die `accept`-Funktion „kennt“ den Typ ihrer Instanz und ruft wiederum direkt die virtuelle Methode `apply` des Visitors mit einer Referenz auf ihre Instanz als Paramter auf. Der Visitor hat für jeden ihn interessierenden Knotentypen eine eigene `apply`-Methode. Die Standardimplementation (in der Klasse NodeVisitor) der `apply`-Methode ruft entsprechend der Vererbungshierarchie die `apply`-Methode für die übergeordnete Klasse; die Standard-`apply`-Methode für Object hat keine Funktion.

Nachteilig am Visitor-Pattern ist, daß alle Knotentypen innerhalb der NodeVisitor-Klasse eine entsprechende `apply`-Methode benötigen (dies ist jedoch gleichzeitig ein Vorteil – die möglichen Knotentypen sind an genau einer Stelle konzentriert und nur dort müssen neue Typen ergänzt werden) und daß die `accept`-Methode jedes Knotentyps explizit reimplementiert werden muß (die Implementation ist jedoch für jede Klasse identisch, deshalb existiert dafür ein Präprozessor-Makro). Dies ließe sich durch das Pattern „Make non-leaf classes abstract“ erzwingen, jedoch wurde davon aus Gründen der Übersichtlichkeit abgesehen.

C.3.3. Modul „Parser“

Das Modul Parser müßte genauer CadavrParser heißen, aus historischen Gründen wurde der Name jedoch beibehalten. Der Parser besteht klassich aus zwei Komponenten: Dem Lexer und dem Parser selbst. Der Lexer, realisiert mit Hilfe von flex++, zerlegt den Eingabedatenstrom in Token, die vom Parser, realisiert mit bison++ gemäß der CADaVR-Grammatik weiterverarbeitet werden.

Sowohl flex++ als auch bison++ sind verhältnismäßig alte Programme. Sie unterstützen deshalb leider keine Namespaces für den generierten Code und es ist notwendig, den generierten Lexercode automatisch

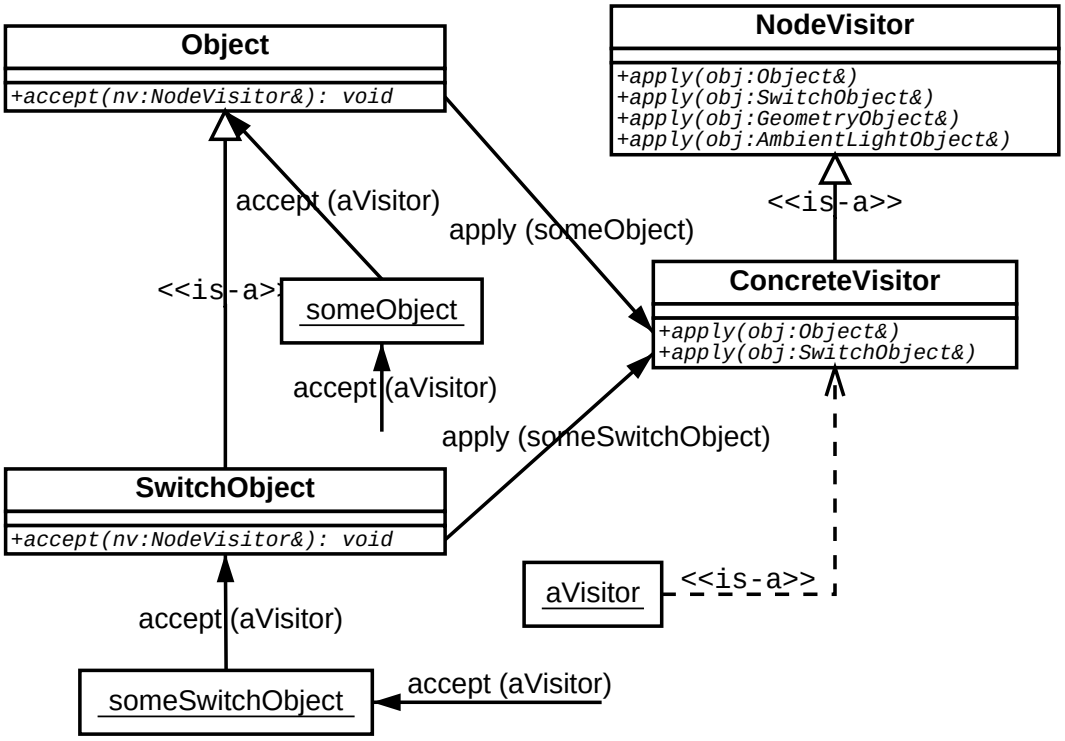


Abbildung C.1.: Das Visitor-Pattern. Der genaue Typ von `someSwitchObject` und `someObject` muß zur Compilierungszeit nicht bekannt sein. Es wird die dem Typ entsprechende `apply`-Methode von ConcreteVisitor aufgerufen. Darstellung in UML [16].

