Brandenburgische Technische Universität Cottbus
Institut für Informatik
Lehrstuhl Praktische Informatik / Grafische Systeme

**Bachelorarbeit**




# Generierung von Baumdarstellungen in VRML für den Branitzer Park

Stephan Rogge

*Datum der Abgabe*
14.12.2007

1. Betreuer: Prof. Dr. Winfried Kurth
2. Betreuer: Dipl.-Inf. Reinhard Hemmerling

# Aufgabenstellung

Im Rahmen des Projekts „Virtueller Branitzer Park“ soll untersucht werden, mit welchen Mitteln Baumdarstellungen realisiert werden können. Dabei besteht die Möglichkeit, auf vorhandenes Bildmaterial zurückzugreifen und dieses für ein 3D-Modell des Parks aufzubereiten und/oder Bäume virtuell zu erzeugen. Für die Generierung virtueller Bäume soll das Programm GroIMP (entwickelt am Lehrstuhl Grafische Systeme) benutzt werden, um zu testen, in wie weit grammatikbasierte Modellspezifikationen für eine Baumdarstellung geeignet sind. Ein weiterer Schwerpunkt der Arbeit ist die Darstellung der erzeugten Bäume in VRML. Die Arbeit beinhaltet insbesondere:

- die Erstellung eines Katalogs von billboardbasierten Einzelbaumdarstellungen in VRML für ausgewählte Baumarten des Branitzer Parks,
- die Prüfung der Integration von durch Grammatiken erzeugten 3D-Baummodellen anhand des Beispiels eines speziellen Solitärbaumes,
- den Vergleich mit der konventionellen 3D-Modellierung von Bäumen mit kommerzieller Modeller-Software,
- die Vorbereitung der Integration der erzeugten Modelle in virtuelle Parkansichten aus einer parallel laufenden Bachelorarbeit.

Zum Abschluss der Arbeit gehören, neben den VRML-Dateien, eine internetfähige, aussagekräftige Präsentation, die die Ergebnisse erläutert, eine detaillierte Dokumentation aller ggf. neu implementierten Schnittstellentools, XL-Codes und Skripte, sowie ein knapper Reflexionsbericht über den Projektverlauf.

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst habe. Weiterhin versichere ich, dass ich keine anderen als die angegebenen Quellen benutzt beziehungsweise die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Cottbus, den 14. Dezember 2007

_____

Stephan Rogge

# Danksagung

Diese Bachelorarbeit entstand am Lehrstuhl Praktische Informatik / Grafische Systeme der Brandenburgischen Technischen Universität in Cottbus. Bei der Umsetzung wurde ich von einigen Personen unterstützt, denen ich hiermit meinen Dank aussprechen möchte:

- Prof. Dr. Winfried Kurth, für die interessante Aufgabenstellung und die Unterstützung. Die Verbesserungsvorschläge und Hinweise halfen mir beim Erstellen dieser Arbeit.
- Dipl. Inf. Reinhard Hemmerling, konnte mit seinem Wissen rund um das Programm GroIMP und dem Gebiet der Computergrafik so manche Frage beantworten.
- Dipl. Phys. Ole Kniemeyer, für die detaillierten Antworten, wenn es zu GroIMP Fragen gab.

Weitere Unterstützung erhielt ich von meinen Eltern, meiner Freundin Dajana Strecker, Ilona Stecker, Michael Dahley, Steven Strecker, Benji Strecker sowie Ralf-Peter Stahr vom Systemhaus Stahr und Dr. Erwin Roth. Auch diesen Menschen möchte ich recht herzlich danken.

# Inhaltsverzeichnis

**Inhaltsverzeichnis** **1**

**1 Einleitung** **3**
1.1 Motivation . . . 3
1.2 Gliederung . . . 4
1.3 Begrifflichkeiten . . . 4

**2 Der Branitzer Park als virtuelles Modell** **7**
2.1 Einführung zum Projekt „Virtueller Branitzer Park“ . . . 7
2.1.1 Allgemeines zum Park . . . 7
2.1.2 Der virtuelle Park . . . 8
2.2 Baumauswahl – Grundlage zur Veranschaulichung . . . 9
2.2.1 Die Erle . . . 9
2.2.2 Die Eiche . . . 10
2.2.3 Die Buche . . . 11

**3 Billboards in VRML** **12**
3.1 Prinzipien, Einsatzgebiete . . . 12
3.1.1 Was sind Billboards? . . . 12
3.2 Fahrplan zur Umsetzung mit Bildvorlagen . . . 16
3.2.1 Anforderungen an das Bildmaterial . . . 16
3.2.2 Vorhandenes Bildmaterial . . . 17
3.2.3 Arbeitsschritte . . . 17
3.2.4 Ergebnis . . . 23
3.3 Möglichkeiten in VRML . . . 27
3.3.1 Dynamische Billboards . . . 27
3.3.2 Rechteck - Fächer . . . 31
3.3.3 Raster . . . 34
3.4 Stochastische Billboards . . . 38

**4 Erstellung mit Hilfe eines 3D-Editors** **40**
4.1 Prinzip einer Modellierung an einem kleinen Beispiel . . . 40
4.1.1 Modellierung mit NURBS . . . 41
4.1.2 Modellierung . . . 41
4.1.3 Ergebnis . . . 44
4.2 Alternative Modellierung . . . 45

**5 Erstellung mit Hilfe von Grammatiken** **49**
5.1 Einführung Lindenmayer-Systeme, relationale Wachstumsgrammatiken . . 49
5.2 XL und GroIMP . . . . . 54
5.2.1 XL: Implementierung der RGG . . . . . 54
5.2.2 GroIMP . . . . . 61
5.3 Baumerstellung in GroIMP am Beispiel der Erle . . . . . 65
5.3.1 Herangehensweise (vom Feinen ins Grobe) . . . . . 65
5.3.2 Erstellung eines einzelnen Triebs . . . . . 67
5.3.3 Generierung des gesamten Baumes . . . . . 81
5.3.4 Rendern mit externem Renderer Sunflow . . . . . 89
5.3.5 Export nach VRML . . . . . 92
5.3.6 Billboard-Generierung für VRML . . . . . 93
5.4 Möglichkeiten für den Einsatz von GroIMP . . . . . 110

**6 Auswertung** **113**
6.1 Vergleich und Bewertung der Billboard-Systeme . . . . . 113
6.1.1 Kriterien . . . . . 113
6.1.2 Bewertung der drei Billboard-Systeme . . . . . 114
6.1.3 Fazit . . . . . 117
6.2 Vergleich der drei Methoden . . . . . 118
6.2.1 Kriterien . . . . . 118
6.2.2 Bewertung der drei Methoden . . . . . 119
6.2.3 Fazit . . . . . 125
6.3 Ausblick . . . . . 127

**Literaturverzeichnis** **129**

# 1 Einleitung

Einer der Schwerpunkte bei der Erstellung des virtuellen Branitzer Parks ist die Erzeugung und Darstellung von Pflanzen, beziehungsweise Bäumen im Speziellen. In der Natur sind Bäume sehr komplexe Objekte. Sie darzustellen bedeutet einen großen Aufwand.
Die Art der Erzeugung eines Baummodells ist nur ein Teil des Schwerpunktes, denn im Rahmen des Projektes muss die Repräsentation den Anforderungen einer internetfähigen Darstellung genügen. Somit stellt sich auch das Problem, wie diese Baummodelle dargestllt werden sollen.
Diese beiden Probleme, das Erzeugen sowie das Darstellen, werden in dieser Arbeit besprochen.

## 1.1 Motivation

Bei der Umsetzung einer kompletten Parkanlage ist das Problem der Darstellung der Vegetation einer der kritischen Punkte. Da die virtuelle Vegetation das Aussehen des Park-Modells mitprägt, ist abzuwägen, wie detailliert diese zu erstellen ist. Speziell für das Projekt „Virtueller Branitzer Park“ wird die Vegetation auf das Problem der Baumdarstellung reduziert.
Der Branitzer Park besitzt circa 8000 Bäume. Es stellt sich die Frage, auf welche Weise die hohe Anzahl von Bäumen im virtuellen Modell repräsentiert werden sollte? Denn die Erzeugung der einzelnen Modelle, ob nun als Bild oder als 3D-Modell, ist unterschiedlich aufwändig und muss deshalb miteinander abgewogen werden.

Eine einfache Variante der Erzeugung nutzt Fotovorlagen, welche nach einer Bearbeitung bei der Repräsentation zum Einsatz kommen. Da in der Regel das Bildmaterial begrenzt ist, sind auch die Möglichkeiten der Repräsentation eingegrenzt.
Andere Varianten basieren auf dem Erzeugen von 3D-Modellen, die mehr Möglichkeiten bei der Repräsentation bieten. Ein Weg der Modellierung ist die Nutzung von konventionellen 3D-Modellierungsanwendungen. Durch einen manuellen Prozess entsteht ein 3D-Modell des betroffenen Baumes. Der Aufwand hängt davon ab, wie komplex der Baum ist und wie detailliert das Ergebnis sein soll. Andere Wege basieren auf der Nutzung von sogenannten Grammatiken, die 3D-Modelle unter Verwendung von Regeln erzeugen.
Neben der Erzeugung der einzelnen Baummodelle ist es ebenso wichtig, wie diese dargestellt werden. Da die Darstellung von 8000 detaillierten 3D-Modellen in einem internetfähigen Park-Modell kaum möglich ist, müssen die Baummodelle vereinfacht werden. Die Qualität der Repräsentation hängt dabei von den zugrundliegenden Prinzipien der Vereinfachung ab.

## 1.2 Gliederung

Zunächst wird in Kapitel 2 das vorhandene Datenmaterial des Projektes „Virtueller Branitzer Park“ gesichtet. Besonders wichtig ist das Bildmaterial, denn im Laufe dieser Arbeit werden einige Verfahren an ausgewählten Bäumen demonstriert. Die Auswahl der genutzten Bäume und das zur Verfügung stehende Bildmaterial wird an dieser Stelle näher beleuchtet.
Wie bereits in der Motivation erwähnt, können Bäume durch präparierte Bildvorlagen dargestellt werden. In Kapitel 3 wird, neben der Vorbereitung des Bildmaterials anhand von drei Beispielen, auch das Problem der geeigneten Repräsentation näher erläutert und mögliche Lösungen in Form von Systemen formuliert. Diese werden später aufgegriffen und darauf hin untersucht, ob sie für einen Einsatz im Projekt geeignet sind.
Als Alternative zur Darstellung mit bildbasierten Baumvorlagen wird in Kapitel 4 die Erzeugung eines 3D-Modells mit einem konventionellen 3D-Modellierungsprogramm näher betrachtet und einzelne Arbeitsabläufe erläutert.
Unter der Verwendung von Grammatiken wird in Kapitel 5 ebenfalls ein 3D-Modell erzeugt. Dazu kommt das Programm GroIMP zum Einsatz. Es wird am Lehrstuhl Grafische Systeme entwickelt und bietet die Möglichkeit, 3D-Modelle auf Regeln basierend zu erzeugen. Die in Kapitel 3 eingeführten Systeme werden dabei auf das erzeugte 3D-Modell angewendet.
Das letzte Kapitel vergleicht die drei Systeme der Repräsentation sowie die drei Methoden zur Erzeugung einer Darstellung und bewertet diese. Dabei werden auch jeweils mögliche Einsatzgebiete beziehungsweise Verbesserungen besprochen.

## 1.3 Begrifflichkeiten

Im Zusammenhang mit dieser Arbeit werden einige Begriffe verwendet, die an dieser Stelle erst einmal näher erläutert werden.

### Topologie

Topologie (oder auch topologische Struktur) beschäftigt sich mit Eigenschaften von Objekten, die bei geometrischen Verzerrungen invariant bleiben. Das Beispiel Kaffeetasse und Donut veranschaulicht die Bedeutung für Objekte: Durch Strecken, Ziehen, Stauchen oder Drehen kann man aus der Kaffeetasse einen Donut formen, ohne dabei die Struktur des Ausgangsobjektes zerschneiden oder zerstören zu müssen. Dieser Prozess ist auch umkehrbar.

### Morphologie

Beschreibt die äußere Form und Gestalt (griech. *morphé* = Gestalt), insbesondere von Lebewesen. Ein Teilgebiet der Morphologie ist dabei die Beziehung zwischen Pflanzengestalt und Umwelt [4, Seite 9].

### Knospe

Beinhaltet teilungsaktives Gewebe (sog. apikale Meristeme). Meristeme sind verantwortlich für das primäre Triebwachstum und Verzweigungen. Dabei kann eine Knospe drei verschiedene Schicksale ereilen, die die Morphologie des Baumes direkt beeinflussen [12, Seite 41]:

- Bildung eines Triebes
- Übergang in den Ruhezustand - und eine erneute Aktivierung
- Absterben

### Internodium

Ein Internodium ist ein Längensegement, das zwei Blattansätze (Knoten = Nodien) verbindet. Es ist wichtig für den Prozess des Längenwachstums. Ein apikales Meristem produziert innerhalb eines „Wachstumsschubes“ mehrere solcher Internodien. Wachstumsschübe können dabei mehrfach in einem Jahr auftreten. Zusammen mit den dazugehörigen Blättern und den neu gebildeten axillären Knospen, spricht man dann von einer Wachstumseinheit [12, Seite 43].

### Ordnung

Hilfsmittel um festzustellen, wie viele Verzweigungen auf einem gedachten Weg von einem Ast bis zur Wurzel des Baumes durchschritten werden müssen. Dabei gilt, je häufiger eine Verzweigung durchschritten wird, desto höher ist die Ordnung des betroffenen Astes. Der Ast, der direkt aus der Wurzel stammt, hat dabei die Ordnung Null.

### Vitalität

Ist eine Art Parameter der Knospe und beeinflusst deren Schicksal mit. Je größer die Ordnung, desto geringer die Vitalität, und somit besteht auch eine geringere Wahrscheinlichkeit, dass die Knospe eine Verzweigung oder einen neuen Trieb erzeugt.

### Graph

Ist ein Konstrukt, bestehend aus einer Menge von Eckpunkten, sogenannten **Knoten**, die in Verbindung mit anderen Knoten stehen. Diese Verbindungen heißen **Kanten** [23, Seite 1].

### Szene-Graph

Ist die zugrundeliegende Datenstruktur von virtuellen Szenen. Der Szene-Graph ist ein gerichteter azyklischer Graph. Objekte und deren Eigenschaften, die einer virtuellen Szene angehören, werden als Knoten im Szene-Graphen repräsentiert. Die Kanten zwischen den Knoten repräsentieren die Beziehungen der Objekte einer Szene untereinander [2, Seite 991]. Somit entsteht eine hierarchische Struktur.

### VRML

Ist die Kurzform für „Virtual Reality Modeling Language“. Diese Sprache dient dazu, 3D-Szenarien zu beschreiben. Sie nutzt ebenfalls Szene-Graphen als grundlegende Datenstruktur. Die Knoten des Szene-Graphen sind in Knoten-Typen eingeteilt. In der Spezifikation der Sprache sind bereits einige Knoten-Typen vorgegeben. Es können jedoch auch selbst solche Typen definiert werden.
Wird in dieser Arbeit in Zusammenhang mit VRML zum Beispiel von einem *Billboard*-Knoten gesprochen, so ist dies gleichbedeutend mit der Formulierung „Knoten des Typs *Billboard*“.
Da es zu diesem Thema viel Literatur gibt, wird in dieser Arbeit keine weitere Einführung zu VRML gegeben. Das Buch „*Dynamische Virtuelle Welten mit VRML 2.0*“ [6] gibt einen Einstieg in die Thematik.

### Alpha-Maske

Eine Alpha-Maske ist ein Graustufen-Bild, das neben den Farbkanälen existiert. Die Maske repräsentiert die Sichtbarkeitsinformationen jedes Bildpunktes eines Bildes. Weiß heißt voll sichtbar, schwarz komplett transparent. Bei Grautönen bestimmt der Schwarz-Anteil, wie durchsichtig ein Bildpunkt ist.

# 2 Der Branitzer Park als virtuelles Modell

## 2.1 Einführung zum Projekt „Virtueller Branitzer Park“

### 2.1.1 Allgemeines zum Park

Der Branitzer Park[5] liegt am Rande der Stadt Cottbus und wurde von Fürst Hermann von Pückler-Muskau (1785-1871) gestaltet. Die Parkanlage gilt als letzter großer Landschaftspark des 19. Jahrhunderts mit internationaler Bedeutung. Auch wenn in der Öffentlichkeit nur der innere Park als „der Branitzer Park“ bezeichnet wird, gehört auch der äußere Park zu dem Gesamtkunstwerk, das von Pückler geschaffen wurde. Das vom Fürsten gestaltete Areal umfasst circa 600 Hektar.
Die Trennung zwischen Außen- und Innenpark gab es zu Lebzeiten von Fürst Pückler nicht. Sie entstand im Laufe der Entwicklungsgeschichte des Parks durch mehrere Umstände. Aufgrund von finanziellen Problemen zum Beispiel, wurden einige Bereiche des Außenparks für die Agrarwirtschaft genutzt.
Der Zusammenhang zwischen innerem und äußerem Park nahm mit den Entwicklungen nach 1945 immer mehr ab. Völlig verloren schien der Außenpark, als die DDR offenbar plante, einige Bereiche der Förderung von Braunkohle zu opfern. Als diese Planungen nach 1990 abgesetzt wurden, begann man mit der Arbeit, den äußeren und inneren Park zu einem Gesamtkunstwerk wieder zusammenzuführen.

Zunächst war eine farbige Zeichnung aus dem Jahr 1903 die Arbeitsgrundlage zum Zusammenführen. Doch reichte dieser Plan nicht aus, um eine Gartendenkmalpfelge im Außenpark zu koordinieren. Diverse Quellen aus der Zeit von 1845 bis heute sind die Grundlage für alle Pflege- und Wiederherstellungsarbeiten. Auch eine Bestandsaufnahme des gesamten Geländes wurde gemacht. Aktuelle Fernerkundungsdaten wie Airbone-Laserscanning-Aufnahmen[1] und georeferenzierten Orthofotos[2] wurden mit den historischen Daten verglichen, um weitere Schritte planen zu können. Desweiteren wurde ein Baumkataster erstellt, um sich ein Bild über den Baumbestand zu machen. Es beinhaltet eine Auflistung aller Bäume des inneren Parks sowie deren Position und weitere exemplarspezifische Daten.

[1] Aus einem Flugzeug heraus wird die Oberfläche mittels Laser präzise abgetastet. Durch die Abtastung lässt sich ein Höhenprofil des überflogenen Bereichs erstellen, welches für spätere Zwecke, wie beispielsweise das Erstellen einer virtuellen Umgebung des Bereichs, verwendet werden können.

[2] Ein Orthofoto ist eine verzerrungsfreie, natur- und maßstabsgetreue fotographische Abbildung der Erdoberfläche.

### 2.1.2 Der virtuelle Park

In Zusammenarbeit des Systemhauses Stahr[3] aus Cottbus mit der Verwaltung des Branitzer Parks wurde damit begonnen, den inneren Park virtuell „nachzubauen“. Das Areal um das Schloss wurde bereits mit den Gebäuden virtuell nachempfunden und in VRML umgesetzt. Die Informationen des Geländes sind von einem CAD[4]-Modell abgeleitet worden. Zwar basiert das CAD-Modell auf Messdaten, doch ist die Umsetzung in das bisherige VRML-Modell manuell erfolgt. Desweiteren fehlen einige Teile des Parks in dem CAD-Modell. Somit ist das bisherige VRML-Modell zwar eine visuelle Nachempfindung, jedoch kein adäquater virtueller Ersatz. Dies gilt auch für die Vegetation, welche ebenfalls nur andeutungsweise visualisiert wurde.

Damit ein virtuelles Modell des Parks aussagekräftige Darstellungen liefern kann, wird der zugrundeliegende Datenbestand um die Fernerkundungsdaten ergänzt. Mit georeferenzierten Daten sowie Orthofotos wird derzeit, in einer zu dieser Arbeit parallel laufenden Bachelor-Arbeit, ein adäquates virtuelles Modell des gesamten inneren Parks in VRML erstellt.
Das Positionieren der Bäume sowie deren Modellierung kann nun durch das Baumkataster und die georeferenzierten Daten besser koordiniert werden. Mit dem Baumkataster können neben allen vorhanden Baumarten auch die Anzahl und die Positionen der einzelnen Exemplare im Geländemodell bestimmt werden. Dies reduziert den Aufwand der virtuellen Parkgestaltung, einzelne Elemente des Parks (wie Bäume zum Beispiel) genau zu positionieren.

Das Erzeugen der Baummodelle sowie von deren Repräsentationen wurde im bisherigen Projektverlauf nicht genauer verfolgt. Da dies ein wichtiges Thema für das Gesamt-Projekt des virtuellen Branitzer Parks ist, soll dieses Problem aufgegriffen, Varianten diskutiert und mögliche Lösungen präsentiert werden.

[3] http://www.shstahr.de/ – Letzter Zugriff: 07.12.2007
[4] *Computer Aided Design* (Rechnerunterstützte Konstruktion)

## 2.2 Baumauswahl – Grundlage zur Veranschaulichung

Die Auswahl von drei Bäumen wird dazu genutzt, das Verfahren in Abschnitt 3.2 zu veranschaulichen. Zusätzlich dient einer dieser Musterbäume, eine Erle, auch als Vorbild für die Modellierung in Kapitel 5. An dieser Stelle wird nun kurz ein Überblick über die verwendeten Bäume gegeben und gezeigt, welches Bilddaten-Material zur Verfügung stand.
Die gezeigten Aufnahmen sind dabei eine Auswahl von mehreren Aufnahmen, die jeweils von einem Baum gemacht wurden. Die Auswahl der Bilder beschränkt sich auf vier Seiten. Zusätzliche Detail-Aufnahmen werden hier nicht gezeigt, sind aber in den Anhängen mit aufgelistet.

### 2.2.1 Die Erle



Abbildung 2.1: Aufnahmen von vier Seiten der dreistämmigen Schwarz-Erle am Schloss im Branitzer Park.

Eine der zu betrachtenden Möglichkeiten, 3D-Modelle von Bäumen zu erzeugen, ist die Nutzung von Wachstums-Regeln[5]. Dies wird mit dem Programm GroIMP realisiert. Damit Wachstums-Regeln für einen bestimmten Baum erzeugt werden können, ist es wichtig, die Eigenschaften des betreffenden Baumes in Erfahrung zu bringen.
Für die Demonstration mit dem Programm GroIMP ist die Wahl auf die *Schwarz-Erle* (siehe Abbildung 2.1) am Schloss im Branitzer Park gefallen, da dieser Baum einfache geradlinige Stämme hat. Dies erleichtert das Ableiten von Wachstumsregeln. Zusätzlich ist der Standort in der Nähe des Schlosses optimal.
Die Schwarz-Erle (lat. *Alnus glutinosa*) ist ein Laubbaum. Sie ist in fast ganz Europa verbreitet. Die maximale Höhe, die sie erreicht, ist 35 m, und sie wird bis zu 150 Jahre alt. Ihr Stamm wächst gerade. Eine typische Erscheinung der Schwarz-Erle ist, dass sie,

[5]Die Erklärung zum Begriff Regeln wird zu Beginn des Kapitels 5 gegeben.

wie auf Abbildung 2.1 zu erkennen ist, mehrstämmig wächst. Die Blätter werden 4-10 cm lang und sind doppelt gesägt[22, Seite 35]. Die Blätter sind im Sommer grün, die Rinde ist schwarzbraun, rissig und schuppig.
Sie kommt in ebenem Gelände bis in mittleren Gebirgslagen vor, und ist häufig an Bächen oder Fluss-Ufern zu finden [22, Seite 35].

### 2.2.2 Die Eiche



Abbildung 2.2: Aufnahmen von vier Seiten der verwendeten Eiche im Park.

Die Stiel-Eiche ist eine der wichtigsten Laubbaumarten im forstwirtschaftlichen Bereich. Sie ist in fast ganz Europa verbreitet und wächst auf eine maximale Höhe von 50 m. Ihr maximales Alter liegt bei 800 Jahren [22, Seite 432].
Steht der Baum einzeln, befindet sich der Anfang der Baumkrone dicht über dem Erdboden. Anders als bei der Erle beginnt der Stamm früh, sich in starke Äste aufzuteilen.

### 2.2.3 Die Buche



Abbildung 2.3: Aufnahmen von vier Seiten der verwendeten Buche im Park.

Die Blutbuche *Fagus sylvatica purpurea* ist eine genetisch differenzierte Variante der Rotbuche *Fagus sylvatica* [22, Seite 169]. Sie erreicht eine Höhe von maximal 45 m und wird 300-500 Jahre alt. Steht sie im Freien, ist sie tiefbeastet. Die Baumkrone ist sehr breit.

# 3 Billboards in VRML

## 3.1 Prinzipien, Einsatzgebiete

### 3.1.1 Was sind Billboards?

Der Name *Billboard* (auf deutsch „Reklametafel“) bezieht sich auf die Art der Darstellung. Wie in der Realität auch wird ein Bild in Form einer Textur auf eine Fläche, in der Regel ein Rechteck, gelegt und dargestellt. Dabei hat die Textur teilweise transparente Bereiche. So können auch Gegenstände dargestellt werden, die nicht rechteckig sind. *Billboard*s dienen dazu, komplexe Modelle stark vereinfacht darzustellen. Dies ist notwendig, wenn es darum geht, möglichst viele Objekte darzustellen und den Aufwand für die benötigten Berechnungen so gering wie möglich zu halten. So kann ein Modell eines Baumes, das aus vielen 1.000 Polygonen besteht, durch 1 oder 2 Polygone mit einer Textur dargestellt werden.

*Billboards* werden auf zwei Arten eingesetzt. Die erste Variante benutzt ein Rechteck, das immer zur Kamera ausgerichtet wird, so dass immer die Vorderseite zu sehen ist. Die andere Variante nutzt zwei ineinander verschränkte Rechtecke. Diese werden nicht zur Kamera ausgerichtet.
Beide Varianten werden als nächstes aus Sicht von VRML erläutert.

#### Billboards als Rechtecke in VRML

Die einfachste Form eines *Billboard*s in VRML ist ein Rechteck, dessen Textur aus einem Bild besteht. Dazu wird der Knoten-Typ *IndexedFaceSet* genutzt. Durch Angabe von 4 Punkten im Raum werden Breite und Höhe des Rechtecks definiert. Wie ein *IndexedFaceSet* in VRML erstellt wird, zeigt Abbildung 3.1. Nach der Angabe der Punkte wird die Reihenfolge der Punkte, mit der die Fläche gezeichnet werden soll, bestimmt. Durch Festlegung der Reihenfolge wird die Orientierung der Fläche bestimmt. Die Erstellung ist mit der Angabe einer Textur abgeschlossen.
Ein Problem von Rechtecken mit Textur ist, dass beim Versuch, die Rechtecke von der Seite zu betrachten, diese scheinbar verschwinden. Der Grund ist einfach: Rechtecke, wie in Abbildung 3.1 definiert, sind zweidimensionale Objekte und sind somit unendlich dünn.

Eine Möglichkeit, dies zu unterdrücken, ist die Nutzung von Rechteck-Kreuzen. Hierzu werden zwei Rechtecke senkrecht zueinander aufgestellt. Von oben betrachtet erinnert die Anordnung der beiden Rechtecke an ein Kreuz. Die Methode hat den Vorteil, dass

```
#VRML V2.0 utf8

Shape {
   appearance Appearance {
      texture ImageTexture { url "Pfad zum Bild" }
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [ 1  0.5 0, -1  0.5 0,
            -1 -0.5 0,  1 -0.5 0 ]
      }
      coordIndex [ 0 1 2 3 -1 ]
      texCoord TextureCoordinate {
         point [ 1 1, 0 1, 0 0, 1 0 ]
      }
   }
}
```

Abbildung 3.1: Beispiel eines Rechtecks mit Textur.

beim Betrachten des Objektes mindestens ein Rechteck die Vorder- oder Rückseite zeigt. Verschwindet das erste Rechteck mit Textur, weil der Winkel zwischen dessen Oberfläche und dem Betrachter 0° beträgt, ist zum gleichen Zeitpunkt das zweite Rechteck sichtbar, da der Winkel der Oberfläche des zweiten Rechtecks nun 90° beträgt.
Somit können Objekte nicht mehr verschwinden. Jedoch erzeugt die Methode ein weiteres Problem, das mit dem VRML-Betrachter zusammenhängt. Beim Zeichnen der Flächen werden diese zunächst gewichtet. Dies dient als Festlegung der Reihenfolge, mit der die Flächen gezeichnet werden. Die Gewichtung ist abhängig vom Schwerpunkt einer Fläche. Das bedeutet, dass Flächen mit dem kleinsten Abstand des Schwerpunktes zum Betrachter als letztes gezeichnet und von keiner anderen Fläche überlagert werden. Sie liegen im Vordergrund. Die Fläche wird beim Zeichnen jedoch als Einheit betrachtet. Dadurch kommt es vor, dass Texturbereiche, die eigentlich verdeckt sein müssten, im Vordergrund liegen. Somit werden Texturbereiche wiederum verdeckt, obwohl sie im Vordergrund liegen sollten.
Dieses Problem wird von einigen VRML-Betrachtern abgefangen. Auch BS Contact der Firma Bitmangement stellt durchdringende Flächen mit Textur korrekt dar. Sobald die Textur transparente Bereiche besitzt, hat auch dieser VRML-Betrachter bei der Darstellung Probleme. Das linke Bild in der Abbildung 3.2 zeigt die fehlerhafte Darstellung.

Die in dieser Arbeit erzeugten Methoden und Überlegungen wurden hauptsächlich mit dem genannten VRML-Betrachter BS Contact untersucht. Damit solche Darstellungsfehler nicht mehr auftreten, werden Flächen an dem Punkt, wo sie sich durchdringen, geschnitten. Somit wird eine Seite durch zwei Rechtecke repräsentiert. Ein Rechteck stellt dabei die linke Hälfte, ein anderes Rechteck die rechte Hälfte der Textur einer Seite dar.

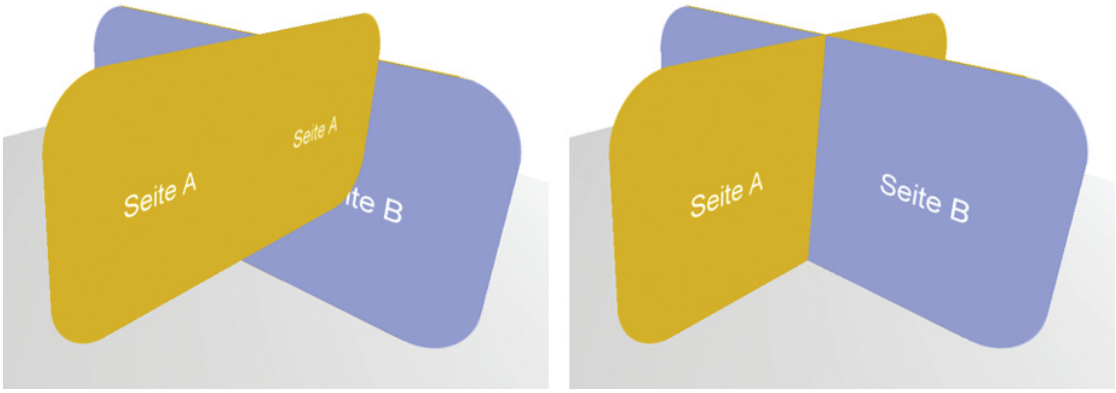


Abbildung 3.2: Das linke Bild zeigt ein Rechteckkreuz, aus zwei Rechtecken bestehend. Deutlich ist hierbei zu erkennen, dass der hintere Teil der Seite A normalerweise durch den vorderen Teil der Seite B verdeckt sein müsste. Auf dem rechtem Bild ist die Darstellung korrekt. Hier wurde an der Stelle der Durchdringung ein Schnitt gemacht. So besteht eine Seite aus zwei Rechtecken, die in der Mitte getrennt sind.

Die richtige Darstellung des Objektes ist auf dem rechten Bild der Abbildung 3.2 zu sehen.

#### Billboard als eigener Knoten in VRML

In VRML gibt es eine weitere Möglichkeit, das Verschwinden von Objekten beim Betrachten aus einen ungünstigen Winkel zu verhindern. Der *Billboard*-Knoten richtet die Vorderseite des Rechtecks zum Betrachter hin. Somit ist auch die Textur auf der Vorderseite stets sichtbar. Wie ein solcher Knoten in VRML angewendet wird, zeigt Abbildung 3.3.
Wenn das lokale Koordinatensystem zum Betrachter ausgerichtet wird, rotiert dieses um eine vorher angegebene Achse. Für die Zwecke der Baum-Darstellungen genügt eine Rotation um die y-Achse (vertikale Achse) des lokalen Koordinatensystems. Als Standard-Wert ist für *axisOfRotation* der Wert *0 1 0* angegeben. Daher ist die Angabe wie in Abbildung 3.3 nicht notwendig.
Ebenso wie die Achse, um die der *Billboard*-Knoten bei der Ausrichtung rotiert wird, kann auch der Rotationspunkt angegeben werden (*bboxCenter 0 0 0*). Dieser Parameter hat als Standard-Wert *0 0 0* und muss somit nicht extra angegeben werden, sofern das Zentrum nicht verschoben werden muss.
Mit dieser Methodik verschwinden Objekte nicht mehr, und es bleibt ein dreidimen-

sionaler Eindruck bestehen [20, Seite 51], was insbesondere für komplexe Objekte, wie Pflanzen oder Bäume, von Vorteil ist. Durch das Ausrichten wird unterdrückt, dass es sich beim betrachteten Objekt um eine Fläche mit Bild handelt.

```
#VRML V2.0 utf8

Billboard {
   axisOfRotation 0 1 0
   bboxCenter 0 0 0
   children Shape {
      appearance Appearance {
         texture ImageTexture { url "Pfad zum Bild" }
      }
      geometry IndexedFaceSet {
         coord Coordinate {
            point [ 1  0.5 0, -1  0.5 0,
               -1 -0.5 0,  1 -0.5 0 ]
         }
         coordIndex [ 0 1 2 3 -1 ]
         texCoord TextureCoordinate {
            point [ 1 1, 0 1, 0 0, 1 0 ]
         }
      }
   }
}
```

Abbildung 3.3: Beispiel eines Billboard-Knotens mit einem texturierten Rechteck.

## 3.2 Fahrplan zur Umsetzung mit Bildvorlagen

Die erste der drei vorgestellten Varianten zum Erzeugen eines *Billboard*-Baummodells, ist die Verwendung von Bildvorlagen. Es werden die Hintergründe auf den Bildvorlagen der betroffenen Bäume „herausgeschnitten“. Die nun freigestellten Bäume sind somit bereit zum Einsatz in einer VRML-Szene. Die einzelnen Schritte vom Bild über das Freistellen bis hin zum Einsatz in einer VRML-Szene werden in diesem Kapitel anhand der drei im Abschnitt 2.2 bereits vorgestellten Bäume demonstriert.

Es wird zunächst auf die Anforderungen an das Bildmaterial eingegangen und diese mit dem gegebenen Bildmaterial verglichen. Daraus leiten sich die Arbeitschritte für die drei Bäume ab.

### 3.2.1 Anforderungen an das Bildmaterial

Damit ein Baum optimal dargestellt wird und nicht aufgrund eines ungünstigen Betrachtungswinkels scheinbar verschwindet, sollte er mittels *Billboard*-Knoten oder Rechteck-Kreuz dargestellt werden. Da der *Billboard*-Knoten immer die gleiche Seite dem Betrachter zeigt, ist auch nur eine Aufnahme des Baumes für die Repräsentation vorzubereiten. Sollen mehrere Seiten des Baumes repräsentiert werden, bietet sich die Nutzung eines Rechteck-Kreuzes an.
Da die Rechtecke aus Vorder- und Rückseite bestehen und ein Rechteck-Kreuz zwei Rechtecke besitzt, besteht das Kreuz insgesamt aus vier Seiten. Das bedeutet, dass für jeden Baum mehrere Ansichten repräsentiert werden können. Damit ein Rechteck-Kreuz optimal genutzt wird, sollten daher mindestens zwei, optimal vier Seiten-Ansichten vorliegen. Gibt es für den Baum nur zwei Bildvorlagen, so werden auf die einzelnen Rechtecke der Vorder- und Rückseiten die gleichen Bilder gelegt. Dies kann das dargestellte Ergebnis bei der Betrachtung in der VRML-Szene verfälschen. Daher werden die Anforderungen für die optimale Darstellung mit vier Seiten aufgelistetet. Die Anforderungen für zwei Seiten sind analog dazu.
Bei der Erstellung des Bildmaterials sollten die folgenden Anforderungen berücksichtigt werden, da sonst die Qualität des Ergebnisses darunter leidet. Die Anforderungen lauten wie folgt:

- Freie Sicht auf den betreffenden Baum.
- Alle Bilder wurden mit gleichem Abstand zur Mitte des Baums erstellt.
- Der Winkel im Mittelpunkt des Baums zwischen zwei benachbarten Seiten beträgt 90°.
- Auf den Bildern ist eine Orientierungshilfe, die es später bei der Bearbeitung erleichtert, die einzelnen Seiten auseinander zu halten.
- Die Kamera muss parallel zum Horizont gehalten werden: Kein Neigen nach vorn oder hinten, links oder rechts.

- Die Höhenlage der Kamera in Bezug zum Baum darf sich nicht ändern.
- Die Aufnahmen für eine Jahreszeit sollten am gleichen Tag zu gleicher Zeit gemacht sein, da die Lichtverhältnisse auf allen vier Bildern identisch sein sollen.
- Für eine gleichmäßige Beleuchtung des Baumes ist möglichst ein Tag mit Bewölkung zu wählen. Scheint die Sonne, ist auch auf den vorbereiteten Bildern zu erkennen, von wo die Sonne scheint. Das macht das Anpassen an eine Szenerie mit anderem Sonnenlicht-Einfall schwieriger.
- Gleiche Belichtungseinstellungen der Kamera.

Damit Bäume auch in anderen Jahreszeiten darstellbar sind, bietet sich die Prozedur für jede Jahreszeit an. Dabei sollten die Aufnahmen jeder Seite von der gleichen Position aus gemacht werden.

### 3.2.2 Vorhandenes Bildmaterial

Wie in Abbildung 3.4 exemplarisch zu sehen ist, erfüllen nicht alle Aufnahmen die Anforderungen. So ist der Abstand und die Position zum jeweiligen Baum auf jeder Aufnahme unterschiedlich. Hinzu kommt, dass auf einigen Bildern die Kamera eine leichte Neigung nach unten oder nach oben hat. Die Aufnahmen von zwei Seiten unterscheiden sich in der Belichtung. So erscheint der gleiche Baum aus einer Perspektive dunkler als aus einer anderen. Wenn ein Großteil des Bildes den Himmel abbildet, ist der Rest des Bildes verhältnismäßig dunkel, so dass keine Details der Baumkrone zu erkennen sind, sondern nur Umrisse.
Einige der Aufnahmen zeigen den Baum nicht komplett. Das heißt, es fehlen Elemente des Baums. Diese sind auf anderen Aufnahmen aus ähnlicher, jedoch leicht versetzter Position wiederzufinden.

Die genannten Probleme müssen vor dem Freistellen behoben werden. So müssen die Lichtverhältnisse und Farben auf allen vier Aufnahmen eines Baumes einander angepasst werden. Zusätzlich ist es notwendig, fehlende oder überdeckte Bereiche auf Aufnahmen zu ergänzen. Dazu muss das gesamte Bildmaterial nach ähnlichen Aufnahmen durchsucht werden, die diese Bereiche zeigen. Diese werden dann ausgewählt und über den Bereich, der ergänzt werden soll, gelegt.

### 3.2.3 Arbeitsschritte

Bevor mit den Arbeitsschritten begonnen werden kann, werden zunächst aus dem vorhandenen Bildmaterial die Bilder ausgewählt, die jeweils eine Seite des Baumes am besten repräsentieren und die Anforderungen am besten erfüllen. Das restliche Bildmaterial kann bei dem Arbeitsschritt *Ergänzung* jedoch noch zum Einsatz kommen.
Die folgenden Arbeitsschritte lauten:

- Anpassung: Farben, Helligkeit und Kontrast der Aufnahmen für einen Baum miteinander abgleichen
- Ergänzung: Fehlende oder überdeckte Elemente ergänzen
- Freistellung: der Hintergrund auf dem Bild wird transparent
- Erzeugung: der Darstellung in VRML

Anhand des Beispiels der Erle werden nun alle Arbeitsschritte durchgeführt.

### Anpassung

In diesem Schritt werden nach Möglichkeit alle vier Bilder nebeneinander gelegt. Danach können Helligkeit, Kontrast sowie Farbwerte aneinander angepasst werden. Da der Abstand vom Baum zur Kamera auf allen Bildern variiert, müssen die Bilder unterschiedlich skaliert (vergrößert oder verkleinert) werden. Der betroffene Baum muss auf allen Aufnahmen die gleiche Größe besitzen. Es empfiehlt sich, alle Bilder auf das Bild mit der kleinsten Baumdarstellung hin anzupassen, also zu verkleinern. Denn das Vergrößern von Bilder lässt diese unscharf werden.
Abbildung 3.4 zeigt oben die vier Aufnahmen vor der Bearbeitung. Die vier Aufnahmen darunter sind das Ergebnis nach der Bearbeitung. Zur besseren Orientierung wurden die Seiten nummeriert.

### Ergänzung

War es nicht möglich, den Baum zu fotografieren, ohne dass die Sicht versperrt wurde, werden die fehlenden Elemente in diesem Arbeitsschritt ergänzt. Das Prinzip dieses Schrittes besteht darin, die überdeckten Bereiche mit ähnlichen Bildelementen zu überlagern. Diese Elemente können auf dem gleichen Bild liegen oder werden aus anderen Bildern entnommen. Wichtig ist, dass im Ergebnis diese überlagerten Bereiche nicht auffallen. Daher müssen die Licht, Farb- und Strukturverhältnisse auf dem neu hinzukommenden Element möglichst mit der Vorlage übereinstimmen.
Fehlende Elemente des Baumes auf dem selben Bild zu suchen, bietet sich dann an, wenn der Bereich, der damit zu überlagern ist, nicht besonders groß ist. Da das menschliche Auge Symmetrien gut erkennt, sind Elemente, die mehr als einmal im Bild auftauchen, vom Betrachter schnell ausgemacht. Abbildungen 3.5 und 3.6 demonstrieren die Arbeitsschritte für das Ersetzen mit einem Element eines anderen Bildes.

