

Studienarbeit
Landschaftsvisualisierung mit Java 3D

Martin Barbisch

29.04.2002

Inhaltsverzeichnis

1	Einleitung	5
1.1	Worum geht es?	5
1.2	Motivation	5
1.3	Übersicht	5
1.4	Warum Java (3D)?	6
1.5	C-LOD's Vorteile gegenüber anderen Ansätzen	6
I	C-LOD mit Java 3D	9
2	Der C-LOD-Algorithmus	11
2.1	Vorbereitungsphase	11
2.2	Darstellungsphase	14
2.2.1	Triangulierung	14
2.2.2	Rendern	14
2.3	Geomorphing	15
2.4	Clipping	16
3	Kurzer Überblick über Java 3D	17
3.1	Szenengraph-Architektur	17
3.2	Die Terrain-Klasse - Ein Shape3D	17
3.2.1	Appearance-Objekt	19
3.2.2	Geometry-Objekt	19
3.3	Render Loop und Behaviors	19
3.4	Probleme bei der Umsetzung	19
3.4.1	Triangle Fans	19
3.4.2	GeometryArrays	20
II	Ergebnisse	21
4	Messergebnisse und Schlußfolgerungen	23
4.1	Messergebnisse	23
4.2	Schlußfolgerungen	24
4.3	Mögliche Verbesserungen	24
III	Kurzdokumentation des Quellcodes	27
5	Quellcode-Beschreibung	29
5.1	Das Applet	29
5.1.1	TeVi.java	29
5.2	Das Terrain	30

5.2.1	Terrain.java	30
5.3	Die Ablaufsteuerung - Die Behaviors	32
5.3.1	TEachFrameBehavior.java	32
5.3.2	TMouseBehavior.java	33
5.3.3	TKeyBehavior.java	33
IV	Einsatz des Programms	35
6	Voraussetzungen	37
6.1	Benötigte Software	37
6.2	Benötigte Hardware	38
7	Testflug - Steuerung und Konfiguration	39
7.1	Steuerung	39
7.1.1	Die Basics - Fliegen	39
7.1.2	Konfigurationstasten	40
7.2	Konfiguration - Die Parameter	40
7.2.1	Standardmodus	40
7.2.2	Region-of-Interest-Modus	41
8	Ins Internet stellen - Tutorial	43
8.1	Ein Applet erstellen	43
8.1.1	JAR-Datei erstellen	43
8.2	Daten vorbereiten	43
8.2.1	Benötigte Dateien	43
8.2.2	Höhenfeld	44
8.2.3	Textur	44
8.2.4	Auswahlkarte	44
8.2.5	Höhenfeld-Kacheln	44
8.3	Die HTML-Seite	45
8.3.1	Vorgegebenes Terrain	45
8.3.2	Mit wählbarer Region of Interest	45
8.3.3	HTML-Dateien konvertieren: HTMLConverter	46

Kapitel 1

Einleitung

1.1 Worum geht es?

Das Thema dieser Studienarbeit ist die Landschaftsvisualisierung (beziehungsweise Terrain-Visualisierung, daher der Name des Programms: TeVi). Das bedeutet hier die Darstellung der Landschaft als 3D-Grafik. Dazu wurde der C-LOD (Continuous Level of Detail) Algorithmus von Röttger et al. ("Real-Time Generation of Continuous Levels of Details for Height Fields", [1]) nach Java3D portiert.

Die Terrain-Engine ist in einem "Terrain-Knoten" gekapselt, so dass sie einfach in neue Java-3D Applikationen integriert werden kann. So kann die Funktionalität des Applets später modular an neue Anforderungen angepasst werden.

1.2 Motivation

Höhenfelder spielen eine wichtige Rolle auf dem Gebiet der Geographischen Informationssysteme (GIS). In einem Höhenfeld werden die Höhendaten einer Landschaft abgespeichert. Um einen solchen Höhen-Datensatz erforschen zu können, sollte er mit Frameraten dargestellt werden die eine interaktive Benutzung erlauben. Da solche Datensätze meist sehr komplex sind, stoßen auch schnelle Computer früh an ihre Leistungsgrenzen. Die Anzahl zu zeichnender Dreiecke muss also reduziert werden, wobei die Darstellungsqualität möglichst wenig sinken sollte.

Das Programm/Applet soll im Rahmen des *GeoVis* Projekts der Universität Erlangen-Nürnberg eingesetzt werden. Hierbei geht es um die Präsentation von Lehrinhalten für Geographie-Studenten im Internet, wobei das Wissen nicht nur elektronisch sondern auch interaktiv vermittelt werden soll (siehe [2]). Hierzu gehört unter anderem auch das Betrachten einer dreidimensionalen Landschaftsdarstellung.

1.3 Übersicht

Diese Ausarbeitung ist in vier Teile gegliedert. In Teil I wird erklärt wie Stefan Röttger's *C-LOD-Algorithmus* funktioniert. Ein Kapitel dieses Teils liefert außerdem eine kurze Einführung in Java 3D, und erwähnt Probleme die beim Verwenden aufgetaucht sind. Teil I richtet sich also an die Leser, die wissen wollen wie TeVi seine 3D-Grafik produziert.

Teil II beschreibt Vor- und Nachteile des Algorithmus, und zeigt Messergebnisse, die belegen, dass der C-LOD-Ansatz deutliche Vorteile gegenüber dem simplen Rendern des kompletten Datensatzes hat.

Teil III liefert einen groben Leitfaden für die Erforschung des Quellcodes, und ist somit hauptsächlich für diejenigen gedacht, die Änderungen am Programm vornehmen wollen.

Teil IV schließlich zeigt sowohl wie TeVi gesteuert und konfiguriert wird, als auch wie man damit 3D-Grafiken ins Internet stellt. Wer sich nur für diese praktischen Aspekte interessiert kann die anderen Teile ohne Bedenken überspringen.

1.4 Warum Java (3D)?

Um Plattformunabhängigkeit zu erreichen, und die Verwendung über das Internet zu ermöglichen, wurde das Programm in Java (Java 3D) implementiert. Beim Aufrufen der Internetseite wird das Java-Applet auf den lokalen Rechner heruntergeladen und dort ausgeführt. Der Anwender kann dann, je nach Einstellung in der HTML-Seite, auf einer Landkarte eine Region auswählen, woraufhin das Applet die entsprechenden Höhen- und Texturdaten vom Server nachlädt, oder eine vorgegebene Landschaft betrachten. Durch das gezielte Laden von Höhendaten soll das Terrain auch bei langsamer Internetanbindung interaktiv erkundet werden können.

Außerdem sollte ausprobiert werden, ob anspruchsvolle 3D-Grafik mit Java 3D möglich ist.

1.5 C-LOD's Vorteile gegenüber anderen Ansätzen

Die Algorithmen vor C-LOD fokussierten sich auf die globale Reduktion der Datenmenge oder Multiresolutions-Techniken, die die Auflösung auf der Oberflächenrauigkeit basierend reduzierten.

Die meisten existierenden Algorithmen bauen ein niedrig aufgelöstes Mesh auf ([4]), oder ein *Triangulated Irregular Network* (TIN) ([9]). Bei der Triangulierung eines Höhenfelds wird seine Rauigkeit in Betracht gezogen, flache Regionen können so mit wesentlich weniger Dreiecken trianguliert werden als unebene. Höhenfelder haben jedoch noch einige Eigenschaften, die man sich zu Nutze machen kann. So können Gebiete die weiter entfernt sind mit geringerer Detailstufe gezeichnet werden. Durch den sich bewegenden Augpunkt muss sich die Triangulierung ständig anpassen; der Begriff *Level of Detail* steht für alle Algorithmen, die dies anwenden.

Die *Progressive Meshes* von H. Hoppe ([5]) sind eine auf diesem Gebiet bekannte Technik. Die neuere Version ([6]) kann auch für Betrachter-abhängige Triangulierungen benutzt werden, benötigt jedoch große Datenstrukturen.

Einige Algorithmen unterteilen die Höhenfelder in mehrere Blöcke, die jeweils in unterschiedlichen Auflösungen vorliegen ([8] und [7]). Je nach Entfernung und Prozessor-Auslastung werden die Blöcke dann ausgewählt.

Allerdings können hier Löcher auftreten, wenn Blöcke unterschiedlicher Auflösung nebeneinander liegen. Ausserdem kommt es zu einem Popping-Effekt, wenn zwischen unterschiedlichen Auflösungen hin und her geschaltet wird. Das Popping kann durch Geomorphing zwischen zwei Detailstufen reduziert werden, ist aber nicht optimal, da die Betrachterentfernung für einen kompletten Block als konstant angenommen wird.

Ein Algorithmus speziell für Höhenfelder ist der von Lindstrom et al. ([3]). Er führte eine hierarchische Quadtree-Technik ein. Um den projizierten Pixelfehler zu reduzieren wird das Höhenfeld dynamisch, von der Betrachterentfernung abhängig, bottom-up trianguliert. Die Pixelfehler-Funktion muss für alle Punkte des Höhenfelds berechnet werden. Das ist sehr teuer, es sei denn man würde ein Fehlerintervall berechnen, das die Unterteilung und das Zusammenführen von vielen Vertices vermeiden würde. Wenn die Triangulierung verändert werden muss, dann werden alle betroffenen Knoten besucht. In diesem Fall müssen dann alle benachbarten Knoten bottom-up angepasst werden.