nachzubearbeiten, damit dieser ANSI C99-konform wird (siehe Dateien `src/Parser/cleanup-lexer.sed` und `src/Parser/Makefile`).

Weiterhin waren einige Kniffe notwendig, da sich mit bison++ theoretisch nur kontextfreie Grammatiken verarbeiten lassen, die CADaVR-Grammatik jedoch an drei Stellen kontextsensitiv ist: Für Objektgeometrien wird jeweils die Anzahl der Punkte und Polygone vor deren Definition angegeben, es gibt keine weiteren Trenner, die das Ende der Definition signalisieren. Weiterhin ist die Angabe von Objektreferenzen für Ereignisattribute mehrdeutig: Diese können als Ganzzahl, Gleitkommazahl oder Zeichenkette interpretiert werden. Um dieses Problem zu lösen existiert ein spezielles Interface zum Lexer, über das dieser instruiert werden kann, in diesen Fällen einen anderen Tokentyp zu liefern (Variablen `expect_pointidx`,

`expect_points`, `expect_qualifier` der Klasse cCadavrParser, deklariert in der Datei `src/Parser/cadavr_parser.y`).

Die Ausgabe des Parsers ist eine Repräsentation der CADaVR-Datei durch die Datenstrukturen aus `src/Parser/Filestruct`. Das Parser-Modul hängt nur vom Modul Common ab. Siehe Abb. C.2 für ein Beispiel zur Verwendung des Moduls.

```
#include <iostream>

#include "Parser/parser.hh"

int main (argc, char **argv)
{
    cCadavrFile *new_object;

    if (argc <= 1) // parse stdin if no arguments
    {
        new_object = new cCadavrFile (std::string());
    }
    else
    {
        new_object = new cCadavrFile (std::string (argv[1]));
    }

    if (! new_object->Parse ())
    {
        std::cerr << "Parsing failed." << std::endl;
        return 1;
    }

    // output whole file again
    std::cout << *new_object << std::endl;

    return 0;
}
```

Abbildung C.2.: Ein Programm, das eine CADaVR-Datei parst und die geparsten Daten wieder im CADaVR-Format ausgibt.

C.3.4. Modul „CadavrSimulator“

Um das SceneGraph-Modul prinzipiell unabhängig vom Format der Eingabedaten für die VR-Simulation zu halten, wurde besonderer Wert darauf ge-

legt, keine CADaVR-spezifischen Datenstrukturen darin aufzunehmen (lediglich abstrakte Interfaces, wie z.B. SceneGraph::Simulator, die auch für andere Anwendung notwendig sein könnten, werden zur Verfügung gestellt). Alle CADaVR-spezifischen Funktionen (außer dem Parser) sind im Modul CadavrSimulator vereint. Das Modul deklariert alle seine Klassen im Namespace CADaVR.

Das Modul beinhaltet drei Klassen: Converter, Simulator und MaterialFile. Die Klasse Converter ist für die Überführung von CADaVR-Daten (erzeugt von Parser) in entsprechende SceneGraph-Datenstrukturen zuständig. Parallel dazu werden einer Instanz von Simulator alle notwendigen Informationen für die Szenedynamik geliefert.

C.3.5. Modul „WtkRenderer“

Das WtkRenderer beinhaltet einen Renderer, der WTK als Renderingtoolkit verwendet. Die Abstraktion des Renderers ist leider nicht so weit fortgeschritten, daß das Hauptprogramm unabhängig vom verwendeten Renderer ist. Deshalb ist der eigentliche CADaVR-Viewer Teil dieses Moduls (Programm `wtkrendertest`). Wenn in Zukunft weitere Renderer hinzugefügt werden, muß die Schnittstelle zwischen SceneGraph und Renderer weiter ausgebaut und abstrahiert werden. Auch die Behandlung von Eingabegeräten müßte weiter verbessert werden – derzeit liefert das Wtk::Window die Eingabeereignisse.