Auf der Bildvorlage für Seite 4 der Baum-Darstellung (Abbildung 3.4 Bild ganz rechts) fehlen im linken und rechten Bereich Bildelemente. Speziell der linke Bereich wird sogar großflächig verdeckt. Da auf dem selben Bild passende Elemente nicht zu finden sind, wurde aus dem gesamten Bildmaterial eine Aufnahme mit ähnlicher Perspektive und ähnlichen Lichtverhältnissen gesucht. Das zweite Bild der Abbildung 3.5 erwies sich am geeignetsten. Es ist eine Aufnahme zu einem anderen Zeitpunkt und von einer leicht veränderten Position. Jedoch ähneln sich die Aufnahmen und der betroffene Bereich.




Abbildung 3.4: Die oberen vier Bilder dieser Abbildung zeigen die Aufnahmen ohne, die unteren vier Aufnahmen mit der Farb-, Helligkeits- und Kontrastanpassung. Die oberen Bilder sind bereits in der Größe aneinander angepasst.

Auf dem gefundenen Bild wurde der linke untere Bereich der Erle kopiert und in das zu bearbeitende Bild (Arbeitsbild) eingefügt. Die Größe wurde soweit angepasst, dass die Größe der Blätter auf dem neuen Element mit der Größe der Blätter auf dem Arbeitsbild übereinstimmt. Das Ergebnis nach dem Einfügen des Bildelementes ist auf dem ersten Bild der Abbildung 3.6 zu sehen.
Damit der Übergang zwischen dem neuen Element und dem darunter liegenden Arbeitsbild nicht mehr zu erkennen ist, müssen die Übergänge „verwischt“ werden. Dies wird mit einer sogenannten Alpha-Maske realisiert. Für den Bereich der Kante werden Graustufen-Verläufe gezeichnet. Das Ergebnis ist das zweite Bild in Abbildung 3.6.

Dies muss für alle verdeckten Bereiche im Bild durchgeführt werden. Das Endergebnis dieses Schrittes ist, dass alle verdeckten Bereiche durch andere Elemente ersetzt sind. Abbildung 3.7 zeigt das gleiche Arbeitsbild vor und nach dem Arbeitschritt.

### Freistellung

Da nun alle vier Bilder des Baumes angeglichen sind und keine Bereiche der Bilder verdeckt werden, kann nun der Hintergrund ausgeschnitten werden. Dazu kommt erneut die




Abbildung 3.5: Das linke Bild zeigt das Arbeitsbild mit dem überdeckten Bereich (linker weißer Kasten). Dieser Bereich wird durch ein Bildelement aus einem anderen Bild (rechts) ersetzt.

Alpha-Maske zum Einsatz. Das Ausschneiden des Hintergrundes geschieht nach zwei Methoden. Die eine nutzt den starken Kontrast zwischen Baum und Himmel aus – die Erle hebt sich deutlich vom Himmel (Hintergrund) ab. Die andere Methode benötigt einen manuellen Eingriff. Da der untere Bereich des Baumes keine klare Trennung zwischen Vordergrund (Erle) und dem Hintergrund (Vegetation) erkennen lässt, muss hier mit einem Zeichenwerkzeug direkt auf der Alpha-Maske gezeichnet werden. Dabei werden die sichtbaren und die nicht sichtbaren Bereiche des Bildes festgelegt. Mit dem Zeichenwerkzeug werden auf der Alpha-Maske die Bereiche schwarz gezeichnet, die nicht zum Vordergrund gehören.

Abbildung 3.8 zeigt, wie zunächst der Himmel entfernt wurde. Dazu wird eine Auswahl erstellt, die all diejenigen Bildpunkte auswählt, deren Farbwerte mit denen des Himmels annähernd übereinstimmen. Somit sind alle Bereiche auf dem Bild bekannt, die den Himmel darstellen. Diese Bereiche werden in der Alpha-Maske auf schwarz gesetzt. Die entsprechenden Bereiche des Bildes werden dadurch transparent. Im nächsten Zwischenschritt wird der Hintergrund des unteren Bereiches entfernt. Dazu kommt ein Zeichenwerkzeug zum Einsatz. Hiermit zeichnet der Benutzer des Werkzeugs direkt auf der Alpha-Maske und bestimmt die sichtbaren und nicht sichtbaren Bereiche des Bildes.




Abbildung 3.6: Das Ergebnis nach Anpassung der Größe zeigt das Bild links. Der Übergang vom neuen Element zum Arbeitsbild ist noch hart. Daher werden die Kanten weicher gezeichnet, so dass der Übergang zwischen dem Element und dem Bild sanfter ist. So schmiegt sich das neue Element in das Arbeitsbild ein und der fehlende Bereich ist überlagert (rechtes Bild).

Dies ist ein aufwändiger Arbeitschritt. Die Dauer hängt vom freizustellenden Objekt sowie der Bildqualität ab. Das Endergebnis ist, dass die Erle vom Hintergrund befreit wurde.

Wie auf dem dritten Bild der Abbildung 3.8 zu sehen ist, ist der untere Bereich wesentlich dichter als der obere Bereich des Baumes. Dies hängt von den Methoden des Freistellens ab, welche sich im oberen und unteren Bereich unterscheiden. Während im oberen Bereich alle Bildpunkte mit bestimmtem Farbwert entfernt wurden, musste im unteren Bereich der Benutzer selbst entscheiden, ob ein Bildpunkt zum Vorder- oder Hintergrund gehört. Da der Hintergrund wie der Vordergrund größtenteils aus Vegetation besteht ist eine Unterscheidung kaum möglich. Dies ist ein Nachteil dieser Methodik.



Abbildung 3.7: Das Ergebnis vor (links) und nach (rechts) diesem Arbeitsschritt.

### Erzeugung

Von jeder der vier Seiten existiert nun ein vom Hintergrund befreites Bild. Diese werden im folgenden Arbeitsschritt auf ein Rechteck-Kreuz gelegt. Wie bei den Rechteck-Kreuzen in Abschnitt 3.1 beschrieben, wird die Textur einer Seite halbiert und auf zwei Rechtecke aufgespannt.

Aus dem Arbeitsschritt *Freistellung* werden nun die Bilddateien für die Textur erstellt. Die Ausmaße der Bilddateien sind quadratisch. Das heißt, dass sie genauso breit wie hoch sind. Dadurch müssen beim Positionieren der Textur keine Seitenverhältnisse beachtet werden.
Der Baum einer Ansicht wird in der Bilddatei so ausgerichtet, dass der Ansatz des Stammes auf der Hälfte der Breite liegt. Dadurch liegen die jeweils gegenüberliegenden



Abbildung 3.8: Das Ausgangsbild (links) wird zunächst vom Himmel befreit (Mitte). Als nächstes wird der untere Bereich freigestellt (rechts).

Seiten im Rechteck-Kreuz möglichst deckungsgleich.
Wie die Erle als Rechteck-Kreuz in VRML dargestellt aussieht, zeigt Abbildung 3.9.

### 3.2.4 Ergebnis

Die aufgeführten Arbeitsschritte wurden auf alle aus Punkt 2.2 genannten Bäume angewendet und entsprechende VRML-Dateien erzeugt. Zur Erstellung der einzelnen Baumdarstellungen werden einige der aufgetretenen Probleme kurz angerissen. In der Auswertung in Abschnitt 6.2 wird dann darauf eingegangen.

#### Erle

Bei der Betrachtung der Erle von verschiedenen Seiten entsteht der Eindruck, dass sie aus mehr als drei Hauptstämmen besteht. Dieses Problem rührt von der Darstellung von Elementen, die zeitgleich aus verschiedenen Perspektiven zu sehen sind. Auf dieses Problem wird in Punkt 3.3.2 gesondert eingegangen.
Dass ein- und dieselben Hauptstämme mehrmals von einem Betrachtungspunkt aus zu sehen sind, hängt auch mit der Struktur der Baumkrone zusammen. Diese Erle besitzt eine sehr offene Struktur. Das heißt, dass man verhältnismäßig viel vom Inneren der Baumkrone zu sehen bekommt, anders als beispielsweise bei der Eiche. Die Eiche besitzt



Abbildung 3.9: Die Erle mit fotobasierter Bildvorlage in VRML dargestellt. Die Erstellung dauerte circa 5 Stunden.

eine sehr dichte Baumkrone und versperrt somit den Blick in das Innere.
Durch unterschiedliche Lichtverhältnisse beim Fotografieren ist auf einigen der Aufnahmen der Himmel überbelichtet. Das führte dazu, dass die feinen Strukturen überblendet wurden. Dies erschwerte den Freistellungsprozess und minderte die Qualität des Ergebnisses.
Das Freistellen im unteren Bereich der Erle erwies sich als besonders schwierig. Hier war kaum zu erkennen, was zum Baum gehört und was zum Hintergrund. Daher ist der untere Bereich verhältnismäßig dicht im Gegensatz zu höheren Bereichen der Baumkrone.

### Eiche

Anders als bei der Erle hat die Eiche eine relativ geschlossene Baumkrone, die die Sicht in das Innere versperrt. Daher ist das Problem der Falsch-Darstellung in VRML auch nicht so präsent. Auch wenn Struktur-Elemente zeitgleich auf mehreren Seiten des Rechteck-Kreuzes zu sehen sind, können diese in der Regel vom Betrachter nicht zugeordnet werden. Das heißt, dass dieser Effekt nicht so störend auftritt, wie bei der Erle mit ihren drei Hauptstämmen, die mehrmals auf den Rechtecken des Kreuzes zu sehen waren.
Auf keiner der Aufnahmen der Eiche waren Elemente großflächig verdeckt. Auch das Freistellen erwies sich aufgrund der geschlossenen Baumkrone als einfacher. So konnte dieser Baum innerhalb von circa 2 Stunden freigestellt werden.

Die Qualität des Bildmaterials der Eiche war im Vergleich zu den anderen beiden Bäumen am besten. Deshalb benötigte das Freistellen dieses Baumes weniger Zeit.



Abbildung 3.10: Die Eiche mit fotobasierter Bildvorlage in VRML dargestellt. Die Erstellung dauerte circa 2 Stunden.

### Buche



Abbildung 3.11: Die Buche mit fotobasierter Bildvorlage in VRML dargestellt. Die Erstellung dauerte circa 7 Stunden.

Von allen drei freizustellenden Bäumen war das Bildmaterial der Blut-Buche qualitativ das schlechteste. So mussten auf allen Seiten-Ansichten fehlende Elemente ergänzt werden.
Auch wenn die Baumkrone nicht so offen ist wie die der Erle, so sind doch einige Bereiche des Hintergrundes durch die Baumkrone hindurch zu erkennen. Zusätzlich wurde das Freistellen durch Überbelichtung auf fast allen Aufnahmen des Baumes erschwert. Daher dauerte das Freistellen dieses Baumes auch am längsten, im Vergleich zu den anderen beiden Bäumen.

Wie schon angesprochen, ist die Qualität des gegebenen Bildmaterials im Vergleich zu den anderen Bäumen schlecht. Dies schlägt sich auch in der Qualität der Repräsentation

in VRML nieder. Durch die Überbelichtung war die Farbe der Baumkrone so verwaschen, dass diese mit Rot „nachgefärbt“ werden musste. Ein weiterer negativer Effekt der Überbelichtung war in Bereichen mit feineren Strukturen zu beobachten. Kleine Zweige und Äste waren an ihren Kanten so sehr verwischt, dass auch an diesen Stellen eine zusätzliche Einfärbung durchgeführt werden musste.

## 3.3 Möglichkeiten in VRML

Auf Grundlage der Nutzung von Rechtecken mit Textur und des *Billboard*-Knotens, wie in Abschnitt 3.2 beschrieben, soll nun untersucht werden, welche weiteren Möglichkeiten der Darstellung in VRML zur Verfügung stehen. Dabei werden drei Verfahren näher betrachtet und das jeweilige Prinzip dahinter erklärt. Diese Verfahren werden zur Baum-Darstellung in Abschnitt 5.3.6 implementiert und genutzt.
Die drei Verfahren werden miteinander verglichen und mögliche Einsatzgebiete besprochen.

Um die einzelnen Verfahren zu demonstrieren, wurden entsprechende VRML-Szenen angelegt. Sie bestehen aus dem Billboard-System, einer Grundfläche und einfachen geometrischen Objekten an den Ecken der Grundfläche. Sie dienen zur Orientierung im Raum beim Betrachten, haben also nichts mit dem jeweiligen Billboard-System zu tun.

### 3.3.1 Dynamische Billboards

Wie bereits beschrieben wurde, kann der *Billboard*-Knoten in VRML seine Kind-Knoten immer zum Betrachter hin ausrichten. Eine einfache Verwendung dieses Knotens ist ein Rechteck mit einer Textur darauf, welches immer zum Betrachter gedreht wird.
Der Nachteil ist jedoch, dass der Betrachter IMMER die gleiche Seite ansieht. Das mag bei Ansichten mit vielen Objekten kaum auffallen, wie zum Beispiel ein Park mit vielen Bäumen. Jedoch bei näherer Begutachtung des Objekts durch den Betrachter ist dies bemerkbar. Eine vom Betrachtungs-Winkel abhängige Darstellung würde dieses Problem lösen. Je nachdem, wie der Winkel vom Betrachter zum Objekt ist, wird auf dem *Billboard* ein anderes Bild mit einer anderen Ansicht des Objektes angezeigt. Abbildung 3.12 verdeutlicht das Prinzip. Um das Objekt herum werden Sektoren eingeteilt. Für jeden Sektor wird ein Bild auf dem *Billboard* angezeigt.
Das bedeutet zum Einen, dass es von einem Objekt mindestens zwei oder mehr Ansichten geben muss, zum Anderen muss eine Möglichkeit gefunden werden, wie der Winkel berechnet und das Bild gewechselt werden kann. Um das Prinzip hier näher zu erläutern, wird davon ausgegangen, dass es von einem Objekt bereits mehrere Ansichten gibt. Wie ein 3D-Modell in mehreren Ansichten gerendert werden kann, wird im Abschnitt 5.3.6 demonstriert. So ist in diesem Abschnitt ausschließlich die Umsetzung in VRML interessant.

Für die Umsetzung dieses Prinzips sind Ereignisse und Routen sehr wichtig. Ereignisse sind Nachrichten, die von einem VRML-Knoten zur Kommunikation genutzt werden [6, Seite 31 - 33], wenn es zu Zustandsänderungen kommt. Über eine angegebene Route kann dann die Kommunikation zugeleitet und eventuell ein neues Ereignis ausgelöst werden.

Als erstes wird in einer neuen VRML-Datei ein Prototyp definiert. Er besteht aus einem *Billboard*-Knoten mit einem *IndexedFaceSet* und einer Textur darüber. Der Pfad zu

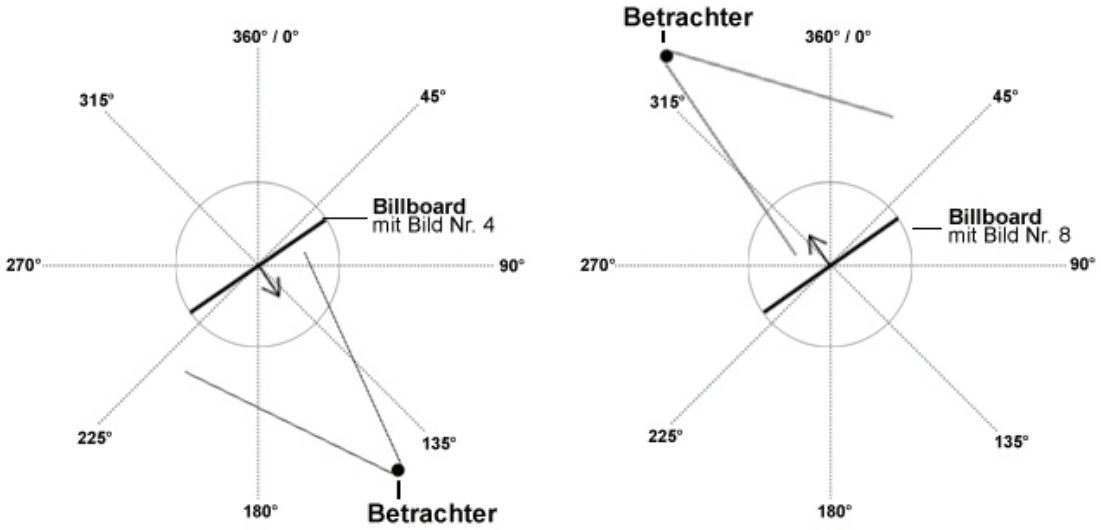


Abbildung 3.12: Prinzip eines dynamischen Billboards.

dieser Textur ist ein Feld des Parameters. Der Wert (Pfad zu dem Bild) wird dann bei der Instanzierung des Prototypen festgelegt. Dies geschieht mit der Auflistung der Instanzen des Prototypen in einem *Switch*-Knoten (zu deutsch Schalter). Dieser Knoten dient dazu, im Rahmen der Interaktivität Knoten bei Ereignissen zu aktivieren oder zu deaktivieren [20, Seite 47]. Innerhalb dieses Knotens wird für jede Ansicht des Objektes ein Prototyp instanziert. Ein Objekt mit 8 Seiten hat dann 8 Instanzen des Prototyps, und jede der Instanzen hat eine eigene Textur. Der *Switch*-Knoten zeigt immer nur eine Instanz zu einem Zeitpunkt an. Über das Feld *whichChoice* kann dann die gewünschte Ansicht des Objektes „angeschaltet“ werden.
Als nächstes muss die Position des Betrachters zum Objekt ermittelbar sein. Dazu kommt der Knoten-Typ *ProximitySensor* zum Einsatz. Dieser Knoten überwacht dabei einen quadratförmigen Bereich und sendet Ereignisse, wenn sich der Betrachter innerhalb des Bereichs bewegt. Dieser Bereich kann so eingestellt werden, wie es nötig ist, um die Winkelberechnung durchzuführen. Da es nicht unbedingt erforderlich ist, dass immer solch eine Berechnung durchgeführt werden muss, kann man dies auf den nahen Bereich des Objektes beschränken. Beim Festlegen der Position des *ProximitySensor* ist darauf zu achten, dass das Zentrum des Quaders im Zentrum des *Billboard*-Systems liegt.
Mit den Positionsdaten und dem *Billboard*-Schalter kann nun durch eine Funktion die winkelabhängige Darstellung umgesetzt werden. VRML bietet die Möglichkeit, Java-Scripte einzubinden. Erst mit ihnen ist dieses Verfahren möglich. Der *Script*-Knoten wird als *SCRIPT* definiert. Dies ist wichtig, da für die Ereignisse auch Routen definiert werden müssen, und dazu werden Namen von Knoten benötigt. Als Eingangs-Ereignis erwartet der Knoten Positionsdaten, also einen Vektor (x, y, z).

Da bei diesem verwendeten Billboard-System nur eine Rotation auf der y-Achse stattfindet, spielen Neigung und Blickrichtung des Betrachters keine Rolle und können bei der Berechnung außer Acht gelassen werden.
Diese Daten kommen später von dem *ProximitySensor*. Das Ausgangs-Ereignis ist dann eine Nummer, die abhängig von den Positionsdaten ist. Diese Nummer entspricht einer Instanz im Schalter. Hat ein Betrachter den Winkel 0° zum Objekt, so wird das erste Bild im Schalter angezeigt. Wann welche Instanz angeschaltet wird, hängt auch von der Anzahl der Ansichten für dieses Objekt ab. Hat das Objekt nur zwei Ansichten, so wird das erste Bild angeschaltet, wenn der Betrachter einen Winkel im Bereich 0° – 180° hat. Das zweite Bild wird dann bei 180° – 360° angezeigt. Die allgemeine Formel zur Berechnung der Bildnummer $b$ in Abhängigkeit des Betrachtungswinkels $\alpha$ und der Anzahl aller Seitenansichten $n$ lautet:

$$b = \frac{n}{360°} \cdot \alpha$$

$$n : 1 \leq n < \infty; n \in \mathbb{N}$$

$$\alpha : 0 \leq \alpha \leq 360; \alpha \in \mathbb{R}$$

$$b : 0 \leq b < n; b \in \mathbb{N}$$

Wohin welches Ereignis geleitet werden soll, wird durch eine Routen-Definition angegeben. So leitet der *ProximitySensor* die Positionsdaten an den *Script*-Knoten, welches dann die entsprechende Instanznummer berechnet. Diese wird dann vom *Script*-Knoten zum Schalter geleitet. Durch die Routen-Definition wird nun automatisch die Sicht auf das Objekt gewechselt, wenn sich der Winkel zum Objekt ändert. Der Ablauf wird in Abbildung 3.13 vereinfacht dargestellt.

Um das Ergebnis zu begutachten, wird der Baum aus der Start-Beispiel-Datei des Programms GroIMP[1] genommen. Der Baum besteht aus Zylindern und Kugeln. Durch die offene Baumkrone ist die Struktur im Innern gut erkennbar. Dadurch eignet sich das Bild ideal zur Verdeutlichung der einzelnen Methoden.
Die auf Abbildung 3.14 zu sehenden Screen-Shots stammen aus einer VRML-Szene mit einem dynamischem Billboard. Von dem Objekt wurden 8 Seiten gerendert, so dass 8 verschiedene Ansichten des Objektes zu sehen sind. Geht man um den Baum herum, wird das gerade angezeigte Bild immer zum Betrachter hin gedreht. Gelangt man nun in einen Grenz-Bereich des derzeitigen Sektors, ist ein „Sprung“ des Bildes wahrzunehmen, wenn der derzeitige Sektor verlassen und ein neuer betreten wird. Da der Betrachter nun in einem anderen Sektor ist, wird auch abrupt ein anderes Bild angezeigt. Das heißt, dass der Übergang nicht fließend ist. Dem kann entgegen gewirkt werden, indem man die Anzahl der Seiten erhöht. So haben Tests gezeigt, dass ab 20 Bildern die Übergänge zwischen den einzelnen Ansichten schon relativ sanft sind. Also gilt hier, je mehr Bilder,

[1] Das Programm GroIMP wird in Kapitel 5 genauer besprochen.

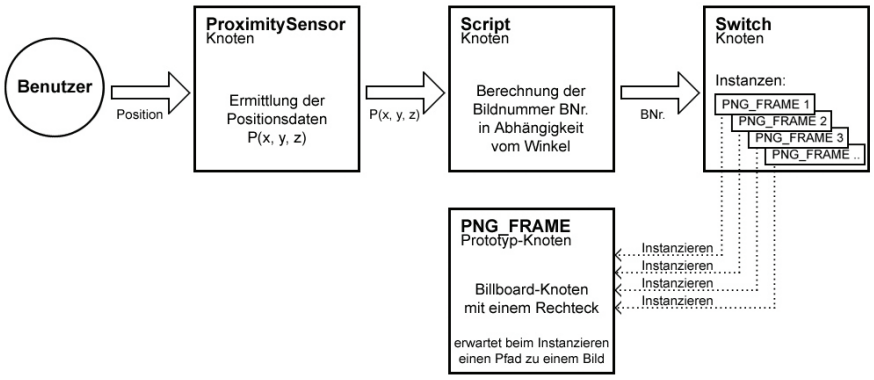


Abbildung 3.13: Veranschaulichung der Routen-Definition. Der Benutzer bewegt sich innerhalb des *ProximitySensor*s und löst somit ein Ereignis aus. Das Ereignis und somit auch die Positionsdaten des Benutzers innerhalb des Sensors werden an den *Script*-Knoten übermittelt. Der Script-Knoten berechnet anhand der x- und z-Werte den Winkel des Betrachters relativ zum dargestellten Objekt, also dem Ursprung des *Billboard*-Systems. Mit diesem Winkel wird danach berechnet, welche Bildnummer anzuzeigen ist. Die Bildnummer wird an den *Switch*-Knoten gesendet. Dieser schaltet dann die entsprechende Instanz des *Billboard*-Knoten ein.

desto sanfter die einzelnen Übergänge.

Da viele VRML-Betrachter nur so viele Informationen laden, wie zu diesem Zeitpunkt auch wirklich angezeigt werden, muss eine Vorlade-Funktion integriert werden, die alle Bilder des Billboard-Systems vorauslädt, auch wenn sie noch nicht benötigt werden. Denn geht der Betrachter um das Objekt und ein Bild muss geladen werden, das noch nicht im Cache des VRML-Betrachters liegt, erscheint kurzzeitig eine graue Fläche. Dies zerstört – auch bei vielen Ansichten – einen möglichen fließenden Übergang zwischen zwei Ansichten. Das Vorausladen für jedes Objekt mit mehreren Seiten-Ansichten führt jedoch zu längeren Wartezeiten.
Eine andere Möglichkeit wäre, zwei Billboards ineinander zu blenden. Das bedeutet, dass zwischen zwei Ansichten die Bilder transparent übereinandergelegt werden. Dies kann dann aber zu Laufzeit-Einbußen führen, wenn relativ häufig Transparenz-Berechnungen durchgeführt werden müssen. Aber auch diese Methode würde viele Ansichten voraussetzen, damit ein fließender Eindruck entsteht.

### Zwischenfazit

Durch die benötigte hohe Anzahl von Bildern, die noch zu Beginn geladen werden sollten, ist diese Variante nicht unbedingt für die internetfähige Präsentation des Parks geeig-

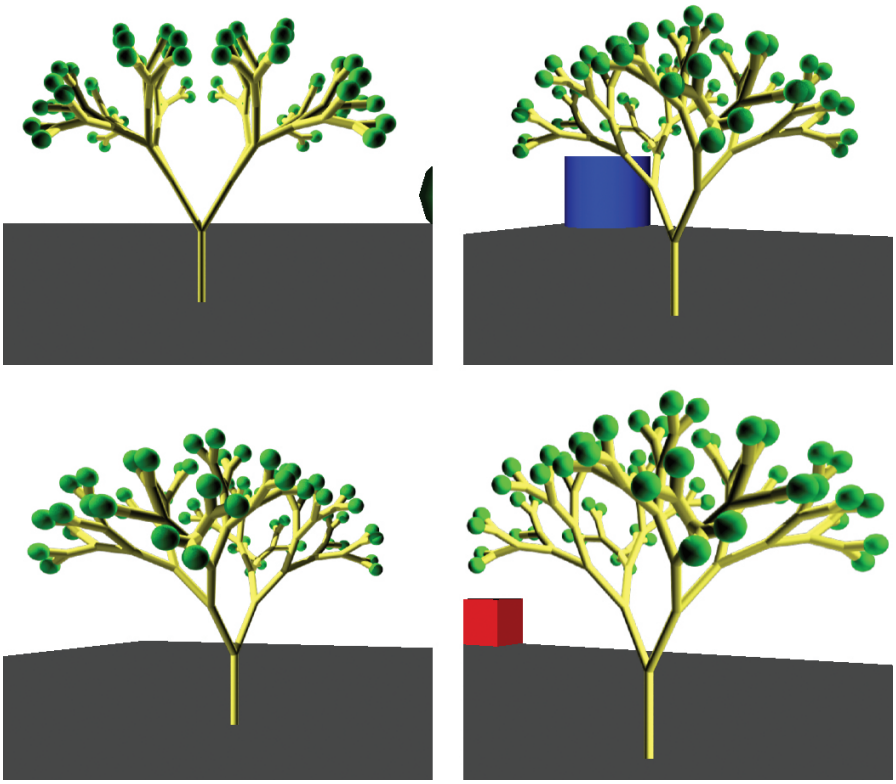

Abbildung 3.14: Beispiel mit dynamischem Billboard.

net. Für eine Präsentation, die schon auf einem Rechner mit dem gesamten Bildmaterial abgelegt ist und wo Verzögerungen durch Herunterladen nicht auftreten, ist dieses Prinzip aber durchaus brauchbar. Hinzu kommt das Problem der Plattform-Unabhängigkeit. Tests zeigten, dass nicht jeder VRML-Betrachter den *Script*-Knoten interpretieren konnte. Das Ergebnis war, dass kein Objekt angezeigt wurde. Getestet wurde das Prinzip unter Linux mit dem Programm FreeWRL (Version 1.19.6). Dieser konnte den Script-Knoten nicht interpretieren. Unter Windows XP wurde das Prinzip mit dem Betrachter BS Contact Player (Version 6.207), sowie dem Cosmo Player (Version 5.1) erfolgreich getestet.

### 3.3.2 Rechteck - Fächer

Die einfachste Variante eines *Billboard*s ist die eines Rechtecks mit einer Textur darauf. Doch dies hinterlässt beim Betrachten von der Seite einen flächigen Eindruck. Um dem entgegen zu wirken, kann man einen Fächer von Rechtecken erstellen, wo auf jedem Rechteck eine andere Perspektive des Objektes abgebildet ist (siehe Abbildung 3.15). Jedoch hat diese Darstellung den Nachteil, dass ein und dieselbe Geometrie mehrmals

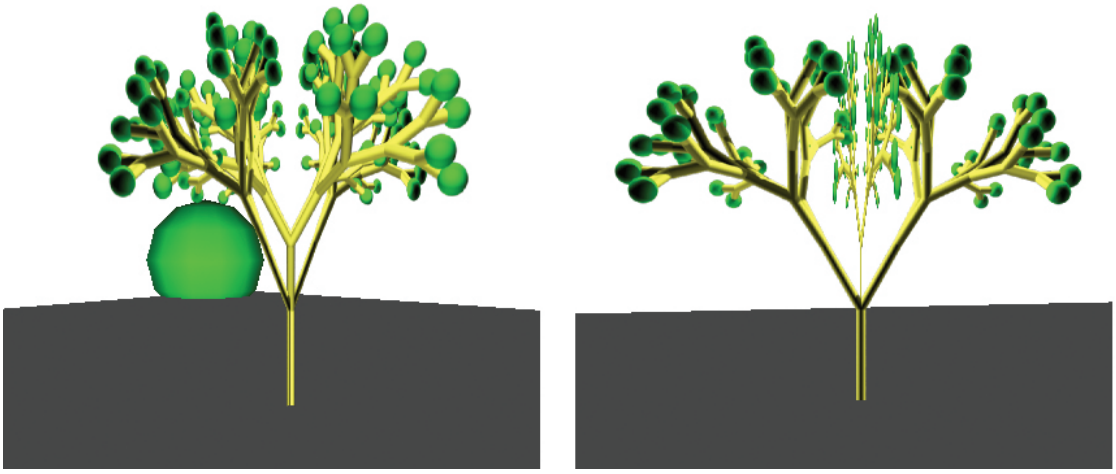

Abbildung 3.15: Einfache *Billboard*-Anwendung: Zwei miteinander verschränkte Rechtecke.

angezeigt wird. Abbildung 3.16 verdeutlicht das Problem. In dem Baum wurde eine Kugel rot markiert. Mit dem Fächer, aus acht Seiten-Ansichten des Baums bestehend, taucht die rote Kugel in der VRML-Ansicht mehrmals auf. Es entsteht auch der Eindruck, dass das angezeigte VRML-Modell wesentlich dichter ist als das Ausgangsmodell (erstes Bild der Abbildung 3.16).

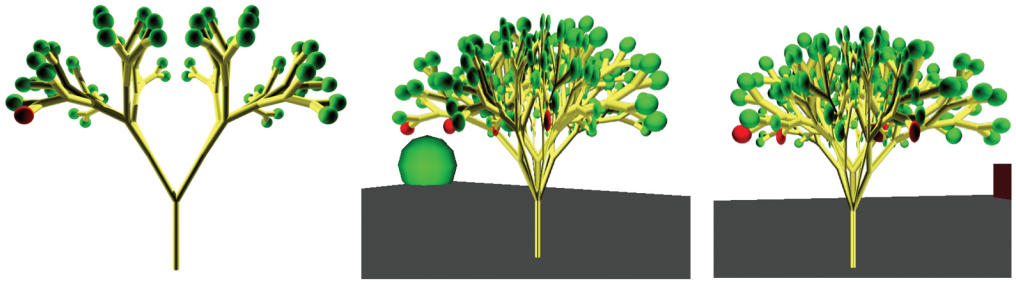

Abbildung 3.16: Das Bild ganz links zeigt eine einfache Darstellung des Baumes. Zu sehen ist die einzige rote Kugel. Die zwei Bilder rechts sind Aufnahmen aus einem VRML-Betrachter. Demonstriert wird die Fächer-Methode.

Bei Betrachtung eines Rechtecks stören die benachbarten Rechtecke. Das heißt, man sieht einen Teil des Objektes zu einem Zeitpunkt gleichzeitig aus mehreren Perspektiven. Damit dies nicht mehr geschieht, muss dafür gesorgt werden, dass die Geometrie des Objektes nicht mehrmals zu sehen ist.
Die Sichtachse der Kamera trifft immer senkrecht auf eine gedachte Projektionsebene. Diese wird um den Mittelpunkt rotiert und zwar mit dem gleichen Winkel-Betrag wie die

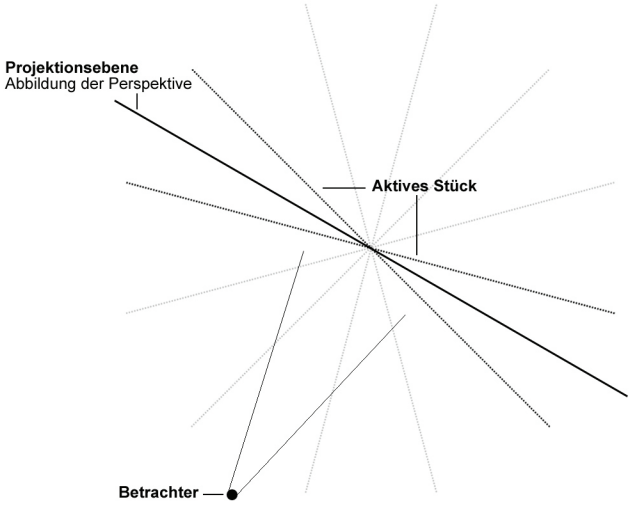


Abbildung 3.17: Tortenstück-Aufteilung einer 12-Seiten Darstellung.

Kamera. Dadurch verändert sich die relative Lage zwischen Kamera und Ebene nicht. Zusätzlich wird das Objekt in Sektoren eingeteilt, so dass es von oben wie „Tortenstückchen“ aussieht. Die Projektionsebene liegt durch die Einteilung immer in genau zwei Sektoren. Diese werden als aktiv bezeichnet. Beim Erstellen der jeweiligen Ansicht werden dann nur Elemente des Objektes sichtbar, die in einem solchen aktiven Sektor liegen. Alle anderen Elemente werden ausgeblendet. Das bedeutet, dass nur dann Elemente sichtbar sind, wenn sie in der Nähe der Projektionsebene liegen. Abbildung 3.17 zeigt die Methodik. Die Anzahl der „Tortenstücke“ ist abhängig von der Anzahl der zu erstellenden Seiten-Ansichten. Somit ist auch die Größe der Sektoren abhängig von der Anzahl der Seiten. Je mehr Seiten, desto kleiner die Sektoren.

Wie schon in Punkt 3.1.1 erklärt, müssen auch diese Rechtecke in der Mitte zerschnitten werden. Es wird zunächst ein Prototyp definiert. Dieser hat zwei Rechtecke (*IndexedFaceSet*s). Für die Darstellung wird dann für jede Seite eine Instanz des Prototypen erzeugt und in einem bestimmten Winkel um die vertikale Achse des lokalen Koordinatensystems gedreht. Somit entsteht ein Fächer von Rechtecken. In Abbildung 3.18 ist dies als Beispiel für eine Darstellung mit acht Seiten demonstriert.
Wie in der Abbildung 3.19 zu erkennen ist, ist jetzt deutlich weniger Geometrie auf einmal zu sehen. Jedoch tritt nun ein neues Problem auf: Es entstehen Sprünge, Freiräume zwischen benachbarter Geometrie. Dies hängt stark von der Art und Weise ab, wie das Ausblenden umgesetzt wurde. Darauf wird im Teil 5.3.6 genauer eingegangen.

### Zwischenfazit

Da dieses System aus statischen Rechtecken besteht, kann es vorkommen, dass man schräg auf eines der Rechtecke schaut. Dies hat den Nachteil, dass bei Betrachtung

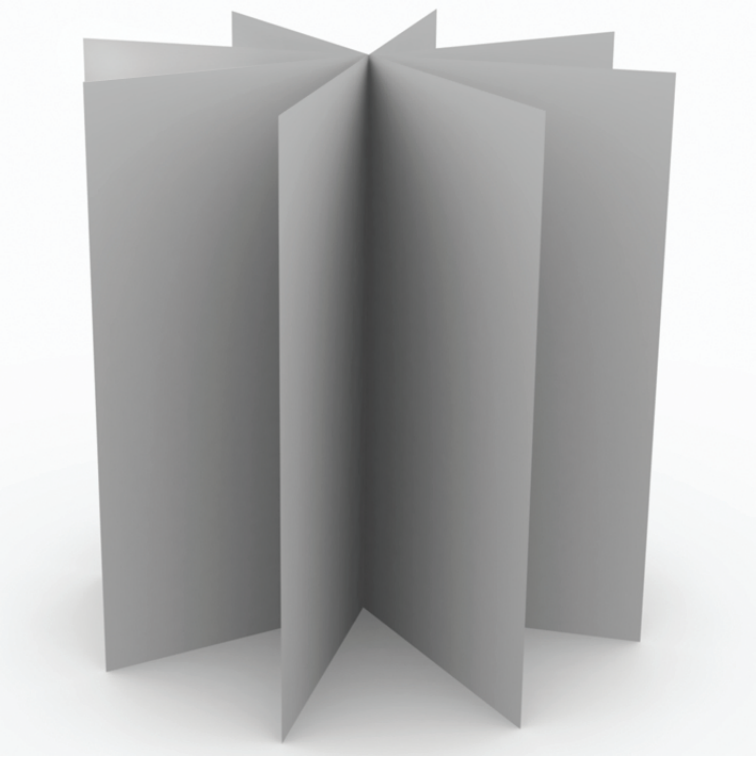

Abbildung 3.18: Nach dem Erzeugen der Instanzen entsteht ein Fächer von Rechtecken. Hier dargestellt ohne Textur und mit leichter Transparenz.

des Objekts in der Mitte solch perspektivisch verzerrte Rechtecke auftauchen und die Gesamt-Ansicht stören. Hinzu kommt, dass einige Elemente in der Luft hängen. Dies ist aus jeder Perspektive wahrzunehmen.

Dieses Prinzip kommt ohne eine gesonderte Berechnung von Positionsdaten und Winkeln aus, da es sich hier um ein statisches *Billboard*-System handelt. Dadurch ergeben sich gegenüber dem dynamischen *Billboard*-System drei Vorteile. Es braucht zum Zeitpunkt der Betrachtung weniger Rechenleistung, da die Positions-Berechnung entfällt, es kommt mit weniger Bildern aus, und die Plattform-Unabhängigkeit ist gewährleistet, da kein Script-Knoten benötigt wird.

### 3.3.3 Raster

Ähnlich wie bei der Fächer-Methode werden bei der Raster-Methode Elemente ein- und ausgeblendet. Auch hier wird das Objekt in Bereiche unterteilt, jedoch nicht in Stücke wie bei einer Torte, sondern in Scheiben. Dies geschieht von vorn nach hinten und von links nach rechts, so dass es von oben betrachtet wie ein Raster aussieht. Aus Sicht von

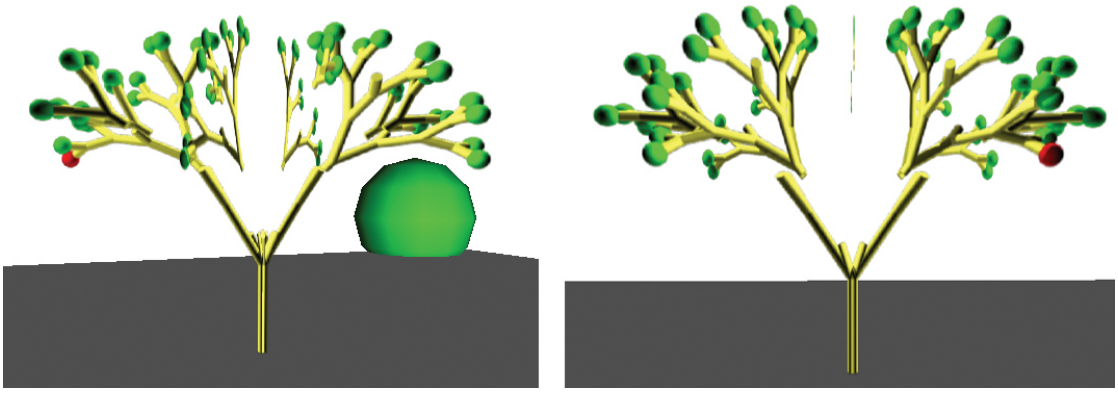

Abbildung 3.19: Fächer-Methode umgesetzt mit einem einfachen Baum und 8 Seiten-Ansichten. Deutlich zu erkennen: Die rote Kugel ist nur einmal zu sehen.

VRML geschieht die Einteilung nur entlang der x- und z-Achse[2].
Bei der Erstellung des Billboard-Systems werden somit vier Seiten des Objektes gerendert. Für jede Ansicht werden dann die einzelnen Scheiben berechnet, so dass sich eine Ansicht aus mehreren Scheiben zusammensetzt. Begonnen wird aus Sicht der Kamera mit der hintersten Scheibe des Objektes. Alle Objekte, die nicht in dieser Scheibe liegen, werden ausgeblendet. Ist das Rendern dieser Scheibe beendet, wird diese ausgeblendet, die nächste eingeblendet und gerendert. Dies wiederholt sich für eine Ansicht so lange, bis alle Scheiben durchschritten wurden. Danach wird die Kamera um 90° nach links gedreht und es werden wieder alle Scheiben berechnet. Bei der ersten Ansicht werden nur die Scheiben von hinten nach vorne betrachtet, bei der Ansicht zwei von rechts nach links, in Ansicht drei von vorne nach hinten, bei Ansicht vier von links nach rechts.