Der Algorithmus hatte jedoch noch einen Schwachpunkt, der sich derart auswirkt, dass bei sich bewegendem Blickpunkt ein Phänomen namens *Vertex Popping* auftritt. Nähert man sich einem Gebiet mit vielen Details, so tauchen diese ab einer bestimmten Entfernung

völlig unvermittelt auf. Um dieses Problem zu umgehen entwickelten Röttger et al. einen neuen, schnellen Geomorphing-Algorithmus, der top-down auf Quadrees operiert. Das Geomorphing ist nicht teuer und durch den top-down-Ansatz muss nur ein Bruchteil des Quadrees traversiert werden.

Teil I

C-LOD mit Java 3D

Kapitel 2

Der C-LOD-Algorithmus

Der *C-LOD-Algorithmus* verwendet als Datenstruktur einen Quadtree, wobei die Größe des Höhenfelds $2^n + 1 \times 2^n + 1$ sein muss. Abbildung 2.4 zeigt eine Beispieltriangulierung. Der Quadtree wird durch ein Array dargestellt, wobei Werte gleich MAX_FLOAT den weiteren Abstieg im Baum beenden (oder das Erreichen der tiefsten Stufe).

Da die Anzahl der von diesem Top-Down-Algorithmus zu besuchenden Knoten nur von der gewünschten Darstellungsqualität und nicht von der Größe des Höhenfelds abhängt, wird die Bildqualität nur von der Speicherbandbreite begrenzt.

2.1 Vorbereitungsphase

Das Höhenfeld wird am Anfang einmalig analysiert, um später das Höhenfeld ohne Risse, crackfrei, zeichnen zu können.

Es wird ein Subdivisionskriterium eingeführt anhand dessen entschieden werden kann, ob ein Knoten weiter unterteilt werden muss. Für dieses Kriterium müssen mehrere Aspekte in Betracht gezogen werden. Ein sehr wichtiger Punkt ist, dass die Landschaftsteile, die weit vom Benutzer entfernt sind, weniger detailliert gezeichnet werden sollten. Dies kann durch folgende Formel garantiert werden (siehe auch Abbildung 2.1):

$$\frac{l}{d} < C$$

Hierbei ist l die Entfernung zum Augpunkt und d die Kantenlänge eines Blocks. Die Konstante C ist ein Qualitätskriterium, das für die minimale globale Auflösung steht. Wird C erhöht, so wächst die Anzahl der zu zeichnenden Vertices quadratisch. Das Kriterium wird genau einmal pro Triangle Fan ausgewertet; die Entfernung wird daher der Effizienz halber per L^1 -Norm berechnet ($abs(x_1 - x_2) + abs(y_1 - y_2) + abs(z_1 - z_2)$).

Desweiteren sollen Gebiete mit hoher Oberflächenrauigkeit detaillierter dargestellt werden. Es soll im Endeffekt also der projizierte Pixelfehler minimiert werden, der ein gutes Maß für die Bildqualität darstellt. Wenn man in der Quadtree-Hierarchie um einen Level nach unten steigt, so wird an genau fünf Punkten ein neuer Fehler eingeführt: im Zentrum des Knotens und an den vier Mittelpunkten seiner Kanten. Eine obere Grenze für diesen Fehler im dreidimensionalen Raum erhält man indem man das Maximum der Absolutwerte der Höhenunterschiede dh_i nimmt (siehe Abbildung 2.2). Diese Höhenunterschiede werden sowohl entlang der Kanten des Knotens, als auch an den zwei Diagonalen gemessen, was insgesamt sechs Werte ergibt. Der Fehler, der sich durch Umschalten auf einen neuen Level ergibt, kann durch Vorberechnen eines Oberflächenrauigkeitswertes, der hier $d2$ genannt wird, bestimmt werden:

$$d2 = \frac{1}{d} \max |dh_i|, i = 1..6$$

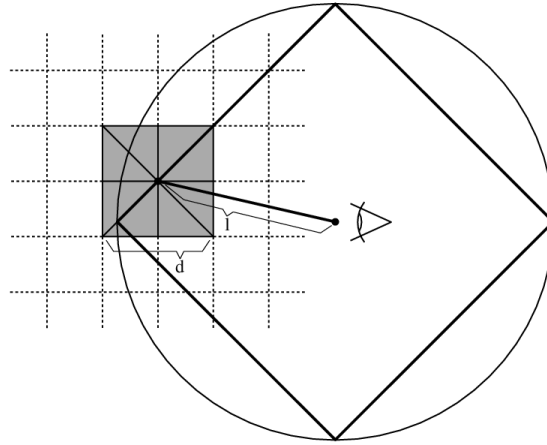


Abbildung 2.1: Globales Auflösungs-Kriterium, Entfernung im Verhältnis zur Grösse einer Quadtree-Zelle.

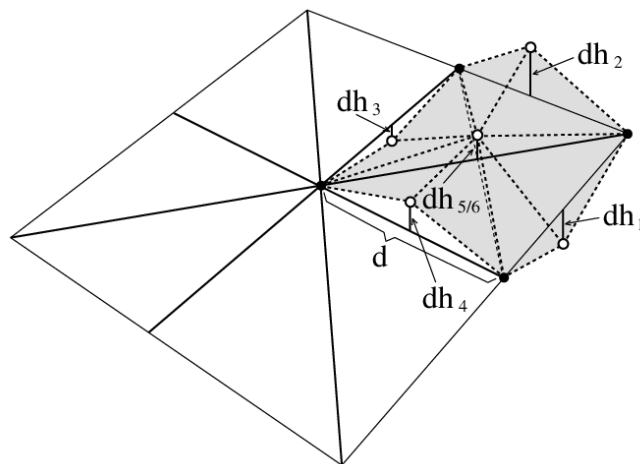
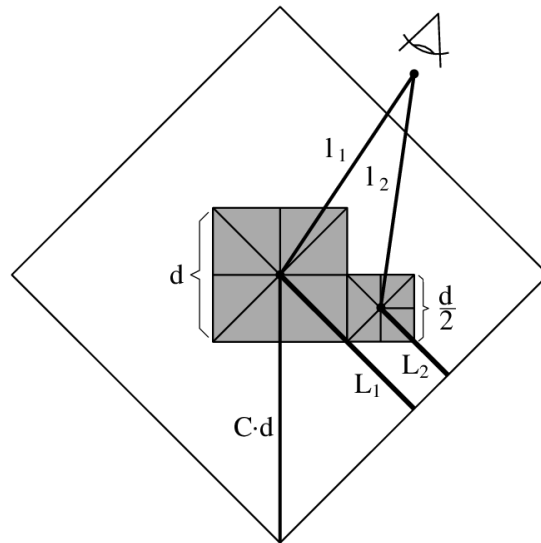


Abbildung 2.2: 6 Messpunkte für Oberflächenrauigkeit.

Abbildung 2.3: Beschränkungen der d_2 -Werte benachbarter Blöcke.

Die d_2 -Werte eines Knotens mal der Kantenlänge d des Knotens entsprechen dem Näherungsfehler im dreidimensionalen Raum. Daher ist der d_2 -Wert mal d eine Obergrenze für den Fehler, der beim Übergang zur nächsten Detailstufe eingeführt wird.

Das Subdivisionskriterium, das die d_2 -Werte berücksichtigt, kann jetzt wie folgt angegeben werden:

$$f = \frac{l}{d \cdot C \cdot \max(c \cdot d_2, 1)}$$

Falls $f < 1$, dann unterteile den aktuellen Knoten weiter. Die Konstante C steht wieder für die minimale globale Auflösung; die neue Variable c gibt die gewünschte Auflösung an. c beeinflusst direkt die Anzahl zu rendernder Polygone pro Frame. Man könnte c zum Beispiel an die aktuelle Auslastung der CPU anpassen, um so eine konstante Framerate zu erhalten.

Nun muss noch garantiert werden, dass benachbarte Blöcke sich maximal um eine Levelstufe unterscheiden. Das ist notwendig, da sich benachbarte Blöcke in der Oberflächenrauigkeit stark unterscheiden können, was Löcher im Gittermodell erzeugen könnte. Wie dieser maximale Unterschied garantiert werden kann wird nun beschrieben.

Angenommen f ist für einen Block kleiner 1, dann muss dieser Block unterteilt werden. In diesem Fall müssen aber alle benachbarten Blöcke, die die doppelte Kantenlänge haben, auch unterteilt werden. Die folgende Bedingung muss für einen benachbarten Block mit der Variable f_1 gelten um die Levelunterschiede zu begrenzen (f_2 steht dabei für den untersuchten Block):

$$f_1 < f_2 \Leftrightarrow \frac{l_1}{d \cdot d_{21}} < \frac{l_2}{\frac{d}{2} \cdot d_{22}}$$

Wenn der Augpunkt sich innerhalb des rechteckigen Gebiets, das in Abbildung 2.3 angedeutet ist, befindet, dann gilt dies immer, da $\frac{l_1}{d}$ immer kleiner ist als C . Ausserhalb dieses Gebiets wird der Wert des Bruches $\frac{l_1}{2l_2}$ begrenzt durch $\frac{1}{2}$ (Augpunkt ist unendlich weit entfernt) und der Konstante K :

$$\frac{1}{2} < \frac{l_1}{2l_2} < K \quad (C > 2)$$

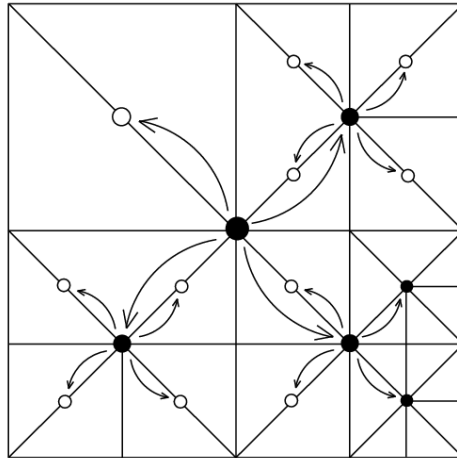


Abbildung 2.4: Beispiel für ein 9x9 Höhenfeld. Die Pfeile zeigen von Eltern zu Kindern.

$$K = \frac{L_1}{2L_2} = \frac{C}{2(C-1)}$$

Wenn also $\frac{d_2}{d_1}$ grösser ist als K , dann ist die Bedingung $f_1 < f_2$ wahr, da $\frac{L_1}{2L_2}$ zwischen $\frac{1}{2}$ und K liegt. Da die d_2 -Werte jedoch beliebig groß werden können, liegt $\frac{L_1}{2L_2}$ nicht unbedingt zwischen $\frac{1}{2}$ und K . Wenn $\frac{d_2}{d_1} \leq K$, dann müssen die d_2 -Werte folgendermaßen verändert werden: Bei der kleinsten Blockgröße beginnend werden die d_2 -Werte aller Blöcke berechnet und dann den Baum hochpropagiert. Der d_2 -Wert eines jeden Blocks ist das Maximum des lokalen Werts und K mal dem Wert der vorher berechneten Werte der Blöcke benachbarter Knoten des nächstniedrigeren Levels.