C.3.6. Modul „DynVLod_CadvisPlugin“

Dieses Modul ist die Schnittstelle zwischen CADaVR-Viewer und der verteilten Applikation zur Objektreduktion. Seine Funktionen teilen sich auf zwei Klassen auf: DynVLodObject und ObjectManager.

ObjectManager

Die Klasse ObjectManager verwaltet Daten, die alle reduzierbaren Objekte gemeinsam benötigen. So stellt sie zum Beispiel die Verbindung zum ReducerScheduler her, konvertiert den Szenegraph in die entsprechende Repräsentation zum Versand im Netzwerk, integriert die DynVLodObject Instanzen in den Szenegraph, nimmt Reduktionsaufträge entgegen und übermittelt die Parameter für das nächste Bild.

Eine Instanz dieser Klasse wird automatisch beim Laden des Plugins erzeugt und klinkt sich dann in den Renderer ein um aktiviert zu werden. Dieses Vorgehen macht eine explizite Aktivierung des Plugins durch das Hauptprogramm überflüssig.

DynVLodObject

DynVLodObject ist eine von SceneGraph::SwitchObject abgeleitete Klasse, die ein automatisch reduzierbares Objekt repräsentiert.. Der ObjectManager veranlasst bei der ersten Aktivierung die Ersetzung aller LodObject Knoten im Szenegraph durch eine Instanz von DynVLodObject (siehe Methode `DynVLodObject::takeOverObject`). Wird durch den CullVisitor die Sichtbarkeit eines Objektes festgestellt, so sucht diese Klasse im Objektcache nach einer geeigneten Reduktion und bestellt bei Bedarf eine neue. Sie ist außerdem für den Empfang der Reduktion (im CORBA-Sinn ist sie ein DynVLod::ReducedObjectReceiver) und deren Integration in den Szenegraph (als Kindknoten) zuständig.

C.3.7. Modul „ReducerScheduler“

ReducerScheduler ist im eigentlichen Sinn nur ein Interface, beschrieben in CORBA IDL. Die Implementation DynVLod_ReducerScheduler_i fungiert als zentraler Anlaufpunkt für alle anderen Teile der verteilten CORBA-Applikation. Der ReducerScheduler registriert seine Instanz beim CORBA-NameService und die anderen Komponenten (ObjectReducer und das Plugin) ermitteln jeweils über diesen Namensdienst eine Referenz auf den ReducerScheduler.

Das Plugin kommuniziert nur direkt mit dem Scheduler – von den Reduktionskomponenten erhält es ausschließlich die reduzierten Objekte (das Zielobjekt für die Reduktion ist Teil des Reduktionsauftrages). Der Scheduler entkoppelt also den CADaVR-Viewer von den Reduktionskomponenten und übernimmt deren Verwaltung (Versorgung mit den Szenedaten und Bildparametern, Verteilung der Reduktionsaufträge).

Das Progarmm `reducerschedulerserver` erzeugt lediglich eine Instanz von DynVLod_ReducerScheduler_i und Threads für die CORBA-Implementation. Weiterhin ermöglicht es den kontrollierten Stop der Applikation (via `STRG-C` oder `kill`).

C.3.8. Modul „ObjectReducer“

ObjectReducer ist ebendfalls nur ein Interface, beschrieben in IDL. Die Implementation DynVLod_ObjectReducer_i führt noch keine Reduktion durch, sondern stellt gemeinsame Funktionen für alle Reduktionsalgorithmen zur Verfügung (z.B. Ermittlung der Objektreferenz zum ReducerScheduler, Ermittlung der Projektion eines Vektors in den Bildraum, Berechnung von Bounding Boxes). Ein Reduktionsalgorithmus kann in einer abgeleiteten Klasse implementiert werden und muß nur die Methode `PerformReduction` überschreiben.

Die eigentlichen Reduktionsalgorithmen bestehen meist nur aus einer von DynVLod_ObjectReducer_i abgeleiteten Klasse. Jeder Algorithmus wird zu einem Plugin kompiliert, welches vom Programm `objectreducerserver` geladen wird. Beim Laden eines Plugins wird automatisch eine Instanz der Klasse des Algorithmus erzeugt, die sich bei ObjectReducerManager registriert (siehe Template-Funktion `createAndInitializeObjectReducer` in `Implementation/ObjectReducer/ObjectReducer.h`).