Die VRML-Datei besteht wieder aus einem Prototyp. Wie schon im Punkt 3.1.1 erwähnt, muss auch hier die Textur auf mehrere Rechtecke aufgeteilt werden. Deshalb besitzt der Prototyp mehrere *IndexedFaceSet*s, die Rechtecke darstellen. Die Anzahl dieser hängt von der Anzahl der Scheiben für eine Ansicht ab. Bei einem 3x3 Raster beispielsweise wird jede Scheibe drei mal geschnitten, wodurch vier Rechteck-Teile entstehen. So hat ein Prototyp eines 3x3 Rasters vier *IndexedFaceSet*s mit je einem Viertel der gesamten Textur einer Scheibe. Allgemeiner formuliert heißt das, dass für $n$ Scheiben $n+1$ Rechtecke pro Prototyp erstellt werden müssen.
Vom Beispiel des 3x3 Rasters ausgehend werden für jede Seite drei Instanzen des Prototyps erstellt. Sie bekommen einen Parameter für die Pfad-Angabe zur jeweiligen Textur,

[2]Die senkrechte Achse im VRML-Koordinaten-System entspricht der y-Achse. Somit bilden die x- und z-Achse eine Grundfläche im Raum.

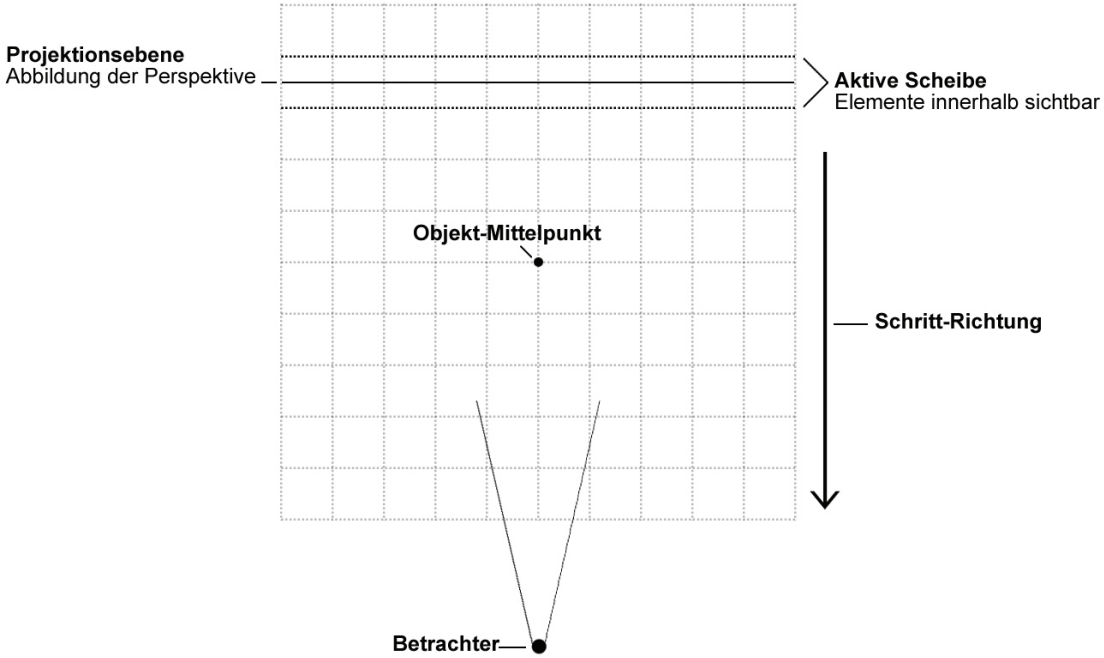


Abbildung 3.20: Einteilung eines 10x10 Rasters. In diesem Zustand wird gerade für die erste Ansicht auf das Objekt die 2. Scheibe von hinten berechnet. Die Kamera läuft nach allen Berechnungen der Scheiben für die jeweilige Ansicht im Uhrzeigersinn.

zusätzlich einen Parameter für die Verschiebung. Der Abstand der einzelnen Instanzen berechnet sich wie folgt:

$$Abstand = \frac{\text{Breite des Objektes}}{\text{Anzahl der Scheiben pro Sicht} + 1}$$

Die Breite des gesamten Objektes wird auf eins gesetzt. Die Höhe dazu ist relativ zu eins angegeben, so dass das Seiten-Verhältnis noch stimmt. Somit beträgt der Abstand zwischen den einzelnen Instanzen für das 3x3 Raster 0.25 VRML-Einheiten. Wie eine Anordnung der Rechtecke bei einem 3x3-Raster aussieht, zeigt Abbildung 3.21.

Für die nächsten Sichten werden *Transform*-Knoten erstellt und die zur jeweiligen Sicht gehörenden Instanzen um 90°, 180° bzw. 270° (also -90°) gedreht. Das Ergebnis ist in Abbildung 3.22 demonstriert.

### Zwischenfazit

Auch diese Methode kommt ohne zusätzliche Berechnungen bei der Betrachtung des Objekts aus, doch erhält man bei einer Raster-Darstellung relativ schnell eine hohe Anzahl an *IndexedFaceSet*s. Ein 10x10 Raster mit einer vier-Seiten-Ansicht hat bereits 440

*IndexedFaceSet*s. Die Anzahl der zu ladenden Bilder liegt hier bei 40, da eine Textur einer Scheibe entspricht. Der subjektive Eindruck gegenüber der Fächer-Methode ist besser, da die Anzahl der Sprünge zwischen den Elementen geringer sind. Jedoch scheinen viele Elemente einer anderen Perspektive die Sicht auf das Objekt zu stören.

Die drei gezeigten Methoden wurden bis jetzt nur mit einem einfachen Baum demonstriert. Dieser Baum hatte die Eigenschaft, dass man in das Innere der Baumkrone schauen konnte. Es bleibt daher abzuwarten, wie die Methoden auf komplexere Modelle angewandt funktionieren, bei denen die Struktur feiner ausgeprägt ist. Dies wird in Abschnitt 5.3.6 besprochen. Danach kann eine konkretere Aussage über die jeweilige Anwendbarkeit bei Visualisierungen getroffen werden.

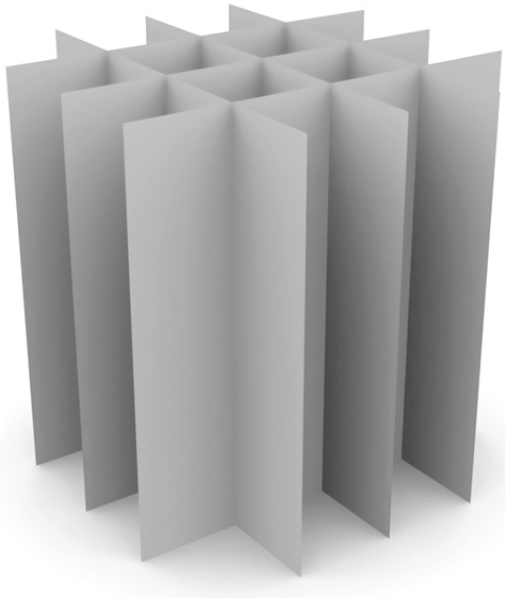

Abbildung 3.21: Ein 3x3 Raster ohne Textur.

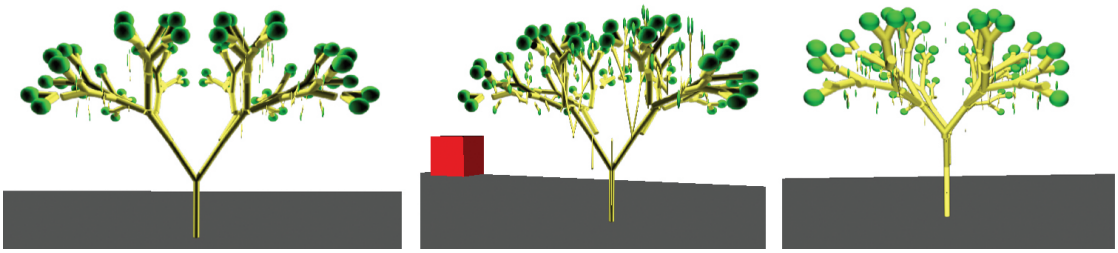

Abbildung 3.22: Eine 10x10 Raster-Darstellung.

## 3.4 Stochastische Billboards

Die in diesem Kapitel vorgestellten *Billboard*-Systeme basieren alle auf einer vergleichsweise einfachen Anordnung von Rechtecken. Einen anderen Ansatz verfolgt das Prinzip des stochastischen *Billboard*-Systems. In dem Artikel „*Billboard Clouds for Extreme Model Simplification*“ [3] wurde ein solches stochastisches Verfahren umgesetzt.
Der Ansatz ist, dass eine Sammlung von *Billboard*s erstellt wird, die das Objekt abhängig von einem angegebenen Fehler-Wert repräsentiert. Die *Billboard*s bestehen dabei aus Ebenen. Nach deren Generierung werden die brauchbaren Ebenen ausgesucht. Brauchbar sind zum Beispiel solche Ebenen, die fast tangential zur darzustellenden Oberfläche verlaufen. Diese werden bei der Auswahl bevorzugt [3, Seite 689].

In diesem Verfahren wird das Optimierungsproblem durch eine Fehler- und Kostenfunktion definiert. Es wird ein maximal tolerierbarer Fehler angegeben der mit einem Minimum an Kosten einzuhalten ist [3, Seite 689].
Dabei wird für jede Ebene ermittelt, wie gut sie den darzustellenden Bereich des Modells repräsentiert. Dies hängt zum Einen von der Größe des Bereiches, den die Ebene darstellt, ab, zum Anderen ist die Qualität der Abdeckung entscheidend dafür, ob eine Ebene den Bereich des Modells abdecken soll oder nicht. Die ermittelten Ebenen bilden einen Ebenen-Raum.

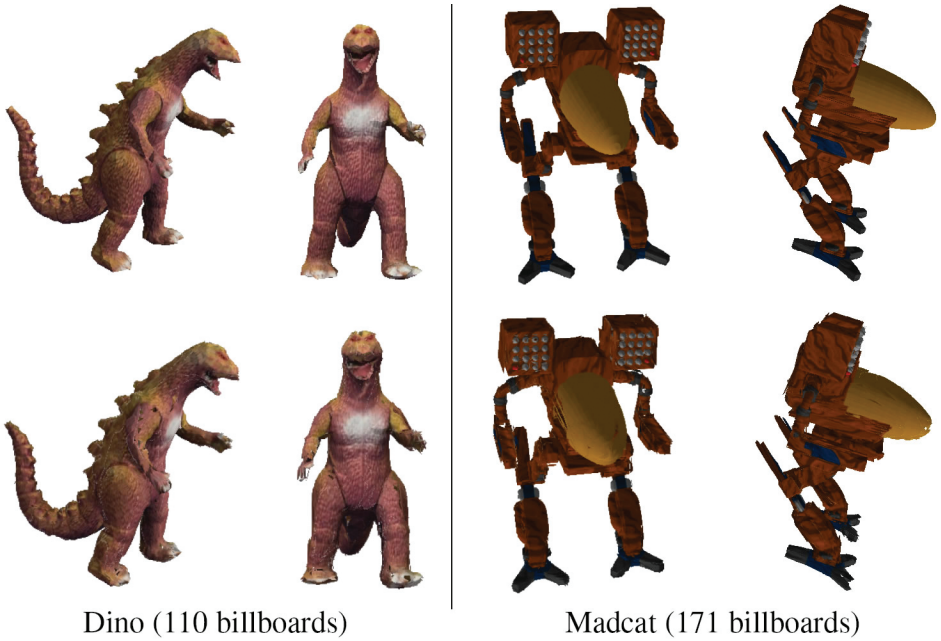


Abbildung 3.23: Beispiel aus „*Billboard Clouds for Extreme Model Simplification*“ [3]. In der oberen Reihe sind die 3D-Modelle abgebildet. In der Reihe darunter die entsprechenden vereinfacht dargestellten Modelle.

Ein 3D-Modell lässt sich aus Polygonen darstellen. Ein Polygon besteht aus mindestens 3 unabhängigen Vertices. Diese bilden die Ecken eines Polygons. Durch Angabe eines geometrischen Fehler-Wertes werden von allen Vertices *gültige* Ebenen ermittelt. Gültig bedeutet, dass der Abstand zwischen den zu repräsentierenden Vertices und der Ebene kleiner sein muss als der Betrag des geometrischen Fehler-Wertes.
Da es möglich ist, dass für eine Oberfläche mehrere Ebenen eine gültige Repräsentation bilden, werden die Ebenen nun gewichtet. Die Gewichtung zielt darauf ab, Ebenen mit einer großen Polygonabdeckung zu bevorzugen. Neben der Gewichtung großer Abdeckungen wird auch eine Art *Bestrafung* eingeführt. Das heißt, dass Ebenen benachteiligt werden, wenn sie Polygone in der direkten Nachbarschaft auslassen.
Aus den existierenden Ebenen sollen nun diejenigen ausgewählt werden, die das Modell am besten repräsentieren. Dabei gilt, dass so wenig *Billboard*s wie möglich erstellt werden. Denn jedes weitere *Billboard* bedeutet, dass eine neue Textur erzeugt werden muss. Das verringert die Effizienz des Verfahrens.
In einem letzten Schritt werden nun die entsprechenden Texturen erstellt. Es werden die Teile des Modells entsprechend auf die ausgewählten Ebenen projiziert und anschließend gerendert. Von der Textur nicht abgedeckte Bereiche der Ebenen werden auf der Textur als transparent dargestellt.

Die Methode arbeitet auf Polygon-Basis und benötigt keine Informationen darüber, wie die Polygone miteinander verbunden sind. Jedoch müssen die Polygone als Dreiecke existieren, was dazu führt, dass die Modelle trianguliert werden müssen.
Für das Einbetten der *Billboard*s in eine VRML-Szene ist keine komplizierte Umrechnung nötig. Die erstellten Ebenen können durch Rechtecke in VRML adäquat ersetzt werden.

# 4 Erstellung mit Hilfe eines 3D-Editors

## 4.1 Prinzip einer Modellierung an einem kleinen Beispiel

Die zweite zu untersuchende Methode, um ein Baummodell zu erzeugen, ist die Nutzung eines konventionellen 3D-Modellierungsprogrammes. Die einzelnen Schritte dieser Methode werden mit dem Programm 3D Studio Max 2008[1] der Firma Autodesk demonstriert. Das Modellierungsprogramm ist als Demo direkt von der Internetpräsenz des Herstellers herunterladbar und kann 30 Tage ohne Einschränkungen genutzt werden.
Am Beispiel der Erle soll nun gezeigt werden, auf welchem Weg ein 3D-Baummodell erstellt werden kann. Dazu wurde die Modellierung in vier Teilschritte eingeteilt, die wie folgt lauten:

- Stämme
- Haupt-Äste
- Zweige
- Blätter

Als Grundlage der Modellierung dienen Aufnahmen aus verschiedenen Perspektiven sowie zu verschiedenen Jahreszeiten.
Für die Modellierung der drei Stämme und der Hauptäste empfiehlt es sich, einen Arbeitshintergrund zu erstellen. Dieser Hintergrund besteht aus einem Rechteck mit Textur, welche ein Foto der Erle ist. Vor dem Rechteck kann dann die Erle modelliert, mit dem Original verglichen, und Fehler bei der Modellierung schneller entdeckt und korrigiert werden.
Nachdem die Arbeitsumgebung vorbereitet wurde, kann nun mit der Erstellung des Modells begonnen werden. Dazu werden *NURBS*[2] genutzt. Wie diese in 3D Studio Max verwendet werden, wird zunächst demonstriert.

[1] Fähigkeiten im Umgang mit dieser werden vorausgesetzt. Im Zusammenhang mit der Nutzung dieses Programms fallen einige Begriffe, deren Funktion in der programminternen Hilfe dokumentiert sind. Daher werden auftretende Begriffe nur kurz erläutert.

[2] Heißt *Non-Uniform Rational B-Splines* (zu deutsch Nicht-uniforme rationale B-Splines). Mit Hilfe dieser Funktionen werden Kurven oder Flächen beschrieben. Dazu sei das Buch *The NURBS Book* [17] von Piegl und Tiller empfohlen.

### 4.1.1 Modellierung mit NURBS

Es werden drei Kreis-Formen erzeugt und im Raum verteilt. Die jeweiligen Größen der Kreise spielen zunächst keine Rolle, da sie später noch anpassbar sind. Als nächstes werden die Kreis-Formen in *NURBS*-Kurven umgewandelt (Abbildung 4.1 Bild 1). Anschließend wird der oberste Kreis ausgewählt. Es erscheint ein Menü für das Erstellen und Modifizieren von *NURBS*-Kurven und -Oberflächen.

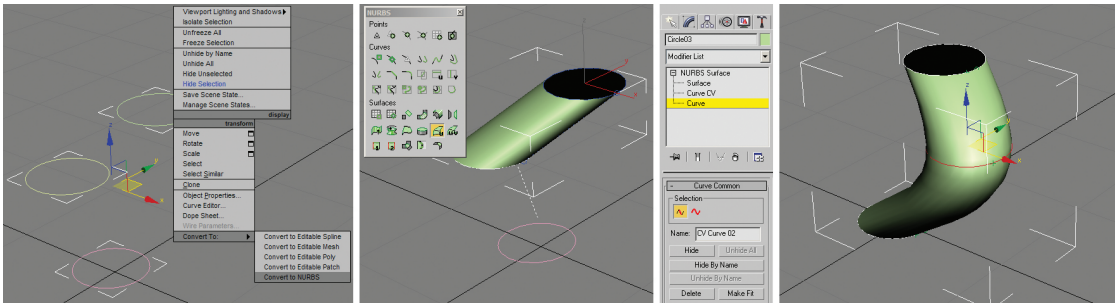

Abbildung 4.1: Kreis-Formen werden in *NURBS*-Kurven umgewandelt und anschließend mit einer *U-LOFT-NURBS*-Oberfläche überzogen. Nachdem eine Oberfläche erstellt wurde, können die *NURBS*-Kurven noch modifiziert werden.

Mit einer *U-LOFT*-Oberfläche wird über alle drei *NURBS*-Kurven eine Oberfläche gezeichnet (Abbildung 4.1 Bild 2). Die zu berücksichtigenden *NURBS*-Kurven werden ausgewählt. Wichtig ist, dass mit dem Zeichnen bei der obersten *NURBS*-Kurve begonnen wird. Beginnt man unten, ist die Vorderseite der Oberfläche nach innen gerichtet.

Ist die Oberfläche gezeichnet, besteht dennoch die Möglichkeit, einzelne *NURBS*-Kurven zu modifizieren. Dazu wird die Oberfläche durch Anklicken ausgewählt. Rechts erscheinen die Eigenschaften des Objektes. Oben ist eine Auflistung der Elemente, die das Objekt definieren, zu finden (Abbildung 4.1 Bild 3). Wählt man das Element *Curve*, aus können die einzelnen Kurven des Objektes modifiziert werden. Zum Beispiel kann die Position der Kurve verändert werden (Abbildung 4.1 Bild 4). Es ist auch möglich, die Größe der Kurve mit dem Transformationswerkzeug anzupassen.

### 4.1.2 Modellierung

#### Stämme

Für das Erstellen des Stammes bietet sich die Front-Ansicht auf die Arbeitsfläche an. Somit ist die Bildvorlage nicht durch schräges Betrachten verzerrt. Vor der Bildvorlage können nun die Kreis-Formen erzeugt und positioniert werden. Die Anzahl der Kreis-Formen kann beliebig gewählt werden.



Abbildung 4.2: Mit *NURBS* modellierte Stämme. Wurde alles richtig gemacht, sind bei der Front-Ansicht die drei Stämme relativ deckungsgleich mit der Bildvorlage, wie es auf dem dritten Bild zu sehen ist.

Sind alle Kreis-Formen für einen Stamm erzeugt, werden sie jeweils entlang eines Stammes verteilt. Die Größe des Kreises kann nun an die Dicke des Stammes im Bild angepasst werden. Einige Arbeitsschritte sind in Abbildung 4.2 dargestellt.

### Haupt-Äste

Nach dem gleichen Verfahren wie bei der Modellierung der Stämme wird nun ein Haupt-Ast modelliert. Dieser wird beliebig oft kopiert und positioniert. Damit die Haupt-Äste unterschiedlich aussehen, werden sie mit dem Transformationswerkzeug in der Größe verändert. Da es nicht um die exakte Nachbildung der Erle geht, reicht es aus, den Haupt-Ast einmal zu modellieren, zu kopieren und zu modifizieren. Abbildung 4.3 zeigt einige Zwischenschritte.

### Zweige

Nach dem Erstellen der Haupt-Äste kann das Modell mit einer Textur versehen werden. Darauf wird an dieser Stelle jedoch nicht weiter eingegangen.

Die Haupt-Äste des Modells zu erstellen, war sehr aufwändig. Würde man nun alle feineren Zweige auch auf diese Weise modellieren, wäre der Aufwand um einiges höher. Um dies zu vermeiden, kommt das Werkzeug *Hair and Fur* (Haar und Fell) zum Einsatz. Es ist ein Modifikator, der auf 3D-Objekte angewendet wird. Dieser Modifikator wird normalerweise dazu genutzt, Haar beziehungsweise Fell zu erstellen.
Anstatt Haare zu generieren, können auch beliebige andere Objekte eingesetzt werden. Dazu wird zunächst eine kleine, sehr einfache Verzweigungsstruktur erstellt. Diese be-

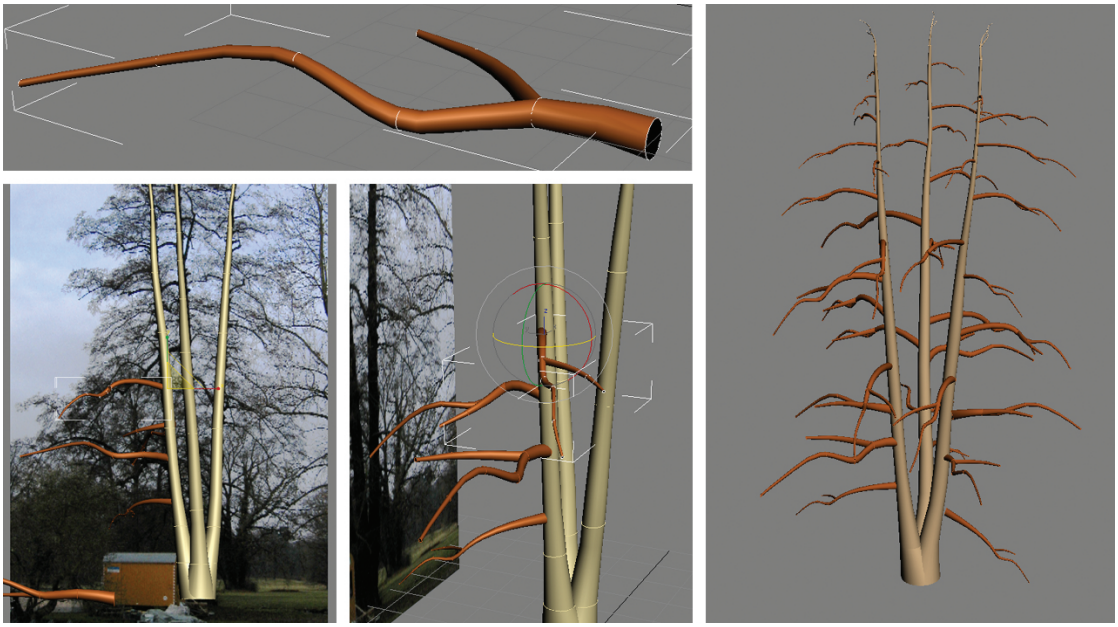

Abbildung 4.3: Oben links ist ein einzelner Haupt-Ast ebenfalls mit *NURBS* modelliert. Diese werden an den einzelnen Stämmen verteilt. Je höher der Haupt-Ast, desto mehr wurde er verkleinert. Ganz rechts ist das fertige Ergebnis.

steht aus drei Zylindern (siehe Abbildung 4.4 oben links). Mit Angabe des Objektes wird nun über das gesamte 3D-Modell dieses Objekt kopiert und verteilt. Das Ergebnis ist auf Abbildung 4.4 zu sehen.

Alle kleinen Äste werden zu einem Objekt zusammengefasst, da dies für den folgenden Schritt benötigt wird. Damit ist die Modellierung der Ast-Struktur abgeschlossen, so dass nun ein Winter-Modell der Erle erstellt wurde.

### Blätter

In diesem Schritt wird nun die Erle „belaubt“, das heißt, es werden Blätter erzeugt und auf dem 3D-Modell verteilt. Zunächst wird dazu ein Rechteck erstellt und ein Material daraufgelegt. Damit die Blattform nicht modelliert werden muss, wird einfach mit einem Schwarz-weiß-Bild festgelegt, an welchen Stellen das Rechteck sichtbar sein soll. Dadurch ist die „Form“ des Blattes gegeben.
Die Blätter werden mit Hilfe des Werkzeugs *Scatter* auf dem 3D-Modell verteilt. Aus Gründen der Speicherauslastung und Rechenzeit ist es nicht zu empfehlen, auf die kleinen Zweige erneut den *Hair and Fur*-Modifikator anzuwenden. Zur Blattverteilung eignet sich das *Scatter*-Werkzeug besser.
Zunächst wird ein Objekt ausgewählt, das vervielfältigt werden soll. In diesem Fall ist das das Rechteck. Als nächstes wird das Objekt bestimmt, auf das die Verteilung angewendet



Abbildung 4.4: Oben links ist das Objekt, das die Verzweigung bilden soll. Diese wird mit dem *Hair and Fur*-Modifikator über das gesamte Modell verteilt. Durch Parameter kann die Erscheinung der Verzweigungen angepasst werden.

werden soll. Als letztes wird die Anzahl der zu erstellenden Vervielfältigungen angegeben.

Da Blätter nur an den dünnen Zweigen hängen sollen, wird die Verteilung auch nur auf die kleinen Äste angewendet. Dieser Schritt schließt die Modellierung der Erle mit einem konventionellen 3D-Modellierungsprogramm ab.

### 4.1.3 Ergebnis

Abbildung 4.5 zeigt das Ergebnis der Modellierung. Die Modellierung dauert circa zwei Tage. Das Modell im belaubten Zustand bestand aus circa 500.000 Polygonen. Mit Blättern waren es circa 900.000 Polygone.
Auch wenn dies nur ein kleines Beispiel der konventionellen 3D-Modellierung darstellt, lässt sich doch sagen, dass das Ergebnis dem Original im Ansatz ähnelt. Ob diese Methode das beste 3D-Modell der Erle liefert, bleibt abzuwarten.



Abbildung 4.5: Links das Original. In der Mitte ist das „Winter“-Modell der Erle zu sehen. Rechts sieht man die Erle im belaubten Zustand mit ungefähr 16.000 Blättern.

## 4.2 Alternative Modellierung

Ein Alternative zur Modellierung „per Hand“ bietet eine Methode, die auf der SIGGRAPH 2007 von Neubert et al. [16] vorgestellt wurde. Es handelt sich hierbei um die Erstellung eines 3D-Modells von einem Baum auf Grundlage von Fotografien. Die Methode zielt darauf ab, dass die Erstellung so wenig Eingriff durch den Benutzer bedarf wie möglich.
Durch diese Methode gelingt es sehr schnell, einen Baum zu modellieren. Da der Zeitfaktor für die Modellierung des gesamten Parks auch eine wichtige Rolle spielt, soll diese Methode an dieser Stelle näher besprochen werden.

### Vorbereitung

Der zu erstellende Baum liegt als mehrseitige Bildvorlage vor. Zunächst werden von den Bildvorlagen sogenannte *trimaps* [8, Seite 186] erstellt. Eine *trimap* dient dazu, ein Bild in drei Bereiche einzuteilen: Vordergrund, Hintergrund und der Bereich, in dem Pixel aus dem Vorder- und Hintergrund vermischt sind. Das Erstellen geschieht durch eine Farbauswahl, die jeweils den Hintergrund beziehungsweise den Vordergrund am besten repräsentiert. Für diese Methode ist jedoch keine genaue Freistellung nötig, daher ist dieser Schritt auch einfach gehalten.
Das Ergebnis der Farbauswahl ist ein Graustufen-Bild, wie das in Abbildung 4.6 auf dem rechten Bild. Es repräsentiert die Alpha-Werte des Vordergrund-Objektes. Schwarz bedeutet sichtbar, weiß transparent. Diese Informationen dienen gleichzeitig zur Abschätzung der Dichte-Verteilung des Baumes. Je dunkler ein Pixel auf dem Graustufen-Bild

ist, desto dichter ist der Bereich auf dem Original-Bild. Weiß besitzt die geringste Dichte.

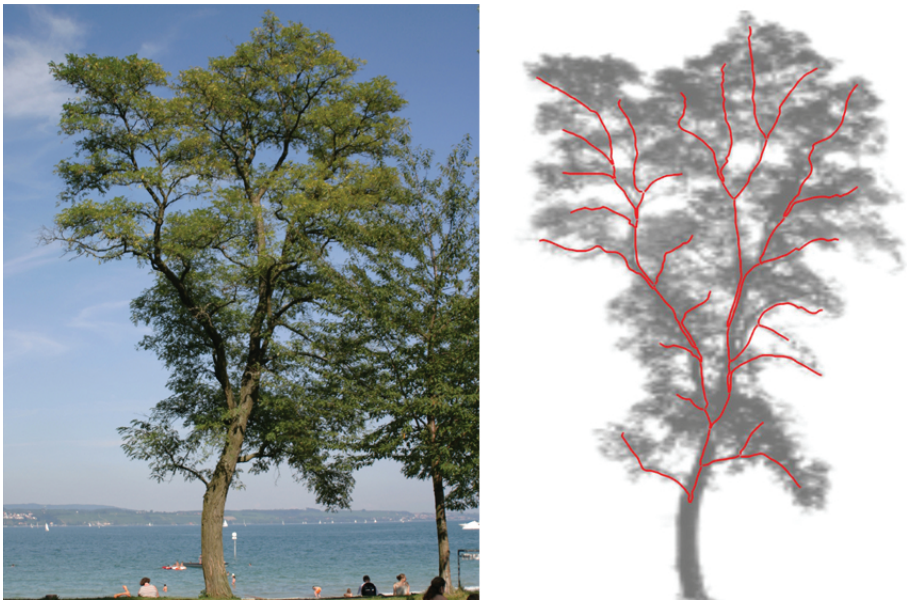

Abbildung 4.6: Links das Original-Bild. Rechts die Dichte-Verteilung sowie der *Attraktor*-Graph (rot gefärbt) [16, Seite 3].

Als nächstes wird die Ast-Struktur – das Haupt-Skelett – für jedes Graustufen-Bild definiert. Dazu wird ein Zielpunkt auf dem *Dichte-Bild* angegeben. Zusätzlich werden für alle Verzweigungen im Baum Startpunkte gesetzt. Die Punkte dienen als Knoten in einem Graphen. Für jeden Startpunkt wird nun auf Grundlage der Bild-Vorlage ein Weg zum Zielpunkt gesucht und im Graphen gespeichert. Das Ergebnis ist ein *Attraktor*-Graph [16, Seite 3]. Der *Attraktor*-Graph dient zu einem späteren Zeitpunkt als Orientierungshilfe. Das Setzen der Punkte kann automatisch oder manuell erfolgen.

Die Freistellung sowie das Zeichnen des *Attraktor*-Graphen geschieht für jede der Bild-Vorlagen, so dass am Ende dieses Schrittes genauso viele Graustufen-Bilder und *Attraktor*-Graphen existieren, wie es Seiten-Aufnahmen von dem betroffenen Baum gibt.

### Partikelsimulation

Die Abschätzung der Dichte-Verteilung liegt zunächst als 2D-Repräsentation vor. Durch eine Rückprojektionstechnik wird ein 3D-Gitter, aus kleinen Boxen bestehend (Voxel), mit der Dichte-Verteilung erstellt. Somit gibt es eine Abschätzung der Dichte-Verteilung des Baumes im Raum.
Da die *Attraktor*-Graphen zunächst als 2D-Repräsentationen der jeweiligen Seite des Baumes vorliegen, müssen diese zu einem 3D-Graphen zusammengeführt werden. Dieser bildet danach das Haupt-Skelett des Baumes.
Mit der Information über die Verteilung der Dichte im Raum werden Partikel zufällig



Abbildung 4.7: Links das Ergebnis der Partikel-Simulation ohne *Attraktor*-Graphen, rechts mit [16, Seite 5].

im Raum verteilt. Wie viele Partikel in ein Voxel gesetzt werden, hängt von der Dichte des Voxels ab. Nun wird ein Partikelfluss simuliert. Dabei streben Partikel dazu, den *Attraktor*-Graphen zu erreichen und von dort aus nach unten zu fließen. Beim Fließen versuchen die Partikel, sich mit benachbarten Partikeln zu vereinigen und gemeinsam den Weg fortzusetzen. Das Ergebnis ist auf dem rechten Bild der Abbildung 4.7 demonstriert. Was geschieht, wenn die Partikel-Simulation ohne den *Attraktor*-Graphen durchgeführt wird, zeigt das linke Bild der Abbildung 4.7.

### Erzeugung der Geometrie

Für jeden Pfad im Graphen werden nun Segmente geformt. Die Formung hängt von der Anzahl der vereinigten Partikel ab, die den Pfad entlang geflossen sind. Je mehr Partikel, desto dicker das Segment. Die Segmente werden danach mit einer Textur überzogen. Das Ergebnis ist in Abbildung 4.8 gezeigt.
Das Blattwerk wird in einem letzten Schritt, durch den Benutzer kontrollierbar, zusammen mit kleinen Zweigen erstellt und in das Modell integriert. Die Positionierung der Blätter hängt von der Dichte-Verteilung ab. Ein Beispiel eines Endergebnisses zeigt Abbildung 4.9.



Abbildung 4.8: Erzeugte Baumgeometrie mit Textur [16, Seite 6].



Abbildung 4.9: Fertiges 3D-Modell [16, Seite 7].

# 5 Erstellung mit Hilfe von Grammatiken

## 5.1 Einführung Lindenmayer-Systeme, relationale Wachstumsgrammatiken

### Lindenmayer-System (L-System)

An dieser Stelle soll kurz der Begriff der „Lindenmayer-Systeme“ eingeführt werden. Dazu gibt es eine Vielzahl an Literatur, die dieses Gebiet genauer erläutern. Als Einstieg in dieses Thema ist das Buch *The Algorithmic Beauty Of Plants* von *Prusinkiewicz* und *Lindenmayer* [19] sehr zu empfehlen. Einen kurzen Überblick bietet auch die Diplom-Arbeit von Michael Henke [7, Seite 9 – 17].

Der Begriff „Lindenmayer-System“, oder kurz „L-System“, geht auf den Namen des Pioniers dieses Gebietes zurück: Aristid Lindenmayer (1925-1989). Auf der Suche nach einer Möglichkeit, einfache Zellorganismen zu beschreiben, führte er diese formale Sprache 1968 ein [7, Seite 9]. Im Laufe der Jahre wurden immer neue Interpretationen vorgestellt, die es ermöglichten, L-Systeme zur Pflanzen-Modellierung zu nutzen. Das Grundkonzept ist dabei das der Texterzeugung [19, Seite 1].
Bei der Texterzeugungstechnik wird aus einem initialen Zustand mit Hilfe von Ersetzungsregeln, oder auch Produktionen, in mehreren Schritten (Iterationen) ein neues Objekt erzeugt. Aus einem Zeichenvorrat wird eine Zeichenfolge für den initialen Zustand sowie die Zeichen (Symbole), die durch andere Symbole in der Produktion ersetzt werden, vorgegeben. Nach jeder Ersetzung kann somit eine neue Zeichenfolge entstehen, welche in der Regel dann komplexer ist als in dem Schritt zuvor. Diese Form eines Regelsystems nennt man im Compilerbau Grammatik [1, Seite 32 – 33]. Der Unterschied zum L-System liegt jedoch darin, dass die Anwendung der Ersetzungsregeln, also das Austauschen der einzelnen Zeichen, beim L-System parallel erfolgt [19, Seite 3].

Mit dem Zeichenvorrat *F, +, –* soll die Texterzeugung demonstriert werden. Der initiale Zustand, das Axiom, ist durch das Zeichen $F$ gegeben. Die Produktion lautet, dass ein $F$ durch die Zeichenfolge $F + F - F + F$ ersetzt werden soll. Eine Produktion wird mit einem „$\rightarrow$“ gekennzeichnet. Alles, was auf der linken Seite steht, soll durch die rechte ersetzt werden. Somit ergibt sich folgende Notation:

- *Axiom*: $F$
- *Produktion*: $F \rightarrow F + F - F + F$

So ergibt sich Schritt für Schritt eine neue Zeichenfolge:

- *Schritt 0*: $F$
- *Schritt 1*: $F + F - F + F$
- *Schritt 2*: $F + F - F + F + F + F - F + F - F + F - F + F + F + F - F + F$

Um eine erzeugte Zeichenfolge auch graphisch darstellen zu können, muss es eine Interpretation geben, die die Zeichen mit entsprechenden graphischen Symbolen verknüpft. Diese Möglichkeit bietet die sogenannte *Turtle*-Interpretation. Dabei kann man sich die Turtle als eine Art Zeiger mit der Möglichkeit zu zeichnen vorstellen. Die Zeichen einer Zeichenfolge sollen dabei eine graphische Form oder einfach nur einen Bewegungsbefehl darstellen. In *The Algorithmic Beauty Of Plants* [19, Seite 7] werden die Symbole wie folgt interpretiert:

- $F$: Gehe einen Schritt vorwärts und zeichne eine Linie
- $f$: Gehe einen Schritt vorwärts ohne eine Linie zu zeichnen
- $+$: Rotiere von der aktuellen Position aus nach links um 90°
- $-$: Rotiere von der aktuellen Position aus nach rechts um 90°

Desweiteren gilt, dass das erste $F$ nach oben zeigt und die Länge der vom $F$ erzeugten Linie konstant ist, sofern nicht anders angegeben.

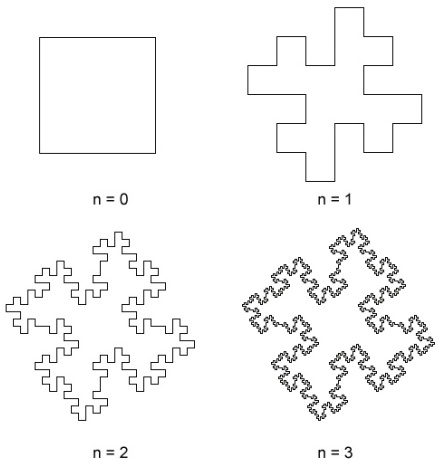


Abbildung 5.1: Beispiel einer *Turtle*-Interpretation: *quadratic Koch island* aus *The fractal geometry of nature* [15, Seite 51].

Ein Beispiel für eine *Turtle*-Interpretation ist die quadratische *Koch*-Insel (*quadratic Koch island*) aus Mandelbrot's Buch *The fractal geometry of nature* [15, Seite 51]:

- *Axiom*: F – F – F – F
- *Produktion*: F → F – F + F + F F – F – F + F

Die jeweiligen Ergebnisse der Iterationschritte ($n$) werden in Abbildung 5.1 demonstriert. Um pflanzenähnliche Strukturen zu erstellen, muss neben dem Zeichnen von Linienzügen auch eine Verzweigungsstruktur erstellbar sein. Dies ist mit der bis hier vorgestellten Notation nicht möglich. Der Zeichenvorrat wird daher um die zwei Zeichen „[“ und „]“ erweitert. Das Zeichen „[“ sorgt dafür, dass der aktuelle Zustand der *Turtle* auf einen Stapel, englisch *stack*, gelegt wird. Der Zustand besteht zu diesem Zeitpunkt aus der Orientierung und Position im Raum. Dies wird in 5.2 noch durch weitere Attribute (Farbe, Größe, usw) ergänzt. Die *Turtle* arbeitet die Zeichen nach dem Öffnen der Klammer wie gewohnt ab. Erreicht sie eine schließende Klammer („]“), wird das oberste Element vom *stack* geholt. Dieses Element beinhaltet den Zustand der *Turtle* vor dem Öffnen der Klammer, also die Position und Orientierung vor der Verzweigung [19, Seite 24].

#### Relationale Wachstumsgrammatiken

L-Systeme nutzen vornehmlich das Ersetzen von Zeichenfolgen (Strings). Dies ermöglicht das Modellieren von Wachstum und Architektur einer Pflanze durch Vorgabe von Regeln. Strukturelle Aspekte einer Pflanze sind einfach implementiert. Um interne Prozesse zu repräsentieren, ist ein „Ausgliedern“ in externe Programme notwendig [9, Seite 57]. Obwohl die doppelte String-Struktur eines DNA-Strangs gerade für das Beschreiben in einem Texterzeugungs-Formalismus geeignet scheint, wurde dies nie in einem L-System dargestellt [10, Seite 625]. Dies ist nach wie vor ein Problem, welches Prusinkiewicz et al. [18, Seite 20] einräumt.

Nach Kurth [13, Seite 36] sollen Modelle von Pflanzen in den Agrarwissenschaften möglichst aussagekräftig sein, und dies auch unter veränderten Bedingungen wie Umwelt, Anbauform und so weiter. Diese Modelle sind recht komplex und laut Kurth in drei Extrem-Typen einzuteilen [13, Seite 36]:

- Funktionsmodell: Pflanzen-interne Prozess-Modellierung. Teilweise in Wechselwirkung mit Umwelt.
- Strukturmodell: Modellierung von Triebbildung und der morphologischen Struktur der Pflanze.
- Statistisches Modell: Aggregation von empirischen Daten.

Um die Anforderungen der Modelle zu gewährleisten, sollte von allen drei Modell-Typen etwas enthalten sein. Dies spiegelt sich im „Struktur-Funktions-Modell“ (englisch: *functional structural plant model (FSPM)*) wieder. Hier wird versucht, interne Funktionen räumlich auf Substrukturen der Pflanze (Organe) zu verteilen und zu simulieren. Dabei wird die Wechselwirkung mit anderen Organen beachtet.
Diese FSPM sind in der Regel komplexe Programme, die in den heute gängigen Programmiersprachen geschrieben sind, und verfolgen den objektorientierten Ansatz. Durch die

Komplexität der Programme ist ihre Spezifikation auch sehr unübersichtlich, da sie oft aus vielen Zeilen Code besteht. Für Menschen wie Biologen oder Agrarwissenschaftler mit wenig Programmiererfahrung dürfte dies sehr abschreckend wirken.