Das gilt nun für den 2D-Fall. Es kann aber für den 3D-Fall angepasst werden. Dazu wird die Höhe des Augpunktes über dem Boden als Annäherungswert für die dritte Entfernungs-Komponente herangezogen.

2.2 Darstellungsphase

Das Zeichnen der Landschaft geschieht pro Frame in zwei Schritten. Zuerst wird eine von der Kameraposition abhängige Triangulierung generiert, dann werden in einer zweiten Phase auf dieser Triangulierung basierend Triangle Fans generiert.

2.2.1 Triangulierung

Bevor eine Szene gerendert werden kann muss eine Triangulierung durch rekursiven Abstieg im Quadtrees aufgebaut werden. An jedem Unterknoten wird das in Abschnitt 2.1 erklärte Subdivisionskriterium ausgewertet und dessen Ergebnis gespeichert. Wenn das Subdivisionskriterium erfüllt wird und der feinste Detaillevel noch nicht erreicht wurde, dann wird in die vier Kindknoten hinabgestiegen.

2.2.2 Rendern

Das im vorigen Schritt triangulierte Höhenfeld wird durch rekursives Durchlaufen des Quadtrees gezeichnet. Immer dann wenn ein Blattknoten erreicht wird, wird ein vollständiger oder partieller Triangle Fan gezeichnet.

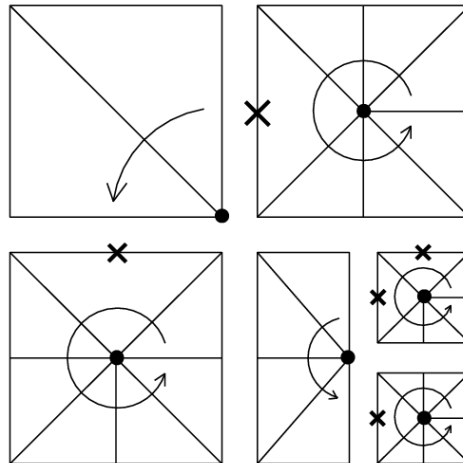


Abbildung 2.5: Rekursiv erstellter Dreiecksfächer. Die Kreuze markieren Stellen an denen kein Vertex gezeichnet werden darf.

Triangle Fans eignen sich sehr gut für Triangulierungen mit unterschiedlichen Auflösungen. Man muss nur darauf achten, dass man Mittelpunkte an Kanten zwischen Knoten unterschiedlichen Levels weglassen muss (siehe Abbildung 2.5). Damit das funktioniert dürfen sich die Levels benachbarter Knoten jedoch um maximal 1 unterscheiden. Um dies sicherzustellen werden, wie im Abschnitt über die Vorbereitungsphase besprochen, Informationen über die Oberflächenrauigkeit abgespeichert.

Während der Generierung der Triangle Fans muss bestimmt werden können, ob benachbarte Knoten bis zum gleichen Level unterteilt wurden, um dann bei ungleichem Level den Mittel-Vertex der gemeinsamen Kante wegzulassen. Dies kann durch Prüfen des Nachbar-knotens festgestellt werden, dessen Eintrag dann MAX_FLOAT sein muss. Das Zugreifen auf nicht gesetzte Matrixeinträge wird dadurch verhindert, dass die Levelunterschiede kleiner oder gleich 1 sein müssen.

2.3 Geomorphing

Um die Popping-Effekte, die auftreten wenn man sich einem Landschaftsdetail nähert, zu vermeiden, verwendet man Geomorphing um sanft von einem Detaillevel zum nächsten überzublen den. Die Subdivisionsvariable f aus dem Abschnitt über die Triangulierung kann in drei Bereiche unterteilt werden:

- f zwischen 0.5 und 1.0, der Knoten ist ein Blatt und wird nicht weiter unterteilt
- $f < 0.5$, der Knoten hat mindestens ein Kind
- $f > 1$, der Knoten wird nicht gezeichnet

Da Morphing nur in den Blattknoten vorkommt, kann die Formel $b = 2(1 - f)$ als Blendfaktor zwischen zwei Detailstufen verwendet werden, wobei b in den Bereich $[0,1]$ eingepasst werden muss. Abhängig davon, wie tief die benachbarten Quadtree-Knoten unterteilt sind, gibt es bis zu fünf Vertices an denen gemorpt werden muss (Nord, Ost, Süd, West, Zentrum). Die Höhe dieser Punkte wird linear mit dem Faktor b zwischen realer Höhe und der gemittelten Höhe der entsprechenden Eckpunkte interpoliert.

Da die Blendwerte benachbarter Knoten aufgrund unterschiedlicher Entfernung zum Betrachter leicht variieren könnten Risse (Cracks) in der Landschaft entstehen. Um das zu vermeiden wird das Minimum der Blendwerte der beiden benachbarten Knoten verwendet.

2.4 Clipping

Um die Anzahl der zu rendernden Polygone zu beschränken, wird häufig Clipping am *Viewing Frustum* verwendet. Da sowieso ein Quadtree eingesetzt wird, kann er auch zum Clipping benutzt werden. So lange die Hierarchiestufe des aktuellen Knotens im Baum nicht zu tief ist wird eine *Bounding Box* (beziehungsweise bei TeVi eine *BoundingSphere*) um ihn herumgelegt. Dann wird getestet, ob sie den Sichtkegel schneidet. Auf diese Weise können viele Vertices schon in einem sehr frühen Stadium einfach von der weiteren Verwendung ausgeschlossen werden.

Dieses Clipping kann sowohl in der Triangulierungs- als auch in der Rendering-Phase angewendet werden. In der Triangulierungsphase sollten die Bounding Boxes jedoch circa drei mal so groß sein wie in der Rendering-Phase, da die Blendwerte einiger Knoten zur Mesh-Generierung beitragen können, ohne selbst sichtbar zu sein.

Kapitel 3

Kurzer Überblick über Java 3D

Dieses Kapitel soll einen kurzen Überblick über Java 3D geben, der das API aber nur insoweit erklärt, wie es nötig ist, um TeVi darin einordnen zu können. Dabei werden einige Aspekte vereinfacht dargestellt. Ein ausführliches und brauchbares Tutorial ist *Getting Started with the Java 3D API* (siehe [10]); eine komplette Beschreibung der API findet sich in *The Java 3D API Specification* (siehe [11]). Beide Dokumente können, wie auch andere nützliche Unterlagen, von der Java 3D-Seite heruntergeladen werden (unter *Collateral*).

Der letzte Abschnitt geht auf Probleme bei der Umsetzung mit Java 3D ein.

3.1 Szenengraph-Architektur

Java 3D ist eine objektorientierte Szenengrapharchitektur. Alle Objekte, die in den Szenengraph eingefügt werden können haben die Klasse *SceneGraphObject* als Vater (siehe Abbildung 3.1). *Node* und seine Unterklassen sind die Klassen die direkt in den Baum eingefügt werden. *NodeComponents* sind die Bestandteile der *Leafs*. So enthält ein *Shape3D*, von dem die *Terrain*-Klasse von TeVi abgeleitet ist, ein *Geometry*-Objekt und ein *Appearance*-Objekt. Die von *Group* abgeleiteten Klassen wie *BranchGroup* und *TransformGroup* sind die inneren Knoten des Baumes.

Abbildung 3.2 zeigt den prinzipiellen Aufbau eines Java 3D Szenengraphen. Jedes Java 3D Programm besitzt als Vater des Szenengraphen ein *VirtualUniverse*-Objekt (oder eine davon abgeleitete Klasse: *SimpleUniverse*). Darunter hängt dann mindestens ein *Locale*-Objekt (und meist auch genau eines), welches den Ursprungspunkt eines Koordinatensystems bildet.

Auf der rechten Seite befindet sich der Teilast, der verschiedene Details des Betrachterstandpunktes beinhaltet. Soll zum Beispiel der Betrachterstandpunkt geändert werden, so wird einfach die Transformationsmatrix des *TransformGroup*-Knotens, der über dem *ViewPlatform*-Objekt hängt, geändert.

Die linke Seite beinhaltet den Geometrieteil des Szenengraphen. Angenommen der *Shape3D*-Knoten wäre ein Stuhl, so kann dieser mit Hilfe des darüberliegenden *TransformGroup*-Knotens auf dieselbe Art verschoben werden, wie vorher der Betrachterstandpunkt.

Auf den *Behavior*-Knoten in Abbildung 3.2 geht Abschnitt 3.3 ein.