ObjectReducerManager ist eine Klasse, die die Realisierung der Reduktionsalgorithmen als Plugins unterstützt. Sie ist nach dem Singleton-Pattern entworfen, das heißt von dieser Klasse existiert immer nur eine Instanz. Alle aktiven Reduktionskomponenten innerhalb einer Applikation (d.h. aktive Instanz von `objectreducerserver`) registrieren sich bei ObjectReducerManager.

Diese Registrierung ist notwendig, um ein kontrolliertes Beenden der Applikation zu ermöglichen. Eine Reduktionskomponente kann auf zwei Wegen beendet werden: Durch den ReducerScheduler oder durch den Anwender. Im ersten Fall muß das Hauptprogramm feststellen können, daß kein Reduzierer mehr aktiv ist, während im letzteren Fall der Scheduler informiert werden muß.

C.4. Zusammenwirken der Komponenten

Die Interaktion zwischen den einzelnen Komponenten ist recht umfangreich. Stellvertretend sollen an dieser Stelle folgende Szenarien vorgestellt werden:

1. Laden und Darstellen einer CADaVR-Szene
2. Reduktion eines Objektes

C.4.1. Laden und Darstellen einer CADaVR-Szene

Das Laden und Darstellen einer Szene im CADaVR-Dateiformat läßt sich durch folgenden Pseudocode beschreiben:

```
bool load_and_display ()
{
    // 0. leeren Szenegraph erzeugen
    SceneGraph::Object::pointer root_node = new SceneGraph::Object ("root");

    // 1. Datei parsen
    cCadavrFile* parsed_file = new cCadavrFile (filename);

    if (! parsed_file->Parse ()) { return false; }

    // 2. Konvertierung mittels Modul CadavrSimulator
    CADaVR::Simulator* simulator = new CADaVR::Simulator();

    //    Konverter erzeugt Szenegraph und initialisiert Simulator
    CADaVR::Converter* converter =
        new CADaVR::Converter (parsed_file, root_node, simulator);

    if (! converter->Convert ()) { return false; }

    // 3. Simulator vollständig initialisieren
    //    (Objektreferenzen für Dynamik auflösen etc.)
    if (! simulator->PrepareSimulation ()) { return false; }

    // 4. Fenster erzeugen
    Wtk::Window* window = new Wtk::Window ();

    // 5. Renderer erzeugen und initialisieren
    Wtk::Renderer* renderer = new Wtk::Renderer (root_node);

    window->renderer (renderer);
    window->rootNode (root_node);

    // 6. Kamera erzeugen
    SceneGraph::ReferencedPtr<SceneGraph::Camera> camera =
        new SceneGraph::Camera();

    // 7. Interaktion mit der Kamera ermöglichen
    SceneGraph::WalkCameraManipulator camera_manipulator (camera);
    window->registerHandler (&camera_manipulator);

    simulator->Restart ();
```

```
    // 8. Simulation durchführen
    do
    {
        simulator->Simulate (current_time);
        window->redraw (); // ruft renderer->Render (root_node, camera, window)
    } while (! quit)

    return true;
}
```

Während dieser Teil noch recht übersichtlich ist, passiert intern eine Menge mehr. Sowohl der Simulator als auch der Renderer müssen für die Objekte (= Knoten im Szenegraph) Zusatzinformationen verwalten. So muß der WTK-Renderer zum Beispiel den WTK-spezifischen Szenegraph konstruieren und aktuell halten (dabei kann es vorkommen, daß ein Knoten im Original-Szenegraph mehr als einem Knoten im WTK-Szenegraph entspricht). Der Simulator muß außerdem die Dynamik-spezifischen Datenstrukturen (Tasks, Events etc.) verwalten.

Für das Hinzufügen knotenspezifischer Informationen gibt es zwei prinzipielle Möglichkeiten: Die Verwaltung innerhalb oder außerhalb der Knoten. WTK bietet zu diesem Zweck ein sogennantes Property-System, welches es dem Anwender (Programmierer) ermöglicht, einem Objekt zusätzliche Informationen zuzuordnen. Der Hauptnachteil dieses Ansatzes ist die fehlende Typsicherheit – der Typ der zugeordneten Information muß extra verwaltet werden und es wird immer nur ein `void*`-Pointer gespeichert, es sind also fehlerträchtige Typecasts notwendig.