Der Lehrstuhl Grafische Systeme an der BTU Cottbus hat deshalb eine regelbasierte „Metasprache“ eingeführt. Sie führt ebenfalls den objektorientierten Ansatz fort und ergänzt die Sprache, um die Bedürfnisse der Biologen zu erfüllen. Die Metasprache erweitert die regelbasierte Ersetzung von Substrukturen in Zeichenketten, indem sie auf Graphen (aus mathematischer Sicht; bestehend aus Kanten und Knoten) angewendet wird. Diese Graphen werden auch relationale Strukturen genannt, da sie verschiedene Relationen auf einer Knotenmenge besitzen. Es handelt sich hierbei um gerichtete Graphen, deren Knoten mit unterschiedlichen Kantentypen verbunden sind. Wachstumsgrammatiken, die darauf angewendet werden, heißen demzufolge relationale Wachstums-Grammatiken (englisch: *relational growth grammar (RGG)*).

Im Gegensatz zu L-Systemen, die praktisch linearisierte Bäume transformieren, werden Bäume in *RGG* als echte Graphen dargestellt. Somit sind L-Systeme ein Spezialfall von *RGG*. Dadurch, dass Graphen die Basis bilden, entfällt auch die Trennung zwischen Graph und Darstellung, denn der Graph kann auch als Szene-Graph interpretiert werden. Eine einfache gerichtete Verbindung von Knoten A nach Knoten B hat den Kanten-Typ *successor*, zu deutsch Nachfolger oder Nachfahre. Die im L-System dargestellte Verzweigung „[“ und „]“ wird im Graphen als Kanten-Typ *branch* (Verzweigung) repräsentiert. Die Knoten im Graphen werden als Objekte betrachtet und können zum Beispiel Instanzen von Java-Klassen sein.

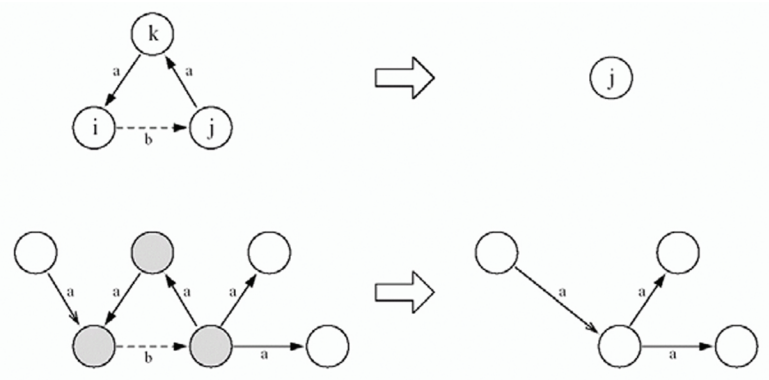


Abbildung 5.2: Beispiel einer Graphersetzung: Oben die Regel, die drei Knoten durch einen anderen Knoten ersetzt. Unten ist die Anwendung der Regel auf einen Graphen zu sehen. (Bsp. aus [13, Seite 39]).

Mit relationalen Wachstumsgrammatiken ist es möglich, ganze Graphen zu ersetzen. Dies geschieht in Form von Regeln. Eine Regel besteht, wie bei den L-Systemen auch, aus einer linken und einer rechten Seite. Die linke Seite soll durch die rechte Seite er-

setzt werden. Dies führt zu einer Suche im Graphen mit dem Ziel eines *matching*. Ist der Teilgraph gefunden, wird er durch die rechte Seite der Regel ersetzt. Abbildung 5.2 demonstriert dies. Auf der linken Seite können Kontexte angegeben werden, die das Vorhandensein von bestimmten Knoten voraussetzen, um eine Regel überhaupt anwenden zu können.

Es ist also festzuhalten, dass relationale Wachstumsgrammatiken den Formalismus bestehender Regelsysteme übernehmen. Sie erweitern die Datenstruktur von Zeichenketten auf Graphen und die Ersetzung von Zeichen auf die Graphenersetzung. Durch die Einbettung in eine höhere Programmiersprache bleibt der prozedurale und objektorientierte Ansatz bestehen.

## 5.2 XL und GroIMP

### 5.2.1 XL: Implementierung der RGG

Der Name XL ist die Kurzform für *extended L-system language* – erweiterte L-System-Sprache. Diese Sprache wurde entworfen, um zum Beispiel Pflanzen relativ einfach zu modellieren. Die Grundlage dafür bietet Java. So wurden alle Konstrukte von Java auch in XL übernommen, so dass alle Laufzeit-Bibliotheken ebenfalls zur Verfügung stehen. Ein einfaches „Hallo-Welt"-Programm in XL wäre also identisch mit einem Java basierten „Hallo-Welt"-Programm [9, Seite 58].
Für die Modellierung im Teil 5.3 werden einige Konzepte von XL genutzt, auf die an dieser Stelle anhand von Beispielen eingegangen wird. Diese kleine Einführung soll lediglich dazu dienen, die Baum-Modellierung nachvollziehen zu können. Eine detailliertere Spezifikation ist der Internet-Seite[1] und der entstehenden Dissertation von Ole Kniemeyer zu entnehmen.

#### Umsetzung eines L-Systems in XL

Das in XL umgesetzte Beispiel *quadratic Koch island* aus 5.1 ist in Abbildung 5.3 gezeigt. Die aus den L-Systemen bekannte Notation für den Pfeil bei einer Regelanwendung ($\rightarrow$) wird in XL durch ==> ersetzt.

```
class QuadraticKochIsland extends RGG
{
    void run()
    [
        Axiom ==> F RU(-90) F RU(-90) F RU(-90) F;

        F ==>     F RU(-90) F RU(90)
                  F RU(90) F F RU(-90)
                  F RU(-90) F RU(90) F;
    ]
}
```

Abbildung 5.3: Beispiel in XL: *quadratic Koch island*

Die Symbole „+" und „−" wurden durch den Rotationsbefehl *RU()* ersetzt. Er sorgt für eine Drehbewegung der *Turtle* um den angegebenen Winkel. Da das „−" nach der Interpretation in *The Algorithmic Beauty Of Plants* [19, Seite 7] eine Drehbewegung um 90° nach links darstellt, wird diese Rotation mit dem Befehl *RU(-90)* ausgeführt. Für eine Rotation um 90° nach rechts wird der Befehl *RU(90)* genutzt.

Die Konstrukte *F* und *RU* existieren als implementierte Java-Klassen. Bei der Interpretation der Regeln werden die einzelnen Befehle zu den jeweiligen Klassen mit gleichen

[1] http://www-gs.informatik.tu-cottbus.de/grogra.de/grammars/index.html – Letzter Zugriff: 10.12.2007

Namen delegiert und die entsprechenden Operationen ausgeführt. Es werden von den Klassen Instanzen erstellt und in Form eines Knotens im Graphen gespeichert. In diesem Beispiel gibt es keine Verzweigungen. Daher ist der Verbindungstyp in diesem Fall immer eine Nachfolge-Verbindung (*successor*).

### Parametrisierung

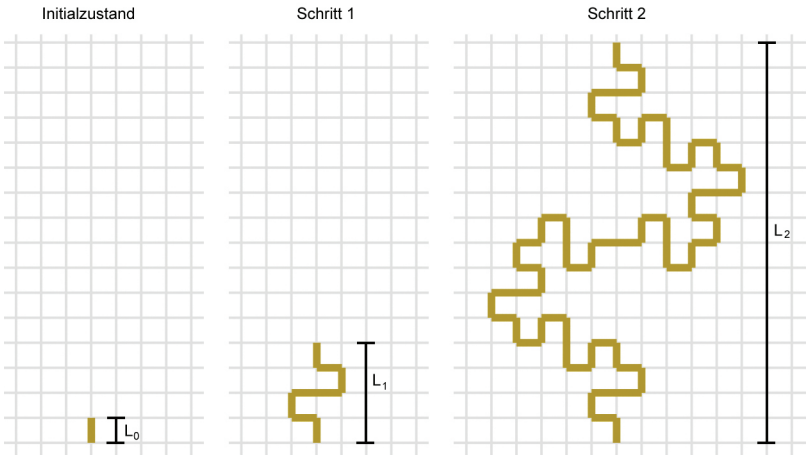


Abbildung 5.4: Regel aus *quadratical Koch island* auf ein einziges Startelement angewendet: Im initialen Zustand entspricht die Gesamtlänge $L_0$ der Länge eines Segmentes, also genau einem Kästchen. Nach dem ersten Schritt steigt die Gesamtlänge $L_1$ auf vier Kästchen. $L_2$ wächst auf 16 Kästchen nach zwei Schritten.

Mit der implementierten Regel würde das Ergebnis nach jeder Ausführung wachsen. Die Abbildung 5.4 zeigt, wie die Ausdehnung der gesamten Struktur ein Vierfaches zunimmt. Damit die Größe der entstehenden „Insel“ gleich bleibt, muss also die Länge der neuen $F$s um den Faktor 4 verkleinert werden. Das heißt, dass die neuen Segmente nur ein Viertel so groß sind, wie das zu ersetzende Segment. Weiterhin besteht die Möglichkeit, bei Objekten auf der linken Seite einer Regel die Attribute zu parametrisieren. Da die einzelnen Befehle in Java-Klassen implementiert und instanzierbar sind, besitzen diese auch mindestens einen Konstruktor. In der Spezifikation der Sprache XL sind für das Objekt $F$ vier verschiedene Konstruktoren vorgesehen. Für dieses Beispiel sind nur die zwei Konstruktoren *F()* und *F(float argument)* interessant. Der erste der beiden Konstruktoren lässt die *Turtle* einen Zylinder der Länge 1 zeichnen. Der zweite Konstruktor erwartet einen Wert für die Länge, mit dem der Zylinder dann gezeichnet wird. So veranlasst der Aufruf *F(2)* das Zeichnen eines Zylinders der Länge 2.
So ist es nun möglich, $F$s unterschiedlicher Länge zu zeichnen. Da aber nun die Länge um den Faktor 4 verkürzt werden soll, muss also die Länge des gefundenen $F$s ermittelbar sein. Parameter der Konstruktoren stehen nicht nur zum Erzeugen der Objekte zur Verfügung, sondern sind auch behilflich bei der Suche nach Objekten im Graphen. So

wird durch die Schreibweise *F(x) ==> ...* nach einem *F* gesucht und die Länge des gefundenen *F*s im lokalen Parameter *x* hinterlegt. Da der Datentyp durch den Konstruktor bereits feststeht, muss keine Deklaration beziehungsweise Initialisierung des Parameters erfolgen. Dies geschieht automatisch.

Nun ist es möglich, mit der Parametrisierung die Gesamt-Größe des Objektes in unserem Beispiel konstant zu halten. Da bekannt ist, dass neue Segmente nur ein Viertel so groß sein dürfen wie das zu ersetzende Segment, wird ein neues Segment mit *F(x/4)* aufgerufen. Die Veränderungen des XL-Codes sind in Abbildung 5.5 demonstriert.

```
class QuadraticKochIsland extends RGG
{
    void run()
    [
        Axiom ==> F(1) RU(-90) F(1) RU(-90) F(1) RU(-90) F(1);

        F(x) ==>  F(x/4) RU(-90) F(x/4) RU(90)
                  F(x/4) RU(90) F(x/4) F(x/4) RU(-90)
                  F(x/4) RU(-90) F(x/4) RU(90) F(x/4);
    ]
}
```

Abbildung 5.5: Beispiel in XL: *quadratic Koch island* mit Parametern

### Benennung von Knoten

Eine andere Möglichkeit, an das Attribut „Länge“ eines gefundenen *F*s heran zu kommen, bietet die *Benennung* von Knoten. Die linke Seite der Regel wird um einen *Bezeichner* ergänzt. Ist eine Suche nach einem Objekt erfolgreich abgeschlossen und es kommt zur entsprechenden Anwendung der Regel, wird das gefundene Objekt dem *Bezeichner* übergeben. Somit können die Attribute direkt vom Objekt abgefragt werden. Auf das Beispiel bezogen, ändert sich der XL-Code wie in Abbildung 5.6 dargestellt.

Durch die Nutzung der Benennung braucht man in diesem Fall nicht mehr die Parametrisierung. Der Bezeichner für ein gefundenes *F* ist in diesem Fall *f*. Dies können aber auch andere Namen sein, solange nicht bestehende Konstrukte den selben Namen haben.
Auf ein gewünschtes Attribut des Objekts wird durch *f[AttributName]* zugegriffen. Da die Länge des gefundenen Zylinders benötigt wird, lautet die Abfrage *f[length]*.

```
class QuadricKochIsland extends RGG
{
   void run()
   [
      Axiom ==> F(1) RU(-90) F(1) RU(-90) F(1) RU(-90) F(1);

      f:F ==>   F(f[length]/4) RU(-90) F(f[length]/4) RU(90)
                F(f[length]/4) RU(90) F(f[length]/4)
                F(f[length]/4) RU(-90) F(f[length]/4) RU(-90)
                F(f[length]/4) RU(90) F(f[length]/4);
   ]
}
```

Abbildung 5.6: Beispiel in XL: *quadratic Koch island* mit Benennung

### Prozeduraler Code in Regelanwendungen

Für das Erzeugen der $F$s wird bei einer Regel-Anwendung immer die gleiche Länge angegeben. Daher liegt es nahe, die Länge nicht für jedes $F$ neu, sondern nur einmal zu berechnen. Dies ist möglich, indem das Ergebnis in einer lokalen Variablen gespeichert wird, die nur zum Zeitpunkt der Regelanwendung existiert. Dazu besteht die Möglichkeit, prozeduralen Code zum Zeitpunkt der Regelanwendung auszuführen. Ein Code-Segement wird mit „{“ begonnen und endet mit „}“. Demonstriert ist dies in Abbildung 5.7.

```
class QuadraticKochIsland extends RGG
{
   void run()
   [
      Axiom ==> F(1) RU(-90) F(1) RU(-90) F(1) RU(-90) F(1);

      f:F ==>   {
                      float laenge = f[length]/4;
                }
                F(laenge) RU(-90) F(laenge) RU(90)
                F(laenge) RU(90) F(laenge) F(laenge) RU(-90)
                F(laenge) RU(-90) F(laenge) RU(90) F(laenge);
   ]
}
```

Abbildung 5.7: Beispiel in XL: *quadratic Koch island* mit Benennung und prozeduralem Code.

### Turtle-Befehle

Jedes Objekt besitzt neben der Position im globalen Koordinaten-System auch ein lokales Koordinaten-System. So wird die lokale x-Achse auch als *left*-Achse bezeichnet, die y-Achse als *up*-Achse und die z- als *head*-Achse.

| Datei | Funktion |
|---|---|
| F(x) | Malt ein zylindrisches Segment der Länge x. Dabei bewegt sich die *Turtle* entlang ihrer *head*-Achse (also lokalen z-Achse). |
| F0 | Ähnlich wie bei F(x). Die Länge des zu zeichnenden Zylinders ist durch den Zustand der *Turtle* vorgegeben. Dieser wird wiederum durch L(x) gesetzt. |
| M(x) | Bewegt sich entlang der lokalen z-Achse und legt dabei die Entfernung $x$ zurück. |
| M0 | Ähnlich wie bei M(x). Die Länge des Schrittes ist durch den Zustand der *Turtle* vorgegeben. Dieser wird wiederum durch L(x) gesetzt. |
| RL(a) | Rotiert die *Turtle* um die lokale x-Achse (*left*-Achse) mit dem Winkel $a$. |
| RU(a) | Rotiert die *Turtle* um die lokale y-Achse (*up*-Achse) mit dem Winkel $a$. |
| RH(a) | Rotiert die *Turtle* um die lokale z-Achse (*head*-Achse) mit dem Winkel $a$. |
| D(x) | Setzt den Durchmesser des nächsten zu zeichnenden Segments auf $x$. |
| L(x) | Setzt die Länge des nächsten zu zeichnenden Segments auf $x$. |
| P(x) | Setzt die Farbe des nächsten zu zeichnenden Segments auf $x$. Dabei kann $x$ ein Zahl sein, die einer von 16 Farben entspricht. Wird in GroIMP durch die Möglichkeit erweitert, Material oder Shader zuzuweisen. |
| RG | Rotiert die *Turtle* so, dass sie global gesehen nach unten zeigt. Die globale z-Achse des Objekts schaut nach unten. |
| AdjustLU | Rotiert die lokale z-Achse so, dass die lokale x-Achse so horizontal und die lokale y-Achse so vertikal wie möglich liegen. |
| RV(x) | Rotiert in Richtung der globalen z-Achse mit dem Wert $x$. Dabei hängt die Stärke der Rotation von der derzeitigen Orientierung der *Turtle* ab. Somit lassen sich Objekte in eine bestimmte Lage relativ zur Erdanziehung ausrichten. Dies ist hilfreich, Pflanzen gerade nach oben wachsen zu lassen, auch wenn der Untergrund geneigt ist. |

Tabelle 5.1: Turtle-Befehle

### Eigene Symbole definieren

Wie bereits beschrieben, basiert XL auf der Sprache Java. So sind die Symbole, die von der *Turtle* interpretiert werden, Java-Klassen. Bei der Interpretation der Symbole werden von diesen Klassen Instanzen erstellt. Neben den vordefinierten Symbolen in Abbildung 5.1 bietet XL auch die Möglichkeit, eigene Symbole zu definieren. Dies geschieht

in Form einer Moduldefinition (siehe Abbildung 5.8). Ein neues Modul wird mit dem Schlüsselwort *module* eingeleitet.

```
module X;

class ModuleTest extends RGG
{
    void run()
    [
        Axiom ==> X;

        X ==>       F(1) RU(90) X;
    ]
}
```

Abbildung 5.8: Moduldefinition

Während der Compilierung des XL-Codes wird das Modul $X$ in eine Java-Klasse umgewandelt. So steht die Klasse $X$ so zur Verfügung, wie andere Klassen in Java auch. Das bedeutet, dass sie instanziert werden kann.

Neben dem Festlegen eines Namens für das Modul können auch Parameter angegeben werden, die dieses Modul näher beschreiben sollen. Abbildung 5.9 zeigt, dass nun das Modul $X$ einen Parameter *length* besitzt.

```
module X(float length);

class ModuleTest2 extends RGG
{
    void run()
    [
        Axiom ==> X(1);

        // Nach jeder Ausfuehrung wird ein neues X erzeugt,
        // das den Wert des zu ersetzenden X um 0.1 reduziert.
        X(len) ==>          F(len) RU(90) X(len-0.1);
    ]
}
```

Abbildung 5.9: Moduldefinition mit Parametern.

Bei der Ausführung des XL-Codes werden auch aus Klassen, die aus der Moduldefinition hervor gehen, Knoten erzeugt und in den Graphen eingefügt. Im Beispiel der Abbildung 5.8 und 5.9 wird zwar der Knoten $X$ erzeugt und in den Graphen eingefügt, jedoch ist dieser Knoten nicht sichtbar.

### Verzweigungen

Um Verzweigungen im Quellcode anzugeben, kommen, wie auch bei L-Systemen üblich, eckige Klammern („[“ und „]“) zum Einsatz. Die *Turtle* interpretiert die Symbole und erzeugt entsprechende Knoten. Aufeinanderfolgende Knoten (Symbole) werden mit Kanten vom Typ *successor* (Nachfahre) verbunden. Bei einer Verzweigung wird der Knoten, resultierend aus dem Symbol vor der öffnenden Klammer, mit dem erzeugten Knoten vom Symbol nach der öffnenden Klammer durch eine Kante vom Typ *branch* (Verzweigung) verbunden. Die Symbol-Folge

$$F[RU(-90)F][RU(90)F]F[RU(-90)F][RU(90)F]F$$

wurde in Abbildung 5.10 auf zwei verschiedene Arten dargestellt. Die linke Seite demonstriert die grafische Interpretation, wobei ein $F$ einem Zylinder entspricht. Auf der rechten Seite ist der interne Graph abgebildet.

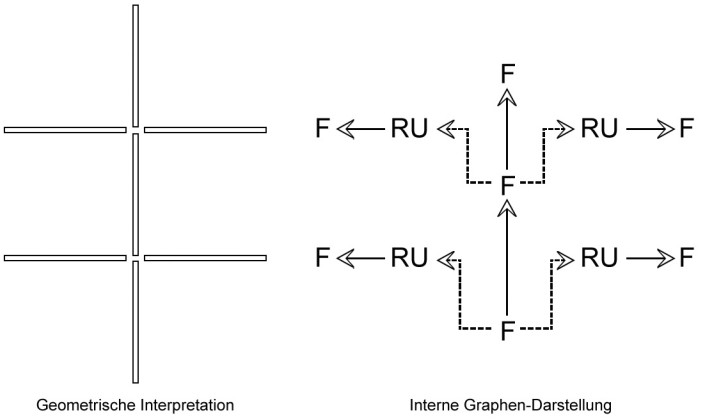


Abbildung 5.10: Darstellung einer Verzweigung: Nachfolge-Verbindungen werden als durchgehende, Verzweigungen als gestrichelte Linien dargestellt.

### Suchausdrücke

Regeln dienen dazu, den Graphen zu verändern oder zu erzeugen. Suchausdrücke (englisch queries) dienen dazu, in einem bereits bestehenden Graphen nach bestimmten Objekten zu suchen. Da die Syntax der linken Seite einer Regel exakt der Syntax eines Suchausdrucks entspricht, kann eine linke Seite auch als eine Art Teilgraph angesehen werden, der durch die rechte Seite der Regel ersetzt wird.

Bei einer erfolgreichen Suche gibt ein Suchausdruck als Ergebnis alle möglichen Übereinstimmungen nacheinander zurück. So gibt der Suchausdruck

(* F *)

alle gefundenen $F$s nacheinander zurück. Eine Beispiel-Implementierung, die alle $F$s um 1 verlängert, ist in Abbildung 5.11 demonstriert. Wie bei der *Benennung* können gefundene Elemente mit einem Namen versehen werden, so dass man mit ihnen arbeiten kann. So wird, wie in diesem Beispiel, jedes gefundene $F$ mit einem $f$ benannt.

```
class Suchausdruck extends RGG
{
        for((* f:F *))
        {
                f[length]++;
        }
}
```

Abbildung 5.11: Es wird im Graphen nach allen $F$s gesucht. Jedes gefundene $F$ wird anschließend um den Wert 1 verlängert.

Suchausdrücke können auch genutzt werden, um arithmetische Operationen auszuführen. Die Funktion

$$count((* \text{ F } *))$$

gibt die Anzahl aller $F$s wieder. Oder die Funktion

$$sum((* \text{ F } *).length)$$

summiert alle Längen der $F$s auf und gibt somit die Summe aller Längen zurück.

Will man nicht alle Elemente eines Graphen, sondern nur Elemente, die einen bestimmten Vorfahren haben, finden, können ebenfalls Suchausdrücke genutzt werden. Abbildung 5.12 demonstriert den Fall, dass alle Nachfolge-Knoten vom Typ $F$, ausgehend von einem bestimmten Knoten, gesucht und durch zwei $F$s ersetzt werden sollen.
Von dem Start-Knoten $x$ ausgehend, werden alle Nachfahren (*descendants*) gesucht, die vom Typ $F$ sind.

### 5.2.2 GroIMP

Um die Sprache XL optimal nutzen zu können, wurde GroIMP entwickelt. GroIMP steht für *Growth Grammar-related Interactive Modelling Platform* – was auf deutsch so viel heißt wie „auf Relationale Wachstums-Grammatiken bezogene interaktive Modellierplattform“ – und wird wie XL auch am Lehrstuhl Grafische Systeme der BTU Cottbus entwickelt. GroIMP ist ein Open-Source-Projekt[2].

Neben der Einbettung der Sprache XL bietet das Programm eine Vielzahl nützlicher Funktionen:

- Interaktives Modellieren von Szenen

[2]http://www.grogra.de – Letzter Zugriff: 13.12.2007

```
module X;
class Suchausdruck extends RGG
{
        // Mit einer zuvor global definierten Variablen kann
        // man sich Objekte merken, wenn auf ihnen zu einem
        // späteren Zeitpunkt gearbeitet werden soll.
        X x = new X();

        // Die Methode init() wird automatisch beim Laden
        // der Szene ausgefuehrt.
        protected void init()
        [
                // Erzeugte Objekte muessen in den Graphen
                // eingefuegt werden. Sie sind nicht
                // automatisch bei der Erzeugung im
                // Graphen enthalten.
                Axiom ==>       F RU(90) x F ;
        ]
        void run()
        [
                // Es werden alle Knoten von x ausgehend gesucht.
                // Handelt es sich bei dem gefunden Knoten um ein
                // F, wird dieses durch zwei neue Fs ersetzt.
                (* x -descendants-> *) F ==>     F F;
        ]
}
```

Abbildung 5.12: Alle Nachfahren vom Knoten $x$ werden durch zwei $F$ ersetzt.

- Umfangreiche Bibliothek von 3D-Objekten, wie Primitiven, NURBS-Kurven und Oberflächen (Surfaces)
- Materialien erweiterbar mit Farben oder Texturen
- Echtzeit-Darstellung mit OpenGL
- Export der Szene in verschiedene Formate
- Eingebauter Renderer (Twilight) zum Berechnen der Bilder von einer Szene
- Eingebautes Lichtmodell

### Graphendarstellung

Neben der interpretierten Darstellung im 3D-Ansichtsfenster von GroIMP bietet das Programm auch die Möglichkeit, den Graphen selbst darzustellen. Über *Panels* → *2D* → *Graph* lässt sich dieser anzeigen. In GroIMP wurde das Beispiel aus Abbildung 5.10 umgesetzt. Dies ist auf Abbildung 5.13 zu sehen.

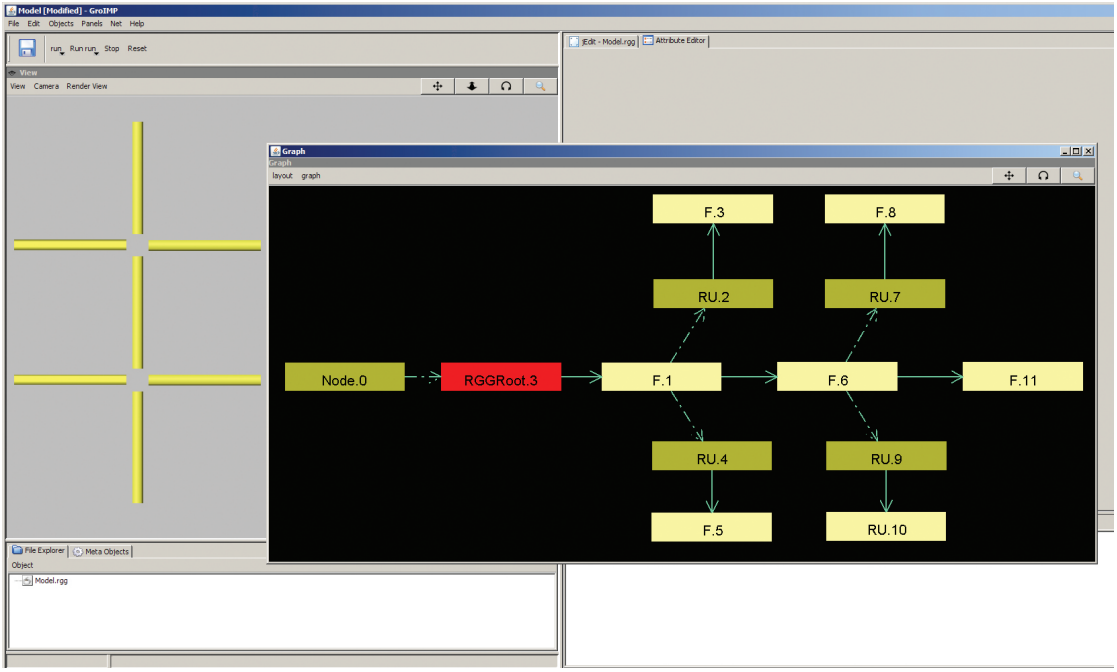


Abbildung 5.13: Darstellungen in GroIMP

### Erweiterte Moduldefinition

Damit selbst definierte Knoten auch sichtbar werden, müssen sie von Klassen abgeleitet sein, die von der *Turtle* geometrisch interpretierbar sind. Solch eine Klasse ist zum Beispiel *Sphere* aus dem *de.grogra.imp3d.objects*-Paket. Vererbungen können in XL, wie in Java, über ein *extends* angegeben werden. Somit übernehmen neu angelegte Klassen Eigenschaften bereits existierender Klassen. Das Modul $X$ soll nun von der Klasse *Sphere* erben. Da $X$ nun alle Eigenschaften der Klasse *Sphere* besitzt, wird beim Zeichnen vom Symbol $X$ eine Kugel dargestellt. Abbildung 5.14 zeigt die veränderte Moduldefinition.
Mit Angabe des Radius im Konstruktoraufruf der Klasse *Sphere* wird der Radius der zu zeichnenden Kugel angegeben. Damit bei der Instanzierung von $X$ auch die Größe der Kugel angegeben werden kann, muss für den Konstruktor von $X$ ein *super.radius* mit angegeben werden. Das *super* sorgt dafür, dass dieser Parameter an die Klasse *Sphere* weitergleitet wird.

### Texturen

Das Modul $X$ ist nun als Kugel sichtbar. Allen sichtbaren Objekten kann ein Material, bestehend aus Farben oder Texturen, zugewiesen werden.
Zuerst wird nun dem Modul $X$ eine grüne Farbe zugewiesen. Dazu gibt es die Methode *setShader()*. Sie wird bei der Modul-Definition mit angegeben (siehe Abbildung 5.15).

```
module X(float length, super.radius) extends Sphere(radius);

class ModuleTest3 extends RGG
{
    void run()
    [
        Axiom ==> X(1, 0.1);

        X(len, r) ==>       F(len) RU(90) X(len-0.1, r);
    ]
}
```

Abbildung 5.14: Modul als Kugel.

Das Ergebnis ist eine Kugel mit einer diffusen Oberfläche in grüner Farbe.

```
module X(float length, super.radius) extends Sphere(radius);
{
        {
                setShader(GREEN);
        }
}
```

Abbildung 5.15: Modul als Kugel mit Farbdefinition.

Um Texturen zuzuweisen, muss als erstes unter *Panels* → *Object Explorers* → *3D* → *Shaders* ein neuer *Shader* angelegt (*Object* → *New*) werden. Zur Auswahl stehen verschiedene Arten von *Shader*. Für einfache Texturen genügt der *Lambert-Shader*.
Nun existiert ein neuer *Shader* mit dem Namen *Lambert*. Ist dieser markiert, kann er umbenannt werden (Taste F2). Als Beispiel wird der Name „Test“ genommen. Mit einem Doppelklick auf den *Shader* öffnet sich das Attribut-Fenster. Nun kann in *Diffuse Color* → *Surface Maps* → *Image* ein Bild als Textur angegeben werden. Im Feld *Image* wird nun der Pfad zur Bilddatei angegeben (*?* → *From File*).
Nun ist der *Shader* mit einer Textur, aus einer Bilddatei bestehend, komplett und kann genutzt werden. Dazu wird ein neuer *Shader* in XL definiert. Die Implementierung ist in Abbildung 5.16 zu sehen.

```
Shader test = shader("Test")
module X(float length, super.radius) extends Sphere(radius);
{
        {
                setShader(test);
        }
}
```

Abbildung 5.16: Modul als Kugel mit einem zuvor definierten *Shader*.

## 5.3 Baumerstellung in GroIMP am Beispiel der Erle

Mit der vorgestellten Plattform GroIMP und der Sprache XL wird nun am Beispiel der Erle eine Modellierung demonstriert. Zunächst wird die Herangehensweise erläutert, um einen groben Fahrplan für diesen Teil des Kapitels zu skizzieren. Das Produkt ist ein Modell der Erle aus dem Branitzer Park, sowie Darstellungen als gerendertes Bild und als Repräsentationen in VRML.
Im Zuge der Entwicklung des Modells gilt es, die Wirkung von neu eingeführten Methoden oder Konstrukten zu verdeutlichen. Dazu wird der klassische „Vorher-Nachher-Vergleich“ bemüht. Da im Modell auch mit stochastischen Funktionen gearbeitet, wird sind optisch verschiedene Ergebnisse bei verschiedenen Programm-Aufrufen, aber identischem Quelltext, die Regel. Um Veränderungen zu verdeutlichen, ist es jedoch wichtig, dass zunächst keine Variationen des Modells auftreten. Variationen könnten Eindrücke verfälschen, und erschweren es somit, Veränderungen zu verdeutlichen. Damit Änderungen des Modells nur auf Änderungen des Quelltextes zurückzuführen sind, wird eine Eigenschaft der stochastischen Funktionen ausgenutzt: Sie basieren auf so genannten Pseudozufallszahlen. Diese Zahlen sind auf deterministische Weise erzeugt, wirken jedoch als hinreichend regellos [12, Seite 24].
Mit dem Zufallszahlen-Generator Mersenne Twister werden Pseudozufallszahlen erzeugt. Diese Zahlen bestehen zahlreiche statistische Test und können deshalb als beliebig verteilt betrachtet werden. Von einer gegebenen Zahl *seed* wird der initiale Zustand des Generators abgeleitet. Auf dieser Basis erzeugt nun der Generator stets die gleichen Zahlenreihen (Pseudozufallszahlen). Mit *setSeed()* und der Angabe einer Zahl kann der initiale Zustand des Generators abgeleitet werden. Somit geben alle stochastischen Funktionen wie zum Beispiel *random()* bei verschiedenen Programmaufrufen die Folge von gleichen Werten wieder. Dies gilt jedoch nur solange, wie die Reihenfolge der Abarbeitung der Regeln in verschiedenen *Threads* (Prozessen) auch bei einem erneuten Programm-Start nicht verändert wird.

### 5.3.1 Herangehensweise (vom Feinen ins Grobe)

Bevor begonnen wird, sei gesagt, dass nicht ein biologisch vollständiges Modell einer Erle erstellt werden soll, sondern es soll ein Ergebnis erzielt werden, was dem Vorbild in visueller Hinsicht möglichst nahe kommt. Dies gilt auch für das Wachstum, das in

den Hintergrund gestellt sei, da es sich hier um eine Zustandsmodellierung handelt. Der Baum wird in seiner derzeitigen Ausprägung modelliert. Wenn von Wachstum in Zusammenhang mit dem Modell gesprochen wird, so handelt es sich hierbei um die grammatikalische Erzeugung von geometrischen Objekten, die einen Teil des Baumes darstellen. Um das gewünschte Ergebnis zu erzielen, muss im Groben die Morphologie betrachtet werden. Hier kann man von einem ad-hoc Modell [12, Seite 34] sprechen, welches einem vorgegebenen Ziel schneller entspricht. Auch wenn keine Messungen mit Lineal und Winkelmesser durchgeführt wurden, so wurden Originalaufnahmen von der Erle herangezogen, um Längen- und Winkelverhältnisse festzustellen. Auch eine Besichtigung vor Ort war durchaus hilfreich, um die Morphologie des Baumes zu begreifen. Aus diesen Beobachtungen sind folgende Erkenntnisse gereift:

- der Baum ist aufgeteilt in 3 Hauptstämme, die sich bis kurz unter die Spitze der Krone erstrecken
- Schätzung der Zahl von Ästen 1. Ordnung auf ca. 10 – 11 pro Stamm
- Abschätzung des Verzweigungswinkels der Äste (ab 2. Ordnung und aufwärts) auf ca. 45°

Weitere morphologische Gegebenheiten wie Biegung, Blattform, Blattansätze usw. werden im nächsten Teil ausführlicher besprochen.

Um Topologie und Morphologie des Modells besser kontrollieren zu können, wird das Modell in einzelne Wachstumszustände (einfacher: Arbeitsmodi) eingeteilt. Somit ist gewährleistet, dass zum Beispiel das Breiten-Höhen-Verhältnis des Baums oder die Anzahl der Hauptäste nicht degeneriert. Zudem ist es dann einfacher, die Verzweigungsstruktur der höheren Ordnung – bis hin zu den Blättern – besser zu steuern, um auch hier eine Degeneration zu verhindern. Daraus lassen sich drei Arbeitsmodi bereits ableiten:

#### Stammwachstum

Mit dem Stammwachstum ist die maximale Höhe, die Anzahl der Hauptstämme, und die Verteilung der Hauptäste (Äste 1. Ordnung – 3. Ordnung) kontrollierbar.

#### Hauptast-Wachstum

Hier lässt sich der Grad der Biegung und die Verteilung von Trieben kontrollieren.

#### Triebwachstum

Als Triebe werden an dieser Stelle Verzweigungen höchster Ordnung mit ihren Blättern bezeichnet. Über das Triebwachstum soll sich dann unter anderem die Blattdichte der Hauptäste steuern lassen. Wie ein solcher Trieb in GroIMP erzeugt wird, wird im Punkt 5.3.2 besprochen.

Durch das Auftrennen der Wachstumsprozesse musste man bei der Entwicklung des Modells nicht den gesamten Baum gleichzeitig betrachten, sondern nur einen kleinen Teil. Wie sich herausgestellte, ersparte dies eine Menge Wartezeit bei den Berechnungen. Dies wird jedoch im Punkt 5.3.3 näher betrachtet.

### 5.3.2 Erstellung eines einzelnen Triebs

Nachdem die Vorbetrachtung abgeschlossen, ist kann nun mit der Umsetzung begonnen werden. Die einzelnen Prinzipien werden systematisch erklärt und eingeführt, so dass sich am Ende dieses Teils des Kapitels ein Modell eines Erlen-Triebes ergibt.

Dieser Teil des Kapitels ist in die folgenden Punkte untergliedert:

- *Bauplan*
- *Umsetzung des Schrittes in GroIMP*
- *Einbeziehung von Lichtverhältnissen*
- *Mortalität von Blättern und Zweigen*
- *Dickenwachstum*
- *Biegung unter eigenem Gewicht*
- *Vitalität*

Zu jedem dieser Punkte (bis auf den Punkt *Bauplan*) gibt es eine entsprechende GroIMP – Projektdatei

#### Bauplan

Bevor mit der Generierung begonnen werden kann, ziehen wir ein Bild eines Triebes der Erle heran. Anhand der Abbildung 5.17 kann dann eine Art „Bauplan“ erstellt werden. Dabei werden jedoch Früchte vernachlässigt.
Der Bauplan veranschaulicht – stark vereinfacht – die Abfolge der geometrischen Objekte. Mit anderen Worten: Wie könnte man den Trieb mit einfachen geometrischen Primitiven (Rechtecken und Zylinder) nachbauen? Durch die vereinfachte Darstellung fällt es dann später leichter, eine Wachstumsregel herauszufiltern. Ein solcher Plan kann wie in Abbildung 5.18 aussehen. Neben den Strukturelementen wird auch deren Anordnung stark vereinfacht. Wie auf dem Foto (Abbildung 5.17) zu erkennen ist, sind die Blattansätze dicht zusammen. Somit kann auch die Anordnung der Blätter im Bauplan auf eine paarweise Anordnung reduziert werden.
Anhand des Bauplans wird eine Wachstumsregel – im weiteren als Schritt bezeichnet – abgeleitet. Eine solcher Schritt kann dann so aussehen: Internodium – Knospe für Verzweigung – Blatt links – Blatt rechts – Knospe für Längenwachstum. Dieser Schritt kann nun in GroIMP umgesetzt werden.



Abbildung 5.17: Einzelner Trieb einer Erle

### Umsetzung des Schrittes in GroIMP

Die geometrischen Objekte, die zum „Wachsen“ benötigt werden, müssen zunächst definiert werden. Dabei spielt die Größe der Objekte in diesem Abschnitt erst einmal keine Rolle und kann beliebig gewählt werden. Die Definition der Objekte ist der Abbildung 5.19 zu entnehmen.
Nun kann mit den definierten Objekten der Schritt in GroIMP umgesetzt werden. Die Implementierung in GroIMP sieht analog zu der oben getroffenen Formulierung eines einzelnen Schrittes aus. Der Wert des Rotationsbefehls $RU$ ist dabei erst einmal beliebig gewählt – darauf wird später eingangen. Als Ausgangspunkt – das Axiom – wird eine Knospe gesetzt. Sie wird dann während der Ausführung des Schrittes ersetzt. Die Implementierung ist in Abbildung 5.20 gezeigt.
Nachdem die Szene in GroIMP geladen oder kompiliert wurde, kommt es zum Aufruf der Funktion *init()*. Es wird das Axiom gesetzt. Die Befehle des Axioms setzen sich aus der Rotation um 90° nach rechts und dem „Zeichnen“ einer Knospe zusammen. Sie erscheint im 3D-Editor-Fenster als rote Kugel (siehe Abbildung 5.21 erstes Bild von links). Durch den Funktionsaufruf *schritt()* wird nun die „alte“ Knospe durch ein Internodium, zwei Knospen und zwei Blätter ersetzt. Die alte Knospe existiert nach dem Schritt nicht mehr. Wie sich das „Modell“ mit drei Funktionsaufrufen entwickelt, zeigt Abbildung 5.21. Es ist zwar nur eine rote Kugel auf dem zweiten Bild zu erkennen, tatsächlich sind dort zwei Kugeln (Knospen). Sie liegen nur deckungsgleich übereinander, was für die weiteren Schritte jedoch keine Folgen hat.