3.2 Die Terrain-Klasse - Ein Shape3D

Die *Terrain*-Klasse von TeVi ist von *Shape3D* abgeleitet, und kann somit als ein Blatt in den Java 3D Szenengraphen eingehängt werden. Die beiden wichtigsten Bestandteile sind ein *Appearance*-Objekt und ein *Geometry*-Objekt.

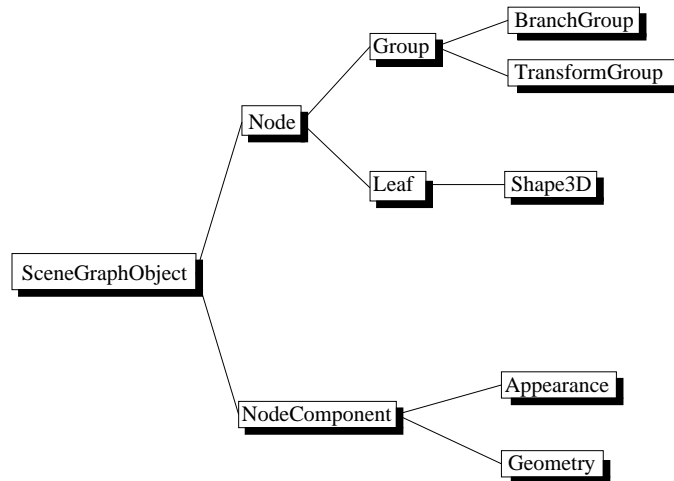


Abbildung 3.1: Ein Auszug der Klassen, die in einen Szenengraph eingefügt werden können.

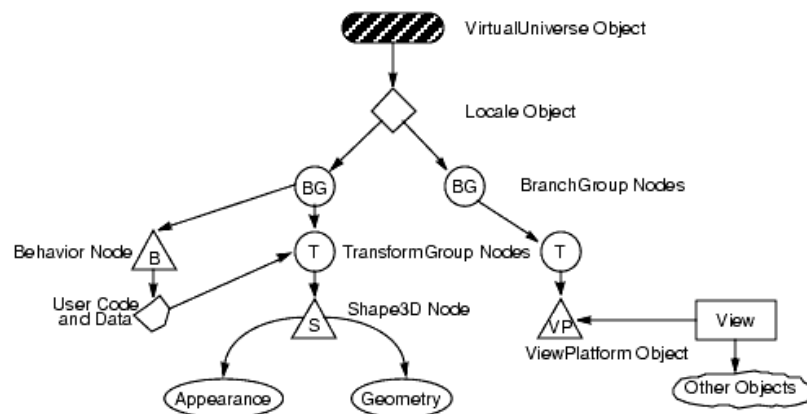


Abbildung 3.2: Ein typischer Java 3D Szenengraph.

3.2.1 Appearance-Objekt

Das *Appearance*-Objekt beinhaltet eine Textur, sowie Information darüber wie diese gezeichnet werden soll. Man kann hier einstellen, ob Backface-Culling verwendet werden soll, oder ob die Polygone gefüllt oder als Drahtgittermodell angezeigt werden sollen. Hier legt man also all die Parameter fest, die entscheiden, wie die Geometrie “erscheinen” soll. Die Geometrie-Informationen selbst werden im Geometrie-Objekt festgelegt.

3.2.2 Geometry-Objekt

TeVl verwendet eine von *Geometry* abgeleitete Klasse, nämlich *TriangleStripArray* (warum nicht die auch vorhandene Klasse *TriangleFanArray* benutzt werden konnte steht in Abschnitt 3.4 auf Seite 19). Da die Vertices ständig verändert werden müssen wird eine Geometrie *by reference* verwendet. Dazu wird Speicher für ein Float-Array reserviert, in dem die Vertex- und Textur-Koordinaten *interleaved* abgespeichert werden (also: $s_1, t_1, x_1, y_1, z_1, s_2, t_2, \dots$). In diesem Array könnten auch noch Farbwerte stehen, die TeVl aber nicht benötigt, da es eine Textur verwendet; sowie Normalen-Koordinaten, die auch nicht verwendet werden, da keine Beleuchtung benutzt wird.

Um nun die Geometriedaten verändern zu können, implementiert die Klasse *Terrain* das *GeometryUpdater*-Interface. Dies dient dazu, die Daten atomar ändern zu können. So ein Änderungsaufwurf wird durch einen *Behavior* initiiert.

3.3 Render Loop und Behaviors

Behaviors sind Objekte, die auch in den Szenengraph eingehängt werden. Wenn die Traversierung bei ihnen ankommt, dann können sie Code ausführen. Dies tun sie aber nur wenn sie von einer *WakeupCondition* aktiviert wurden. So werden die beiden TeVl-Klassen *TMouseBehavior* und *TKeyBehavior* von Maus- beziehungsweise Tastatur-Ereignissen geweckt. *TEachFrameBehavior* wird einmal pro Frame geweckt und ist dafür verantwortlich, das *Terrain*-Objekt, falls nötig, zu einem Geometrie-Update zu veranlassen.

Nachfolgend wird, vereinfacht, die Java 3D Render Loop dargestellt:

```
while(true) {
    Process input
    If (request to exit) break
    Perform Behaviors, Traverse scene graph, render visible objects
}
Cleanup and exit
```

3.4 Probleme bei der Umsetzung

3.4.1 Triangle Fans

Die OpenGL-Implementierung des C-LOD-Algorithmus kann *Triangle Fans* zum Zeichnen der Knoten verwenden.

Java 1.2.1 bietet zwar ein *TriangleFanArray* an, allerdings kann hier keine geringere Anzahl an Vertices verwendet werden, als bei der Erzeugung angegeben wurde. Was bei Verwendung eines dynamischen Datenvolumens natürlich schlecht ist. Und bei jedem Frame ein neues *TriangleFanArray* anzulegen, verbietet sich durch den damit entstehenden Overhead von selbst.

Es ist von *Sun* geplant dieses Manko in Java 3D 1.3 zu entfernen, welches zur Zeit (April 2002) aber nur als Beta-Version für Windows erhältlich ist. Dann würde ein *TriangleFanArray* in TeVl Sinn machen; womit sehr viel weniger Vertices in ein Array ge-

geschrieben und an die Grafikkarte geschickt werden müssten. Statt maximal 24 würden dann nur noch maximal 9 pro Fan benötigt, was den Speicherdurchsatz verbessern würde.

3.4.2 GeometryArrays

Die beim Erzeugen eines *GeometryArrays* (hier: *TriangleStripArray*) angegebene Größe des Arrays, welches die Vertices *by reference* zur Verfügung stellt, kann nachträglich nicht erhöht werden. Um nun nicht den Speicher für den *worst case* zur Verfügung stellen zu müssen (der normalerweise bei weitem nicht benötigt wird), wird mit einem vernünftigen Startwert begonnen, der bei Bedarf verdoppelt wird. Dabei muss ein neues *TriangleStripArray* angelegt werden, was sich durch ein kurzes Ruckeln bemerkbar machen kann. Dies geschieht, wenn überhaupt, im Normalfall aber nur ein bis zwei Mal pro Programmablauf.

Teil II

Ergebnisse

Kapitel 4

Messergebnisse und Schlußfolgerungen

4.1 Messergebnisse

Die Messungen wurden auf einem AMD Athlon Rechner mit 900 MHz und einer GeForce 1 (mit 32 MB DDR RAM) durchgeführt. Die minimale globale Auflösung wurde auf den Wert 12 gesetzt.

Es wurden pro Datensatz zwei verschiedene Messungen durchgeführt: einmal wurde das C-LOD-Verfahren verwendet, und einmal wurden einfach alle Dreiecke gerendert. Das Programm wurde mit der Taste 'p' auf eine Umlaufbahn gebracht, die einmal durchflogen wurde.

In der ersten Spalte der Tabelle steht die Größe des verwendeten Datensatzes, in der 2. Spalte steht die durchschnittliche Framerate bei aktiviertem C-LOD, in der 3. Spalte steht die durchschnittliche Anzahl der gerenderten Dreiecke. In der 4. Spalte steht die Framerate, wenn alle Dreiecke gezeichnet werden, und in der 5. Spalte die entsprechende Anzahl.

Auflösung	FPS (C-LOD)	# Dreiecke	FPS (pur)	# Dreiecke
17x17	1000	128	1000	512
33x33	800	460	500	2048
65x65	300	1200	160	8192
129x129	150	1650	43	32768
257x257	90	1750	11	131072
513x513	75	1850	2	524288
1025x1025	65	1950	0	2097152

Die Anzahl der Dreiecke ist bei Verwendung des C-LOD-Verfahrens bei den größeren Höhenfeldern relativ konstant, was ja auch der Sinn der Sache ist. Hier sieht man schön wie der Verwaltungsaufwand (hauptsächlich das Clipping) die Framerate beeinflusst. Die Anzahl eingesparter Dreiecke ist enorm, wobei die Unterschiede in der Darstellungsqualität mit bloßem Auge kaum sichtbar sind. Es macht eben wenig Sinn zig Dreiecke in einem Pixel zu zeichnen.

Die Kurven in Abbildung 4.1 stellen die Framerate dar. Auf der x-Achse ist die Auflösung des Höhenfelds aufgetragen, auf der y-Achse die Framerate. Die dicke Linie steht für den C-LOD-Algorithmus, und die dünnere, gestrichelte für das "Zeichne-Alles-Verfahren".

Auch wenn die Kurven ähnlich verlaufen, so ist doch (auch anhand der Tabelle) zu erkennen, dass das Brute-Force-Verfahren sehr schnell einbricht, während sich der C-LOD-Algorithmus der x-Achse langsam annähert.