Dieses Vorgehen widerspricht dem prinzipiellen Programmierstil in dieser Implementation. Hier wird die Philosophie verfolgt, daß möglichst viele Fehler vom Compiler erkannt werden (getreu dem Motto: Wenn das Programm kompiliert ist es wahrscheinlich korrekt). Die Verwaltung knotenspezifischer Information wird also durch die jeweilige Komponente realisiert. Um dies zu vereinfachen erhält jeder erzeugte Knoten eine zur Laufzeit eindeutige Nummer (vorzeichenlose Ganzzahl). Werden zu einem Knoten zusätzliche Daten benötigt, so kann die Verbindung Knoten-zu-Daten mit Hilfe eines assoziativen Arrays hergestellt werden (die STL stellt dafür z.B. den Datentyp std::map bereit). Siehe dazu auch `CadavrSimulator/Simulator.hh`, Datentyp `tNode2DataMap` und Member-Variable `mNodes`.

C.4.2. Reduktion eines Objektes

Das Rendering wird in zwei Schritte unterteilt: Culling und Rendering. Vor dem Culling löst der Renderer die `preCulling`-Trigger aus. Der ObjectManager hat genau diesen Trigger bestellt und wird somit über den Beginn eines neuen Bildes informiert. Ist die Reduktion aktiviert, so wird beim ersten Aufruf der Szenegraph für die Reduktion vorbereitet.

Renderer erzeugt einen CullVisitor, der den Szenegraph traversiert und für alle Objekte, die potenziell sichtbar sind (deren Bounding Box nicht komplett außerhalb der Sichtpyramide liegt), wird die Methode Object::CallCullTriggers aufgerufen. DynVLodObject überschreibt diese Methode und sucht eine passende Reduktion bzw. fordert eine neue via ObjectManager an.

Nachdem der CullVisitor den Szenegraph traversiert hat, wird vom Renderer der `preRender`-Trigger ausgelöst. Auch diesen Trigger hat ObjectManager angefordert und wartet jetzt, bis alle Reduktionen eingetroffen sind.

Ein reduziertes Objekt wird direkt beim anfordernden DynVLodObject abgeliefert (via Methode `ReceiveReducedObject`). Das Objekt wird in den Szenegraph eingefügt und der ObjectManager benachrichtigt.

C.5. Zusammenfassung

Die vorgestellte Implementation wurde unter den Gesichtspunkten der Plattformunabhängigkeit, Wiederverwendbarkeit und Erweiterbarkeit entwickelt. Es ergab sich ein modulares Design, welches den CADaVR-Viewer nicht abhängig von der verteilten Reduktion macht. Die Hauptkomponenten für die verteilte Reduktion sind das Plugin für den Viewer, der Scheduler und Infrastruktur für die Reduktionsalgorithmen.

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

ACE	Adaptive Communication Environment. Ein objektorientiertes, plattformübergreifendes C++ Toolkit für die Netzwerkprogrammierung, wie TAO am „Center for Distributed Object Computing“ entwickelt. Siehe [22].
AR	Augmented Reality. „Überlagerte Realität.“ Die AR befaßt sich mit der Überlagerung von realen durch virtuelle Bilder. Ein beliebtes Beispiel für eine AR-Anwendung ist ein Wartungstechniker, dem vom Rechner ein Schema einer komplexen Maschine in das Sichtfeld eingeblendet wird. Auch die Instrumenteneinblendungen im Cockpit von Kampfflugzeugen sind der AR zuzurechnen.
Bounding-Box	Kleinster Quader, der ein Objekt komplett einschließt. Aus Effizienzgründen sind die Bounding Boxes meist achsenparallel. Verwendet werden Bounding Boxes zum Beispiel zur Vereinfachung von Kollisionserkennung oder für die Markierung von Objekten zur Interaktion.
CAD	Computer Aided Design. (Computerunterstützter Entwurf). Oberbegriff für Soft- und Hardware, die für den Entwurf von Gegenständen aller Art eingesetzt wird.
CADaVR	Eine textbasierte Schnittstelle für die Beschreibung virtueller, dynamischer Welten. Siehe auch [29].
CORBA	Common Object Request Broker Architecture. Standard der OMG (`http://www.omg.org/`) für verteilte, objektorientierte Anwendungen. Zum CORBA-Standard zählen unter anderem die IDL (Interface Description Language) und das Protokoll IIOP. Siehe [17].