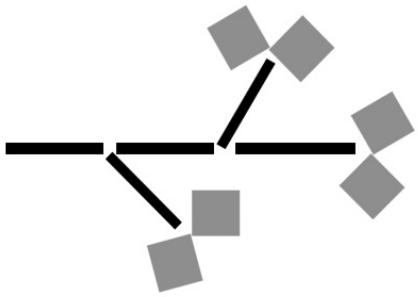

Abbildung 5.18: Möglicher Bauplan: Dabei sind die Internodien schwarze und die Blätter graue Rechtecke.

```
/* Blatt entspricht einem Parallelogramm */
module Blatt extends Parallelogram(0.4, 0.3) {
        { setShader(GREEN); }    // Färbt das Blatt gruen
}

/* Knospe entspricht einer Kugel */
module Knospe extends Sphere(0.1) {
        { setShader(RED); }      // Färbt die Knospe rot
}

/* Internodium entspricht einem Zylinder */
module Internodium extends F(2) {
        { setShader(YELLOW); }  // Färbt das Internodium gelb
}
```

Abbildung 5.19: Modul-Definition; die Einheit ist Meter

Das Ergebnis erinnert eher nicht an den Trieb einer Erle, denn es wurden bisher in diesem Modell die Winkel- und Längenverhältnisse komplett vernachlässigt. Als nächste werden die Winkelverhältnisse mit berücksichtigt.
Um Richtungsänderungen umsetzen zu können, beispielsweise bei Verzweigungen, werden Rotationsbefehle genutzt. Sie gehören zu den *Turtle*-Kommandos. Neben RU() kommt nun *RH()* hinzu. Was jeder einzelne Rotationsbefehl bewirkt, wurde bereits in Abschnitt 5.2 erklärt. Dieser Rotationsbefehl wird benötigt, da der Trieb zur Zeit nur nach oben und rechts wächst. Ein weiteres *Turtle*-Kommando ist *RV()*. Es realisiert den Geotropismus, indem es das Objekt in Richtung der negativen z-Achse rotiert – das Objekt zeigt nach unten. Wie stark die Rotation in Richtung negativer z-Achse sein soll, hängt von dem angegebenem Wert ab. Werden negative Werte für den Parameter des *RV()* angegeben, richtet sich das neue Objekt nach oben.

```
protected void init ()
[
        Axiom ==>
                RU(90)
                Knospe;
]

/* Name kann beliebig gewaehlt sein */
public void schritt ()
[
        Knospe ==>
                Internodium
                [ RU(-45) Knospe ]      // Eine neue Abzweigung
                [ RU(-90) Blatt ]       // Linkes Blatt
                [ RU(90) Blatt ]        // Rechtes Blatt

                // Knospe fuer das Laengenwachstum
                Knospe;
]
```

Abbildung 5.20: Einfache Implementierung eines Wachstumsschrittes

Zwar wurden anhand der Bildvorlage Winkel in das Modell eingefügt, aber auch diese sind noch nicht als „perfekt“ einzustufen. Die Erfahrung hat gezeigt, dass mit dem Einfügen von Biegungen (siehe *Biegung unter eigenem Gewicht*) noch mal ein wenig nachjustiert werden muss, da das Aussehen vom *Turtle*-Kommando *RV()* nachhaltig beeinflusst wurde.
Auch bei den neu zu erzeugenden Blättern werden neue Kommandos eingefügt. Somit ist auch deren Ausrichtung mittels Parameter steuerbar. Dies ist bereits in Abbildung 5.23 Bild 2 zu sehen. Die Implementierung mit den neu hinzugekommenden *Turtle*-Befehlen ist in Abbildung 5.22 gezeigt.

Da nicht alle Triebe an ein und der selben Erle gleich aussehen, ist davon auszugehen, dass eine Variation vorliegt. Das bedeutet, dass sich Teile der Erle ähneln, jedoch nicht gleichen.
Um Variationen in Wachstumsregeln zu integrieren, bietet sich die Nutzung von stochastischen Funktionen an. Zunächst werden für den Trieb zwei verschiedene Funktionen genutzt. Die Funktion *normal(x, y)* generiert eine Normalverteilung um den Wert $x$ mit einer Standardabweichung vom Wert $y$. Eine beliebige Zahl zwischen *min* und *max* gibt die Funktion *random(min, max)* zurück. Damit verschiedene Triebe nicht identisch aussehen, können diese stochastischen Funktionen innerhalb von *Turtle*-Befehlen verwendet werden. So variiert der Wert des auszuführenden *Turtle*-Befehls bei jedem Aufruf innerhalb eines vorgegeben Wertebereichs. Auf der Abbildung 5.23 Bild 3 ist eine Varianz bereits integriert.

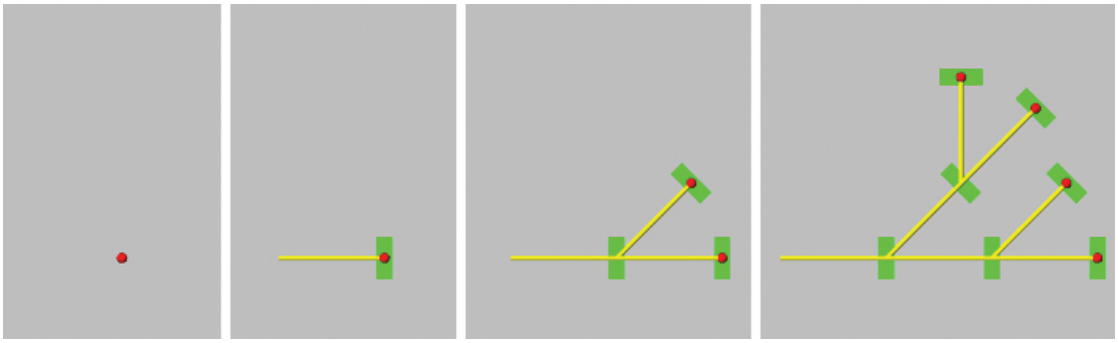

Abbildung 5.21: Ausgangszustand und dreimalige Anwendung der Funktion *schritt()*.

Wie auf der Abbildung 5.18 zu erkennen ist, gibt es einen Seitenwechsel zwischen zwei Verzweigungen. Wir vereinfachen diese Beobachtung dahin gehend, dass eine Verzweigung in jedem geraden Schritt nach links und in jedem ungeraden nach rechts zeigt. Durch einen Zähler, der bei jedem ausgeführten Schritt um eins erhöht wird, kann dann ein Vorzeichen für die Orientierung des Befehls *RH()* berechnet werden.
Das Resultat der modifizierten Implementierung ist auf dem 4. Bild in Abbildung 5.23 zu sehen. Zu erkennen ist das Abwechseln der Seiten bei den Verzweigungen: der vordere Zweig zeigt nach links, der hintere Zweig nach rechts.

Von nun an werden Internodien und Blätter mit der entsprechenden Textur dargestellt. Dazu wurden Fotos von der Rinde sowie der Blätter der Erle herangezogen und entsprechend präpariert. Wie man in GroIMP Texturen definiert und benutzt, ist bereits in Punkt 5.2 besprochen worden. Damit die Verzweigungsstruktur besser zu erkennen ist, werden die Blätter zunächst ausgeblendet.

Auf den Abbildungen 5.21 und 5.23 ist zu sehen, dass die Internodien sehr geradlinig verlaufen. Um dies zu vermeiden, wird ein Internodium durch mehrere Internodien und entsprechende *Turtle*-Kommandos ersetzt. Doch zuerst wird die Moduldefinition des Internodiums erweitert; das Modul des Internodiums bekommt einen Längen-Parameter hinzu. Da das Internodium ein $F$ (Zylinder) erweitert und die Länge des $F$s modifiziert werden soll, muss der Parameter *super.length* heißen. So ändert sich auch der Aufruf bei der Erzeugung eines Internodiums: es kann nun die Länge in Metern mit angegeben werden. Der Wert dieses Parameters ist in diesem Fall konstant und kann daher auch als Konstante definiert werden. In Abbildung 5.24 kann die modifizierte Modul-Definition und die neu hinzugekommene Konstanten-Deklaration betrachtet werden.
Nachdem die Länge des Internodiums per Parameter verändert werden kann, wird nun die Struktur durch das Ersetzen mehrerer Internodien mit *Turtle*-Befehlen verfeinert. Mit einem weiteren $RV$-Kommando soll der Schwerkraft-Einfluss die *Turtle*-Bewegungen angedeutet werden. Zusätzlich wird der Befehl *RU()* eingesetzt. Mit einer zufälligen Zahl

```
public void schritt ()
[
        Knospe ==>
                Internodium
                [       // Eine neue Abzweigung
                        RH(160) RU(45) RV(-0.06)
                        Knospe
                ]
                [       // Linkes Blatt
                        RH(-45) RU(60) RV(-0.5)
                        Blatt
                ]
                [       // Rechtes Blatt
                        RH(45) RU(45) RV(-0.5)
                        Blatt
                ]
                // Knospe fuer das Laengenwachstum
                Knospe;
]
```

Abbildung 5.22: Einfache Implementierung eines Wachstumsschrittes

zwischen -4 und +4 entstehen im Zusammenwirken mit *RH()* Windungen beim Wachsen der Internodien und der geradlinige Eindruck verschwindet etwas, wie auf der Abbildung 5.25 zu erkennen ist.

#### Einbeziehung von Lichtverhältnissen

Das derzeitige Modell erzeugt bei jedem Schritt eine neue Verzweigung. Die Abbildung 5.26 zeigt, was nach 8 ausgeführten Schritten passiert: es wuchert. Das Ergebnis ist sehr befremdlich und erinnert nicht an einen Erlen-Trieb. In der Natur wächst nicht automatisch aus jeder Knospe auch ein Internodium. Hier gibt es verschiedene Faktoren, die darüber entscheiden, ob die Knospe schläft, austreibt oder abstirbt. Der Faktor Licht ist dabei einer der wichtigsten Faktoren [12, Seite 206].

Durch bestimmte Lichtverhältnisse kommt es bei einer Knospe zu den Zustandsübergängen. Im derzeitigen Modell gibt es nur zwei mögliche Zustände einer Knospe: Entweder sie treibt aus oder sie stirbt ab. Den Zustand „schlafend“ gibt es quasi nur zwischen 2 Schritten. Aber spätestens in dem für sie zweiten Schritt entscheidet sich, was mit ihr geschieht. Um zu entscheiden, welches Schicksal die Knospe ereilen soll, muss für jede Knospe das Lichtverhältnis ermittelt werden.
Zur Messung von Lichtverhältnissen wird ein Lichtmodell verwendet. Das in Kniemeyer et al. [11] beschriebene „Radiation model“, zu deutsch Strahlenmodell, implementiert ein solches Lichtmodell, und basiert auf dem Prinzip eines Raytracers. Es werden, ausgehend von einer Lichtquelle, Strahlen durch eine Szene geschickt. Die Anzahl der Strahlen

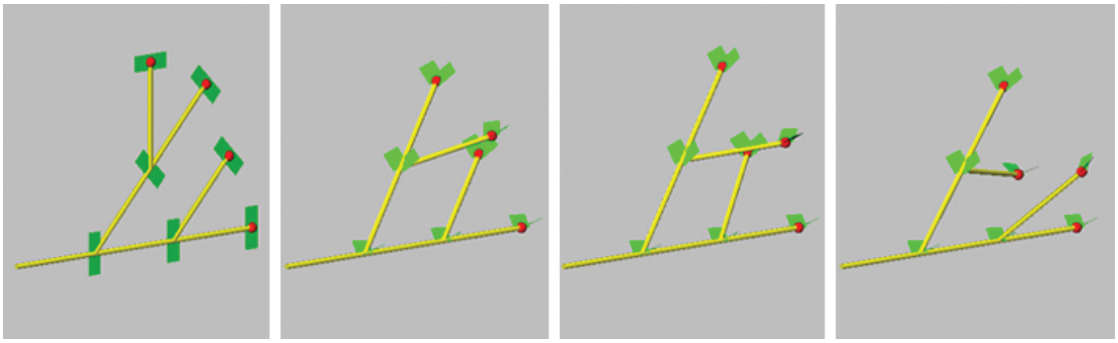

Abbildung 5.23: Aufnahme von halb linker Position.

```
const float INTERNODIUM_LAENGE          = 0.005;

/* Internodium entspricht einem Zylinder */
module Internodium(super.length) extends F(length) {
        { setShader(YELLOW); }  // Macht das Internodium gelb
}
```

Abbildung 5.24: Erweiterte Moduldefinition des Internodiums.

sowie die Leistung der Lichtquelle werden vorher festgelegt. Alle sichtbaren Objekte sind potentielle Strahlen-Empfänger. Für jedes Objekt liefert das Licht-Modell die Intensität, mit der ein Objekt beleuchtet wurde. Die Intensität ist in Watt angegeben. Das Lichtmodell ist fester Bestandteil des Programms GroIMP und lässt sich einfach anwenden.

Je nachdem, wie die Oberfläche des Objektes beschaffen ist, wird ein Strahl reflektiert, absorbiert oder gestreut (in alle Richtungen reflektiert). Um genau zu sein, wird jeder Strahl, der auf ein Objekt trifft, zerstört. Für den Fall, dass es keine vollständige Absorption gibt, wird dann ein neuer Strahl erzeugt, der vom Objekt ausgeht.
Das Lichtmodell wird nun in die bestehende Implementierung des Trieb-Modells integriert. In diesem Fall reicht die Bestimmung der Lichtintensität völlig aus. Sie wird als eine Zahl ausgedrückt. Wie das Lichtmodell erzeugt und benutzt wird, ist in der Abbildung 5.28 demonstriert.

Wie in der Einführung zu XL bereits erwähnt, können bei einer Regelanwendung die Objekte auf der linken Seite der Produktion benannt werden. Dieses Konzept kommt bei der Abfrage der Lichtintensität einer Knospe zum Einsatz. Alle gefundenen Knospen werden mit einem $k$ benannt. So kann die Lichtintensität der Knospe durch die Hilfsfunktion *belichtet()* ermittelt werden, indem die gerade zu behandelnde Knospe übergeben wird. Da das Lichtmodell zur Berechnung eine Lichtquelle benötigt, wird eine Punkt-Lichtquelle in die Szene integriert. Der Lichtquelle ist anschließend der Licht-Typ *Sun Sky* zuzuweisen. Dieser Licht-Typ sorgt dafür, dass nicht nur die Sonne Licht emittiert, sondern auch

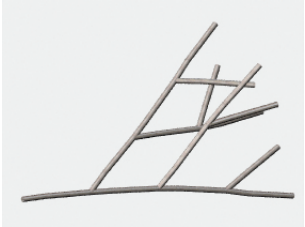

Abbildung 5.25: Leichte Verwindungen der Internodien (Blätter sind ausgeblendet).

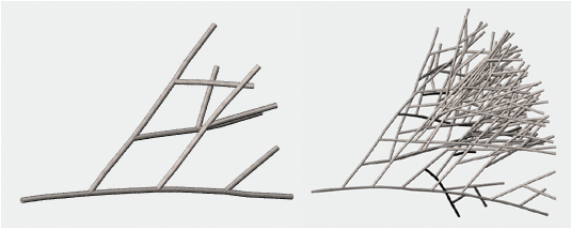

Abbildung 5.26: Links ein Trieb nach 4 Schritten. Rechts nach 8 Schritten.

der Himmel.



Abbildung 5.27: Trieb nach 8 Schritten einmal mit und ohne Blätter.

Wie sehr der Faktor Licht das Erscheinungsbild des Triebes beeinflusst, zeigt Abbildung 5.27. Auch dieser Trieb wurde in 8 Schritten erzeugt, ist jedoch bei Weitem nicht so buschig wie der Trieb auf dem rechten Bild der Abbildung 5.26.
Die Präsenz von Blättern und Ästen erzeugt einen Schatten. Die Lichtintensität nimmt im Schattenbereich ab. Wenn im Schatten die Licht-Intensität zu gering ist, kann die Knospe kein Internodium erzeugen – sie stirbt ab. Dieses Verhalten ist auch in der Natur zu beobachten: große Bäume haben in der Regel im Inneren der Baumkrone weniger Blätter als auf der Außenseite.
Die Werte, die entscheidend sind, ab wann eine Knospe ein Internodium bilden darf oder nicht, sind beliebig gewählt worden. Bei der Entwicklung des Modells wurde mit verschiedenen Werten experimentiert. Die in der Implementierung benutzten Werte hatten bis dato die besten Ergebnisse geliefert.

Zur Zeit ist ein Wachstumsschub gleichbedeutend mit einem Jahr. Damit in einem Jahr mehrere Wachstumsschübe stattfinden wird, nun eine *for*-Schleife integriert. Die Schleife wird beim Aufruf der Methode *schritt()* zwischen zwei, und viermal durchlaufen. Somit gibt es innerhalb eines Jahres (1 Aufruf *schritt()*) zwischen zwei bis vier Wachstumsschübe.

```
// Erzeugen eines Lichtmodells mit 200.000 Licht-
// strahlen und einer Rekursionstiefe von 10
LightModel lm = new LightModel(200000, 10);

/* Hilfsfunktion zur Wiedergabe der Lichtstärke */
float belichtet(Node n){
        return lm.getRadiantPowerFor(n).integrate();
}

public void schritt ()
{
        // Um das Lichtmodell auf die aktuelle Szene
        // anzuwenden, muss es neu kalkulieren.
        lm.compute();
        [
                k:Knospe  ==> if(belichtet(k)>30)
                        ( /* Wachse */ );
        ]
}
```

Abbildung 5.28: Lichtmodell: Ein einfaches Beispiel der Nutzung.

### Mortalität von Blättern und Zweigen

Wie bei einer Erle üblich, sollten Blätter im Herbst abfallen, in diesem Fall also zwischen zwei Schritten. Dafür wird eine Methode *blattSterben()* erstellt. Sie sucht im Graphen nach allen Blättern und soll das Blattsterben umsetzen.
Da der Methoden-Aufruf am Ende der Methode *schritt()* erfolgt und vorher die Veränderungen am Graphen noch nicht vorgenommen sind, werden nur all die Blätter entfernt, die aus dem letzten Schritt-Aufruf stammen. In Abbildung 5.29 ist der Unterschied zwischen einem Trieb ohne (1. Bild) und mit Blatt-Sterben (2. und 3. Bild) zu erkennen.

Wie auf dem Bild 2 der Abbildung 5.29 zu sehen ist, sieht der Trieb nun sehr kahl aus. Um dies zu vermeiden und eine Kontrolle über das Blattsterben zu erlangen, wird das Modul *blatt* und die Methode *blattSterben()* erweitert. Das *blatt* bekommt einen Parameter namens *alter*. Bei jedem Schritt kann dann in der Methode *blattSterben()* das Alter um eins erhöht werden. Mit dem Alter des Blattes und einer Wahrscheinlichkeitsfunktion *probability()* kann das Absterben der Blätter dosiert werden. Die Funktion *probability(p)* gibt den Wert *wahr* mit der angegebenen Wahrscheinlichkeit $p$ zurück. Bild 3 auf der

Abbildung 5.29 hat die Einschränkung, dass alle Blätter jünger als vier Jahre mit einer Wahrscheinlichkeit von 80% überleben. Alle anderen sterben ab. Eine mögliche Implemenotierung ist in Abbildung 5.30 zu sehen. Diese Vorgehensweise erscheint zunächst unbiologisch, da in der Realität bei der Erle in jedem Herbst alle Blätter abgeworfen werden. Sie kann jedoch gerechtfertigt werden durch das Vorhandensein von Kurztriebketten, die jedes Jahr um ein sehr kurzes Internodien mit genau einem Blatt weiterwachsen. Daher wächst in der Tat über mehrere Jahre immer wieder an fast der gleichen Stelle ein neues Blatt. Auf Abbildung 5.17 sind solche Kurztriebketten zu erkennen.



Abbildung 5.29: Trieb nach 10 Schritten: Links ohne, rechts mit Blatt-Sterben. Ganz rechts ist die dosierte Version vom Blattsterben

```
public void schritt ()
{
        [ ... ]
        blattSterben();
}

protected void blattSterben ()
[
        b:Blatt(a)  ==>>
                {
                        b[alter] :+= 1;
                }
                if (a<4 && probability(0.8))
                (       break   /* Bleib am Leben! */   );
]
```

Abbildung 5.30: Die Module mit ihren neuen Parametern

Bei der Entstehung der Baumstruktur ist das Absterben von Teilen des Baumes ein wichtiger Bestandteil [12, Seite 169]. Jeder Ast hat eine Tendenz, irgendwann in seiner

Entwicklung keine neuen Internodien zu produzieren. Das führt ein Absterben des Astes und somit den Abwurf in einer bestimmten Zeitspanne herbei. Bilden sich an Tochterästen keine neuen Zweige, wachsen auch keine neuen Blätter. Der Ast wird kahl.
Das Integrieren des Ast-Sterbens in das Modell der Erle setzt Bedingungen voraus, die gelten müssen, damit ein Ast als tot gilt. Damit ein Ast als tot gilt, darf er keine:

- Blätter
- Knospen und
- Tochter-Äste

besitzen. Dazu ist eine weitere Veränderung der Moduldefinition notwendig. Zuerst wird ein neues Konstrukt *Lebend* deklariert. Dieses Konstrukt wird als *interface* angegeben und dient als eine Art Markierung[3] für Blätter und Knospen. Die Module *Knospe* und *Blatt* sollen dieses *interface* implementieren. Durch diese Markierung kann nun nach Blättern und Knospen zeitgleich gesucht werden, indem man einfach nach Objekten sucht, die als *Lebend* markiert sind. Das vereinfacht die Realisierung der Suche, um feststellen zu können, ob ein Ast sterben muss oder nicht.

Ein weiteres Modul *Ast* wird deklariert. Es implementiert ebenfalls *Lebend* und wird dazu benötigt, die Bedingung zu überprüfen, ob ein Ast Tochter-Äste besitzt. Das neu hinzugekommene Modul wird an den Stellen im Graphen platziert, die den Beginn einer neuen Verzweigung bilden. Somit ist das Modul *Ast* quasi ein Ansatz einer Verzweigung (desweiteren als Astansatz-Punkt bezeichnet). Die veränderte Modul-Definition zeigt Abbildung 5.31.

Alle Vorbereitungen sind getroffen, das eigentliche Ast-Sterben kann nun umgesetzt werden. Die Methode *astSterben* soll diese Aufgabe übernehmen.
Es werden alle Module vom Typ *Ast* „besucht“ und von ihnen ausgehend eine Suche nach Kinder-Objekten durchgeführt, die als *Lebend* markiert sind. Nur wenn keine solchen Objekte gefunden werden, gibt die Funtktion *empty()* ein *true* zurück und der Ast wird „abgeschnitten“. Der Schnitt wird am Astansatz-Punkt durchgeführt. Alle Kinder-Objekte von *Ast* werden aus dem Graphen entfernt. Auf Abbildung 5.32 ist zum Vergleich ein Trieb ohne und ein Trieb mit Ast-Sterben dargestellt. Beide Triebe wurden mit je 20 Schritten generiert.

### Dickenwachstum

Komplett vernachlässigt wurde bisher die Dicke eines Internodiums. Sie wurde zu Beginn dieses Abschnitts des Kapitels beliebig festgelegt. Hierfür gibt es in der Natur eine grundlegende Gesetzmäßigkeit: Je mehr Tochter-Äste an einem Ast sind, desto dicker ist dieser Ast. Das hat in der Natur den Hintergrund, dass Tochter-Äste versorgt werden müssen (zum Beispiel mit Wasser). Je mehr Zweige zu versorgen sind, desto breiter

[3] Im Englischen wird dieses Konzept als *tagging interface* bezeichnet.

```
interface Lebend{}

module Blatt(int alter)
        extends Parallelogram(0.06, 0.05)
        implements Lebend { ... }

module Knospe
        extends Sphere(0.1)
        implements Lebend {      ... }

module Ast implements Lebend;

protected void astSterben()
[
        x:Ast, (empty((* x ->minDescendants-> Lebend *))) ==>> ;
]
```

Abbildung 5.31: Die Module mit ihren neuen Parametern

müssen die Versorgungswege des Astes sein. Außerdem ist die Dicke auch relevant für Stabilität und Zugfestigkeit. Je mehr Äste und Zweige an einem Blatt hängen, desto mehr Gewicht muss dieser halten.
Da Vinci stellte mit dem „Röhrenmodell“ die These auf, dass die Querschnittsfläche eines Hauptastes die Summe aller Querschnittsflächen der Tochter-Äste ist [7, Seite 22]. Dies wird nun mit zwei Methoden realisiert. Die Methode *berechneDurchmesser()* geht durch den gesamten Graphen und besucht dabei alle Internodien. Zu jedem Internodium werden die Querschnittsflächeinhalte der Kinder-Internodien aufsummiert und der daraus resultierende Durchmesser in der Methode *durchmesser()* berechnet und zurückgegeben (siehe 5.33).
Die Methode *berechneDurchmesser()* berechnet mit einer Tiefensuche die Durchmesser jedes einzelnen Internodiums. Dies geschieht mit der Methode `synthesize()`. Ihr wird durch das Zeichen `^` im ersten Suchausdruck mitgeteilt, dass mit der Suche bei der Wurzel des Gesamt-Graphen begonnen werden soll. Der Rest des Suchausdrucks legt fest, welche Art von Konten dabei berücksichtigt werden sollen, in diesem Fall Knoten vom Typ `Internodium`. Beim Suchen im Graphen werden alle gefundenen Internodien in einer internen Objekt-Liste `ObjectList` gemerkt. Von jedem gefundenen Internodium wird eine erneute Tiefensuche vorgenommen. Der aktuelle Knoten der Tiefensuche wird in der Instanz `syn` der Klasse `FloatSynth` zwischengespeichert. Diese Klasse repräsentiert all die Informationen, die für eine Berechnung benötigt werden. In diesem Fall bedeutet das, dass alle Durchmesser-Werte der Internodien ausgelesen werden müssen. Somit können zu jedem relevanten Internodium durch `sum(syn.values())` die Durchmesser-Werte der Kind-Internodien aufsummiert werden. Das Ergebnis der Summierung wird in der Variable `v` zwischengespeichert und der Funktion *durchmesser()* übergeben, welche den resultierenden Durchmesser-Wert für den aktuell betrachteten Konten nach dem Ansatz

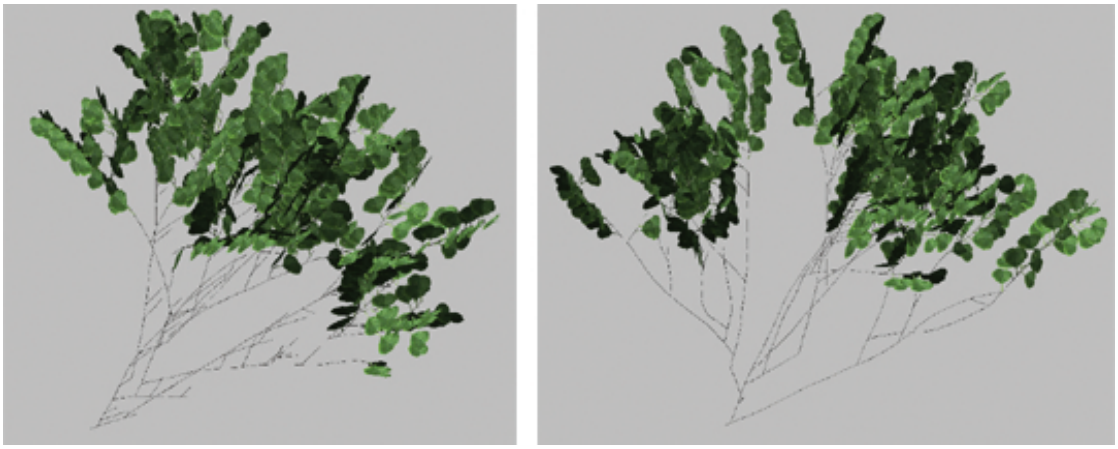

Abbildung 5.32: Triebe nach 20 Schritten ohne und mit Ast-Sterben.

von Da Vinci zurückgibt.

### Biegung unter eigenem Gewicht

Für die Biegung wird nochmals das *Turtle*-Kommando *RV()* genutzt. Dazu muss zunächst die Modul-Definition der *Knospe* um den Parameter *ordnung* erweitert werden. Die Äste mit den meisten „Kinder-Zweigen“ haben meist auch die geringste Ordnungs-Zahl. So hat ein Ast 1. Ordnung (niedrige Ordnung) sehr viele Äste höherer Ordnung zu tragen. Daher biegen sie sich stärker als junge Zweige (hoher Ordnung), die weniger Kinder-Äste tragen müssen. Für die Implementierung bedeutet das, dass der Wert für das *RV()* bei Ästen niedriger Ordnung höher sein muss als bei Ästen höherer Ordnung. In diesem Fall soll der Wert oberhalb von der 0. Ordnung geringer sein. Ab welcher Ordnung welcher *RV*-Wert wirken soll, wurde durch Testen verschiedener Werte herausgefunden. Durch zahlreiche Versuche haben sich diese Werte als visuell ansprechend erwiesen. Die Änderungen der Implementierung sind der Abbildung 5.35 zu entnehmen. Wie sich das Modell mit dem *RV()*-Befehl verhält, ist wiederum in Abbildung 5.36 zu erkennen.

### Vitalität

Bevor der Trieb im Gesamt-Modell zum Einsatz kommt, muss ein weiteres Konstrukt integriert werden. Der Grund dafür ist der, dass der Trieb in der jetzigen Form unkontrolliert und unendlich lange wächst. Jedoch nimmt das Wachsen eines Triebes mit zunehmender Entfernung zur Wurzel ab. Der Grund ist der lange Weg für die Versorgung mit Nährstoffen. Dies ist auch ein Faktor, der das Bild eines Baumes maßgeblich prägt. Deswegen muss eine Art Lebenskraft-Steuerung integriert werden: die *Vitalität*. Sie ist im unseren Modell eine Zahl, die bei der Festlegung des Schicksals der Knospe mit Einfluss nimmt. Die *Vitalität* nimmt bei jeder neuen Knospen-Bildung um den Wert 0.8 ab. Auch hier wurden durch Versuche die „richtigen“ Parameter-Werte näherungsweise

```
static float durchmesser(float flaechenSumme)
{
        float r = Math.sqrt (flaechenSumme/Math.PI);

        return r*2;
}

public void berechneDurchmesser()
{
        synthesize ((* ^ -minDescendants-> Internodium *),
                Node n => Internodium*
                        (* n -minDescendants-> Internodium *),
                FloatSynth syn => float
                        (float v = sum(syn.values()),
                        ((Internodium) syn.object)[diameter] =
                            durchmesser(v),
                         v)
                );
}
```

Abbildung 5.33: Methoden zur Durchmesserberechnung

ermittelt.

Ein Trieb kann mit unterschiedlichen Anfangs-Vitalitäten erzeugt werden. Durch die stetige Abnahme um den Faktor 0.8 ist die *Vitalität* in unserem Modell nichts anderes als ein Stopp-Mechanismus. Das bedeutet: wird ein bestimmter Wert unterschritten, kann es sein, dass die Knospe abstirbt, oder sie lässt Internodien wachsen, bildet jedoch keine Verzweigungen aus.

Neben der *Vitalität* in der *Knospe* wird dieses Konstrukt auch bei den Blättern eingeführt. Da im weiteren Verlauf der Modellierung mehrere Triebe mit verschiedenen Anfangs-Vitalitäten „wachsen“, kommt es zu unterschiedlichen Stopp-Zeiten. Das heißt, jeder Trieb stellt zu unterschiedlichen Zeitpunkten sein Wachstum ein. Da aber die *blattSterben()*-Methode unaufhörlich weiter läuft, würden Blätter von kleinen Trieben schneller sterben. Die Folge: die Blattdichte am Trieb nimmt ab oder er verschwindet im schlimmsten Fall ganz, da ja auch die Methode *astSterben()* weiter ausgeführt wird.
Die Blätter bekommen die *Vitalität* von der *Knospe* übergeben. Zusätzlich wird in der Methode *blattSterben()* per Abfrage die *Vitalität* des Blattes geprüft. Unterschreitet einen bestimmten Wert, darf es auf jeden Fall überleben, egal wie alt es ist. Abbildung 5.37 zeigt drei Triebe mit jeweils unterschiedlicher Anfangs-Vitalität. Auf alle drei Triebe wurden die gleichen Regeln angewendet. Lediglich die Anfangs-*Vitalität* unterschied sich. So stoppte das Wachstum vom linken Trieb als erstes. Kurz danach stoppte dann das Wachstum des mittleren, wobei der rechte Trieb noch wuchs.



Abbildung 5.34: Triebe nach 20 Schritten ohne und mit Durchmesserberechnung.

### 5.3.3 Generierung des gesamten Baumes

Mit der Modellierung des Triebes ist nun ein entscheidener Bestandteil der Erle betrachtet und realisiert. Jetzt geht es darum, mit Hilfe des Triebes einen ganzen Baum zu modellieren. Es stellt sich jedoch die Frage, warum den Baum nicht einfach direkt aus dem Trieb erzeugen lassen? Nun, dies ist theoretisch möglich, jedoch zeigt Abbildung 5.38, wie schnell der Trieb degeneriert.
Da auf den ersten Blick der Stamm sowie die Hauptäste beim Wachsen scheinbar anderen Gesetzmäßigkeiten als die Triebe unterliegen, liegt es nahe, die Bereiche voneinander zu trennen. Daher auch das Aufteilen, wie schon im Teil 5.3.1 erwähnt, auf die drei Wachstums-Arbeitsmodi *Stammwachstum, Hauptastwachstum* und *Triebwachstum.*
Was in den einzelnen Arbeitsmodi geschieht, soll in diesem Teil näher beleuchtet werden. Dabei wird nicht nur auf das Ergebnis geachtet, sondern auch auf die Faktoren Laufzeit und Speicherverbrauch. Das Modell des Triebes aus dem Teil 5.3.2 hat bereits nach 20 Schritten ca. 12.000 sichtbare Objekte. Da ein Vielfaches an Objekten im gesamten Baum zu erwarten ist, muss auch auf eventuelle Speicherprobleme in diesem Kapitel eingegangen werden.

#### Arbeitsmodi des Modells

Wie bereits erwähnt, ist die Gesamt-Baum-Generierung in 3 Arbeitsmodi aufgeteilt. Unter einem Arbeitsmodus sei hier zu verstehen, dass zu einem bestimmten Zeitpunkt im Modell nur bestimmte Methoden oder Funktionen ausgeführt werden. Die Abbildung 5.39 zeigt unter anderem die drei genannten Arbeitsmodi und im Groben die Folge von Übergängen.

Nach der der *Initialisierung* des Modells kommt es zum Erzeugen des Stammes. Ist dieser

```
module Knospe(int ordnung)
        extends Sphere(0.1)
        implements Lebend {       ... }

public void schritt()
{
        ...
[
        k:Knospe(o) ==>
                        ...
                        if(belichtet(k)>30)
                        (
                                RG(o > 1 ? 0.2 : 0.1)
                        )
```

Abbildung 5.35: Die Module mit ihren neuen Parametern



Abbildung 5.36: Bild eins zeigt das gleiche Modell wie Bild zwei, nur wurde bei der Regelanwendung des zweiten Triebes die Biegung der Äste mit berücksichtigt. Nach einer erneuten „Justierung“ der Winkel und Werte für die *Turtle*-Befehle entstand der Trieb auf dem dritten Bild dieser Abbildung.

„gewachsen“, kommt es zu einem Wechsel in den Arbeitsmodus *Hauptast-Wachstum*, wo ein Haupt-Ast erzeugt wird. Daraufhin wird das *Triebwachstum* gestartet. Sind alle Methoden beziehungsweise Funktionen des Arbeitsmodus durchlaufen, heißt der laufende Arbeitsmodus *Ast-Überprüfung*. In diesem wird überprüft, ob eine gewünschtes Blatt-Anzahl erreicht ist. Wenn dann auch noch eine vorgegebene Haupt-Ast-Anzahl erreicht wurde, kann die gesamte Baum-Generierung gestoppt werden (Arbeitsmodus *Finalisierung*). Andernfalls wird wieder in den *Stammwachstum*-Modus gewechselt. Ab diesem Punkt wiederholt sich der Ablauf, bis der Baum fertig ist.
Nachdem im Modus *Stammwachstum* Stamm-Segmente (Internodien) erzeugt wurden, wird ein Hauptast erzeugt (*Hauptast-Wachstum*). Dieser besitzt mehrere Knospen, aus denen dann im Modus *Triebwachstum* die Triebe entstehen. Diese Reihenfolge wiederholt sich, bis eine gewünschte Haupt-Ast-Anzahl erreicht ist.



Abbildung 5.37: Der Trieb ganz links hatte zu Beginn eine *Vitalität* von 10, in der Mitte 100 und rechts 300.

Bevor das Erzeugen des gesamten Baumes gestartet werden kann, muss zunächst eine Lichtquelle in der Szene vorhanden sein. Auch hier kann wieder ein Punkt-Licht mit dem Licht-Typ *Sun Sky* genommen werden. Wichtig ist jedoch die Position der Sonne am Himmel. In den Optionsfeldern des *Sun Sky* Licht-Typs kann ein Richtungsvektor (x, y, z) angegeben werden. Für die Modellierung wurde der Vektor auf (0, 0, 1) gesetzt, so dass die Sonne im Zenit steht. Dadurch ist eine gleichmäßige Ausleuchtung der Szene gewährleistet.

Was in den einzelnen Arbeitsmodi geschieht, soll nun näher beleuchtet werden. Dabei werden die Methoden/ Funktionen, die zum Einsatz kommen, erklärt.

**Initialisierung** Der Einstiegspunkt für XL-Modelle ist die Methode *init()*. Zu allererst wird das Axiom gesetzt. Es erzeugt drei Abzweigungen im Graphen und setzt pro Abzweigung ein *ErleStamm*-Modul. Das Modul wird dann durch Internodien ersetzt, die den Stamm bilden sollen.
Ist dies geschehen, wird *initStammWachsen()* aufgerufen. Diese Methode ersetzt jedes der *ErleStamm*-Module mit Internodien. Je nachdem, wie hoch die ersten Haupt-Äste sein sollen, kann *initStammWachsen()* mehrmals aufgerufen werden.
Auch wenn die Methode *init()* nicht beendet ist, so gibt es an dieser Stelle einen Wechsel des Arbeitsmodus. Die *Initialisierung* ist abgeschlossen und das *Stammwachstum* gestartet. Die Methode *init()* ist nach dem ersten *Hauptast-Wachstum* beendet. Wie das Modell nach dem Ende der Methode *init()* aussieht zeigt Abbildung 5.40.

**Stammwachstum** Mit dem Methoden-Aufruf *StammWachsen()* startet dieser Modus. Es werden ebenfalls Internodien erzeugt, die den Hauptstamm bilden sowie einen Ansatz-Punkt für den Haupt-Ast.

**Hauptast-Wachstum** Der erzeugte Haupt-Ast-Ansatz wird nun in der Methode *hauptAst()* durch Internodien ersetzt. Als Parameter kann man dieser Methode mittei-



Abbildung 5.38: Einzelner Trieb nach 20, 40 und 60 Schritten.

len, wie oft sich der Haupt-Ast verzweigen soll. Bei der Bildung der Internodien werden auch gleich die Knospen mit einer entsprechenden Verzweigung für das *Triebwachstum* erzeugt.

**Triebwachstum** Die Methode *schritt()* ist auch gleichzeitig der Einstiegspunkt, um die gesamte Modellierung zu starten. Durch entsprechendes „Anklicken“ des Buttons wird die Methode ausgeführt. Mit ihr wird zunächst das Licht-Modell anhand der Szene berechnet. Dann kommt es durch den internen Aufruf *wachsen()* zur Regel-Anwendung des *Triebwachstum*s. Die Regelanwendung wurde bereits im Teil 5.3.2 ausführlich besprochen. Aus den einzelnen Knospen auf dem aktuellen Haupt-Ast wachsen dann die einzelnen Triebe. Nach dieser Regel-Anwendung werden die Regeln für das Blatt-Sterben (*blattSterben()*) und Ast-Sterben (*astSterben()*) ausgeführt.

Am Ende der Methode wird wieder der Arbeitsmodus gewechselt. Dies geschieht durch den Aufruf *astUeberpruefung()*.

**Astüberprüfung** In diesem Modus wird auf dem gerade zu bearbeitenden Haupt-Ast gezählt, wie viele Blätter existieren. Ist die Anzahl der Blätter über einer vorgegebenen Zahl, wird von hier aus in den Modus *Stammwachstum* und anschließend in *Hauptast-Wachstum* gewechselt. Zu den beiden zuletzt genannten Modi müsste somit im Schema 5.39 theoretisch jeweils ein Pfeil vom Modus *Astüberprüfung* kommen. Da dies jedoch die Abbildung verkomplizieren würde und das Schema einfach nur die zeitliche Abfolge der Modi skizzieren sollte, sind die Pfeile ausgelassen worden. Tatsächlich ruft niemals eine Methode des Arbeitsmodus *Stammwachstum* eine Methode des *Hauptast-Wachstum*

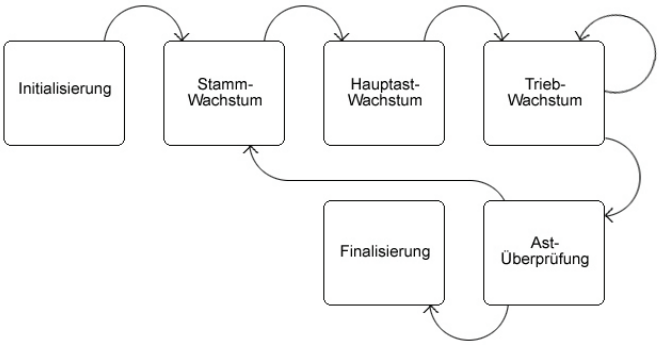


Abbildung 5.39: Übergänge von einem zum anderen Arbeitsmodus.



Abbildung 5.40: Nach dem ersten Aufruf von *Hauptast-Wachstum*: Da noch keine Durchmesser-Berechnung vollzogen wurde, sind die Durchmesser der Internodien noch gering.

auf.

**Finalisierung** Wenn eine bestimmte Anzahl der Haupt-Äste erreicht ist, wechselt das Modell in den Arbeitsmodus *Finalisierung*. Damit die Spitzen der Hauptstämme nicht abgehackt aussehen wird, auch dort ein Triebwachstum angewendet. Auf den Spitzen der einzelnen Stämme werden jeweils drei Knospen gesetzt. Danach wird das Triebwachstum mittels *wachsen()* durchgeführt. Sind alle Methoden dieses Arbeitsmodus durchlaufen, ist die Baum-Modellierung beendet. Anschließend kann das Modell in GroIMP gerendert werden. Abbildung 5.41 zeigt ein gerendertes Bild eines Erlen-Modells.