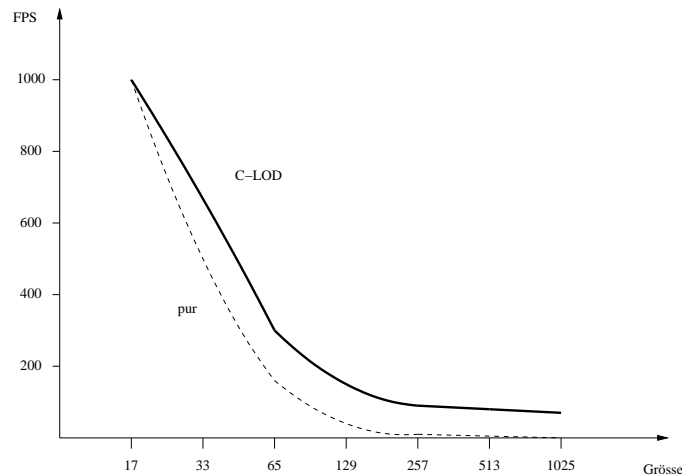


Abbildung 4.1: Vergleich der Frameraten zwischen C-LOD und reinem Dreiecke-Zeichnen.

Java 3D kann also durchaus für anspruchsvolle 3D-Grafik verwendet werden. Es wird aber wohl nie die Geschwindigkeit von OpenGL oder Direct3D (auf denen es aufsetzt) erreichen, da es ja noch eine Szenengraphenarchitektur daraufsetzt. Und der Funktionsumfang muss immer eine Schnittmenge dessen sein, was die darunterliegenden Grafikbibliotheken bieten.

4.2 Schlußfolgerungen

Der größte Vorteil des C-LOD-Verfahrens ist, dass die Geschwindigkeit nicht von der Größe des Höhenfelds abhängt, sondern nur von der gewünschten Darstellungsqualität. Und selbst bei geringer Detailstufe ist die Grafik durch das Geomorphing, welches unangenehme Poppingeffekte vermeidet, stets ansehnlich.

Landschaftsdetails die nahe an der Kamera sind werden detailliert dargestellt, und solche die weiter davon weg sind weniger detailliert; was auf Grund der Entfernung auch Sinn macht, weil man in der Ferne meist sowieso nicht mehr Details erkennen könnte.

Der Aufwand, der durch die zusätzlichen Berechnungen entsteht, wird durch den Nutzen bei weitem wieder wett gemacht.

Bei sehr kleinen Höhenfeldern ist der C-LOD-Algorithmus wegen seines “Verwaltungsaufwandes”, der natürlich einen Overhead erzeugt, minimal langsamer als wenn einfach alle möglichen Dreiecke gerendert werden.

Overhead wird unter anderem durch die folgenden Aufgaben erzeugt:

- durch die Triangulierung
- Berechnung von *Bounding Boxes* (beziehungsweise *BoundingSpheres*)
- Berechnung des *Viewing Frustums*

Aber mit diesem winzigen Nachteil kann man, angesichts der Vorteile sehr gut leben.

4.3 Mögliche Verbesserungen

In Abschnitt 3.4 auf Seite 19 wird besprochen warum an Stelle der Klasse *TriangleFanArray* die Klasse *TriangleStripArray* für die Speicherung der Vertices verwendet wird. Die Laufzeit des Programms wird aber nur insofern beeinträchtigt, dass mehr Vertex-Daten in

das Array geschrieben werden müssen. Es werden jedoch gleich viele Vertices berechnet. Die Geschwindigkeit hängt daher mehr vom Speicherdurchsatz ab. Wenn eine Grafikkarte nur wenige Dreiecke pro Sekunde zeichnen kann, so sinkt natürlich die Framerate.

TeVi speichert Vertices und Texturkoordinaten ab, um Rechenzeit zu sparen. Wenn man einen niedrigeren Speicherverbrauch möchte kann man sich durch Berechnungen einen Teil dieses verwendeten Speicherplatzes sparen. Ein typischer Zeit versus Platz Tradeoff.

Teil III

**Kurzdokumentation des
Quellcodes**

Kapitel 5

Quellcode-Beschreibung

Dieses Kapitel ist für diejenigen gedacht, die den TeVi-Quellcode verändern wollen. Es ist keine komplette Dokumentation des Quellcodes, sondern geht nur auf die wichtigsten Dinge ein, und ist als Orientierungshilfe gedacht. Da im Programmcode sprechende Namen für Variablen und einige Kommentare benutzt wurden, und Java selbst sehr ausführliche Klassen- und Methodennamen verwendet, sollte man sich mit Hilfe dieses Leitfadens leicht zurechtfinden.

TeVi besteht aus drei Teilen:

- Dem Applet (das auch als Applikation ausgeführt werden kann), welches das Programm initialisiert.
- Dem Terrain-Darstellungs-Knoten (abgeleitet von *Shape3D*).
- Und den *Behaviors*, die die Eingaben von Maus und Tastatur verarbeiten, und dafür sorgen, dass das Terrain dementsprechend dargestellt wird.

Diese Teile werden in den folgenden Abschnitten beschrieben.

5.1 Das Applet

Das in 'TeVi.java' definierte Applet übernimmt die administrativen Aufgaben, wie zum Beispiel die Initialisierung von Java 3D und des Terrainknotens, sowie den Start der drei Behaviors.

5.1.1 TeVi.java

Applet-Methoden

Als von *Applet* abgeleitete Klasse überschreibt 'TeVi.java' einige der dort deklarierten Funktionen:

- *public void init()* Einstiegspunkt in das Programm. Wird aufgerufen, wenn das Applet erstmals vom Browser geladen wird. Alle Initialisierungen finden hier statt; die Verwendung eines Konstruktors macht hier keinen Sinn.
- *public String getAppletInfo()* Wird vom Browser aufgerufen um an Informationen über das Applet zu gelangen.
- *public String[][] getParameterInfo()* Wird aufgerufen um an Informationen über die Parameter zu gelangen, auf die das Applet reagiert.
- *public void destroy()* Wird aufgerufen, wenn das Applet aus dem Browser entfernt wird. Ruft *SimpleU.removeAllLocales()*; auf, um den Java3D-Thread zu beenden.

Alternativer Einstiegspunkt

Für den Aufruf als Applikation (siehe Kapitel 7) wird noch die *main()*-Methode definiert:

- *public static void main(String[] args)* Verwendet die Klasse *MainFrame* um ein Applet zu simulieren.

Andere Methoden

- *private void selectROI()* Stellt die Auswahlkarte dar und startet den ROI-Modus.
- *public void selectROI2(int selX, int selY)* Wird von *TMouseBehavior* aufgerufen und lädt daraufhin die benötigten Daten.
- *private String createFileName(int x, int y)* Erstellt die Dateinamen für die Höhenfeld-Kacheln.
- *private void loadHeightData(String hFile)* Lädt eine Höhenfeld-Datei.
- *private void grabPixels(Image img, int x, int y, int w, int h, int[] pix, int off, int scansize)* Kopiert den Inhalt eines *Image* in ein Array.
- *private void initTerrain()* Lädt im Standardmodus Höhendaten und Textur.
- *private int checkWidth(int w)* Prüft, ob *w* der Formel $2^n + 1$ entspricht, falls nicht dann wird der nächstniedrigere passende Wert zurückgeliefert.
- *private BranchGroup createSceneGraph(SimpleUniverse su)* Hier wird der Knoten im Szenengraph eingerichtet, unter dem der Terrain-Knoten hängt.

5.2 Das Terrain

Die Klasse *Terrain* ist von *Shape3D* abgeleitet, und beinhaltet die *Geometry* und die *Appearance*. 'Terrain.java' ist in mehrere Abschnitte unterteilt; einer davon kümmert sich um die einmalige Analyse der Daten, einer um die Triangulierung, und einer um das Rendern der Daten. Zum genaueren Verständnis des Algorithmus wird auf Kapitel 2 auf Seite 11 verwiesen.

5.2.1 Terrain.java

Konstruktor

- *public Terrain(SimpleUniverse su, int[] hMap, Texture2D tex2D, float cx, float cz, float vertSpacing, float maxHeight)* Initialisierung.

Vertex-Manipulation

Funktionen um Vertices in Arrays zu schreiben, oder sie daraus auszulesen.

- *private void setVertex(float[] array, int vnum, float x, float y, float z)*
- *private void setVertex2(float[] array, int vnum, float s, float t)*
- *private void getVertex(int vnum, Point3d point)*

Analyse

Berechnung der *ErrorMatrix*.

- *private void calcD2ErrorMatrix()* Ruft die anderen Funktionen dieses Abschnitts auf.
- *private void calcD2ErrorMatrixRec(int centerX, int centerZ, int width, int level)* Baut die Fehlermatrix rekursiv auf.
- *private float calcD2Value(int centerX, int centerZ, int width)* Berechnet den Fehlerwert eines Knotens.
- *private void propagateD2Errors()* Propagiert die Fehler nach oben.

Triangulierung

Berechnung der *QuadMatrix*.

- *private void triangulateMeshRec(int c, int width, int level)* Rekursives Aufbauen der Triangulation.
- *private float calcSubDiv(int nodeIndex, int width)* Berechnet den Wert, der über die Unterteilung eines Knotens entscheidet.
- *private float calcBlend(float subDiv)* Berechnet den Blendwert zu einem SubDiv-Wert.
- *private void deleteNode(int index, int width)* Löscht einen Knoten und die vier Kinder.
- *private float getHeightAboveGround()* Ermittelt die Höhe des Betrachters über dem Terrain. Die Betrachterposition wird der Variable *EyePos* entnommen.