HMD — Head Mounted Display. Ein Anzeigegerät, daß wie eine Brille getragen wird und für jedes Auge ein anderes Bild darstellt, wodurch ein realistischer 3D-Eindruck erweckt werden kann.

IIOP — Interoperable Inter ORB Protocol. Ein im CORBA-Standard spezifiziertes Protokoll zur Kommunikation zwischen ORBs. Zweck des Protokolls ist es, die Interoperabilität von ORBs verschiedener Hersteller zu gewährleisten. IIOP trägt dazu bei, daß in den Komponenten einer verteilten Applikation gleichzeitig verschiedene CORBA-Implementierungen eingesetzt werden können. Siehe [17].

IOR — Interoperable Object Reference. Innerhalb des CORBA-Standards definierte Schnittstelle zur Beschreibung einer Objektreferenz. In einer IOR sind sowohl die Adresse als auch das Protokoll für den Zugriff auf ein Objekt kodiert. Dabei kann ein Objekt über verschiedene Protokolle und Adressen gleichzeitig erreichbar sein.

LOD — Level-of-Detail. Oberbegriff für eine Technik zur Beschleunigung von 3D-Darstellungen. Die Grundidee ist, daß ein komplexes Objekt durch ein viel einfacheres ersetzt werden kann, wenn die Details vom Beobachter sowieso nicht erkennbar wären, weil das Objekt zu weit entfernt ist und somit nur durch wenige Pixel repräsentiert wird. Ein Haus könnte also zum Beispiel ab einer bestimmten Entfernung als einfacher Quader approximiert werden oder bei einem Fahrzeug müßte nicht mehr die komplette Inneneinrichtung in die Darstellung einbezogen werden.

MPI — Message Passing Interface, ein Standard für die Kommunikation von verteilten Programmen. Es existieren mehrere hochoptimierte, freie MPI-Implementationen, u.a. MPIch (`http://www-unix.mcs.anl.gov/mpi/mpich/`) und LAM-MPI (`http://www.lam-mpi.org/`).

Raytracing — Verfahren zur Visualisierung von Objekten. Für jeden Pixel des zu berechnenden Bildes wird mindestens ein Strahl aus Betrachtersicht im virtuellen Raum platziert und dessen Weg durch die Szene verfolgt. Die Farbe des Pixels er-

gibt sich dann aus den geschnittenen Objekten, wobei Reflexionen und Transmissionen rekursiv zu berücksichtigen sind. Siehe auch Rendering.

Rendering — Abbildung einer Modelldatenbasis in eine bildliche Darstellung. In dieser Arbeit wird der Begriff „Rendering" im Sinne von „Bilderzeugung durch Rasterisierung" verstanden, d.h. objekt- bzw. polygonweise Konvertierung der Szenedaten in Pixel. Siehe auch Raytracing.

Shutterbrille — Eine Alternative zum HMD. Eine Brille, die computergesteuert synchron zum Anzeigegerät (z. B. Monitor) abwechselnd das linke oder rechte Auge des Betrachters verdeckt. Dadurch kann jedem Auge ein anderes Bild angezeigt und ein 3D-Eindruck erweckt werden. Vorteil: Wenn die Synchronisierung per Funk oder Infrarot erfolgt, können mehrere Betrachter die gleiche Szene 3-dimensional betrachten, zum Beispiel wenn diese an eine Wand projiziert wird.

Smart Pointer — Über Templates realisierte C++-Klassen, die sich wie Pointer verhalten, die Speicherverwaltung aber vereinfachen, indem sie z.B. Referenzzähler mitführen und Instanzen automatisch zerstören, wenn diese nicht mehr referenziert werden. Siehe auch C.3.1, Seite 88.

TAO — The ACE ORB, eine CORBA-Implementierung, die am „Center for Distributed Object Computing" der Washington University in St. Louis' begonnen wurde. Siehe [23].