### Gründe für die Methodik

Durch die Aufteilung in einzelne Arbeitsmodi ist zu entnehmen, dass der Baum zu keinem Zeitpunkt an allen Stellen gleichzeitig betrachtet wird. Beim „Wachsen“ des Baumes wird Haupt-Ast für Haupt-Ast erzeugt. Wie in Abbildung 5.40 zu betrachten ist, existiert nach dem ersten Durchlauf des Arbeitsmodus *Haupt-Ast-Wachstum* nur ein Haupt-Ast. Wenn auf ihm nach einigen Durchläufen das Trieb-Wachstum beendet ist, wird ein neuer Haupt-Ast im entsprechenden Arbeitsmodus generiert. So „wächst“ die Erle sukzessive



Abbildung 5.41: Bild eines mit GroIMP erzeugten und gerenderten Erlen-Modells.

nach oben. In Abbildung 5.42 sind einige Etappen des Wachstums in die Höhe zu sehen. Mit der Entscheidung, das Trieb-Wachstum separat zu betrachten und zu entwickeln, ist es daher auch nur für bestimmte Lichtverhältnisse definiert. Fällt ein Schatten von darüberliegenden Ästen mit ihren Blättern auf den gerade wachsenden Trieb, würden die Knospen weniger Licht empfangen. Damit sinkt die Lichtintensität, was die Möglichkeit eines „Austreibens“ der Knospe verringert. Es könnten dann unter Umständen Triebe entstehen, die sehr wenig Blätter hätten beziehungsweise sehr kahl aussehen würden.
Was bei einem großflächigen Schattenwurf mit dem Trieb passieren kann, zeigt Abbildung 5.43. Der rechte Trieb hat den Bereich unter dem Objekt (Rechteck) beim Wachsen „gemieden“. Damit die Triebe möglichst ohne Schatten wachsen können, dürfen keine Objekte über den noch wachsenden Trieben sein. Da erst ein neuer Haupt-Ast erzeugt wird, wenn der aktuelle Haupt-Ast fertig ist, und die einzelnen Haupt-Äste von unten nach oben erzeugt werden, ist ein Schattenwurf von bereits fertigen Haupt-Ästen so gut

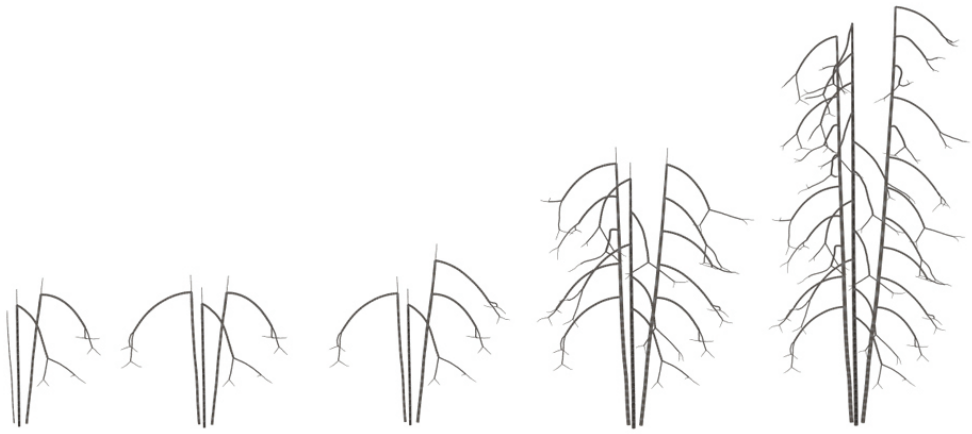

Abbildung 5.42: Etappen des Wachstums in die Höhe. Das Trieb-Wachstum wurde hierbei ausgelassen.

wie ausgeschlossen.



Abbildung 5.43: Durch die veränderte Licht-Situation des rechten Triebes ist dessen Erscheinungsbild ein anderes.

Ein weiterer Grund ist, dass GroIMP bei Operationen auf größeren Graphen mehr Zeit und Speicher braucht. Kann man einen aktiven Bereich des Graphen einschränken, bringt dies Laufzeit-Vorteile. Dies soll im nächsten Teil dieses Kapitels näher betrachtet werden. Es wird dabei unter anderem untersucht, wie sich ein zwischenzeitliches Herausnehmen von Teil-Graphen auf die Laufzeit auswirkt.

#### Speicherplatz-Optimierung

Wie bereits erwähnt, ist in GroIMP bei Operationen auf großen Graphen mehr Prozessoren-Leistung und Speicher nötig. Dies haben durchgeführte Messungen ergeben. Mit ein paar Tricks kann man dem entgegenwirken, welche an dieser Stelle näher erläutert werden. Ließ man beispielsweise drei Haupt-Äste gleichzeitig wachsen, dauerte dies circa 2 Minuten und benötigte circa 500 MB Arbeitsspeicher. Ließ man die Haupt-Äste einzeln wachsen, konnten 15 Sekunden und circa 100 MB an Arbeitsspeicher einspart werden. Dies haben mehrere Durchläufe ergeben.

**Reduzierung von Ansicht-Aktualisierungen** Ist die Entstehung eines Modells bei der Generierung uninteressant, kann die Aktualisierung im 3D-Ansichtsfenster reduziert werden. Das Zeichnen des 3D-Modells im Ansicht-Fenster kostet Zeit. Wird das Modell nicht nach jeder Regelanwendung neu angezeigt, kann die Zeit des Zeichnens im Ansicht-Fenster eingespart werden. Wird bei jedem Schritt neu gezeichnet, dauert die Generierung länger. Bei dem genannten Beispiel mit den 3 Haupt-Ästen hat das Auslassen der Aktualisierung 30-40 Sekunden ausgemacht. Abbildung 5.44 zeigt die Verwendungsweise.

```
public void mehrereSchritte ()
{
        for(apply(30))
        {
                schritt();
        }
}
```

Abbildung 5.44: Jeder 30. Aufruf von *schritt()* führt zu einer Aktualisierung der Anzeige.

**Knoten-Reduzierung** Eine von zahlreichen Funktionen, die GroIMP bereits liefert, heißt *mergeTransformations()*. Sie fasst in einem Graphen Rotationsknoten (z.B. *RU* oder *RH*) in einem *Null*-Knoten – so genannte Transformationsknoten – zusammen. Dies reduziert bei größeren Graphen schnell die Anzahl von Knoten. Graphen-Teile, die bereits zusammengefasste Rotationen haben, brauchen nicht mehr „besucht“ zu werden. Daher kann der Methode ein Start-Punkt im Gesamt-Graphen gegeben werden. Ab da wird dann die Zusammenfassung gestartet.

**Start-Punkt setzen** Eine weitere Möglichkeit ist das Setzen eines Start-Punktes. Der Start-Punkt ist dabei ein Knoten im Gesamt-Graph. Nur Nachfahren des Knoten werden bei der Regelanwendung beachtet. Das bedeutet, dass der Start-Knoten gleichzeitig die Wurzel des Teil-Graphen bildet. Das Setzen dieses Start-Punktes wird mit so genannten Queries (Suchausdrücken) realisiert. Abbildung 5.45 zeigt, wie ein solcher Start-Punkt angegeben werden kann. Bei der Beispiel-Regel-Anwendung wird zuerst nach dem *start-Knoten* im Graphen gesucht. Wenn er gefunden ist, wird ab dem Punkt nach jeder Knospe gesucht. Dabei muss jede Knospe ein Nachfahre vom *startKnoten* sein. Somit ist es möglich, andere Teil-Graphen, die durch Regel-Anwendungen keine weiteren Veränderungen erfahren, nicht mehr zu beachten.

**Ausblenden von Teil-Graphen** Wie bereits bei der Start-Punkt-Methode angedeutet, kann es vorkommen, dass Teil-Graphen nicht mehr verändert werden. Deshalb ist es naheliegend, sie aus dem Gesamt-Graphen zwischenzeitlich auszublenden oder gar komplett auszugliedern. Bei der Ausblend-Methode wird die Verbindung (Kante) vom Gesamt-Graphen zum Teil-Graphen markiert. Wenn es zu Such-Operationen auf dem Gesamt-

```
public void schritt (Node startKnoten)
[
        ((* startKnoten -descendants-> *)) Knospe  ==> ... ;
]
```

Abbildung 5.45: Beispiel für die Nutzung von Start-Punkten

Graphen kommt, wird der Teil-Graph nicht mehr beachtet. Dennoch existiert der Teil-Graph nach wie vor in der Szene und benötigt auch dementsprechend Speicher. Soll das Modell der Erle in GroIMP gerendert werden, kommt die Methode *ausblenden()* zum Einsatz und wird immer dann aufgerufen, wenn ein Haupt-Ast fertig ist. Nachdem alle Haupt-Äste erzeugt sind, kann mittels Aufruf *einblenden()* die Erle zusammengesetzt werden und steht anschließend als komplettes Modell zum Rendern zur Verfügung.

**Ausgliedern von Teil-Graphen** Da der fertiggestellte Haupt-Ast und somit auch der entsprechende Teil-Graph für die restliche Zeit der Generierung des Modells nicht benötigt wurden, konnte dieser zwischenzeitlich aus dem Graph ausgegliedert werden. Dies kann in der Form geschehen, dass der Teil-Graph in eine Datei ausgegliedert und bei Bedarf wieder eingelesen und eingegliedert wird. Nachdem der Teil-Graph gespeichert wurde, kann er aus dem Gesamt-Graph gelöscht werden. Diese Methode wurde nicht konkret umgesetzt. Jedoch kommt das Prinzip im Teil 5.3.4 und 5.3.5 zum Einsatz.

### 5.3.4 Rendern mit externem Renderer Sunflow

Neben der Möglichkeit, den GroIMP-Renderer zu nutzen, soll nun gezeigt werden, wie ein externer Renderer die Erle rendert. Dies soll mit dem Open-Source Renderer *Sunflow*[4] demonstriert werden, einem auf Java basierenden Raytracer, der auch Global Illumination beherrscht.

Wie *Sunflow* installiert und benutzt wird, ist in einem „Quick-Start“-Manual beschrieben. Es ist eine inoffizielle kompakte Bedienungsanleitung, die von Community-Mitgliedern zusammengestellt wurde. Die Anleitung ist dem Forum der Projekt-Seite zu entnehmen.

Um *Sunflow* zu nutzen, muss eine Szene-Datei erstellt werden. In ihr ist die komplette Beschreibung der Szenerie (Geometrie, Licht, Texturen, Shader, usw) enthalten. Wie das *Sunflow*-Szenen-Datei-Format zu nutzen ist, ist in der erwähnten Anleitung beschrieben. Da der Baum nur aus 2 verschiedenen Grundkörpern besteht, kann bei der Erstellung der Szene-Datei das Instanzierungs-Prinzip angewendet werden. Instanzierung heißt, dass das geometrische Objekt, z.B. ein Zylinder, einmal definiert wird. Das heißt, es wird festgelegt, aus wievielen Dreiecken (Polygonen) das Objekt besteht und welche Lage sie im Raum zueinander haben. Danach kann es beliebig oft benutzt werden. Größe, Lage und Position im Raum können dann separat angegeben werden. Bei dem Erstellen

[4] http://sunflow.sourceforge.net/ – Letzter Zugriff: 30.11.2007

```
object {
        type generic-mesh
        name "Zylinder"
        points 14
        0.0 0.0 0.0
        ....
        0.0 0.0 1.0
        triangles 24
        0 2 1
        ....
        13 12 7
}

instance {
        name "internode_1"
        geometry Zylinder
        transform row
                1 0 0 0
                0 1 0 0
                0 0 1 0
                0 0 0 1
}
```

Abbildung 5.46: Beispiel einer Szene-Datei für Sunflow

der Instanz können diese drei Parameter über eine Transformations-Matrix angegeben werden. Da auch GroIMP mit Transformations-Matrizen arbeitet, ist keine komplexe Umwandlung notwendig. Als erster Schritt müssen die zwei Grund-Objekte, die dann später instanziert werden, definiert werden.

Zur Zeit hat *Sunflow* keine Zylinder als Primitive. Daher muss in einem 3D-Modellierungsprogramm ein solcher Zylinder erstellt und in das *Sunflow*-Szene-Format exportiert werden. Exporter für die einzelnen Programme werden auf der Internet-Seite von *Sunflow* angeboten. Da die Größe eines Zylinders über die Transformations-Matrix skaliert wird, ist ein festes Verhältnis zwischen Durchmesser und Länge des Zylinders wichtig. Das Verhältnis wurde auf 1:10 festgelegt.
Neben dem Zylinder ist das Blatt der zweite Grundkörper, der auch erstellt werden muss, da *Sunflow* noch keine transparenten Texturen verwendet. Rechtecke eignen sich nicht, um Blätter darzustellen. Deshalb muss ein kleines Blatt modelliert werden. Da es später in gerenderten Bildern nur einen kleinen Teil einnimmt, reicht eine polygonarme Darstellung des Blattes aus. Das Blatt wird ebenfalls in einem 3D-Editor modelliert und in das Szenen-Format von *Sunflow* exportiert.
Es empfiehlt sich, mit mehreren Szene-Dateien zu arbeiten und sie erst mit Aufruf des Programms zusammenzufügen. Auf diese Weise kann man an Licht-Einstellungen oder Kamera-Positionen arbeiten, ohne dabei die Datei mit allen Instanzen laden zu müssen.



Abbildung 5.47: Mit Sunflow gerenderte Erle.

Dies ist von Vorteil, wenn mit großen Szene-Dateien gearbeitet werden muss.

Nachdem nun alle Vorbereitungen getroffen worden sind, kann in GroIMP die Methode *export()* eingeführt werden. Sie geht nach dem Aufruf durch den gesamten Szene-Graphen und sucht alle sichtbaren Objekte. Wurde ein Objekt gefunden, wird geprüft, um welchen Typ Objekt es sich handelt. Ist der Typ ermittelt, wird der entsprechende Eintrag in die Szene-Datei vorgenommen. Das bedeutet, es wird eine neue Instanz angelegt. Da der Typ des Objektes durch die Prüfung bekannt ist, wird der Instanz die entsprechende Geometrie, Textur und der *Shader* zugewiesen.
Die Transformations-Matrix ist relativ einfach von jedem Objekt in GroIMP zu ermitteln und entsprechend zu übergeben. Jedoch muss die Größe der Instanz angepasst werden. Dies geschieht mit den ersten 3 Werten der Diagonale einer 4x4 Matrix. Es wird der Skalierungsfaktor für die Höhe, Breite und Tiefe des Objektes berechnet und mit der Matrix multipliziert. Somit sind die Größen-Verhältnisse zwischen den einzelnen Objekten wieder hergestellt.

Diese Methode hat den Vorteil, dass, wie schon im Teil **Speicheroptimierung** angedeutet, nach jedem fertigen Haupt-Ast ein Export vorgenommen werden kann. Nach diesem Export kann dann der Haupt-Ast aus dem Szene-Graphen gelöscht werden.

### 5.3.5 Export nach VRML

Nachdem ein Export zu einem externen Renderer beschrieben wurde, wird nun das gleiche Prinzip in GroIMP angewendet, um ein Modell nach VRML zu exportieren. Ähnlich wie bei dem *Sunflow*-Export wird auch bei dieser Methode der gesamte Graph nach allen sichtbaren Objekten abgesucht, und abhängig vom Typ des gefundenen Objektes wird dann ein entsprechendes VRML-Objekt erstellt. Wie bei der *Sunflow*-Szene-Datei werden in einer VRML-Datei zwei Prototypen erzeugt und können je nach Bedarf instanziert werden.

In der VRML-Datei werden zwei Prototypen, Zylinder für die Internodien und eine Box für die Blätter, definiert. Neben der geometrischen Erscheinung und der Textur werden auch Parameter, jeweils *rotate*, *trans*, *innerTrans*, angegeben. Separat erhält der Zylinder die Parameter *radius* und *height*, die Box den Parameter *size*. Mit diesen Angaben für die Prototypen kann die Lage, Größe und Position im Raum angegeben werden.
Bei der Darstellung von Objekten ist auf die Art der Erzeugung zu achten. In VRML werden Objekte aus der Mitte heraus gezeichnet, was bei der Positionierung der Objekte im Raum zu berücksichtigen ist. Der Sachverhalt wurde bereits von Schneider [21, Seite 75] für X3D, dem Nachfolger von VRML, besprochen.
Das Problem wird auf dem linken Bild der Abbildung 5.48 verdeutlicht. Zwei Zylinder unterschiedlicher Länge sollen aufeinandergestellt werden. Der grüne Zylinder hat die Länge 1, der rote Zylinder die Länge 0.8. Zusätzlich bekommen beide Zylinder unterschiedliche Angaben zur Position. Der grüne Zylinder soll die Koordinate (0,0,0) besitzen. Der rote Zylinder soll direkt auf dem grünen Zylinder, also auf dem Punkt (0,1,0), stehen. Jedoch entsteht eine Lücke zwischen beiden Zylindern.
Im Gegensatz zu GroIMP, wo die Position eines Objektes im Raum auch gleichzeitig die Startposition für das Zeichnen des Objektes ist, wird in VRML von der angegebenen Position im Raum nach außen hin gezeichnet. Der rote Zylinder auf dem linken Bild der Abbildung 5.48 wird von der Stelle $y = 1$ jeweils 0,5 Einheiten nach oben und unten gezeichnet. Eine Lösung des Problems besteht darin eine weitere Translation im Prototypen hinzuzufügen, die den Startpunkt des Zeichnens so verschiebt, dass die äußere Startposition mit der Position im Raum übereinstimmt. In dem Beispiel verschiebt die innere Translation den Startpunkt des Zeichnens um genau die Hälfte der Höhe im lokalen Koordinaten-System nach oben. Somit stimmen nach dem Zeichnen des Objektes äußerer Startpunkt und globale Position überein. Durch die innere Translation ist somit von GroIMP aus keine weitere Umrechnung der Koordinaten notwendig.
Mit den nun erstellten Prototypen kann in GroIMP jetzt der eigentliche Export umgesetzt werden. Neben der Position im Raum und der Größe muss auch die Lage bestimmt werden. Dies geschieht mit einer Achsen-Rotations-Angabe. Das bedeutet, dass durch die Angabe eines Richtungs-Vektors und eines Drehwinkels die Lage definiert wird. Dazu bietet das Paket *vecmath* die Klasse *AxisAngle4d()*. Mit der Transformations-Matrix kann man über Quaternionen die Achsen-Rotation berechnen lassen.

Mit einer Traversierung durch den Graphen von einem Startpunkt aus, der als Parame-

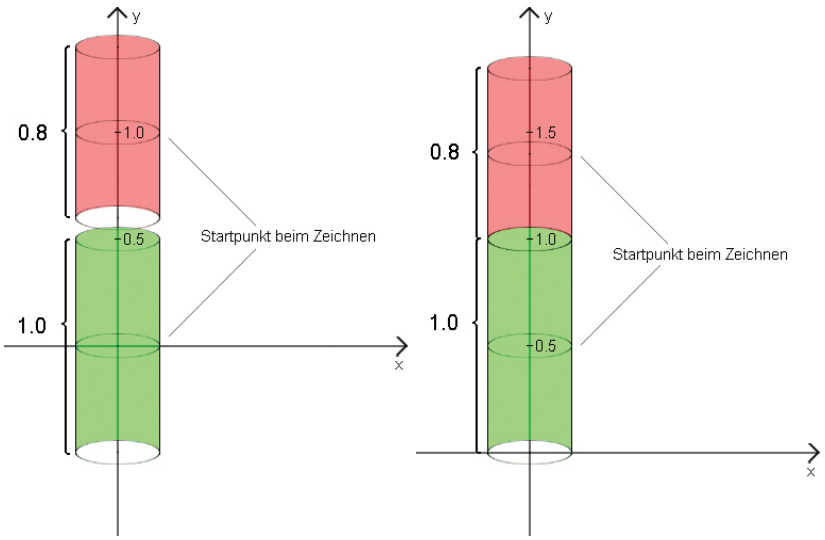


Abbildung 5.48: Linkes Bild: Die angegebenen Positionsdaten sind auch gleichzeitig die Start-Punkte zum Zeichnen der Zylinder. Somit entsteht eine Lücke zwischen beiden Zylindern.
Rechtes Bild: Durch Angabe einer inneren Translation kann der Startpunkt zum Zeichnen verschoben werden. Somit können beide Zylinder (grün auf den Punkt (0,0,0), rot auf den Punkt (0,1,0)) direkt positioniert werden, ohne dass eine Lücke entsteht.

ter der Methode übergeben wird, können nun die entsprechenden Instanzen für VRML erzeugt und in eine Datei geschrieben werden. Genau wie bei dem *Sunflow*-Export kann nach dem Schreiben in die Datei der Teil-Graph vom Startpunkt aus gelöscht werden. Somit kann man auch einen kompletten Baum als VRML-Datei abspeichern. Jedoch hat es sich gezeigt, dass mit einer zu großen Anzahl an Objekten eine Betrachtung in einem VRML-Betrachter nicht zu empfehlen ist. Daher ist es zur Zeit nur möglich, einen einzelnen Trieb mit ungefähr 4000 Objekten, wie in Abbildung 5.49 zu sehen, anzeigen zu lassen.

### 5.3.6 Billboard-Generierung für VRML

Beim Rendern des Objekts wird das, was die Kamera sieht, gerendert. Um *Billboard*s zu erstellen, muss also als allererstes die Kamera um ein Objekt gleichmäßig rotieren können. Gleichmäßig heißt, dass der Abstand zum Objekt und der Betrag des Winkels zwischen zwei aufeinanderfolgenden Perspektiven gleich bleiben muss. Denn würde man die erstellten Bilder in einer Video-Sequenz bündeln, käme es beim Abspielen des Videos zu Sprüngen, da sich von einem Bild zum anderen die Perspektive zu schnell oder zu langsam ändert. Damit eine kontinuierliche Reihe von Bildern erstellt werden kann, muss also die Kamera möglichst genau ausgerichtet werden. Dazu wird ein Werkzeug



Abbildung 5.49: Screenshot aus einem VRML-Betrachter

gebraucht, das diese Anforderungen erfüllt und eine exakte Rotation durchführt. Dieses Werkzeug wird nun eingeführt.

#### Kamera-Rotation

Um die Kamera-Rotation umzusetzen, wird die aktuelle Kamera aus der 3D-Ansicht genutzt. Da GroIMP für die Positionierung der Kamera eine 4x4 Matrix nutzt, muss lediglich eine Logik entworfen werden, die eine Kamera um einen bestimmten Punkt mit einen bestimmten Winkel rotiert. Diese Logik ist in der Klasse *RotationCamera* im Paket *de.grogra.billboard* implementiert worden.

Bei der Initialisierung der Klasse wird zunächst aus der aktuellen 3D-Ansicht das aktuelle Kamera-Objekt geholt. Dass bereits die aktive Kamera genutzt und keine neue instanziert wird, hat den Vorteil, dass das Starten des Renderers erleichtert wird.
Als nächstes wird eine neue Initial-Matrix erstellt, für spätere Berechnungen gespeichert und der Kamera zugewiesen. Alle Rotationen werden dann, von dieser Initial-Matrix ausgehend, berechnet. Somit kann bei Ausführung einer Rotation ein absoluter Winkel angegeben werden.
Nachdem ein Objekt der *RotationCamera* erzeugt wurde, kann die Entfernung zum Koordinaten-Ursprung und die Höhen-Lage der Kamera angegeben werden. Da es sich in unserem Fall um einen einzelnen Baum handelt, ist der Koordinaten-Ursprung gleichbedeutend mit dem Objekt-Ursprung, denn der Ursprung des Objekts liegt im Bereich des Koordinaten-Ursprungs.
Zusätzlich ist es auch möglich, die Kamera um die lokale X-Achse rotieren zu lassen, sodass sich die Kamera nach oben oder unten neigt. Damit die Kamera um ein Objekt rotiert, muss lediglich die Methode *moveAroundZ()* aufgerufen werden. Ihr wird die Grö-

ße des Winkels übergeben, der angibt, wie weit die Kamera vom initialen Ausgangspunkt ausgehend rotiert werden soll. Somit ist die Winkel-Angabe absolut.

Durch Aufruf der Methode *moveAroundZ()* wird eine weitere Methode *calculateMatrix()* aufgerufen. Sie berechnet mit dem gegebenen Winkel und der Initial-Matrix die neue, resultierende Matrix und gibt diese zurück.

### Aufruf des Renderers

Nachdem die Kamera nun punktgenau gesetzt werden kann, wird nun ein weiteres Hilfsmittel eingesetzt. Es ermöglicht, nach dem Rotieren der Kamera den Renderer automatisch aufzurufen. Somit ist der Ablauf automatisiert, und es muss für die gesamte *Billboard*-Generierung nur ein Methoden-Aufruf getätigt werden. Damit der zu nutzende Renderer über die Benutzeroberfläche konfigurierbar ist, muss der Renderer wie in Abbildung 5.50 aufgerufen werden.

```
public void billboarding()
{
    ObjectExpr objExpr    = (ObjectExpr) Item.resolveItem
                            (workbench(), "/renderers/3d/ray2");
    Raytracer raytracer   = (Raytracer) objExpr.evaluate
                            (workbench(), new StringMap());

    // Startet den Renderer
    view3d.getViewComponent().render((Renderer) raytracer);
}
```

Abbildung 5.50: Aufruf des Raytracing-Renderers Twilight

Die Implementierung ruft zwar aus XL heraus den Renderer auf, führt aber nach Beendigung des Renderns keine weiteren Befehle aus. Deshalb muss dem Renderer mitgeteilt werden, was nach der Bildfertigstellung geschehen soll. Dazu führt der Renderer eine interne Liste von so genannten *ImageObserver*n (zu deutsch Bild-Überwacher). Sie ermöglichen es, ein Bild zu überwachen und mitzuteilen, wenn bestimmte Ereignisse mit den Bild-Daten geschehen. Beim Laden eines Bildes zum Beispiel kann der Observer mitteilen, ob das Bild fertig geladen wurde oder ob sich der Daten-Bestand des Bildes immer noch erweitert. Ähnlich ist dies auch beim Rendern. Verändert sich der Datenbestand des Bildes, weil Bildbereiche fertig gerendert wurden, spricht man davon, dass das Bild aktualisiert wurde. Kommen keine Veränderungen hinzu, also hat der Datenbestand sich nicht geändert, ist das Bild fertig. Der *ImageObserver* ist ein Interface und erwartet eine Implementierung der Methode *imageUpdate()*. Diese Methode überprüft, ob das Bild eine Aktualisierung erfahren hat, oder nicht und gibt einen *boolean*-Wert zurück. Ist der Renderer mit der Bildberechnung noch nicht fertig, wird *true* zurückgegeben, das Bild wurde aktualisiert, und die Methode *imageUpdate()* ist beendet. Ist das Bild fertig, wird kein *true* zurückgegeben, und der Rest der Methode wird ausgeführt. Hier kann

nun das fertige Bild gespeichert, die Kamera auf die nächste Position und der nächste Render-Aufruf gesetzt werden.

```
raytracer.addImageObserver(new ImageObserver() {
        public boolean imageUpdate(Image img,
                int infoflags, int x, int y,
                final int width, final int height
        )
        {
                if ((infoflags & ALLBITS) == 0)
                    return true;

                // Bild speichern
                IMP.writeImage(view3d,
                    (RenderedImage) img,
                    new MimeType("image/png", null),
                    new File(pathAndFileName));

                // Aufruf für die nächsten Aktionen.
                // Im lpc gekapselt.
                view3d.getWorkbench().getJobManager().runLater(lpc,
                    null, view3d, 0);

                // Bild ist fertig
                return false;
        }
);
```

Abbildung 5.51: Festlegen, was nach einem Render-Vorgang passieren soll.

Der Aufruf für das Rendern des nächsten Bildes steckt im *lpc*, einer Instanz der Klasse *LockProtectedCommand*. Diese Klasse dient dazu, Befehle zur Laufzeit zu kapseln und zu übergeben. Das *lpc*-Objekt wird dem *JobManager* übergeben. Dabei wird ihm mitgeteilt, dass alle Befehle innerhalb von *lpc* später ausgeführt werden sollen. Der Grund ist, dass alle Aktionen im Prozess des Renderers laufen, der gerade das Bild gerendert hat. Würde der nächste Render-Vorgang starten und man würde den alten Render-Vorgangs beenden, würde auch der neue beendet werden, da er ein Kind-Prozess des alten Render-Vorgang ist. Um das zu vermeiden und den alten Prozess trotzdem zu beenden, wird der neue Prozess vorgemerkt und zu einem späteren Zeitpunkt gestartet. Abbildung 5.54 zeigt eine mögliche Implementierung des *LockProtectedCommand*.

### Billboard – Architektur in GroIMP

Da sich die Abfolge der auszuführenden Befehle bei der Erstellung eines *Billboard*s oft ähnelt (rotiere Kamera; rendere Ansicht; speichere Bild), wurde eine Klasse *Billboarder* erstellt. Von ihr wird jedes implementierte *Billboard*-System abgeleitet und dann in den speziellen *Billboard*-Klassen erweitert. So ist in der Klasse *Billboarder* der Aufruf der

```
LockProtectedCommand lpc;

lpc = new LockProtectedCommand(view3d.getGraph(),true,0)
{
        protected void runImpl(Object arg, de.grogra.pf.ui.Context c)
        {
                // Kamera um 20° nach links rotieren
                camera.moveAroundZ(20)

                // Naechstes Bild rendern
                billboarding();
        }
};
```

Abbildung 5.52: Kapseln von Befehlen, zur späteren Ausführung.

Rotations-Kamera sowie der Aufruf des Renderers realisiert. Diese Klasse ist abstrakt, da sie eine abstrakte Methode *setRunImpl()* besitzt. Jede Klasse eines *Billboard*-Systems muss diese Methode implementieren. Sie setzt das *lpc*, also die Abfolge der Befehle, die nach einem fertig gerenderten Bild ausgeführt werden sollen. Die Implementierung dieser Methode in der Klasse *BasicBillboarder* setzt den Aufruf zum Rotieren der Kamera auf die nächste Position sowie den Aufruf, die neue Ansicht zu rendern.
In der Klasse *Billboarder* ist eine Methode *initialize()* angelegt. Da sie vererbt wird, steht sie jeder anderen *Billboard*-Klasse zur Verfügung. Diese Methode muss nach jeder Initialisierung aufgerufen werden. Mit dem Aufruf und den übergebenen Parametern wird definiert, wie weit die Kamera vom Objekt entfernt sein soll, welche Höhe sie hat und, das Wichtigste, wie viele Seiten erstellt werden sollen. Mit der Angabe der Seite (*sides*) steht auch implizit fest, wie groß der Winkel zwischen zwei aufeinanderfolgenden Perspektiven sein muss. Dieser Winkel (*RotationAngle*) wird wie folgt berechnet:

$$RotationAngle = \frac{360°}{\text{sides}}$$

Da die Klasse *RotationCamera* nur mit absoluten Winkeln arbeitet, muss sich also die *Billboarder*-Klasse merken, welche Seiten bereits gerendert wurden. Aus der derzeitigen Seite (*currentSide*) und dem *RotationAngle* lässt sich dann der Winkel berechnen, mit dem die Kamera rotiert werden soll. Da sie immer wieder von der Start-Position aus rotiert wird, ergibt sich der Winkel für die aktuelle Ansicht aus der Multiplikation von *RotationAngle* und *currentSide*. Diese Berechnung geschieht in der Methode *nextSide()*, wo dann der aus der Multiplikation resultierende Winkel der *RotationCamera* durch *moveAroundZ()* mitgeteilt und ausgeführt wird. So ist der Aufruf, um auf die nächste Position zu rotieren, sehr einfach, da keine weiteren Parameter beim Aufruf an die Methode zu übergeben sind.
Der Aufruf des Renderes ist für alle von *Billboarder* abgeleiteten Klassen gleich. Daher ist er auch in dieser implementiert. Innerhalb der Methode wird auch die Zählung der

bereits gerenderten Seiten-Ansichten vorgenommen. Nach jedem berechneten Bild wird *currentSide* um eins erhöht.

Eine weitere Methode der abstrakten Klasse ist *setDestination()*. Durch die Übergabe eines *File*-Objektes wird der Pfad angegeben, in welchem die Bilder gespeichert werden sollen. Der mit anzugebende *Mime-Type* legt das Format der zu speichernden Datei fest. Die Formate *JPG*, *GIF*, *PNG* und *OPEN-EXR* werden unterstützt. Jedoch für die Darstellung in VRML wird ausschließlich das *PNG*-Format genutzt, da es das gängigste Format ist, welches auch Transparenz mit speichert und von den meisten VRML-Betrachtern auch so dargestellt wird.

Wie bereits erwähnt, werden die einzelnen Klassen für die drei eingeführten *Billboard*-Systeme von der Klasse *Billboarder* abgeleitet. Die abgeleiteten Klassen erzeugen lediglich die Bilder. Für das Erstellen der VRML-Datei kommt zu der jeweiligen Billboard-Klasse eine VRML-Klasse hinzu. Sie erbt von dem dazugehörigen *Billboard*-System und implementiert die Methode *writeVRML()* des Interfaces *VRMLBuilder*s. Da *Billboard*-Systeme nicht nur in VRML zum Einsatz kommen, wurde deswegen eine Trennung der Bilderstellung für das *Billboard*-System von dem Erstellen der entsprechenden VRML-*Billboard*-Datei vorgenommen. Alle Klassen, die die Bilder erstellen, haben am Ende des Klassen-Namens das Suffix *Billboarder*, da sie auch von der Klasse *Billboarder* direkt erben. Die Klassen, die für die Erzeugung der VRML-Datei eines *Billboard*-Systems zuständig sind, beginnen mit dem Präfix *VRML* und Enden mit dem Suffix *Builder*. Der Mittelteil des Namens entspricht der englischen Bezeichnung des jeweiligen Systems. Das daraus resultierende Klassendiagramm ist in Abbildung 5.53 dargestellt.

#### Generierung der VRML-Dateien

Da unterschiedlich viele Bilder auf *Billboard*s angezeigt werden können, muss auch der Inhalt der VRML-Datei flexibel anpassbar sein. Dazu sind GroIMP-intern so genannte Templates angelegt. Das sind VRML-Dateien mit einem vorgegebenem Rahmen. Es muss nur noch die Anzahl der Bilder und für jedes Bild eine Instanz des definierten Prototyps angelegt werden. Dies geschieht mit der *String*-Operation *replaceAll()*. In dem Template gibt es Platz-Halter, die mit entsprechendem Inhalt aufgefüllt werden. So wird für den Platzhalter *::PNG_PROTO_CALL::* die Liste aller Prototyp-Instanzen eingesetzt.

#### Erstellung der dynamischen Billboards

Dies geschieht mit der Klasse *VRMLDynamicBuilder*. Sie erbt von der Klasse *Basic-Billboarder*. Dieses *Billboard*-System hat keine besondere Anforderung an die Implementierung. Es wird lediglich die Kamera rotiert und bei jedem Schritt ein Bild gerendert. Daher ist die Implementierung der *BasicBillboarder* relativ simpel. In der Methode *setRunLater()* dieser Klasse wird das *lpc* gesetzt. Sie kapselt die Befehle für das Setzen der Kamera auf die nächste Position und für das Starten eines neuen Render-Prozesses.

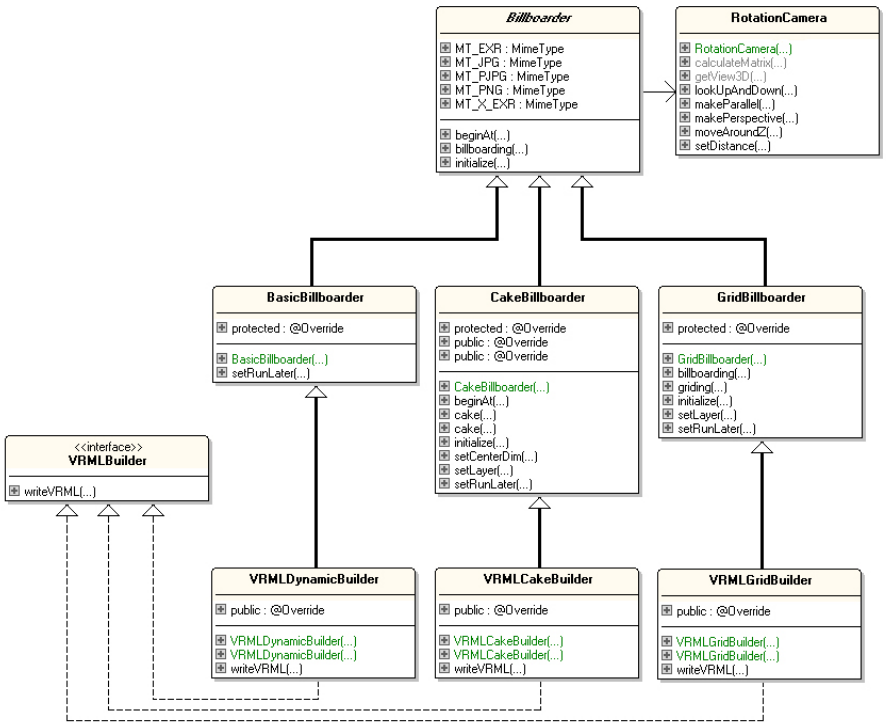


Abbildung 5.53: Klassendiagramm der *Billboarder*-Architektur in GroIMP.

Bei der Erzeugung einer Instanz *dynbb* der Klasse *VRMLDynamicBuilder* wird ein *File*-Objekt und der dazugehörige *MimeType* angegeben. Wie schon erwähnt wird, dies in den meisten Fällen das *PNG*-Format sein.
Danach wird auf dem Objekt *dynbb* ein *initialize()* aufgerufen. Es wird dem *Billboarder* der Abstand der Kamera zum Objekt und die Höhe der Kamera über dem Grund mitgeteilt. Außerdem wird festgelegt, wie viele Bilder erstellt werden sollen. Danach kann bereits *billboarding()* aufgerufen werden. Ab jetzt läuft alles automatisch. Sind alle Seiten-Ansichten gerendert, wird die VRML-Datei erstellt. Abbildung 5.54 zeigt die Art und Weise der Verwendung des *VRMLDynamicBuilder*s.

Die Aufnahmen in Abbildung 5.55 stammen von einem dynamischen *Billboard*-System mit 72 Seiten. Bei dieser Darstellung sind die Sprünge zwischen 2 Perspektiven nicht sehr groß, jedoch sichtbar. Dadurch wird der plastische Eindruck, der entstehen soll, vor allem bei langsamen Bewegungen um den Baum, gestört. Eine höhere Anzahl von Ansichten würde dem entgegenwirken. Mehr Ansichten bedeutet jedoch auch mehr Speicherplatz-Verbrauch.

```
import de.grogra.billboard.*;

public void run () { ... }

public void erstelleBillboards()
{
        MimeType mt;
        VRMLDynamicBuilder dynbb;

        mt      = VRMLDynamicBuilder.MT_PNG;
        dynbb   = new VRMLDynamicBuilder("Pfadangabe", mt);

        // Setzt den Abstand auf 20E zum Objekt
        // Verschiebt die Kamera um 4E nach oben
        // Macht ein 35-Bilder-Billboard
        dynbb.initialize(20, 4, 35);

        dynbb.billboarding();
}
```

Abbildung 5.54: Durch den Aufruf *erstelleBillboards()* wird ein dynamisches *Billboard*-System für VRML erstellt und gespeichert.

### Erstellung des Fächer-Billboards

Dieses System ist in der Klasse *CakeBillboarder* implementiert. Die Klasse *VRML-CakeBuilder* erbt von ihr. Im Gegensatz zur Erstellung dynamischer *Billboard*s liegt der Schwerpunkt der Implementierung nicht in der VRML-Datei in Form eines *Script*-Knotens, sondern im Erstellen der Bilder. Denn es müssen Elemente des Objektes ein- und ausblendbar sein, da, wie in Abschnitt 3.3.2 beschrieben, nicht alle Objekte aus einer Perspektive sichtbar sein dürfen. Die Erstellung der VRML-Datei reduziert sich dadurch auf das Erzeugen eines Fächers von Rechtecken mit der entsprechenden Textur.

Um Teile von Objekten ein- und auszublenden, bietet sich die Nutzung der *Layer* in GroIMP an. Diese können über die Benutzeroberfläche im Menü *View* → *Layer* ein- und ausgeschaltet werden. Jeder sichtbare Knoten, also ein Element des Objekts, wird bei der Erstellung einem *Layer* zugewiesen. Den *Layer* kann man auch im Nachhinein ändern. Wird nun ein Knoten in einen *Layer* gelegt, der ausgeschaltet wurde, wird beim Rendern eben dieser Knoten nicht beachtet, er ist ausgeblendet. Dies kann wieder rückgängig gemacht werden, indem der Knoten auf einen sichtbaren *Layer* gelegt wird.
Wann ein Element sichtbar ist und wann nicht, muss für jede Perspektive neu berechnet werden. Dies übernimmt die Methode *cake()* in der Klasse *CakeBillboarder*. GroIMP bietet die Möglichkeit, einzelne Objekte als Elemente von Mengen zu betrachten. So ist es möglich, Mengen-Operationen wie Vereinigung oder Schnitt durchzuführen.



Abbildung 5.55: Dynamisches *Billboard*-System im Einsatz: Das *Billboard*-System besteht aus 72 Seiten-Aufnahmen.