Rendering

Füllen des Geometrie-Arrays mit Vertex-Positionen und Texturkoordinaten.

- *private boolean renderMeshRec(int centerX, int centerZ, int width, int level, int dirToFather)* Traversieren der *QuadMatrix*.
- *private void createFanAround(int x, int z, int width, boolean[] corners, boolean isLeaf, int dirToFather)* "Zeichnen" eines Fans, das heißt Ermittlung der beteiligten Vertices.
- *private float getHeight(int index, int width, boolean isLeaf, int dirToFather, int neswc, float blend1)* Die eventuell geblendete Höhe eines Vertex zurückliefern.
- *private void setInterleaved(int index, float height)* Vertex-Position und Texturkoordinaten in das 'interleaved' Geometrie-Array schreiben.

Administratives zum Erzeugen/Updaten der Geometrie

- *private void calcViewFrustum()* Berechnung des *Viewing Frustums*.
- *private void createGeometry()* Triangulierung und Rendering anstoßen.

Ändern des Erscheinungsbildes (Appearance)

- *private Appearance createAppearance(boolean filled)* Erscheinungsbild festlegen (Textur, Backface culling, Wire frame mode).
- *public void setFilledPolys(boolean filled)* Textur oder Drahtgitter anzeigen.
- *public void toggleGeoMorphing()* Geomorphing an-/ ausschalten.
- *public void moreDetail(boolean updateNow)* Detailstufe erhöhen.
- *public void lessDetail(boolean updateNow)* Detailstufe verringern.

Geometrie-Update von außen anstoßen

- *public void transformChanged(int type, Transform3D transform)* Die Klasse *Terrain* erweitert das Interface *MouseBehaviorCallback*, um von *MouseBehaviors* über Veränderungen unterrichtet werden zu können.
- *public void updateTerrain()* Wird von außen (hier: *TEachFrameBehavior*) aufgerufen, um das *Terrain* upzudaten.
- *public void updateData(Geometry geometry)* Die Klasse *Terrain* erweitert das Interface *GeometryUpdater*, um die Updates atomar zum 'richtigen' Zeitpunkt durchzuführen. Diese Funktion darf nicht direkt aufgerufen werden.

5.3 Die Ablaufsteuerung - Die Behaviors

TEachFrameBehavior, *TMouseBehavior* und *TKeyBehavior* sind alle drei von der Klasse *Behavior* abgeleitet.

5.3.1 TEachFrameBehavior.java

TEachFrameBehavior wird einmal pro Frame aufgerufen, was durch verwenden des *WakeUpCriterion WakeupOnElapsedFrames* erreicht wird. Hier werden die Eingaben, die *TMouseBehavior* und *TKeyBehavior* entgegengenommen haben, gemanagt, und der *Terrain*-Knoten aufgefordert entsprechend zu reagieren.

Konstruktor

- *TEachFrameBehavior(Terrain t, TransformGroup tg, TKeyBehavior kb, TMouseBehavior mb)* Initialisiert einige Variablen mit den Argumenten und richtet einen Anfangsblickpunkt ein.

Behavior-Methoden

Als von *Behavior* abgeleitete Klasse überschreibt *TEachFrameBehavior* einige der dort deklarierten Funktionen:

- *public void initialize()* Wird von außen zwecks Initialisierung aufgerufen. Einstellung, dass die Klasse einmal pro Frame aufgerufen werden soll.
- *public void processStimulus(Enumeration criteria)* Wird aufgerufen, wenn das registrierte Ereignis (neuer Frame) eintritt. Ermittelt anhand der Abfrage von *TMouseBehavior* und *TKeyBehavior* neuen View-Point und teilt dies dem *Terrain*-Knoten mit.

5.3.2 TMouseBehavior.java

TMouseBehavior behandelt Mauseingaben und kennt zwei Modi: ROISelect und Navigation.

Konstruktoren

Es gibt für unterschiedlichen Bedarf die folgenden vier Konstruktoren:

1. *public TMouseBehavior()*
2. *public TMouseBehavior(TransformGroup tg, float factor)*
3. *public TMouseBehavior(TransformGroup tg, boolean inv)*
4. *public TMouseBehavior(TransformGroup tg, float factor, boolean inv)*

Behavior-Methoden

Als von *Behavior* abgeleitete Klasse überschreibt *TMouseBehavior* einige der dort deklarierten Funktionen:

- *public void initialize()* Wird von außen zwecks Initialisierung aufgerufen. Einstellung, dass die Klasse von *WakeupOnAWTEvent* über Mausereignisse informiert werden soll.
- *public void processStimulus(Enumeration criteria)* Wird aufgerufen, wenn ein Mausereignis eintritt. Ruft *doProcess(MouseEvent evt)* auf um neue Blickrichtung zu berechnen.

Andere Methoden

- *private void init(TransformGroup tg, float factor, boolean inv)* Von den Konstruktoren aufgerufene Initialisierungsroutine.
- *public void setSelectROIMode(TeVi t)* Gebietsauswahl-Modus setzen.
- *public void setup(TransformGroup tg, float factor, boolean inv)* Von außen aufrufbare Funktion zum nachträglichen Setzen von Parametern.
- *private void doProcess(MouseEvent evt)* Berechnet neue Blickrichtung, oder, falls im ROI-Modus, Aufruf von Methode *selectROI2()* des Applets, um Daten zu laden.
- *public void getDir(Vector3f v3f)* Schreibt aktuelle Blickrichtung in *v3f*.
- *public void forceT3DRead()* Erzwingt Neueinlesen der Transformationsmatrix des Betrachterstandpunkts (falls dieser von außerhalb geändert wurde).
- *public void toggleInvert()* Vertauscht die y-Achse der Maus (hoch/ runter).

5.3.3 TKeyBehavior.java

TKeyBehavior behandelt Tastatureingaben.

Konstruktoren

- *public TKeyBehavior(Terrain terrain)*
- *public TKeyBehavior(Terrain terrain, float speedInc)*

Behavior-Methoden

Als von *Behavior* abgeleitete Klasse überschreibt auch *TKeyBehavior* einige der dort deklarierten Funktionen:

- *public void initialize()* Wird von außen zwecks Initialisierung aufgerufen. Einstellung, dass die Klasse von *WakeupOnAWTEvent* über Tastatureignisse informiert werden soll.
- *public void processStimulus(Enumeration criteria)* Wird aufgerufen, wenn ein Tastaturereignis eintritt. Ruft *processKeyEvent(KeyEvent event)* auf um den Tastendruck zu verarbeiten.

Andere Methoden

- *private void processKeyEvent(KeyEvent event)* Verarbeitet Tastatureingaben.
- *public boolean isResetDemanded()* Wurde 'r' gedrückt?
- *public boolean isInvertDemanded()* Wurde 'i' gedrückt?
- *public boolean isPilot()* Autopilot an?
- *public float getSpeed()* Geschwindigkeit abfragen.
- *public float getStrafe()* Seitwärtsgeschwindigkeit abfragen.

Teil IV

Einsatz des Programms

Kapitel 6

Voraussetzungen

Dieses Kapitel erläutert welche Soft- und Hardware für den Einsatz von TeVi auf Ihrem Rechner vorhanden sein muss.

6.1 Benötigte Software

Java

Da TeVi ein Java-Applet (sowie eine Java-Applikation) ist, benötigt es zum Betrieb ein installiertes Java. Ob dabei das *Java Development Kit (JDK)* oder die *Java Runtime Environment (JRE)* verwendet wird ist unerheblich. Wer das Programm nur ausführen, nicht aber verändern will kann sich auf das kleinere *JRE* beschränken. Es muss mindestens die Java-Version 1.3.1 sein. Herunterladen kann man sich die selbstentpackenden Installationsdateien von <http://java.sun.com> ([12]) für alle gängigen Betriebssysteme. Unter Windows reicht es im Normalfall, einfach das Installationsprogramm zu starten, die Dateien werden dann in sinnvolle Standardverzeichnisse geschrieben. Unter Linux sollte man vorher einen Blick in die README-Datei werfen; aber auch hier ist die Installation sehr einfach.

Java 3D

Java 3D ab Version 1.2.1 bekommt man auch auf der Sun-Homepage (unter [13]). Auch hier hat man wieder die Wahl zwischen einer größeren Version für Entwickler und einer kleineren für reine Anwender. Allerdings werden hier noch nicht alle Systeme bedient; so sind zur Zeit (April 2002) zum Beispiel Versionen für Windows und Linux erhältlich, Mac OS X bleibt aber noch außen vor.

Für Windows muss man sich außerdem noch für die OpenGL- oder die Direct3D-Variante entscheiden. Wobei die OpenGL-Version mehr Grafikfunktionen kennt und angeblich schneller sein soll. Das kann sich aber bei Java 3D 1.3 schon wieder ändern (wo DirectX 8.0 unterstützt werden soll). Für TeVi sollte es aber keinen Unterschied machen.

Browser

Alle aktuellen Versionen von *Internet Explorer* und *Netscape Communicator* sind in der Lage Java2-Plugins zu verwenden. Bei anderen Browsern muss das nicht unbedingt der Fall sein. Falls Java2-Plugins unterstützt werden, sollte dem Betrachten von TeVi's 3D-Welt jedoch nichts im Wege stehen.

6.2 Benötigte Hardware

Prozessor

Jeder Prozessor auf dem Java2 ausführbar ist. Da die Detailstufe angepasst werden kann, sollten auch ältere Computer-Modelle das Programm ausführen können. Generell gilt aber wie immer in der 3D-Grafik: je schneller desto schöner.