Templates — Eine Technik zur Parametrisierung von Funktionen und Klassen durch Datentypen. So ist die Implementierung einer Klasse für Vektorarithmetik prinzipiell unabhängig vom Datentyp der Vektorkomponenten (in diesem Fall float oder double). Man parametrisiert also die Klasse mit diesem Datentyp und der Compiler generiert bei Bedarf die Implementation je nach benötigtem Basisdatentyp. Die STL (Standard-Template-Library) ist z. B. Teil des C++-Standards und enthält eine umfangreiche Sammlung von Templates, z.B. zur Verwaltung von Listen, assoziativen Arrays etc.

Literaturverzeichnis

[1] Andújar, Carlos. 1999. *Geometry Simplification*. Universitat Politècnica de Catalunya.

[2] Crockett, Thomas W. 1995. *Parallel Rendering*. Institute for Computer Applications in Science and Engineering NASA, Langley Research Center. ICASE Report No. 95-31. `http://www.icase.edu/~tom/95-31.pdf`

[3] Cruz-Neira, C. / Sandin, D. / DeFanti, T. 1992. *The CAVE: A Virtual Reality Theater*. Homepage. Stand 14.08.2002. `http://www.evl.uic.edu/pape/CAVE/oldCAVE/CAVE.html`.

[4] Erikson, Carl. 1996. *Polygonal Simplification: An Overview*. University of North Carolina, Chapel Hill. Technical Report TR96-016.

[5] Gaede, Volker / Günther, Oliver. 1996. *Multidimensional Access Methods*. Humboldt-Universität zu Berlin. Technical Report TR96-043.

[6] Garland, Michael / Heckbert, Paul S. 1997. *Surface Simplification using Quadric Error Metrics*. SIGGRAPH '97. `http://graphics.cs.uiuc.edu/~garland/research/quadrics.html`

[7] Garland, Michael / Heckbert, Paul S. 1998. *Simplifying Surfaces with Color and Texture using Quadric Error Metrics*. IEEE Visualization 98.

[8] Garland, Michael. 1999. *Quadric-Based Polygonal Surface Simplification*. Doktorarbeit. Carnegie Mellon University. 09.05.1999. `http://graphics.cs.uiuc.edu/~garland/research/thesis.html`

[9] Garland, Michael. 1999. *Multiresolution Modeling: Survey & Future Opportunities*. EUROGRAPHICS '99 State of The Art Reports.

[10] Heckbert, Paul S. / Garland, Michael. 1997. *Survey of Polygonal Surface Simplification Algorithms*. Multiresolution Surface Modeling Course, SIGGRAPH '97.

[11] Hilbert, Karsten. 2003. *Texturierungsverfahren*. Studienarbeit. Technische Universität Chemnitz.

[12] Hoppe, Hugues. 1999. *New Quadric Metric for Simplifying Meshes with Appearance Attributes*. IEEE Visualization 1999, October 1999, pages 59-66.

[13] Hübner, Uwe. 2001. *Untersuchungen zur Kommunikation am CLiC*. Homepage. Stand 14.08.2002. `http://www-user.tu-chemnitz.de/~huebner/cliccom/`.

[14] Lindstrom, Peter. 2000. *Out-of-Core Simplification of Large Polygonal Models*. In: Proceedings of ACM SIGGRAPH 2000.

[15] Low, K.-L. / Tan, T.-S. 1997. *Model simplification using vertex-clustering*. In: Proceedings of the Symposium on Interactive 3D Graphics. ACM Press. New York.

[16] Object Management Group. 2000. *OMG Unified Modeling Language Specification*. Version 1.3. March 2000.

[17] Object Management Group. 2002. *The Common Object Request Broker: Architecture and Specification*. Revision 2.6.1. May 2002.

[18] Pasman, W. / Jansen, F. W. 2001. *Distributed Low-latency Rendering for Mobile AR*. Delft University of Technology.

[19] Professur für Graphische Datenvearbeitung und Visualisierung. 2002. *Hewlett-Packard Visualize Center II*. Homepage. Stand 24.01.2003. `http://www.informatik.tu-chemnitz.de/~gdv/new/Ausstattung/Aus/VCenter.html`.

[20] Rossignac, Jarek / Borrel, Paul. 1993. *Multi-resolution 3D approximations for rendering complex scenes*. In: Falcidieno, I. und Kunii, T., Hrsg. Modeling in Computer Graphics: Methods and Applications. Seiten 455-465. Springer-Verlag. Berlin. 1993.

[21] Schaufler, G. / Stürzlinger, W. 1996. *A Three Dimensional Image Cache for Virtual Reality*. In: EUROGRAPHICS '96 Proceedings. Seiten 227-236.