Im Paket *de.grogra.vecmath.geom* gibt es die Klasse *HalfSpace* (*HS*). Übersetzt bedeutet der Name „halber Raum“. Die Klasse dient dazu, den Raum in zwei Bereiche einzuteilen. Die Aufteilung des Raumes geschieht durch Angabe eines Ursprungs und der Richtung des *HS*s. Durch diese Angaben entsteht eine Trennung des Raums entlang einer gedachten Ebene. Nun kann für jedes Element ermittelt werden, ob es INNERHALB oder AUSSERHALB *HS* liegt.
Mit Hilfe der Angabe der Richtung des *HalfSpace* in Form eines Vektors kann nun eine Rotation um den Ursprung realisiert werden. Da die Rotation um die z-Achse erfolgt ist der z-Wert des Vektors 0. Die Werte für x und y berechnen sich wie folgt: Aus der Multiplikation des Rotations-Winkels für einen Rotations-Schritt *RotationAngle* und der aktuellen Seiten-Zahl *currentSide* ergibt sich ein Radiant. Für den x-Wert des Vektors wird dann der Sinus und für den y-Wert der Kosinus berechnet. Durch die Änderung des Wertes *currentSide* ändert sich somit auch die Richtung des *HalfSpace*s.
Da die Angabe INNERHALB oder AUSSERHALB allein nicht ausreicht, um einen bestimmten Bereich von Elementen einzugrenzen, bedarf es eines weiteren *HalfSpace*. Durch Angabe eines anderen Richtungsvektors bei identischem Ursprung für den neuen *HS* entstehen zwischen den beiden halben Räumen zwei neue Zwischen-Räume. In Abbildung 5.57 ist die Raum-Aufteilung demonstriert. Die Schraffur mit jeweiliger Farbe symbolisiert dabei das INNERHALB eines Raumes. Das bedeutet, dass Objekte, die auf der roten Schraffur liegen, im *HalfSpace* 1 liegen. Durch die Rotation gibt es Bereiche im Raum, die *HS* 1 und *HS* 2, jeweils nur einem halben Raum, oder gar keinem der beiden halben Räume angehören.
Damit jedes Element des Objektes auch mindestens einmal in einem solchen Zwischen-Raum (aktiver Sektor) liegt, muss der Winkel zwischen beiden Räumen exakt dem eines Rotations-Schrittes (also *RotationAngle*) entsprechen. Außerdem ist es wichtig, dass die Kamera die aktiven Sektoren von der Seite aus betrachtet. Deswegen wird aus Sicht der Kamera *HalfSpace* 1 immer um die Hälfte von *RotationAngle* nach hinten und *HalfSpace* 2 um die Hälfte nach vorn rotiert. Somit ergibt sich für die Rotation der beiden Räume

```
HalfSpace hs     = new HalfSpace();

// Winkel, mit dem HalfSpace rotiert werden soll
double angle     = RotationAngle * currentSide;

// Radianten daraus berechnen
double radiant   = Math.toRadians(angle);

// Ursprung des halben Raums
Point3d origin   = new Point3d(0.0,0,0.0);

// Richtung des halben Raums
TVector3d axis   = new TVector3d(Math.sin(radiant),
                                 Math.cos(radiant),0);

// Rotation dem HS zuweisen
hs.setTransformation(origin, axis);
```

Abbildung 5.56: *HalfSpace* wird um den gleichen Betrag wie die Kamera rotiert.

die Implementierung in Abbildung 5.58.
Durch das Einbeziehen der Variablen *RotationAngle* und *currentSide* existiert nun eine direkte Abhängigkeit zur Position der Kamera, da der Rotations-Winkel wiederum abhängig von den Variablen ist. Nun sollen alle Objekte in dem neuen Zwischen-Raum als sichtbar, alle anderen als unsichtbar gesetzt werden.
Betrachtet man das Problem aus Sicht der Mengenlehre und sieht beide Räume als Mengen an, kann der Zwischen-Raum durch die symmetrische Differenz-Menge der beiden Mengen gebildet werden. Sie erhält man durch den Einsatz eines sogenannten exklusiven Oder (XOR). Es besagt, dass Elemente genau einem *HalfSpace* angehören. Also entweder *HalfSpace* 1 ($Hs1$) oder *HalfSpace* 2 ($Hs2$). Elemente, die in beiden Mengen oder gar keiner liegen, werden dadurch ausgeschlossen.

Mit der Definition für die Zwischen-Räume kann nun für jedes Elemente geprüft werden, ob es sich in diesem Raum befindet. Dazu bietet die Klasse *HalfSpace* die Methode *contains()* an. Mit der Angabe eines Punktes gibt die Funktion ein *true* zurück, wenn er innerhalb des *HalfSpace*s liegt. Liegt der Punkt außerhalb, wird *false* zurückgegeben. Somit kann nun für ein Element des Objekts durch die Implementierung in Abbildung 5.59 geprüft werden, ob es sich im aktiven Sektor befinedet.
Die Angabe des Punktes erfolgt durch die Abfrage der Position des Elementes. Dies geschieht mit der Funktion *location()* der Klasse *Library*, der das Element als Parameter übergeben wird.

Nachdem die Position der Elemente im Raum bestimmt wurde, können nun die Ele-

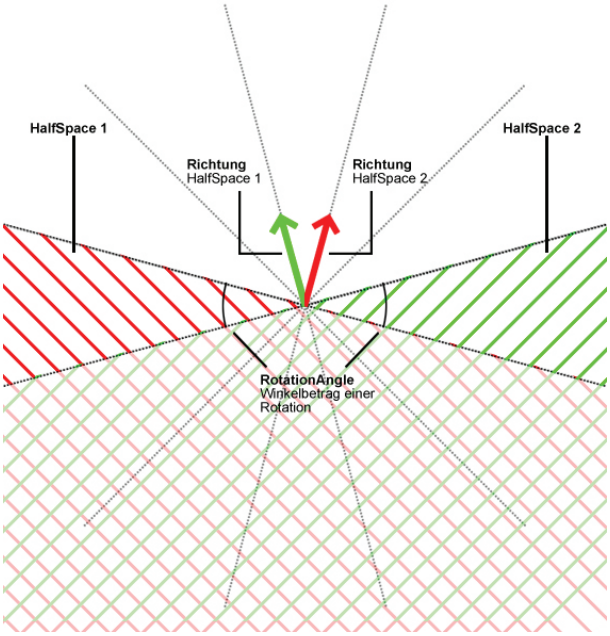


Abbildung 5.57: Einteilung des Raums in Sektoren.

mente, die in einem der beiden Zwischen-Räume liegen, als sichtbar gesetzt werden, alle anderen als unsichtbar. Dies geschieht mit den lokalen Variablen *visibleLayer* für die sichtbaren Elemente und *invisibleLayer* für alle unsichtbaren Elemente. Je nach Lage des Elementes wird einer der beiden Werte der Variablen als der neue *Layer* für das Element genutzt. Welche *Layer* für die sichtbaren und welche für die unsichtbaren Elemente genutzt werden, kann mit der Methode *setLayer()* der Klasse *CakeBillboarder* vorher festgelegt werden. Als Standard-Wert ist *Layer* 0 für die sichtbaren und *Layer* 15 für die unsichtbaren gesetzt.

Neben den beiden *HalfSpace*s kommt noch das Konstrukt *Line* zum Einsatz. Der Grund ist in Abbildung 5.57 zu sehen. Der Raum in der Mitte des Objektes ist dort sehr dünn. Dies kann unter Umständen bedeuten, dass hier gar keine Elemente als sichtbar betrachtet werden könnten. Damit keine Lücke in der Mitte des darzustellenden Objektes entsteht, wird eine Linie gezeichnet, die vom Objekt-Ursprung nach oben zeigt. Elemente, die zwar nicht im Zwischen-Raum, aber innerhalb eines vorgegebenen Abstandes zu der Linie liegen, werden trotzdem als sichtbar gesetzt. Somit ist die Mitte bis zu einem bestimmten Abstand immer sichtbar. Dieser Abstand sollte aber nicht zu groß gesetzt werden, da sonst das in 3.3.2 beschriebene Problem auftritt, dass die gleichen Elemente aus mehreren Perspektiven dem Betrachter zeitgleich von verschiedenen Seiten erscheinen. Die Abfrage, ob ein Objekt innerhalb eines gegebenen Abstandes liegt, geschieht mit der Überprüfung, ob die Distanz der Position des Objektes kleiner als ein vorgegebener Wert ist. Die Distanz vom Objekt zum Punkt gibt die Funktion *distance()* der

```
HalfSpace hs1    = new HalfSpace();
HalfSpace hs2    = new HalfSpace();

// Winkel, mit dem HalfSpace rotiert werden soll
double angle             = RotationAngle * currentSide;

// Halber Rotations-Winkel-Schritt
double halfangle         = RotationAngle/2;

// Aus Sicht der Kamera ...
// ... nach hinten gedreht.
double radiant1          = Math.toRadians((angle - halfangle));

// ... nach vorn gedreht.
double radiant2          = Math.toRadians((angle + halfangle));

Point3d origin           = new Point3d(0.0,0,0.0);

TVector3d axis1          = new TVector3d(Math.sin(radiant1),
                                         Math.cos(radiant1),0);
TVector3d axis2          = new TVector3d(Math.sin(radiant2),
                                         Math.cos(radiant2),0);

hs1.setTransformation(origin, axis1);
hs2.setTransformation(origin, axis2);
```

Abbildung 5.58: Zwei *HalfSpace* erzeugen einen Zwischen-Raum.

Klasse *Line* zurück.
Der Abstand von der Linie zum Element gilt in x- sowie y-Richtung. Betrachtet man nun den Abstand als Radius und geht die Linie entlang, beschreibt sie einen Raum in zylindrischer Form, einen Quasi-Zylinder. Dieser kann mit der Methode *setCenterDim()* der Klasse *CakeBillboarder* durch Angabe von Abstand und Höhe definiert werden. So ist *centerWidth* der maximale Abstand zur Linie und *centerHeight* die Länge der Linie.

Nun kann mit einer Traversierung durch den gesamten Graphen für alle Elemente ermittelt werden, ob sie bei der aktuellen Seiten-Ansicht sichtbar oder unsichtbar sind. Das Ergebnis dieser Form der *Billboard*-Erzeugung ist in Abbildung 5.60 zu sehen.

Durch das Ein- und Ausblenden von Elementen aus bestimmten Perspektiven kommt es jedoch später zu einer verfälschten Darstellung des Objektes im VRML-Betrachter. So hängen einige Elemente scheinbar in der Luft, da sie keinerlei Verbindungen zum „Rest“ des Objektes besitzen. Dies hat den Grund, dass zum Beispiel ein Blatt im sichtbaren Sektor liegt, jedoch der Ast, an dem das Blatt hängt, aus der aktuellen Perspektive nicht.

```
// Element des Objektes
NullWithShader n;
Point3d location = Library.location(n);
...
// Ein XOR wird durch "^" dargestellt
if(      ( hs1.contains(location, true) ^
           hs2.contains(location, true )) ||
         ( line.distance(location) < centerWidth ))
{
                  // Sichtbar
                  n.setLayer(visibleLayer);
} else {
                  // Unsichtbar
                  n.setLayer(invisibleLayer);
}
```

Abbildung 5.59: Abfrage, ob ein Element des Objektes sichtbar ist.



Abbildung 5.60: Darstellung der Erle mit einem 72-Seiten Fächer-*Billboard*-System.

Das Auswählen der sichtbaren Elemente hat einen gravierenden Nachteil: Beim Erstellen von geometrischen Objekten in GroIMP wird bei einem Punkt mit dem Zeichnen begonnen und in einem anderen Punktgeendet. Die Abfrage der Position eines Elementes des Objekts bezieht sich lediglich auf die Startposition. Das bedeutet, dass zum Beispiel die Startposition eines Zylinders im sichtbaren Bereich, jedoch der Rest außerhalb liegt. Dennoch wird der Zylinder vollständig angezeigt. Elemente, die wiederum zum Großteil im sichtbaren Bereich liegen, für die jedoch die Startposition außerhalb liegt, werden dagegen nicht angezeigt. Das heißt, dass Elemente an der Grenze von sichtbaren und unsichtbaren Bereichen nicht abgeschnitten werden, sondern immer als ganze Einheit aus- oder eingeblendet werden.
Ein weiteres Problem kommt hinzu, wenn die zu erstellenden *Billboard*s keine gleichmäßige Beleuchtung erfahren. Licht- und Schattenverhältnisse würden mit dem Ausgangs-

modell nicht mehr übereinstimmen, denn ausgeblendete Elemente verursachen keinen Schatten mehr. So gelangt Licht an Stellen, die im Original-Zustand im Schatten gelegen hätten. Dadurch wirkt die Darstellung der Erle im Vergleich zum dynamischen *Billboard*-System heller.
Der Vorteil an dieser Methode ist jedoch, dass sie mit wenig Seiten-Ansichten für die VRML-Darstellung auskommt. Außerdem werden durch das Ausblenden die zu rendernden Elemente kurzzeitig reduziert, was das Rendern wiederum beschleunigt.

Bei dieser Form der Darstellung entsteht in der Mitte des *Billboard*s ein freier Raum. Dies hat zum Einen den Grund, dass der Baum drei Hauptstämme hat, die sich vom Zentrum wegbewegen. Zum Anderen verschwinden Texturen auf Rechtecken, wenn sie von der Seite betrachtet werden. Somit erscheint für den Betrachter eine Lücke in der Mitte des Baumes.
Im Vergleich zum dynamischen *Billboard* gibt es keine Sprünge zwischen 2 Perspektiven, da hier kein Bildwechsel auf der Textur erfolgt. Somit entsteht ein „fließender“, dreidimensionaler Eindruck bei einem „Rundflug“ um das Objekt.

#### Erstellung des Raster-Billboards

Genau wie das Fächer-*Billboard*-System nutzt auch das Raster-*Billboard*-System die Möglichkeit, Elemente ein- und auszublenden. Jedoch wird dabei nicht von einer Einteilung des Raums in Torten-Stücke ausgegangen, sondern von Scheiben, die den Raum aufteilen. Jede dieser Scheiben beinhaltet Elemente. Elemente einer Scheibe, die als aktiv bezeichnet wird, sind sichtbar, alle anderen Elemente nicht.

Bei der Initialisierung eines Objektes der Klasse *GridBillboarder* kommt neben dem Abstand der Kamera zum Objekt und der Höhenverschiebung die Angabe der Anzahl der Scheiben hinzu, die angibt, in wie viele Scheiben das gesamte Objekt pro Seiten-Ansicht eingeteilt werden soll. Somit ist, aus Sicht der Kamera, die Tiefe einer Scheibe festgelegt, da vorher bestimmt wird, welche Ausdehnung das gesamte Objekt im Raum hat.
Die Ausdehnung des Objektes wird zunächst ermittelt. Mit einer Traversierung durch den gesamten Graphen wird von jedem sichtbaren Objekt das Volumen abgefragt und in ein Gesamt-Volumen eingefügt. Dieses vergrößert sich somit um jedes neu hinzugefügte Element. Nachdem alle Volumen des gesamten Graphen hinzugefügt wurden, können die Ausmaße des Gesamt-Volumens ermittelt werden. Wie dies umgesetzt ist, zeigt Abbildung 5.61. Durch die Ausmaße im Raum ist nun auch die Breite (in x-Richtung) und Tiefe (y-Richtung) des Objektes bekannt. Dies ist wichtig, da die Breite beziehungsweise Tiefe der Scheiben von der Breite / Tiefe des gesamten Objektes abhängt. Die Scheibenbreite und -tiefe wird nach der Ausmaß-Berechnung in der Methode *initialize()* durchgeführt. Es muss dabei eine Unterscheidung von Tiefe und Breite getroffen werden, da Objekte in der Regel eine unregelmäßige Ausdehnung in alle Richtungen haben.

Die Besonderheit bei dieser Implementierung des *Billboard*-Systems ist, dass von einer

```
// Gesamt-Volumen
SimpleUnion cv  = new SimpleUnion(){};

// Traversierung
while (!stack.isEmpty ())
{
        ...
        // Volumen hinzufuegen
        cv.volumes.add( Library.volume(n));
        ...
}

// Ausmaße des Gesamtvolumens ermitteln
cv.getExtent(min, max, new Variables());

// Breite des Objekts
double xWidth   =       Math.abs(max.x - min.x );

// Tiefe des Objekts
double yWidth   =       Math.abs(max.y - min.y);

// Breite einer Scheibe auf der X-Achse
double stepWidthX       = xWidth/steps;

// Breite einer Scheibe auf der Y-Achse
double stepWidthY       = yWidth/steps;
```

Abbildung 5.61: Ausmaße des gesamten Objekts ermitteln.

Seitenansicht mehrere Bilder gerendert werden. Und zwar geschieht das in der Form, dass mit der von der Kamera aus hintersten Scheibe begonnen wird. Elemente, die in dem Bereich dieser Scheibe liegen, werden ein-, alle anderen ausgeblendet.
Die Umsetzung nutzt, wie auch bei der Fächer-Implementierung, zwei *HalfSpace*s. Nur sind sie diesmal parallel zur Kamera ausgerichtet. Bei der Berechnung der ersten Scheibe wird ein, von der Kamera aus betrachtet, *HalfSpace* (HS1) genau am hintersten Ende des Objektes angelegt. Ein weiterer *HalfSpace* (HS2) wird ebenfalls parallel zur Kamera angelegt. Jedoch wird dieser um genau eine Scheibenbreite beziehungsweise -tiefe nach vorn zur Kamera hin verschoben. So entsteht ein Korridor. In ihm liegen einige Elemente des Objektes. Diese erfüllen die Bedingung *hs1.contains(location, true))* UND NICHT *hs2.contains(location, true))*. Das bedeutet, dass alle Elemente als innerhalb und somit sichtbar betrachtet werden, wenn sie in HS1 liegen, jedoch nicht in HS2. So sind alle Objekte vor und hinter einer aktiven Scheibe als unsichtbar zu setzen.

Die Abbildung 5.62 zeigt die Lage der *HalfSpace*s in zwei verschiedenen Phasen der Erzeugung des *Billboard*s. Die Einteilung des vom Objekt eingenommenen Raumes ist, von

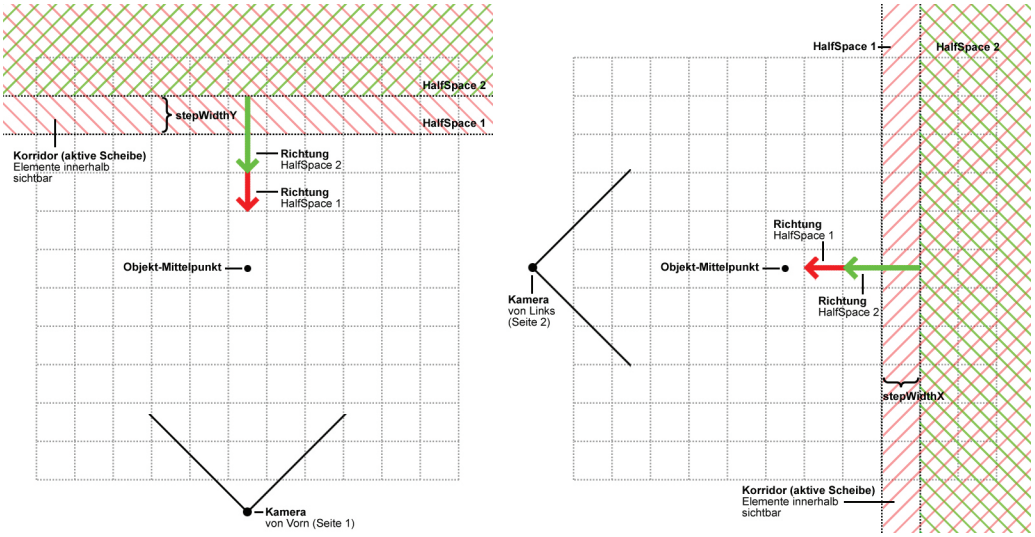


Abbildung 5.62: Beispiel einer 11x11 Rasterung: Das linke Bild zeigt die Situation beim Erstellen der zweiten Scheibe der ersten Seite (Vorderseite). Auf dem Bild rechts wird gerade die zweite Seite mit der dritten Scheibe betrachtet.

oben betrachtet, durch vertikale und horizontale Grenzen gegeben. So entstehen mehrere Rechtecke, die, wenn sie zusammen auf einer vertikalen oder horizontalen Linie liegen, eine Scheibe im Raum bilden. Bei der Betrachtung des Objektes von vorne und hinten, was den Seiten 1 und 3 entspricht, werden nur die horizontalen Scheiben betrachtet. Blickt die Kamera von links oder rechts (Seiten 2 und 4) auf das Objekt, werden nur die vertikalen Scheiben genommen.
Die Reihenfolge, in der die entsprechenden Scheiben ein- und ausgeblendet werden, hängt in erster Linie von der Position der Kamera ab, da aus ihrer Sicht stets mit der jeweils entferntesten Scheibe begonnen wird. Der Korridor wird dann bei jedem Schritt um eine Scheiben-Breite von hinten nach vorne verschoben. Der letzte Korridor einer Seite ist somit die Scheibe direkt vor der Kamera.

Da für jede Seite mehrere Scheiben getrennt gerendert werden, ist es notwendig, neben dem Zählen der einzelnen schon gerenderten Seiten auch die bereits gerenderten Scheiben zu zählen. Denn erst, wenn alle Scheiben einer Seite erstellt wurden, wird die Kamera auf die nächste Position rotiert und der Vorgang beginnt von Neuem.
Dazu dient *stepCount* in der Klasse *GridBillboarder*. Mit jeder erstellten Scheibe wird dieser Zähler um eins erhöht, bis er die vorgegebene Anzahl von Scheiben erreicht hat. Die Anzahl wurde durch den Parameter *steps* in der Methode *initialize()* dieser Klasse festgelegt. Nach dem Erreichen des vorgegebenen Wertes wird die Kamera um 90° nach links gedreht und der Zähler für die Scheiben zurückgesetzt.
Mit Hilfe dieses Zählers werden auch die zwei *HalfSpaces* entsprechend im Raum positioniert. Die Ausgangspunkte für die Positionierungen sind stets die äußersten Stellen

der Objekt-Ausdehnung im Raum. Für die jeweilige Seite gibt es dadurch folgende Ausgangspunkte:

- Vorne (Seite 1): Der maximale y-Wert, also (0, *max.y*, 0)
- Links (Seite 2): Der maximale x-Wert, also (*max.x*, 0, 0)
- Hinten (Seite 3): Der minimale y-Wert, also (0, *min.y*, 0)
- Rechts (Seite 4): Der minimale x-Wert, also (*min.x*, 0, 0)

Bei der ersten Scheibe der ersten Seite ergibt sich somit die Position des HS1 aus dem y-Wert des Punktes *max*. Von der Position des HS1 ausgehend, wird von nun die Position des HS2 berechnet. Der zweite HS ist immer eine Scheibenbreite (für Seite 1 wäre das *stepWidthY*), von der Kamera aus betrachtet, hinter dem ersten HS. Ist die aktuelle Scheibe berechnet, sollen beide HSs um eine Scheibenbreite nach vorn geschoben werden. Dies stellt die Multiplikation von *stepCount* und der jeweiligen Breite, in diesem Fall *stepWidthY*, sicher. Das Ergebnis der Multiplikation wird vom Ausgangspunkt abgezogen, da die HS auf die Kamera zu bewegt werden sollen. Das „Hinterherlaufen“ von HS2 wird durch das „Nachhintenschieben“ von HS2 um den Betrag der Scheibenbreite realisiert. Die Orientierung der beiden HS ist gleich: beide „schauen“ in die Kamera. Abbildung 5.63 zeigt eine mögliche Implementierung, um für die Vorderseite die HS zu positionieren. Für die anderen Seiten sieht das ähnlich aus, deshalb wird an dieser Stelle darauf nicht weiter eingegangen.

```
// Position fuer HS1
double pos1    = max.y - ((++stepCount) * stepWidthY);
Point3d origin1 = new Point3d(0.0, pos1, 0.0);

// Position fuer HS2
double pos2    = pos1 + stepWidthY;
Point3d origin2 = new Point3d(0.0, pos2, 0.0);

// Richtung ist fuer beide HS gleich
TVector3d axis  = new TVector3d(0, -1, 0);
```

Abbildung 5.63: Positionsberechnung für die *HalfSpace*s.

Abbildung 5.64 zeigt das *Billboard*-System im Einsatz.

#### Bewertung der einzelnen Billboard-Systeme

Die drei vorgestellten *Billboard*-Systeme nutzen unterschiedliche Prinzipien. Dadurch ergeben sich bei der Anwendung Vor- sowie Nachteile, diese werden in Abschnitt 6.1 näher besprochen.

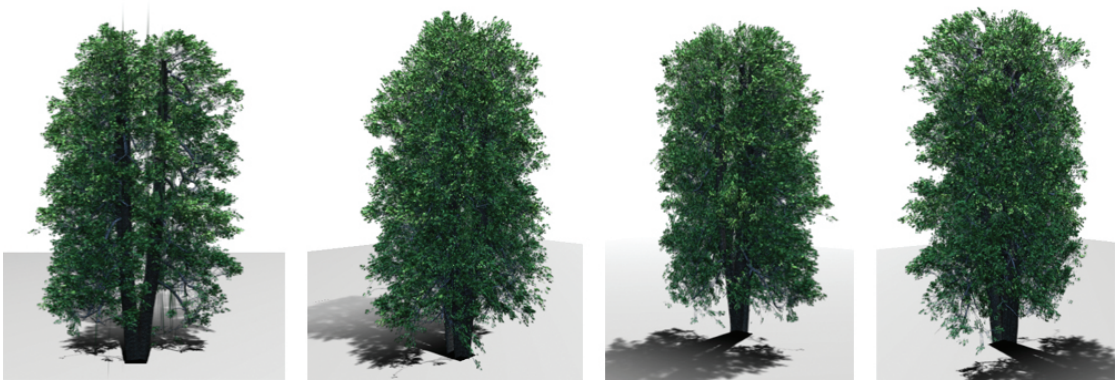

Abbildung 5.64: Raster-Darstellung der Erle mit einem 9x9-Raster.

## 5.4 Möglichkeiten für den Einsatz von GroIMP

Bislang wurde GroIMP als Werkzeug für eine Zustandsmodellierung genutzt. Da die Plattform auf Basis von XL viele Fähigkeiten besitzt, um 3D-Modelle auf Regeln basierend zu erzeugen, werden nun weitere mögliche Einsatzgebiete besprochen. Für fehlende Funktionen werden mögliche Lösungsansätze mitgegeben.

**Simulation von verschiedenen Zeitpunkten** Das Beispiel der Erle in GroIMP hat gezeigt, dass es möglich ist, auf einer Bildvorlage basierend, ein 3D-Modell mit Hilfe von Regeln zu erzeugen. Dabei wurde bislang jedoch nur ein fester Zeitpunkt der Ausprägung des Baumes angestrebt. Das bedeutet, dass die Entstehung der Erle vernachlässigt wurde. Daher ist es naheliegend, eben diesen Entstehungsprozess mit zu modellieren. Dadurch gibt es dann zu diesem Baum zu verschiedenen Zeitpunkten verschiedene Modelle. Für die Umsetzung verschiedener historischer Zustände des Parks wäre dies ein Ansatzpunkt.
Das bedeutet jedoch auch, dass ein größerer Aufwand zu betreiben ist. Es reicht nicht, den betroffenen Baum zu einem Zeitpunkt genau zu beobachten und umzusetzen, sondern es sollte nach Möglichkeit Aufnahmen von mehreren Zeitpunkten geben. Dies kann jedoch bereits ein Problem sein, wenn es darum geht, alte Bäume nachzumodellieren, wenn sie zu einer Zeit gepflanzt wurden, wofür nur unzureichend bis gar kein Bildmaterial vorhanden ist.

Zusätzlich ist eine wesentlich tiefere Untersuchung der Wachstumseigenschaften des Baumes notwendig, als dies bei der Erle geschehen ist. Während der Modellierung der Erle wurde das Erzeugen der räumlichen Strukturen in drei Prozesse aufgeteilt, was die Betrachtung und Steuerung erheblich vereinfachte. In einem Wachstumsmodell fällt diese Methode aus, da bei jedem Wachstumsschritt der gesamte Baum mit all seinen Strukturelementen zeitgleich betrachtet werden muss. Somit ist es schwieriger, das Wachstum so zu lenken, dass das Ergebnis der Vorlage möglichst nahe kommt.
So kann die Modellierung eines komplett ausgewachsenen Baumes schnell mehrere Mo-

nate dauern, zumal auch der Umgang mit GroIMP und der Sprache XL geübt werden muss. Auf der anderen Seite ist das Modellieren eines weiteren Baumes einfacher, wenn die Techniken in XL gelernt sind.

**Modellierung mit Umweltfaktoren** Eine weitere Möglichkeit, GroIMP einzusetzen, ist das Simulieren von Umwelteinflüssen und der dadurch auftretenden Veränderungen. Dabei geht es weniger um das Aussehen der einzelnen Bäume, als mehr um das Aufzeigen von Entwicklungstendenzen ganzen Baumbestände. Dies macht eine längerfristige Planung bei der Baumpflege möglich.

Speziell das Problem des Wasserverbrauchs der verschiedenen Baumarten im Branitzer Park ließe sich in GroIMP visualisieren. Mit Wasser muss in Zukunft sparsamer umgegangen werden, so dass die möglichst ökonomische Verteilung immer notwendiger wird.

Michael Henke hat in seiner Master-Arbeit das *Arrange*-Konstrukt eingeführt [7, Seite 70]. Es dient als Hilfsmittel bei der räumlichen Verteilung von vielen Objekten, wenn eine individuelle Positionierung zu aufwändig wäre. Die bereits existierende Baum-Karte des Branitzer Parks kann nun mit einer Karte, die die Wasserkonzentrationen im Park repräsentiert, kombiniert werden. Sind zu den Bäumen jeweils die Wasserverbrauchszahlen näherungsweise bekannt, kann in einem Modell nun simuliert werden, welche Auswirkungen dies auf die Baum-Vegetation mit veränderter Wasserkonzentration an bestimmten Stellen hat. Somit kann nun eine Prioritäten-Liste festgelegt werden, welchen Bäumen wie viel Wasser zugeführt werden sollte.

Das mit GroIMP auch großflächige Simulationen durchgeführt werden könnten, zeigte das Beispiel eines berechneten Waldes anlässlich der Präsentation für das Paper *The rule-based language XL and the modelling environment GroIMP, illustrated with simulated tree competition*[11]. Hierzu wurde auf einem Netzwerk von Rechnern ein Wald, bestehend aus den Baumarten Buche und Fichte erzeugt. Dabei wurde der Umweltfaktor Licht und dessen Einfluss beim Wachsen von Bäumen simuliert.

**Erzeugung von Baumgruppen** Durch die Anwendung von stochastischen Funktionen und Parametern zur Steuerung des Wachstum eines Baumes in GroIMP können die gleichen Wachstums-Regeln bei verschiedenen Aufrufen Variationen des 3D-Modells erzeugen. So gleicht kein Baum dem anderen, obwohl die gleichen Wachstumsregeln angewendet wurden.
Daher lassen sich Baumgruppen einer Art einfach erzeugen, da sie „nur“ einmal zu modellieren und an die einzelnen Bäume der Gruppe anzupassen sind. Jedoch kann auch das Anpassen der Parameter aufwändig werden, wenn der Unterschied zwischen zwei Bäumen zu groß ist. Das heißt, es muss abgewogen werden, ob nicht eine separate Modellierung sinnvoller wäre.
Ebenfalls ist denkbar, dass Bäume unterschiedlicher Art durch einen Baum repräsentiert werden und individuell leicht anpassbar sind.

**Verwendung der Billboard-Systeme** Extern erzeugte 3D-Modelle lassen sich auch in GroIMP importieren. So könnten die implementierten *Billboard*-Systeme auf bestehende 3D-Modelle angewendet werden. Um das Fächer- und Raster-*Billboard*-System auf externe 3D-Modelle anzuwenden, muss das Ein- und Ausblenden modifiziert werden. Das Problem ist, dass geometrische Objekte wie Zylinder und Parallelogramme betrachtet werden. Ein importiertes Objekt besteht jedoch in der Regel aus Polygonen.
Daher sollten in diesem Fall die Ein- und Ausblende-Mechanismen nicht mehr mit speziellen Objekten, sondern mit Polygonen arbeiten. Eine Möglichkeit bietet hierzu die Methode der *Constructive Solid Geometry* (*CSG*)[7, Seite 26] die auch in GroIMP in Form eines speziellen Knotentyps unterstützt wird. Mit Hilfe des *CSG*-Knotens können Elemente der Objekte ein- und ausgeblendet werden. Dabei spielt es keine Rolle, ob ein Objekt ausschließlich aus Polygonen oder auch aus Zylindern oder anderen geometrischen Primitiven zusammengesetzt wurde.

# 6 Auswertung

## 6.1 Vergleich und Bewertung der Billboard-Systeme

Die in Abschnitt 3.3 eingeführten *Billboard*-Systeme werden nun einzelnen nach Kriterien bewertet. Dazu müssen zunächst die Kriterien genannt werden, die der Bewertung zu Grunde liegen. Um eine Bewertung vornehmen zu können, wurden die Systeme an dem Beispiel der Erle aus Abschnitt 5.3.6 angewendet und entsprechende VRML-Szene-Dateien erzeugt, welche den Anhängen beiliegen und betrachtet werden können.

### 6.1.1 Kriterien

**Dreidimensionaler Eindruck** Durch *Billboard*s werden komplexe dreidimensionale Objekte als eine geringe Zahl von Rechtecken dargestellt. Eine derartige Vereinfachung der Geometrie kann jedoch den Verlust des räumlichen Eindrucks des Objektes herbeiführen. Ob die Repräsentation eher flächig oder räumlich wird, soll mit diesem Kriterium bewertet werden.

**Fehler bei der Darstellung** Da die Darstellung des 3D-Modells bei der Anwendung eines *Billboard*-Systems stark vereinfacht wird, gehen Informationen verloren. Das Fehlen dieser Informationen kann dazu führen, dass ein Modell nicht korrekt angezeigt wird. Ob und wie die Fehler auftreten und wie sehr sie den „Anblick“ des Objektes stören, wird mit diesem Kriterium bewertet.

**Speicherplatz** Die Systeme dienen in erste Linie dazu, Modelle zu vereinfachen. Da die Baum-Darstellungen auch internetfähig sein sollen, ist eine Abschätzung des Speicherplatzbedarfs notwendig. Denn die Größe des benötigten Speicherplatz lässt automatisch Rückschlüsse auf die Verzögerung bei der Übertragung zum Nutzer des virtuellen Parks zu. Das Problem des Speicherbedarfs nimmt an Bedeutung zu, da eine Vielzahl an Bäumen im virtuellen Park darzustellen sind.
Zusätzlich gibt es eine Auflistung des Speicherbedarfs der einzelnen *Billboard*-Systeme mit verschiedenen Vereinfachungsgraden.

**Anwendbarkeit** Die Umsetzung der *Billboard*-Systeme erfolgte ausschließlich in GroIMP. Es stellt sich die Frage, in wie weit die einzelnen Systeme in anderen Anwendungen umgesetzt werden können und ob eine Umsetzung mit fotobasierten Bildern möglich ist. Die Erzeugung der VRML-Datei wird bei diesem Kriterium vernachlässigt.

**Ausführbarkeit** Alle *Billboard*-Systeme müssen auch auf allen Systemen beziehungsweise VRML-Betrachtern funktionieren, damit eine einheitliche Betrachtung der virtuellen Szenerie möglich ist. Da eines der *Billboard*-Systeme nicht plattform-unabhängig funktionierte, muss dieses Kriterium der Vollständigkeit halber mit aufgeführt werden.

### 6.1.2 Bewertung der drei Billboard-Systeme

#### Dynamisches Billboard-System

**Dreidimensionaler Eindruck** Bei der Betrachtung dieses Systems bleibt ein räumlicher Eindruck bestehen, sofern die Kamera nicht von oben auf das System herabschaut. Dies liegt daran, dass der VRML-Knoten *Billboard* genutzt wird. Dieser richtet sein lokales Koordinaten-System immer zur Kamera hin aus. In diesem Fall werden Objekte um die senkrechte Achse (y-Achse) rotiert, so dass die Vorderseite zur Kamera zeigt. Dadurch ist eine Betrachtung von der Seite unmöglich ,und dies verhindert das Entstehen eines flächigen Eindrucks.
Bei einer gleichmäßigen Kreisbewegung um das System ist ein räumlicher Eindruck vorhanden. Die Übergänge zwischen den Perspektiven erscheinen ab 72 Bildern fließend, so dass das Objekt als ein 3D-Modell wahrgenommen wird. Dies gilt von der Nähe und aus der Ferne. Von allen drei Systemen ist der dreidimensionale Eindruck mit genügend Ansichten am stärksten bei diesem System.

**Fehler bei der Darstellung** Da bei der Erstellung der Bilder für die einzelnen Perspektiven keine Elemente ein- und ausgeblendet werden, sondern alle Elemente des Modells sichtbar sind, sind keine Fehler zu erkennen. Einzig eine in Bezug auf das System überhöhte Kameraposition verzerrt das auf dem Rechteck abgebildete Bild. Dies fällt aber nur bei genauerer Betrachtung von nahem auf. Bei der Betrachtung aus der Ferne besteht dieses Problem gar nicht mehr.
Dieses System erzeugt am wenigsten Darstellungsfehler.

**Speicherplatz** Alle aus einer Perspektive sichtbaren Elemente werden auf einer Ebene abgebildet und als ein Bild gespeichert. Dadurch braucht diese Variante den meisten Speicherplatz. Der Grund liegt darin, dass, um einen dreidimensionalen Eindruck zu erhalten, möglichst viele Ansichten existieren sollten und die Menge an Informationen pro Bild, in Vergleich zu den anderen beiden Systemen, maximal ist.
Werden doppelt so viele Ansichten erzeugt, verdoppelt sich auch der Speicherbedarf. Dies ist ein Nachteil gegenüber den anderen beiden Systemen, die für die optimale Darstellung weniger Speicher brauchen. Das bedeutet, dass dieses System weniger für die Nutzung im Internet geeignet erscheint als das Fächer- oder Raster-*Billboard*-System.
Speicherbedarf:

- 18 Seiten: 9,98 Megabyte
- 36 Seiten: 19,9 Megabyte
- 72 Seiten: 39,9 Megabyte

**Anwendbarkeit** Die Herausforderung, bei der Umsetzung dieses Systems ist das Einteilen der Kamera-Rotationsschritte. Das heißt, dass die Kamera um das darzustellende Objekt immer mit dem gleichen Winkel-Betrag rotiert werden muss, solange, bis um das Objekt 360° rotiert wurde. Dies ist in konventionellen 3D-Programmen möglich – entweder durch Definieren von Pfaden, die eine Kamera in gleichgroßen Schritten abschreiten soll, oder durch Nutzung von programminternen Scripten. Ein Beispiel ist MaxScript in 3D Studio Max.
Um fotobasierte Bäume mit diesem System darzustellen, ist der zu betreibende Aufwand zu groß. Der Abstand zum Objekt sowie die einzelnen Rotationsschritte müssen gleich bleiben. Zusätzlich darf die Sicht nicht gestört sein und alle Aufnahmen müssen zeitnah beieinander liegen, so dass die Lichtsituation ähnlich bleibt. Dies ist kaum umzusetzen, dennoch theoretisch möglich.

**Ausführbarkeit** Die Nutzung des Script-Knotens macht einigen VRML-Betrachtern Probleme. So konnten VRML-Betrachter für Linux den Script-Knoten nicht interpretieren. Die Folge war, dass kein Bild angezeigt wurde. Die einzigen VRML-Betrachter, die dieses System korrekt darstellten, waren BS Contact Player[1] sowie Cosmo Player[2]. Diese sind jedoch zur Zeit nur für Windows verfügbar.

### Fächer-Billboard-System

**Dreidimensionaler Eindruck** Von allen drei *Billboard*-Systemen ist der räumliche Eindruck dieses Systems am schlechtesten. Dies liegt daran, dass, anders als bei dem dynamischen System, Rechtecke von der Seite betrachtet werden können, denn sie werden nicht zur Kamera ausgerichtet. So werden Bilder der verschiedenen Perspektiven so verzerrt, dass ein flächiger Eindruck entsteht. Dieser Eindruck kann durch eine größere Anzahl von Seiten abgeschwächt werden, verschwindet aber nicht komplett.
Bei zunehmender Entfernung zum Objekt ist jedoch kein flächiger Eindruck wahrzunehmen, das Objekt erscheint als dreidimensionale Darstellung.

**Fehler bei der Darstellung** Der schwerwiegendste und auch auffälligste Fehler ist der Spalt in der Mitte. Dies liegt daran, dass ein Rechteck von der Seite betrachtet wird und die Textur in Form des Bildes dadurch so sehr verzerrt wird, dass sie nicht mehr sichtbar ist. Dadurch ist die Mitte des Baumes immer leer. Dies lässt sich mit einer höheren Anzahl von Seiten vermindern, verschwindet aber nicht ganz. Auch aus der Ferne ist der Spalt in der Mitte deutlich zu erkennen.
Ein anderer Fehler sind Elemente, die scheinbar in der Luft hängen. Dies liegt daran, dass Elemente des 3D-Modells abhängig von der Perspektive ein- und ausgeblendet werden. Werden zwei benachbarte Perspektiven in diesem System betrachtet, so entsteht bei einem bestimmten Betrachtungswinkel der Eindruck, dass eigentlich zusammengehörende Elemente nicht mehr miteinander verbunden sind. Dieser Eindruck verschwindet jedoch mit zunehmender Entfernung zum System.

[1] http://www.bitmanagement.com/
[2] http://cic.nist.gov/vrml/cosmoplayer.html

**Speicherplatz** Hier liegt der Vorteil gegenüber den anderen zwei *Billboard*-Systemen. Da immer auf den verschiedenen Perspektiven nicht alle Elemente des 3D-Modells abgebildet werden, sind demzufolge auch weniger Bildinformationen zu speichern. So steigt der Speicherplatz-Bedarf nur geringfügig an, wenn die Anzahl der Seiten erhöht wird. Wird bei dem Erlen-Beispiel die Anzahl von 9 auf 72 Seiten erhöht, steigt der Speicherbedarf gerade mal um ca. 173%, obwohl die Anzahl der Seiten verachtfacht wurde.
Speicherbedarf:

- 9 Seiten: 3,25 Megabyte
- 18 Seiten: 4,41 Megabyte
- 36 Seiten: 5,95 Megabyte
- 72 Seiten: 7,67 Megabyte

**Anwendbarkeit** Dies ist ein Schwachpunkt der Methode. Denn es müssen abhängig von der Kamera-Position Elemente ein- und ausgeblendet werden. Das Problem der Kamera-Rotation ist das gleiche wie bei dem dynamischen *Billboard*-System.
Auf fotobasierte Verfahren ist dieses Prinzip praktisch gar nicht anwendbar, es sei denn, es gibt Kameras, die neben den Bildpunkten auch die Tiefeninformationen zu jedem Bildpunkt speichern.