Grafikkarte

Da Java 3D auf OpenGL aufsetzt (unter Windows auch auf Direct3D), sollte die Grafikkarte diese Grafik-Bibliotheken unterstützen. Es werden keine aussergewöhnlichen Grafik-Befehle verwendet, daher ist nur die reine Füllrate interessant. Aber auch hier gilt wie schon oben beschrieben, dass ein Herunterschalten der Detailstufe auch langsameren Karten eine flüssige Darstellung ermöglichen sollte.

Hauptspeicher

Der Speicherbedarf des Programms variiert je nach Größe des verwendeten Höhenfelds und der Größe der Textur. Für ein Höhenfeld der Größe 129x129 werden beispielsweise 700 - 1000 kB benötigt; für ein 257x257er Feld 2,3 - 2,5 MB (jeweils ohne Texturen). Ein heutzutage üblicher PC hat normalerweise nicht weniger als 64 MB RAM, was für Höhenfelder ausreicht, die der entsprechende Prozessor sinnvoll (das heisst mit ansprechender Detailstufe) darstellen kann.

Kapitel 7

Testflug - Steuerung und Konfiguration

Dieses Kapitel beschreibt kurz die Flugsteuerung, besondere Tasten sowie die Parameter, die an TeVi übergeben werden können wenn es lokal als Applikation und nicht als Applet ausgeführt wird.

7.1 Steuerung

TeVi wird mit einer Maus-Tastatur-Kombination gesteuert. Die Steuerung ähnelt der vieler First Person Spiele mit zwei kleinen Ausnahmen:

- So erhöht zum Beispiel jeder Druck auf die Vorwärtstaste die Geschwindigkeit; um anzuhalten muss nun entweder entsprechend oft, beziehungsweise entsprechend lange, die Rückwärtstaste gedrückt werden. Alternativ dazu gibt es aber auch eine spezielle Stoptaste. Ungewöhnlich, aber sehr angenehm um sich die Landschaft anzuschauen.
- Zum Umsehen wird die Maus benutzt, wobei jedoch eine Maustaste gedrückt sein muss (Neudeutsch: draggen). Das hat den Vorteil, dass die Maus beim Umsehen auch aus dem Fenster bewegt werden kann.

7.1.1 Die Basics - Fliegen

Taste	Beschreibung
w	jeder Druck erhöht die Geschwindigkeit
s	Geschwindigkeit verringern, bzw. rückwärts fliegen
a	nach links ausweichen
d	nach rechts ausweichen
space	stoppt jegliche Bewegung
r	zurück zum Ausgangspunkt
i	invertiert Maus hoch/runter
p	Autopilot, auf Umlaufbahn mit Blick zum Mittelpunkt
Maus klicken+ziehen	umsehen

Mit der Taste 'p' wird der Autopilot eingeschaltet. Das bewirkt, dass sich die Kamera auf einer runden Umlaufbahn um den Mittelpunkt bewegt. Mit den Tasten 'a' und 'd' kann die Umlaufrichtung und Geschwindigkeit beeinflusst werden.

7.1.2 Konfigurationstasten

Taste	Beschreibung
1	Texturen an
2	Texturen aus (Drahtgittermodell)
3	Darstellungsqualität verringern
4	Darstellungsqualität erhöhen
g	Geomorphing an/aus

Falls die Grafik zu stark ruckelt, so kann die Darstellungsqualität mit der Taste '3' herabgesetzt werden; wenn die voreingestellte Detailstufe nicht ausreicht kann sie entsprechend mit '4' erhöhen. Der Drahtgitter-Modus ist hauptsächlich für Demonstrationszwecke gedacht.

7.2 Konfiguration - Die Parameter

TeVi kennt zwei Betriebsmodi. Im *Standardmodus* wird sofort nach Programmstart ein vorgegebenes Höhenfeld gerendert. Und im *Region-Of-Interest-Modus* wird zuerst eine Übersichtskarte angezeigt, auf der man sich dann per Mausclick ein Gebiet auswählen kann; TeVi lädt daraufhin nur die benötigten Daten (eventuell von einem Server, siehe auch Kapitel 8 auf Seite 43).

Wenn man das Programm, verschiedene Höhenfelder und Texturen austesten will, dann kann man das Applet auch als parametergesteuerte Applikation von der Kommandozeile aus aufrufen.

7.2.1 Standardmodus

Wenn TeVi ohne Parameter, also wie folgt aufgerufen wird:

```
java TeVi
```

dann verwendet es als Höhenfeld die Bilddatei 'Default_HeightMap.png' und als Textur die Datei 'Default_TexMap.png'. Diese müssen sich im selben Verzeichnis befinden wie die TeVi-Programmdateien (also die .class-Dateien, oder eine .jar-Datei wie zum Beispiel 'TeVi.jar'; eine Anleitung, welche Dateien wo hin müssen, und wie man eine .jar-Datei erstellt findet sich in Kapitel 8).

Diese Default-Dateien können natürlich durch eigene überschrieben werden. Man kann aber auch als Parameter angeben, welche Dateien man ansehen möchte:

```
java TeVi [heightfield.png [texture.png]]
```

Gibt man nur die Höhenfeld-Datei an, zum Beispiel:

```
java TeVi Hawaii_HeightMap_257.png
```

so wird als Textur die Datei 'Default_TexMap.png' geladen. Um beides gezielt zu laden, weil zum Beispiel die Textur nicht zum Höhenfeld passt, oder man eine Textur mit einer anderen Auflösung verwenden möchte, dann gibt man als zweiten Parameter den Namen der Textur-Datei an:

```
java TeVi Hawaii_HeightMap_129.png Hawaii_TexMap_1024.png
```

Es ist also nicht möglich nur eine andere Textur zu wählen.

7.2.2 Region-of-Interest-Modus

Um den ROI-Modus auszuprobieren, müssen drei oder vier Parameter angegeben werden:

```
java TeVi regionmap.jpg numTilesX numTilesY [width]
```

'regionmap.jpg' kann eine GIF-, JPG- oder PNG-Grafikdatei sein, deren Höhe und Breite eine Potenz von 2 sein muss. 'numTilesX' und 'numTilesY' stehen für die Anzahl der Kacheln in horizontaler, beziehungsweise vertikaler Richtung. Für die mitgelieferten Test-Kacheln würde der Aufruf beispielsweise so aussehen:

```
java TeVi WorldMap.jpg 6 3
```

Der vierte Parameter definiert die Ausdehnung des Höhenfeldes sowohl in x- als auch in y-Richtung, und muss $2^n + 1$ sein (3, 5, 9, 17,...; ansonsten wird einfach der nächste passende Wert, der kleiner als die angegebene Zahl ist, genommen).

```
java TeVi WorldMap.jpg 6 3 129
```

Nach einem Klick auf die Karte lädt das Programm die entsprechenden Daten und geht in den Standardmodus über.

Kapitel 8

Ins Internet stellen - Tutorial

8.1 Ein Applet erstellen

Dieser Abschnitt erklärt wie man die Programmdateien kompiliert und wie man daraus eine JAR-Datei erstellt, die für den Aufruf über das Internet verwendet wird. Wurden die Dateien nicht verändert, so können die mitgelieferten Class-Dateien, beziehungsweise die Datei 'TeVi.jar' benutzt und dieser Abschnitt übersprungen werden.

8.1.1 JAR-Datei erstellen

Falls die Programmdateien verändert wurden, müssen sie zuerst neu kompiliert werden. Das geschieht am einfachsten mit:

```
javac TeVi.java
```

Die anderen (in Kapitel 5 auf Seite 29 beschriebenen) Java-Dateien werden dann bei Bedarf automatisch mitübersetzt; selbst wenn 'TeVi.java' selbst gar nicht verändert wurde.

Nun hat man die entsprechenden fünf Class-Dateien, die mit dem Programm *jar* gepackt werden können:

```
jar cvf TeVi.jar *.class
```

Wie beim bekannten Unix-Programm *tar* stehen die Parameter für 'c'ompress, 'v'erböse (welches natürlich weggelassen werden kann) und 'f'ile. Die Datei wird automatisch komprimiert.

8.2 Daten vorbereiten

In diesem Abschnitt wird beschrieben in welchem Format die Dateien vorliegen müssen. Der benötigte Inhalt in einer HTML-Seite wird in Abschnitt 8.3 beschrieben.

8.2.1 Benötigte Dateien

Es empfiehlt sich, der Übersichtlichkeit halber, für jedes Projekt ein eigenes Verzeichnis anzulegen. Nennen wir es hier zum Beispiel 'TeVi_Hawaii'.

Für die Webseite wird von den in Abschnitt 8.1 erwähnten Dateien nur 'TeVi.jar' benötigt. Für den Standardmodus muss eine Höhenfeld-Bitmap (siehe Abschnitt 8.2.2) und eine Textur-Bitmap (Abschnitt 8.2.3) in unser Beispielverzeichnis 'TeVi-Hawaii' kopiert werden. Für den ROI-Modus werden eine Auswahlkarte (Abschnitt 8.2.4), sowie mehrere Höhenfeld-Kacheln (Abschnitt 8.2.5) benötigt.

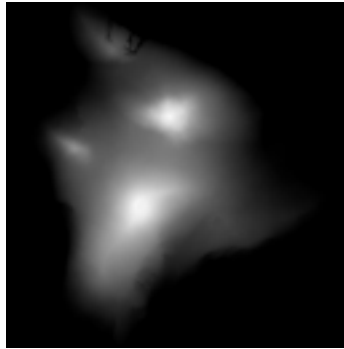


Abbildung 8.1: Ein Höhenfeld als Graustufen-Bitmap.

Die Bitmaps müssen in einem Format vorliegen, das von Java eingelesen werden kann. Zur Zeit sind dies die Formate GIF, PNG und JPG.