[22] Schmidt, Douglas C. *The Adaptive Communication Environment*. Homepage. Stand 24.01.2003. `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[23] Schmidt, Douglas C. *TAO - The ACE ORB*. Homepage. Stand 24.01.2003. `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[24] Schroeder, William J. / Zarge, Jonathan A. / Lorensen, William E. 1992. *Decimation of Triangle Meshes*. General Electric Company, Schenectady, NY.

[25] Schwarze, Tino. 2002. *Paralleles, adaptives Level-of-Detail für VR-Simulationen*. Studienarbeit. Technische Universität Chemnitz.

[26] Taubin, Gabriel / Rossignac, Jarek. 1998. *Geometric Compression Through Topological Surgery*. ACM Transactions on Graphics, Vol. 17, No. 2, April 1998, pages 84-115.

[27] Universitätsrechenzentrum der Technischen Universität Chemnitz. 2001. *Chemnitzer Linux Cluster (CLiC)*. Homepage. Stand 24.01.2003. `http://www.tu-chemnitz.de/urz/anwendungen/CLIC/`.

[28] Varshney, Amitabh. 1997. *A Hierarchy of Techniques for Simplifying Polygonal Models*. State University of New York at Stony Brook.

[29] Wenisch, M. / Langer, T. / Wagner, H. 1998. *CADaVR-Schnittstelle und Dateiformat*. Version 0.70 vom 25.08.1998. Technische Universität Chemnitz.

[30] Xia, Julie C. / Varshney, Amitabh. 1996. *Dynamic View-Dependent Simplification for Polygonal Models*. In: Proceedings of the IEEE Visualization 96.

Selbständigkeitserklärung

```
-----BEGIN PGP SIGNED MESSAGE-----


Hiermit erklaere ich, dass ich die vorliegende Arbeit
selbstaendig und nur mit den erlaubten Hilfsmitteln
angefertigt habe.



Tino Schwarze <tino.schwarze@informatik.tu-chemnitz.de>

Chemnitz, 5. Maerz 2003.

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.0.6 (GNU/Linux)
Comment: For info see http://www.gnupg.org

iQCVAwUBPmU0RKWjWF5o9P/NAQFYsQQA4B5lkdfkG6Y//7r83ezIpMy+m7CPiAZ8
vfT4bjs5FEv6JEEVT37QytrDe/0aawXN48D6VW6FGu3ywnv0fSvZekwApn26nNK4
MOQ2jI6EQYp0EjpqAasPhO8J8Khr6bESKCwA1Yu8psepfeZtkTyl97yYQcT7yw/7
fJ4u3ibFK6Q=
=V+QZ
-----END PGP SIGNATURE-----
```

Thesen

- Die komplexen Modelle in VR-Simulationen können bislang nicht mit zufriedenstellender Geschwindigkeit dargestellt werden. Darunter leidet der immersive Eindruck des Benutzers.
- VR-Simulationen enthalten viele komplexe Objekte. Die Betrachtungssituation erfordert nur relativ selten eine Darstellung aller Details dieser Objekte.
- Mit herkömmlichem, statischen Level-of-Detail werden die dargestellten Detailstufen nicht optimal an die Anzeigesituation angepaßt, da die Bildgeometrie nicht in die Auswahl der Detailstufe einfließt.
- Die bedarfsorientierte Generierung von Detailstufen erlaubt die optimale Anpassung des Detailgrades an die aktuelle Betrachtungssituation.
- Es existieren einige Verfahren zur Geometrievereinfachung, die für die dynamische Generierung der Detailstufen prinzipiell geeignet sind.
- Aufgrund des hohen Rechenaufwandes für die Geometriereduktion bietet sich eine Parallelisierung an.
- Der hohe Rechenaufwand legt außerdem eine Wiederverwendung bereits berechneter Reduktionen nahe.
- Je nach Reduktionsalgorithmus ist die Wiederverwendbarkeit einer Reduktion eingeschränkt. Die Kriterien „Abstand“, „Blickwinkel“ und „Beleuchtungsabhängigkeit“ stellen mögliche Einschränkungen dar.
- Die Wiederverwendung der Reduktionen durch Caching führt zu einer Geschwindigkeitssteigerung des Rendering. Das Verfahren ist mit Einschränkungen praktisch sinnvoll einsetzbar.