**Ausführbarkeit** Keines der getesteten Programme zeigte Probleme bei der Darstellung.

#### Raster-Billboard-System

**Dreidimensionaler Eindruck** Bei einem Raster der Größe 9x9 entsteht bei der Betrachtung ein räumlicher Eindruck. Dieser wird bei der Bewegung um das Objekt verstärkt. Ist jedoch die Anzahl der Ebenen reduziert, entsteht ein kastenförmiger Eindruck des Baum-Modells. Dieser nimmt jedoch mit Zunahme der Entfernung ab.

**Fehler bei der Darstellung** Im Gegensatz zum Fächer-*Billboard*-System ist die Mitte nicht leer. Dies liegt daran, dass die Sicht durch ein Rechteck versperrt ist, dessen Vorderseite nun sichtbar ist. Da die Rechtecke in einem Winkel um 90° zueinander verdreht sind, ist somit aus jedem Blickwinkel immer eine Vorderseite eines Rechtecks zu sehen. Dies ist jedoch gleichzeitig auch ein Nachteil, da eine Sicht auf eine Perspektive durch Rechtecke mit anderen Perspektiven gestört werden. Im Gegensatz zur Lücke in der Mitte der Fächer-Methode fällt dies jedoch nicht so störend auf.
Der Fehler verschwindet allerdings auch mit zunehmender Entfernung.

**Speicherplatz** Ein bisschen mehr als die Fächer-Methode benötigt dieses System an Speicherplatz. Daher ist dies auch ein Vorteil gegenüber der dynamischen Methode.
Speicherbedarf:

- 3x3 Seiten: 5,06 Megabyte

- 5x5 Seiten: 6,36 Megabyte
- 9x9 Seiten: 8,21 Megabyte

**Anwendbarkeit** Wie bei dem Fächer-*Billboard*-System.

**Ausführbarkeit** Wie bei dem Fächer-*Billboard*-System.

### 6.1.3 Fazit

Die Methode des dynamischen *Billboard*-Systems erzeugte den qualitativ besten Eindruck, benötigt jedoch den meisten Speicher. Für eine internetfähige Visualisierung ist der hohe Speicherbedarf und die damit zu erwartende langen Wartezeiten nicht geeignet. Außerdem lief dieses Prinzip nicht auf allen Plattformen, was ein weiterer Nachteil ist.
Die Fächer-Methode dämmt das Problem des Speicherplatzbedarfs ein, und wäre eher für das Internet geeignet. Sie hat jedoch ihre Schwäche in der Darstellung. Das gleiche 3D-Modell wirkte mit diesem System nicht so dicht belaubt wie auf einem der anderen beiden *Billboard*-Systeme. Hier sollte noch eine Verfeinerung des Prinzips vorgenommen werden, welches das Problem des Spalts in der Mitte löst.
Als eine Kompromiss-Lösung zwischen dem dynamischen und Fächer-*Billboard*-System kann die Raster-Methode angesehen werden. Sie kommt mit weniger Speicher aus als die dynamische Methode und hat tolerierbare Darstellungsfehler. Daher wäre diese Methode für die Internet-Darstellung als am geeignetsten zu betrachten.

## 6.2 Vergleich der drei Methoden

Die drei vorgestellten Methoden der Erzeugung von Baumdarstellungen werden nun miteinander verglichen. Dazu werden zunächst Kriterien eingeführt und auf die Methoden angewendet. Die Bewertung der drei Methoden geschieht nach dem Gesichtspunkt, dass eine Vielzahl von Bäumen zu erstellen ist.

Im Zuge der Kriterien-Definition fällt der Begriff Nutzer. Als Nutzer wird diejenige Person bezeichnet, die im Begriff ist, eine Baumdarstellung mittels einer der genannten Methoden zu erzeugen.

### 6.2.1 Kriterien

**Vorkenntnisse** Die drei Methoden setzen unterschiedliche Fertigkeiten des Nutzers voraus. Diese Fertigkeiten betreffen hauptsächlich den Umgang mit entsprechenden Computeranwendungen. Welche der drei Methoden bei der Erstellung von Baumdarstellungen zum Einsatz kommt, hängt nicht zuletzt davon ab, wie lange sich der Nutzer in ein Computer-Programm oder in Arbeitsschritte einarbeiten muss. Ob ein Nutzer die notwendigen Voraussetzungen schnell erlernen kann, oder ob die Einarbeitung viel Zeit in Anspruch nimmt, soll mit diesem Kriterium aufgezeigt werden.
Da die Baumdarstellungen für die Sprache VRML erzeugt werden sollen, gilt für alle drei Methoden, dass es Vorkenntnisse in dem Bereich dieser Sprache geben sollte. Daher werden diese bei der einzelnen Bewertung als vorausgesetzt betrachtet.

**Datenmaterial** Damit ein Nutzer Baumdarstellungen erzeugen kann, ist er auf Datenmaterial angewiesen. Je nach Methode unterscheiden sich die Anforderungen beziehungsweise das Datenmaterial. Welche Art von Daten gebraucht werden und welche Qualitäten diese besitzen müssen, soll mit diesem Kriterium geklärt werden.

**Arbeitsaufwand** Sind die Vorkenntnisse des Nutzers ausreichend und hat er ein akzeptables Ergebnis bei der Erstellung der Baumdarstellung erreicht, ist es wichtig, eine grobe Abschätzung zu geben, wie viel Zeit dafür benötigt wurde.

**Wiederverwendbarkeit** Für das gesamte Projekt des Branitzer Parks muss eine Vielzahl von Baumdarstellungen erzeugt werden. Zeit-ökonomisches Arbeiten ist dabei ein wichtiger Punkt. Daher muss untersucht werden, ob es möglich ist, die Effizienz bei der Erstellung zu erhöhen.
Damit die erzeugte Baumdarstellung nach Möglichkeit nicht mehrfach im Modell des Parks auftritt, müssen unterschiedliche Baumdarstellungen erzeugt werden. Am besten wäre das Erstellen nur einer Baumdarstellung. Alle anderen Baumdarstellungen würden durch leichte Anpassungen ableitbar sein. Da die Variationen der Baum-Vegetation im Park zu groß sind, wird dieser Idealfall nicht eintreten. Dennoch gibt es Baumarten, die eventuell Gemeinsamkeiten besitzen und durch Modifizieren von anderen, bestehenden

Baumdarstellungen abzuleiten sind. Dies ist je nach Methode mal mehr, mal weniger möglich und soll hiermit besprochen werden.

**Anforderungen an Hardware** Die Anforderungen an die Hardware sind je nach Methode recht unterschiedlich. So ist die Größe des Arbeitsspeichers und die Geschwindigkeit des Prozessors ein Kriterium, das über die Umsetzbarkeit einer Methode Auskunft gibt.

**Anwendbarkeit** Die Art der Ergebnisse der Methoden unterscheidet sich zunächst. Doch für alle gilt: sie müssen in VRML präsentierbar sein. Daher ist es wichtig zu untersuchen, welcher Aufwand betrieben werden muss, damit eine Darstellung in VRML möglich ist.

**Qualität** Unterschiedliche Methoden erzeugen unterschiedliche Ergebnisse. Dabei schwankt auch die Qualität des Ergebnisses. Ein Baum sollte auch nach einem Baum aussehen. Wie realistisch ein Ergebnis ist, soll mit diesem Kriterium geklärt werden.

**Repräsentativität** Die Baumdarstellungen, die erzeugt werden, dienen dazu, real existierende Vorbilder virtuell zu repräsentieren. Da sich die Prinzipien der Methoden unterscheiden, werden die Ergebnisse mal mehr, mal weniger ihren Baumvorlagen entsprechen. Die Qualität der Darstellung spielt dabei keine Rolle.

### 6.2.2 Bewertung der drei Methoden

#### Fotobasierte Bildvorlagen als Billboards

**Vorkenntnisse** Diese Methode nutzt in erster Linie Fotovorlagen von Bäumen, die vom Hintergrund befreit werden. Dies geschieht in einem Bildbearbeitungsprogramm wie GIMP oder PHOTOSHOP. Die Handhabung dieser Anwendungen ist schnell zu erlernen, und es gibt viel Literatur, die den Einstieg erleichtert. Für diese Methode ist es nicht notwendig, den kompletten Funktionsumfang des Bildbearbeitungsprogramms zu beherrschen.
Dies ist ein Vorteil dieser Methode: im Vergleich zu den anderen beiden Methoden sind Vorkenntnisse nicht so wichtig. Die nötigen Fertigkeiten können leicht bei der Erstellung einer Baumdarstellung erlernt werden.

**Datenmaterial** Das Datenmaterial reduziert sich auf Foto-Aufnahmen der betroffenen Bäume. Die Qualität der Darstellung hängt stark von den Aufnahmen ab. Daher wurde eine Anforderungsliste definiert, die die Qualität der Aufnahmen sicherstellen soll.
Dies ist für diese Methode der wichtigste Punkt, da die Qualität der Fotografien das Ergebnis beeinflusst. Von allen drei Methoden ist diese Methode am meisten auf die Qualität des Datenmaterials angewiesen.

**Arbeitsaufwand** Wie viel Zeit die Freistellung benötigt, hängt stark von der Qualität der Fotos ab. Sind Bereiche des Baumes verdeckt, müssen diese erst ergänzt werden. Hinzu kommt, dass das Freistellen durch eine ungünstige Belichtungseinstellung der Kamera erschwert werden kann. Dies verzögert den Freistellungsprozess.
Das Beispiel der Erle wurde innerhalb von 5 Stunden erstellt. Dies ist der geringste zu betreibende Aufwand bei der Erzeugung einer Baumdarstellung im Vergleich zu den anderen Methoden.

**Wiederverwendbarkeit** Dies ist ein großer Nachteil dieser Methode. Die erzeugte Darstellung ist nicht ohne großen Aufwand für andere Bäume modifizierbar. Das bedeutet, dass lediglich Bäume gleicher Art dargestellt werden können, indem die Baumkrone der Baumdarstellung leicht verändert wird. Jedoch dürfte es auch schwierig sein, einen großen ausgewachsenen Baum mit einem kleinen Baum der gleichen Art darzustellen. Daher ist die mit dieser Methode erzeugte Baumdarstellung kaum wieder zu verwenden. Geht es darum, Bäume in verschiedenen Jahreszeiten darzustellen, kann mit der Erzeugung einer Baumdarstellung des Sommers durch Einfärbung der Blätter eine Herbst-Baum-Darstellung erzeugt werden. Geht es jedoch darum, den Frühling und Winter darzustellen, wo viele Laubbäume laublos sind, muss der komplette Arbeitsprozess durchschritten werden.

**Anforderungen an Hardware** Da es nur 2D-Inhalte sind, die es bei der Freistellung zu bearbeiten gilt, und kaum aufwändige Berechnungen durchgeführt werden müssen, fordert die Methode am wenigsten Arbeitsspeicher und Prozessorleistung.
Da es notwendig ist, in einem Alpha-Kanal mit einem Pinselwerkzeug zu arbeiten, bietet sich die Nutzung eines Grafiktabletts an, da damit wesentlich genauer gearbeitet werden kann als mit der Maus. Dies ist zwar keine Voraussetzung für die Anwendung dieser Methode, es kann jedoch Zeit eingespart werden, was in Anbetracht der Anzahl der freizustellenden Bäume nicht unerheblich sein dürfte.

**Anwendbarkeit** Nach der Freistellung eines Baumes liegen in der Regel 4 Dateien vor. Diese werden in VRML auf ein Rechteck-Kreuz gelegt. Die Anordnung der Rechtecke muss dabei von Hand vorgenommen werden und in einer VRML-Datei gespeichert werden. Da die Anordnung der Rechtecke für die Darstellung im Prinzip für alle fotobasierten Baumdarstellungen gleich ist, muss nur der Dateiname für die Textur angepasst werden. Daher ist mit dieser Methode unter dem Strich kein gesonderter Aufwand zu betreiben, um die Darstellung in VRML zu überführen.

**Qualität** Bei der Betrachtung der Darstellung in VRML entsteht bei der Betrachtung von der Seite ein flächiger Eindruck. Dies lässt sich mit den in dieser Arbeit genannten *Billboard*-Systemen nicht verbessern. Einzig die direkte Draufsicht auf jeweils eine Seite des Baumes liefert ein gutes Ergebnis.
Einige der freigestellten Bäume hatten eine offene Baumkrone. Offen heißt, dass die innere Struktur und der Hintergrund stellenweise sichtbar ist. Diese Bäume freizustellen

war am schwersten. Der Grund lag häufig darin, dass der Vordergrund – also der freizustellende Baum – und der Hintergrund nicht klar auseinanderhaltbar waren. Dies galt oft im unteren Bereich der Baumkrone, da im Hintergrund andere Vegetation sichtbar ist. Im oberen Bereich der Baumkrone war der Hintergrund der Himmel. Somit konnte klar Baum-Struktur und Himmel auseinandergehalten werden. Durch die unterschiedlichen Hintergründe ist auf den Ergebnissen der Freistellung zu sehen, dass die unteren Bereiche wesentlich dichter als die oberen Bereiche sind.
Bei Aufnahmen, die mit einer ungünstigen Belichtungseinstellung erstellt wurden, war oft der Himmel zu hell und der Rest des Bildes zu dunkel. Dadurch traten Überbelendungen an Bildbereichen auf, die eine feine Struktur der Baumkrone direkt vor dem Himmel abbildeten. So hatten feine Äste einen weichen Übergang zum Himmel. Dies macht das Freistellen fast unmöglich, da die Farbwerte des Himmels in den Ast hineinreichten. Diese Bereiche mussten manuell überzeichnet werden oder wurden ganz entfernt.
Die Qualität der Darstellung ist somit direkt abhängig von der Qualität des Ausgangsmaterials. War das Ausgangsmaterial gut, war das Ergebnis entsprechend.

**Repräsentativität** Von allen drei Methoden lieferte diese Methode die beste Repräsentativität. Denn diese Methode nutzt das Bildmaterial als Basis zu Darstellung. Im Gegensatz dazu sind die anderen Methoden bestenfalls eine Annäherung an die Vorlage. Die erzeugte Darstellung entspricht dem Vorbild zum Zeitpunkt der Aufnahme.

#### Modellierung mit konventionellem 3D-Programm

**Vorkenntnisse** Um mit einem konventionellen 3D-Programm zu arbeiten, sind Vorkenntnisse im Umgang mit diesem Programm sehr wichtig. Diese sind in zweierlei Hinsicht bedeutend: Das Ziel ist es, ein 3D-Modell eines Baums zu erzeugen. Dazu müssen zunächst die notwendigen Techniken der Modellierung bekannt sein. Um das Ziel zu erreichen gibt es mehrere Wege. Nicht jeder führt zum optimalen Ergebnis. Das Modell soll der Realität entsprechen, aber nicht zu komplex in seiner Struktur sein, also aus so wenig Dreiecken wie möglich bestehen.
Deshalb sind nicht nur Fähigkeiten in der Modellierung gefragt, sondern auch in der Optimierung von 3D-Modellen. Diese Vorkenntnisse sind aus einer umfangreichen Literatur zu entnehmen. Auch in einschlägigen Internetseiten wird der Einstieg erleichtert.
Im Vergleich zur Methode der fotobasierten Bildvorlagen als *Billboard*s ist aber mehr Aufwand in die Einarbeitung zu betreiben.

**Datenmaterial** Im Gegensatz zu den *Billboard*s mit Fotovorlage sind Aufnahmen sehr hilfreich bei der Umsetzung, aber nicht zwingend erforderlich, da es bereits reicht, den Baum vor Ort zu begutachten. Wenn Bildvorlagen verwendet werden, dann sollten Aufnahmen zum Zeitpunkt verwendet werden, als der Baum im unbelaubten Zustand war. Somit ist die Ast- und Zweigstruktur besser zu erkennen und nachzubilden.
Damit die Textur für den Baum erzeugt werden kann, sind Detail-Aufnahmen von einzelnen Blättern und der Rinde zu erstellen.

Somit sind die Anforderungen an das Datenmaterial nicht sehr hoch, da die Qualität der Aufnahmen das Ergebnis nicht direkt beeinflusst.

**Arbeitsaufwand** Der zu betreibende Aufwand hängt stark von der Komplexität der Baumvorlage ab. Dabei spielt weniger die Größe des Baumes eine Rolle, als die Anzahl der Äste und Verzweigungen, die zu erstellen sind.
Auch hängt die Arbeits-Geschwindigkeit von den Fähigkeiten des Nutzers ab. Je mehr Erfahrungen dieser im Umgang mit der Anwendung hat, desto schneller sind brauchbare Ergebnisse zu erzielen. Auch wenn eine genaue Abschätzung unmöglich ist, so steht zumindest fest, dass der zu betreibende Aufwand höher ist als bei den fotobasierten *Billboard*s.

**Wiederverwendbarkeit** Ein erstelltes 3D-Modell ist nur bedingt auf andere Bäume anwendbar. Der Aufwand, eine Struktur des Modells an eine andere Vorlage anzupassen, steigt schnell an, so dass die Zeitersparnis gegenüber einer komplett neuen Modellierung gegen Null geht.
Jedoch könnte eine Anpassung von Modellen angebracht sein, wenn sich zwei Vorlagen kaum voneinander unterscheiden. Auch wenn sich eine Anpassung kaum rentieren dürfte, so ist sie doch, im Gegensatz zur Erstellung der fotobasierten *Billboard*s, möglich. Zum Beispiel bei der Modellierung verschiedener Jahreszeiten sind ohne größeren Aufwand Blätter ausblendbar oder deren Texturen leicht anzupassen.

**Anforderungen an Hardware** Bei einer detaillierten Modellierung eines komplexen Baumes kann der Arbeitsspeicher schnell knapp werden, da mit der Komplexität auch die Anzahl der Dreiecke beziehungsweise der Polygone zunimmt.
Zwar ist viel Prozessorleistung bei der Berechnung eines Bildes des 3D-Modells notwendig, jedoch nicht während der Modellierung.

**Anwendbarkeit** Das 3D-Modell kann in seiner Komplexität nicht ohne weiteren Aufwand in das VRML-Modell überführt werden. Daher muss die Darstellung zunächst stark vereinfacht werden. Diese Vereinfachung geschieht mittels der *Billboard*-Technik. Dazu muss das Objekt aus verschiedenen Ansichten gerendert werden, damit die so entstandenen Bildinformationen auf den *Billboard*s präsentiert werden können.
Die in den eingeführten *Billboard*-Systemen verwendeten Prinzipien benötigen ein genaues Platzieren der Kamera, sowie das Ein- und Ausblenden von Elementen des Modells. Dies ist grundsätzlich in einem konventionellen 3D-Programm möglich, muss jedoch zuvor durch den Nutzer beschrieben oder programmiert werden. Speziell das Umrunden des Modells mit der Kamera kann durch eine Kreisbahn realisiert werden. Der Kamera wird die Kreisbahn als Bewegungspfad vorgegeben.

**Qualität** Im Gegensatz zu den fotobasierten *Billboard*s ist die Qualität nicht vom Datenmaterial abhängig, sondern einzig und allein von den Fähigkeiten des Nutzers im

Umgang mit der 3D-Anwendung. Je besser die Vorkenntnisse sind, desto besser ist das Ergebnis.
Bei der Erstellung der Bilder für die *Billboard*s ist auch der genutzte Renderer ein ausschlaggebender Punkt für die Qualität.

**Repräsentativität** In der Rangordnug der Repräsentativität steht diese Methode an zweiter Stelle. Mit dieser Methode wird versucht, das Aussehen der Baumvorlage zu modellieren. Dies wird im Ergebnis jedoch immer nur eine Annäherung an die Vorlage sein.

### Modellierung mittels Wachstumsgrammatiken

**Vorkenntnisse** Von allen drei Methoden benötigt man bei dieser Methode am meisten Vorkenntnisse. Für diese Methode sind Kenntnisse in der Programmierung, sowie Grundkenntnisse von Wachstumsgrammatiken, von Vorteil, aber nicht zwingend notwendig. Zwar erschwert es die Einarbeitung in GroIMP, wenn diese Kenntnisse noch nicht vorhanden sind, doch ist es grundsätzlich möglich, wie ein Seminar mit Architektur-Studenten der BTU Cottbus zeigte. Ohne Vorkenntnisse erlangten sie dennoch bemerkenswerte Ergebnisse.
Da diese Arbeits-Methode nicht ohne eine Einarbeitungszeit auskommt, ist dies eher ein negativer Aspekt dieser Methode. Dennoch ist es möglich, die Einarbeitungszeit mit der Zeit der Erzeugung des Baummodells zu verschmelzen: Während der Erstellung des Baummodells können einzelne Aspekte und Konstrukte von GroIMP beziehungsweise XL erlernt und direkt umgesetzt werden.

**Datenmaterial** Das Datenmaterial hat, wie bei den fotobasierten *Billboard*s, eine besondere Stellung. Jedoch nicht die Qualität der Aufnahmen des Baumes, sondern die Qualität der Informationen über die Eigenschaften der Baumart sind entscheidend für das Endergebnis. Denn bei der Modellierung mit Wachstumsgrammatiken ist es wichtig zu wissen, unter welchen Gesetzmäßigkeiten der betroffene Baum gewachsen ist. Mit Gesetzmäßigkeiten ist zum Beispiel gemeint: „In welchem Winkel stehen die Äste zueinander?“. Oder: „Wie wächst der Stamm?“. Das bedeutet, dass ein Auseinandersetzen mit botanischen Prozessen sehr wichtig ist, um überhaupt ein Baumart-ähnliches Ergebnis zu erzielen.
Wie tief die Vorbereitung sein muss, hängt vom gewünschten Detailgrad des Objektes ab. Soll die Ausprägung des Baumes zu einem bestimmten Zeitpunkt modelliert werden, ist weitaus weniger Wissen über die Baumart notwendig, als das bei einer Simulation des kompletten Wachstumsprozesses der Fall wäre.

**Arbeitsaufwand** Dies hängt von der Struktur der Baumvorlage ab. Die in dieser Arbeit besprochene Erle hat eine verhältnismäßig einfach nachzuvollziehende Struktur. Der Hauptstamm erstreckt sich bis hoch in die Spitze. Bäume mit komplexeren Verzweigungsstrukturen sind tendenziell schwieriger in Grammatiken zu überführen.

Ein weiteres, zeitaufwändiges Problem ist, dass bei der Modellierung mit Wachstumsgrammatiken häufig ein Parameter-Wert ausprobiert und solange angepasst wird, bis ein gewünschtes Ergebnis erscheint. Da bei Grammatiken oft eine kleine Veränderung große Auswirkungen hervorbringt, kann allein die Suche nach einem geeigneten Parameter-Wert sehr lange dauern.
Somit ist diese Methode mit Abstand die aufwändigste von allen drei genannten, da hier im Bereich von Tagen bis Wochen kalkuliert werden muss, bis ein ansehnliches Ergebnis erzielt wurde.

**Wiederverwendbarkeit** Am besten können in dieser Methode andere Modelle aus einer Modellierung erzeugt werden. Sobald bei der Programmierung der Grammatiken Zufallswerte auftreten, wird bei jeder erneuten Anwendung dieser Grammatiken eine Variation des 3D-Modells auftreten. Die Modelle würden sich ähneln, aber nicht gleichen.
Ein weiterer Pluspunkt ist, dass durch das Anpassen der Grammatiken eine Anpassung der Modelle leichter ist, als in den anderen zwei Methoden. Auch die Modellierung von Jahreszeiten ist ähnlich einfach wie bei der Modellierung mit konventionellen Anwendungen.

**Anforderungen an Hardware** Das 3D-Modell wird nach und nach durch Anwendung der Grammatiken erzeugt. Im Beispiel der Erle wurden in GroIMP circa 1000 mal die gleichen Grammatiken angewendet. Dies benötigt neben der Prozessorleistung auch sehr viel Arbeitsspeicher, da auf dem programminternen Graphen komplizierte Such- und Ersetzungsoperationen durchgeführt werden müssen. Als die Erle fertiggestellt wurde, bestand sie aus circa 400.000 Objekten und benötigte ungefähr 600MB Arbeitsspeicher. Somit liegen die Anforderungen an die Hardware ungefähr auf dem Niveau der Modellierung mit konventionellen 3D-Anwendungen.

**Anwendbarkeit** Auch in dieser Methode steht am Ende ein 3D-Modell des Baumes und muss ebenfalls vereinfacht werden, so dass es in eine VRML-Szene überführbar ist. Speziell für GroIMP wurde eine Automatisierung erstellt und integriert, die diese Vereinfachung übernimmt und entsprechende *Billboard*-Systeme zur Darstellung erzeugt. Das Automatisieren der Vereinfachung ist ein Pluspunkt für die Anwendung von GroIMP, da sich der Nutzer über die Art der Darstellung in GroIMP keine Gedanken machen muss.

**Qualität** Die Qualität des 3D-Modells hängt ebenfalls vom Nutzer ab. Neben den Fähigkeiten im Umgang mit Wachstumsgrammatiken ist es wichtig, wie sehr er sich mit den Gesetzmäßigkeiten des betroffenen Baumes auseinandergesetzt hat. Anders als bei den anderen zwei Methoden muss am Ende die Frage gestellt werden, in wie weit die Baumart wieder zu erkennen ist; da durch Wachstumsgrammatiken erzeugte 3D-Modelle mit zunehmender Größe immer schlechter zu kontrollieren sind, kann das Ergebnis degenerieren und daher befremdlich wirken.
Dennoch gilt festzustellen, dass die qualitativ beste Baumdarstellung der Erle mit GroIMP erzeugt wurde. Die Erle aus 3D Studio Max kann die Qualität der in GroIMP

erstellten Erle nicht erreichen, da der Detailgrad des in GroIMP erzeugten Modells höher ist.

**Repräsentativität** Wie schon erwähnt, ist es schwer, das Erzeugen des 3D-Modells durch Anwendung von Grammatiken so zu steuern, dass ein gewünschtes Ergebnis erreicht wird. Wenn es schon schwer ist, die Baumart wiederzuerkennen, wird es wohl noch schwerer sein, genau die Baumvorlage im erzeugten Modell wieder zu finden.
Um ein hohes Maß an Repräsentativität zu erreichen und den Aufwand bei der Erstellung einzudämmen, könnten Elemente des Baumes in GroIMP modelliert und anschließend exportiert werden. Die exportieren Elemente könnten dann mit einem konventionellen 3D-Modellierungsprogramm weiterverarbeitet werden.

### 6.2.3 Fazit

Zieht man den Maßstab der Kriterien heran, wären die meisten Vorteile auf Seiten der fotobasierten *Billboard*-Darstellung. Auch das Verhältnis zwischen Aufwand und Ergebnis beziehungsweise Repräsentativität spricht deutlich für diese Methode. Doch hatte diese Schwächen in der Qualität der Darstellung, wo die anderen beiden Methoden punkten konnten. Vor allem der Einsatz der *Billboard*-Systeme und die dadurch verbesserte Qualität der Darstellung spricht daher gegen die Verwendung der fotobasierten *Billboard*s. Denn zur Erzeugung eines der *Billboard*-Systeme wird ein 3D-Modell eines Baumes benötigt. Das 3D-Modell kann mittels konventioneller 3D-Anwendungen oder unter Zuhilfenahme von Wachstumsgrammatiken, wie es mit GroIMP geschehen ist, erzeugt werden. Beide Methoden unterscheiden sich im Vorkenntnis-Stand und dem Aufwand. Die beiden Kriterien sprechen eher dafür, eine konventionelle 3D-Modellierung gegenüber Grammatiken vorzuziehen, auch wenn die Erzeugung der *Billboard*-Systeme nicht automatisiert sein dürfte.
Wo die ersten beiden Methoden ihre Nachteile haben, hat die Nutzung von Wachstumsgrammatiken einen erheblichen Vorteil. Mit der Programmierung eines Modells ist eine Variation des Ergebnisses schnell zu erstellen, was bei den ersten beiden Methoden kaum bis gar nicht möglich ist. Somit sind Baumgruppen, aus einer Baumart bestehend, leicht zu erzeugen, wobei nur ein Modell entwickelt werden muss. Dies spricht also für die Nutzung von Grammatiken; da es um das Erstellen von vielen Bäumen geht, kann der Aufwand in Summe reduziert werden. Zumal die Fähigkeiten bei der Nutzung von GroIMP zum Beispiel, von Baum zu Baum zunehmen dürften.

Welche Methode für die Visualisierung des Branitzer Parks zum Einsatz kommen sollte, hängt von der Zielstellung ab. Geht es darum, den Park so wirklichkeitsgetreu wie möglich darzustellen, ist die Methode der fotobasierten *Billboard*s am besten geeignet, da die Darstellungen der Bäume durch Fotos realisiert werden. Somit können virtuelle Sichtachsen den real existierenden Sichtachsen am besten nachempfunden werden. Das bedeutet jedoch, dass jeder Baum freigestellt werden muss. Bei der Vielzahl der Bäume ist das ein sehr großer Aufwand, damit die Zielstellung erreicht werden kann.

Zur Darstellung von Solitärbäumen ist die fotobasierte *Billboard*-Methode mit ihren Rechteck-Kreuzen nicht geeignet. Die Darstellungs-Qualität von den drei vorgestellten *Billboard*-Systemen ist besser. Somit muss also ein 3D-Modell nach einem Vorbild erstellt werden. Dies spricht für die Methode der konventionellen 3D-Modellierung, da die Repräsentativität besser ist, als bei der Nutzung von Grammatiken. Ein weiterer Grund für deren Einsatz ist, dass die betroffene Baumvorlage höchstwahrscheinlich nicht noch einmal im Park erscheint. Somit spielt das Kriterium der Wiederverwendbarkeit keine Rolle bei der Erstellung von Solitärbaum-Darstellungen.
Dennoch ist auch die Modellierung mit Grammatiken durchaus eine Variante für die Erzeugung einer Solitärbaum-Drastellung. Denn der erreichte Detailgrad des Baummodells, ist bei der konventionellen 3D-Modellierung, nur mit sehr großem Aufwand zu erreichen.

Geht es weniger um eine exakte Nachempfindung eines speziellen Baumes, sondern mehr darum, eine 3D-Szene mit Bäumen „aufzufüllen“, ist die Nutzung von Grammatiken am besten geeignet. Durch Anpassung von Parametern wie Größe oder Alter ist es im Vergleich zu den anderen Methoden einfacher, andere Modelle zu erzeugen. Dies ist von Vorteil, wenn es darum geht, Baumgruppen darzustellen. Bei deren Darstellung ist die Repräsentativität nicht so wichtig. Entscheidend ist da eher der zu betreibende Aufwand, um mehrere Bäume darzustellen.

Alle drei Methoden sind somit adäquate Ansätze, um die verschiedenen Ziele zu erreichen. Die fotobasierte *Billboard*-Methode ist für den Fall einer möglichst exakten Darstellung der einzelnen Bäume geeignet. Für die detaillierte Darstellung von Solitärbäumen bieten sich die Modellierung mit einem konventionellen 3D-Programm sowie die Modellierung mit Wachstumsgrammatiken an. Bei der Erstellung ganzer Baumgruppen ist die Verwendung von Grammatiken die beste Methode.

## 6.3 Ausblick

Mit dieser Arbeit wurde das Problem der Erzeugung von virtuellen Bäumen sowie deren Repräsentation besprochen und es wurden mögliche Lösungen gegeben. Als Lösungansatz der Repräsentation in VRML wurden drei *Billboard*-Systeme erklärt. Die qualitativen Ergebnisse bei der Darstellung waren unterschiedlich. Die qualitativ beste Wiedergabe eines 3D-Modells bietet das dynamische *Billboard*-System, benötigte jedoch den meisten Speicher-Platz. Das Fächer-*Billboard*-System brauchte den wenigsten Speicher, lieferte aber das schlechteste Ergebnis. Da Baum-Elemente in der Mitte der Darstellung verschwanden, entstand eine Lücke, was somit keiner adäquaten Repräsentation des 3D-Modells entsprach. Besser war die Qualität mit dem Raster-*Billboard*. Es repräsentierte das 3D-Modell recht gut und benötigte wenig Speicher. Somit ist dieses System eine gute Kompromisslösung.

Das Problem des Fächer- und Raster-*Billboard*s ist, dass sie ein 3D-Modell benötigen. Denn zum Ein- und Ausblenden von Elementen werden deren Abstände zur Kamera, also deren Tiefeninformationen benötigt. Diese Informationen sind naturgemäß nicht auf Fotoaufnahmen von Bäumen vorhanden. Somit ist eine Anwendung der beiden *Billboard*-Systeme auf Fotovorlagen nicht ohne weiteres möglich. Das Problem der volumetrischen Darstellung von Bäumen wurde von Linz et al. [14] besprochen. Es wurde eine Methode verwendet, die aus mehreren Bild-Vorlagen eines Baumes Tiefeninformationen erzeugte. Mit diesen Informationen war eine volumetrische Darstellung von Bäumen anhand von Fotovorlagen möglich. Jedoch muss die Kamera bei jeder Aufnahme ausgerichtet werden. Das wäre eine weitere zu beachtende Anforderung bei der Erstellung des Bildmaterials und erhöht somit den Aufwand.

Neben der Repräsentation wurde ebenfalls in dieser Arbeit besprochen, wie ein Baum für die Repräsentation erzeugt werden soll. Die Methode der fotobasierten *Billboard*s konnte in Form eines *Billboard*-Kreuzes oder durch ein einzelnes sich drehendes *Billboard* repräsentiert werden. Die drei *Billboard*-Systeme konnten aus dem genannten Grund der fehlenden Tiefeninformationen nicht angewendet werden. Die Billboard-Methode brachte jedoch die beste Repräsentativität. Um qualitativ gute Ergebnisse zu erzielen, sollte der dazu aufgestellte Anforderungskatalog an das zu erstellende Bildmaterial eingehalten werden.

Die Modellierung mit konventionellen 3D-Modellierungsprogrammen ist für die Erzeugung von Einzelbaumdarstellungen gut geeignet, hat aber bei der Erzeugung ganzer Baumgruppen den Nachteil, dass jeder Baum einzeln zu modellieren wäre. Für diesen Zweck eignet sich die Nutzung von Grammatiken. Da jedoch der Aufwand zur Einarbeitung und Erstellung eines Modells mittels Grammatiken groß ist, kann diese Methode abschreckend wirken. Um den Einstieg in das Nutzen von Grammatiken zu erleichtern, bietet sich das Einfügen von „Baumbaukästen“ an. Das Beispiel der Erle in GroIMP zeigt, wie ein Baum in Struktur-Elemente einteilbar ist. Somit war eine bessere Kontrolle und Steuerung des „Wachstums“ möglich. Dieser Ansatz sollte daher weiter ausgebaut

werden, so dass grundsätzliche Probleme wie zum Beispiel Durchmesserwachstum oder Stamm-Höhe nicht extra zu programmieren sind.

Ein anderer Ansatz wäre, eine Art Einheitsbaum in GroIMP mit auszuliefern. Lediglich durch Anpassung von Parametern könnte somit ein Baum erzeugt werden. Da Baumarten grundverschieden in ihrer Art des Wachstums sein können, wäre hier vielleicht ein Einheitsbaum für einen Bereich von Baumarten zu erstellen. Dies setzt jedoch ein tiefgreifendes Studium von vielen Baumarten voraus. Damit ein Baum einen möglichst großen Bereich abdeckt, wird auch der Aufwand in der Entwicklung beträchtlich sein.

# Literaturverzeichnis


[1] Aho, Alfred V. ; Sethi, Ravi ; Ullmann, Jeffrey D.: *Compilerbau.* Oldenbourg Verlag München Wien, 1999. – ISBN 3486252941

[2] Chawla, Ravi ; Banerjee, Amarnath: A virtual environment for simulating manufacturing operations in 3D. In: *WSC '01: Proceedings of the 33nd conference on Winter simulation.* Washington, DC, USA : IEEE Computer Society, 2001. – ISBN 0–7803–7309–X, S. 991–997

[3] Décoret, Xavier ; Durand, Frédo ; Sillion, François X. ; Dorsey, Julie: Billboard clouds for extreme model simplification. In: *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers.* New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–709–5, S. 689–696

[4] E.Strasburge ; F.Noll ; H.Schenk ; Schimper, A.F.W. ; Sitte, Peter (Hrsg.) ; Ziegler, Hubert (Hrsg.) ; Ehrendorfer, Friedrich (Hrsg.) ; Bresinsky, Andreas (Hrsg.): *Lehrbuch der Botanik für Hochschulen.* Gustav Fischer Verlag Stuttgart Jena New York, 1991

[5] Ettrich, Berthold: *150 Jahre Branitzer Park.* Bd. 1. Branitz. Heft 3. Stiftung Fürst-Pückler-Museum Park und Schloss Branitz, 1998

[6] Hase, Hans-Lothar: *Dynamische Virtuelle Welten mit VRML 2.0.* 1. Auflage 1997. dpunkt - Verlag für digitale Technologie GmbH, 1997. – ISBN 3920993632

[7] Henke, Michael: *Entwurf und Implementation eines Baukastens zur 3D-Pflanzenvisualisierung in GroIMP mittels Instanzierungsregeln*, Lehrstuhl Grafische Systeme, Informations- und Medientechnik, Brandenburgische Technische Universität Cottbus (BTU), Diplomarbeit, 2006

[8] Juan, Olivier ; Keriven, Renaud: Trimap Segmentation for Fast and User-Friendly Alpha Matting. In: Paragios, Nikos (Hrsg.) ; Faugeras, Olivier D. (Hrsg.) ; Chan, Tony (Hrsg.) ; Schnörr, Christoph (Hrsg.): *VLSM* Bd. 3752, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–29348–5, S. 186–197

[9] Kniemeyer, Ole: Rule-based modelling with the XL/GroIMP software. In: Schaub, Harald (Hrsg.) ; Detje, Frank (Hrsg.) ; Brüggemann, Ulrike (Hrsg.): *The Logic of Artificial Life. Proceedings of 6th GWAL*, AKA Akademische Verlagsges. Berlin, 2004, S. 56–65

[10] Kniemeyer, Ole ; Buck-Sorlin, Gerhard ; Kurth, Winfried: Representation of Genotype and Phenotype in a Coherent Framework Based on Extended L-Systems. In: Banzhaf, Wolfgang (Hrsg.) ; Christaller, Thomas (Hrsg.) ; Dittrich, Peter (Hrsg.) ; Kim, Jan T. (Hrsg.) ; Ziegler, Jens (Hrsg.): *ECAL 2003* Bd. 2801, Springer, 2003 (Lecture Notes in Artificial Intelligence). – ISBN 3–540–20057–6, S. 625–634

[11] Kniemeyer, Ole ; Hemmerling, Reinhard ; Buck-Sorlin, Gerhard ; Kurth, Winfried: The rule-based language XL and the modelling environment GroIMP, illustrated with simulated tree competition. In: Prusinkiewicz, Przemyslaw (Hrsg.) ; Hanan, Jim (Hrsg.) ; Lane, Brendan (Hrsg.): *Proceedings of the 5th International Workshop on Functional-Structural Plant Models, Napier, New Zealand, 4–9 November 2007.*, Print Solutions Hawke's Bay Limited, Oktober 2007, S. 23.1–23.5

[12] Kurth, Winfried: *Die Simulation der Baumarchitektur mit Wachstumsgrammatiken.* Wissenschaftlicher Verlag Berlin Olaf Gaudig & Peter Veit GbR, 1999. – 327 S. – ISBN 3–932089–37–5

[13] Kurth, Winfried ; Buck-Sorlin, Gerhard ; Kniemeyer, Ole: Relationale Wachstumsgrammatiken: Ein Formalismus zur Spezifikation multiskalierter Struktur-Funktions-Modelle von Pflanzen. In: *Modellierung pflanzlicher Systeme aus historischer und aktueller Sicht. Symposium zu Ehren von Prof. Dr. Dr. h.c. Eilhard Alfred Mitscherlich, Schriftenreihe des Landesamtes für Verbraucherschutz, Landwirtschaft und Flurneuordnung Brandenburg, Reihe Landwirtschaft, Band 7* 76 (2006), S. 36–45

[14] Linz, Christian ; Drettakis, George ; Magnor, Marcus ; Reche-Martinez, Alex: Effective Multi-resolution Rendering and Texture Compression for Captured Volumetric Trees. In: Galin, Eric (Hrsg.) ; Chiba, Norishige (Hrsg.): *Proceedings of the Eurographics Workshop on Natural Phenomena.* Vienna, Austria : Eurographics, September 2006, S. 83–90

[15] Mandelbrot, B. B.: *The fractal geometry of nature.* W. H. Freeman, San Francisco, 1982. – ISBN 0716711869

[16] Neubert, Boris ; Franken, Thomas ; Deussen, Oliver: Approximate image-based tree-modeling using particle flows. In: *ACM Trans. Graph.* 26 (2007), July, Nr. 3, 88.1-88.8. `http://dx.doi.org/10.1145/1276377.1276487`. – DOI 10.1145/1276377.1276487. – ISSN 0730–0301

[17] Piegl, Les ; Tiller, Wayne: *The NURBS book (2nd ed.).* New York, NY, USA : Springer-Verlag New York, Inc., 1997. – ISBN 3–540–61545–8

[18] Prusinkiewicz, Przemyslaw: Art and Science for Life: Designing and Growing Virtual Plants with L-systems. In: Fernandez, T. (Hrsg.) ; Davidson, C. G. (Hrsg.): *Acta Horticulturae*, Nursery Crops: Development, Evaluation, Production and Use, 2004, S. 15–28

[19] Prusinkiewicz, Przemyslaw ; Lindenmayer, Aristid: *The Algorithmic Beauty of Plants (The Virtual Laboratory).* Springer, 1991. – ISBN 0387972978

[20] Schlüter, Oliver: *VRML: Sprachmerkmale, Anwendungen, Perspektiven.* O´Reilly Essentials, 1998

[21] Schneider, Sören: *Konzeption eines Subsystems für die GroIMP-Plattform sowie eines zugrunde liegenden XML-Datenformats zum Austausch graphbasierter, multiskalierter Strukturen*, Brandenburgische Technische Universität Cottbus Lehrstuhl Praktische Informatik / Grafische Systeme, Diplomarbeit, 2006

[22] Schütt, Peter ; Schuck, Hans J. ; Stimm, Bernd: *Lexikon der Baum- und Straucharten.* Nikol Verlagsgesellschaft mbH & Co. KG Hamburg, 2002. – ISBN 3–933203–53–8

[23] Tutte, W. T.: *Graph Theory.* Cambridge University Press, 2001. – ISBN 0–521–79489–7