Nun fehlt nur noch eine HTML-Datei, wie zum Beispiel 'TeVi_Hawaii.html' oder 'TeVi_ROI.html', die auf das Applet verweist. Wenn jetzt noch von einer anderen Webseite auf diese HTML-Seite verlinkt wird, dann ist die 3D-Darstellung der Landschaft im Internet verfügbar.

8.2.2 Höhenfeld

Das Höhenfeld ist eine 8Bit-Graustufen-Bitmap, die zwei Bedingungen erfüllen muss:

- Die Bitmap muss quadratisch sein, die Größe in x-Richtung muss also gleich der Größe in y-Richtung sein.
- Dieser Größenwert muss der Formel $2^n + 1$ für $n \geq 1$ genügen. Also 3, 5, 9, 17, 33, ...

Abbildung 8.1 zeigt ein solches Höhenfeld.

8.2.3 Textur

Das Textur ist eine Bitmap, die eine für Texturen übliche Bedingung erfüllen muss: Die Größe in x-Richtung und die Größe in y-Richtung muss eine Potenz von 2 sein (also 2^n für $n \geq 1$). Normalerweise müssen Texturen nicht quadratisch sein, da aber bei einem quadratischen Höhenfeld eigentlich nur eine quadratische Textur Sinn macht wird auch eine solche verlangt.

Eine zum Höhenfeld aus Abbildung 8.1 passende Textur ist in Abbildung 8.2 dargestellt.

8.2.4 Auswahlkarte

Für den ROI-Modus wird eine Auswahlkarte benötigt, die beim Programmstart angezeigt wird. Im Beispieldatensatz wäre dies 'WorldMap.jpg'.

8.2.5 Höhenfeld-Kacheln

Damit die vom Server herunterzuladende Datenmenge nicht zu groß wird müssen die Höhendaten gekachelt vorliegen. Sie müssen wie bei einem einzelnen Höhenfeld als quadratische 8Bit-Graustufen-Dateien vorliegen und zwei Bedingungen erfüllen:

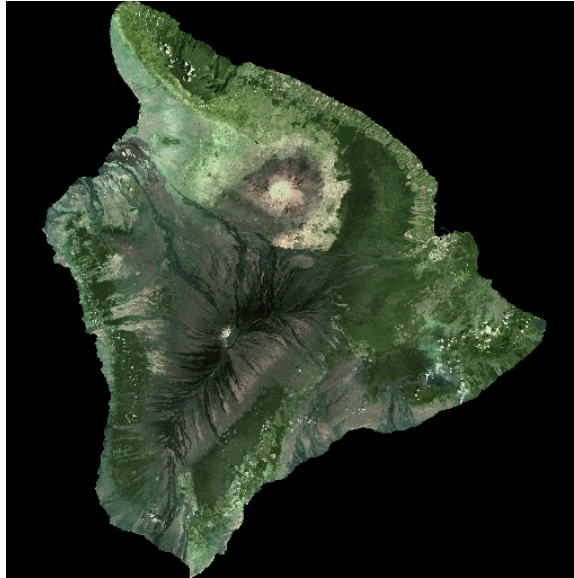


Abbildung 8.2: Eine Textur, die auf das Drahtgittermodell gelegt wird.

- Alle Kacheln müssen die selbe Größe haben (die Dimensionen betreffend, nicht die Dateigröße).
- Sie müssen einem speziellen Namensschema folgen: 'HeightMap_' + dreistellige Nummer in x-Richtung + '_' + dreistellige Nummer in y-Richtung + '.png'. Also zum Beispiel 'HeightMap_005_002.png'.

8.3 Die HTML-Seite

Die beiden folgenden Abschnitte zeigen die notwendigen Applet-Codeabschnitte, die in eine HTML-Datei eingefügt werden müssen, um TeVi auf dieser Seite zu verwenden. Die Parameter müssen natürlich entsprechend angepasst werden.

Darüber könnte man zum Beispiel eine Überschrift setzen, und darunter eine kurze Beschreibung der Steuerung.

8.3.1 Vorgegebenes Terrain

```
<APPLET archive = "TeVi.jar" code = "TeVi.class"
        width = 512 height = 512>

<PARAM name = "HeightField" value = "Default_HeightMap.png">
<PARAM name = "Texture"      value = "Default_TexMap.png">

</APPLET>
```

Die Parameter werden in Abschnitt 7.2.1 auf Seite 40 erklärt.

8.3.2 Mit wählbarer Region of Interest

```
<APPLET archive = "TeVi.jar" code = "TeVi.class"
        width = 512 height = 512>
```

```
<PARAM name = "SelectionMap" value = "WorldMap.jpg">
<PARAM name = "NumTilesX" value = "6">
<PARAM name = "NumTilesY" value = "3">
<PARAM name = "ROIWidth" value = "129">

</APPLET>
```

Die Parameter werden in Abschnitt 7.2.2 auf Seite 41 erläutert.

8.3.3 HTML-Dateien konvertieren: HTMLConverter

Die HTML-Dateien müssen jetzt noch mit dem Programm *HTMLConverter* umgewandelt werden (siehe [14]), damit der aufrufende Internet Browser das Java 2 Plugin verwendet und nicht das eingebaute Java 1. Unter Linux erfolgt der Aufruf mit 'HTMLConverter.sh', und unter Windows mit 'HC.bat' beziehungsweise 'HTMLConverter.bat'.

Nun muss man das Verzeichnis auswählen in dem sich die HTML-Dateien befinden, ein Template auswählen (bevorzugt: Extended (Standard + All Browsers/ Platforms)), und auf 'Convert...' klicken.

HTMLConverter legt im übergeordneten Verzeichnis ein Unterverzeichnis (im aktuellen Beispiel namens 'TeVi-Hawaii_BAK') an, in dem die Originalversion(en) der geänderten HTML-Dateien abgespeichert werden.

Bei Unklarheiten einfach einen Blick in die gute 'readme.txt' oder 'Readme.html' werfen.

Index

- Ablaufsteuerung, 32
- Applet, 29, 43
- Applikation, 30, 39
- Auswahlkarte, 44

- Behaviors, 19
- Browser, 37

- C-LOD, 5, 6
- C-LOD-Algorithmus, 5, 11
- Clipping, 16

- d2-Wert, 13
- Darstellung, 14
- Darstellungsphase, 14

- Geomorphing, 6, 15
- GeoVis, 5
- GIS, 5
- Grafikkarte, 38

- Höhenfeld, 44
- Höhenfeld-Kacheln, 44
- Höhenfelder, 5
- Hauptspeicher, 38
- HTML-Seite, 45
- HTMLConverter, 46

- Java, 37
- Java 3D, 6, 17, 37

- Landschaftsvisualisierung, 5

- minimale globale Auflösung, 11

- Pixelfehler, 11
- Popping-Effekt, 6
- Progressive Meshes, 6
- Prozessor, 38

- Quadtree, 6, 11

- Region-Of-Interest-Modus, 40
- Render Loop, 19

- Shape3D, 17
- Standardmodus, 40

- Steuerung, 39
- Szenengraph-Architektur, 17

- TEachFrameBehavior.java, 32
- Terrain, 30
- Terrain-Visualisierung, 5
- Terrain.java, 30
- TeVi, 5
- TeVi.java, 29
- Textur, 44
- TKeyBehavior.java, 33
- TMouseBehavior.java, 33
- Top-Down-Ansatz, 7
- Triangle Fans, 14
- Triangulated Irregular Network, 6
- Triangulierung, 14

- Viewing Frustum, 16
- Vorbereitungsphase, 11

Literaturverzeichnis

- [1] Stefan Röttger, W. Heidrich, P. Slusallek, H.P. Seidel. *Real-Time Generation of Continuous Levels of Detail for Height Fields*
- [2] *GeoVis*, <http://www.geographie.uni-erlangen.de/geovis>
- [3] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, G. Turner. Real-time, continuous level of detail rendering of height fields. In *Computer Graphics (Proceedings Siggraph '96)*, pages 109-118. 1996.
- [4] M.H. Gross, R. Gatti, O. Staadt. Fast multiresolution surface meshing. In G. Nielson and D. Silver, editors, *Proceedings Visualization '95*, pages 135-142. IEEE Computer Society Press, 1995.
- [5] H. Hoppe. Progressive meshes. In *Computer Graphics (Proceedings of Siggraph '96)*, pages 99-108, 1996.
- [6] H. Hoppe. View-dependent refinement of progressive meshes. In *Computer Graphics (Proceedings of Siggraph '97)*, pages 189-198, 1997.
- [7] D. Koller, P. Lindstrom, W. Ribarsky, L.F. Hodges, N. Faust, G. Turner. Virtual GIS: A real-time 3D geographic information system. In G. Nielson and D. Silver, editors, *Proceedings Visualization '95*, pages 94-100. IEEE Computer Society Press, 1996.
- [8] M. Suter and D. Nüesch. Automated generation of visual simulation databases using remote sensing and GIS. In G. Nielson and D. Silver, editors, *Proceedings Visualization '95*, pages 135-142. IEEE Computer Society Press, 1995.
- [9] D.C. Taylor and W.A. Barrett. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proceedings of Graphics Interface '94*, pages 33-42, 1994.
- [10] Dennis J. Bouvier, *Getting Started with the Java 3D API*, <http://java.sun.com/products/java-media/3D/collateral/>
- [11] *The Java 3D API Specification*, <http://java.sun.com/products/java-media/3D/collateral/>
- [12] *Java*, <http://java.sun.com>
- [13] *Java 3D*, <http://java.sun.com/products/java-media/3D/>
- [14] *HTMLConverter*, <http://java.sun.com>